CHAPTER 7

Series

On 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 his 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[Ifrah].

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

7.1 Introduction

It will not come as a surprise 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.

This chapter begins by exposing a general framework on how to compute infinite series and continued fractions. Then, I show two examples of application of this framework by implementing two functions, which are very important to

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

compute probabilities: the incomplete gamma function and the incomplete beta function.

For illustrative purposes, the implementation in Smalltalk is using a different architecture from the one used by the Java implementation. It should be noted that each implementation could have been implemented in the other language. Figures 7.1, and 7.2 show the class diagram of the Smalltalk and Java implementations, respectively.

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

The Java implementation uses two abstract classes to implement an infinite series and a continued fraction. Each concrete implementation necessitates the creation of a concrete subclass.

In spite of the difference in architecture, the reader can verify on each class diagram that the number of classes needed for a concrete implementation is the same in each case.

An interesting exercise is to implement the architecture presented in Java in Smalltalk and vice versa.

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 given by equation 7.1:

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

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 the scope of this book. Let me 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

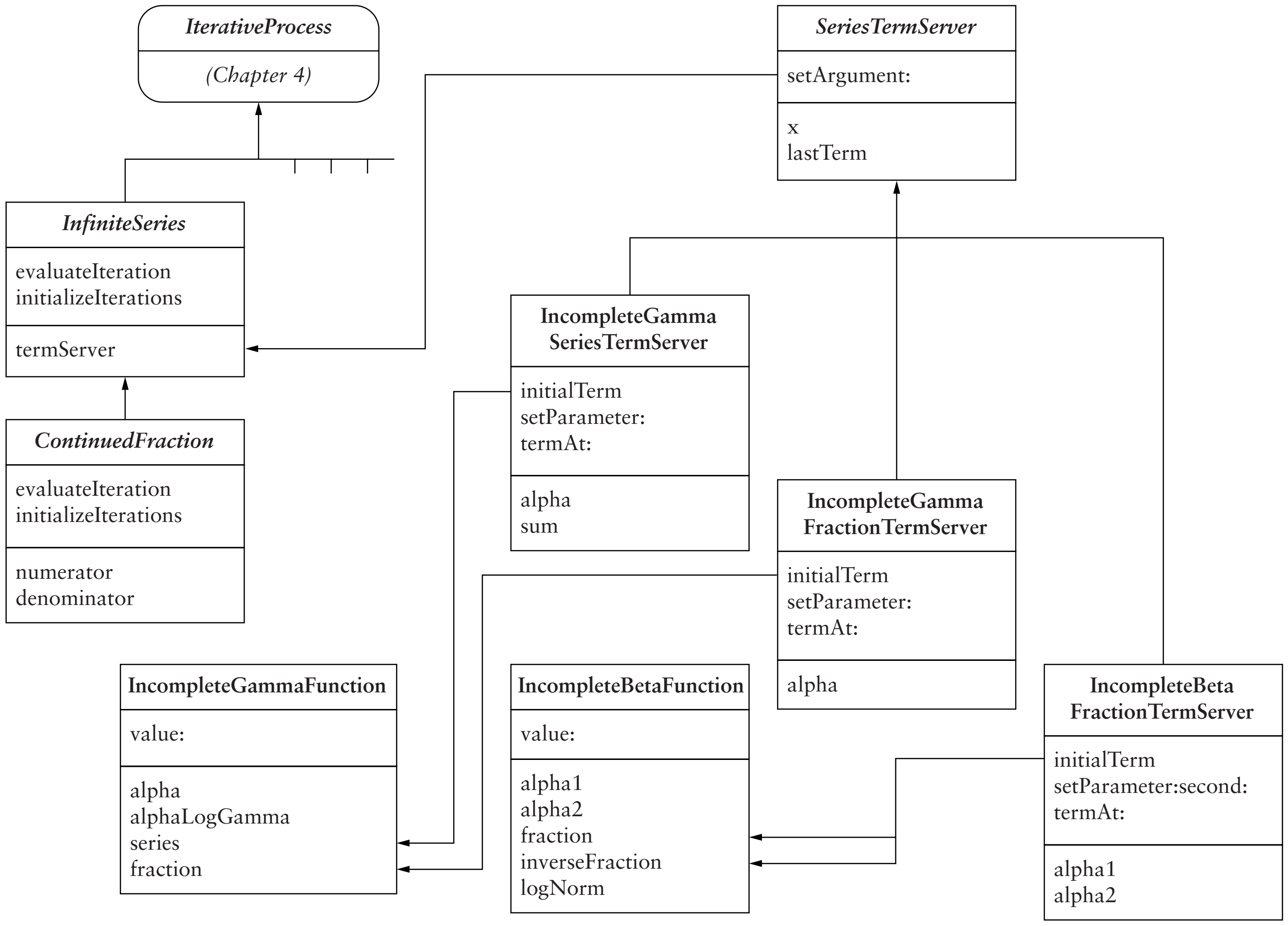

FIG. 7.1 Smalltalk class diagram for infinite series and continued fractions

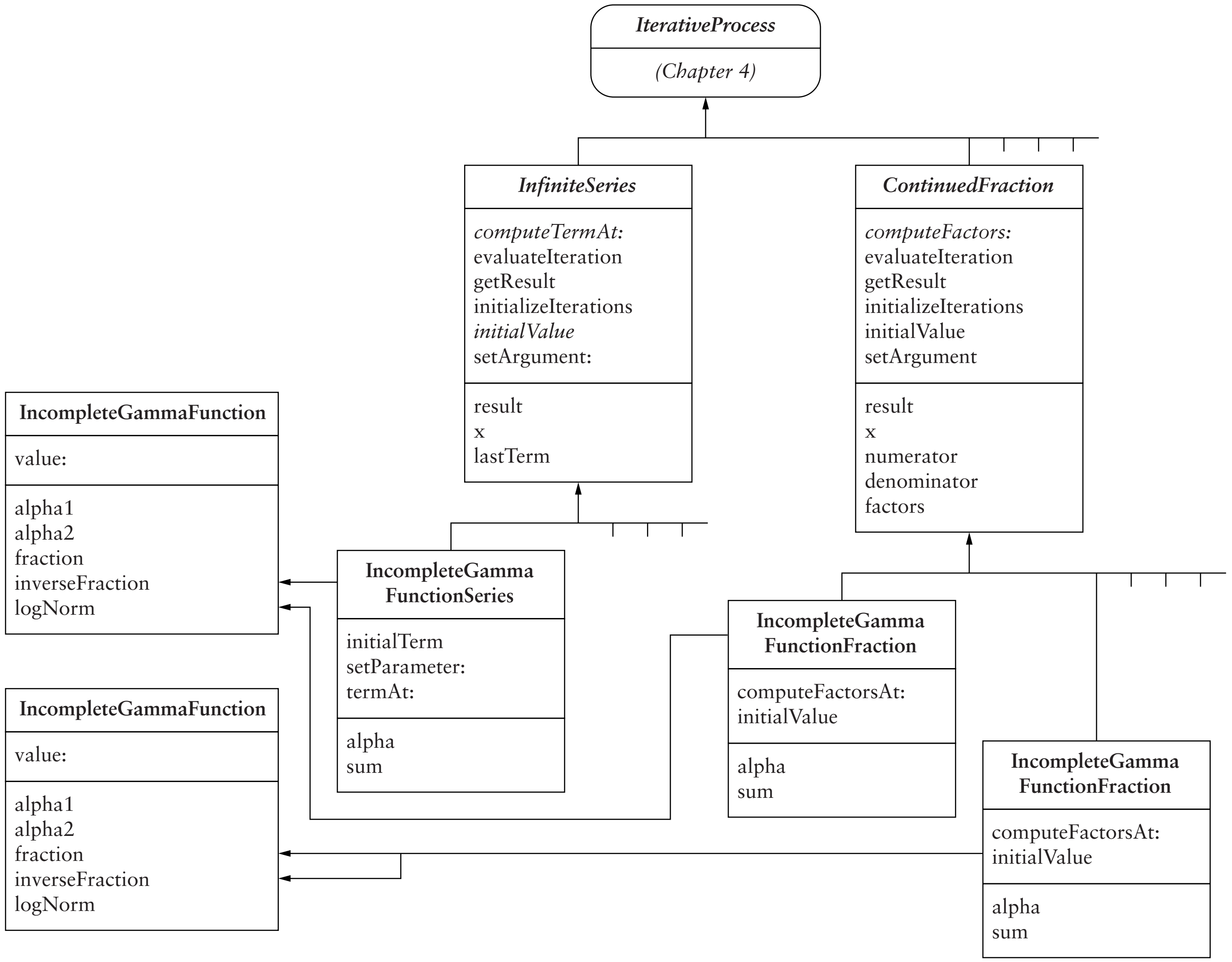

FIG. 7.2 Java class diagram for infinite series and continued fractions

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 avoiding rounding errors, the safest way is to do this computation 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, I 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 class discussed in chapter 4.

7.2.1 Infinite Series—Smalltalk Implementation

Listing 7.1 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 7.4.2.

The Smalltalk implementation uses a STRATEGY pattern. The class `DhbInfiniteSeries` is a subclass of the class `DhbIterativeProcess`, discussed in Section 4.1.1. 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 4.2.1.

To implement a specific series, an object of the class `DhbInfiniteSeries` is instantiated with a specific term server. A concrete example will be shown in Section 7.4.2.

Because of its generic nature, the class `DhbInfiniteSeries` does not implement the function behavior described in Section 2.1.1 (method `value:`). It is 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 7.4.2.

Listing 7.1 Smalltalk implementation of an infinite series

Class	`DhbInfiniteSeries`
Subclass of	`DhbIterativeProcess`
Instance variable names:	`termServer`

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

Class Methods

server: `aTermServer`

```
^self new initialize: aTermServer
```

Instance Methods

evaluateIteration

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

initialize: `aTermServer`

```
termServer := aTermServer.
^self
```

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 `DhbInfiniteSeriesTermServer`. It is responsible for computing 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 7.2

Listing 7.2 Smalltalk implementation of a term server

Class	`DhbSeriesTermServer`
Subclass of	`Object`
Instance variable names:	`x lastTerm`

Instance Methods

setArgument: `aNumber`

```
x := aNumber asFloat.
```

7.2.2 Infinite Series—Java Implementation

Listing 7.3 shows the implementation of an infinite series in Java. Because the class is abstract, I do not give examples here. Concrete examples are given in Section 7.4.3.

The Java implementation uses an abstract class, `DhbInfiniteSeries`—a subclass of the class `IterativeProcess` described in Section 4.1.2—to evaluate the series. Abstract methods define the common interface needed to retrieve the terms of the series. Because a series is a function, it must implement the `OneVariableFunction` interface. Specific series are implemented as a subclass of the class `DhbInfiniteSeries`, each subclass implementing the methods needed to compute each term.

The argument of the series and each last term are kept in protected instance variables for efficiency purposes. Thus, subclasses can have direct access to these variables without the need to call an accessor method.

The method `evaluateIteration` uses the method `relativePrecision` to return a relative precision, as discussed in Section 4.2.2.

Listing 7.3 Java implementation of an infinite series

```
package DhbIterations;

// InifiniteSeries

// @author Didier H. Besset

public abstract class InifiniteSeries extends IterativeProcess
{

    // Best approximation of the sum.

    private double result;

    // Series argument.

    protected double x;

    // Value of the last term.

    protected double lastTerm;

// Computes the n-th term of the series and stores it in lastTerm.
// @param n int

protected abstract void computeTermAt ( int n);
public double evaluateIteration()
{
    computeTermAt( getIterations());
    result += lastTerm;
    return relativePrecision( Math.abs( lastTerm), Math.abs( result));
}

// @return double

public double getResult ( )
{
    return result;
}

// Set the initial value for the sum.

public void initializeIterations()
{
    result = initialValue();
}

// @return double        the 0-th term of the series

protected abstract double initialValue ( );

// @param r double    the value of the series argument.

public void setArgument ( double r)
{
    x = r;
    return;
}
}
```

7.3 Continued Fractions

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

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

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 this expression is quite awkward to read (besides being a printer's nightmare) one usually uses a linear notation such as that in equation 7.3 [Abramovitz &

Stegun], [Press *et al.*]:

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

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 17th century. A detailed discussion of several methods is given in [Press *et al.*]. In this book, I shall only discuss the modified Lentz's method, which has the advantage to work for a large class of fractions.

Implementing the other methods discussed in [Press *et al.*] 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:

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

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 (i.e., a value that is zero within the precision of the numerical representation). To avoid rounding errors, Thompson and Barnett, in 1986, proposed a modification of the Lentz method in which any value of the coefficients smaller than a small floor value is adjusted to the floor value [Press *et al.*]. 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.

7.3.1 Continued Fractions—Smalltalk Implementation

Listing 7.4 shows the implementation of a continued fraction in Smalltalk.

The class `DhbContinuedFraction` is built as a subclass of the class `DhbInfiniteSeries`. 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

Class `DhbContinuedFraction`

Subclass of `DhbInfiniteSeries`

Instance variable names: `numerator denominator`

Instance Methods

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
```

initializeIterations

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

limitedSmallValue: `aNumber`

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

7.3.2 Continued Fractions—Java Implementation

Listing 7.5 shows the implementation of a continued fraction in Java.

As for infinite series, an abstract class is in charge of implementing the modified Lentz method. As the two classes have nothing in common, there is no reason for the class `ContinuedFraction` to be a subclass of `InfiniteSeries`. It is a subclass of the class `IterativeProcess` instead.

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

Listing 7.5 Java implementation of an infinite series

```
package DhbIterations;
```

```
import DhbFunctionEvaluation.DhbMath;

// Continued fraction
//
// @author Didier H. Besset

public abstract class ContinuedFraction extends IterativeProcess
{

    // Best approximation of the fraction.

    private double result;

    // Fraction's argument.

    protected double x;

    // Fraction's accumulated numerator.

    private double numerator;

    // Fraction's accumulated denominator.

    private double denominator;

    // Fraction's next factors.

    protected double[] factors = new double[2];

// Compute the pair numerator/denominator for iteration n.
// @param n int

protected abstract void computeFactorsAt(int n);

// @return double

public double evaluateIteration()
{
    computeFactorsAt( getIterations());
    denominator = 1 / limitedSmallValue( factors[0] * denominator
                                                    + factors[1]);
    numerator = limitedSmallValue( factors[0] / numerator + factors[1]);
    double delta = numerator * denominator;
    result *= delta;
    return Math.abs( delta - 1);
}
// @return double

public double getResult ( )
{
    return result;
}
public void initializeIterations()
{
    numerator = limitedSmallValue( initialValue());
    denominator = 0;
    result = numerator;
    return;
}

// @return double

protected abstract double initialValue();

// Protection against small factors.
// @return double
// @param r double

private double limitedSmallValue ( double r)
{
    return Math.abs( r) < DhbMath.smallNumber()
                                        ? DhbMath.smallNumber() : r;
}

// @param r double    the value of the series argument.

public void setArgument ( double r)
{
    x = r;
    return;
}
}
```

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 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 ???.

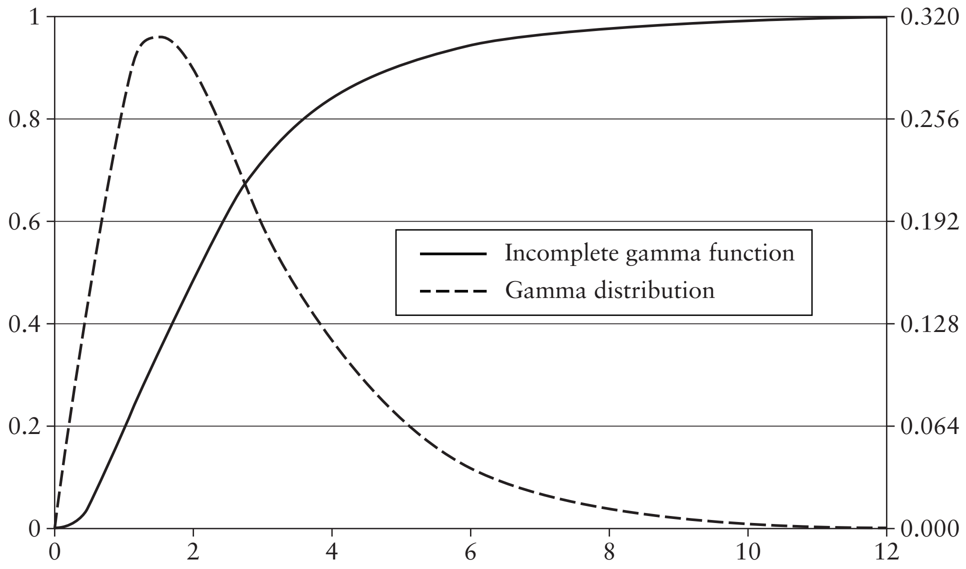

FIG. 7.3 The incomplete gamma function (solid line) and the gamma distribution (dotted line)

Figure 7.3 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 ???. General χ^2 fits are discussed in Section ???.

7.4.1 Mathematical Definitions

The incomplete gamma function is defined by the following integral:

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

Thus, the value of the incomplete gamma function lies between aero 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 (see Section 9.7 for a description of the gamma distribution and its parameters). This integral can be expressed as the following infinite series [Abramovitz & Stegun]:

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

We can see that each term of the series, written in this form, can be computed from the previous one. Using the recurrence formula for the gamma function (equation 2.24 in Section 2.4.1), we have

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

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

The incomplete gamma function can also be written as [Abramovitz & Stegun]:

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

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

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

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

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

It turns out that the continued fraction in equation 7.9 converges for $x > \alpha + 1$ [Press *et al.*]—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 (see the discussion in Section 2.4.1). 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, zero—must be returned separately.

7.4.2 Incomplete Gamma Function—Smalltalk Implementation

Three classes are needed to implement the incomplete gamma function in Smalltalk. The class `DhbIncompleteGamaFunction` is in charge of computing the function itself. This is the object, that 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

Class	`DhbIncompleteGammaFunction`
Subclass of	`Object`
Instance variable names:	`alpha alphaLogGamma series fraction`

Class Methods

shape: `aNumber`

```
^super new initialize: aNumber
```

Instance Methods

evaluateFraction: `aNumber`

```
fraction isNil
    ifTrue:
        [fraction := DhbIncompleteGammaFractionTermServer new.
        fraction setParameter: alpha].
fraction setArgument: aNumber.
^(DhbContinuedFraction server: fraction)
    desiredPrecision: DhbFloatingPointMachine new
                                    defaultNumericalPrecision;
    evaluate
```

evaluateSeries: `aNumber`

```
series isNil
    ifTrue: [ series := DhbIncompleteGammaSeriesTermServer new.
              series setParameter: alpha.
            ].
series setArgument: aNumber.
^(DhbInfiniteSeries server: series)
    desiredPrecision: DhbFloatingPointMachine new
                                    defaultNumericalPrecision;
    evaluate
```

initialize: `aNumber`

```
alpha := aNumber asFloat.
alphaLogGamma := alpha logGamma.
^self
```

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; and 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

Class	`DhbIncompleteGammaSeriesTermServer`
Subclass of	`DhbSeriesTermServer`
Instance variable names:	`alpha sum`

Instance Methods

initialTerm

```
lastTerm := 1 / alpha.
sum := alpha.
^lastTerm
```

setParameter: `aNumber`

```
alpha := aNumber asFloat
```

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

Class	`DhbIncompleteGammaFractionTermServer`
Subclass of	`DhbSeriesTermServer`
Instance variable names:	`alpha`

Instance Methods

initialTerm

```
lastTerm := x - alpha + 1.
^lastTerm
```

setParameter: `aNumber`

```
alpha := aNumber asFloat
```

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 9.7.1.

7.4.3 Incomplete Gamma Function—Java Implementation

In spite of the difference of architecture with Smalltalk, three classes are also needed in the Java implementation.

Listing 7.9 shows the implementation of the class `IncompleteGammaFunction` implementing the incomplete gamma function proper. It is constructed as the corresponding Smalltalk class.

Listing 7.9 Java implementation of the incomplete gamma function

```
package DhbIterations;

import DhbFunctionEvaluation.DhbMath;
import DhbFunctionEvaluation.GammaFunction;

// IncompleteGamma function

// author Didier H. Besset

public class IncompleteGammaFunction implements
DhbInterfaces.OneVariableFunction
{

    // Function parameter.

    private double alpha;

    // Constant to be computed once only.

    private double alphaLogGamma;

    // Infinite series.

    private IncompleteGammaFunctionSeries series;

    // Continued fraction.

    private IncompleteGammaFunctionFraction fraction;

// Constructor method.

public IncompleteGammaFunction ( double a)
{
    alpha = a;
    alphaLogGamma = GammaFunction.logGamma( alpha);
}

// @return double
// @param x double

private double evaluateFraction ( double x)
{
    if ( fraction == null )
    {
        fraction = new IncompleteGammaFunctionFraction( alpha);
        fraction.setDesiredPrecision(
                                DhbMath.defaultNumericalPrecision());
    }
    fraction.setArgument( x);
    fraction.evaluate();
    return fraction.getResult();
}

// @return double        evaluate the series of the incomplete
gamma function.
// @param x double

private double evaluateSeries ( double x)
{
    if ( series == null )
    {
        series = new IncompleteGammaFunctionSeries( alpha);
        series.setDesiredPrecision(
                                DhbMath.defaultNumericalPrecision());
    }
    series.setArgument( x);
    series.evaluate();
    return series.getResult();
}

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

public double value(double x)
{
    if ( x == 0 )
        return 0;
    double norm = Math.exp( Math.log(x) * alpha - x - alphaLogGamma);
    return x - 1 <alpha
                ? evaluateSeries( x) * norm
                : 1 - norm / evaluateFraction( x);
}
}
```

Listing 7.10 shows the implementation of the term server used by the infinite series defining the incomplete gamma function in Java. It is implemented as a subclass of the class `InifiniteSeries` defined in Section 7.2.1. The class `IncompleteGammaFunctionSeries` needs two instance variables: one to store the parameter α; and 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 `initialValue` (for $n = 0$) and `computeTermAt:` (for $n \geq 1$).

Listing 7.10 Java implementation of the infinite series term server for the incomplete gamma function

```
package DhbIterations;

// Series for the incompleteGamma function

// @author Didier H. Besset

public class IncompleteGammaFunctionSeries extends InifiniteSeries
{

    // Series parameter.

    private double alpha;

    // Auxiliary sum.

    private double sum;

// Constructor method
// @param a double    series parameter

public IncompleteGammaFunctionSeries ( double a)
{
    alpha = a;
}

// Computes the n-th term of the series and stores it in lastTerm.
// @param n int

protected void computeTermAt(int n)
{
    sum += 1;
    lastTerm *= x / sum;
    return;
}

// initializes the series and return the 0-th term.

protected double initialValue()
{
    lastTerm = 1 / alpha;
    sum = alpha;
    return lastTerm;
}
}
```

Listing 7.11 shows the implementation of the term server used by the continued fraction defining the incomplete gamma function in Java. It is implemented as a subclass of the class `ContinuedFraction` defined in Section 7.3.2. The class `IncompleteGammaFunctionFraction` needs one instance variable to store the parameter. Equation 7.11 is implemented by the methods `initialValue` (for $n = 0$) and `computeFactorsAt:` (for $n \geq 1$).

Listing 7.11 Java implementation of the fraction term server for the incomplete gamma function

```
package DhbIterations;

// Continued fraction for the incompleteGamma function

// @author Didier H. Besset

public class IncompleteGammaFunctionFraction extends ContinuedFraction
{

    // Series parameter.

    private double alpha;

    // Auxiliary sum.

    private double sum;

// Constructor method.
// @param a double

public IncompleteGammaFunctionFraction ( double a)
{
    alpha = a;
}

// Compute the pair numerator/denominator for iteration n.
// @param n int

protected void computeFactorsAt(int n)
{
    sum += 2;
    factors[0] = ( alpha - n) * n;
    factors[1] = sum;
    return;
}
protected double initialValue()
```

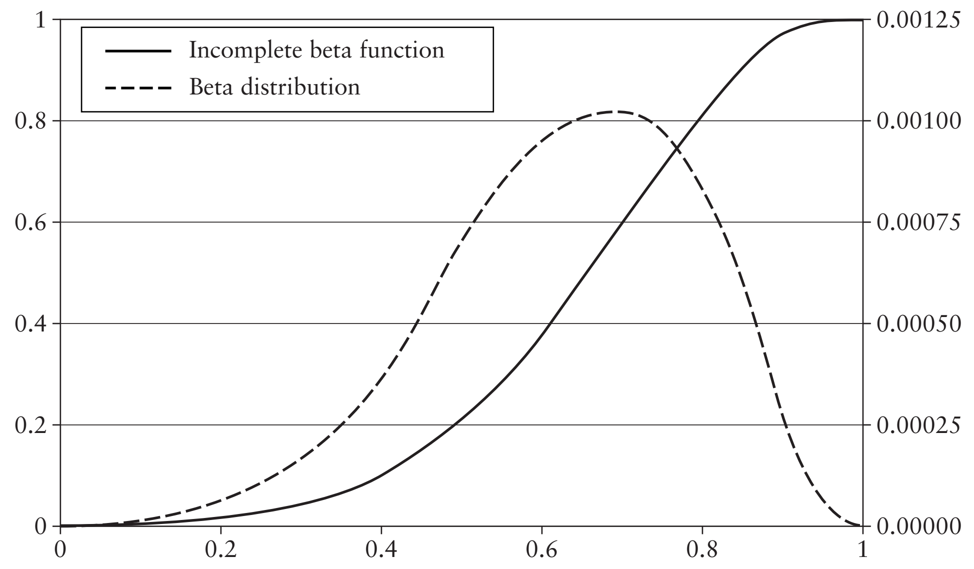

FIG. 7.4 The incomplete beta function (solid line) and the beta distribution (solid line)

```
{
    sum = x - alpha + 1;
    return sum;
}
}
```

An example of use of the incomplete gamma function can be found in Section 9.7.2.

7.5 Incomplete Beta Function

The incomplete beta function is the integral of a beta 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 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, in Section ???, and *F*-test, in Section ???.

Figure 7.4 shows the incomplete beta function (solid line) and its corresponding probability density function (dotted line) for $\alpha_1 = 4.5$ and $\alpha_2 = 2.5$.

7.5.1 Mathematical Definitions

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

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

where $B\left(\alpha_1,\alpha_2\right)$ is the beta function defined in Section 2.5. The function has two parameters, α_1 and α_2. By definition, the value of the incomplete beta function is between 0 and 1.

None of the series expansions of this integral have good numerical convergence. There is, however, a continued fraction development that converges over a sufficient range [Abramovitz & Stegun]:

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

where

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

Using the notation introduced in Section 7.3, we have:

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

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

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

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.

7.5.2 Incomplete Beta Function—Smalltalk Implementation

Listing 7.12 shows the implementation of the incomplete beta function in Smalltalk.

Two classes are needed to implement the incomplete beta function. The class `DhbIncompleteBetaFunction` 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 7.13. 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 7.12 Smalltalk implementation of the incomplete beta function

Class	`DhbIncompleteBetaFunction`
Subclass of	`Object`
Instance variable names:	`alpha1 alpha2 fraction inverseFraction logNorm`

Class Methods

shape: `aNumber1` **shape:** `aNumber2`

```
^super new initialize: aNumber1 shape: aNumber2
```

Instance Methods

evaluateFraction: `aNumber`

```
fraction isNil
    ifTrue:
        [fraction := DhbIncompleteBetaFractionTermServer new.
        fraction setParameter: alpha1 second: alpha2].
fraction setArgument: aNumber.
^(DhbContinuedFraction server: fraction)
    desiredPrecision: DhbFloatingPointMachine new
                                            defaultNumericalPrecision;
    evaluate
```

evaluateInverseFraction: `aNumber`

```
inverseFraction isNil
    ifTrue:
        [inverseFraction := DhbIncompleteBetaFractionTermServer
                                                               new.
        inverseFraction setParameter: alpha2 second: alpha1].
inverseFraction setArgument: (1 - aNumber).
^(DhbContinuedFraction server: inverseFraction)
    desiredPrecision: DhbFloatingPointMachine new
                                     defaultNumericalPrecision;
    evaluate
```

initialize: `aNumber1` **shape:** `aNumber2`

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

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 7.13 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 7.13 Smalltalk implementation of the term server for the incomplete beta function

Class	`DhbIncompleteBetaFractionTermServer`
Subclass of	`DhbSeriesTermServer`
Instance variable names:	`alpha1 alpha2`

Instance Methods

initialTerm

```
^1
```

setParameter: aNumber1 **second:** aNumber2

```
alpha1 := aNumber1.
alpha2 := aNumber2
```

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 ??? and ???.

7.5.3 Incomplete Beta Function—Java Implementation

The Java implementation of the incomplete beta function needs two classes.

Listing 7.14 shows the implementation of the class IncompleteBetaFunction implementing the incomplete beta function proper. It is constructed as the corresponding Smalltalk class.

Listing 7.14 Java implementation of the incomplete beta function

```
package DhbIterations;

import DhbFunctionEvaluation.DhbMath;
import DhbFunctionEvaluation.GammaFunction;
import DhbInterfaces.OneVariableFunction;

// Incomplete Beta function

// @author Didier H. Besset

public class IncompleteBetaFunction implements OneVariableFunction
```

```
{

    // Function parameters.

    private double alpha1;
    private double alpha2;

    // Constant to be computed once only.

    private double logNorm;

    // Continued fractions.

    private IncompleteBetaFunctionFraction fraction;
    private IncompleteBetaFunctionFraction inverseFraction;

// Constructor method.
// @param a1 double
// @param a2 double

public IncompleteBetaFunction ( double a1, double a2)
{
    alpha1 = a1;
    alpha2 = a2;
    logNorm = GammaFunction.logGamma( alpha1 + alpha2)
                        - GammaFunction.logGamma( alpha1)
                        - GammaFunction.logGamma( alpha2);
}

// @return double
// @param x double

private double evaluateFraction ( double x)
{
    if ( fraction == null )
    {
        fraction = new IncompleteBetaFunctionFraction( alpha1, alpha2);
        fraction.setDesiredPrecision( DhbMath.
                              defaultNumericalPrecision());
    }
    fraction.setArgument( x);
    fraction.evaluate();
    return fraction.getResult();
}

// @return double
// @param x double
private double evaluateInverseFraction ( double x)
{
    if ( fraction == null )
    {
        fraction = new IncompleteBetaFunctionFraction( alpha2, alpha1);
        fraction.setDesiredPrecision( DhbMath.
                                      defaultNumericalPrecision());
    }
    fraction.setArgument( x);
    fraction.evaluate();
    return fraction.getResult();
}
public double value(double x)
{
    if ( x == 0 )
        return 0;
    if ( x == 1 )
        return 1;
    double norm = Math.exp( alpha1 * Math.log(x)
                            + alpha2 * Math.log(1 - x) + logNorm);
    return ( alpha1 + alpha2 + 2) * x < ( alpha1 + 1)
                    ? norm / ( evaluateFraction( x) * alpha1)
                    : 1 - norm / ( evaluateInverseFraction(1 - x)
                                                        * alpha2);
}
}
```

Listing 7.15 shows the implementation of the term server used by the continued fraction defining the incomplete beta function in Java. 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 `computeFactorsAt:` (for $n \geq 1$).

Listing 7.15 Java implementation of the continued fraction for the incomplete beta function

```
package DhbIterations;

// Incomplete Beta function fraction

// @author Didier H. Besset

public class IncompleteBetaFunctionFraction extends ContinuedFraction
{

    // Fraction's parameters.
    private double alpha1;
    private double alpha2;

// Constructor method.
// @param a1 double
// @param a2 double

public IncompleteBetaFunctionFraction ( double a1, double a2)
{
    alpha1 = a1;
    alpha2 = a2;
}

// Compute the pair numerator/denominator for iteration n.
// @param n int

protected void computeFactorsAt(int n)
{
    int m = n / 2;
    int m2 = 2 * m;
    factors[0] = m2 == n
                    ? x * m * ( alpha2 - m)
                                / ( (alpha1 + m2) * (alpha1 + m2 - 1))
                    : -x * ( alpha1 + m) * (alpha1 + alpha2 + m)
                                / ( (alpha1 + m2) * (alpha1 + m2 + 1));
    return;
}
protected double initialValue()
{
    factors[1] = 1;
    return 1;
}
}
```

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 college, and solving systems of linear equations are even taught in high school. Of course, one must get used to the bookkeeping 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 examples of the 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 with the problem of finding eigenvalues and eigenvectors.

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

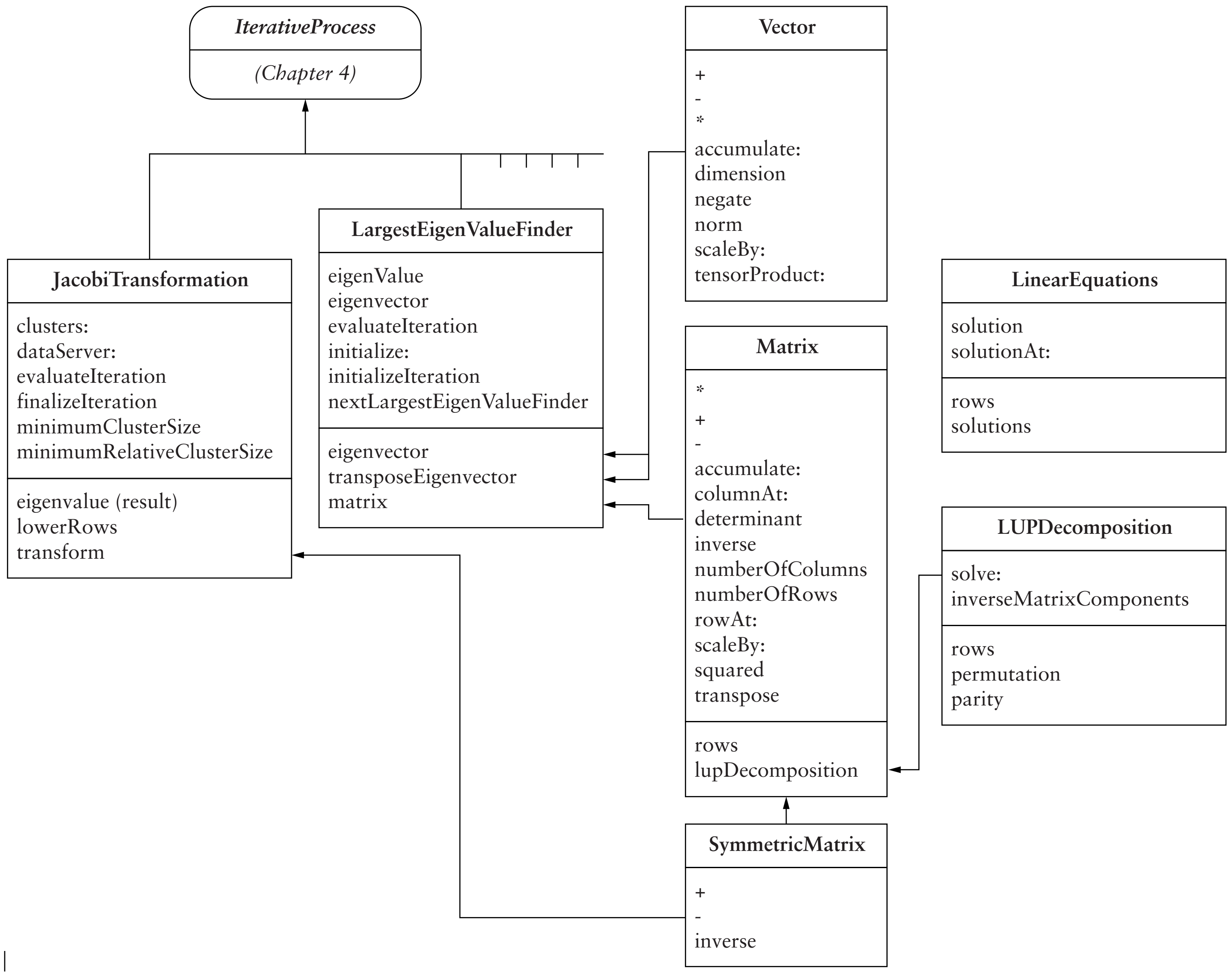

FIG. 8.1 Linear algebra classes

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 lowercase bold letter, $\mathbf{v}$, 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.

A matrix is commonly represented with an uppercase bold letter $\mathbf{M}$, for example. The application of the matrix $\mathbf{M}$ on the vector $\mathbf{M}$ 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 $\mathbf{M}$, any vectors $\mathbf{u}$ and $\mathbf{v}$, and 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 $\mathbf{v}$ 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 $\mathbf{A}$ is represented by

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

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

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 $\mathbf{A}$ 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 $\mathbf{B}$, $\mathbf{C}$, $\mathbf{D}$, and $\mathbf{E}$ are all matrices of lower dimensions. Let $\mathbf{B}$ be a p by q matrix. Then, $\mathbf{C}$ is a p by $m-q$ matrix, $\mathbf{D}$ is a $n-p$ by q matrix and $\mathbf{E}$ 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

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

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 in the following equations. 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:

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

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

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

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, respectively dimensions n and m is an n by m matrix:

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

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

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

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

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

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

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

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:

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

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

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

A *square* matrix is a matrix that has the same number of rows and columns. To shorten sentences, I 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 symmetrical 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 **A**. If the matrix **A** represented in equation 8.4 is symmetric, we have $\mathbf{D}=\mathbf{C}^{\mathrm{T}}$.

8.1.1 Vector and Matrix—Smalltalk Implementation

Listings 8.1 and 8.2 show, respectively, the implementation of vectors and matrices as Smalltalk classes. A special implementation for symmetrical matrices is shown in Listing 8.3.

The public interface is designed so as to map itself as closely as possible to the mathematical definitions. Code Example 8.1 uses operations between vectors and matrices. In the first two lines after the declarative statement, the vectors **u** and **v** are defined from their component array using the creator method `asVector`. They are three-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 (cf. this expression with equation 8.1).

Code Example 8.1

```
| u v w a b c|
u := #(1 2 3) asVector.
v := #(3 4 5) asVector.
a := DhbMatrix rows: #( ( 1 0 1 ) (-1 -2 3)).
b := DhbMatrix 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.
```

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 two-dimensional vector. Here again the correspondence between the Smalltalk 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 transposition of the matrix **a** explicit, whereas the second line uses 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 Smalltalk class `Array`, for which mathematical operations have been defined. Thus, a vector in Smalltalk 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
3. a vector and a matrix

Thus, the multiplication will be implemented using double dispatching as explained in Section 2.2.1 for operations between polynomials. Double dispatching is described in Section ???.

The method `asVector` is defined for compatibility with a similar method defined in the class `Array` to construct a vector out of an array object. A method `asVector` must also be defined for instances of the class `Collection`. This is left as an exercise for the reader.

The method `tensorProduct` returns an instance of a symmetrical matrix. This class is defined in Listing 8.3.

The method `accumulate` is meant to be used when there is a need to add several vectors. Indeed, Code Example 8.2

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.

Code Example 8.2

```
|a b c d e|
a := #(1 2 3 4 5) asVector.
b := #(2 3 4 5 6) asVector.
c := #(3 4 5 6 7) asVector.
d := #(4 5 6 7 8) asVector.
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 (see Code Example 8.3).

Code Example 8.3

```
|a b c d e|
a := #(1 2 3 4 5) asVector.
b := #(2 3 4 5 6) asVector.
c := #(3 4 5 6 7) asVector.
d := #(4 5 6 7 8) asVector.
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 on the garbage collector.

Listing 8.1 Vector class in Smalltalk

Class	`DhbVector`
Subclass of	`Array`

Instance Methods

* `aNumberOrMatrixOrVector`

```
^aNumberOrMatrixOrVector productWithVector: self
```

+ `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
```

- 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
```

accumulate: aVectorOrAnArray

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

accumulateNegated: aVectorOrAnArray

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

asVector

```
^self
```

dimension

```
^self size
```

negate

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

norm

```
^(self * self) sqrt
```

normalized

```
^(1 / self norm) * self
```

productWithMatrix: aMatrix

```
^aMatrix rowsCollect: [ :each | each * self]
```

productWithVector: aVector

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

scaleBy: `aNumber`

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

tensorProduct: `aVector`

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

The class `DhbMatrix` 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 `DhbLUPDecomposition` 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.

The reader should compare the Smalltalk code with the Java code. The Java implementation makes the index management explicit. Since there are no iterator methods in Java, there is no other choice.

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 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 8.2 Matrix classes in Smalltalk

Class	`DhbMatrix`
Subclass of	`Object`
Instance variable names:	rows lupDecomposition

Class Methods

new: `anInteger`

```
^self new initialize: anInteger
```

rows: `anArrayOrVector`

```
^self new initializeRows: anArrayOrVector
```

Instance Methods

* `aNumberOrMatrixOrVector`

```
^aNumberOrMatrixOrVector productWithMatrix: self
```

+ `aMatrix`

```
^aMatrix addWithRegularMatrix: self
```

- `aMatrix`

```
^aMatrix subtractWithRegularMatrix: self
```

accumulate: `aMatrix`

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

accumulateNegated: `aMatrix`

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

addWithMatrix: `aMatrix` **class:** `aMatrixClass`

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

addWithRegularMatrix: `aMatrix`

```
^aMatrix addWithMatrix: self class: aMatrix class
```

addWithSymmetricMatrix: `aMatrix`

```
^aMatrix addWithMatrix: self class: self class
```

asSymmetricMatrix

```
^DhbSymmetricMatrix rows: rows
```

columnAt: `anInteger`

```
^rows collect: [ :each | each at: anInteger]
```

columnsCollect: `aBlock`

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

columnsDo: `aBlock`

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

initialize: `anInteger`

```
rows := ( 1 to: anInteger) asVector collect: [ :each |
                                        DhbVector new: anInteger].
```

initializeRows: `anArrayOrVector`

```
rows := anArrayOrVector asVector collect: [ :each | each asVector].
```

isSquare

```
^rows size = rows last size
```

isSymmetric

```
^false
```

lupDecomposition

```
lupDecomposition isNil
    ifTrue: [ lupDecomposition :=
                                  DhbLUPDecomposition equations: rows].
^lupDecomposition
```

negate

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

numberOfColumns

```
^rows last size
```

numberOfRows

```
^rows size
```

printOn: `aStream`

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

productWithMatrix: `aMatrix`

```
^self productWithMatrixFinal: aMatrix
```

productWithMatrixFinal: `aMatrix`

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

productWithSymmetricMatrix: aSymmetricMatrix

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

productWithTransposeMatrix: aMatrix

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

productWithVector: aVector

```
^self columnsCollect: [ :each | each * aVector]
```

rowAt: anInteger

```
^rows at: anInteger
```

rowsCollect: aBlock

```
^rows collect: aBlock
```

rowsDo: aBlock

```
^rows do: aBlock
```

scaleBy: aNumber

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

squared

```
^DhbSymmetricMatrix rows: ( self columnsCollect: [ :col |
                     self columnsCollect: [ :colT | col * colT]])
```

subtractWithMatrix: aMatrix **class:** aMatrixClass

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

subtractWithRegularMatrix: aMatrix

```
^aMatrix subtractWithMatrix: self class: aMatrix class
```

subtractWithSymmetricMatrix: aMatrix

```
^aMatrix subtractWithMatrix: self class: self class
```

transpose

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

transposeProductWithMatrix: `aMatrix`

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

Listing 8.3 shows the implementation of the class `DhbSymmetricMatrix` representing symmetrical matrices. A few algorithms are implemented differently for symmetrical 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 symmetrical matrices yields a symmetrical matrix whereas the same operations between a symmetrical matrix and a normal matrix yield a normal matrix. Product requires quadruple dispatching.

Listing 8.3 Symmetric matrix classes in Smalltalk

Class	`DhbSymmetricMatrix`
Subclass of	`DhbMatrix`

Class Methods

identity: `anInteger`

```
^self new initializeIdentity: anInteger
```

Instance Methods

+ `aMatrix`

```
^aMatrix addWithSymmetricMatrix: self
```

- `aMatrix`

```
^aMatrix subtractWithSymmetricMatrix: self
```

addWithSymmetricMatrix: `aMatrix`

```
^aMatrix addWithMatrix: self class: self class
```

clear

```
rows do: [ :each | each atAllPut: 0].
```

initializeIdentity: `anInteger`

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

isSquare

```
^true
```

isSymmetric

```
^true
```

productWithMatrix: `aMatrix`

```
^aMatrix productWithSymmetricMatrix: self
```

productWithSymmetricMatrix: `aSymmetricMatrix`

```
^aSymmetricMatrix productWithMatrixFinal: self
```

subtractWithSymmetricMatrix: `aMatrix`

```
^aMatrix subtractWithMatrix: self class: self class
```

8.1.2 Vector and Matrix—Java Implementation

Listings 8.4 and 8.5 show, respectively, the implementation of vectors and matrices as Smalltalk classes.

The public interface is designed so as to map itself as closely as possible to the mathematical definitions. In Java, however, one cannot overload primitive operators such as + or *. Thus, named methods are used instead. The resulting code is admittedly less readable than its Smalltalk equivalent.

Some operations (e.g., a scalar product between two vectors of different dimension) are not possible. In our implementation, we have introduced a few specific exceptions—`DhbIllegalDimension`, `DhbNonSymmetricComponents`—that must be trapped on each operation.

Code Example 8.4 uses operations between vectors and matrices:

Code Example 8.4

```
double[] comp_u = {1, 2, 3};
double[] comp_v = {3, 4, 5};
double[][] comp_a = {{1,0,1}, {-1, -2, 3}};
double[][] comp_b = {{1,2,3}, {-2, -1, 7},{5,6,7}}};
try{
    DhbVector u = new DhbVector(components);
    DhbVector v = new DhbVector(components);
```

```
 Matrix a = new Matrix(comp_a);
 Matrix b = new Matrix(comp_b);
 DhbVector w = u.product(4).add(v.product(3));
 Matrix c = a.product(b);
 v = a.product(u));
 w = c.transpose().product(v);
 w = v.product(c);
} catch ( DhbIllegalDimension e){};
```

The first four lines define the component arrays used to define the vectors and matrices. Within the `try...catch`, the vectors **u** and **v** are created from their component arrays. They are three-dimensional vectors. The matrices **a** and **b** are also created from their component arrays. 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**.

Next the matrix **c** is defined as the product of the matrices **a** and **b** in this order.

The next assignment redefines the vector **v** as the product of the matrix **A** with the vector **u**. It is now a two-dimensional vector.

The last two lines compute the vector **w** as the transpose product with the matrix **a**. The result of both line is the same[4]. The first line builds the transpose of the matrix **a** explicitly, whereas the second line uses the 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 named methods—as opposed to using primitive operators—obscures somewhat the readability of the code. For an objective comparison, Java programmers are advised to take a look at the code example in Smalltalk at the beginning of Section 8.1.1.

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.

Since a Java array is not an object, a vector is created as a subclass of object. The name `Vector` could not be used since it is already part of the base classes of Java. An instance of the Java class `Vector`, however, has little to do with a mathematical vector. The Java class `Vector` has been totally misnamed. In fact, it corresponds to the Smalltalk class `OrderedCollection`. An instance of the Java class `Vector` has a variable size, whereas the dimension of a mathematical vector is fixed. In the rest of this book, whenever the word *vector* is used, it means a mathematical vector and not an instance of the Java class `Vector`.

Operations such as addition and products cannot be implemented with primitive operators (e.g., $+$, $-$) since Java does not allow the definition of primitive operators on objects. Thus, we are forced to use method names such as `add` or `product`.

4. See footnote 3 on page 205

In Java a method is not only defined by its name but also by the type of its arguments. Thus, the same method name can be used to define implementations of an operator acting on different objects without the need to test the arguments. The Java compiler does that for us; double or triple dispatching is not needed.

Listing 8.4 defines the method `tensorProduct` using the class `SymmetricMatrix` that will be defined in Section 8.5 (see Listing 8.12), as most of the code for symmetrical matrices is devoted to matrix inversion.

An exception specific for this package has been created: `DhbIllegalDimension`. This exception is thrown when an operation is attempted between vectors or matrices with incompatible dimensions. The code for this exception is elementary and is not shown here. It is left as an exercise for the reader.

Vector components are kept in a protected instance variable. *Protected* means that it is available only to classes of the same package. This allows the optimization of many methods from other classes of the same package needing direct access to the components (e.g., `Matrix`). Other users of the class have no access to the vector's components. This choice is deliberate: it provides a strict hiding of the implementation. A vector as mathematical object should be manipulated without any access to its components.

The method `toString` is provided to allow printing of the components. The method `toComponents` provides a way to access the coordinates if needed. Note that this method is making a copy of the components, thus preserving the object from any indirect change.

Listing 8.4 Mathematical vector class in Java

```
package DhbMatrixAlgebra;

// Vector implementation

// @author Didier H. Besset

public class DhbVector
{
    protected double[] components;

// Create a vector of given dimension.
// NOTE: The supplied array of components must not be changed.
// @param comp double[]

public DhbVector(  double comp[]) throws NegativeArraySizeException
{
    int n = comp.length;
    if ( n <= 0 )
        throw new NegativeArraySizeException(
                                "Vector components cannot be empty");
    components = new double[n];
    System.arraycopy( comp, 0, components, 0, n);
}

// Create a vector of given dimension.
// @param dimension int dimension of the vector; must be positive.

public DhbVector ( int dimension) throws NegativeArraySizeException
{
    if ( dimension <= 0 )
        throw new NegativeArraySizeException(
                                "Requested vector size: "+dimension);
    components = new double[dimension];
    clear();
}

// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
// and supplied vector do not have the same dimension.

public void accumulate ( double[] x) throws DhbIllegalDimension
{
    if ( this.dimension() != x.length )
        throw new DhbIllegalDimension("Attempt to add a "
                    +this.dimension()+"-dimension vector to a "
                                +x.length+"-dimension array");
    for ( int i = 0; i < this.dimension(); i++)
        components[i] += x[i];
}

// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
// and supplied vector do not have the same dimension.

public void accumulate ( DhbVector v) throws DhbIllegalDimension
{
    if ( this.dimension() != v.dimension() )
        throw new DhbIllegalDimension("Attempt to add a "
                    +this.dimension()+"-dimension vector to a "
                            +v.dimension()+"-dimension vector");
    for ( int i = 0; i < this.dimension(); i++)
        components[i] += v.components[i];
}

// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
// and supplied vector do not have the same dimension.

public void accumulateNegated( double[] x) throws DhbIllegalDimension
{
    if ( this.dimension() != x.length )
        throw new DhbIllegalDimension("Attempt to add a "
                        +this.dimension()+"-dimension vector to a "
                                        +x.length+"-dimension array");
    for ( int i = 0; i < this.dimension(); i++)
        components[i] -= x[i];
}

// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
// and supplied vector do not have the same dimension.

public void accumulateNegated( DhbVector v) throws DhbIllegalDimension
{
    if ( this.dimension() != v.dimension() )
        throw new DhbIllegalDimension("Attempt to add a "
                        +this.dimension()+"-dimension vector to a "
                                +v.dimension()+"-dimension vector");
    for ( int i = 0; i < this.dimension(); i++)
        components[i] -= v.components[i];
}

// @return DHBmatrixAlgebra.DhbVector sum of the vector with
//                            the supplied vector
// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
//              and supplied vector do not have the same dimension.

public DhbVector add ( DhbVector v) throws DhbIllegalDimension
{
    if ( this.dimension() != v.dimension() )
        throw new DhbIllegalDimension("Attempt to add a "
                        +this.dimension()+"-dimension vector to a "
                                +v.dimension()+"-dimension vector");
    double[] newComponents = new double[ this.dimension()];
    for ( int i = 0; i < this.dimension(); i++)
        newComponents[i] = components[i] + v.components[i];
    return new DhbVector( newComponents);
}

// Sets all components of the receiver to 0.
```

```
public void clear()
{
    for ( int i = 0; i < components.length; i++) components[i] = 0;
}

// @return double
// @param n int

public double component( int n)
{
    return components[n];
}

// Returns the dimension of the vector.
// @return int

public int dimension ( )
{
    return components.length;
}

// @return true if the supplied vector is equal to the receivereec
// @param v DHBmatrixAlgebra.DhbVector

public boolean equals( DhbVector v)
{
    int n = this.dimension();
    if ( v.dimension() != n )
        return false;
    for ( int i = 0; i < n; i++)
    {
        if ( v.components[i] != components[i] )
            return false;
    }
    return true;
}

// Computes the norm of a vector.

public double norm ( )
{
    double sum = 0;
    for ( int i = 0; i < components.length; i++)
        sum += components[i]*components[i];
    return Math.sqrt( sum);
}
```

```
// @param x double

public DhbVector normalizedBy ( double x )
{
    for ( int i = 0; i < this.dimension(); i++)
        components[i] /= x;
    return this;
}

// Computes the product of the vector by a number.
// @return DHBmatrixAlgebra.DhbVector
// @param d double

public DhbVector product( double d)
{
    double newComponents[] = new double[components.length];
    for ( int i = 0; i < components.length; i++)
        newComponents[i] = d * components[i];
    return new DhbVector(newComponents);
}

// Compute the scalar product (or dot product) of two vectors.
// @return double the scalar product of the receiver with the
argument
// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the
dimension
//                                of v is not the same.

public double product ( DhbVector v) throws DhbIllegalDimension
{
    int n = v.dimension();
    if ( components.length != n )
        throw new DhbIllegalDimension(
                    "Dot product with mismatched dimensions: "
                    +components.length+", "+n);
    return secureProduct( v);
}

// Computes the product of the transposed vector with a matrix
// @return MatrixAlgebra.DhbVector
// @param a MatrixAlgebra.Matrix

public DhbVector product ( Matrix a) throws DhbIllegalDimension
{
    int n = a.rows();
    int m = a.columns();
    if ( this.dimension() != n )
        throw new DhbIllegalDimension(
                    "Product error: transposed of a "+this.dimension()
                    +"-dimension vector cannot be multiplied with a "
                                            +n +" by "+m+" matrix");
    return secureProduct( a);
}

// @param x double

public DhbVector scaledBy ( double x )
{
    for ( int i = 0; i < this.dimension(); i++)
        components[i] *= x;
    return this;
}

// Compute the scalar product (or dot product) of two vectors.
// No dimension checking is made.
// @return double the scalar product of the receiver with the argument
// @param v DHBmatrixAlgebra.DhbVector

protected double secureProduct ( DhbVector v)
{
    double sum = 0;
    for ( int i = 0; i < v.dimension(); i++)
        sum += components[i]*v.components[i];
    return sum;
}

// Computes the product of the transposed vector with a matrix
// @return MatrixAlgebra.DhbVector
// @param a MatrixAlgebra.Matrix

protected DhbVector secureProduct ( Matrix a)
{
    int n = a.rows();
    int m = a.columns();
    double[] vectorComponents = new double[m];
    for ( int j = 0; j < m; j++ )
    {
        vectorComponents[j] = 0;
        for ( int i = 0; i < n; i++)
            vectorComponents[j] += components[i] * a.components[i][j];
    }
    return new DhbVector( vectorComponents);
}

// @return DHBmatrixAlgebra.DhbVector    subtract the supplied vector
//                                      to the receiver
// @param v DHBmatrixAlgebra.DhbVector
// @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
// and supplied vector do not have the same dimension.

public DhbVector subtract ( DhbVector v) throws DhbIllegalDimension
{
    if ( this.dimension() != v.dimension() )
        throw new DhbIllegalDimension("Attempt to add a "
                        +this.dimension()+"-dimension vector to a "
                                +v.dimension()+"-dimension vector");
    double[] newComponents = new double[ this.dimension()];
    for ( int i = 0; i < this.dimension(); i++)
        newComponents[i] = components[i] - v.components[i];
    return new DhbVector( newComponents);
}

// @return MatrixAlgebra.Matrix    tensor product with the specified
//                                          vector
// @param v MatrixAlgebra.DhbVector    second vector to build tensor
//                                      product with.

public Matrix tensorProduct ( DhbVector v)
{
    int n = dimension();
    int m = v.dimension();
    double [][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < m; j++)
            newComponents[i][j] = components[i] * v.components[j];
    }
    return n == m ? new SymmetricMatrix( newComponents)
                      : new Matrix( newComponents);
}

// @return double[]    a copy of the components of the receiver.

public double[] toComponents ( )
{
    int n = dimension();
    double[] answer = new double[ n];
```

```
        System.arraycopy( components, 0, answer, 0, n);
        return answer;
    }

    // Returns a string representation of the vector.
    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        char[] separator = { '[', ' '};
        for ( int i = 0; i < components.length; i++)
        {
            sb.append( separator);
            sb.append( components[i]);
            separator[0] = ',';
        }
        sb.append(']');
        return sb.toString();
    }
    }
```

Listing 8.5 shows the implementation of matrices as Java classes.

Most of the remarks made for vectors are also valid for matrices. In particular, the components of the matrix are strictly encapsulated within the package using the protected declaration keyword.

The CRL inversion algorithm discussed in section 8.5 uses multiplication with a transpose matrix. Thus, a special method has been introduced to optimize this operation. This avoids allocating a memory block needed to hold the transposed matrix. Since a matrix can be very large, allocating a block of memory can seriously slow down execution, especially when it causes paging.

This implementation of the class `DhbMatrix` uses the fact that a class `SymmetricMatrix` will be introduced to implement the inversion algorithm discussed in section 8.5. Thus, the computation of addition, subtraction, and product is made in two steps. First the components are created, then the resulting matrix is created with the appropriate class. In the class `SymmetricMatrix`, only the creation of the resulting matrix needs to be implemented.

For further optimization, each matrix operation comes in two versions: secure and normal. The secure version assumes that the dimensions of the supplied parameters are already checked. These methods are declared as protected since they are reserved for use by other classes of the same package. The normal operations available to any classes first check the arguments for the proper dimensions before carrying the operation using the secure version of the same method.

Unlike the vector implementation, however, strict encapsulation of matrix components is not guaranteed since creation of a matrix from a two-dimensional array does not copy the components into a new array. This was done to reduce memory usage. Memory consumption does more to slow down program execution than anything else with modern-day operating systems. The size of a matrix can become quite significant when working with large dimensions. Thus, a small breach to encapsulation was made in the constructor method. A fully encapsulated version would copy the components as is done for the constructor method of vectors. The user of the package must be aware of this potential problem.

Listing 8.5 Matrix class in Java

```
package DhbMatrixAlgebra;


// Class representing matrix

//  @author Didier H. Besset

public class Matrix
{
    protected double[][] components;
    protected LUPDecomposition lupDecomposition = null;

// Creates a matrix with given components.
// NOTE: the components must not be altered after the definition.
// @param a double[][]

public Matrix ( double[][] a)
{
    components = a;
}

// Creates a null matrix of given dimensions.
// @param n int    number of rows
// @param m int    number of columns
// @exception NegativeArraySizeException

public Matrix ( int n, int m) throws NegativeArraySizeException
{
    if ( n <= 0 || m <= 0)
        throw new NegativeArraySizeException(
                                  "Requested matrix size: "+n+" by "+m);
    components = new double[n][m];
    clear();
}
```

```
// @param a MatrixAlgebra.Matrix
// @exception MatrixAlgebra.DhbIllegalDimension if the supplied matrix
//                         does not have the same dimensions.

public void accumulate ( Matrix a) throws DhbIllegalDimension
{
    if ( a.rows() != rows() || a.columns() != columns() )
        throw new DhbIllegalDimension("Operation error: cannot add a"
                                +a.rows()+" by "+a.columns()
                                    +" matrix to a "+rows()+" by "
                                                +columns()+" matrix");
    int m = components[0].length;
    for ( int i = 0; i < components.length; i++)
    {
        for ( int j = 0; j < m; j++)
            components[i][j] += a.component(i,j);
    }
}

// @return MatrixAlgebra.Matrix      sum of the receiver with the
//                                   supplied matrix.
// @param a MatrixAlgebra.Matrix
// @exception MatrixAlgebra.DhbIllegalDimension if the supplied matrix
//                         does not have the same dimensions.

public Matrix add ( Matrix a) throws DhbIllegalDimension
{
    if ( a.rows() != rows() || a.columns() != columns() )
        throw new DhbIllegalDimension("Operation error: cannot add a"
                                    +a.rows()+" by "+a.columns()
                                        +" matrix to a "+rows()+" by "
                                                +columns()+" matrix");
    return new Matrix( addComponents( a));
}

// Computes the components of the sum of the receiver and
//                                   a supplied matrix.
// @return double[][]
// @param a MatrixAlgebra.Matrix

protected double[][] addComponents ( Matrix a)
{
    int n = this.rows();
    int m = this.columns();
    double[][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            newComponents[i][j] = components[i][j] + a.components[i][j];
    }
    return newComponents;
}
public void clear()
{
    int m = components[0].length;
    for ( int i = 0; i < components.length; i++)
    {
        for ( int j = 0; j < m; j++) components[i][j] = 0;
    }
}

// @return int    the number of columns of the matrix

public int columns ( )
{
    return components[0].length;
}

// @return double
// @param n int
// @param m int

public double component( int n, int m)
{
    return components[n][m];
}

// @return double
// @exception MatrixAlgebra.DhbIllegalDimension if the supplied
//                                    matrix is not square.

public double determinant () throws DhbIllegalDimension
{
    return lupDecomposition().determinant();
}

// @return true if the supplied matrix is equal to the receiver.
// @param a MatrixAlgebra.Matrix

public boolean equals( Matrix a)
{
    int n = this.rows();
    if ( a.rows() != n )
        return false;
    int m = this.columns();
    if ( a.columns() != m )
        return false;
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
        {
            if ( a.components[i][j] != components[i][j] )
                return false;
        }
    }
    return true;
}

// @return DhbMatrixAlgebra.DhbMatrix        inverse of the receiver
//            or pseudoinverse if the receiver is not a square matrix.
// @exception java.lang.ArithmeticException if the receiver is
//                                         a singular matrix.

public Matrix inverse ( ) throws ArithmeticException
{
    try { return new Matrix(
                    lupDecomposition().inverseMatrixComponents());}
        catch ( DhbIllegalDimension e )
            {  return new Matrix(
                    transposedProduct().inverse()
                            .productWithTransposedComponents( this));}
}

// @return boolean

public boolean isSquare ( )
{
    return rows() == columns();
}

// @return LUPDecomposition    the LUP decomposition of the receiver.
// @exception DhbIllegalDimension if the receiver is not
//                                         a square matrix.

protected LUPDecomposition lupDecomposition()
                                        throws DhbIllegalDimension
{
    if ( lupDecomposition == null )
```

```
        lupDecomposition = new LUPDecomposition( this);
    return lupDecomposition;
}

// @return MatrixAlgebra.Matrix      product of the matrix with
//                                   a supplied number
// @param a double   multiplicand.

public Matrix product ( double a)
{
    return new Matrix( productComponents( a));
}

// Computes the product of the matrix with a vector.
// @return DHBmatrixAlgebra.DhbVector
// @param v DHBmatrixAlgebra.DhbVector

public DhbVector product ( DhbVector v) throws DhbIllegalDimension
{
    int n = this.rows();
    int m = this.columns();
    if ( v.dimension() != m )
        throw new DhbIllegalDimension("Product error: "+n+" by "+m
            +" matrix cannot by multiplied with vector of dimension "
                                                    +v.dimension());
    return secureProduct( v);
}

// @return MatrixAlgebra.Matrix      product of the receiver with the
//                                   supplied matrix
// @param a MatrixAlgebra.Matrix
// @exception MatrixAlgebra.DhbIllegalDimension If the number of
//                columns of the receiver are not equal to the
//                      number of rows of the supplied matrix.

public Matrix product ( Matrix a) throws DhbIllegalDimension
{
    if ( a.rows() != columns() )
        throw new DhbIllegalDimension(
                            "Operation error: cannot multiply a"
                                    +rows()+" by "+columns()
                                        +" matrix with a "+a.rows()
                                            +" by "+a.columns()
                                                        +" matrix");
    return new Matrix( productComponents( a));
}
```

```
// @return double[][]
// @param a double

protected double[][] productComponents ( double a)
{
    int n = this.rows();
    int m = this.columns();
    double[][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < m; j++)
            newComponents[i][j] = a * components[i][j];
    }
    return newComponents;
}

// @return double[][]    the components of the product of the receiver
//                                    with the supplied matrix
// @param a MatrixAlgebra.Matrix

protected double[][] productComponents ( Matrix a)
{
    int p = this.columns();
    int n = this.rows();
    int m = a.columns();
    double[][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < m; j++)
        {
            double sum = 0;
            for ( int k = 0; k < p; k++)
                sum += components[i][k] * a.components[k][j];
            newComponents[i][j] = sum;
        }
    }
    return newComponents;
}

// @return MatrixAlgebra.Matrix    product of the receiver with the
//                                tranpose of the supplied matrix
// @param a MatrixAlgebra.Matrix
// @exception MatrixAlgebra.DhbIllegalDimension If the number of
//                      columns of the receiver are not equal to
//                      the number of columns of the supplied matrix.
```

```
public Matrix productWithTransposed ( Matrix a)
                                            throws DhbIllegalDimension
{
    if ( a.columns() != columns() )
        throw new DhbIllegalDimension(
                        "Operation error: cannot multiply a "+rows()
                            +" by "+columns()
                                +" matrix with the transpose of a "
                                    +a.rows()+" by "+a.columns()
                                                    +" matrix");
    return new Matrix( productWithTransposedComponents( a));
}

// @return double[][]   the components of the product of the receiver
//                      with the transpose of the supplied matrix
// @param a MatrixAlgebra.Matrix

protected double[][] productWithTransposedComponents ( Matrix a)
{
    int p = this.columns();
    int n = this.rows();
    int m = a.rows();
    double[][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < m; j++)
        {
            double sum = 0;
            for ( int k = 0; k < p; k++)
                sum += components[i][k] * a.components[j][k];
            newComponents[i][j] = sum;
        }
    }
    return newComponents;
}

// @return int   the number of rows of the matrix

public int rows ( )
{
    return components.length;
}

// Computes the product of the matrix with a vector.
// @return DHBmatrixAlgebra.DhbVector
// @param v DHBmatrixAlgebra.DhbVector
```

```
protected DhbVector secureProduct ( DhbVector v)
{
    int n = this.rows();
    int m = this.columns();
    double[] vectorComponents = new double[n];
    for ( int i = 0; i < n; i++ )
    {
        vectorComponents[i] = 0;
        for ( int j = 0; j < m; j++)
            vectorComponents[i] += components[i][j] * v.components[j];
    }
    return new DhbVector( vectorComponents);
}

// Same as product(Matrix a), but without dimension checking.
// @return MatrixAlgebra.Matrix      product of the receiver with the
//                                   supplied matrix
// @param a MatrixAlgebra.Matrix

protected Matrix secureProduct ( Matrix a)
{
    return new Matrix( productComponents( a));
}

// Same as subtract ( DhbMarix a), but without dimension checking.
// @return MatrixAlgebra.Matrix
// @param a MatrixAlgebra.Matrix

protected Matrix secureSubtract ( Matrix a)
{
    return new Matrix( subtractComponents( a));
}

// @return MatrixAlgebra.Matrix      subtract the supplied matrix to
//                                   the receiver.
// @param a MatrixAlgebra.Matrix
// @exception MatrixAlgebra.DhbIllegalDimension if the supplied matrix
//                          does not have the same dimensions.

public Matrix subtract ( Matrix a) throws DhbIllegalDimension
{
    if ( a.rows() != rows() || a.columns() != columns() )
        throw new DhbIllegalDimension(
                    "Product error: cannot subtract a"+a.rows()
                            +" by "+a.columns()+" matrix to a "
                                +rows()+" by "+columns()+" matrix");
    return new Matrix( subtractComponents( a));
}

// @return double[][]
// @param a MatrixAlgebra.Matrix

protected double[][] subtractComponents ( Matrix a)
{
    int n = this.rows();
    int m = this.columns();
    double[][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            newComponents[i][j] = components[i][j] - a.components[i][j];
    }
    return newComponents;
}

// @return double[][]    a copy of the components of the receiver.

public double[][] toComponents ( )
{
    int n = rows();
    int m = columns();
    double[][] answer = new double[ n][m];
    for ( int i = 0; i < n; i++ )
    {
        for ( int j = 0; j < m; j++)
            answer[i][j] = components[i][j];
    }
    return answer;
}

// Returns a string representation of the system.
// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    char[] separator = { '[', ' '};
    int n = rows();
    int m = columns();
    for ( int i = 0; i < n; i++)
    {
        separator[0] = '{';
        for ( int j = 0; j < m; j++)
        {
            sb.append( separator);
            sb.append( components[i][j]);
            separator[0] = ' ';
        }
    sb.append('}');
    sb.append('\n');
    }
    return sb.toString();
}

// @return MatrixAlgebra.Matrix        transpose of the receiver

public Matrix transpose ( )
{
    int n = rows();
    int m = columns();
    double[][] newComponents = new double[m][n];
    for ( int i = 0; i < n; i++)
    {
        for( int j = 0; j < m; j++)
            newComponents[j][i] = components[i][j];
    }
    return new Matrix( newComponents);
}

// @return DhbMatrixAlgebra.SymmetricMatrix    the transposed product
//                                  of the receiver with itself.

public SymmetricMatrix transposedProduct()
{
    return new SymmetricMatrix( transposedProductComponents( this));
}

// @return MatrixAlgebra.Matrix    product of the transpose of the
//                              receiver with the supplied matrix
// @param a MatrixAlgebra.Matrix
// @exception MatrixAlgebra.DhbIllegalDimension If the number of rows
//                          of the receiver are not equal to
//                          the number of rows of the supplied matrix.

public Matrix transposedProduct ( Matrix a) throws DhbIllegalDimension
{
    if ( a.rows() != rows() )
        throw new DhbIllegalDimension(
                    "Operation error: cannot multiply a transposed "
                            +rows()+" by "+columns()
                                +" matrix with a "+a.rows()+" by "
                                            +a.columns()+" matrix");
    return new Matrix( transposedProductComponents( a));
}

// @return double[][]    the components of the product of the
//                              transpose of the receiver
// with the supplied matrix.
// @param a MatrixAlgebra.Matrix

protected double[][] transposedProductComponents ( Matrix a)
{
    int p = this.rows();
    int n = this.columns();
    int m = a.columns();
    double[][] newComponents = new double[n][m];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < m; j++)
        {
            double sum = 0;
            for ( int k = 0; k < p; k++)
                sum += components[k][i] * a.components[k][j];
            newComponents[i][j] = sum;
        }
    }
    return newComponents;
}
}
```

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 that can be solved for the three 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[5]. 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** (see Section 8.5).

Computing the inverse of a matrix in the general case is numerically quite demanding (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

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

8.2.1 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

5. It is possible to solve a system of linear equations defined with a nonsquare matrix using a 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.

on the index, calculating each component using all components with a higher index.

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

8.2.2 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 in the form of a single matrix **S** defined as follows:

$$\mathbf{S} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} & y_1 \\ a_{21} & a_{22} & \ldots & a_{2n} & y_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} & y_n \end{pmatrix}. \tag{8.28}$$

Among all transformations leaving the solution of the system represented by the matrix **S** invariant, there are two transformations that help to obtain a system corresponding to a 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[6] 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

$$\mathbf{S}' = \begin{pmatrix} a'_{11} & a'_{12} & \ldots & a'_{1n} & y'_1 \\ 0 & a'_{22} & \ldots & a'_{2n} & y'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & a'_{n2} & \ldots & a'_{nn} & y'_n \end{pmatrix}. \tag{8.29}$$

This step is called *pivoting* 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:

6. In such linear combination, the coefficient of the replaced row must not be zero.

$$\mathbf{S}^{(n)} = \begin{pmatrix} a_{11}^{(n)} & a_{12}^{(n)} & a_{13}^{(n)} & \dots & a_{1n}^{(n)} & y_1^{(n)} \\ 0 & a_{22}^{(n)} & a_{23}^{(n)} & \dots & a_{2n}^{(n)} & y_2^{(n)} \\ 0 & 0 & a_{33}^{(n)} & \dots & a_{3n}^{(n)} & y_3^{(n)} \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn}^{(n)} & y_n^{(n)} \end{pmatrix}. \tag{8.30}$$

This algorithm is called *Gaussian elimination*. Gaussian elimination will fail if we are unable to find a row with a nonzero 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 $\mathbf{T}$ 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.

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.

8.2.3 Fine Points

A efficient way to avoid a null pivot is to look systematically 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 $k = i, \dots, n$ if such row exists. The systematic search for the largest pivot ensures optimal numerical precision [Phillips & Taylor].

The reader will notice that all operations can be performed in place since the original matrix $\mathbf{S}$ is not needed to compute the final result.

Finally, it should be noted that the pivoting step can be performed on several vectors $\mathbf{y}$ 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 $\mathbf{S}$ with as many columns as there are additional constant vectors as follows:

$$\mathbf{S} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} & y_{11} & \dots & y_{m1} \\ a_{21} & a_{22} & \dots & a_{2n} & y_{12} & \dots & y_{m2} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} & y_{1n} & \dots & y_{mn} \end{pmatrix}. \tag{8.31}$$

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.

8.2.4 Linear Equations—General 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 retrieval 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.

8.2.5 Linear Equations—Smalltalk Implementation

Listing 8.6 shows the class `DhbLinearEquationSystem` implementing Gaussian elimination in Smalltalk.

To solve the system of equations 8.22 using Gaussian elimination, one needs to write to evaluate the expression in Code Example 8.5.

Code Example 8.5

```
( DhbLinearEquationSystem 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 `DhbLinearEquationSystem` 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. In this case, the creation method differs in that the two constant vectors are supplied in an array column by column. Similarly, the two solutions must be fetched one after the other.

Code Example 8.6

```
| s sol1 sol2|
s := DhbLinearEquationSystem 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.
```

The class method `equations:constants:` class `DhbLinearEquationSystem` allows creation of 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 nonexistent solution.

Listing 8.6

Smalltalk implementation of a system of linear equations

Class	`DhbLinearEquationSystem`
Subclass of	`Object`
Instance variable names:	`rows solutionsr`
Pool dictionaries:	`SystemExceptions`

Class Methods

equations: `anArrayOfArrays` **constant:** `anArray`

```
OfArrays constant: anArray
    ^self new initialize: anArrayOfArrays constants: (Array with:
                                                              anArray)
```

equations: `anArrayOfArrays` **constants:** `anArrayOfConstantArrays`

```
^self new initialize: anArrayOfArrays constants:
                                          anArrayOfConstantArrays
```

Instance methods

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.
```

initialize: `anArrayOfArrays` **constants:** `anArrayOfConstantArrays`

```
| n |
n := 0.
rows := anArrayOfArrays collect: [ :each | n := n + 1. each,
             ( anArrayOfConstantArrays collect: [ :c | c at: n])].
^self
```

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
```

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)).
        ].
    ].
```

pivotStepAt: anInteger

```
self swapRow: anInteger withRow: ( self largestPivotFrom: anInteger);
       pivotAt: anInteger.
```

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: $).
    ].
```

solution

```
^self solutionAt: 1
```

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
```

solve

```
1 to: rows size do: [ :n | self pivotStepAt: n].
solutions := Array new: ( (rows at: 1) size - rows size).
```

swapRow: `anInteger1` **withRow:** `anInteger2`

```
| swappedRow |
anInteger1 = anInteger2
    ifFalse:[ swappedRow := rows at: anInteger1.
                  rows at: anInteger1 put: ( rows at:
                                                 anInteger2).
                  rows at: anInteger2 put: swappedRow.
             ].
```

8.2.6 Linear Equations—Java Implementation

Listing 8.7 shows the class `LinearEquations` implementing Gaussian elimination in Java.

To solve the system of equations 8.22, one needs to write to evaluate the expression given in Code Example 8.7.

Code Example 8.7

```
double[][] s = {{3,2,4}, {2, -5, -1},{1,-2,2}}};
double[] c = { 16,6,10};
try{
   LinearEquations system = new LinearEquations( s, c);
   double[] solution = system.solution();
   ⋮  <processing of the solution>
} catch(DhbIllegalDimension e){}
```

The first two lines define the components of the components of the system's matrix (**A**) and the components of the constant vector (**y**). The `try...catch` block delineates the solution of the system proper. The first line of the block creates an instance of the class `LinearEquations` by feeding the coefficients of the equations, row by row, and the constant vector. The second line is a call to the method `solution` retrieving the solution of the system.

Solving the same system with an additional constant vector requires an additional line of code, as given in Code Example 8.8. In this case, the creation method differs in that the two constant vectors are supplied in an array column by column. Similarly, the two solutions must be fetched one after the other.

Code Example 8.8

```
double[][] s = {{3,2,4}, {2, -5, -1},{1,-2,2}}};
double[][] c = {{16,6,10}, {7,10,9}};
try{
   LinearEquations system = new LinearEquations( s, c);
   DhbVector solution1 = system.solution(1);
   DhbVector solution2 = system.solution(2);
   ⋮  <processing of the solutions>
} catch(DhbIllegalDimension e){}
```

The class `LinearEquationSystem` has the following instance variables:

`components` An array whose elements contain the rows of the matrix **S**

`solutions` A vector (an instance of the class `DhbVector`) whose elements contain the solutions of the system corresponding to each constant vector.

The class `LinearEquations` has three constructor methods. One uses the components of the matrix and the components of single constant vectors. However, the reader should be reminded that LUP decomposition is more effective in this case. The next uses an arbitrary number of constant vectors. Finally, a constructor with matrix and vector is supplied since the system's matrix and constant vector are often obtained after matrix and vector computations.

The solution to the system is always returned as a mathematical vector—that is, an object of class `DhbVector`. If the solutions were kept as an array, an additional instance variable would be required to flag whether a solution was computed. Individual components, if needed, can be extracted from the vector using the method `toComponents`.

There are two methods named `solution`. The one with an integer argument 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 one without argument is a convenience method to retrieve the solution of a system with a single constant vector.

The method `pivotingStep` first swaps rows to bring a pivot having the largest absolute value in place and then invokes the method `pivot` to perform the actual pivoting.

If the system does not have a solution—that is, if the system's matrix is singular—an `ArithmeticException` is thrown. This exception must be caught by the calling object.

Listing 8.7 Java implementation of a system of linear equations

```
package DhbMatrixAlgebra;

// Class representing a system of linear equations.

// @author Didier H. Besset

public class LinearEquations
{

// components is a matrix build from the system's matrix and
// the constant vector

    private double[][] rows;

// Array containing the solution vectors.

    private DhbVector[] solutions;


// Construct a system of linear equation Ax = y1, y2,....
// @param m double[][]
// @param c double[][]
// @exception DhbMatrixAlgebra.DhbIllegalDimension
//                 if the system's matrix is not square
//                 if constant dimension does not match
//                         that of the matrix

public LinearEquations ( double[][] m, double[][] c)
                                            throws DhbIllegalDimension
{
    int n = m.length;
    if ( m[0].length != n )
        throw new DhbIllegalDimension("Illegal system: a"+n+" by "
                    +m[0].length+" matrix is not a square matrix");
    if ( c[0].length != n )
        throw new DhbIllegalDimension("Illegal system: a "+n+" by "+n
                            +" matrix cannot build a system with a "
                                +c[0].length+"-dimensional vector");
    rows = new double[n][n+c.length];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            rows[i][j] = m[i][j];
        for ( int j = 0; j < c.length; j++)
        rows[i][n+j] = c[j][i];
    }
}

// Construct a system of linear equation Ax = y.
// @param m double[][]       components of the system's matrix
// @param c double[]    components of the constant vector
// @exception DhbMatrixAlgebra.DhbIllegalDimension
//                       if the system's matrix is not square
//                       if constant dimension does not match
//                              that of the matrix

public LinearEquations ( double[][] m, double[] c)
                                            throws DhbIllegalDimension
{
    int n = m.length;
    if ( m[0].length != n )
        throw new DhbIllegalDimension("Illegal system: a"+n+" by "
                    +m[0].length+" matrix is not a square matrix");
    if ( c.length != n )
        throw new DhbIllegalDimension("Illegal system: a "+n+" by "+n
                            +" matrix cannot build a system with a "
                                    +c.length+"-dimensional vector");
    rows = new double[n][n+1];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            rows[i][j] = m[i][j];
        rows[i][n] = c[i];
    }
}

// Construct a system of linear equation Ax = y.
// @param a MatrixAlgebra.Matrix    matrix A
// @param y MatrixAlgebra.DhbVector    vector y
// @exception MatrixAlgebra.DhbIllegalDimension
//                       if the system's matrix is not square
//                       if vector dimension does not match
//                              that of the matrix

public LinearEquations ( Matrix a, DhbVector y)
```

```
                                        throws DhbIllegalDimension
{
    this( a.components, y.components);
}

// Computes the solution for constant vector p applying
// backsubstitution.
// @param p int
// @exception java.lang.ArithmeticException if one diagonal element
//                          of the triangle matrix is zero.

private void backSubstitution ( int p) throws ArithmeticException
{
    int n = rows.length;
    double [] answer = new double[n];
    double x;
    for ( int i = n - 1; i >= 0; i--)
    {
        x = rows[i][n+p];
        for ( int j = i + 1; j < n; j++)
            x -= answer[j] * rows[i][j];
        answer[i] = x / rows[i][i];
    }
    solutions[p] = new DhbVector( answer);
    return;
}

// Finds the position of the largest pivot at step p.
// @return int
// @param p int    step of pivoting.

private int largestPivot ( int p)
{
    double pivot = Math.abs( rows[p][p]);
    int answer = p;
    double x;
    for ( int i = p + 1; i < rows.length; i++)
    {
        x = Math.abs(rows[i][p]);
        if ( x > pivot )
        {
            answer = i;
            pivot = x;
        }
    }
    return answer;
}

// Perform pivot operation at location p.
// @param p int
// @exception java.lang.ArithmeticException if the pivot element
//                                           is zero.

private void pivot ( int p) throws ArithmeticException
{
    double inversePivot = 1 / rows[p][p];
    double r;
    int n = rows.length;
    int m = rows[0].length;
    for ( int i = p + 1; i < n; i++)
    {
        r = inversePivot * rows[i][p];
        for ( int j = p; j < m; j++)
            rows[i][j] -= rows[p][j] * r;
    }
    return;
}

// Perform optimum pivot operation at location p.
// @param p int

private void pivotingStep ( int p)
{
    swapRows( p, largestPivot( p));
    pivot(p);
    return;
}

// @return DhbVector    solution for the 1st constant vector

public DhbVector solution ( ) throws ArithmeticException
{
    return solution( 0);
}

// Return the vector solution of constants indexed by p.
// @return DHBmatrixAlgebra.DhbVector
// @param p int   index of the constant vector fed into the system.
// @exception java.lang.ArithmeticException
//                              if the system cannot be solved.

public DhbVector solution ( int p) throws ArithmeticException
```

```
{
    if ( solutions == null )
        solve();
    if ( solutions[p] == null )
        backSubstitution( p);
    return solutions[p];
}

// @exception java.lang.ArithmeticException
//                         if the system cannot be solved.

private void solve ( ) throws ArithmeticException
{
    int n = rows.length;
    for ( int i = 0; i < n; i++)
        pivotingStep( i);
    solutions = new DhbVector[rows[0].length-n];
}

// Swaps rows p and q.
// @param p int
// @param q int

private void swapRows ( int p, int q)
{
    if ( p != q)
    {
        double temp;
        int m = rows[p].length;
        for (int j = 0; j < m; j++)
        {
            temp = rows[p][j];
            rows[p][j] = rows[q][j];
            rows[q][j] = temp;
        }
    }
    return;
}

// Returns a string representation of the system.
// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    char[] separator = { '[', ' '};
```

```
        int n = rows.length;
        int m = rows[0].length;
        for ( int i = 0; i < n; i++)
        {
            separator[0] = '(';
            for ( int j = 0; j < n; j++)
            {
                sb.append( separator);
                sb.append( rows[i][j]);
                separator[0] = ',';
            }
            separator[0] = ':';
            for ( int j = n; j < m; j++)
            {
                sb.append( separator);
                sb.append( rows[i][j]);
                separator[0] = ',';
            }
        sb.append(')');
        sb.append('\n');
        }
        return sb.toString();
    }
    }
```

8.3 LUP Decomposition

LUP decomposition is another technique to solve a system of linear equations. It is an alternative to the Gaussian elimination [Cormen *et al.*]. 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 nonsingular square matrix **A** can be decomposed into a product of three square matrices of the same dimension as follows:

$$\mathbf{A} = \mathbf{L} \cdot \mathbf{U} \cdot \mathbf{P}, \tag{8.32}$$

where **L** is a matrix whose components located above the diagonal are zero (lower triangular matrix), **U** is a matrix whose components located below the diagonal are zero (upper triangular matrix), and **P** is a permutation matrix. The decomposition of equation 8.32 is nonunique. One can select a unique decomposition by requiring that all diagonal elements of the matrix **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 **v** and **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 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

$$\mathbf{A}' - \frac{\mathbf{v}\otimes\mathbf{w}}{a_{11}} = \mathbf{L}' \cdot \mathbf{U}'. \tag{8.35}$$

Then we have

$$\begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{I}_{n-1} \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{L}' \cdot \mathbf{U}' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{I}_{n-1} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ 0 & \mathbf{L}' \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{U}' \end{pmatrix}$$

$$*[3ex] = \begin{pmatrix} 1 & 0 \\ \frac{\mathbf{v}}{a_{11}} & \mathbf{L}' \end{pmatrix} \cdot \begin{pmatrix} a_{11} & \mathbf{w}^{\mathrm{T}} \\ 0 & \mathbf{U}' \end{pmatrix}. \tag{8.36}$$

The equality of equation 8.36 can be verified by carrying the multiplication with matrix components. The second line of equation 8.3 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 (equation 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 nonzero 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 (see the similar discussion in section 8.2).

Now that we have proven that an LUP decomposition exists for any nonsingular 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.

8.3.1 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 8.3, we can see that all operations can be performed within a matrix of the same size as the original matrix[7].

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 (see 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 purposes.[8]

The instance variable `permutation` is set to undefined (`nil` in Smalltlak, `null` in Java) at initialization time by default. It is used to check whether the decomposition has already been made.

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.

8.3.2 LUP Decomposition—Smalltalk Implementation

Listing 8.8 shows the methods of the class `DhbLUPDecomposition` implementing LUP decomposition in Smalltalk.

To solve the system of equation 8.22 using LUP decomposition, one needs to write to evaluate the expression given in Code Example 8.9.

Code Example 8.9

```
( DhbLUPDecomposition equations: #( (3 2 4) (2 -5 -1) (1 -2 2)) )
          solve: #(16 6 10).
```

7. 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.

8. 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.

This expression has been written on two lines to delineate the various steps. The first line creates an instance of the class `DhbLUPDecomposition` by giving the coefficients of the equations, row by row. 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 to storing the LUP decomposition in a variable (see Code Example 8.10).

Code Example 8.10

```
| s sol1 sol2|
s := DhbLUPDecomposition 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 default creation class method `new` has been overloaded to prevent creating an object without initialized instance variables. 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 1. Copying the coefficients is necessary since the storage is reused during the decomposition steps. In addition, some Smalltalk implementations protect constants such as the one used in the prior code examples. In this latter case, copying is necessary to prevent a read-only exception.

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 (see Sections ??? and ??? 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 8.8 Smalltalk implementation of the LUP decomposition

Class	`DhbLUPDecomposition`
Subclass of	`Object`
Instance variable names:	`rows permutation parity`

Class Methods

direct: `anArrayOfArrays`

```
^self new initialize: anArrayOfArrays.
```

equations: `anArrayOfArrays`

```
^self new initialize: ( anArrayOfArrays collect: [ :each |
                                            each deepCopy]).
```

Instance methods

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
```

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.
    ].
```

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
```

initialize: anArrayOfArrays

```
    rows := anArrayOfArrays.
    parity := 1.
    ^self
```

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
```

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)).
                     ]
             ].
```

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: $).
        ].
```

protectedDecomposition

```
[ self decompose] when: ExAll do: [ :signal | permutation := 0.
                                              signal exitWith: nil].
```

solve: `anArrayOrVector`

```
permutation isNil
    ifTrue: [ self protectedDecomposition].
^permutation = 0
    ifTrue: [ nil]
    ifFalse:[ self backwardSubstitution: ( self
                            forwardSubstitution: anArrayOrVector)]
```

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.
            ].
```

8.3.3 LUP Decomposition—Java Implementation

Listing 8.9 shows the methods of the class `LUPDecomposition` implementing LUP decomposition in Java.

To solve the system of equations 8.22, one needs to write to evaluate the expression as given in Code Example 8.11.

Code Example 8.11

```
double[][] s = {{3,2,4}, {2, -5, -1},{1,-2,2}}};
double[] c = { 16,6,10};
try{
   LUPDecomposition system = new LUPDecomposition( s);
   double[] solution = system.solve(c);
   ⋮  <processing of the solution>
} catch(DhbIllegalDimension e){}
```

The first two lines define the components of the system's matrix (**A**) and the components of the constant vector (**y**). The `try...catch` block delineates the solution of the system proper. The first line of the block creates an instance of the class `LUPDecomposition` using the constructor method taking a double dimensional array as argument. 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 with an additional constant vector requires a couple additional lines of code (see Code Example 8.12).

Code Example 8.12

```
double[][] s = {{3,2,4}, {2, -5, -1},{1,-2,2}}};
double[] c1 = {16,6,10};
double[] c2 = {7,10,9};
try{
   LUPDecomposition system = new LUPDecomposition( s, c);
   double[] solution1 = system.solve(c1);
   double[] solution2 = system.solve(c2);
   ⋮  <processing of the solution>
} catch(DhbIllegalDimension e){}
```

In this case, the second solution is fetched with a second call to the method `solve`. Of course, LUP decomposition is only performed once—namely, when the first solution is fetched. When the second solution is retrieved, only forward and backward substitution are performed.

The class has three constructor methods. The first constructor takes a two dimensional array of doubles as argument. The second constructor takes a matrix as argument. The third constructor takes a symmetrical matrix as argument. When a matrix is supplied, the components of the matrix are copied into the instance variable `rows`. Copying the matrix is necessary since the storage is reused during the decomposition steps. The first two constructor methods throw the exception `DhbIllegalDimension` since LUP decomposition is only defined for square matrix. A symmetrical matrix being always square, the exception is not needed for the third constructor method.

In case of a singular matrix, the method `pivot` throws an `ArithmeticException`. When this happens, the instance variable `permutation` is set to an array of zero length to flag the singular case. This is to prevent the failing decomposition from being attempted more than one time. Then, any subsequent calls to the method `solve` throws an `ArithmeticException`.

Listing 8.9 Java implementation of the LUP decomposition

```
package DhbMatrixAlgebra;

// Lower Upper Permutation (LUP) decomposition

// @author Didier H. Besset

public class LUPDecomposition
{

// Rows of the system

    private double[][] rows;

// Permutation

    private int[] permutation = null;

// Permutation's parity

    private int parity = 1;

// Constructor method
// @param components double[][]
// @exception DhbMatrixAlgebra.DhbIllegalDimension
//                          the supplied matrix is not square

public LUPDecomposition ( double[][]components)
                                            throws DhbIllegalDimension
{
    int n = components.length;
    if ( components[0].length != n )
        throw new DhbIllegalDimension("Illegal system: a"+n+" by "
            +components[0].length+" matrix is not a square matrix");
    rows = components;
    initialize();
}
```

```
// Constructor method.
// @param m DhbMatrixAlgebra.Matrix
// @exception DhbMatrixAlgebra.DhbIllegalDimension
//                         the supplied matrix is not square

public LUPDecomposition ( Matrix m) throws DhbIllegalDimension
{
    if ( !m.isSquare() )
        throw new DhbIllegalDimension(
                        "Supplied matrix is not a square matrix");
    initialize( m.components);
}

// Constructor method.
// @param m DhbMatrixAlgebra.DhbSymmetricMatrix

public LUPDecomposition ( SymmetricMatrix m)
{
    initialize( m.components);
}

// @return double[]
// @param xTilde double[]

private double[] backwardSubstitution( double[] xTilde)
{
    int n = rows.length;
    double[] answer = new double[n];
    for ( int i = n - 1; i >= 0; i--)
    {
        answer[i] = xTilde[i];
        for ( int j = i + 1; j < n; j++)
            answer[i] -= rows[i][j] * answer[j];
        answer[i] /= rows[i][i];
    }
    return answer;
}
private void decompose()
{
    int n = rows.length;
    permutation = new int[n];
    for ( int i = 0; i < n; i++ )
        permutation[i] = i;
    parity = 1;
    try {
            for ( int i = 0; i < n; i++)
```

```
                {
                    swapRows( i, largestPivot( i));
                    pivot( i);
                }
            } catch ( ArithmeticException e) { parity = 0;};
    }
```

// @return boolean true if decomposition was done already

```
    private boolean decomposed()
    {
        if ( parity == 1 && permutation == null )
            decompose();
        return parity != 0;
    }
```

// @return double[]
// @param c double[]

```
    public double determinant()
    {
        if ( !decomposed() )
            return Double.NaN;
        double determinant = parity;
        for ( int i = 0; i < rows.length; i++ )
            determinant *= rows[i][i];
        return determinant;
    }
```

// @return double[]
// @param c double[]

```
    private double[] forwardSubstitution( double[] c)
    {
        int n = rows.length;
        double[] answer = new double[n];
        for ( int i = 0; i < n; i++)
        {
            answer[i] = c[permutation[i]];
            for ( int j = 0; j <= i - 1; j++)
                answer[i] -= rows[i][j] * answer[j];
        }
        return answer;
    }
    private void initialize ()
    {
```

```
    permutation= null;
    parity = 1;
}
```

// @param components double[][] components obtained from constructor methods.

```
private void initialize ( double[][] components)
{
    int n = components.length;
    rows = new double[n][n];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            rows[i][j] = components[i][j];
    }
    initialize();
}
```

// @return double[]
// @param c double[]

```
public double[][] inverseMatrixComponents()
{
    if ( !decomposed() )
        return null;
    int n = rows.length;
    double[][] inverseRows = new double[n][n];
    double[] column = new double[n];
    for ( int i = 0; i < n; i ++)
    {
        for ( int j = 0; j < n; j++ )
            column[j] = 0;
        column[i] = 1;
        column = solve( column);
        for ( int j = 0; j < n; j++ )
            inverseRows[i][j] = column[j];
    }
    return inverseRows;
}
```

// @return int
// @param k int

```
private int largestPivot(int k)
{
    double maximum = Math.abs( rows[k][k]);
    double abs;
    int index = k;
    for ( int i = k + 1; i < rows.length; i++)
    {
        abs = Math.abs( rows[i][k]);
        if ( abs > maximum )
        {
            maximum = abs;
            index = i;
        }
    }
    return index;
}

// @param k int

private void pivot( int k)
{
    double inversePivot = 1 / rows[k][k];
    int k1 = k + 1;
    int n = rows.length;
    for ( int i = k1; i < n; i++)
    {
        rows[i][k] *= inversePivot;
        for ( int j = k1; j < n; j++)
            rows[i][j] -= rows[i][k] * rows[k][j];
    }
}

// @return double[]
// @param c double[]

public double[] solve( double[] c)
{
    return decomposed()
                    ? backwardSubstitution( forwardSubstitution( c))
                    : null;
}

// @return double[]
// @param c double[]

public DhbVector solve( DhbVector c)
{
    double[] components = solve( c.components);
    if ( components == null )
        return null;
    return components == null ? null : new DhbVector( components);
}

// @param i int
// @param k int

private void swapRows( int i, int k)
{
    if ( i != k )
    {
        double temp;
        for ( int j = 0; j < rows.length; j++ )
        {
            temp = rows[i][j];
            rows[i][j] = rows[k][j];
            rows[k][j] = temp;
        }
        int nTemp;
        nTemp = permutation[i];
        permutation[i] = permutation[k];
        permutation[k] = nTemp;
        parity = -parity;
    }
}

// Make sure the supplied matrix components are those of
// a symmetric matrix
// @param components double

public static void symmetrizeComponents( double[][] components)
{
    for ( int i = 0; i < components.length; i++ )
        {
            for ( int j = i + 1; j < components.length; j++ )
            {
                components[i][j] += components[j][i];
                components[i][j] *= 0.5;
                components[j][i] = components[i][j];
            }
        }
}

// Returns a String that represents the value of this object.
// @return a string representation of the receiver
```

```
public String toString()
{
    StringBuffer sb = new StringBuffer();
    char[] separator = { '[', ' '};
    int n = rows.length;
    for ( int i = 0; i < n; i++)
    {
        separator[0] = '{';
        for ( int j = 0; j < n; j++)
        {
            sb.append( separator);
            sb.append( rows[i][j]);
            separator[0] = ' ';
        }
        sb.append('}');
        sb.append('\n');
    }
    if ( permutation != null )
    {
        sb.append( parity == 1 ? '+' : '-');
        sb.append("( " + permutation[0]);
        for ( int i = 1; i < n; i++)
            sb.append(", " + permutation[i]);
        sb.append(')');
        sb.append('\n');
    }
    return sb.toString();
}
}
```

8.4 Computing the Determinant of a Matrix

The determinant of a matrix is defined as

$$\det \mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{vmatrix} = \sum_{\pi} \operatorname{sign}(\pi)\, a_{1\pi(1)} a_{2\pi(1)} \cdots a_{n\pi(1)}, \qquad (8.40)$$

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 these 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.

8.4.1 Computing the Determinant of Matrix—General Implementation

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.

8.4.2 Computing the Determinant of Matrix—Smalltalk Implementation

Listing 8.10 shows the methods of classes `DhbMatrix` and `DhbLUPDecomposition` needed to compute a matrix determinant.

Listing 8.10 Smalltalk methods to compute a matrix determinant

Class	`DhbMatrix`
Subclass of	`Object`
Instance variable names:	`rows lupDecomposition`

Instance Methods

determinant

```
    ^self lupDecomposition determinant
```

Class	`DhbLUPDecomposition`
Subclass of	`Object`
Instance variable names:	`rows permutation parity`

Instance Methods

determinant

```
| n |
permutation isNil
    ifTrue: [ self protectedDecomposition].
permutation = 0
    ifTrue: [ ^0].
```

8.4.3 Computing the Determinant of Matrix—Java Implementation

The code computing the determinant of a matrix consists of the method `determinant` of the class `Matrix` (see Listing 8.2) and the method `determinant` of the class `LUPDecomposition` (see Listing 8.9).

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 (see Section 8.2) or LUP decomposition (see 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 column of the matrix $\mathbf{A}^{-1}$. Specifically, $\mathbf{c}_i$, the ith 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 ith column of the identity matrix—that is, a vector with zero components except for the ith component whose value is 1.

For large matrices, however, using LUP decomposition becomes quite slow. A cleverer algorithm for symmetrical matrices is given in [Cormen *et al.*] with no name.

In this book, we shall refer to this algorithm as the *CLR algorithm* (acronym of the authors of [Cormen *et al.*]).

Let **A** be a symmetrical 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 symmetrical 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 symmetrical 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 symmetrical 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 matrices **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. [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[9]. 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 1,024 by 1,024 would require 8 megabytes of storage.

8.5.1 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

9. I thank Thomas Cormen for enlightening E-mails on this subject.

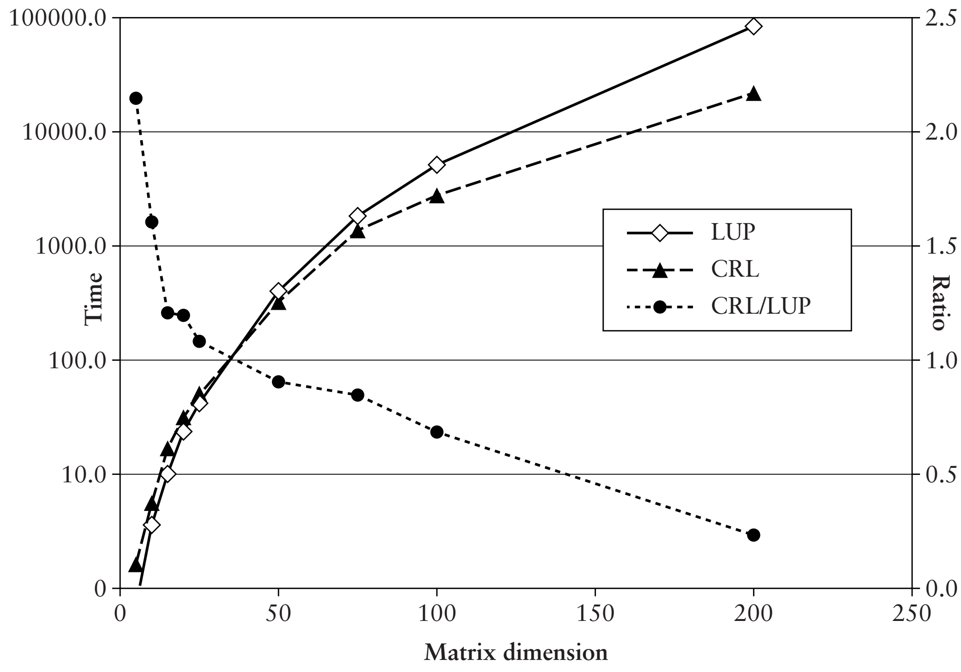

FIG. 8.2 Comparison of inversion time for nonsymmetrical matrices

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 8.2 shows the time needed to inverse a nonsymmetrical 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.

In Figure 8.2 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 advised 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 symmetrical matrices. In [Cormen *et al.*] 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 symmetrical 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 (see Section 8.5.4).

8.5.2 Matrix Inversion—Smalltalk Implementation

Listing 8.11 shows the complete implementation in Smalltalk. It contains additional methods for the classes `DhbMatrix` and `DhbSymmetricMatrix`.

For symmetrical 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 8.1.1.

For nonsymmetrical matrices the method `inverse` first tests whether the matrix is square. 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 that must be handled by the calling method.

Listing 8.11

Smalltalk implementation of matrix inversion

Class	`DhbSymmetricMatrix`
Subclass of	`DhbMatrix`

Class Methods

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
```

lupCRLCriticalDimension

```
^36
```

Instance Methods

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
```

inverse

```
^(rows size < self class lupCRLCriticalDimension or:
                                    [lupDecomposition notNil])
        ifTrue: [self lupInverse]
        ifFalse: [self crlInverse]
```

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]))
```

Class	`DhbMatrix`
Subclass of	`Object` itemInstance variable names: `rows lupDecomposition`

Class Methods

lupCRLCriticalDimension

```
^40
```

Instance Methods

inverse

```
^self isSquare
    ifTrue: [self lupInverse]
    ifFalse: [self squared inverse * self transpose]
```

largestPowerOf2SmallerThan: `anInteger`

```
| m m2|
m := 2.
[ m2 := m * 2.
  m2 < anInteger] whileTrue:[ m := m2].
^m
```

lupInverse

```
^self class rows: self lupDecomposition inverseMatrixComponents
```

8.5.3 Matrix Inversion—Java Implementation

Listing 8.12 shows the implementation of the class `SymmetricMatrix` in Java.

Addition and product of two symmetrical matrices yield a symmetrical matrix. Thus, the corresponding methods have been redefined for that class.

Because the method to create a symmetrical matrix from components is used frequently in the rest of the package, no check is made for the validity of the components. Of course, this method is declared as protected to prevent classes from outside the package from using it. Such classes must use the static method `fromComponents` to create a symmetrical matrix with supplied components, which throws the exception `DhbNonSymmetricComponents` if the supplied components are not those of a symmetrical matrix. The code for this exception is elementary and is not shown here. It is left as an exercise for the reader.

The method `inverse` for symmetrical matrices first tests whether the dimension of the matrix is below a given threshold—defined by the static variable `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 a direct transcription of equations 8.43 to 8.45 using the methods defined in Section 8.1.2.

Listing 8.12 Java implementation of the class

```
SymmetricMatrix

package DhbMatrixAlgebra;

// Symmetric matrix

// @author Didier H. Besset

public class SymmetricMatrix extends Matrix {

    private static int lupCRLCriticalDimension = 36;

// Creates a symmetric matrix with given components.
// @param a double[][]

protected SymmetricMatrix(double[][] a)
{
    super(a);
}

// @param n int
// @exception java.lang.NegativeArraySizeException if n ¡= 0

public SymmetricMatrix (int n ) throws NegativeArraySizeException
{
    super( n, n);
}
```

```
// Constructor method.
// @param n int
// @param m int
// @exception java.lang.NegativeArraySizeException if n,m ¡= 0

public SymmetricMatrix(int n, int m) throws NegativeArraySizeException {
    super(n, m);
}

// @return SymmetricMatrix    sum of the matrix with the supplied
matrix.
// @param a DhbMatrix
// @exception DhbIllegalDimension if the supplied matrix does not
//                                 have the same dimensions.

public SymmetricMatrix add ( SymmetricMatrix a)
                                            throws DhbIllegalDimension
{
    return new SymmetricMatrix( addComponents( a));
}

// Answers the inverse of the receiver computed via the CRL algorithm.
// @return DhbMatrixAlgebra.SymmetricMatrix
// @exception java.lang.ArithmeticException if the matrix is singular.

private SymmetricMatrix crlInverse ( ) throws ArithmeticException
{
    if ( rows() == 1)
        return inverse1By1();
    else if ( rows() == 2)
        return inverse2By2();
    Matrix[] splitMatrices = split();
    SymmetricMatrix b1 = (SymmetricMatrix) splitMatrices[0].inverse();
    Matrix cb1 = splitMatrices[2].secureProduct( b1);
    SymmetricMatrix cb1cT = new SymmetricMatrix(
                cb1.productWithTransposedComponents(splitMatrices[2]));
    splitMatrices[1] = ( (SymmetricMatrix)
                    splitMatrices[1]).secureSubtract( cb1cT).inverse();
    splitMatrices[2] = splitMatrices[1].secureProduct( cb1);
    splitMatrices[0] = b1.secureAdd( new SymmetricMatrix(
                cb1.transposedProductComponents( splitMatrices[2])));
    return SymmetricMatrix.join( splitMatrices);
}

// @return DhbMatrixAlgebra.SymmetricMatrix
// @param    comp double[][]    components of the matrix
// @exception DhbMatrixAlgebra.DhbIllegalDimension
//            The supplied components are not those of a square matrix.
// @exception DhbMatrixAlgebra.DhbNonSymmetricComponents
//            The supplied components are not symmetric.

public static SymmetricMatrix fromComponents ( double[][] comp)
                throws DhbIllegalDimension, DhbNonSymmetricComponents
{
    if ( comp.length != comp[0].length)
        throw new DhbIllegalDimension( "Non symmetric components: a "
                                    +comp.length+" by "+comp[0].length
                                    +" matrix cannot be symmetric");
    for ( int i = 0; i < comp.length; i++)
    {
        for ( int j = 0; j < i; j++)
        {
            if ( comp[i][j] != comp[j][i])
                throw new DhbNonSymmetricComponents(
                    "Non symmetric components: a["+i+"]["+j
                                    +"]= "+comp[i][j]+", a["+j+"]["
                                                +i+"]= "+comp[j][i]);
        }
    }
    return new SymmetricMatrix( comp);
}

// @return SymmetricMatrix    an identity matrix of size n
// @param n int

public static SymmetricMatrix identityMatrix ( int n)
{
    double[][] a = new double[n][n];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++) a[i][j] = 0;
        a[i][i] = 1;
    }
    return new SymmetricMatrix( a);
}

// @return DhbMatrix        inverse of the receiver.
// @exception java.lang.ArithmeticException if the receiver is
//                                        a singular matrix.

public Matrix inverse ( ) throws ArithmeticException
{
    return rows() < lupCRLCriticalDimension
                    ? new SymmetricMatrix(
            (new LUPDecomposition( this)).inverseMatrixComponents())
                    : crlInverse( );
}
```

// *Compute the inverse of the receiver in the case of a 1 by 1 matrix.*
// *Internal use only: no check is made.*
// *@return DhbMatrixAlgebra.SymmetricMatrix*

```
private SymmetricMatrix inverse1By1 ( )
{
    double[][] newComponents = new double[1][1];
    newComponents[0][0] = 1 / components[0][0];
    return new SymmetricMatrix( newComponents);
}
```

// *Compute the inverse of the receiver in the case of a 2 by 2 matrix.*
// *Internal use only: no check is made.*
// *@return DhbMatrixAlgebra.SymmetricMatrix*

```
private SymmetricMatrix inverse2By2 ( )
{
    double[][] newComponents = new double[2][2];
    double inverseDeterminant = 1 / ( components[0][0] * components[1][1]
                            - components[0][1] * components[1][0]);
    newComponents[0][0] = inverseDeterminant * components[1][1];
    newComponents[1][1] = inverseDeterminant * components[0][0];
    newComponents[0][1] = newComponents[1][0] =
                        -inverseDeterminant * components[1][0];
    return new SymmetricMatrix( newComponents);
}
```

// *Build a matrix from 3 parts (inverse of split).*
// *@return DhbMatrixAlgebra.SymmetricMatrix*
// *@param a DhbMatrixAlgebra.Matrix[]*

```
private static SymmetricMatrix join ( Matrix[] a)
{
    int p = a[0].rows();
    int n = p + a[1].rows();
    double[][] newComponents = new double[n][n];
    for ( int i = 0; i < p; i++)
    {
        for ( int j = 0; j < p; j++)
            newComponents[i][j] = a[0].components[i][j];
        for ( int j = p; j < n; j++)
            newComponents[i][j] = newComponents[j][i] =
                                          -a[2].components[j-p][i];
    }
    for ( int i = p; i < n; i++)
    {
        for ( int j = p; j < n; j++)
            newComponents[i][j] = a[1].components[i-p][j-p];
    }
    return new SymmetricMatrix( newComponents);
}

// @return int
// @param n int

private int largestPowerOf2SmallerThan ( int n)
{
    int m = 2;
    int m2;
    while (true)
    {
        m2 = 2 * m;
        if ( m2 >= n )
            return m;
        m = m2;
    }
}

// @return DhbMatrixAlgebra.SymmetricMatrix
// @param a double

public Matrix product ( double a)
{
    return new SymmetricMatrix( productComponents( a));
}

// @return Matrix    product of the receiver with the supplied matrix
// @param a Matrix
// @exception DhbIllegalDimension If the number of columns of
// the receivers are not equal to the number of rows
// of the supplied matrix.

public SymmetricMatrix product ( SymmetricMatrix a)
                                 throws DhbIllegalDimension
{
    return new SymmetricMatrix( productComponents( a));
```

```
}

// @return DhbMatrixAlgebra.Matrix    product of the receiver with
//                            the transpose of the supplied matrix
// @param a DhbMatrixAlgebra.Matrix
// @exception DhbMatrixAlgebra.DhbIllegalDimension If the number of
//           columns of the receiver are not equal to the number of
//           columns of the supplied matrix.

public SymmetricMatrix productWithTransposed ( SymmetricMatrix a)
                                        throws DhbIllegalDimension
{
    if ( a.columns() != columns() )
        throw new DhbIllegalDimension(
                    "Operation error: cannot multiply a "
                    +rows()+" by "+columns()
                    +" matrix with the transpose of a "
                    +a.rows()+" by "+a.columns()+" matrix");
    return new SymmetricMatrix( productWithTransposedComponents( a));
}

// Same as add ( SymmetricMatrix a), but without dimension checking.
// @return DhbMatrixAlgebra.SymmetricMatrix
// @param a DhbMatrixAlgebra.SymmetricMatrix

protected SymmetricMatrix secureAdd ( SymmetricMatrix a)
{
    return new SymmetricMatrix( addComponents( a));
}

// Same as product(DhbSymmetricMatrix a), but without dimension checking.
// @return DhbMatrixAlgebra.SymmetricMatrix
// @param a DhbMatrixAlgebra.SymmetricMatrix

protected SymmetricMatrix secureProduct ( SymmetricMatrix a)
{
    return new SymmetricMatrix( productComponents( a));
}

// Same as subtract ( SymmetricMatrix a), but without dimension checking.
// @return DhbMatrixAlgebra.SymmetricMatrix
// @param a DhbMatrixAlgebra.SymmetricMatrix

protected SymmetricMatrix secureSubtract ( SymmetricMatrix a)
{
    return new SymmetricMatrix( subtractComponents( a));
```

```
}

// Divide the receiver into 3 matrices of approximately equal dimension.
// @return DhbMatrixAlgebra.Matrix[]    Array of splitted matrices

private Matrix[] split ( )
{
    int n = rows();
    int p = largestPowerOf2SmallerThan( n);
    int q = n - p;
    double[][] a = new double[p][p];
    double[][] b = new double[q][q];
    double[][] c = new double[q][p];
    for ( int i = 0; i < p; i++)
    {
        for ( int j = 0; j < p; j++)
            a[i][j] = components[i][j];
        for ( int j = p; j < n; j++)
            c[j-p][i] = components[i][j];
    }
    for ( int i = p; i < n; i++)
    {
        for ( int j = p; j < n; j++)
            b[i-p][j-p] = components[i][j];
    }
    Matrix[] answer = new Matrix[3];
    answer[0] = new SymmetricMatrix( a);
    answer[1] = new SymmetricMatrix( b);
    answer[2] = new Matrix( c);
    return answer;
}

// @return DHBmatrixAlgebra.SymmetricMatrix
// @param a DHBmatrixAlgebra.SymmetricMatrix
// @exception DHBmatrixAlgebra.DhbIllegalDimension (from constructor).

public SymmetricMatrix subtract ( SymmetricMatrix a)
                                        throws DhbIllegalDimension
{
    return new SymmetricMatrix( subtractComponents( a));
}

// @return DHBmatrixAlgebra.Matrix       the same matrix

public Matrix transpose()
{
    return this;
}

// @return DhbMatrixAlgebra.SymmetricMatrix    product of the transpose
//                        of the receiver with the supplied matrix
// @param a DhbMatrixAlgebra.SymmetricMatrix
// @exception DhbMatrixAlgebra.DhbIllegalDimension If the number of
//                    rows of the receiver are not equal to
//                    the number of rows of the supplied matrix.

public SymmetricMatrix transposedProduct ( SymmetricMatrix a)
                                            throws DhbIllegalDimension
{
    if ( a.rows() != rows() )
        throw new DhbIllegalDimension(
                "Operation error: cannot multiply a transposed "
                +rows()+" by "+columns()+" matrix with a "+
                a.rows()+" by "+a.columns()+" matrix");
    return new SymmetricMatrix( transposedProductComponents( a));
}
}
```

For nonsymmetrical matrices the method `inverse` (see Listing 8.5) attempts to compute the inverse using LUP decomposition within a `try...catch` block trapping the exception `DhbIllegalDimension`. This kind of exception will occur for any nonsquare matrix. If the exception occurs, the pseudo inverse is computed using equation 8.46.

8.5.4 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, 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 (equation 8.46).

Now is the time to unveil the mystery example of Section 1.3.3 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 **A** be the matrix to be inverted. The matrix **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 *general* data correspond to a matrix whose components were generated by a random number generator (see Section 9.4). They were all comprised between zero and 1. For the *special* data, the matrix **A** is a covariance matrix (see Section ???) obtained by generating 1000 vectors with random components comprised between zero and 1. For method B general data, the inverse of a nonsymmetrical 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 in Figure 1.1.

8.6 Matrix Eigenvalues and Eigenvectors of a Nonsymmetrical Matrix

A nonzero vector **u** is called an *eigenvector* 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 nonsingular matrix of dimension n has n eigenvalues and eigenvectors. Some eigenvalues, however, may be double.[10] 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.[11] One can prove that two eigenvectors of the same matrix, but corresponding to two different eigenvalues, are orthogonal to each other [Bass]. Thus, the eigenvectors of a matrix form a complete set of reference in an 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, I give an algorithm that 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 $\lambda_{\max}$ be the eigenvalue with the largest absolute value. Let us define the following series of

10. Eigenvalues are the roots of a polynomial of degree n. A double eigenvalue has two different eigenvectors.

11. If the vector **u** is an eigenvector of the matrix **M** with eigenvalue λ, so are all vectors $\alpha\mathbf{u}$ for any $\alpha \neq 0$.

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[12] 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 $\lambda = u'_1$, which 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 choose 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

$$\mathbf{M}' = \mathbf{M} \cdot (\mathbf{I} - \mathbf{u}_{\max} \otimes \mathbf{v}_{\max}), \tag{8.51}$$

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 $\lambda_{\max}$.[13] 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 zero. A complete proof of the above can be found in [Bass].

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 one 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.

12. Hint: one must write the vector $\mathbf{u}$ as a linear combination of the eigenvectors of the matrix $\mathbf{M}$. Such a linear combination exists because the eigenvectors of a matrix form a complete system of reference.

13. The transpose of a matrix has the same eigenvalues, but not necessarily the same eigenvectors.

8.6.1 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 Section 4.1. 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
`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 `next-LargestEigenValueFinder` 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.

8.6.2 Finding the Largest Eigenvalue—Smalltalk Implementation

Listing 8.13 shows the Smalltalk implementation of the class `DhbLargestEigenVal-ueFinder`, subclass of the class `DhbIterativeProcess`.

Code Example 8.13 shows how to use the class to find the first two largest eigenvalues of a matrix.

Code Example 8.13

```
| m finder eigenvalue eigenvector nextFinder nextEigenvalue
   nextEigenvector |
m := DhbMatrix rows: #((84 -79 58 55)
                       (-79 84 -55 -58)
                       (58 -55 84 79)
                       (55 -58 79 84)).
finder := DhbLargestEigenValueFinder 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 `DhbLargestEigenValueFinder` is created for this matrix. The iterative process is started as described in Section 4.1.1. Its result is the eigenvalue. The eigenvector is retrieved using an accessor method. Then, a new instance of `DhbLargestEigen-ValueFinder` is obtained from the first one. The next largest eigenvalue and its eigenvector are retrieved from this new instance exactly as before.

Listing 8.13 Smalltalk implementation of the search for the largest eigenvalue

Class	`DhbLargestEigenValueFinder`
Subclass of	`DhbIterativeProcess`
Instance variable names:	`matrix eigenvector transposeEigenvector`

Class Methods

defaultMaximumIterations

```
^100
```

matrix: `aMatrix`

```
^( self new) initialize: aMatrix; yourself
```

matrix: `aMatrix` **precision:** `aNumber`

```
^( self new) initialize: aMatrix; desiredPrecision: aNumber;
                                                   yourself
```

Instance Methods

eigenvalue

```
^result
```

eigenvector

```
^eigenvector * (1 / eigenvector norm)
```

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]
```

initialize: `aMatrix`

```
matrix := aMatrix.
```

initializeIterations

```
eigenvector := DhbVector new: matrix numberOfRows.
eigenvector atAllPut: 1.0.
transposeEigenvector := DhbVector new: eigenvector size.
transposeEigenvector atAllPut: 1.0
```

nextLargestEigenValueFinder

```
| norm |
norm := 1 / (eigenvector * transposeEigenvector).
^self class
    new: matrix * ((DhbSymmetricMatrix identity: eigenvector size)
                - (eigenvector * norm tensorProduct:
                                        transposeEigenvector))
    precision: desiredPrecision
```

8.6.3 Finding the Largest Eigenvalue—Java Implementation

Listing 8.13 shows the Java implementation of the class `LargestEigenValueFinder`, subclass of class `IterativeProcess`.

Code Example 8.14 shows how to use the class to find the first two largest eigenvalues of a matrix.

Code Example 8.14

```
double[][] c = {{84,-79,58,55},{-79,84,-55,-58},
                {58,-55,84,79},{55,-58,79,84}};
Matrix m = new Matrix(c);
LargestEigenValueFinder finder = new LargestEigenValueFinder( m);
finder evaluate().
double eigenvalue = finder.eigenvalue();
DhbVector eigenvector = finder.eigenvector();
LargestEigenValueFinder nextFinder =
finder.nextLargestEigenValueFinder();
nextFinder.evaluate();
double nextEigenvalue = nextFinder.eigenvalue();
nextEigenvector = nextFinder.eigenvector();
```

First the matrix `m` is defined from its components. Then, an instance of the class `LargestEigenValueFinder` is created for this matrix. The iterative process is started as described in Section 4.1.2. The eigenvalue and the corresponding eigenvector are retrieved using accessor methods. Then, a new instance of `LargestEigenValueFinder` is obtained from the first one. The new iterative process is started. The next largest eigenvalue and its eigenvector are retrieved from this new instance exactly as before.

Listing 8.14 Java implementation of the search for the largest eigenvalue

```
package DhbMatrixAlgebra;

import DhbIterations.*;

// Object used to find the largest eigen value and the corresponding
// eigen vector of a matrix by successive approximations.

// @author Didier H. Besset

public class LargestEigenvalueFinder extends IterativeProcess
{

// Eigenvalue

    private double eigenvalue;

// Eigenvector
```

```
    private DhbVector eigenvector;

// Eigenvector of transposed matrix

    private DhbVector transposedEigenvector;

// Matrix.

    private Matrix matrix;


// Constructor method.
// @param prec double
// @param a DhbMatrixAlgebra.Matrix

public LargestEigenvalueFinder ( double prec, Matrix a)
{
    this(a);
    this.setDesiredPrecision ( prec);
}

// Constructor method.
// @param a DhbMatrixAlgebra.Matrix

public LargestEigenvalueFinder ( Matrix a)
{
    matrix = a;
    eigenvalue = Double.NaN;
}

// Returns the eigen value found by the receiver.
// @return double

public double eigenvalue ( )
{
    return eigenvalue;
}

// Returns the normalized eigen vector found by the receiver.
// @return DhbMatrixAlgebra.DhbVector

public DhbVector eigenvector ( )
{
    return eigenvector.product( 1.0 / eigenvector.norm());
}
```

```
// Iterate matrix product in eigenvalue information.

public double evaluateIteration()
{
    double oldEigenvalue = eigenvalue;
    transposedEigenvector =
                    transposedEigenvector.secureProduct( matrix);
    transposedEigenvector = transposedEigenvector.product( 1.0
                        / transposedEigenvector.components[0]);
    eigenvector = matrix.secureProduct( eigenvector);
    eigenvalue = eigenvector.components[0];
    eigenvector = eigenvector.product( 1.0 / eigenvalue);
    return Double.isNaN( oldEigenvalue)
                ? 10 * getDesiredPrecision()
                : Math.abs( eigenvalue - oldEigenvalue);
}

// Set result to undefined.

public void initializeIterations()
{
    eigenvalue = Double.NaN;
    int n = matrix.columns();
    double [] eigenvectorComponents = new double[ n];
    for ( int i = 0; i < n; i++) { eigenvectorComponents [i] = 1.0;}
    eigenvector = new DhbVector( eigenvectorComponents);
    n = matrix.rows();
    eigenvectorComponents = new double[ n];
    for ( int i = 0; i < n; i++) { eigenvectorComponents [i] = 1.0;}
    transposedEigenvector = new DhbVector( eigenvectorComponents);
}

// Returns a finder to find the next largest eigen value of the receiver's matrix.
// @return DhbMatrixAlgebra.LargestEigenvalueFinder

public LargestEigenvalueFinder nextLargestEigenvalueFinder ( )
{
    double norm = 1.0 / eigenvector.secureProduct(
                                    transposedEigenvector);
    DhbVector v1 = eigenvector.product( norm);
    return new LargestEigenvalueFinder( getDesiredPrecision(),
            matrix.secureProduct(SymmetricMatrix.identityMatrix(
                v1.dimension()).secureSubtract(v1.tensorProduct(
                                    transposedEigenvector))));
}
```

```
// Returns a string representation of the receiver.
// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( eigenvalue);
    sb.append(" (");
    sb.append( eigenvector.toString());
    sb.append(')');
    return sb.toString();
}
}
```

8.7 Matrix Eigenvalues and Eigenvectors of a Symmetrical Matrix

In the 19th century, Carl Jacobi discovered an efficient algorithm to find the eigenvalues of a symmetrical matrix. Finding the eigenvalues of a symmetrical matrix is easier since all eigenvalues are real.

In Section 8.6 we have mentioned that the eigenvectors of a matrix are orthogonal. Let $\mathbf{u}^{(1)}, \ldots, \mathbf{u}^{(n)}$ be 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

$$\mathbf{O} = \begin{pmatrix} u_1^{(1)} & u_2^{(1)} & \ldots & u_n^{(1)} \\ u_1^{(2)} & u_2^{(2)} & \ldots & u_n^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ u_1^{(n)} & u_2^{(n)} & \ldots & u_n^{(n)} \end{pmatrix}, \tag{8.52}$$

where $u_i^{(k)}$ is the ith component of the kth eigenvector, is an orthogonal[14] matrix. That is, we have

$$\mathbf{O}^{\mathrm{T}} \cdot \mathbf{O} = \mathbf{I}. \tag{8.53}$$

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:

14. An orthogonal matrix of dimension n is a rotation in the n-dimensional space.

$$\mathbf{O}^{\mathrm{T}} \cdot \mathbf{M} \cdot \mathbf{O} = \begin{pmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \lambda_n \end{pmatrix}, \tag{8.54}$$

where $\lambda_1, \dots, \lambda_n$ 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 a 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 annihilate 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 the following components:

$$\begin{cases} r_{kk}^{(1)} = \cos\vartheta, \\ r_{ll}^{(1)} = \cos\vartheta, \\ r_{kl}^{(1)} = -\sin\vartheta, \\ r_{lk}^{(1)} = \sin\vartheta, \\ r_{ii}^{(1)} = 1 \quad \text{for } i \neq k, l, \\ r_{ij}^{(1)} = 0 \quad \text{for } i \neq j, i \text{ and } j \neq k, l. \end{cases} \tag{8.58}$$

The reader can verify that the matrix $\mathbf{R}_1$ is an orthogonal matrix. The new matrix $\mathbf{M}_1 = \mathbf{R}_1^{\mathrm{T}} \cdot \mathbf{M} \cdot \mathbf{R}_1$ 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 [Press *et al.*]. The solution of equation 8.61 has already been discussed in Section 1.3.4 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:

$$\begin{cases} \cos\vartheta = \frac{1}{\sqrt{t^2+1}}, \\ \sin\vartheta = t\cos\vartheta. \end{cases} \tag{8.63}$$

Let us now introduce the quantities σ and τ defined as

$$\begin{cases} \sigma = \sin\vartheta, \\ \tau = \frac{\sin\vartheta}{1+\cos\vartheta}. \end{cases} \tag{8.64}$$

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 this transformation did not increase the absolute values of the remaining off-diagonal elements of the matrix $\mathbf{M}_1$. Using equations 8.59, we find that the sum of the off-diagonal elements of the matrix $\mathbf{M}_1$ is

$$\sum_{i \neq j} \left(m_{ij}^{(1)} \right)^2 = \sum_{i \neq j} m_{ij}^2 - 2 m_{kl}^2. \tag{8.66}$$

Thus, this sum is always less that the sum of the 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 as follows:

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 $\mathbf{M}_1 = \mathbf{R}_1^{\mathrm{T}} \cdot \mathbf{M} \cdot \mathbf{R}_1$.
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 guarantees 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 earlier can be safely 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 1.3.4, 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.

8.7.1 Jacobi's Algorithm—General Implementation

Jacobi's algorithm is an iterative algorithm. The object implementing Jacobi's algorithm is an instance of the class `JacobiTransform`; it is a subclass of the iterative process discussed in Section 4.1. The instance variables of this class are different in the two language implementations.

When an instance of the class `JacobiTransform` is created, the matrix whose eigenvalues are sought is copied into the matrix **M**. This permits use of 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 symmetrical 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.

8.7.2 Jacobi's Algorithm—Smalltalk Implementation

Listing 8.15 shows the Smalltalk implementation of Jacobi's algorithm.

Code Example 8.15 shows how to use the class to find the eigenvalues and eigenvectors of a symmetrical matrix.

Code Example 8.15

```
| m jacobi eigenvalues eigenvectors |
m := DhbSymmetricMatrix rows: #((84 -79 58 55)
                                (-79 84 -55 -58)
```

```
                                       (58 -55 84 79)
                                       (55 -58 79 84)).
    jacobi := DhbJacobiTransformation 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 `DhbJacobiTransformation` is created for this matrix. The iterative process is started as described in Section 4.1.1. 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 `DhbJacobiTransformation` has two instance variables:

`lowerRows`	An array of array containing the lower part of the matrix
`transform`	The components of the matrix **O**

Since the matrix **M** is symmetrical there is no need to keep all of its components. This not only reduces storage but also speeds up somewhat the algorithm because one needs to transform only 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 symmetrical matrix **O**, whose columns contain the eigenvectors in the same order. Code Example 8.15 shows how to obtain the eigenvectors from the matrix.

Listing 8.15 Smalltalk implementation of Jacobi's algorithm

Class	`DhbJacobiTransformation`
Subclass of	`DhbIterativeProcess`
Instance variable names:	`lowerRows transform`

Class Methods

matrix: `aSymmetricMatrix`

```
    ^super new initialize: aSymmetricMatrix
```

new

```
    ^self error: 'Illegal creation message for this class'
```

Instance Methods

evaluateIteration

```
    | indices |
indices := self largestOffDiagonalIndices.
self transformAt: ( indices at: 1) and: ( indices at: 2).
^precision
```

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.
    ].
```

finalizeIterations

```
| n |
n := 0.
result := lowerRows collect:
                [:each |
                n := n + 1.
                each at: n].
self sortEigenValues
```

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
```

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
```

printOn: `aStream`

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

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.
                ].
```

transform

```
^DhbMatrix rows: transform
```

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.
```

8.7.3 Jacobi's Algorithm—Java Implementation

Listing 8.16 shows the Smalltalk implementation of Jacobi's algorithm.

Code Example 8.16 shows how to use the class to find the eigenvalues and eigenvectors of a symmetrical matrix.

Code Example 8.16

```
double[][] c = {{84,-79,58,55},{-79,84,-55,-58},
                 {58,-55,84,79},{55,-58,79,84}};
SymmetricMatrix m;
try { m = SymmetricMatrix.fromComponents(c);
   } catch( DhbNonSymmetricComponents e) { return;}
     catch( DhbIllegalDimension e) { return;};
JacobiTransformation jacobi = new JacobiTransformation( m);
jacobi.evaluate().
double[] eigenvalues = jacobi.eigenvalues();
DhbVector[] eigenvectors = jacobi.eigenvectors();
```

First the matrix `m` is defined from its components. Then, an instance of the class `JacobiTransformation` is created for this matrix. The iterative process is started as described in Section 4.1.2. The eigenvalues and the corresponding eigenvectors are retrieved using accessor methods.

The class `JacobiTransformation` has four instance variables

`rows`	A double dimensional array containing the components of the matrix,
`transform`	The components of the matrix **O**
`p`	The row index of the largest off-diagonal element
`q`	The column index of the largest off-diagonal element

Unlike Smalltalk, Java does not permit double dimensional arrays whose components are different sizes. Thus, one is forced to keep the components of the entire matrix.

The method `eigenvalues` returns an array containing the sorted eigenvalues, and the method `eigenvectors` returns an array of vectors containing the corresponding eigenvectors.

Listing 8.16 Java implementation of Jacobi's algorithm

```
package DhbMatrixAlgebra;

import DhbIterations.IterativeProcess;

// JacobiTransformation

// @author Didier H. Besset

public class JacobiTransformation extends IterativeProcess
{
    double[][] rows;
    double[][] transform;
    int p,q;    //Indices of the largest off-diagonal element

// Create a new instance for a given symmetric matrix.
// @param m DhbMatrixAlgebra.SymmetricMatrix

public JacobiTransformation (SymmetricMatrix m)
{
    int n = m.rows();
    rows = new double[n][n];
    for ( int i = 0; i < n; i++)
    {
        for ( int j = 0; j < n; j++)
            rows[i][j] = m.components[i][j];
    }
}

// @return double[]

public double[] eigenvalues ( )
{
    int n = rows.length;
    double[] eigenvalues = new double[n];
    for ( int i = 0; i < n; i++ )
        eigenvalues[i] = rows[i][i];
    return eigenvalues;
}

// @return DhbMatrixAlgebra.SymmetricMatrix

public DhbVector[] eigenvectors ( )
{
    int n = rows.length;
    DhbVector[] eigenvectors = new DhbVector[n];
```

```
        double[] temp = new double[n];
        for ( int i = 0; i < n; i++ )
        {
            for ( int j = 0; j < n; j++)
                temp[j] = transform[j][i];
            eigenvectors[i] = new DhbVector( temp);
        }
        return eigenvectors;
    }
    public double evaluateIteration()
    {
        double offDiagonal = largestOffDiagonal();
        transform();
        return offDiagonal;
    }

    // @param m int

    private void exchange ( int m)
    {
        int m1 = m + 1;
        double temp = rows[m][m];
        rows[m][m] = rows[m1][m1];
        rows[m1][m1] = temp;
        int n = rows.length;
        for ( int i = 0; i < n; i++ )
        {
            temp = transform[i][m];
            transform[i][m] = transform[i][m1];
            transform[i][m1] = temp;
        }
    }
    public void finalizeIterations ( )
    {
        int n = rows.length;
        int bound = n - 1;
        int i, m;
        while ( bound >= 0 )
        {
            m = -1;
            for ( i = 0; i < bound; i++ )
            {
                if ( Math.abs( rows[i][i]) < Math.abs( rows[i+1][i+1]) )
                {
                    exchange( i);
                    m = i;
                }
            }
            bound = m;
        }
        return;
    }
    public void initializeIterations()
    {
        transform = SymmetricMatrix.identityMatrix( rows.length).components;
    }
```

// *@return double   absolute value of the largest off diagonal element*

```
private double largestOffDiagonal( )
{
    double value = 0;
    double r;
    int n = rows.length;
    for (int i = 0; i < n; i++)
    {
        for ( int j = 0; j < i; j++)
        {
            r = Math.abs( rows[i][j]);
            if ( r > value )
            {
                value = r;
                p = i;
                q = j;
            }
        }
    }
    return value;
}
```

// *Returns a string representation of the system.*
// *@return java.lang.String*

```
public String toString()
{
    StringBuffer sb = new StringBuffer();
    char[] separator = { '[', ' '};
    int n = rows.length;
    for ( int i = 0; i < n; i++)
    {
        separator[0] = '{';
        for ( int j = 0; j <= i; j++)
        {
            sb.append( separator);
            sb.append( rows[i][j]);
            separator[0] = ' ';
        }
    sb.append('}');
    sb.append('\n');
    }
    return sb.toString();
}

// @return DhbMatrixAlgebra.SymmetricMatrix

private void transform ( )
{
    double apq = rows[p][q];
    if ( apq == 0 )
        return;
    double app = rows[p][p];
    double aqq = rows[q][q];
    double arp = ( aqq - app) * 0.5 / apq;
    double t = arp > 0 ? 1 / ( Math.sqrt( arp * arp + 1) + arp)
                       : 1 / ( arp - Math.sqrt( arp * arp + 1));
    double c = 1 / Math.sqrt( t * t + 1);
    double s = t * c;
    double tau = s / ( 1 + c);
    rows[p][p] = app - t * apq;
    rows[q][q] = aqq + t * apq;
    rows[p][q] = 0;
    rows[q][p] = 0;
    int n = rows.length;
    for ( int i = 0; i < n; i++ )
    {
        if ( i != p && i != q )
        {
            rows[p][i] = rows[i][p] - s *( rows[i][q]
                                    + tau * rows[i][p]);
            rows[q][i] = rows[i][q] + s *( rows[i][p]
                                    - tau * rows[i][q]);
            rows[i][p] = rows[p][i];
            rows[i][q] = rows[q][i];
        }
        arp = transform[i][p];
        aqq = transform[i][q];
        transform[i][p] = arp - s * ( aqq + tau * arp);
        transform[i][q] = aqq + s * ( arp - tau * aqq);
    }
}
}
```

CHAPTER 9

Elements of Statistics

La statistique est la premiѐre 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 ???. Some sections of Chapter ??? also use 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 in which 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. Examples 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 need of additional information should consult the numerous textbooks on the subject, (e.g., [Phillips & Taylor] or [Law & Kelton].

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 corresponds 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. *Statistical moments* can contribute to the characterization of the distribution.[2]

1. Statistics is the first of the inexact sciences.

2. Central moments are related to the coefficients of the Taylor expansion of the Fourier transform of the distribution function.

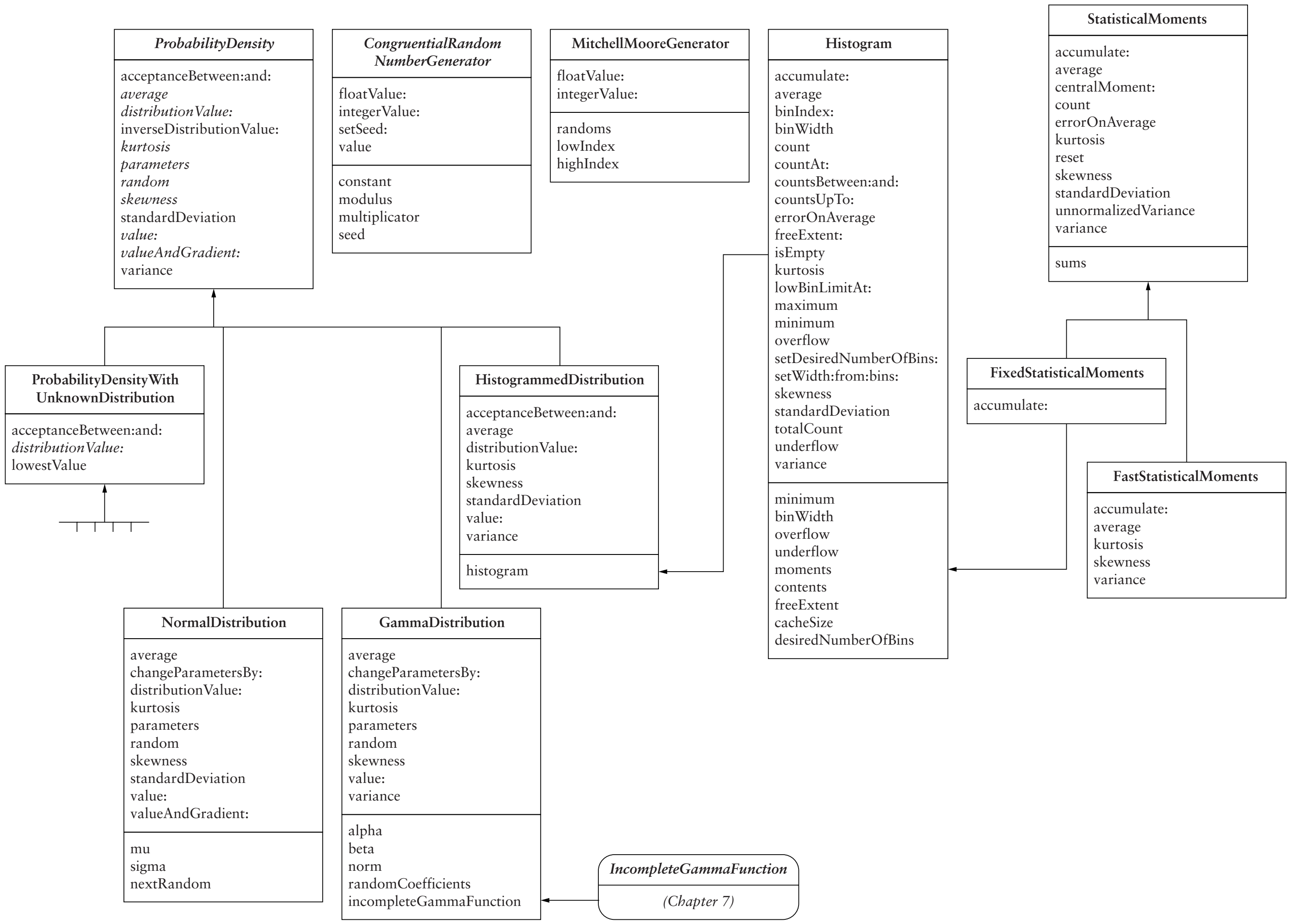

FIG. 9.1 Classes related to statistics

Given a set of measurements, $x_1, \ldots, x_n$, of the values measured for a random variable, one defines the moment of kth 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 kth order is defined by

$$m_k = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^k , \tag{9.3}$$

where $k > 1$. The central moments are easily expressed in terms of the moments. We have

$$m_k = \sum_{j=0}^{k} \binom{k}{j} (-\bar{x})^{k-j} M_j, \tag{9.4}$$

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, for example.

Two quantities are related to the central moments of 30.44rd and 4th order. Each of these quantities are normalized by a proper power of the standard deviation to yield a quantity without dimension.

The *skewness* is defined by

$$a = \frac{n}{\left(n-1\right)\left(n-2\right)s^3} m_3 = \frac{1}{\left(n-1\right)\left(n-2\right)} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s}\right)^3 . \tag{9.7}$$

The skewness is a measure of the asymmetry of a distribution. If the skewness is positive, the observed distribution is asymmetrical toward large values and vice versa.

The *kurtosis* is defined by

$$\begin{aligned} k &= \frac{n\left(n+1\right)}{\left(n-1\right)\left(n-2\right)\left(n-3\right)s^4} m_4 - \frac{3\left(n-1\right)^2}{\left(n-2\right)\left(n-3\right)} \\ &= \frac{\left(n+1\right)}{\left(n-1\right)\left(n-2\right)\left(n-3\right)} \sum_{i=1}^{n} \left(\frac{x_i - \bar{x}}{s}\right)^4 - \frac{3\left(n-1\right)^2}{\left(n-2\right)\left(n-3\right)}. \end{aligned} \tag{9.8}$$

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 zero.[3]

As we have seen, the average, standard deviation, skewness, and kurtosis are parameters, which helps to characterize 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.

9.1.1 Statistical Moments—General Implementation

To describe this implementation, we must anticipate the next section: the class implementing is a subclass of the class defined in Section 9.2.

Space allocation is handled by the superclass. The class `FastStatisticalMoments` uses this allocation to store the moments (instead of the central moments). The method `accumulate:` performs 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 (see Listings 9.3 and 9.5).

3. One talks about a *platykurtic* distribution when the kurtosis is negative; that is, the peak of the distribution is flatter than that of the normal distribution. Student (see Section ???) and Cauchy (see Section ???) distributions are platykurtic. The opposite is called *leptokurtic*. The Laplace (see Section ???) distribution is leptokurtic.

9.1.2 Statistical Moments—Smalltalk Implementation

Listing 9.1 shows the Smalltalk implementation. The class `DhbFastStatistical-Moments` is a subclass of class `DhbStatisticalMoments` presented in Listing 9.3 of Section 9.2.2. The reason for the split into two classes will become clear in Section 9.2.

Code Example 9.1 shows how to use the class `DhbFastStatisticalMoments` to accumulate measurements of a random variable and to extract the various distribution parameters discussed in Section 9.1.

Code Example 9.1

```
| accumulator valueStream average stdev skewness kurtosis |
accumulator := DhbFastStatisticalMoments new.
[ valueStream atEnd]
       whileFalse:[ accumulator accumulate: valueStream next].
average := accumulator average.
stdev := accumulator standardDeviation.
skewness := accumulator skewness.
kurtosis := accumulator kurtosis.
```

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 `DhbFastStatisticalMoments` with the default dimension. This default allocates enough storage to accumulate up to the moment of fourth 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, the returned value is `nil`.

Listing 9.1 Smalltalk fast implementation of statistical moments

class	`DhbFastStatisticalMoments`
Subclass of	`DhbStatisticalMoments`

Instance Methods

accumulate: `aNumber`

```
| var |
var := 1.
1 to: moments size
    do:
        [:n |
            moments at: n put: (moments at: n) + var.
            var := var * aNumber]
```

average

```
    self count = 0 ifTrue: [^nil].
    ^(moments at: 2) / self count
```

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
```

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))
```

variance

```
    | n |
    n := self count.
    n < 2 ifTrue: [^nil].
    ^((moments at: 3) - ((moments at: 2) squared / n)) / (n - 1)
```

9.1.3 Statistical Moments—Java Implementation

Listing 9.2 shows the complete implementation in Java.

Code Example 9.2 shows how to use the class `FastStatisticalMoments` to accumulate measurements of a random variable and to extract the various distribution parameters discussed in Section 9.1.

Code Example 9.2

```
double[] values;
   ⋮  Collecting measurements into the array values
FastStatisticalMoments accumulator = new FastStatisticalMoments();
for ( int i = 0; i < values.length; i++)
   accumulator.accumulate( values[i]);
double average = accumulator.average();
double stdev = accumulator.standardDeviation();
double skewness = accumulator.skewness();
double kurtosis = accumulator.kurtosis();
```

This example assumes that the measurements of the random variable are collected into an array of double.

After collecting the measurements, an instance of the class `FastStatisticalMoments` is created with the default dimension. This default allocates enough storage to accumulate up to the moment of fourth order. The next two lines are the loop accumulating the values into the moments. 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, the returned value is `NaN`.

The computation of the standard deviation and of the error on the average are handled by the superclass (see Listing 9.5).

Listing 9.2 Java implementation of statistical moments

```
package DhbStatistics;

// Fast StatisticalMonents (at the cost of accuracy)

// @author Didier H. Besset

public class FastStatisticalMoments extends StatisticalMoments
{

// Default constructor method.

public FastStatisticalMoments()
```

```
{
    super();
}

// Constructor method.
// @param n int

public FastStatisticalMoments(int n)
{
    super(n);
}

// Accumulate a random variable.
// @param x value of the random variable.

public void accumulate( double x)
{
    double value = 1.0;
    for ( int n=0; n < moments.length; n++)
    {
        moments[n] += value;
        value *= x;
    }
}

// @return double average.

public double average()
{
    return moments[1] / moments[0];
}

// The kurtosis measures the sharpness of the distribution near the maximum.
// Note: The kurtosis of the Normal distribution is 0 by definition.
// @return double kurtosis.

public double kurtosis() throws ArithmeticException
{
    if ( moments[0] < 4 )
        return Double.NaN;
    double x1 = average();
    double x2 = moments[2] / moments[0];
    double x3 = moments[3] / moments[0];
    double x4 = moments[4] / moments[0];
    double xSquared = x1 * x1;
    double m4 = x4 - (4 * x1 * x3) + 3 * xSquared
```

```
                                                * ( 2 * x2- xSquared);
    double kFact = moments[0] * (moments[0] + 1)
                        / ( (moments[0] - 1) * (moments[0] - 2)
                                                * (moments[0] - 3));
    double kConst = 3 * (moments[0] - 1) * (moments[0] - 1)
                        / ( (moments[0] - 2) * (moments[0] - 3));
    x4 = variance();
    x4 *= x4;
    return kFact * (m4 * moments[0] / x4) - kConst;
}

// @return double skewness.

public double skewness() throws ArithmeticException
{
    if ( moments[0] < 3 )
        return Double.NaN;
    double x1 = average();
    double x2 = moments[2] / moments[0];
    double x3 = moments[3] / moments[0];
    double m3 = x3 + x1 * ( 2 * x1 * x1 - 3 * x2);
    x1 = standardDeviation();
    x2 = x1 * x1;
    x2 *= x1;
    return m3 * moments[0] * moments[0] / ( x2 * ( moments[0] - 1)
                                                * ( moments[0] - 2));
}

// Unnormalized central moment of 2nd order
// (needed to compute the t-test).
// @return double

public double unnormalizedVariance()
{
    double average = average();
    return moments[2] - average * average * moments[0];
}

// Note: the variance includes the Bessel correction factor.
// @return double variance.

public double variance() throws ArithmeticException
{
    if ( moments[0] < 2 )
        return Double.NaN;
    double average = average();
```

```
        return ( moments[0] / ( moments[0] - 1) )
                        * (moments[2] / moments[0] - average * average);
    }
    }
```

Note: The method `unnormalizedVariance` computes the expression $\sum_{i=1}^{n} (x_i - \bar{x})^2$ needed to compute the confidence level for the t-test (see Section ???).

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:

$$\left\langle f(x)\right\rangle_n = \frac{1}{n}\sum_{i=1}^{n} f(x_i)\,, \tag{9.9}$$

where the values $x_1, \ldots, x_n$ are the measurements of the random variable. A comparison of equations 9.9 and 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 $(n+1)$th 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

$$\begin{aligned}
\Delta_{n+1} &= \langle x\rangle_n - \langle x\rangle_{n+1} \\
&= \frac{\langle x\rangle_n - x_{n+1}}{n+1}.
\end{aligned} \tag{9.12}$$

The expression in the numerator of equation 9.12 subtracts two quantities of comparable magnitude which 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

$$\left\langle (x-\bar{x})^k \right\rangle_{n+1} = \frac{n}{n+1}\left\{\left[1-(-n)^{k-1}\right]\Delta_{n+1}^k + \sum_{l=2}^{k}\binom{l}{k}\left\langle (x-\mu)^l\right\rangle_n \Delta_{n+1}^{k-l}\right\}. \tag{9.13}$$

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 rounding errors. For the central moment of order 2, we have

$$\left\langle (x-\bar{x})^2 \right\rangle_{n+1} = \frac{n}{n+1}\left\{(1+n)\,\Delta_{n+1}^2 + \left\langle (x-\bar{x})^2\right\rangle_n\right\}. \tag{9.14}$$

For the central moment of order 3, we have

$$\left\langle (x-\bar{x})^3 \right\rangle_{n+1} = \frac{n}{n+1}\left\{\left(1-n^2\right)\Delta_{n+1}^3 + 3\left\langle (x-\bar{x})^2\right\rangle_n \Delta_{n+1} + \left\langle (x-\bar{x})^3\right\rangle_n\right\}. \tag{9.15}$$

For the central moment of order 4, we have

$$\begin{aligned}
\left\langle (x-\bar{x})^4 \right\rangle_{n+1} = \frac{n}{n+1}\Big\{&\left(1+n^3\right)\Delta_{n+1}^4 + 6\left\langle (x-\bar{x})^2\right\rangle_n \Delta_{n+1}^2 \\
&+4\left\langle (x-\bar{x})^3\right\rangle_n \Delta_{n+1} + \left\langle (x-\bar{x})^4\right\rangle_n\Big\}.
\end{aligned} \tag{9.16}$$

9.2.1 Robust Central Moments—General Implementation

The class `StatisticalMoments` 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 two times slower than with the brute force method exposed in Section 9.1. The reader must decide between speed and accuracy to choose between the two implementations.

The class `FixedStatisticalMoments` is a subclass of class `StatisticalMoments` 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.

9.2.2 Robust Central Moments—Smalltalk Implementation

Listing 9.3 shows the implementation of the robust statistical moments. Listing 9.4 shows a specialization to optimize the speed of accumulation for the most frequently used case (accumulation up to the 4th order).

Using the class is identical for all classes of the hierarchy. Thus, Code Example 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—use the method `centralMoment:` retrieving the central moment of a given order. They will return `nil` if not enough data have been accumulated in the moments.

Listing 9.3 Smalltalk implementation of accurate statistical moments

class	`DhbStatisticalMoments`
Subclass of	`Object`
Instance variable names:	`moments`

Class Methods

new

```
^self new: 4
```

new: `anInteger`

```
^super new initialize: anInteger + 1
```

Instance methods

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]
```

average

```
self count = 0 ifTrue: [^nil].
^moments at: 2
```

centralMoment: `anInteger`

```
^moments at: anInteger + 1
```

count

```
^moments at: 1
```

errorOnAverage

```
^( self variance / self count) sqrt
```

initialize: `anInteger`

```
moments := Array new: anInteger.
self reset.
^self
```

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
```

reset

```
moments atAllPut: 0
```

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))
```

standardDeviation

```
^self variance sqrt
```

unnormalizedVariance

```
^( self centralMoment: 2) * self count
```

variance

```
| n |
n := self count.
n < 2
    ifTrue: [ ^nil].
^self unnormalizedVariance / ( n - 1)
```

The class `DhbFixedStatisticalMoments` is a specialization of the class `Dhb-StatisticalMoments` for a fixed number of central moments going up to the 4th 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

class	`DhbFixedStatisticalMoments`
Subclass of	`DhbStatisticalMoments`

Class Methods

new

```
^super new: 4
```

new: `anInteger`

```
^self error: 'Illegal creation message for this class'
```

Instance methods

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
```

9.2.3 Robust central moments—Java implementation

Listing 9.5 shows the implementation of the robust statistical moments. Listing 9.6 shows a specialization to optimize the speed of accumulation for the most frequently used case (accumulation up to the 4th order).

Using the class is identical for all classes of the hierarchy. Thus Code Example 9.2 is also valid for these two classes.

The class constructor method of class `StatisticalMoments` takes as argument the highest order of the accumulated moments and allocates the required storage. A class constructor method without argument creates an instance corresponding to the most frequent usage: the highest order is 4.

The methods computing the distribution parameters—average, variance, skewness, and kurtosis—return `NaN` if the result is undefined for lack of measurements.

Listing 9.5 Java implementation of accurate statistical moments

```
package DhbStatistics;

// A StatisticalMoments accumulates statistical moments of a random variable.

// @author Didier H. Besset

public class StatisticalMoments
{

    // Vector containing the points.

    protected double[] moments;

// Default constructor methods: declare space for 5 moments.

public StatisticalMoments()
{
    this( 5);
```

```
}

// General constructor methods.
// @param n number of moments to accumulate.

public StatisticalMoments( int n)
{
    moments = new double[n];
    reset();
}

// @param x double    value to accumulate

public void accumulate( double x)
{
    double n = moments[0];
    double n1 = n + 1;
    double delta = ( moments[1] - x) / n1;
    double[] sums = new double[ moments.length];
    sums[0] = moments[0];
    moments[0] = n1;
    sums[1] = moments[1];
    moments[1] -= delta;
    int[] pascal = new int[moments.length];
    pascal[0] = 1;
    pascal[1] = 1;
    double r1 = (double) n / (double) n1;
    double nk = -1;
    n = -n;
    double cterm = delta;
    double term;
    for ( int k = 2 ; k < moments.length; k++)
    {
        sums[k] = moments[k];
        nk = nk * n;
        cterm *= delta;
        term = (1 + nk) * cterm;
        for ( int l = k; l >= 2; l--)
        {
            pascal[l] += pascal[l-1];
            term += pascal[l] * sums[l];
            sums[l] *= delta;
        }
        pascal[1] += pascal[0];
        moments[k] = term * r1;
    }
```

```
}

 * @return double average.

public double average()
{
    return moments[1];
}

// Returns the number of accumulated counts.
// @return number of counts.

public long count()
{
    return (long) moments[0];
}

// Returns the error on average. May throw divide by zero exception.
// @return error on average.

public double errorOnAverage()
{
    return Math.sqrt( variance() / moments[0]);
}

// @return double    F-test confidence level with data accumulated
//                                  in the supplied moments.
// @param m DhbStatistics.StatisticalMoments

public double fConfidenceLevel( StatisticalMoments m)
{
    FisherSnedecorDistribution fDistr = new FisherSnedecorDistribution(
             (int) count(), (int) m.count());
    return fDistr.confidenceLevel( variance() / m.variance());
}

// The kurtosis measures the sharpness of the distribution near
//                                          the maximum.
// Note: The kurtosis of the Normal distribution is 0 by definition.
// @return double kurtosis or NaN.

public double kurtosis() throws ArithmeticException
{
    if ( moments[0] < 4 )
        return Double.NaN;
    double kFact = ( moments[0] - 2) * ( moments[0] - 3);
    double n1 = moments[0] - 1;
    double v = variance();
    return ( moments[4] * moments[0] * moments[0] * ( moments[0] + 1)
                    / ( v * v * n1) - n1 * n1 * 3) / kFact;
}

// Reset all counters.

public void reset( )
{
    for ( int n=0; n < moments.length; n++)
        moments[n] = 0;
}

// @return double skewness.

public double skewness() throws ArithmeticException
{
    if ( moments[0] < 3 )
        return Double.NaN;
    double v = variance();
    return moments[3] * moments[0] * moments[0]
                    / ( Math.sqrt(v) * v * ( moments[0] - 1)
                                                * ( moments[0] - 2));
}

// Returns the standard deviation. May throw divide by zero exception.
// @return double standard deviation.

public double standardDeviation()
{
    return Math.sqrt( variance());
}

// @return double    t-test confidence level with data accumulated
//                          in the supplied moments.
 * Approximation for the case where the variance of both sets may
//                              differ.
// @param m DhbStatistics.StatisticalMoments

public double tApproximateConfidenceLevel( StatisticalMoments m)
{
    StudentDistribution tDistr = new StudentDistribution(
                                    (int) ( count()+m.count()-2));
    return tDistr.confidenceLevel( ( average() / standardDeviation()
                                    - m.average()
```

```
                                            / m.standardDeviation())
                                            / Math.sqrt(1/count()
                                                    +1/m.count()));
}

// @return double    t-test confidence level with data accumulated
 *                                              in the supplied moments.
// The variance of both sets is assumed to be the same.
// @param m DhbStatistics.StatisticalMoments

public double tConfidenceLevel( StatisticalMoments m)
{
    int dof =  (int) ( count()+m.count()-2);
    double sbar = Math.sqrt( ( unnormalizedVariance()
                            + m.unnormalizedVariance()) / dof);
    StudentDistribution tDistr = new StudentDistribution( dof);
    return tDistr.confidenceLevel( ( average() - m.average())
                                / ( sbar * Math.sqrt(1/count()
                                                +1/m.count())));
}

// @return double

public double unnormalizedVariance()
{
    return moments[2] * moments[0];
}

// Note: the variance includes the Bessel correction factor.
// @return double variance.

public double variance() throws ArithmeticException
{
    if ( moments[0] < 2 )
        return Double.NaN;
    return unnormalizedVariance() / ( moments[0] - 1);
}
}
```

Note: The method `tConfidenceLevel` implements the *t*-test discussed in Section ???. The method `fConfidenceLevel` implements the *F*-Test discussed in Section ???.

The class `FixedStatisticalMoments` is a specialization of the class `StatisticalMoments` for a fixed number of central moments going up to the 4th order.

There is only one class constructor method—namely, the one corresponding to the default number of orders.

Listing 9.6 ava implementation of accurate statistical moments with fixed order

```
package DhbStatistics;

// Statistical moments for a fixed set (1-4th order)

// @author Didier H. Besset

public class FixedStatisticalMoments extends StatisticalMoments
{

// Constructor method.

public FixedStatisticalMoments()
{
    super();
}

// Quick implementation of statistical moment accumulation up to order 4.
// @param x double

public void accumulate ( double x)
{
    double n = moments[0];
    double n1 = n + 1;
    double n2 = n * n;
    double delta = ( moments[1] - x) / n1;
    double d2 = delta * delta;
    double d3 = delta * d2;
    double r1 = (double) n / (double) n1;
    moments[4] += 4 * delta * moments[3] + 6 * d2 * moments[2]
                                        + (1 + n * n2) * d2 * d2;
    moments[4] *= r1;
    moments[3] += 3 * delta * moments[2] + (1 - n2) * d3;
    moments[3] *= r1;
    moments[2] += (1 + n) * d2;
    moments[2] *= r1;
    moments[1] -= delta;
    moments[0] = n1;
    return;
}
}
```

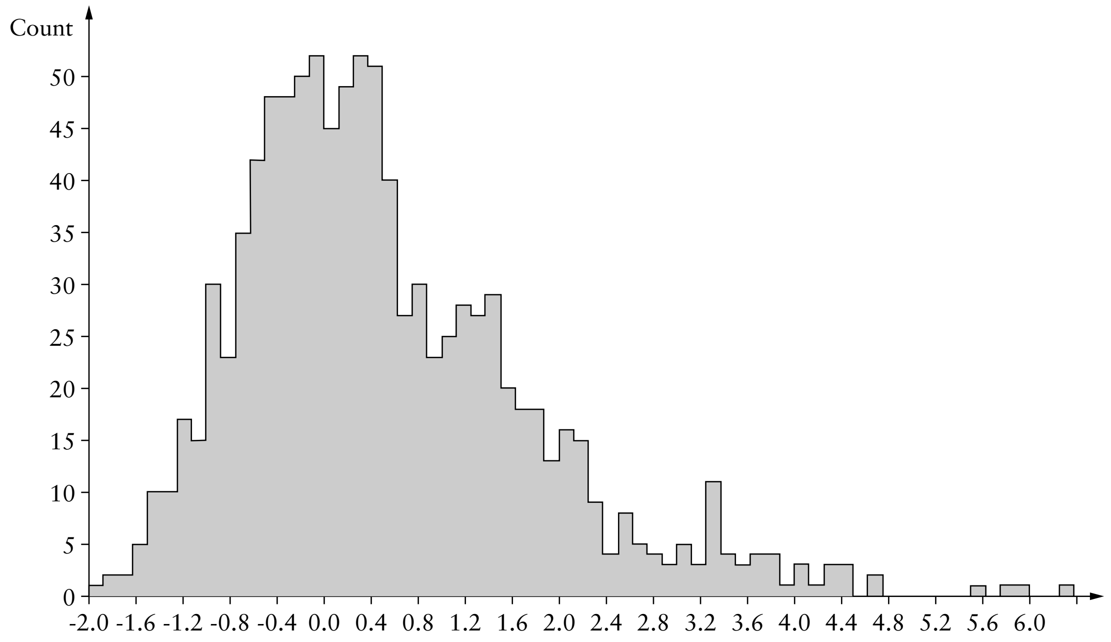

FIG. 9.2 A typical histogram

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 9.2 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 ith bin of a histogram is the interval $\left[x_{\min} + \left(i-1\right) w, x_{\min} + iw\right[$. The customary convention is that the lower limit is included in the interval and the higher limit is 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 bin. The bin width is then computed as

$$w = \frac{x_{\max} - x_{\min}}{n}, \tag{9.17}$$

where $x_{\max}$ is the maximum of the accumulated values.

In Section ???, we shall need the error on the contents of a histogram. In the 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^{th} 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 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.

9.3.1 Histograms—General Implementation

Our implementation of histogram also accumulates the values into statistical moments. One can in principle compute the statistical moments of the measured 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, my 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 accumulate 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

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.

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, these are 2.1 and 5.175, and the number of bins becomes 41 instead of 50.

9.3.2 Histograms—Smalltalk Implementation

Listing 9.7 shows the implementation of a histogram in Smalltalk. Code Example 9.3 shows how to use the class DhbHistogram to accumulate measurements into a histogram.

Code Example 9.3

```
| histogram valueStream |
histogram := DhbHistogram new.
[ valueStream atEnd]
   whileFalse:[ histogram accumulate: valueStream next].
      ⋮  <printing or display of the histogram>
```

This example assumes that the measurements 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 DhbHistogram 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 9.1.2. Extracting the parameters of the distribution can also be performed from the histogram.

Code Example 9.4 shows how to declare a histogram with given limits (two and seven) and a desired number of bins of 50.

Code Example 9.4

```
| histogram valueStream |
histogram := DhbHistogram new.
histogram setRangeFrom: 2.0 to: 7.0 bins: 100.
⋮  <the rest is identical to the previous example>
```

7. Let us recall that the order of magnitude is the power of 10 of a number.

The class `DhbHistogram` has the following instance variables:

`minimum` The minimum of the accumulated values (i.e., $x_{\min}$)

`binWidth` The bin width (i.e., 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 `DhbFixedStatisticalMoments` to accumulate statistical moments up to the 4th order (see Section 9.2.2) with minimal rounding errors

`contents` The contents of the histogram (i.e., 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 one can declare a histogram in many ways, there is a single creation method `new` that 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 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 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.

Many methods are used to retrieve information from a histogram, and 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 9.1.2, 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 ???.

Listing 9.7 Smalltalk implementation of histograms

class	`DhbHistogram`
Subclass of	`Object`
Instance variable names:	`minimum binWidth overflow underflow moments contents freeExtent cacheSize desiredNumberOfBins`

Class Methods

defaultCacheSize

```
^100
```

defaultNumberOfBins

```
^50
```

integerScales

new

```
^super new initialize
```

scales

semiIntegerScales

Instance Methods

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].
```

accumulateInCache: aNumber

```
contents add: aNumber.
contents size > cacheSize
    ifTrue: [ self flushCache].
```

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.
```

average

```
^moments average
```

binIndex: aNumber

```
^( ( aNumber - minimum) / binWidth) floor + 1
```

binWidth

```
self isCached
    ifTrue: [ self flushCache].
^binWidth
```

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
```

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
```

count

```
^moments count
```

countAt: aNumber

```
| n |
n := self binIndex: aNumber.
^( n between: 1 and: contents size)
        ifTrue: [ contents at: n]
```

```
            ifFalse:[ 0]
```

countOverflows: `anInteger`

```
    anInteger > 0
        ifTrue: [ overflow := overflow + 1]
        ifFalse:[ underflow := underflow + 1].
```

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
```

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
```

errorOnAverage

```
    ^moments errorOnAverage
```

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].
```

freeExtent: aBoolean

```
( underflow = 0 and: [ overflow = 0])
    ifFalse: [ self error: 'Histogram extent cannot be
                                                redefined'].
freeExtent := aBoolean.
```

growContents: anInteger

```
anInteger > 0
    ifTrue: [ self growPositiveContents: anInteger]
    ifFalse:[ self growNegativeContents: anInteger].
```

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.
```

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.
```

initialize

```
freeExtent := false.
cacheSize := self class defaultCacheSize.
desiredNumberOfBins := self class defaultNumberOfBins.
contents := OrderedCollection new: cacheSize.
moments := DhbFixedStatisticalMoments new.
overflow := 0.
underflow := 0.
^self
```

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
```

isCached

```
^binWidth isNil
```

isEmpty

```
^false
```

kurtosis

```
^moments kurtosis
```

lowBinLimitAt: anInteger

```
^( anInteger - 1) * binWidth + minimum
```

maximum

```
self isCached
    ifTrue: [ self flushCache].
^contents size * binWidth + minimum
```

maximumCount

```
self isCached
    ifTrue: [ self flushCache].
^contents inject: ( contents isEmpty ifTrue: [ 1]
                                     ifFalse:[ contents at: 1])
                into: [ :max :each | max max: each]
```

minimum

```
self isCached
    ifTrue: [ self flushCache].
^minimum
```

overflow

```
^overflow
```

processOverflows: `anInteger` **for:** `aNumber`

```
freeExtent
    ifTrue: [ self growContents: anInteger.
                 moments accumulate: aNumber
               ]
    ifFalse:[ self countOverflows: anInteger].
```

roundToScale: `aNumber`

setDesiredNumberOfBins: `anInteger`

```
anInteger > 0
    ifFalse:[ self error: 'Desired number of bins must be
                                                     positive'].
desiredNumberOfBins := anInteger.
```

setRangeFrom: `aNumber1` **to:** `aNumber2` **bins:** `anInteger`

```
self isCached
    ifFalse: [ self error: 'Histogram limits cannot be
                                                    redefined'].
minimum := aNumber1.
self setDesiredNumberOfBins: anInteger;
       adjustDimensionUpTo: aNumber2.
```

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).
```

skewness

```
^moments skewness
```

standardDeviation

```
^moments standardDeviation
```

totalCount

```
^moments count + underflow + overflow
```

underflow

```
^underflow
```

variance

```
^moments variance
```

9.3.3 Histograms—Java Implementation

Listing 9.8 gives the implementation of a histogram in Java. Code Example 9.5 shows how to use the class `Histogram` to accumulate measurements of a random variable and to extract the various distribution parameters discussed in Section 9.1.

Code Example 9.5

```
double[] values;
⋮  Collecting measurements into the array values
Histogram histogram = new Histogram();
for ( int i = 0; i < values.length; i++)
   histogram.accumulate( values[i]);
⋮  printing or display of the histogram
```

This example assumes that the measurements of the random variable are collected into an `array of double`. There is very little difference with the example of Section 9.1.3. One can also extract the parameters of the distribution from the histogram in the exact same way as it was done for statistical moments.

A histogram can also be declared with given limits and a desired number of bins. If the limits are 2.0 and 7.0 and the desired number of bins is 50, the histogram declaration of the previous example must be replaced by the line

```
Histogram histogram = new Histogram(2.0,7.0,50);
```

The class `Histogram` has the following instance variables:

`minimum` The minimum of the accumulated values (i.e., $x_{\min}$)

`binWidth` The bin width, (i.e., w)

`contents` The contents of the histogram, (i.e., an array of integers)

`growthAllowed` A Boolean flag denoting whether the limits of the histogram can be adjusted to include all possible values

`integerBinWidth` A Boolean flag denoting whether the limits of the histogram must be adjusted to integer scales only

`underflow` A counter to accumulate the underflow of the histogram

`overflow` A counter to accumulate the overflow of the histogram

`moments` An instance of the class `DhbFixedStatisticalMoments` to accumulate statistical moments up to the 4th order (see Section 9.2.2)

`cached` A Boolean flag denoting whether values are currently accumulated in a cache for later determination of the limits

`cache` A cache to accumulate values

There are six contructor methods for a histogram. I have used the fact that the type of the parameters uniquely defines the method. Thus, each contructor methods depends on a different combination of the parameters defining the limits:

`Histogram(double,double,int)` Defines a histogram by specifying the parameters $x_{\min}$, $x_{\max}$, and n in this order

`Histogram(int,double,double)` Defines a histogram by specifying the parameters n, $x_{\min}$, and w in this order

`Histogram(double,double)` Defines a histogram by specifying the parameters $x_{\min}$ and $x_{\max}$ in this order; the number of bins is set to 50

`Histogram(int,int)` Defines a histogram with automatic determination of the limits by specifying the size of the cache and the number of bins n in this order

`Histogram(int)` Defines a histogram with automatic determination of the limits by specifying the size of the cache; the number of bin is set to 50

`Histogram()` Defines a histogram with automatic determination of the limits with a cache size of 100 and a number of bins of 50

The method `setGrowthAllowed` sets the flag to allow the limits of the histogram to be adjusted to the accumulated values. A run time exception is thrown by this method if the histogram has already accumulated some counts in the underflow or the 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—the value is added to the cache. In this mode, the instance variable underflow is used to index the cache position. When the cache is full, limits are computed and the cache is flushed. The corresponding instance variable is reset to null. In direct accumulation mode, the value is compared to the histogram limits. If the value lies within the histogram limits, the value is accumulated. Otherwise, it is treated like an overflow or an underflow. The methods expandDown and expandUp handle the case where the limits must be adjusted.

The adjustment of the histogram limits to reasonable values is performed by the methods getBinParameters and defineParameters. The former is used when the limits are determined automatically. The latter is used when the limits are specified using one of the initialization methods. The Java implementation gives the choice to force the limits to integer values. The scale values and final algorithm can be found in the class DhbMath (see Listing 1.2).

Many methods are used to retrieve information from a histogram, and 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 9.1.3, are implemented by delegating the method to the instance variable moments.

Listing 9.8 Java implementation of histograms

```
package DhbScientificCurves;

import DhbInterfaces.PointSeries;
import DhbStatistics.FixedStatisticalMoments;
import DhbStatistics.ScaledProbabilityDensityFunction;
import DhbStatistics.ChiSquareDistribution;
import DhbFunctionEvaluation.DhbMath;
import DhbEstimation.WeightedPoint;

// A Histogram stores the frequency of hits into bins of equal width.


// @author Didier H. Besset

public class Histogram implements PointSeries
{

    // Lower limit of first histogram bin.

    private double minimum;
```

```
        // Width of a bin.

        private double binWidth;

        // Histogram contents.

        private int[] contents;

        // Flag to allow automatical growth.

        private boolean growthAllowed = false;

        // Flag to enforce integer bin width.

        private boolean integerBinWidth = false;

        // Counts of values located below first bin.
        // Note: also used to count cached values when caching is in effect.

        private int underflow;

        // Counts of values located above last bin.
        // Note: also used to hold desired number of bins when caching is in effect.

        private int overflow;

        // Statistical moments of values accumulated within the histogram limits.

        private FixedStatisticalMoments moments;

        // Flag indicating the histogram is caching values to compute adequate range.

        private boolean cached = false;

        // Cache for accumulated values.

        private double cache[];

    // Constructor method with unknown limits and a desired number
    // of 50 bins. The first 100 accumulated values are cached.
    // Then, a suitable range is computed.

    public Histogram( )
    {
        this( 100);
```

```
}

// Constructor method for approximate range for a desired number
// of 50 bins.
// All parameters are adjusted so that the bin width is a round number.
// @param from approximate lower limit of first histogram bin.
// @param to approximate upper limit of last histogram bin.

public Histogram( double from, double to)
{
    this( from, to, 50);
}

// Constructor method for approximate range and desired number of bins.
// All parameters are adjusted so that the bin width is a round number.
// @param from approximate lower limit of first histogram bin.
// @param to approximate upper limit of last histogram bin.
// @param bins desired number of bins.

public Histogram( double from, double to, int bins)
{
    defineParameters( from, to, bins);
}

// Constructor method with unknown limits and a desired number of
// 50 bins.
// Accumulated values are first cached. When the cache is full,
// a suitable range is computed.
// @param n size of cache.

public Histogram( int n)
{
    this( n, 50);
}

// General constructor method.
// @param n number of bins.
// @param min lower limit of first histogram bin.
// @param width bin width (must be positive).
// @exception java.lang.IllegalArgumentException
//                  if the number of bins is non-positive,
//                  if the limits are inversed.

public Histogram( int n, double min, double width)
                                    throws IllegalArgumentException
```

```
{
    if ( width <= 0 )
        throw new IllegalArgumentException(
                                    "Non-positive bin width: "+width);
    contents = new int[n];
    minimum = min;
    binWidth = width;
    reset();
}

// Constructor method with unknown limits.
// Accumulated values are first cached. When the cache is full,
// a suitable range is computed.
// @param n size of cache.
// @param m desired number of bins

public Histogram( int n, int m)
{
    cached = true;
    cache = new double[n];
    underflow = 0;
    overflow = m;
}

// Accumulate a random variable.
// @param x value of the random variable.

public void accumulate( double x)
{
    if ( cached )
    {
        cache[ underflow++] = x;
        if ( underflow == cache.length)
            flushCache();
    }
    else if ( x < minimum )
    {
        if ( growthAllowed )
        {
            expandDown( x);
            moments.accumulate( x);
        }
        else
            underflow++;
    }
    else
    {
        int index = binIndex(x);
        if ( index < contents.length)
        {
            contents[ index]++;
            moments.accumulate( x);
        }
        else if ( growthAllowed )
        {
            expandUp( x);
            moments.accumulate( x);
        }
        else
            overflow++;
    }
}

// Returns the average of the values accumulated in the histogram bins.
// @return average.

public double average()
{
    if ( cached )
        flushCache();
    return moments.average();
}

// @return int    index of the bin where x is located
// @param x double

public int binIndex( double x)
{
    return (int) Math.floor( (x - minimum) / binWidth);
}

// @param pdf DhbStatistics.ScaledProbabilityDensityFunction
// @return double    chi2 of histogram compared to supplied
//                             probability distribution.


public double chi2Against( ScaledProbabilityDensityFunction pdf)
{
    double chi2 = 0;
    for ( int i = 0; i < contents.length; i++ )
        chi2 += ( new WeightedPoint( 1,
                                contents[i])).chi2Contribution( pdf);
    return chi2;
}

// @param pdf DhbStatistics.ScaledProbabilityDensityFunction
// @return double    chi2 of histogram compared to supplied
//                                  probability distribution.
//

public double chi2ConfidenceLevel(
                                ScaledProbabilityDensityFunction pdf)
{
    return (new ChiSquareDistribution(contents.length -
                                            pdf.parameters().length))
                    .confidenceLevel(chi2Against(pdf));
}

// Returns the number of accumulated counts.
// @return number of counts.

public long count()
{
    return cached ? underflow : moments.count();
}

// Compute suitable limits and bin width.
// @param from approximate lower limit of first histogram bin.
// @param to approximate upper limit of last histogram bin.
// @param bins desired number of bins.
// @exception java.lang.IllegalArgumentException
//                      if the number of bins is non-positive,
//                      if the limits are inversed.

private void defineParameters( double from, double to, int bins)
                                    throws IllegalArgumentException
{
    if ( from >= to )
        throw new IllegalArgumentException(
                "Inverted range: minimum = "+from+", maximum = "+to);
    if ( bins < 1)
        throw new IllegalArgumentException(
                                "Non-positive number of bins: "+bins);
    binWidth = DhbMath.roundToScale( (to - from) / bins,
                                                integerBinWidth);
    minimum = binWidth * Math.floor( from / binWidth);
    int numberOfBins = (int) Math.ceil( ( to - minimum) / binWidth);
```

```
        if ( minimum + numberOfBins * binWidth <= to )
            numberOfBins++;
        contents = new int[numberOfBins];
        cached = false;
        cache = null;
        reset();
    }

    // Returns the error on average. May throw divide by zero exception.
    // @return error on average.

    public double errorOnAverage()
    {
        if ( cached )
            flushCache();
        return moments.errorOnAverage();
    }

    // Expand the contents so that the lowest bin include the specified
    //                                                   value.
    // @param x value to be included.

    private void expandDown( double x)
    {
        int addSize = (int) Math.ceil(( minimum - x) / binWidth);
        int newContents[] = new int[ addSize + contents.length];
        minimum -= addSize * binWidth;
        int n;
        newContents[0] = 1;
        for ( n = 1; n < addSize; n++ )
            newContents[n] = 0;
        for ( n = 0; n < contents.length; n++)
            newContents[n+addSize] = contents[n];
        contents = newContents;
    }

    // Expand the contents so that the highest bin include the specified
    //                                                   value.
    // @param x value to be included.

    private void expandUp( double x)
    {
        int newSize = (int) Math.ceil((x - minimum) / binWidth);
        int newContents[] = new int[ newSize];
        int n;
        for ( n = 0; n < contents.length; n++)
            newContents[n] = contents[n];
        for ( n = contents.length; n < newSize - 1; n++)
            newContents[n] = 0;
        newContents[n] = 1;
        contents = newContents;
    }

    // @return double   F-test confidence level with data accumulated
    //                          in the supplied histogram.
    // @param h DhbScientificCurves.Histogram

    public double fConfidenceLevel( Histogram h)
    {
        return moments.fConfidenceLevel( h.moments);
    }

    // Flush the values from the cache.

    private void flushCache()
    {
        double min = cache[0];
        double max = min;
        int cacheSize = underflow;
        double[] cachedValues = cache;
        int n;
        for ( n = 1; n < cacheSize; n++)
        {
            if ( cache[n] < min )
                min = cache[n];
            else if ( cache[n] > max )
                max = cache[n];
        }
        defineParameters( min, max, overflow);
        for ( n = 0; n < cacheSize; n++)
            accumulate( cachedValues[n]);
    }

    // @return int
    // @param x double

    public double getBinContent( double x)
    {
        if ( x < minimum)
            return Double.NaN;
        int n = binIndex(x);
        return n < contents.length ? yValueAt(n) : Double.NaN;
```

```
}

// Returns the low and high limits and the content of the bin
// containing the specified number or nul if the specified number
// lies outside of the histogram limits.
// @return a 3-dimensional array containing the bin limits and
//                                         the bin content.

public double[] getBinParameters( double x)
{
    if ( x >= minimum )
    {
        int index = (int) Math.floor( (x - minimum) / binWidth);
        if ( index < contents.length)
        {
            double[] answer = new double[3];
            answer[0] = minimum + index * binWidth;
            answer[1] = answer[0] + binWidth;
            answer[2] = contents[index];
            return answer;
        }
    }
    return null;
}

// Returns the bin width.
// @return bin width.

public double getBinWidth()
{
    return binWidth;
}

// @return double
// @param x double
// @param y double

public double getCountsBetween( double x, double y)
{
    int n = binIndex(x);
    int m = binIndex(y);
    double sum = contents[n] * ( (minimum - x) / binWidth - (n+1))
                + contents[m] * ( (y - minimum) / binWidth - m);
    while ( ++n < m)
        sum += contents[n];
    return sum;
```

```
}

// @return double integrated count up to x
// @param x double

public double getCountsUpTo( double x)
{
    int n = binIndex( x);
    double sum = contents[n] * ( (x - minimum) / binWidth - n)
                                                        + underflow;
    for ( int i = 0; i < n; i++)
        sum += contents[i];
    return sum;
}

// Returns the number of bins of the histogram.
// @return number of bins.

public double getDimension()
{
    if ( cached )
        flushCache();
    return contents.length;
}

// @return double

public double getMaximum()
{
    return minimum + ( contents.length - 1) * binWidth;
}

// Returns the lower bin limit of the first bin.
// @return minimum histogram range.

public double getMinimum()
{
    return minimum;
}

// Returns the range of values to be plotted.
// @return An array of 4 double values as follows
// index 0: minimum of X range
//       1: maximum of X range
//       2: minimum of Y range
//       3: maximum of Y range
```

```
public double[] getRange( )
{
    if ( cached )
        flushCache();
    double[] range = new double[4];
    range[0] = minimum;
    range[1] = getMaximum();
    range[2] = 0;
    range[3] = 0;
    for ( int n = 0; n < contents.length; n++ )
        range[3] = Math.max( range[3], contents[n]);
    return range;
}

// Returns the kurtosis of the values accumulated in the histogram bins.
// The kurtosis measures the sharpness of the distribution near the maximum.
// Note: The kurtosis of the Normal distribution is 0 by definition.
// @return double kurtosis.

public double kurtosis()
{
    if ( cached )
        flushCache();
    return moments.kurtosis();
}

// @return FixedStatisticalMoments

protected FixedStatisticalMoments moments()
{
    return moments;
}

// Returns the number of counts accumulated below the lowest bin.
// @return overflow.

public long overflow()
{
    return cached ? 0 : overflow;
}

// Reset histogram.

public void reset()
{
    if ( moments == null )
```

```
            moments = new FixedStatisticalMoments();
        else
            moments.reset();
        underflow = 0;
        overflow = 0;
        for ( int n = 0; n < contents.length; n++ )
            contents[n] = 0;
    }

    // Allows histogram contents to grow in order to contain all
    //                                   accumulated values.
    // Note: Should not be called after counts have been accumulated in
    // the underflow and/or overflow of the histogram.
    // @exception java.lang.RuntimeException
    //                          if the histogram has some contents.

    public void setGrowthAllowed() throws RuntimeException
    {
        if ( underflow != 0 || overflow != 0)
        {
            if ( !cached)
                throw new RuntimeException(
                        "Cannot allow growth to a non-empty histogram");
        }
        growthAllowed = true;
    }

    // Forces the bin width of the histogram to be integer.
    // Note: Can only be called when the histogram is cached.
    // @exception java.lang.RuntimeException
    //                          if the histogram has some contents.

    public void setIntegerBinWidth() throws RuntimeException
    {
        if ( !cached)
            throw new RuntimeException(
                    "Cannot change bin width of a non-empty histogram");
        integerBinWidth = true;
    }

    // Returns the number of points in the series.

    public int size()
    {
        if ( cached )
            flushCache();
        return contents.length;
    }

    // Returns the skewness of the values accumulated in the histogram bins.
    // @return double skewness.

    public double skewness()
    {
        if ( cached )
            flushCache();
        return moments.skewness();
    }

    // Returns the standard deviation of the values accumulated in the histogram bins.
    // @return double standard deviation.

    public double standardDeviation()
    {
        if ( cached )
            flushCache();
        return moments.standardDeviation();
    }

    // @return double    t-test confidence level with data accumulated
    //                                         in the supplied histogram.
    // @param h DhbScientificCurves.Histogram

    public double tConfidenceLevel( Histogram h)
    {
        return moments.tConfidenceLevel( h.moments);
    }

    // @return long

    public long totalCount()
    {
            return cached ? underflow
                          : moments.count() + overflow + underflow;
    }

    // Returns the number of counts accumulated below the lowest bin.
    // @return underflow.

    public long underflow()
    {
        return cached ? 0 : underflow;
```

```
}

// Returns the variance of the values accumulated in the histogram bins.
// @return double variance.

public double variance()
{
    if ( cached )
        flushCache();
    return moments.variance();
}

// @return DhbEstimation.WeightedPoint    corresponding to bin n.
// @param n int

public WeightedPoint weightedPointAt( int n)
{
    return new WeightedPoint( xValueAt( n), contents[n]);
}

// Returns the middle of the bin at the specified index.
// @param index the index of the bin.
// @return middle of bin

public double xValueAt( int index)
{
    return ( index + 0.5) * binWidth + minimum;
}

// Returns the content of the bin at the given index.
// @param index the index of the bin.
// @return bin content

public double yValueAt( int index)
{
    if ( cached )
        flushCache();
    return ( index >= 0 && index < contents.length) ? (double)
        contents[index] : 0;
}
}
```

Note: The method `tConfidenceLevel` implements the t-test discussed in Section ???. The method `fConfidenceLevel` implements the F-test discussed in Section ???.

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 suffices 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.

9.4.1 Linear Congruential Random Generators

The most widely used random number generators are linear congruential random generators. Random numbers are obtained from the following series [Knudth 2]:

$$X_{n+1} = (aX_n + c) \bmod m, \tag{9.18}$$

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 pseudorandom 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 zero and $m - 1$. Then the following variable,

$$x_n = \frac{X_n}{m}, \tag{9.19}$$

is a random rational number uniformly distributed between zero and 1, 1 excluded.

In practice, the modulus, multiplier, and increment must be chosen quite carefully to achieve a good randomness of the series. Knuth [Knudth 2] 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.

9.4.2 Additive Sequence Generators

Another class of random generators are additive sequence generators [Knudth 2]. The series of pseudorandom numbers is generated as follows:

$$X_n = \left(X_{n-l} + X_{n-k}\right) \bmod m, \tag{9.20}$$

where m is the modulus as before and l and k are two indices such that $l < k$. These indices must be selected carefully. A table of suitable indices appears in [Knudth 2]. 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 is case, the modulo operation on the modulus is not needed. Instead,

8. Another wide use for random number generators is games.

one simply checks whether the newly generated number is larger than 1. Thus, the series becomes

$$y_n = x_{n-l} + x_{n-k},$$
$$x_n = \begin{cases} y_n & \text{if } y_n < 1, \\ y_n - 1 & \text{if } y_n \geq 1. \end{cases} \tag{9.21}$$

It is clear that the evaluation here is much faster than that of equation 9.18, in practice, the additive sequence generator is about four 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 [Knudth 2]. The corresponding sequence has a length of $2^{55} - 1$. In our tests (discussed later) 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 [Knudth 2] Knuth describes a wealth of tests to investigate the randomness of random number generators. Some of these tests are also discussed in [Law & Kelton]. To study the goodness of the proposed random generators here, I 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 (see Section ???) 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 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 (two values), gamma distribution (two values), and beta distribution (four values)—one can be fairly confident that short-term correlations[9] are not creating problems. The code for this test is given in Code Examples ??? for Smalltalk and ??? for Java. 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 (see Section ???) 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.

9. Pseudorandom 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.

9.4.3 Bit-Pattern Generators

The generators described earlier are suitable to the generation of random values, but not for the generation of random bit patterns [Knudth 2], [Press *et al.*]. 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 polynomial

$$G(x) = x^{16} + x^{12} + x^5 + 1 \qquad (9.22)$$

is a good generator for random patterns of 16 bits. Of course, the evaluation of equation 9.22 does not require the computation of the polynomial. The following algorithm can be used:

1. Set $X_{n+1}X_n$ 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 zero) 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 [Press *et al.*].

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.

9.4.4 Random Number Generator—Smalltalk Implementation

Listing 9.9 shows the implementation of a congruential random generator in Smalltalk. Listing 9.10 shows the implementation of a additive sequence random generator in Smalltalk. Listing 9.11 shows usage of the generator for standard use.

The class `DhbCongruentialRandomNumberGenerator` has three public methods:

`value` returns the next random number of the series (i.e., X_n, a number between zero and m).

`floatValue` returns the next random floating number (i.e., the value X_n/m).

`integerValue:` returns a random integer, whose values are between zero and the specified argument.

When calling any of these 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. Code Example 9.6 shows how to proceed assuming the application uses the values generated by the random generator directly.

Code Example 9.6

```
| generator x |
generator := DhbCongruentialRandomNumberGenerator new.
:  <here is where the generator is used>
x := generator value.
```

If one needs several series that 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. Code Example 9.7 assumes that the generated numbers must be floating numbers.

Code Example 9.7

```
| generators seeds index x |
seeds := ¡an array containing the desired seeds¿
generators := seeds collect:
   [ :each | DhbCongruentialRandomNumberGenerator seed: each].
:  <here is where the various generators are used>

:  <index is the index of the desired series>
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. Code Example 9.8 shows how to do it.

Code Example 9.8

```
| generator x |
generator := DhbCongruentialRandomNumberGenerator
   seed: Time millisecondClockValue.
:  <here is where the generator is used>
x := (generator integerValue: 20) + 1.
```

In this last example, the generated numbers are integers between 1 and 20.

Implementation

The class `DhbCongruentialRandomNumberGenerator` has the following instance variables:

`constant` The increment c

`modulus` The modulus m

`multiplicator` The multiplier a

`seed` The last generated number X_{n-1}

There are three instance creation methods. The method `new` returns a SINGLETON instance containing parameters from [Knudth 2]. The method `seed:` allows one to create an instance to generate a series of random numbers 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 [Knudth 2] 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[10] 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 9.9 Smalltalk implementation of congruential random number generators

Class	`DhbCongruentialRandomNumberGenerator`
Subclass of	`Object`
Instance variable names:	`constant modulus multiplicator seed`
Class variable names:	`UniqueInstance`

Class Methods

constant: `aNumber1` **multiplicator:** `aNumber2`

modulus: `aNumber3`

```
^super new
    initialize: aNumber1
    multiplicator: aNumber2
    modulus: aNumber3
```

10. I am grateful to Dave N. Smith of IBM for this useful tip.

new

```
UniqueInstance isNil
    ifTrue: [ UniqueInstance := super new initialize.
                 UniqueInstance setSeed: 1.
              ].
^UniqueInstance
```

seed: `aNumber`

```
^( super new) initialize; setSeed: aNumber; yourself
```

Instance Methods

floatValue

```
^self value asFloat / modulus
```

initialize

```
self initialize: 2718281829.0 multiplicator: 3141592653.0
                                            modulus: 4294967296.0.
```

initialize: `aNumber1` **multiplicator:** `aNumber2`

modulus: `aNumber3`

```
constant := aNumber1.
modulus := aNumber2.
multiplicator := aNumber3.
self setSeed: 1.
```

integerValue: `anInteger`

```
^( self value  \\ ( anInteger * 1000)) // 1000
```

setSeed: `aNumber`

```
seed := aNumber.
```

value

```
seed := ( seed * multiplicator + constant) \\ modulus.
^seed
```

The class `DhbMitchellMooreGenerator` 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 zero and the specified argument.

When calling either of these 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 `DhbCongruentialRandomNumberGenerator`. Please refer to Code Examples 9.6 and 9.7. 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 *good* indices k and l in [Knudth 2].

Listing 9.10 Smalltalk implementation of an additive sequence random number generator

Class	`DhbMitchellMooreGenerator`
Subclass of	`Object`
Instance variable names:	`randoms lowIndex highIndex`
Class variable names:	`UniqueInstance`

Class Methods

constants: `anArray` **lowIndex:** `anInteger`

```
^super new initialize: anArray lowIndex: anInteger
```

default

```
| congruentialGenerator |
congruentialGenerator := DhbCongruentialRandomNumberGenerator new.
^self generateSeeds: congruentialGenerator
```

generateSeeds: `congruentialGenerator`

new

```
UniqueInstance isNil
    ifTrue: [ UniqueInstance := self default].
^UniqueInstance
```

seed: `anInteger`

```
| congruentialGenerator |
congruentialGenerator := DhbCongruentialRandomNumberGenerator
                                                      seed: anInteger.
^self generateSeeds: congruentialGenerator
```

Instance Methods

floatValue

```
| x |
x := (randoms at: lowIndex) + (randoms at: highIndex).
x < 1.0 ifFalse: [x := x - 1.0].
randoms at: highIndex put: x.
highIndex := highIndex + 1.
highIndex > randoms size ifTrue: [highIndex := 1].
lowIndex := lowIndex + 1.
lowIndex > randoms size ifTrue: [lowIndex := 1].
^x
```

initialize: `anArray` **lowIndex:** `anInteger`

```
randoms := anArray.
lowIndex := anInteger.
highIndex := randoms size.
^self
```

integerValue: `anInteger`

```
^( self floatValue * anInteger) truncated
```

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 free the user from keeping track of the instance of the random number generator. For example, the Smalltalk expression

```
50 random
```

generates an integer random number between zero and 49 included. Similarly, the Smalltalk expression

```
2.45 random
```

generates a floating random number between zero and 2.45 excluded. Finally, the Smalltalk expression

```
Number random
```

generates a floating random number between zero and 1 excluded.

Listing 9.11 Smalltalk implementation of random number generators

Class	`Integer`
Subclass of	`Number`

Instance Methods

random

```
^DhbMitchellMooreGenerator new integerValue: self
```

Class	`Number`
Subclass of	`Magnitude`

Class Methods

random

```
^DhbMitchellMooreGenerator new floatValue
```

Instance Methods

random

```
^self class random * self
```

9.4.5 Random Number Generator—Java Implementation

Java provides its own congruential random number generator, which is sufficient for most purposes. If one needs a better generator, the Smalltalk code presented in Listing 9.9 can be directly transposed into Java. Listing 9.12 shows the Java implementation of an additive sequence random generator.

The class `MitchellMooreGenerator` has three constructor methods.

`MitchellMooreGenerator()` returns an instance of the generator with standard parameters—$k = 55$ and $l = 24$—with the initial series filled using the random number generator proposed by Java.

`MitchellMooreGenerator(int, int)` returns an instance of the generator with given parameters—k and l, respectively—with the initial series filled using the random number generator proposed by Java.

`MitchellMooreGenerator(double[], int)` returns an instance of the generator by specifying the initial series given by the first argument—the size of the argument being the parameter k—and the parameter l given by the second argument.

The class `MitchellMooreGenerator` has two public methods. The method `nextDouble` returns a random floating number between zero and 1 excluded. The method `nextInteger` returns a random integer between zero and $n - 1$ where n is the specified argument. When calling any of these methods, the next random number of the series is obtained using equation 9.18.

Listing 9.12 Java implementation of an additive sequence random number generator

```
package DhbStatistics;

import java.util.Random;

// MitchellMoore random number generator

// @author Didier H. Besset

public class MitchellMooreGenerator
{

    // List of previously generated numbers

    private double[] randoms;

    // Index of last generated number

    int highIndex;

    // Index of number to add to last number

    int lowIndex;

// Default constructor.

public MitchellMooreGenerator()
{
    this(55,24);
}

// Constructor method.
// @param seeds double[]
// @param index int

public MitchellMooreGenerator( double[] seeds, int index)
{
    highIndex = seeds.length;
    randoms = new double[ highIndex];
    System.arraycopy( seeds, 0, randoms, 0, --highIndex);
    lowIndex = index - 1;
}

// Constructor method.
// @param indexH int    high index
// @param indexL int    low index

public MitchellMooreGenerator( int indexH, int indexL)
{
    Random generator = new Random();
    randoms = new double[indexH];
    for ( int i = 0; i < indexH; i++)
        randoms[i] = generator.nextDouble();
    highIndex = indexH - 1;
    lowIndex = indexL - 1;
}

// @return double    the next random number

public double nextDouble()
{
    double x = randoms[ highIndex--] + randoms[ lowIndex--];
    if( highIndex < 0)
        highIndex = randoms.length - 1;
    if( lowIndex < 0)
        lowIndex = randoms.length - 1;
    return ( randoms[highIndex] = x < 1.0 ? x : x - 1);
}

// @return long    returns a long integer between 0 and n-1
// @param n long

public long nextInteger( long n)
{
    return (long) (n * nextDouble());
}
}
```

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 $\left[x, x+dx\right[$. A probability density function is a special case of a one variable function described in Section 2.1.

The moment of kth 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, $M_0 = 1$.

As for statistical moments, one defines the mean or average of the distribution as

$$\mu = M_1 = \int xP(x)\,dx. \tag{9.24}$$

Then the central moment of kth 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, s, is the square root of m_2. The skewness and kurtosis[11] of a probability density function are defined, respectively, as

$$\omega = \frac{m_3}{\sqrt[3/2]{m_2}} = \frac{m_3}{s^3} \quad \text{and} \tag{9.26}$$

$$\kappa = \frac{m_4}{m_2^2} - 3 = \frac{m_4}{s^4} - 3. \tag{9.27}$$

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 $x \geq x_{\min}$, the distribution function is given by

$$F(t) = \text{Prob}(x < t) = \int_{x_{\min}}^{t} P(x)\,dx. \tag{9.29}$$

Instead of the distribution function, the name *centile* is sometimes used to refer to the value of the distribution function expressed as a percentage. This kind of terminology is frequently used in medical and natural science publications. For example, a value x is said to be at the 10th centile if $F(t) = 1/10$; in other words, there is a ten% chance of observing a value less than or equal to t.[12]

11. In old references the kurtosis, as defined here, is called *excess;* then, the kurtosis is defined as the square of the excess; (see, e.g., [Abramovitz & Stegun])

12. A centile can also be quoted for a value relative to a set of observed values.

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 (see Section 6.4) applied to equation 9.30.

The inverse of the repartition function is frequently used. For example, to determine an acceptance threshold in a decision process, one needs to 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 5.3.2. This 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 zero 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 [Cormen *et al.*], 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 (e.g., it is determined using numerical integration, the computation of the inverse value is best obtained by interpolating the inverse function over a set of by tabulated values (see Chapter 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 zero 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 (see 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 [Law & Kelton]. They are not discussed here, but the code is included in each implementation of the specific probability distribution.

The distributions used in this book are presented in the rest of this chapter. Other important distributions are presented in Appendix ???.

TABLE 9.1 Public methods for probability density functions

Description	Smalltalk	Java
$P(x)$	`value:`	`value(double)`
$F(x)$	`distributionValue:`	`distributionValue(double)`
$F(x_1, x_2)$	`acceptanceBetween:and:`	`distributionValue(double,double)`
$F^{-1}(x)$	`inverseDistributionValue:`	`inverseDistributionValue(double)`
$x^{\dagger}$	`random`	`random()`
$\bar{x}$	`average`	`average()`
s^2	`variance`	`variance()`
s	`standardDeviation`	`standardDeviation()`
skewness	`skewness`	`skewness()`
kurtosis	`kurtosis`	`kurtosis()`

† x represents the random variable itself. In other words, the method `random` returns a random value distributed according to the distribution.

9.5.1 Probability Distributions—General Implementation

Table 9.1 shows the description of the public methods of the implementations of both languages.

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 and `NaN` in Java. A very general implementation could have used explicit integration. The integration interval, however, may be infinite, and a general integration startegy cannot easily be supplied. Subclasses are expected to implement both methods.

As noted earlier, a probability density function is a function, as defined in Section 2.1. Since the distribution function is also a function, an ADAPTER must be provided to create a function (as defined in Section 2.1) for the distribution function.

9.5.2 Probability Distributions—Smalltalk Implementation

Listing 9.13 shows the implementation of a general probability density distribution in Smalltalk. The class `DhbProbabilityDensity` is an abstract implementation. Concrete implementation of probability density distributions is a subclass of it.

The method `distributionValue:` returning the value of the distribution function must be implemented by the subclass. The interval acceptance functions are computed 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 zero and 1. The private method uses the class `DhbNewtonZeroFinder` discussed in Section 5.3.1. 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 (see 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 (see Chapter ???). 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 9.13 Smalltalk implementation of a probability distribution

Class	`DhbProbabilityDensity`
Subclass of	`Object`

Class Methods

distributionName

```
^'Unknown distribution'
```

fromHistogram: `aHistogram`

```
^nil
```

Instance Methods

acceptanceBetween: `aNumber1` **and:** `aNumber2`

```
^( self distributionValue: aNumber2) - ( self distributionValue:
                                                         aNumber1)
```

approximatedValueAndGradient: `aNumber`

```
| delta parameters dp gradient n |
parameters := self parameters.
n := parameters size.
dp := self value: aNumber.
delta := Array new: n.
delta atAllPut: 0.
gradient := DhbVector new: n.
1 to: n do:
    [ :k |
      delta at: k put: ( parameters at: k) * 0.0001.
      self changeParametersBy: delta.
      gradient at: k put: ( ( ( self value: aNumber) - dp) / (
                                                   delta at: k)).
      delta at: k put: ( delta at: k ) negated.
      k > 1
        ifTrue: [ delta at: ( k - 1) put: 0].
    ].
self changeParametersBy: delta.
^Array with: dp with: gradient
```

average

```
self subclassResponsibility.
```

distributionFunction

```
^DhbProbabilityDistributionFunction density: self
```

distributionValue: `aNumber`

```
^self subclassResponsibility
```

inverseDistributionValue: `aNumber`

```
^( aNumber between: 0 and: 1)
        ifTrue: [ self privateInverseDistributionValue: aNumber]
        ifFalse:[ self error: 'Illegal argument for inverse
                                              distribution value']
```

kurtosis

```
^nil
```

parameters

```
^self subclassResponsibility
```

printOn: `aStream`

```
aStream nextPutAll: self class distributionName.
self parameters ifNotNil: [ :params | | first |
    first := true.
    aStream nextPut: $(.
    params do:
        [ :each |
        first ifTrue: [ first := false]
               ifFalse:[ aStream nextPut: $,].
        aStream space.
        each printOn: aStream.
        ].
    aStream nextPut: $).
    ].
```

privateInverseDistributionValue: `aNumber`

```
^( DhbNewtonZeroFinder function: [ :x | ( self distributionValue: x)
                                     - aNumber] derivative: self)
    initialValue: self average / (1 - aNumber); evaluate
```

random

```
^self privateInverseDistributionValue: DhbMitchellMooreGenerator
                                                         new floatValue
```

skewness

```
^nil
```

standardDeviation

```
^self variance sqrt
```

value: `aNumber`

```
self subclassResponsibility.
```

valueAndGradient: `aNumber`

```
^self approximatedValueAndGradient: aNumber
```

variance

```
^self standardDeviation squared
```

The class `DhbProbabilityDensityWithUnknownDistribution` 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:`) use equations 9.28 and 9.30, respectively, using the class `DhbRombergIntegrator` discussed in Section 6.4.2. 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 nonnegative numbers, this method returns zero. If the supplied default is not appropriate, the method `lowestValue` must be redefined by the subclass.

Listing 9.14 Smalltalk implementation of a probability distribution with unknown distribution function

Class `DhbProbabilityDensityWithUnknownDistribution`

Subclass of `DhbProbabilityDensity`

Instance Methods

acceptanceBetween: `aNumber1` **and:** `aNumber2`

```
^( DhbRombergIntegrator new: self from: aNumber1 to: aNumber2)
                                                      evaluate
```

distributionValue: `aNumber`

```
^( DhbRombergIntegrator new: self from: self lowestValue to:
                                             aNumber) evaluate
```

lowestValue

```
^0
```

Listing 9.15 shows the implementation of the ADAPTER for the distribution function. The class `DhbProbabilityDistributionFunction` has a single instance variable containing the corresponding probability density function. The creation method `density:` takes an instance of class `DhbProbabilityDensity` as argument.

Listing 9.15 Smalltalk implementation of a probability distribution function

Class `DhbProbabilityDistributionFunction`

Subclass of `Object`

Instance variable names: `probabilityDensity`

Class Methods

density: `aProbabilityDensity`

```
^self new initialize: aProbabilityDensity
```

Instance Methods

initialize: `aProbabilityDensity`

```
probabilityDensity := aProbabilityDensity.
^self
```

value: `aNumber`

```
^probabilityDensity distributionValue: aNumber
```

9.5.3 Probability Distributions—Java Implementation

Listings 9.13 and 9.17 show the implementation of two abstract classes implementing the general behavior of a probability density distribution in Java. Concrete probability density distributions are implemented as a subclass of one of them.

The class `ProbabilityDensityFunction` assumes that the distribution function can be computed by the subclass, using either an analytical expression (e.g.,Weibull distribution in Section ???) or a numerical approximation (e.g., Normal distribution in Section 9.6). Therefore, the method `distributionValue` with one argument, in charge of computing the distribution function, has been declared abstract. The interval acceptance function—implemented by the method `distributionValue` with two arguments—is implemented 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 zero and 1. It throws the exception `IllegalArgumentException` if this is not the case. The private method uses the class `NewtonZeroFinder` discussed in Section 5.3.1. The derivative for the Newton zero finder is the instance itself since, by definition, the probability density distribution is the derivative of the acceptance function (see equation 9.28).

Strangely enough, Java does not allow abstract constructor methods. For each subclass, a constructor method using a histogram as argument must be provided to be consistent with the rest of the statistical analysis code. This method is the equivalent of the Smalltalk class creation method `fromHistogram:`, creating 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 constructor method is used to compute suitable starting values for least-square or maximum-likelihood fits (see Chapter ???).

Listing 9.16 Java implementation of a probability distribution

```
package DhbStatistics;

import DhbFunctionEvaluation.DhbMath;
import DhbIterations.NewtonZeroFinder;
import DhbInterfaces.ParametrizedOneVariableFunction;
import java.util.Random;

// Subclasses of this class represent probability density function.
// The value of the funtion f (x) represents the probability that a
// continuous random variable takes values in the interval [x, x+dx[.
// A norm is defined for the case where the function is overlayed over
// a set of experimental points or a histogram.

// @author Didier H. Besset

public abstract class ProbabilityDensityFunction
                            implements ParametrizedOneVariableFunction
{

    // Random generator needed if random numbers are needed
    // (lazy initialization used).

    private Random generator = null;


// Compute an approximation of the gradient.
// @return double[]
// @param x double

public double[] approximateValueAndGradient( double x)
{
    double temp, delta;
    double[] params = parameters();
    double[] answer = new double[ params.length + 1];
    answer[0] = value( x);
    for ( int i = 0; i < params.length; i++ )
    {
        temp = params[i];
        delta = Math.abs( temp) > DhbMath.defaultNumericalPrecision()
                        ? 0.0001 * temp : 0.0001;
        params[i] += delta;
        setParameters( params);
        answer[i+1] = ( value(x) - answer[0]) / delta;
        params[i] = temp;
    }
    setParameters( params);
    return answer;
}
```

```
// @return double average of the distribution.

public abstract double average ( );

// Returns the probability of finding a random variable smaller than
// or equal to x.
// This method assumes that the probability density is 0 for x ¡ 0.
// If this is not the case, the subclass must implement this method.
// @return integral of the probability density function from 0 to x.
// @param x double upper limit of intergral.

public abstract double distributionValue ( double x);

// Returns the probability of finding a random variable between x1 and x2.
// Computing is made using the method distributionValue(x).
// This method should be used by distributions whose distributionValue
// is computed using a method overiding the default one.
// @return double integral of the probability density function from x1 to x2.
// @param x1 double lower limit of intergral.
// @param x2 double upper limit of intergral.

public double distributionValue ( double x1, double x2)
{
    return distributionValue(x2) - distributionValue(x1);
}

// @return java.util.Random a random number generator.

protected Random generator( )
{
    if ( generator == null)
        generator = new Random();
    return generator;
}

// @return double the value for which the distribution function is
// equal to x.
// @param x double value of the distribution function.
// @exception java.lang.IllegalArgumentException
//                        if the argument is not between 0 and 1.

public double inverseDistributionValue ( double x)
    throws IllegalArgumentException
{
    if ( x < 0 || x > 1)
        throw new IllegalArgumentException( "argument must be between
        0 and 1");
    return privateInverseDistributionValue( x);
}

//  @return double kurtosis of the distribution.

public double kurtosis( )
{
    return Double.NaN;
}

//  @return java.lang.String the name of the distribution.

public abstract String name ( );

//  This method assumes that the range of the argument has been checked.
//  Computation is made using the Newton zero finder.
//  @return double the value for which the distribution function
//                                    is equal to x.
//  @param x double value of the distribution function.

protected double privateInverseDistributionValue ( double x)
{
    OffsetDistributionFunction distribution =
       new OffsetDistributionFunction( this, x);
    NewtonZeroFinder zeroFinder = new NewtonZeroFinder(
       distribution, this, average());
    zeroFinder.setDesiredPrecision(
       DhbMath.defaultNumericalPrecision());
    zeroFinder.evaluate();
    return zeroFinder.getResult();
}

//  @return double a random number distributed according to the receiver.

public double random ( )
{
    return privateInverseDistributionValue( generator().nextDouble());
}

//  Set the seed of the random generator used by the receiver.
//  @param seed long

public void setSeed(long seed)
{
    generator().setSeed( seed);
    return;
}

// @return double skewness of the distribution.

public double skewness( )
{
    return Double.NaN;
}

// NOTE: subclass MUST implement one of the two methods variance
//       or standardDeviation.
// @return double standard deviation of the distribution from the variance.

public double standardDeviation( )
{
    return Math.sqrt( variance());
}

// @return java.lang.String name and parameters of the distribution.

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( name());
    char[] separator = { '(', ' '};
    double[] parameters = parameters();
    for ( int i = 0; i < parameters.length; i++)
    {
        sb.append( separator);
        sb.append( parameters[i]);
        separator[0] = ',';
    }
    sb.append(')');
    return sb.toString();
}

// Evaluate the distribution and the gradient of the distribution with respect
// to the parameters.
// @return double[]   0: distribution's value, 1,2,...,n distribution's gradient
// @param x double

public double[]valueAndGradient( double x)
{
    return approximateValueAndGradient( x);
}
```

```
//  NOTE: subclass MUST implement one of the two methods variance
//         or standardDeviation.
//  @return double variance of the distribution from the standard deviation.

public double variance ( )
{
    double v = standardDeviation();
    return v*v;
}
}
```

Note: The interface `ParametrizedOneVariableFunction` implemented by the probability density function is needed for nonlinear least square fits and maximum-likelihood fits. It is described in Section ???.

The class `ProbabilityDensityFunctionWithUnknownDistribution` is the superclass of probability distributions whose distribution function cannot be computed either analytically or numerically. The methods `distributionValue` with one or two arguments are computing the integral of equation 9.29 using the class `RombergIntegrator` discussed in Section 6.4.3. The lower limit of the integral for the distribution function—$x_{\min}$ of equation 9.29—is defined by the abtract method `lowestValue`. This method must be implemented by the subclass.

Listing 9.17 Java implementation of a probability distribution with unknown distribution

```
package DhbStatistics;

import DhbFunctionEvaluation.DhbMath;
import DhbIterations.RombergIntegrator;

//  Subclasses are distributions whose distribution cannot be expressed
//  as a closed expression or have no numerical approximation.

//  @author Didier H. Besset

public abstract class ProbabilityDensityWithUnknownDistribution
                                    extends ProbabilityDensityFunction
{


//  Returns the probability of finding a random variable smaller than
//  or equal to x.
//  Computation is done using Romberg integration.
//  @return integral of the probability density function from lowValue() to x.
//  @param x double upper limit of intergral.
public double distributionValue ( double x)
{
    RombergIntegrator integrator = new RombergIntegrator(
       this, lowValue(), x);
    integrator.setDesiredPrecision(
       DhbMath.defaultNumericalPrecision());
    integrator.evaluate();
    return integrator.getResult();
}

//  Returns the probability of finding a random variable between x1 and x2.
//  @return double integral of the probability density function from x1 to x2.
//  @param x1 double lower limit of integral.
//  @param x2 double upper limit of integral.

public double distributionValue ( double x1, double x2)
{
    RombergIntegrator integrator = new RombergIntegrator( this, x1, x2);
    integrator.setDesiredPrecision(
                                DhbMath.defaultNumericalPrecision());
    integrator.evaluate();
    return integrator.getResult();
}

//  @return double    lower limit of the integral used to compute
//                              the distribution function

protected abstract double lowValue();
}
```

Listing 9.15 shows the implementation of the ADAPTER for the distribution function. The class `OffsetDistributionFunction` has the following instance variables:

`probabilityDensity`	An instance of class `ProbabilityDensityFunction`
`offset`	A value subtracted to the distribution function

The constructor method takes as arguments an instance of class `ProbabilityDensityFunction` and a double. Instances of this class are used to compute the inverse value of the distribution function. It can also be used to implement the distribution function as described in Section 2.1.2 using an offset of zero. If parameters cannot be estimated, the constructor method throws the exception `IllegalArgumentException`. 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 9.18 Java implementation of a probability distribution function

```
package DhbStatistics;

import DhbInterfaces.OneVariableFunction;

// This class is used to find the inverse distribution function of
// a probability density function.

// @author Didier H. Besset

public final class OffsetDistributionFunction
                                        implements OneVariableFunction
{

    // Probability density function.

    private ProbabilityDensityFunction probabilityDensity;

    // Value for which the inverse value is desired.

    private double offset;



// Create a new instance with given parameters.
// @param p statistics.ProbabilityDensityFunction
// @param x double

protected OffsetDistributionFunction ( ProbabilityDensityFunction p,
                                                            double x)
{
    probabilityDensity = p;
    offset = x;
}

// @return distribution function minus the offset.

public double value(double x)
{
    return probabilityDensity.distributionValue( x) - offset;
}
}
```

TABLE 9.2 Properties of the normal distribution

Range of random variable	$]-\infty, +\infty[$
Probability density function	$P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2sigma^2}}$
Parameters	$-\infty < \mu < +\infty$
	$0 < \sigma < +\infty$
Distribution function	$F(x) = \mathrm{erf}\left(\frac{x-\mu}{\sigma}\right)$
	(see Section 2.3)
Average	μ
Variance	σ^2
Skewness	0
Kurtosis	0

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[13] to measurement fluctuation usually follow a normal distribution.

Table 9.2 shows the properties of the normal distribution. Figure 9.3 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.

9.6.1 Normal Distribution—Smalltalk Implementation

Listing 9.19 shows the implementation of the normal distribution in Smalltalk.

The distribution function of the normal distribution can be computed with the error function (see Section 2.3). Therefore, the class `DhbNormalDistribution` is implemented as a subclass of `DhbProbabilityDensity`.

Listing 9.19 Smalltalk implementation of the normal distribution

Class	`DhbNormalDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`mu sigma nextRandom`

13. The presence of systematic errors is a notable exception to this rule.

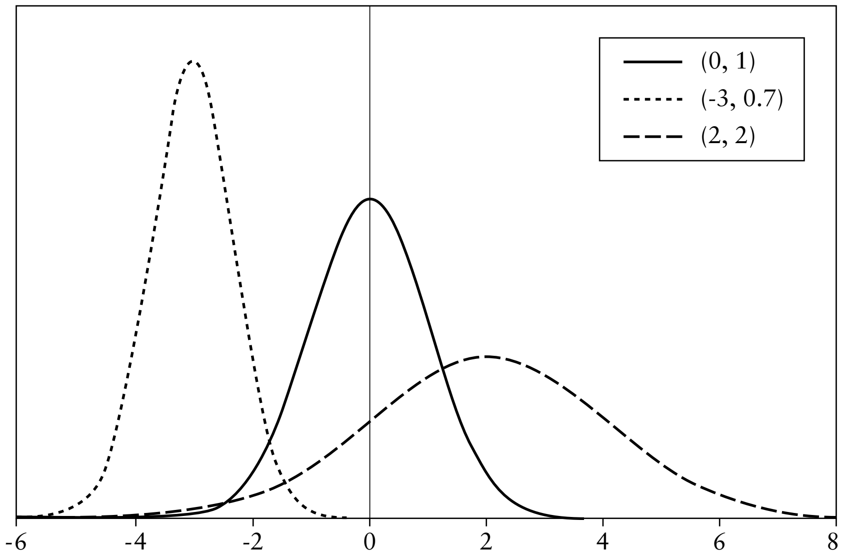

FIG. 9.3 Normal distribution for various values of the parameters

Class variable names: `NextRandom`

Class Methods

distributionName

```
^'Normal distribution'
```

fromHistogram: `aHistogram`

```
^self new: aHistogram average sigma: aHistogram standardDeviation
```

new

```
^self new: 0 sigma: 1
```

new: `aNumber1` **sigma:** `aNumber2`

```
^super new initialize: aNumber1 sigma: aNumber2
```

random

```
| v1 v2 w y |
NextRandom isNil
    ifTrue: [ [ v1 := Number random * 2 - 1.
                v2 := Number random * 2 - 1.
                w := v1 squared + v2 squared.
                w > 1 ] whileTrue: [].
              y := ( ( w ln * 2 negated) / w) sqrt.
            v1 := y * v1.
            NextRandom := y * v2.
            ]
    ifFalse:[ v1 :=NextRandom.
              NextRandom := nil.
            ].
^v1
```

Instance Methods

average

```
^mu
```

changeParametersBy: `aVector`

```
mu := mu + ( aVector at: 1).
sigma := sigma + ( aVector at: 2).
```

distributionValue: `aNumber`

```
^DhbErfApproximation new value: ( ( aNumber - mu) / sigma)
```

initialize: `aNumber1` **sigma:** `aNumber2`

```
mu := aNumber1.
sigma := aNumber2.
^self
```

kurtosis

```
^0
```

parameters

```
^Array with: mu with: sigma
```

random

```
^self class random * sigma + mu
```

skewness

```
^0
```

standardDeviation

```
^sigma
```

value: `aNumber`

```
^( DhbErfApproximation new normal: (aNumber - mu) / sigma) /
                                                         sigma
```

valueAndGradient: `aNumber`

```
| dp y |
y := ( aNumber - mu) / sigma.
dp := ( DhbErfApproximation new normal: y) / sigma.
^Array with: dp
       with: ( DhbVector with: dp * y / sigma
                         with: dp * ( y squared - 1) / sigma)
```

9.6.2 Normal Distribution—Java Implementation

Listing 9.20 shows the implementation of the normal distribution in Java.

The distribution function of the normal distribution can be computed with the error function (see Section 2.3). Therefore, the class `NormalDistribution` is implemented as a subclass of `ProbabilityDensityFunction`.

Listing 9.20 Java implementation of the normal distribution

```
package DhbStatistics;

import DhbFunctionEvaluation.PolynomialFunction;
import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

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

// @author Didier H. Besset

public final class NormalDistribution
                                extends ProbabilityDensityFunction
{

// Average of the distribution.

    private double mu;

// Standard deviation of the distribution.

    private double sigma;

// Constant needed to compute the norm.
```

```
    private static double baseNorm = Math.sqrt( 2 * Math.PI);

//  Series to compute the error function.

    private static PolynomialFunction errorFunctionSeries;
    static     {
        double[] coeffs = { 0.31938153, -0.356563782, 1.781477937,
                                        -1.821255978, 1.330274429};
        errorFunctionSeries = new PolynomialFunction( coeffs);
    };

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

    private static double errorFunctionConstant = 0.2316419;


//  Defines a normalized Normal distribution with average 0
//                                  and standard deviation 1.

public NormalDistribution ( ) throws IllegalArgumentException
{
    this( 0, 1);
}

//  Defines a Normal distribution with known average
//                                  and standard deviation.
//  @param average of the distribution
//  @param standard deviation of the distribution
//  @exception java.lang.IllegalArgumentException
//                      if the standard deviation is non-positive

public NormalDistribution ( double average, double standardDeviation)
                                    throws IllegalArgumentException
{
    if ( standardDeviation <= 0 )
        throw new IllegalArgumentException(
                            "Standard deviation must be positive");
    mu = average;
    sigma = standardDeviation;
}

//  Create an instance of the receiver with parameters estimated from
//  the given histogram using best guesses. This method can be used to
//  find the initial values for a fit.
//  @param h DhbScientificCurves.Histogram
```

```
public NormalDistribution( Histogram h)
{
    this( h.average(), h.standardDeviation());
}

//  @return double average of the distribution.

public double average ( )
{
    return mu;
}

//  Returns the probability of finding a random variable smaller
//  than or equal to x.
//  @return integral of the probability density function from -infinity to x.
//  @param x double upper limit of integral.

public double distributionValue( double x)
{
    return errorFunction( ( x - mu) / sigma);
}

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

public static double errorFunction ( double x)
{
    if ( x == 0 )
        return 0.5;
    else if ( x > 0 )
        return 1 - errorFunction( -x);
    double t = 1 / (1 - errorFunctionConstant * x);
    return t * errorFunctionSeries.value( t) * normal( x);
}

//  @return double kurtosis of the distribution.

public double kurtosis( )
{
    return 0;
}

//  @return java.lang.String     name of the distribution

public String name ( )
{
    return "Normal distribution";
}

//  @return the density probability function for a (0,1) normal distribution.
//  @param x double    value for which the probability is evaluated.

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

//  @return double[]    array containing mu and sigma

public double[] parameters ( )
{
    double[] answer = new double[2];
    answer[0] = mu;
    answer[1] = sigma;
    return answer;
}

//  @return double a random number distributed according to the receiver.

public double random( )
{
    return generator().nextGaussian() * sigma + mu;
}

//  @param average double

public void setAverage( double average)
{
    mu = average;
}

//  @param p double[]    assigns the parameters

public void setParameters( double[] params)
{
    setAverage( params[0]);
    setStandardDeviation( params[1]);
}

//  @param average double

public void setStandardDeviation( double standardDeviation)
```

```
{
    sigma = standardDeviation;
}

//  @return double skewness of the distribution.

public double skewness( )
{
    return 0;
}

//  @return double standard deviation of the distribution

public double standardDeviation( )
{
    return sigma;
}

//  @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    java.text.DecimalFormat fmt = new java.text.DecimalFormat(
                                                         "0.00000");
    sb.append("Normal distribution (");
    sb.append(fmt.format(mu));
    sb.append(',');
    sb.append(fmt.format(sigma));
    sb.append(')');
    return sb.toString();
}

//  @return double probability density function
//  @param x double random variable

public double value( double x)
{
    return normal( ( x - mu) / sigma) / sigma;
}

//  Evaluate the distribution and the gradient of the distribution
//  with respect to the parameters.
//  @return double[]   0: distribution's value, 1,2,...,n distribution's gradient
//  @param x double
```

```
    public double[] valueAndGradient( double x)
    {
        double[] answer = new double[3];
        double y = ( x - mu) / sigma;
        answer[0] = normal( y) / sigma;
        answer[1] = answer[0] * y / sigma;
        answer[2] = answer[0] * ( y * y - 1) / sigma;
        return answer;
    }
    }
```

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 9.3 shows the properties of the gamma distribution. Figure 9.4 shows the shape of the gamma distribution for several values of the parameter α with $\beta = 1$. The shape of the distrubution 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.

TABLE 9.3 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	$F(x) = \left(\frac{x}{\beta}, \alpha\right)$
	(see Section 7.4.1)
Average	$\alpha\beta$
Variance	$\alpha\beta^2$
Skewness	$\frac{2}{\sqrt{\alpha}}$
Kurtosis	$\frac{6}{\alpha}$

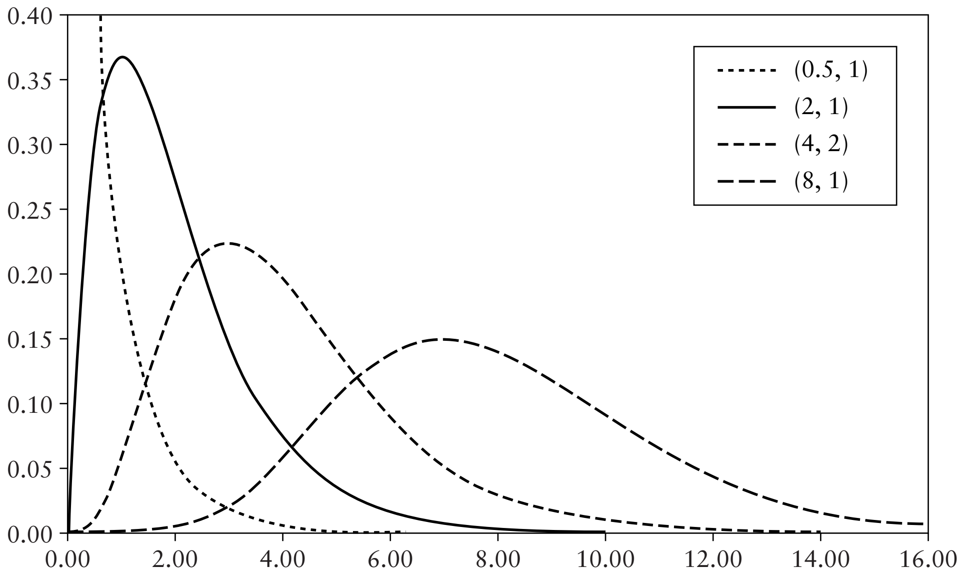

FIG. 9.4 Gamma distribution for various values of α

9.7.1 Gamma Distribution—Smalltalk Implementation

Listing 9.21 shows the implementation of the gamma distribution in Smalltalk.

The distribution function of the gamma distribution can be computed with the incomplete gamma function (see Section 2.4.1). Therefore, the class `DhbGammaDistribution` is implemented as a subclass of `DhbProbabilityDensity`.

Listing 9.21 Smalltalk implementation of the gamma distribution

Class	`DhbGammaDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`alpha beta norm randomCoefficients incompleteGamma-Function`

Class Methods

distributionName

```
^'Gamma distribution'
```

fromHistogram: `aHistogram`

```
| alpha beta |
aHistogram minimum < 0
    ifTrue: [ ^nil].
alpha := aHistogram average.
beta := aHistogram variance / alpha.
^[ self shape: alpha / beta scale: beta] when: ExAll do: [
                                   :signal | signal exitWith: nil]
```

new

```
^self error: 'Illegal creation message for this class'
```

shape: `aNumber1` **scale:** `aNumber2`

```
^super new initialize: aNumber1 scale: aNumber2
```

Instance Methods

average

```
^alpha * beta
```

changeParametersBy: `aVector`

```
alpha := alpha + ( aVector at: 1).
beta := beta + ( aVector at: 2).
self computeNorm.
incompleteGammaFunction := nil.
randomCoefficients := nil.
```

computeNorm

```
norm := beta ln * alpha + alpha logGamma.
```

distributionValue: `aNumber`

```
^self incompleteGammaFunction value: aNumber / beta
```

incompleteGammaFunction

```
incompleteGammaFunction isNil
    ifTrue:
        [incompleteGammaFunction := DhbIncompleteGammaFunction
                                                  shape: alpha].
^incompleteGammaFunction
```

initialize: `aNumber1` **scale:** `aNumber2`

```
( aNumber1 > 0 and: [ aNumber2 > 0])
    ifFalse: [ self error: 'Illegal distribution parameters'].
alpha := aNumber1.
```

```
beta := aNumber2.
self computeNorm.
^self
```

initializeRandomCoefficientsForLargeAlpha

```
| a b q d |
a := 1 / ( 2 * alpha - 1) sqrt.
b := alpha - (4 ln).
q := 1 / a + alpha.
d := 4.5 ln + 1.
^Array with: a with: b with: q with: d
```

initializeRandomCoefficientsForSmallAlpha

```
| e |
e := 1 exp.
^( e + alpha) / e
```

kurtosis

```
^6 / alpha
```

parameters

```
^Array with: alpha with: beta
```

random

```
^( alpha > 1 ifTrue: [ self randomForLargeAlpha]
                    ifFalse:[ self randomForSmallAlpha]) * beta
```

randomCoefficientsForLargeAlpha

```
randomCoefficients isNil
    ifTrue: [ randomCoefficients := self
                        initializeRandomCoefficientsForLargeAlpha].
^randomCoefficients
```

randomCoefficientsForSmallAlpha

```
randomCoefficients isNil
    ifTrue: [ randomCoefficients := self
                        initializeRandomCoefficientsForSmallAlpha].
^randomCoefficients
```

randomForLargeAlpha

```
[ true] whileTrue: [
| u1 u2 c v y z w|
u1 := DhbMitchellMooreGenerator new floatValue.
u2 := DhbMitchellMooreGenerator new floatValue.
c := self randomCoefficientsForLargeAlpha.
v := ( u1 / ( 1 - u1)) ln * (c at: 1).
y := v exp * alpha.
z := u1 squared * u2.
w := ( c at: 3) * v + ( c at: 2) - y.
( c at: 4) + w >= ( 4.5 * z) ifTrue: [ ^y].
z ln <= w ifTrue: [ ^y].
                            ].
```

randomForSmallAlpha

```
[ true] whileTrue: [
| p |
p := DhbMitchellMooreGenerator new floatValue *
                            self randomCoefficientsForSmallAlpha.
p > 1
    ifTrue: [ | y |
                 y := ( ( self randomCoefficientsForSmallAlpha -
                                          p) / alpha) ln negated.
                 DhbMitchellMooreGenerator new floatValue <=
                                       ( y raisedTo: ( alpha - 1))
                    ifTrue: [ ^y].
                ]
    ifFalse: [ | y |
                    y := p raisedTo: ( 1 / alpha).
                 DhbMitchellMooreGenerator new floatValue <=
                                                  ( y negated exp)
                    ifTrue: [ ^y].
                 ].
                            ].
```

skewness

```
^2 / alpha sqrt
```

value: `aNumber`

```
^aNumber > 0
    ifTrue: [ ( aNumber ln * (alpha - 1) - (aNumber / beta) -
                                                    norm) exp]
    ifFalse:[ 0].
```

variance

```
^beta squared * alpha
```

9.7.2 Gamma Distribution—Java Implementation

Listing 9.22 shows the implementation of the gamma distribution in Java.

The distribution function of the gamma distribution can be computed with the incomplete gamma function (see Section 2.4.1). Therefore, the class `GammaDistribution` is implemented as a subclass of `ProbabilityDensityFunction`.

Listing 9.22 Java implementation of the gamma distribution

```
package DhbStatistics;

import DhbIterations.IncompleteGammaFunction;
import DhbFunctionEvaluation.GammaFunction;
import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Gamma distribution.

// @author Didier H. Besset

public class GammaDistribution extends ProbabilityDensityFunction
{

    // Shape parameter of the distribution.

    protected double alpha;

    // Scale parameter of the distribution.

    private double beta;

    // Norm of the distribution (cached for efficiency).

    private double norm;

    // Constants used in random number generator (cached for efficiency).

    private double a;
    private double b;
    private double q;
    private double d;

    // Incomplete gamma function for the distribution (cached for efficiency).

    private IncompleteGammaFunction incompleteGammaFunction;
```

```
// Constructor method (for internal use only).

protected GammaDistribution()
{
}

// Create a new instance of the Gamma distribution with given shape and scale.
// @param shape double shape parameter of the distribution (alpha).
// @param scale double scale parameter of the distribution (beta).
// @exception java.lang.IllegalArgumentException The exception description.

public GammaDistribution ( double shape, double scale)
    throws IllegalArgumentException
{
    if ( shape <= 0 ) throw new IllegalArgumentException(
         "Shape parameter must be positive");
    if ( scale <= 0 ) throw new IllegalArgumentException(
         "Scale parameter must be positive");
    defineParameters( shape, scale);
}

// Create an instance of the receiver with parameters estimated from the
// given histogram using best guesses. This method can be used to
// find the initial values for a fit.
// @param h DhbScientificCurves.Histogram
// @exception java.lang.IllegalArgumentException when no suitable parameter can be
found.

public GammaDistribution( Histogram h) throws IllegalArgumentException
{
    if ( h.getMinimum() < 0 )
        throw new IllegalArgumentException("Gamma distribution is only
           defined for non-negative values");
    double shape = h.average();
    if ( shape <= 0 )
        throw new IllegalArgumentException("Gamma distribution must have
            a non-negative shape parameter");
    double scale = h.variance() / shape;
    if ( scale <= 0 )
        throw new IllegalArgumentException("Gamma distribution must have
            a non-negative scale parameter");
    defineParameters( shape / scale, scale);
}

// @return double average of the distribution.
```

```
public double average()
{
    return alpha * beta;
}

// Assigns new values to the parameters.
// This method assumes that the parameters have been already checked.

public void defineParameters ( double shape, double scale)
{
    alpha = shape;
    beta = scale;
    norm = Math.log( beta) * alpha + GammaFunction.logGamma( alpha);
    if ( alpha < 1)
        b = (Math.E + alpha) / Math.E;
    else if ( alpha > 1)
    {
        a = Math.sqrt( 2 * alpha - 1);
        b = alpha - Math.log( 4.0);
        q = alpha + 1 / a;
        d = 1 + Math.log( 4.5);
    }
    incompleteGammaFunction = null;
    return;
}

// Returns the probability of finding a random variable smaller
// than or equal to x.
// @return integral of the probability density function from 0 to x.
// @param x double upper limit of integral.

public double distributionValue ( double x)
{
    return incompleteGammaFunction().value( x / beta);
}

// @return DhbIterations.IncompleteGammaFunction

private IncompleteGammaFunction incompleteGammaFunction()
{
    if ( incompleteGammaFunction == null )
        incompleteGammaFunction = new IncompleteGammaFunction( alpha);
    return incompleteGammaFunction;
}

// @return double kurtosis of the distribution.
```

```
public double kurtosis( )
{
    return 6 / alpha;
}

//  @return java.lang.String name of the distribution.

public String name()
{
    return "Gamma distribution";
}

//  @return double[] an array containing the parameters of
//                                   the distribution.

public double[] parameters()
{
    double[] answer = new double[2];
    answer[0] = alpha;
    answer[1] = beta;
    return answer;
}

//  @return double a random number distributed according to the receiver.

public double random( )
{
    double r;

    if ( alpha > 1)
        r = randomForAlphaGreaterThan1();
    else if (alpha < 1)
        r = randomForAlphaLessThan1();
    else
        r = randomForAlphaEqual1();

    return r * beta;
}

//  @return double

private double randomForAlphaEqual1( )
{
    return -Math.log( 1 - generator().nextDouble());
}
```

```
// @return double

private double randomForAlphaGreaterThan1( )
{
    double u1, u2, v, y, z, w;
    while ( true)
    {
        u1 = generator().nextDouble();
        u2 = generator().nextDouble();
        v = a * Math.log( u1 / ( 1 - u1));
        y = alpha * Math.exp( v);
        z = u1 * u1 * u2;
        w = b + q * v - y;
        if ( w + d - 4.5 * z >= 0 || w >= Math.log( z) )
            return y;
    }
}

// @return double

private double randomForAlphaLessThan1( )
{
    double p, y;
    while ( true)
    {
        p = generator().nextDouble() * b;
        if ( p > 1)
        {
            y = -Math.log( ( b - p) / alpha);
            if ( generator().nextDouble() <= Math.pow( y, alpha - 1) )
                return y;
        }
        y = Math.pow( p, 1 / alpha);
        if ( generator().nextDouble() <= Math.exp( -y) )
            return y;
    }
}

// @param p double[]   assigns the parameters

public void setParameters( double[] params)
{
    defineParameters( params[0], params[1]);
}

// @return double skewness of the distribution.
```

```
public double skewness( )
{
    return 2 / Math.sqrt( alpha);
}

//  @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    java.text.DecimalFormat fmt = new java.text.DecimalFormat
       ("####0.00000");
    sb.append("Gamma distribution (");
    sb.append(fmt.format(alpha));
    sb.append(',');
    sb.append(fmt.format(beta));
    sb.append(')');
    return sb.toString();
}

//  @return double probability density function
//  @param x double random variable

public double value( double x)
{
    return x > 0 ? Math.exp( Math.log( x) * ( alpha - 1) - x /
       beta - norm) : 0;
}

//  @return double variance of the distribution.

public double variance( )
{
    return alpha * beta * beta;
}
}
```

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 that 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 distribution 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 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 $x < x_{\min}$, $F(x) = 0$, if $x \geq x_{\max}$, $F(x) = 1$. A similar equation can be derived for the acceptance interval function.

9.8.1 Experimental Distribution—General Implementation

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.

Note: In both implementations, there is no protection against illegal arguments. Illegal arguments can occur when computing the distribution value when the histogram underflow and overflow counts are nonzero. 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 (see Section 9.3).

9.8.2 Experimental Distribution—Smalltalk Implementation

Listing 9.23 shows the implementation of an experimental distribution in Smalltalk.

The class `DhbHistogrammedDistribution` is a subclass of the class `DhbProbabilityDensity`. The class creation method `histogram:` takes as argument the histogram over which the instance is defined. To prevent creating an instance with an undefined instance variable, the default class creation method `new` returns an error.

Listing 9.23 Smalltalk implementation of an experimental distribution

Class	`DhbHistogrammedDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`histogram`

Class Methods

distributionName

```
^'Experimental distribution'
```

histogram: `aHistogram`

```
^super new initialize: aHistogram
```

new

```
^self error: 'Illegal creation message for this class'
```

Instance Methods

acceptanceBetween: `aNumber1` **and:** `aNumber2`

```
^( histogram countsBetween: ( aNumber1 max: histogram minimum)
                    and: ( aNumber2 min: histogram maximum) ) /
                                              histogram totalCount
```

average

```
^histogram average
```

distributionValue: `aNumber`

```
^aNumber < histogram minimum
    ifTrue: [ 0]
    ifFalse:[ aNumber < histogram maximum
                    ifTrue: [ ( histogram countsUpTo:
                                  aNumber) / histogram totalCount]
```

```
                         ifFalse:[ 1]
                 ]
```

initialize: `aHistogram`

```
    aHistogram count = 0
        ifTrue: [ self error: 'Cannot define probability density on
                                                  an empty histogram'].
    histogram := aHistogram.
    ^self
```

kurtosis

```
    ^histogram kurtosis
```

privateInverseDistributionValue: `aNumber`

```
    ^histogram inverseDistributionValue: aNumber
```

skewness

```
    ^histogram skewness
```

standardDeviation

```
    ^histogram standardDeviation
```

value: `aNumber`

```
    ^( aNumber >= histogram minimum and: [ aNumber <
                                                    histogram maximum])
        ifTrue: [ ( histogram countAt: aNumber) /
                             ( histogram totalCount * histogram binWidth)]
        ifFalse:[ 0]
```

variance

```
    ^histogram variance
```

9.8.3 Experimental Distribution—Java Implementation

Listing 9.24 shows the implementation of an experimental distribution in Java.

The class `HistogrammedDistribution` is a subclass of the class `Probability-DensityFunction`. The class constructor method takes as argument the histogram over which the instance is defined.

Listing 9.24

Java implementation of an experimental distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;

// Distribution constructed on a histogram.

// @author Didier H. Besset

public class HistogrammedDistribution extends ProbabilityDensityFunction
{
    Histogram histogram;

// @return double average of the histogram.

public double average()
{
    return histogram.average();
}

// Returns the probability of finding a random variable smaller
// than or equal to x.
// @return integral of the probability density function from -infinity to x.
// @param x double upper limit of integral.

public double distributionValue(double x)
{
    if ( x < histogram.getMinimum())
        return 0;
    else if ( x < histogram.getMaximum())
        return histogram.getCountsUpTo( x) / histogram.totalCount();
    else
        return 1;
}

// @return double
// @param x1 double
// @param x2 double

public double distributionValue ( double x1, double x2)
{
    return histogram.getCountsBetween(Math.max(x1,
                                          histogram.getMinimum()),
                                      Math.min(x2,
                                                      histogram.getMaximum()))
                                            / histogram.totalCount();
}

//  @return double kurtosis of the histogram.

public double kurtosis()
{
    return histogram.kurtosis();
}

//  @return java.lang.String name of the distribution.

public String name()
{
    return "Experimental distribution";
}

//  NOTE: this method is a dummy because the distribution
//  cannot be fitted.
//  @return double[] an array containing the parameters of
//                                the distribution.

public double[] parameters()
{
    return new double[0];
}

//  This method is a dummy method, needed for the compiler because
//  the superclass requires implementation of the
//              interface ParametrizedOneVariableFunction.
//  Histogrammed distributions cannot be fitted.

 public void setParameters( double[] params)
{
}

 // @return double skewness of the histogram.

public double skewness()
{
    return histogram.skewness();
}

//  @return double probability density function
//  @param x double random variable
```

```
public double value( double x)
{
    return ( x >= histogram.getMinimum()
                                || x < histogram.getMaximum() )
                ? histogram.getBinContent(x)
                                / (histogram.totalCount()
                                            + histogram.getBinWidth())
                : 0;
}

//  @return double variance of the histogram.

public double variance()
{
    return histogram.variance();
}
}
```