CHAPTER 10

Statistical Analysis

L'expérience instruit plus sûrement que le conseil.[1]
—André Gide

This chapter is describes how to extract information from large amounts of data using statistical analysis. One of the best books I have read on this subject is titled *How to Lie with Statistics*[2]. This admittedly slanted title seems a little pessimistic. The truth, however, is that most people in their daily job ignore the little statistics they have learned in high school. As a result, statistical argumentation is often used wrongly to produce the wrong conclusions.

The problems addressed in this section pertain to the interpretation of experimental measurements. For a long time such a discipline was reserved to physicists only. Recently natural science disciplines discovered that statistics could be used effectively to verify hypotheses or to determine parameters based on experimental observations. Today, the best papers on statistics and estimations are found primarily in natural science publications (e.g., *Biometrika*). The use of statistics recently, has also been extended to financial analysis.

Statistics can be applied to experimental data in two ways. First, one can test the consistency and/or accuracy of the data. These tests are the subject of the first three sections. Second, the values of unknown parameters can be derived from experimental data. This very important aspect of statistical analysis is treated in the remainder of the chapter.

Figure 10.1 shows the classes described in this chapter. The reader should be aware that the techniques used for nonlinear least square fits (Section 10.9) can also be also applied to solve systems of nonlinear equations.

---

1. Experience teaches more surely than counseling.

2. D. Huff, *How to Lie with Statistics*, Norton and Co., New York 1954.

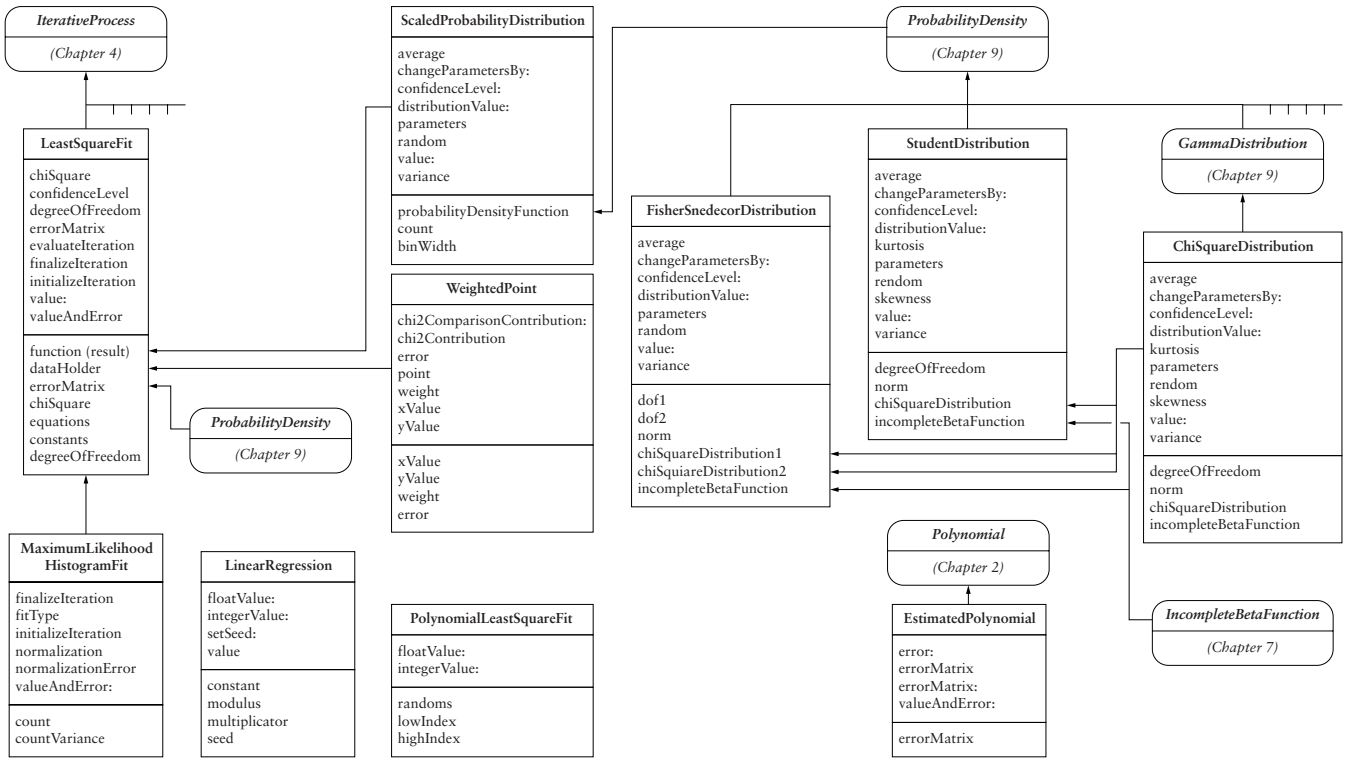

FIG. 10.1 Classes related to statistical analysis

10.1 *F*-test and the Fisher-Snedecor Distribution

The *F*-test tries to answer the following question: given two series of measurements, $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, what is the probability that the two measurements have the same standard deviation? The *F*-test is used when there are not enough statistics to perform a detailed analysis of the data.

Let us assume that the distribution of the two random variables, x and y, are normal distributions with respective averages μ_x and μ_y and respective standard deviations σ_x and σ_y. Then, $\bar{s}_x$, the standard deviation of $x_1, \ldots, x_n$, is an estimator of σ_x; $\bar{s}_y$, the standard deviation of $y_1, \ldots, y_m$, is an estimator of σ_y. The statistics

$$F = \frac{\bar{s}_x^2}{\sigma_x^2} \cdot \frac{\sigma_y^2}{\bar{s}_y^2} \tag{10.1}$$

can be shown to be distributed according to a Fisher-Snedecor distribution with degrees of freedom n and m. In particular, if one wants to test for the equality of the two standard deviations, one constructs the following statistics:

$$F = \frac{\bar{s}_x^2}{\bar{s}_y^2}. \tag{10.2}$$

Traditionally, one chooses $\bar{s}_x > \bar{s}_y$ so that the variable F is always greater than 1.

It is important to recall that this expression is distributed according to a Fisher-Snedecor distribution if and only if the two sets of data are distributed according to a normal distribution. For experimental measurements this is often the case unless systematic errors are present. Nevertheless, this assumption must be verified before making an *F*-test.

Table 10.1 shows the properties of the Fisher-Snedecor distribution. The Fisher-Snedecor distribution is itself rarely used as a probability density function, however.

The main part of Figure 10.2 shows the shape of the Fisher-Snedecor distribution for some values of the degrees of freedom. For large n and m, the Fisher-Snedecor distribution tends toward a normal distribution.

The confidence level of a Fisher-Snedecor distribution is defined as the probability (expressed as a percentage) of finding a value larger than F defined in equation 10.2 or lower than $1/F$. The confidence level is thus related to the distribution function. We have

$$\mathrm{CL}_F(x, n) = 100 \left\{ 1 - \left[F(x) - F\left(\frac{1}{x}\right) \right] \right\}, \tag{10.3}$$

where x is the value F defined in equation 10.2. The change of notation was made to avoid confusion between the acceptance function and the random variable. The confidence level corresponds to the surface of the shaded areas in the insert in the upper-right corner of Figure 10.2.

TABLE 10.1 Properties of the Fisher-Snedecor distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \dfrac{n_1^{\frac{n_1}{2}} n_2^{\frac{n_2}{2}} x^{\frac{n_1-1}{2}}}{B\left(\frac{n_1}{2}, \frac{n_2}{2}\right)(n_1 + n_2 x)^{\frac{n_1+n_2}{2}}}$
Parameters	n_1, n_2, two positive integers
Distribution function	$F(x) = B\left(\dfrac{n_1}{n_1 + n_2 x}; \dfrac{n_1}{2}, \dfrac{n_2}{2}\right)$
Average	$\dfrac{n_2}{n_2 - 2}$ for $n > 2$ undefined otherwise
Variance	$\dfrac{2n_2^2(n_1 + n_2 - 2)}{n_1(n_2 - 2)^2(n_2 - 4)}$ for $n > 4$; undefined otherwise

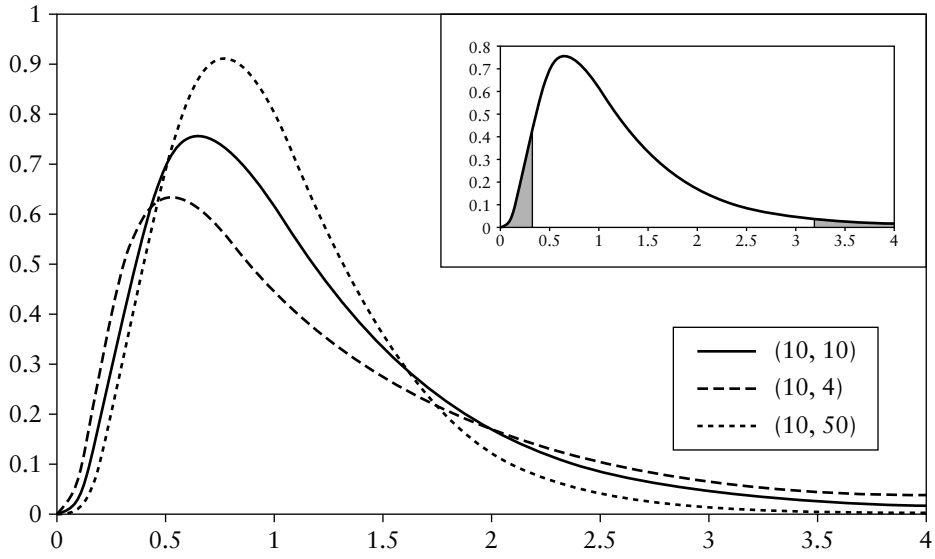

FIG. 10.2 Fisher-Snedecor distribution for a few parameters

Here is an example used to investigate the goodness of the two random number generators discussed in Section 9.4. The collected data are the errors of the covariance test described in Section 9.4. The dimension of the covariance matrix is 7, and the number of trials on each measurement is 1,000. Table 10.2 presents the results, expressed in $\frac{1}{1,000}$. The ratio of the two variances is 1.18, and the degrees of freedom

TABLE 10.2 Covariance test of random number generator

	Congruential	Mitchell-Moore
1	5.56	7.48
2	5.89	6.75
3	4.66	3.77
4	5.69	5.71
5	5.34	7.25
6	4.79	4.73
7	4.80	6.23
8	7.86	5.60
9	3.64	5.94
10	5.70	4.58
Average	5.40	5.80
Std. dev.	1.10	1.19

are both 10. The F-test applied to the two variances gives a confidence level of 20%, which means that there is only a 20% probability that the variances of the two series of measurements are different.

10.1.1 Fisher-Snedecor Distribution—Smalltalk Implementation

Listing 10.1 shows the implementation of the Fisher-Snedecor distribution in Smalltalk. Listing 10.2 shows the implementation of the *F*-test in Smalltalk. Code Example 10.1 shows how to perform an *F*-test between two sets of experimental measurements.

Code Example 10.1

```
| mom1 mom2 confidenceLevel |
mom1 := DhbFixedStatisticalMoments new.
   .
   .   <Collecting measurements of set 1 into mom1>
   .
mom2 := DhbFixedStatisticalMoments new.
   .
   .   <Collecting measurements of set 2 into mom1>
   .
confidenceLevel := mom1 fConfidenceLevel: mom2.
```

Two instances of statistical moments (see Section 9.2.2) are created. Experimental data are accumulated into each set separately (see Code Example 9.1). The last line returns the probability as a percentage that the two sets of data have the same standard deviation.

The class DhbFisherSnedecorDistribution is implemented as a subclass of DhbProbabilityDensity because its distribution function can be computed numerically using the incomplete beta function (see Section 7.5).

Listing 10.1 Smalltalk implementation of the Fisher-Snedecor distribution

Class	DhbFisherSnedecorDistribution
Subclass of	DhbProbabilityDensity
Instance variable names:	dof1 dof2 norm chiSquareDistribution1 chiSquareDistribution2 incompleteBetaFunction

Class Methods

degreeOfFreedom: anInteger1 **degreeOfFreedom:**

anInteger2

```
^super new initialize: anInteger1 and: anInteger2
```

distributionName

```
^'Fisher-Snedecor distribution'
```

fromHistogram: aHistogram

```
| n1 n2 a |
aHistogram minimum < 0 ifTrue: [^nil].
n2 := (2 / (1 - (1 / aHistogram average))) rounded.
n2 > 0 ifFalse: [^nil].
a := (n2 - 2) * (n2 - 4) * aHistogram variance / (n2 squared *
                                                        2).
n1 := (0.7 * (n2 - 2) / (1 - a)) rounded.
^n1 > 0
    ifTrue: [self degreeOfFreedom: n1 degreeOfFreedom: n2]
    ifFalse: [nil]
```

new

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

test: aStatisticalMoment1 **with:**

aStatisticalMoment2

```
^(self class degreeOfFreedom: aStatisticalMoment1 count
    degreeOfFreedom: aStatisticalMoment2 count)
        distributionValue: aStatisticalMoment1 variance
                / aStatisticalMoment2 variance
```

Instance Methods

average

```
^dof2 > 2
    ifTrue: [ dof2 / ( dof2 - 2)]
    ifFalse:[ nil]
```

changeParametersBy: aVector

```
dof1 := ( dof1 + ( aVector at: 1)) max: 1.
dof2 := ( dof2 + ( aVector at: 2)) max: 1.
self computeNorm.
chiSquareDistribution1 := nil.
chiSquareDistribution2 := nil.
incompleteBetaFunction := nil.
```

computeNorm

```
norm := ( dof1 ln * ( dof1 / 2) ) + ( dof2 ln * ( dof2 / 2) )
                    - ( ( dof1 / 2) logBeta: ( dof2 / 2) ).
```

confidenceLevel: aNumber

```
aNumber < 0
    ifTrue: [ self error: 'Confidence level argument must be
                                                        positive'].
^( ( self distributionValue: aNumber) - ( self distributionValue:
                                        aNumber reciprocal) ) * 100
```

distributionValue: aNumber

```
^1 - ( self incompleteBetaFunction value: ( dof2 / ( aNumber *
                                                    dof1 + dof2)))
```

incompleteBetaFunction

```
incompleteBetaFunction isNil
    ifTrue:
        [incompleteBetaFunction := DhbIncompleteBetaFunction
                                                    shape: dof2 / 2
                    shape: dof1 / 2].
^incompleteBetaFunction
```

initialize: anInteger1 **and:** anInteger2

```
dof1 := anInteger1.
dof2 := anInteger2.
self computeNorm.
^self
```

parameters

```
^Array with: dof1 with: dof2
```

random

```
chiSquareDistribution1 isNil
    ifTrue: [ chiSquareDistribution1 := DhbChiSquareDistribution
                                            degreeOfFreedom: dof1.
              chiSquareDistribution2 := DhbChiSquareDistribution
                                            degreeOfFreedom: dof2.
            ].
^chiSquareDistribution1 random * dof2 / ( chiSquareDistribution2
                                                    random * dof1)
```

value: `aNumber`

```
^aNumber > 0
    ifTrue: [ ( norm + ( aNumber ln * ( dof1 / 2 - 1) ) - (
         (aNumber * dof1 + dof2) ln * ( ( dof1 + dof2) / 2))) exp]
    ifFalse:[ 0]
```

variance

```
^dof2 > 4 ifTrue: [ dof2 squared * 2 * ( dof1 + dof2 - 2) / ( (
                          dof2 - 2) squared * dof1 * ( dof2 - 4))]
               ifFalse:[ nil]
```

The computation of the confidence level for the *F*-test is implemented in the method `fConfidenceLevel:` of the class `DhbStatisticalMoments`. It calculates the *F* statistics according to equation 10.2, creates an instance of a Fisher-Snedecor distribution, and passes the value of *F* to the method `confidenceLevel:` of the distribution. The method `fConfidenceLevel:` is also implemented by the class `Histogram` where it is simply delegated to the statistical moments accumulated by the histogram. The argument of the method can be a statistical moment or a histogram since the messages sent by the method are polymorphic to both classes.

Listing 10.2 Smalltalk implementation of the *F*-test

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

Instance Methods

fConfidenceLevel: `aStatisticalMomentsOrHistogram`

```
| fValue |
fValue := self variance/ aStatisticalMomentsOrHistogram variance.
^fValue < 1
    ifTrue: [ (DhbFisherSnedecorDistribution degreeOfFreedom:
                              aStatisticalMomentsOrHistogram count
                    degreeOfFreedom: self count)
                                    confidenceLevel: fValue
                                                      reciprocal]
    ifFalse:[ (DhbFisherSnedecorDistribution degreeOfFreedom:
                                                       self count
                    degreeOfFreedom:
                            aStatisticalMomentsOrHistogram count)
                                    confidenceLevel: fValue]
```

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

Instance Methods

fConfidenceLevel: `aStatisticalMomentsOrHistogram`

```
^moments fConfidenceLevel: aStatisticalMomentsOrHistogram
```

10.1.2 Fisher-Snedecor Distribution—Java Implementation

Listing 10.3 shows the implementation of the Fisher-Snedecor distribution in Java. Code Example 10.2 shows how to perform an *F*-test between two sets of experimental measurements.

Code Example 10.2

```
StatisticalMoments mom1 = new StatisticalMoments();
    :
    : <Collecting measurements of set 1 into mom1>
StatisticalMoments mom2 = new StatisticalMoments();
    :
    : <Collecting measurements of set 2 into mom1¿
double confidenceLevel = mom1.fConfidenceLevel( mom2);
```

Two instances of statistical moments (see Section 9.2.3) are created. Experimental data are accumulated into each set separately (see Code Example 9.2). The last

line returns the probability as a percentage that the two sets of data have the same standard deviation.

Listing 10.3 Java implementation of the Fisher-Snedecor distribution

```
package DhbStatistics;

import DhbFunctionEvaluation.GammaFunction;
import DhbIterations.IncompleteBetaFunction;
import DhbScientificCurves.Histogram;

// Fisher-Snedecor distribution
// (distribution used to perform the F-test).

// @author Didier H. Besset

public final class FisherSnedecorDistribution
                                    extends ProbabilityDensityFunction
{

    // First degree of freedom.

    protected int dof1;

    // Second degree of freedom.

    private int dof2;

    // Norm (stored for efficiency).

    private double norm;

    // Function used to compute the distribution.

    private IncompleteBetaFunction incompleteBetaFunction = null;

    // Auxiliary distributions for random number generation.

    private ChiSquareDistribution chiSquareDistribution1 = null;
    private ChiSquareDistribution chiSquareDistribution2 = null;

// Create a new instance of the Fisher-Snedecor distribution with
//                                  given degrees of freedom.
// @param n1 int   first degree of freedom
// @param n2 int   second degree of freedom
// @exception java.lang.IllegalArgumentException
//         one of the specified degrees of freedom is non-positive.
```

```
public FisherSnedecorDistribution( int n1, int n2)
                                    throws IllegalArgumentException
{
    if ( n1 <= 0 )
        throw new IllegalArgumentException(
                        "First degree of freedom must be positive");
    if ( n2 <= 0 )
        throw new IllegalArgumentException(
                        "Second degree of freedom must be positive");
    defineParameters( n1, n2);
}

// 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 Histogram
// @exception java.lang.IllegalArgumentException
//                  when no suitable parameter can be found.

public FisherSnedecorDistribution( Histogram h)
                                        throws
                                        IllegalArgumentException
{
    if ( h.getMinimum() < 0 )
        throw new IllegalArgumentException(
"Fisher-Snedecor distribution is only defined for non-negative values");
    int n2 = (int) Math.round(2 / (1 - 1 / h.average()));
    if ( n2 <= 0 )
        throw new IllegalArgumentException(
    "Fisher-Snedecor distribution has positive degrees of freedom");
    double a = 1 - ( n2 - 2) * ( n2 - 4) * h.variance() / (2 * 2 * n2);
    int n1 = (int) Math.round( 0.7 * ( n2 - 2) / a);
    if ( n1 <= 0 )
        throw new IllegalArgumentException(
    "Fisher-Snedecor distribution has positive degrees of freedom");
    defineParameters( n1, n2);
}

// @return double average of the distribution.

public double average()
{
    return dof2 > 2
            ? dof2 / ( dof2 - 2)
            : Double.NaN;
}
```

```
// @return double
// @param x double
// @exception java.lang.IllegalArgumentException
//                 if the argument is outside the expected range.

public double confidenceLevel( double x)
                                    throws IllegalArgumentException
{
    return x < 0 ? Double.NaN
                 : distributionValue( x) * 100;
}

// Assigns new degrees of freedom to the receiver.
// Compute the norm of the distribution after a change of parameters.
// @param n1 int    first degree of freedom
// @param n2 int    second degree of freedom

public void defineParameters ( int n1, int n2)
{
    dof1 = n1;
    dof2 = n2;
    double nn1 = 0.5 * n1;
    double nn2 = 0.5 * n2;
    norm = nn1 * Math.log( n1) + nn2 * Math.log( n2)
                               - GammaFunction.logBeta( nn1, nn2);
    incompleteBetaFunction = null;
    chiSquareDistribution1 = null;
    chiSquareDistribution2 = null;
}

// 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 incompleteBetaFunction().value( dof2 / ( x * dof1 + dof2));
}
private IncompleteBetaFunction incompleteBetaFunction()
{
    if ( incompleteBetaFunction == null )
        incompleteBetaFunction = new IncompleteBetaFunction(
                                            0.5 * dof1, 0.5 * dof2);
    return incompleteBetaFunction;
}
```

```
// @return java.lang.String    name of the distribution.

public String name ( )
{
    return "Fisher-Snedecor distribution";
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters ( )
{
    double[] answer = new double[2];
    answer[0] = dof1;
    answer[1] = dof2;
    return answer;
}

// @return double a random number distributed according to the receiver.

public double random( )
{
    if ( chiSquareDistribution1 == null )
    {
        chiSquareDistribution1 = new ChiSquareDistribution( dof1);
        chiSquareDistribution2 = new ChiSquareDistribution( dof2);
    }
    return chiSquareDistribution1.random() * dof2
            / ( chiSquareDistribution2.random() * dof1);
}

// This function cannot be fitted because the parameters are integers.
// @param p double[]    assigns the parameters

public void setParameters( double[] params)
{
    defineParameters( Math.round( (float) params[0]),
                                    Math.round( (float) params[1]));
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append("Fisher-Snedecor distribution (");
```

```
        sb.append(dof1);
        sb.append(',');
        sb.append(dof2);
        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( norm + Math.log( x) * ( dof1 / 2 - 1)
                                - Math.log( x * dof1 + dof2)
                                                * ( ( dof1 + dof2) / 2))
                : 0;
    }

    // @return double variance of the distribution.

    public double variance ( )
    {
        return dof2 > 4
                ? dof2 * dof2 * 2 * ( dof1 + dof2 + 2)
                                    / ( dof1 * ( dof2 - 2) * ( dof2 - 4))
                : Double.NaN;
    }
    }
```

Note: The method `fConfidenceLevel` for classes `StatisticalMoments` and `Histogram` are part of Listings 9.5 and 9.8, respectively.

10.2 *t*-test and the Student Distribution

The *t*-test tries to answer the following question: given two series of measurements, $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, what is the probability that the two measurements have the same average? The *t*-test is used when there are not enough statistics to perform a detailed analysis of the data.

Let us assume that the distribution of the two random variables, x and y, are normal distributions with respective averages μ_x and μ_y and the same standard deviation σ. Then $\bar{x}$, the average of $x_1, \ldots, x_n$, is an estimator of μ_x; $\bar{y}$, the average of $y_1, \ldots, y_m$, is an estimator of μ_y. An estimation $\bar{s}$ of the standard deviation σ can be made using both measurement samples. We have

$$\bar{s}^2 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})^2 + \sum_{i=1}^{m} (y_i - \bar{y})^2}{n + m - 2}. \tag{10.4}$$

One can prove that the following statistic,

$$t = \frac{(\bar{x} - \bar{y}) - (\mu_x - \mu_y)}{\bar{s}\sqrt{\frac{1}{n} + \frac{1}{m}}}, \tag{10.5}$$

is distributed according to a Student distribution with $n + m - 2$ degrees of freedom. In particular, to test for the probability that the two series of measurements have the same average, one uses the following statistics:

$$t = \frac{\bar{x} - \bar{y}}{\bar{s}\sqrt{\frac{1}{n} + \frac{1}{m}}}. \tag{10.6}$$

It is important to recall the two fundamental hypotheses that have been made so far:

1. The two sets of data must be distributed according to a normal distribution.
2. The two sets of data must have the same standard deviation.

Too many people use the *t*-test without first checking the assumptions. Assumption 1 is usually fulfilled with experimental measurements in the absence of systematic errors. Assumption 2, however, must be checked using, for example, the F-test discussed in Section 10.1.

Because the random variable of the distribution is traditionally labeled t, this distribution is often called the *t-distribution*. Table 10.3 shows the properties of the Student distribution. The Student distribution is itself rarely used as a probability density function, however.

For $n = 1$, the Student distribution is identical to a Cauchy distribution with $\mu = 0$ and $\beta = 1$. For large n, the Student distribution tends toward a normal distribution with an average of 0 and a variance of 1. The main part of Figure 10.3 shows the shapes of the Student distribution for a few values of the degrees of freedom. The normal distribution is also given for comparison.

The confidence level of a Student distribution is defined as the probability to find a value whose absolute value is larger than a given value. Thus, it estimates the level of confidence that the hypothesis—namely, that the two sets of measurements have the same average—cannot be accepted. Traditionally, the confidence level is given as a percentage. The confidence level corresponds to the surface of shaded area in the insert in the upper-left corner of Figure 10.3. By definition, the confidence level is related to the interval acceptance function:

$$\mathrm{CL}_t(t, n) = 100\left[1 - F(-|t|, |t|)\right], \tag{10.7}$$

using the definition of the interval acceptance function (equation 9.31). The value of t in equation 10.7 is obtained from equation 10.6.

TABLE 10.3 Properties of the Student distribution

Range of random variable	$]-\infty, +\infty[$
Probability density function	$P(x) = \frac{1}{\sqrt{n}B\left(\frac{n}{2}, \frac{1}{2}\right)} \left(1 + \frac{t^2}{n}\right)^{-\frac{n+1}{2}}$
Parameters	n, a positive integer
Distribution function	$F(x) = \begin{cases} \frac{1 + B\left(\frac{n}{n+x^2}; \frac{n}{2}, \frac{1}{2}\right)}{2} & \text{for } x \geq 0 \\ \frac{1 - B\left(\frac{n}{n+x^2}; \frac{n}{2}, \frac{1}{2}\right)}{2} & \text{for } x < 0 \end{cases}$
Average	0
Variance	$\frac{n}{n-2}$ for $n > 2$; undefined otherwise
Skewness	0
Kurtosis	$\frac{6}{n-4}$ for $n > 4$; undefined otherwise

The distribution function of the Student distribution is calculated with the incomplete beta function (see Section 7.5). Using the fact that the distribution is symmetrical, one can derive the expression

$$F(-|t|, |t|) = B\left(\frac{n}{n+t^2}; \frac{n}{2}, \frac{1}{2}\right), \tag{10.8}$$

from the properties of the distribution (see Table 10.3) and using equations 10.7 and 7.12.

Now, we shall continue the analysis of the results of Table 10.2. The t value computed from the two sets of measurements is 0.112 for a degree of freedom of 18. The corresponding confidence level is 8.76%. That is, there is only a 8.76% probability that the two generators have a different behavior. Thus, we can conclude that the Mitchell-Moore random generator is as good as the congruential random generator.

10.2.1 Student Distribution—Smalltalk Implementation

Listing 10.4 shows the implementation of the Student distribution in Smalltalk. Listing 10.5 shows the implementation of the t-test in Smalltalk. Performing a t-test between two sets of experimental measurements is very similar to performing an F-test. In Code Example 10.1, it suffices to replace the last line with the following:

```
confidenceLevel := mom1 fConfidenceLevel: mom2.
```

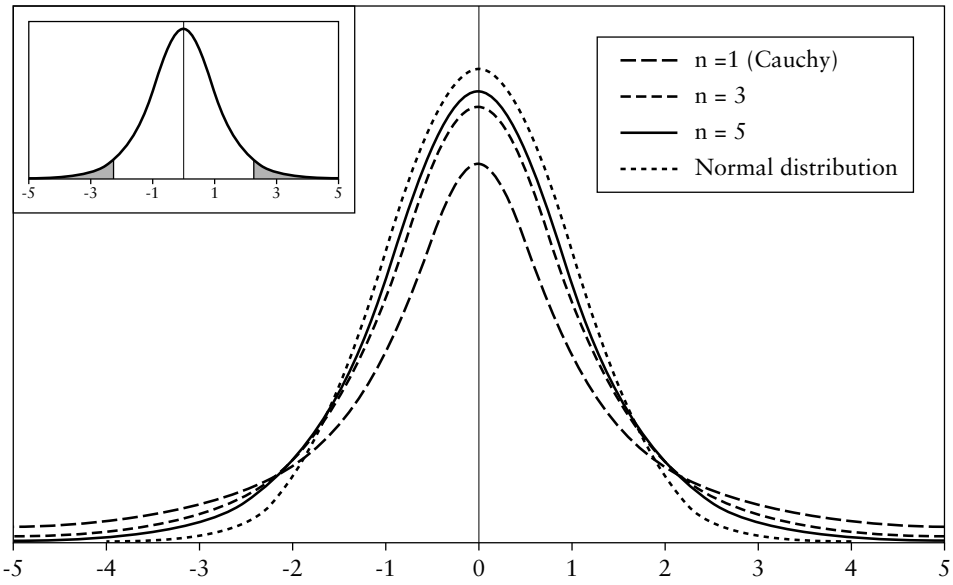

FIG. 10.3 Student distribution for a few degrees of freedom

This last line returns the probability as a percentage that the two sets of data have the same average provided that the two sets have the same standard deviation.

The class `DhbStudentDistribution` is implemented as a subclass of `DhbProbabilityDensity` because its distribution function can be computed numerically using the incomplete beta function (see Section 7.5).

The method `symmetricAcceptance:` computes the symmetrical acceptance function defined by equation 10.8. This method is used to compute the distribution function and the confidence level. The method `confidenceLevel:` gives the confidence level as a percentage.

Listing 10.4 Smalltalk implementation of the Student distribution

Class	`DhbStudentDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`degreeOfFreedom norm chiSquareDistribution incompleteBetaFunction`

Class Methods

asymptoticLimit

```
    ^30
```

degreeOfFreedom: anInteger

```
^anInteger > self asymptoticLimit
    ifTrue: [DhbNormalDistribution new]
    ifFalse:
        [anInteger = 1
            ifTrue: [DhbCauchyDistribution shape: 0 scale: 1]
            ifFalse: [super new initialize: anInteger]]
```

distributionName

```
^'Student distribution'
```

fromHistogram: aHistogram

```
| dof var |
var := aHistogram variance.
var = 0
    ifTrue: [ ^nil].
dof :=  ( 2 / (1 - (1 / aHistogram variance))) rounded max: 1.
^dof > self asymptoticLimit ifTrue: [ nil]
                            ifFalse:[ self degreeOfFreedom: dof]
```

new

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

test: aStatisticalMoment1 **with:**

aStatisticalMoment2

```
  | t |
  t := ( aStatisticalMoment1 average - aStatisticalMoment2 average)
                                                                abs.
  ^1 - ( ( self class degreeOfFreedom: (  aStatisticalMoment1 count
+ aStatisticalMoment2 count - 2)) acceptanceBetween: t negated and:
t)
```

Instance Methods

average

```
^0
```

changeParametersBy: aVector

```
degreeOfFreedom := degreeOfFreedom + ( aVector at: 1).
self computeNorm.
```

chiSquareDistribution

```
chiSquareDistribution isNil
    ifTrue: [ chiSquareDistribution := DhbChiSquareDistribution
                            degreeOfFreedom: (degreeOfFreedom - 1)].
^chiSquareDistribution
```

computeNorm

```
norm := ( ( degreeOfFreedom / 2 logBeta: ( 1 / 2) ) + (
                                    degreeOfFreedom ln / 2)) negated.
```

confidenceLevel: aNumber

```
^( 1 - ( self symmetricAcceptance: aNumber abs)) * 100
```

distributionValue: aNumber

```
aNumber = 0
    ifTrue: [ ^0.5].
^( aNumber > 0
    ifTrue: [ 2 - ( self symmetricAcceptance: aNumber abs)]
    ifFalse:[ self symmetricAcceptance: aNumber abs]) / 2
```

incompleteBetaFunction

```
incompleteBetaFunction isNil
    ifTrue:
        [incompleteBetaFunction := DhbIncompleteBetaFunction
                    shape: degreeOfFreedom / 2
                    shape: 0.5].
^incompleteBetaFunction
```

initialize: anInteger

```
anInteger > 0
    ifFalse: [ self error: 'Degree of freedom must be positive'].
degreeOfFreedom := anInteger.
self computeNorm.
^self
```

kurtosis

```
^degreeOfFreedom > 4 ifTrue: [ 6 / ( degreeOfFreedom - 4)]
                     ifFalse:[ nil]
```

parameters

```
^Array with: degreeOfFreedom
```

random

```
^DhbNormalDistribution random * ( ( (degreeOfFreedom - 1) / self
                                chiSquareDistribution random ) sqrt)
```

skewness

```
^0
```

symmetricAcceptance: aNumber

```
^ self incompleteBetaFunction value: ( degreeOfFreedom / (
                                 aNumber squared + degreeOfFreedom))
```

value: aNumber

```
^( norm - ( ( aNumber squared / degreeOfFreedom + 1) ln * ( (
                                    degreeOfFreedom + 1) / 2))) exp
```

variance

```
^degreeOfFreedom > 2 ifTrue: [ degreeOfFreedom / (
                                                degreeOfFreedom - 2)]
                      ifFalse:[ nil]
```

The computation of the confidence level for the *t*-test is implemented in the method `tConfidenceLevel:` of the class `DhbStatisticalMoments`. It calculates the *t* statistics according to equation 10.6, creates an instance of a Student distribution, and passes the value of *t* to the method `confidenceLevel:` of the distribution. The method `tConfidenceLevel:` is also implemented by the class `Histogram` where it is simply delegated to the statistical moments accumulated by the histogram. The argument of the method can be a statistical moment or a histogram since the messages sent by the method are polymorphic to both classes.

The method `unnormalizedVariance` of class `DhbStatisticalMoments` corresponds to each sum in the numerator of equation 10.4. To allow performing a *t*-test also with instances of class `DhbFastStatisticalMoments`, it was necessary to define this for that class.

Listing 10.5 Smalltalk implementation of the *t*-test

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

Instance Methods

tConfidenceLevel: aStatisticalMomentsOrHistogram

```
| sbar dof |
dof := self count + aStatisticalMomentsOrHistogram count - 2.
sbar := ( ( self unnormalizedVariance +
 aStatisticalMomentsOrHistogram unnormalizedVariance) / dof) sqrt.
^( DhbStudentDistribution degreeOfFreedom: dof)
    confidenceLevel: ( self average -
                         (aStatisticalMomentsOrHistogram average))
                       / ( ( 1 / self count + (
                aStatisticalMomentsOrHistogram count)) sqrt * sbar)
```

Class	DhbFastStatisticalMoments
Subclass of	DhbStatisticalMoments

Instance Methods

unnormalizedVariance

```
^(moments at: 3) - ((moments at: 2) squared * self count)
```

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

Instance Methods

tConfidenceLevel: aStatisticalMomentsOrHistogram

```
^moments tConfidenceLevel: aStatisticalMomentsOrHistogram
```

unnormalizedVariance

```
^moments unnormalizedVariance
```

10.2.2 Student Distribution—Java Implementation

Listing 10.6 shows the implementation of the Student distribution in Java. Performing a *t*-test between two sets of experimental measurements is very similar to performing an *F*-test. In Code Example 10.2, it suffices to replace the last line with the following:

```
confidenceLevel = mom1.tConfidenceLevel( mom2);
```

This last line returns the probability as a percentage that the two sets of data have the same average provided that the two sets have the same standard deviation.

The class `StudentDistribution` is implemented as a subclass of `ProbabilityDensityFunction` because its distribution function can be computed numerically using the incomplete beta function (see Section 7.5).

The private method `symmetricAcceptance` computes the symmetrical acceptance function defined by equation 10.8. This method is used to compute the distribution function and the confidence level. The method `confidenceLevel` gives the confidence level as a percentage.

Listing 10.6 Java implementation of the Student distribution

```
package DhbStatistics;

import DhbFunctionEvaluation.GammaFunction;
import DhbIterations.IncompleteBetaFunction;
import DhbScientificCurves.Histogram;

// Student distribution
// used in computing the t-test.

// @author Didier H. Besset

public final class StudentDistribution
                                        extends ProbabilityDensityFunction
{

    // Degree of freedom.

    protected int dof;

    // Norm (stored for efficiency).

    private double norm;

    // Function used to compute the distribution.

    private IncompleteBetaFunction incompleteBetaFunction = null;

    // Auxiliary distribution for random number generation.

    private ChiSquareDistribution chiSquareDistribution = null;

// Constructor method.
// @param n int    degree of freedom
```

```
// @exception java.lang.IllegalArgumentException\
//          when the specified degree of freedom is non-positive.

public StudentDistribution( int n) throws IllegalArgumentException
{
    if ( n <= 0 )
        throw new IllegalArgumentException(
                                "Degree of freedom must be positive");
    defineParameters( n);
}

// 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 StudentDistribution( Histogram h)
{
    double variance = h.variance();
    if ( variance <= 0 )
        throw new IllegalArgumentException(
        "Student distribution is only defined for positive variance");
    defineParameters( (int) Math.max( 1,
                                Math.round( 2 / (1 - 1 / variance))));
}

// @return double average of the distribution.

public double average()
{
    return 0;
}

// @return double
// @param x double
// @exception java.lang.IllegalArgumentException
//                          if the argument is illegal.

public double confidenceLevel( double x)
                                    throws IllegalArgumentException
{
    return x < 0
            ? Double.NaN
            : symmetricAcceptance( x) * 100;
```

```
}

// @param n int    degree of freedom

public void defineParameters ( int n)
{
    dof = n;
    norm = -( Math.log( dof) * 0.5
                            + GammaFunction.logBeta( dof * 0.5, 0.5));
}

// 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 == 0 )
        return 0.5;
    double acc = symmetricAcceptance( Math.abs( x));
    return x > 0 ? 1 + acc : 1 - acc;
}

// @return DhbIterations.IncompleteBetaFunction

private IncompleteBetaFunction incompleteBetaFunction( )
{
    if ( incompleteBetaFunction == null)
        incompleteBetaFunction = new IncompleteBetaFunction( dof / 2,
                                                                0.5);
    return incompleteBetaFunction;
}

// @return double kurtosis of the distribution.

public double kurtosis( )
{
    return dof > 4 ? 6 / ( dof - 4) : Double.NaN;
}

// @return java.lang.String the name of the distribution.

public String name()
{
    return "Student distribution";
```

```
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters()
{
    double[] answer = new double[1];
    answer[0] = dof;
    return answer;
}

// @return double a random number distributed according to the receiver.

public double random( )
{
    if ( chiSquareDistribution == null )
    {
        chiSquareDistribution = new ChiSquareDistribution( dof - 1);
    }
    return generator().nextGaussian() * Math.sqrt( ( dof - 1)
    / chiSquareDistribution.random());
}

// This distribution cannot be fitted because the parameter is an integer.
// @param p double[]    assigns the parameters

public void setParameters( double[] params)
{
    defineParameters( Math.round( (float) params[0]));
}

// @return double skewness of the distribution.

public double skewness( )
{
    return 0;
}

// @return double    integral from -x to x
// @param x double

private double symmetricAcceptance( double x)
{
    return incompleteBetaFunction().value( dof / ( x * x + dof));
}
```

```
    // @return double probability density function
    // @param x double random variable

    public double value( double x)
    {
        return Math.exp( norm - Math.log( x * x / dof + 1) * ( dof + 1) / 2);
    }

    // @return double variance of the distribution.

    public double variance ( )
    {
        return dof > 2 ? dof / ( dof - 2) : Double.NaN;
    }
    }
```

Note: The method `tConfidenceLevel` for classes `StatisticalMoments` and `Histogram` are part of Listings 9.5 and 9.8, respectively.

10.3 χ^2-test and χ^2 Distribution

The χ^2-test tries to answer the following question: how well a theory is able to predict observed results? Alternatively, a χ^2-test can also tell whether two independent sets of observed results are compatible. This latter formulation is less frequently used than the former. Admittedly, these two questions are somewhat vague. We shall now put them in mathematical terms for a more precise definition.

Let us assume that the measurement of an observable quantity depends on some parameters. These parameters cannot be adjusted by the experimenter but can be measured exactly.[3] Let x_p be the measured values of the observed quantity where p is a label for the parameters; let σ_p be the standard deviation of x_p.

The first question assumes that one can predict the values of the observed quantity: let μ_p be the predicted value of x_p. Then the quantity

$$y_p = \frac{x_p - \mu_p}{\sigma_p} \tag{10.9}$$

is distributed according to a normal distribution with an average of zero and a standard deviation of 1 if and only if the quantities x_p are distributed according to a normal distribution with average μ_p and standard deviation σ_p.

A χ^2 distribution with n degrees of freedom describes the distribution of the sum of the squares of n random variables distributed according to a normal distribution

3. Of course, there is no such thing as an exact measurement. The measurement of the parameters must be far more precise than that of the observed quantity.

with a mean of zero and a standard deviation of 1. Thus, the quantity

$$S = \sum_p \frac{(x_p - \mu_p)^2}{\sigma_p^2} \tag{10.10}$$

is distributed according to a χ^2 distribution with n degrees of freedom where n is the number of available measurements (i.e., the number of terms in the sum of equation 10.10).

To formulate the second question, one must introduce a second set of measurements of the same quantity and at the same values of the parameters. Let x'_p be the second set of measured values; and σ'_p, the corresponding standard deviations. The estimated standard deviation for the difference $x_p - x'_p$ is $\sqrt{\sigma_p^2 + \sigma_p'^2}$. If the two sets of measurements are compatible, the quantity

$$y'_p = \frac{x_p - x'_p}{\sqrt{\sigma_p^2 + \sigma_p'^2}} \tag{10.11}$$

is distributed according to a normal distribution with an average 0 and standard deviation 1. Then the quantity

$$S = \sum_p \frac{\left(x_p - x'_p\right)^2}{\sigma_p^2 + \sigma_p'^2 2} \tag{10.12}$$

is distributed according to a χ^2 distribution with n degrees of freedom.

Table 10.4 shows the properties of the χ^2 distribution. It is a special case of the gamma distribution with $\alpha = \frac{n}{2}$ and $\beta = 2$ (see Section 9.7). For $n > 30$, one can

TABLE 10.4 Properties of the χ^2 distribution

Range of random variable	$[0, +\infty[$
*[1ex] Probability density function	$P(x) = \frac{x^{\frac{n}{2}-1} e^{-\frac{x}{2}}}{2^{\frac{n}{2}} \Gamma\left(\frac{n}{2}\right)}$
Parameters	n, a positive integer
Distribution function	$F(x) = \Gamma\left(\frac{x}{2}; \frac{n}{2}\right)$
Average	n
Variance	$2n$
Skewness	$2\sqrt{\frac{2}{n}}$
Kurtosis	$\frac{12}{n}$

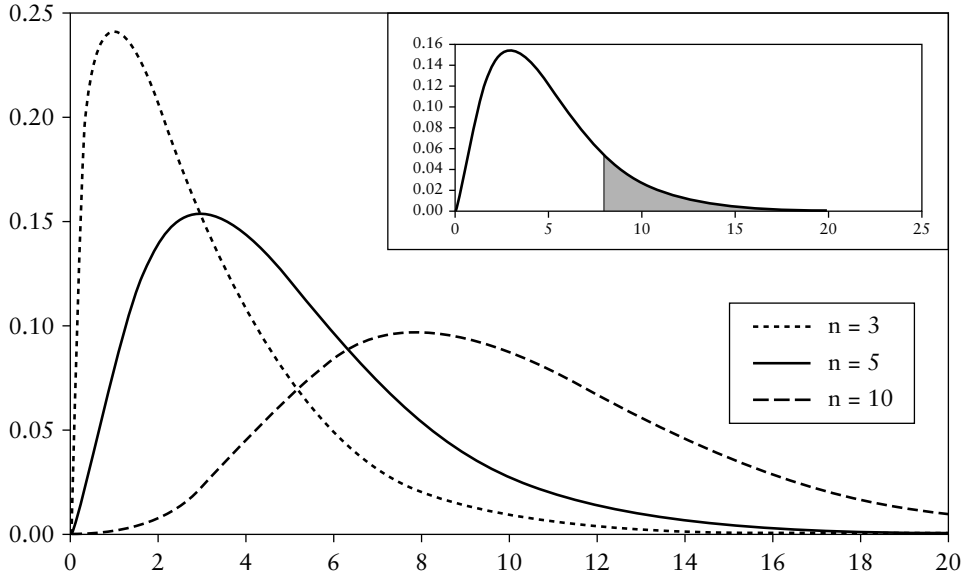

FIG. 10.4 χ^2 distribution for a few degrees of freedom

prove that the variable $y = \sqrt{2x} - \sqrt{2n-1}$ is approximately distributed according to a normal distribution with an average of zero and a standard deviation of 1. If n is very large the χ^2 distribution tends toward a normal distribution with an average of n and a standard deviation of $\sqrt{2n}$.

Figure 10.4 shows the shape of the χ^2 distribution for a few values of the degree of freedom.

To perform a χ^2-test, it is customary to evaluate the probability of finding a value larger than that obtained in equations 10.10 or 10.12. In this form, the result of a χ^2-test gives the probability that the set of measurements is *not* compatible with the prediction or with another set of measurements. The confidence level of a χ^2 value is defined as the probability of finding a value larger than χ^2 expressed as a percentage. It is thus related to the distribution function as follows:

$$\mathrm{CL}_S = 100\left[1 - F\left(S\right)\right], \tag{10.13}$$

where S is the quantity defined in equation 10.10 or 10.12. The confidence level corresponds to the surface of the shaded area of the insert in the upper right corner of Figure 10.4.

Since the χ^2 distribution is a special case of the gamma distribution, the confidence level can be expressed with the incomplete gamma function (see Section 7.4):

$$\mathrm{CL}_S = 100\left[1 - \Gamma\left(\frac{S}{2}, \frac{n}{2}\right)\right]. \tag{10.14}$$

For large n, the χ^2 confidence level can be computed from the error function (see Section 2.3.1):

$$\mathrm{CL}_S = 100\left[1 - \mathrm{erf}\left(\sqrt{2S} - \sqrt{2n-1}\right)\right]. \tag{10.15}$$

10.3.1 χ^2 Distribution—Smalltalk Implementation

Listing 10.7 shows the implementation of the χ^2 distribution in Smalltalk. The asymptotic limit is implemented directly in the class creation method.

Listing 10.7 Smalltalk implementation of the χ^2 distribution

Class `DhbChiSquareDistribution`

Subclass of `DhbGammaDistribution`

Class Methods

degreeOfFreedom: `anInteger`

```
^anInteger > 40
    ifTrue: [ DhbAsymptoticChiSquareDistribution degreeOfFreedom:
                                                          anInteger]
    ifFalse:[ super shape: anInteger / 2 scale: 2]
```

distributionName

```
^'Chi square distribution'
```

fromHistogram: `aHistogram`

```
| dof |
aHistogram minimum < 0
    ifTrue: [ ^nil].
dof := aHistogram average rounded.
^dof > 0 ifTrue: [ self degreeOfFreedom: aHistogram average
                                                          rounded]
         ifFalse:[ nil]
```

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

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

Instance Methods

changeParametersBy: `aVector`

```
super changeParametersBy: (Array with: aVector first / 2 with:
```

```
0).
```

confidenceLevel: aNumber

```
^( 1 - ( self distributionValue: aNumber)) *100
```

parameters

```
^Array with: alpha * 2
```

10.3.2 χ^2 Distribution—Java Implementation

Listing 10.8 shows the implementation of the χ^2 distribution in Java. The asymptotic form of the distribution has not been implemented. The reason is that a constructor method can only return an instance of the class it belongs to. The alternative requires keeping the asymptotic form in an instance variable and overloading all methods of the superclass to add a test of whether the computation must be delegated to the asymptotic form. This process is somewhat tedious. In practice, however, one seldom needs to use the χ^2 distribution as a probability density function. Only the distribution function is necessary to perform the χ^2-test. Therefore, I did not implement an asymptotic form.

Listing 10.8 Java implementation of the χ^2 distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;

// Chi square distribution.
// (as special case of the gamma distribution)

// @author Didier H. Besset

public final class ChiSquareDistribution extends GammaDistribution
{

// Create a new instance as Gamma( n/2, 2).
// @param n int degrees of freedom of the receiver.

public ChiSquareDistribution ( int n)
{
    super( 0.5 * n, 2.0);
}

// 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 Histogram
// @exception java.lang.IllegalArgumentException
//                  when no suitable parameter can be found.

public ChiSquareDistribution( Histogram h)
    throws IllegalArgumentException
{
    if ( h.getMinimum() < 0 )
        throw new IllegalArgumentException(
            "Chi square distribution is only defined for non-negative
                values");
    int dof = (int) Math.round( h.average());
    if ( dof <= 0 )
        throw new IllegalArgumentException(
            "Chi square distribution is only defined for positive
                degrees of freedom");
    setDegreesOfFreedom( dof);
}

// @return double
// @param x double
// @exception java.lang.IllegalArgumentException
//               if the argument is outside the expected range.

public double confidenceLevel( double x)
                                    throws IllegalArgumentException
{
    return x < 0
            ? Double.NaN
            : ( 1 - distributionValue( x)) * 100;
}

// @return java.lang.String name of the distribution.

public String name()
{
    return "Chi square distribution";
}

// @return double[] an array containing the parameters of
//                              the distribution.

public double[] parameters()
{
    double[] answer = new double[1];
    answer[0] = alpha * 2;
    return answer;
}
```

```
// @param n int

public void setDegreesOfFreedom( int n)
{
    super.defineParameters( 0.5 * n, 2.0);
}

// Note: for fitting, non-integer degree of freedom is allowed
// @param params double[]   assigns the parameters

public void setParameters( double[] params)
{
    defineParameters( params[0] * 0.5, 2);
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    java.text.DecimalFormat fmt = new java.text.DecimalFormat("0.00");
    sb.append("Chi square distribution (");
    sb.append(fmt.format(alpha * 2));
    sb.append(')');
    return sb.toString();
}
}
```

10.3.3 Weighted-Point Implementation

As we shall see in the rest of this chapter, the evaluation of equation 10.10 is performed at many places. Thus, it is convenient to create a new class handling this type of calculation. The new class is called `DhbWeightedPoint` in Smalltalk and `WeightedPoint` in Java. As the implementation in both languages is fully identical, the two classes are described in a common section. An open-minded reader will see that, in this case, each language has one shortcoming over the other.
A weighted point has the following instance variables:

`xValue` the x value of the data point, (i.e., x_i)

`yValue` the y value of the data point, (i.e., y_i)

`weight` the weight of the point, (i.e., $1/\sigma_i^2$)

`error` the error of the y value, (i.e., σ_i)

Accessor methods for each of these instance variables are provided. The accessor method for the error is using lazy initialization to compute the error from the weight in case the error has not yet been defined.

The method `chi2Contribution` (with an added semicolon at the end of the name for Smalltalk) implements the computation of one term of the sum in equation 10.10. The argument of the method is any object implementing the behavior of a one-variable function defined in Section 2.1. In Java one can use the same method name to define a similar method to compute the terms of the sum of equation 10.12: in this case, the argument of the method is another weighted point. This is not possible in Smalltalk, which cannot distinguish the types of the arguments. Thus, for Smalltalk the second method must have a different name: `chi2ComparisonContribution:`. Here Java marks a point over Smalltalk.

Creating instances of the classes can be done in many ways. The fundamental method takes as arguments x_i, y_i, and the weight $1/\sigma_i^2$. However, convenience methods are provided for frequent cases:

1. x_i, y_i, and the error on y_i, σ_i
2. x_i and the content of a histogram bin; the weight is derived from the bin contents,as explained in Section 10.4.
3. x_i, y_i without known error; the weight of the point is set to 1; points without error should not be used together with points with errors.
4. x_i and a statistical moment; in this case, the value y_i is an average over a set of measurements; the weight is determined from the error on the average (see Section 9.1).

Examples of the use of weighted points appear in many sections of this chapter (10.4, 10.8, 10.9).

In the Smalltalk class `DhbWeightedPoint` the values x_i and y_i are always supplied as an instance of the class `Point`. The class `DhbWeightedPoint` has the following class creation methods:

`point:weight:` Fundamental method

`point:error:` Convenience method 1

`point:count:` Convenience method 2

`point:` Convenience method

The convenience method 4 is implemented by the method `asWeightedPoint` of the class `DhbStatisticalMoments`. This kind of technique is quite common in Smalltalk instead of making a class creation method with an explicit name (e.g., `fromMoment:`).

Listing 10.9 Smalltalk implementation of the weighted point class

Class DhbWeightedPoi
Subclass of Object
Instance variable names: xValue yValue weight error

Class Methods

point: aPoint

```
^self new initialize: aPoint weight: 1
```

point: aNumber **count:** anInteger

```
^self point: aNumber @ anInteger
    weight: ( anInteger > 0 ifTrue: [ 1 / anInteger]
                            ifFalse:[ 1])
```

point: aPoint **error:** aNumber

```
^self new initialize: aPoint error: aNumber
```

point: aPoint **weight:** aNumber

```
^self new initialize: aPoint weight: aNumber
```

Instance Methods

chi2ComparisonContribution: aWeightedPoint

```
^(aWeightedPoint yValue - yValue) squared / ( 1 / aWeightedPoint
                                              weight + ( 1 / weight))
```

chi2Contribution: aFunction

```
^(yValue - ( aFunction value: xValue)) squared * weight
```

error

```
error isNil
    ifTrue: [ error := 1 / weight sqrt].
^error
```

initialize: aPoint **error:** aNumber

```
error := aNumber.
^self initialize: aPoint weight: 1 / aNumber squared
```

initialize: `aPoint` **weight:** `aNumber`

```
xValue := aPoint x.
yValue := aPoint y.
weight := aNumber.
^self
```

point

```
^xValue @ yValue
```

weight

```
^weight
```

xValue

```
^xValue
```

yValue

```
^yValue
```

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

Instance Methods

asWeightedPoint: `aNumber`

```
^DhbWeightedPoint point: aNumber @ self average error: self
                                                    errorOnAverage
```

The constructor methods for the Java class `WeightedPoint` are as follows:

(double,double,double) Fundamental method

(double,int) Convenience method 2

(double,double) Convenience method 3

(double,StatisticalMoments) Convenience method 4

Convenience method 1 could not be implemented with a constructor method as the type of the arguments is the same as those in the fundamental method. So, it was necessary to create a method `setError` that must be used in combination with convenience method 3 to implement a definition with values and error. A point for Smalltalk that makes it even with Java!

Listing 10.10 Java implementation of the weighted point class

```
package DhbEstimation;

import DhbInterfaces.OneVariableFunction;
import DhbStatistics.StatisticalMoments;

// Point with error used in chi-square test and least square fits

// @author Didier H. Besset

public class WeightedPoint
{
    private double xValue;
    private double yValue;
    private double weight;
    private double error = Double.NaN;

// Constructor method.
// @param x double
// @param y double

public WeightedPoint( double x, double y)
{
    this( x, y, 1);
}

// Constructor method.
// @param x double
// @param y double
// @param w double

public WeightedPoint( double x, double y, double w)
{
    xValue = x;
    yValue = y;
    weight = w;
}

// Constructor method.
// @param x double
// @param n int    a Histogram bin content

public WeightedPoint( double x, int n)
{
    this( x, n, 1.0 / Math.max(n,1));
```

```
}

// Constructor method.
// @param x double
// @param m DhbStatistics.StatisticalMoments

public WeightedPoint( double x, StatisticalMoments m)
{
    this( x, m.average());
    setError( m.errorOnAverage());
}

// @return double    contribution to chi^2 sum against
//                                  a theoretical function
// @param wp WeightedPoint

public double chi2Contribution( WeightedPoint wp)
{
    double residue = yValue - wp.yValue();
    return residue * residue / ( 1 / wp.weight() + 1 / weight);
}

// @return double    contribution to chi^2 sum against
//                                  a theoretical function
// @param f DhbInterfaces.OneVariableFunction

public double chi2Contribution( OneVariableFunction f)
{
    double residue = yValue - f.value( xValue);
    return residue * residue * weight;
}

// @return double    error of the receiver

public double error()
{
    if( Double.isNaN( error) )
        error = 1 / Math.sqrt( weight);
    return error;
}

// @param e double error on the point

public void setError( double e)
{
    error = e;
```

```
        weight = 1 / ( e * e);
    }

    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        sb.append( '(');
        sb.append( xValue);
        sb.append( ',');
        sb.append( yValue);
        sb.append( "+-");
        sb.append( error());
        sb.append( ')');
        return sb.toString();
    }

    // @return double    weight of the receiver

    public double weight() {
        return weight;
    }

    // @return double    x value of the receiver

    public double xValue() {
        return xValue;
    }

    // @return double    y value of the receiver

    public double yValue() {
        return yValue;
    }
    }
```

10.4 χ^2-test on Histograms

As we have seen in Section 9.3, histograms are often used to collect experimental data. Performing a χ^2-test of data accumulated into a histogram against a function is a frequent task of data analysis.

The χ^2 statistics defined by equation 10.10 require an estimate of the standard deviation of the content of each bin. One can show that the contents of a histogram bin is distributed according to a Poisson distribution. The Poisson distribution is

a discrete distribution[4] whose average is equal to the variance. The probability of observing the integer k is defined by:

$$P_\mu(k) = \frac{\mu^k}{k!} e^\mu, \tag{10.16}$$

where μ is the average of the distribution. In the case of a histogram, the estimated variance of the bin content is then the bin content itself. Therefore equation 10.10 becomes

$$S = \sum_{i=1}^{n} \frac{\left\{ n_i - \mu \left[x_{\min} + \left(i + \frac{1}{2} \right) w \right] \right\}^2}{n_i}, \tag{10.17}$$

where n is the number of bins of the histogram, $x_{\min}$ is its minimum, and w is its bin width. The estimation of the bin content against which the χ^2 statistics is computed, μ, is now a function evaluated at the middle of each bin to average out variations of the function over the bin interval.

In fact, the function μ is often related to a probability density function since histograms are measuring probability distributions. In this case, the evaluation is somewhat different. Let $P(x)$ be the probability density function against which the χ^2 statistics are computed. Then, the predicted bin content for bin i is given by:

$$\mu_i = wNP \left[x_{\min} + \left(i + \frac{1}{2} \right) w \right], \tag{10.18}$$

where N is the total number of values accumulated in the histogram. This is a symmetrical version of the definition of a probability density function: w plays the role of dx in equation 9.23. Plugging equation 10.18 into equation 10.17 yields the expression (equation 10.19) of the χ^2 statistics for a histogram computed against a probability density function $P(x)$:

$$S = \sum_{i=1}^{n} \frac{\left\{ n_i - wNP \left[x_{\min} + \left(i + \frac{1}{2} \right) w \right] \right\}^2}{n_i}. \tag{10.19}$$

This equation cannot be applied for empty bins. If the bin is empty, one can set the weight to 1. This corresponds to a 63% probability of observing no counts if the expected number of measurement is larger than zero.

In both implementations, a single class is in charge of evaluating the predicted bin contents. This class is called a *scaled probability density function*. It is defined by a probability distribution and a histogram.

4. A discrete distribution is a probability distribution whose random variable is an integer.

10.4.1 χ^2-test on Histograms—Smalltalk Implementation

Listing 10.11 shows the implementation of a scaled probability density function in Smalltalk. Listing 10.12 shows the additional methods for the class `DhbHistogram` needed to perform a χ^2-test. Examples of use are given in Sections 10.9.2 and 10.10.2. Code Example 10.3 shows how to compute a χ^2 confidence level to estimate the goodness of a random number generator.

Code Example 10.3

```
| trials probDistr histogram |
trials := 5000.
probDistr := DhbNormalDistribution new.
histogram := DhbHistogram new.
histogram freeExtent: true; setDesiredNumberOfBins: 100.
trials timesRepeat: [ histogram accumulate: probDistr random].
histogram chi2ConfidenceLevelAgainst:
       ( DhbScaledProbabilityDensityFunction histogram: histogram
                                             against: probDistr)
```

The first line after the declaration defines the number of data to be generated to 5,000. Afterword, a new instance of a probability distribution—in this case, a normal distribution with an average of zero and a variance of 1—is created. Then, a new instance of a histogram is created and the next line defines it with a rough number of bins of 100 and the ability to adjust its limits automatically. After all instances have been created, random data generated by the probability distribution are generated. The last statement—extending itself over the last three lines—calculates the confidence level. The argument of the method `chi2ConfidenceLevelAgainst:` is a scaled probability distribution constructed over the histogram and the probability distribution used to generate the accumulated data.

The class `DhbScaledProbabilityDensityFunction` has two class creation methods. The class method `histogram:against:` takes two arguments, a histogram and a probability distribution. This method is used to perform a χ^2-test of the specified histogram against the given probability distribution. The class method `histogram:distributionClass:` first creates a probability distribution of the given class using parameters estimated from the histogram. This method is used to create a scaled probability density function whose parameters will be determined with least square or maximum likelihood fits.

Listing 10.11 Smalltalk implementation of a scaled probability density function

Class	`DhbScaledProbabilityDensityFunction`
Subclass of	`Object`
Instance variable names:	`probabilityDensityFunction count binWidth`

Class Methods

histogram: aHistogram **against:**

```
aProbabilityDensityFunction

^self new
    initialize: aProbabilityDensityFunction
    binWidth: aHistogram binWidth
    count: aHistogram totalCount
```

histogram: aHistogram

distributionClass: aProbabilityDensityFunctionClass

```
^(aProbabilityDensityFunctionClass fromHistogram: aHistogram)
    ifNotNil: [:dp | self histogram: aHistogram against: dp]
```

Instance Methods

changeParametersBy: aVector

```
count := count + aVector last.
probabilityDensityFunction changeParametersBy: aVector.
```

distributionFunction

```
^probabilityDensityFunction distributionFunction
```

initialize: aProbabilityDensityFunction **binWidth:**

aNumber **count:** anInteger

```
probabilityDensityFunction := aProbabilityDensityFunction.
binWidth := aNumber.
count := anInteger.
^self
```

parameters

```
^probabilityDensityFunction parameters copyWith: count
```

printOn: aStream

```
super printOn: aStream.
aStream nextPut: $[;
        nextPutAll: probabilityDensityFunction class
                                                distributionName;
        nextPut: $].
```

setCount: aNumber

```
count := aNumber.
```

value: aNumber

```
^(probabilityDensityFunction value: aNumber) * binWidth * count
```

valueAndGradient: aNumber

```
| g temp |
g := probabilityDensityFunction valueAndGradient: aNumber.
temp := binWidth * count.
^Array with: g first * temp
       with: ( (g last collect: [:each | each * temp]) copyWith:
                                                  g first * binWidth)
```

The evaluation of equation 10.19 is performed by the method chi2Against: of the class DhbHistogram. This method uses the iterator method pointsAndErrorsDo:. This method iterates on all bins and performs on each of them a block using as argument a weighted point as described in Section 10.3.3. This iterator method is also used for least-square and maximum-likelihood fits (see Sections 10.9.2 and 10.10.2).

Listing 10.12 Smalltalk implementation of χ^2-test on histograms

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

Instance Methods

chi2Against: aScaledDistribution

```
| chi2 |
chi2 := 0.
self pointsAndErrorsDo:
    [ :each | chi2 := ( each chi2Contribution:
                                      aScaledDistribution) + chi2].
^chi2
```

chi2ConfidenceLevelAgainst: aScaledDistribution

```
^( DhbChiSquareDistribution degreeOfFreedom: ( contents size -
                              aScaledDistribution parameters size))
```

```
confidenceLevel: ( self chi2Against: aScaledDistribution)
```

pointsAndErrorsDo: aBlock

```
| x |
x := self minimum - ( self binWidth / 2).
contents do:
    [ :each |
      x := x + self binWidth.
      aBlock value: ( DhbWeightedPoint point: x count: each).
    ].
```

10.4.2 χ^2-test on Histograms—Java Implementation

Listing 10.13 shows the implementation of a scaled probability density function in Java. Examples of use are given in Sections 10.9.3 and 10.10.3. Code Example 10.4 showing how to compute a χ^2-confidence level to estimate the goodness of a random number generator.

Code Example 10.4

```
NormalDistribution probDistr = new NormalDistribution();
Histogram histogram = new Histogram( 50, 100);
histogram.setGrowthAllowed();
for( int i = 0; i < 5000; i++ )
   histogram.accumulate( probDistr.random());
double chiCL = histogram.chi2ConfidenceLevelAgainst(
       new DhbScaledProbabilityDensityFunction
               (histogram,probDistr));
```

The first line creates an instance of a normal distribution with an average of zero and a variance of 1. The second line creates an instance of a histogram with 100 desired bins and a cache of 50 elements. The next line defines that the limits of the histogram can be adjusted automatically. The `for` loop accumulates random numbers generated according to the normal distribution into the histogram. The final statement—spread over the last three lines—compute the χ^2-confidence level of the histogram's content against the normal distribution.

The method `chi2Against` of class `Histogram` evaluates the sum of equation 10.17. The method `chi2ConfidenceLevelAgainst` in the same class calculates the confidence level of the χ^2-test. The code of these two methods can be found in Listing 9.7.

The class `ScaledProbabilityDensityFunction` has two constructor methods. The first one takes as arguments the probability density function, the total count, and the bin width of the histogram. The second only takes the probability density function and the histogram as arguments.

Listing 10.13 Java implementation of a scaled probability density function

```
package DhbStatistics;

import DhbInterfaces.ParametrizedOneVariableFunction;
import DhbScientificCurves.Histogram;

// Construct a function from a probability density function
// for a given norm.

// @author Didier H. Besset

public class ScaledProbabilityDensityFunction
                            implements ParametrizedOneVariableFunction
{

    // Total count of the histogram.

    private double count;

    // Bin width of the histogram.

    private double binWidth;

    // Probability density function

    private ProbabilityDensityFunction density;


// @param pdf DhbStatistics.ProbabilityDensityFunction
// @param n long
// @param w double

public ScaledProbabilityDensityFunction(
                    ProbabilityDensityFunction pdf, long n, double w)
{
    density = pdf;
    setCount( n);
    binWidth = w;
}

// @param f statistics.ProbabilityDensity
// @param hist curves.Histogram

public ScaledProbabilityDensityFunction (
                        ProbabilityDensityFunction f, Histogram hist)
```

```
{
    this( f, hist.count(), hist.getBinWidth());
}

// The array contains the parameters of the distribution
// and the estimated number of events.
// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters()
{
    double[] parameters = density.parameters();
    double[] answer = new double[ parameters.length + 1];
    for ( int i = 0; i < parameters.length; i++ )
        answer[i] = parameters[i];
    answer[parameters.length] = count;
    return answer;
}

// @param x double    total count in the receiver

public void setCount(double x)
{
    count = x;
}

// @param n int    total count in the receiver

public void setCount(int n)
{
    count = n;
}

// @param n int    total count in the receiver

public void setCount(long n)
{
    count = n;
}

// @param p double[]    assigns the parameters

public void setParameters( double[] params)
{
    count = params[params.length-1];
    density.setParameters( params);
```

```
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append("Scaled ");
    sb.append(density);
    return sb.toString();
}

// @return double the value of the function.
// @param x double

public double value( double x)
{
    return count * binWidth * density.value( x);
}

// Evaluate the function and the gradient of the function with respect
// to the parameters.
// @return double[]   0: function's value, 1,2,...,n function's gradient
// @param x double

public double[] valueAndGradient( double x)
{
    double[] dpg = density.valueAndGradient(x);
    double[] answer = new double[dpg.length+1];
    double r = binWidth * count;
    for ( int i = 0; i < dpg.length; i++ )
        answer[i] = dpg[i] * r;
    answer[dpg.length] = dpg[0] * binWidth;
    return answer;
}
}
```

10.5 Definition of Estimation

Let us assume that an observable quantity y is following a probability distribution described by a set of observable quantities $x_1, x_2 \ldots$ (called the *experimental conditions*) and a set of parameters $p_1, p_2 \ldots$. In other words, the probability density

function of the random variable[5] corresponding to the observable quantity y can be written as

$$P\left(y\right) = P\left(y; \mathbf{x}, \mathbf{p}\right), \tag{10.20}$$

where $\mathbf{x}$ is the vector $(x_1, x_2 \ldots)$ and $\mathbf{p}$ the vector $\left(p_1, p_2 \ldots\right)$.

The estimation of the values of the parameters $p_1, p_2 \ldots$ is the determination of the parameters $p_1, p_2 \ldots$ by performing several measurements of the observable y for different experimental conditions $x_1, x_2 \ldots$.

Let N be the number of measurements; let y_i be the ith measured value of the observable y under the experimental conditions $\mathbf{x}_i$.

10.5.1 Maximum-Likelihood Estimation

The maximum-likelihood estimation of the parameters $\mathbf{p}$ is the set of values $\bar{\mathbf{p}}$ maximizing the following function:

$$L\left(\mathbf{p}\right) = \prod_{i=1}^{N} P\left(y_i; \mathbf{x}_i, \mathbf{p}\right). \tag{10.21}$$

By definition, the likelihood function $L\left(\mathbf{p}\right)$ is the probability of making the N measurements. The maximum likelihood estimation determines the estimation $\bar{\mathbf{p}}$ of the parameters $\mathbf{p}$ such that the series of measurements performed is the most probable, hence the name *maximum likelihood*.

One can show that the maximum-likelihood estimation is robust and unbiased. The robustness and the bias of an estimation are defined mathematically. For short, *robust* means that the estimation converges toward the true value of the parameters for an infinite number of measurements; *unbiased* means that the deviations between the estimated parameters and their true value are symmetrically distributed around zero for any finite number of measurements.

Equation 10.21 is often rewritten in logarithmic form to ease the computation of the likelihood function.

$$I\left(\mathbf{p}\right) = \ln L\left(\mathbf{p}\right) = \sum_{i=1}^{N} \ln P\left(y_i; \mathbf{x}_i, \mathbf{p}\right). \tag{10.22}$$

The function $I\left(\mathbf{p}\right)$ is related to information and is used in information theory.

10.5.2 Least-Square Estimation

Let us assume that the random variable y is distributed according to a normal distribution of given standard deviation σ and that the average of the normal distribution

5. For simplicity I shall use the same notation for the random variable and the observable quantity.

is given by a function $F(\mathbf{x}, \mathbf{p})$ of the experimental conditions and the parameters. In this case, equation 10.20 becomes

$$P(y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{[y-F(\mathbf{x},\mathbf{p})]^2}{2\sigma^2}} \tag{10.23}$$

Plugging equation 10.23 into equation 10.22 yields

$$I(\mathbf{p}) = -N\sqrt{2\pi\sigma^2} - \sum_{i=1}^{N} \frac{[y - F(\mathbf{x}, \mathbf{p})]^2}{2\sigma^2} \tag{10.24}$$

The problem of finding the maximum of $I(\mathbf{p})$ is now equivalent to the problem of finding the minimum of the function:

$$S_{\mathrm{ML}}(\mathbf{p}) = \sum_{i=1}^{N} \frac{[y - F(\mathbf{x}, \mathbf{p})]^2}{\sigma^2}, \tag{10.25}$$

where a redundant factor 2 has been removed. This kind of estimation is called *least-square estimation*. Written as in equation 10.25, least-square estimation is fully equivalent to maximum-likelihood estimation. By definition, the quantity $S_{\mathrm{ML}}(\mathbf{p})$ is distributed as a χ^2 random variable with $N - m$ degrees of freedom, where m is the number of parameters—that is, the dimension of the vector $\mathbf{p}$.

In practice, however, the standard deviation σ is not known and frequently depends on the parameters $\mathbf{p}$. In that case, one uses instead an estimation for the standard deviation. Either the standard deviation of each measurement is determined experimentally by making several measurements under the same experimental conditions, or it is estimated from the measurement error. Then, equation 10.25 can be rewritten as

$$S(\mathbf{p}) = \sum_{i=1}^{N} \frac{[y - F(\mathbf{x}, \mathbf{p})]^2}{\sigma_i^2}. \tag{10.26}$$

The least-square estimation is obtained by minimizing the quantity $S(\mathbf{p})$ with respect to the parameters $\mathbf{p}$. This kind of estimation can be used to determine the parameters of a functional dependence of the variable y from the observable quantities $\mathbf{x}$. For this reason it is also called a *least-square fit* when it is used to fit the parameter of a functional dependence to the measurements.

In general, the distribution of the random variable y may not be a normal distribution. One can nevertheless show that the least square estimation is robust. However, it is biased. Depending on the nature of the distribution of the random variable y, the parameters may be over- or underestimated. This is especially the case when working with histograms.

We have said that all measurements of the observable quantities y must be distributed according to a normal distribution so that the quantity $S(\mathbf{p})$ of equation

10.26 is distributed as a χ^2 random variable. In general, this is often the case[6] when dealing with a large quantity of measurements. Thus, a least-square fit is also called a χ^2 fit. In this case, one can apply the χ^2-test described in Section 10.3 to assess the goodness of the fitted function.

If $S(\mathbf{p})$ has a minimum respective to $\mathbf{p}$, then all partial derivatives of the function $S(\mathbf{p})$ respective to each of the components of the vector $\mathbf{p}$ are zero. Since the function is positive and quadratic in $\mathbf{p}$, it is clear that the function must have at least one minimum. Under these circumstances the minimum can be obtained by solving equation 10.27

$$\frac{\partial}{\partial p_j} F(\mathbf{x}_i; p_1, \ldots, p_m) = 0 \quad \text{for } j = 1, \ldots, m, \tag{10.27}$$

where m is the number of parameters—that is, the dimension of the vector $\mathbf{p}$. When a solution is found, one should in principle verify that it is really a minimum. Solving equation 10.27 gives the following system of equations:

$$\sum_{i=1}^{N} \frac{y - F(\mathbf{x}, \mathbf{p})}{\sigma_i^2} \cdot \frac{\partial}{\partial p_j} S(p_1, \ldots, p_m) = 0 \quad \text{for } j = 1, \ldots, m. \tag{10.28}$$

Once this system has been solved, one can compute the value of $S(\mathbf{p})$ using equations 10.26 or, better, the value $S_{\mathrm{ML}}(\mathbf{p})$ using equation 10.25. Computing the χ^2 confidence level of that value (see Section 10.3) using a χ^2 distribution with $N - m$ degrees of freedom gives the probability that the fit is acceptable.

10.6 Least-Square Fit with Linear Dependence

If the function $F(\mathbf{x}, \mathbf{p})$ is a linear function of the vector $\mathbf{p}$, it can be written in the following form

$$F(\mathbf{x}, \mathbf{p}) = \sum_{j=1}^{m} f_j(\mathbf{x}) \cdot p_j. \tag{10.29}$$

In that case, equation 10.28 become a system of linear equations of the form

$$\mathbf{M} \cdot \mathbf{p} = \mathbf{c}, \tag{10.30}$$

where the coefficients of the matrix $\mathbf{M}$ are given by

$$M_{jk} = \sum_{i=1}^{N} \frac{f_j(\mathbf{x}_i) f_k(\mathbf{x}_i)}{\sigma_i^2} \quad \text{for } j, k = 1, \ldots, m, \tag{10.31}$$

6. This is a consequence of a theorem known as the law of large numbers.

and the components of the constant vector **c** are given by

$$c_j = \sum_{i=1}^{N} \frac{y_i f_j(\mathbf{x}_i)}{\sigma_i^2} \quad \text{for } j = 1, \ldots, m. \tag{10.32}$$

Equation 10.30 is a system of linear equation that can be solved according to the algorithms exposed in Sections 8.2 and 8.3. If one is interested only in the solution, this is all there is to do.

A proper fit, however, should give an estimation of the error in estimating the parameters. The inverse of the matrix **M** is the error matrix for the fit. The error matrix is used to compute the estimation of variance on the function $F(\mathbf{x}, \mathbf{p})$ as follows:

$$\text{var}\left[F(\mathbf{x}, \mathbf{p})\right] = \sum_{j=1}^{m} \sum_{k=1}^{m} M_{jk}^{-1} f_j(\mathbf{x}) f_k(\mathbf{x}) . \tag{10.33}$$

The estimated error on the function $F(\mathbf{x}, \mathbf{p})$ is the square root of the estimated variance.

The diagonal elements of the error matrix are the variance of the corresponding parameter. That is,

$$\text{var}(p_j) = M_{jj}^{-1} \quad \text{for } j = 1, \ldots, m. \tag{10.34}$$

The off-diagonal elements describe the correlation between the errors on the parameters. One defines the correlation coefficient of parameter p_j and p_j by

$$\text{cor}(p_j, p_k) = \frac{M_{jk}^{-1}}{\sqrt{M_{jj}^{-1} M_{kk}^{-1}}} \quad \text{for } j, k = 1, \ldots, m \text{ and } j \neq k. \tag{10.35}$$

All correlation coefficients are comprised between -1 and 1. If the absolute value of a correlation coefficient is close to 1, it means that one of the two corresponding two parameters is redundant for the fit. In other words, one parameter can be expressed as a function of the other.

10.7 Linear Regression

A *linear regression* is a least-square fit with a linear function of a single variable. The dimension of the vector **x** is 1, and the dimension of the vector **p** is 2. The function to fit has only two parameters. The following convention is standard:

$$\begin{cases} p_1 = a, \\ p_2 = b, \\ F(\mathbf{x}, \mathbf{p}) = ax + b. \end{cases} \tag{10.36}$$

With these definitions, the system of equations 10.30 becomes

$$\begin{cases} \sum_{i=1}^{N} \frac{x_i^2}{\sigma^2} a + \sum_{i=1}^{N} \frac{x_i}{\sigma^2} b = \sum_{i=1}^{N} \frac{x_i y_i}{\sigma^2} \\ \sum_{i=1}^{N} \frac{x_i}{\sigma^2} a + \sum_{i=1}^{N} \frac{1}{\sigma^2} b = \sum_{i=1}^{N} \frac{y_i}{\sigma^2}. \end{cases} \tag{10.37}$$

This system can easily be solved. Before giving the solution, let us introduce a shorthand notation for the weighted sums:

$$\langle Q \rangle = \sum_{i=1}^{N} \frac{Q_i}{\sigma_i^2}. \tag{10.38}$$

Using this notation, the solution of the system of equations 10.37 can be written as:

$$\begin{cases} a = \dfrac{\langle xy \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle y \rangle}{\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle} \\ b = \dfrac{\langle xx \rangle \cdot \langle y \rangle - \langle xy \rangle \cdot \langle x \rangle}{\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle} \end{cases}, \tag{10.39}$$

where the symmetry of the expression is quite obvious. It is interesting to note that if we had fitted x as a linear function of y (i.e., $x = \tilde{a}y + \tilde{b}$), we would have the following expression for the slope:

$$\tilde{a} = \frac{\langle xy \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle y \rangle}{\langle yy \rangle \cdot \langle 1 \rangle - \langle y \rangle \cdot \langle y \rangle}. \tag{10.40}$$

If the dependence between x and y is truly a linear function, the product $a\tilde{a}$ ought to be 1. The square root of the product $a\tilde{a}$ is defined as the correlation coefficient of the linear regression, the sign of the square root being the sign of the slope. The correlation coefficient r is thus given by

$$r = \frac{\langle xy \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle y \rangle}{\sqrt{\left(\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle\right)\left(\langle yy \rangle \cdot \langle 1 \rangle - \langle y \rangle \cdot \langle y \rangle\right)}}. \tag{10.41}$$

Since the least-square fit is a biased estimator for the parameters, the square of the correlation coefficient is less than 1 in practice. The value of the correlation coefficient lies between -1 and 1. A good linear fit ought to have the absolute value of r close to 1.

Finally, the error matrix of a linear regression is given by

$$\mathbf{M}^{-1} = \frac{1}{\langle xx \rangle \cdot \langle 1 \rangle - \langle x \rangle \cdot \langle x \rangle} \begin{pmatrix} \langle xx \rangle & -\langle x \rangle \\ -\langle x \rangle & \langle 1 \rangle \end{pmatrix} \tag{10.42}$$

when the vector representing the parameters of the fit is defined as (b, a) in this order.

When fitting a functional dependence with one variable, x, and many parameters, one can use a linear regression to reduce rounding errors when the observed values $y_1, \ldots, y_N$ cover a wide numerical range. Let a and b be the result of the

linear regression of the values y_i as a function of x_i. One defines the new quantity $y'_i = y_i - (ax_i + b)$ for all i. The standard deviation of y'_i is the same as that of y_i since the subtracted expression is just a change of variable. In fact, the linear regression does not need to be a good fit at all. Then, the functional dependence can be fitted on the quantities $y'_1, \ldots, y'_N$. I shall give a detailed example on this method in Section 10.8.

10.7.1 Linear Regression—General Implementation

Linear regression is implemented within a single class using a similar implementation as that of the statistical moments. This means that individual measurements are accumulated and not stored. The drawback is that the object cannot compute the confidence level of the fit. This is not so much a problem since the correlation coefficient is usually sufficient to estimate the goodness of the fit.
The class has the following instance variables:

`sum1` is used to accumulate the sum of weights, (i.e., $\langle 1 \rangle$).

`sumX` is used to accumulate the weighted sum of x_i, (i.e., $\langle x \rangle$).

`sumY` is used to accumulate the weighted sum of y_i, (i.e., $\langle y \rangle$).

`sumXY` is used to accumulate the weighted sum of $x_i \times y_i$, (i.e., $\langle xy \rangle$).

`sumXX` is used to accumulate the weighted sum of x_i^2, (i.e., $\langle xx \rangle$).

`sumYY` is used to accumulate the weighted sum of y_i^2, (i.e., $\langle yy \rangle$).

`slope` the slope of the linear regression, (i.e., a).

`intercept` the value of the linear regression at $x = 0$, (i.e., b).

`correlationCoefficient` the correlation coefficient, (i.e., r in equation 10.41).

When one of the instance variables `slope`, `intercept`, or `correlationCoefficient` is needed, the method `computeResults` calculating the values of the three instance variables is called using *lazy initialization*. When new data are added to the object, these variables are reset. It is thus possible to investigate the effect of adding new measurements on the results.

The methods `asPolynomial` and `asEstimatedPolynomial` return an object used to compute the predicted value for any x. The estimated polynomial is using the error matrix of the least-square fit to compute the error on the predicted value. Estimated polynomials are explained in Section 10.8.

10.7.2 Linear Regression—Smalltalk Implementation

Listing 10.14 shows the complete implementation in Smalltalk. Code Example 10.5 shows how to use the class `DhbLinearRegression` to perform a linear regression over a series of measurements .

Code Example 10.5

```
| linReg valueStream measurement slope intercept
  correlationCoefficient estimation value error|
linReg := DhbLinearRegression new.
[ valueStream atEnd]
       whileFalse:[ measurement := valueStream next.
                    linReg addPoint: measurement point
                             weight: measurement weigth
                  ].
slope := linReg slope.
intercept := linReg intercept.
correlationCoefficient := linReg correlationCoefficient.
estimation := linReg asEstimatedPolynomial.
value := estimation value: 0.5.
error := estimation error: 0.5.
```

This example assumes that the measurement of the random variable is obtained from a stream. The exact implementation of the stream is not shown here. The first line after the declaration creates a new instance of class `DhbLinearRegression`. Next comes the loop over all values found in the stream. This examples assumes that the values are stored on the stream as a single object implementing the following methods:

`point` returns a point containing the measurement, (i.e., the pair (x_i, y_i) for all i).

`weight` returns the weight of the measurement, (i.e., $1/\sigma_i^2$).

Each point is accumulated into the linear regression object with the method `addPoint:weight:`.

After all measurements have been read, the results of the linear regression are fetched. The last three lines show how to obtain a polynomial object used to compute the value predicted by the linear regression at $x = 0.5$ and the error on that prediction.

The mechanism of lazy initialization is implemented by setting the three instance variables `slope`, `intercept`, and `correlationCoefficient` to `nil` in the method `reset`.

Listing 10.14 Smalltalk implementation of linear regression

Class	`DhbLinearRegression`
Subclass of	`Object`
Instance variable names:	`sum1 sumX sumY sumXX sumYY sumXY slope intercept correlationCoefficient`

Class Methods

new

```
^( super new) reset; yourself
```

Instance Methods

add: aPoint

```
self add: aPoint weight: 1.
```

add: aPoint **weight:** aNumber

```
sum1 := sum1 + aNumber.
sumX := sumX + (aPoint x * aNumber).
sumY := sumY + (aPoint y * aNumber).
sumXX := sumXX + (aPoint x squared * aNumber).
sumYY := sumYY + (aPoint y squared * aNumber).
sumXY := sumXY + (aPoint x * aPoint y * aNumber).
self resetResults
```

asEstimatedPolynomial

```
^( DhbEstimatedPolynomial coefficients: self coefficients)
        errorMatrix: self errorMatrix;
        yourself
```

asPolynomial

```
^DhbPolynomial coefficients: self coefficients
```

coefficients

```
^Array with: self intercept with: self slope
```

computeResults

```
| xNorm xyNorm |
xNorm := sumXX * sum1 - (sumX * sumX).
xyNorm := sumXY * sum1 - (sumX * sumY).
slope := xyNorm / xNorm.
intercept := (sumXX * sumY - (sumXY * sumX)) / xNorm.
correlationCoefficient := xyNorm
            / (xNorm * (sumYY * sum1 - (sumY * sumY))) sqrt
```

correlationCoefficient

```
correlationCoefficient isNil
    ifTrue: [ self computeResults].
^correlationCoefficient
```

errorMatrix

```
| c1 cx cxx |
c1 := 1.0 / (sumXX * sum1 - sumX squared).
cx := sumX negated * c1.
cxx := sumXX * c1.
c1 := sum1 * c1.
^DhbSymmetricMatrix rows: (Array with: (Array with: cxx with: cx)
            with: (Array with: cx with: c1))
```

errorOnIntercept

```
^(sumXX / (sumXX * sum1 - sumX squared)) sqrt
```

errorOnSlope

```
^(sum1 / (sumXX * sum1 - sumX squared)) sqrt
```

intercept

```
intercept isNil
    ifTrue: [ self computeResults].
^intercept
```

remove: aPoint

```
sum1 := sum1 - 1.
sumX := sumX - aPoint x.
sumY := sumY - aPoint y.
sumXX := sumXX - aPoint x squared.
sumYY := sumYY - aPoint y squared.
sumXY := sumXY - (aPoint x * aPoint y).
self resetResults
```

reset

```
sum1 := 0.
sumX := 0.
sumY := 0.
sumXX := 0.
sumYY := 0.
sumXY := 0.
self resetResults
```

resetResults

```
slope := nil.
intercept := nil.
correlationCoefficient := nil.
```

slope

```
slope isNil
    ifTrue: [ self computeResults].
^slope
```

value: aNumber

```
^aNumber * self slope + self intercept
```

10.7.3 Linear Regression—Java Implementation

Listing 10.15 shows the complete implementation in Java. Code Example 10.6 shows how to use the class `LinearRegression` to perform a linear regression over a series of measurements .

Code Example 10.6

```
double[] x, y, w;
   .
   .
   .  <Gathering measurements into the arrays x(x_i), y (y_i), and w (1/σ_i^2)>
LinearRegression linReg = new LinearRegression;
for ( int i =  1, i < x.length, i++ )
   linReg.add( x[i], y[i], w[i]);
double slope = linReg.slope();
double intercept = linReg.intercept();
double correlationCoefficient = linReg.correlationCoefficient();
EstimatedPolynomial estimation = linReg.asEstimatedPolynomial();
double value = estimation.value(0.5);
double error = estimation.error(0.5);
```

This example assumes that the measurements and the weights are gathered into three arrays. The exact implementation of the gathering is not shown here. The first line after the gathering creates a new instance of class `LinearRegression`. Next comes the loop accumulating the measurements into the linear regression object with the method `add`.

After all measurements have been accumulated, the results of the linear regression are fetched. The last three lines show how to obtain a polynomial object used to compute the value predicted by the linear regression at $x = 0.5$ and the error on that prediction.

The mechanism of lazy initialization is implemented by setting the three instance variables `slope`, `intercept`, and `correlationCoefficient` to `NaN` in the method `reset`.

Listing 10.15 Java implementation of linear regression

```
package DhbEstimation;

import DhbFunctionEvaluation.PolynomialFunction;
import DhbInterfaces.PointSeries;
import DhbMatrixAlgebra.SymmetricMatrix;
import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbMatrixAlgebra.DhbNonSymmetricComponents;

// linear regression

// @author Didier H. Besset

public class LinearRegression
{

// Number of accumulated points

    private int sum1;

// Sum of X

    private double sumX;

// Sum of Y

    private double sumY;

// Sum of XX

    private double sumXX;

// Sum of XY

    private double sumXY;

// Sum of YY

    private double sumYY;

// Slope

    private double slope;

// Intercept
```

```
    private double intercept;

// Correlation coefficient

    private double correlationCoefficient;

// Cnstructor method.

public LinearRegression()
{
    super();
    reset();
}

// @param x double
// @param y double

public void add( double x, double y)
{
    add( x, y, 1);
}

// @param x double
// @param y double
// @param w double

public void add( double x, double y, double w)
{
    double wx = w * x;
    double wy = w * y;
    sum1 += w;
    sumX += wx;
    sumY += wy;
    sumXX += wx * x;
    sumYY += wy * y;
    sumXY += wx * y;
    resetResults();
}

// @return DhbFunctionEvaluation.PolynomialFunction

public EstimatedPolynomial asEstimatedPolynomial()
{
    return new EstimatedPolynomial( coefficients(), errorMatrix());
}
```

```
// @return DhbFunctionEvaluation.PolynomialFunction

public PolynomialFunction asPolynomial()
{
    return new PolynomialFunction( coefficients());
}

// @return double[]

private double[] coefficients()
{
    double[] answer = new double[2];
    answer[0] = getIntercept();
    answer[1] = getSlope();
    return answer;
}
private void computeResults()
{
    double xNorm = sumXX * sum1 - sumX * sumX;
    double xyNorm = sumXY * sum1 - sumX * sumY;
    slope = xyNorm / xNorm;
    intercept = ( sumXX * sumY - sumXY * sumX) / xNorm;
    correlationCoefficient = xyNorm / Math.sqrt( xNorm *
                                    ( sumYY * sum1 - sumY * sumY));
}

// @return DhbMatrixAlgebra.SymmetricMatrix

public SymmetricMatrix errorMatrix()
{
    double[][] rows = new double[2][2];
    rows[1][1] = 1./ ( sumXX * sum1 - sumX * sumX);
    rows[0][1] = sumXX * rows[1][1];
    rows[1][0] = rows[0][1];
    rows[0][0] = sumXX * rows[1][1];
    SymmetricMatrix answer = null;
    try { try { answer = SymmetricMatrix.fromComponents( rows);
              } catch( DhbIllegalDimension e){};
        } catch( DhbNonSymmetricComponents e){};
    return answer;
}

// @return double

public double getCorrelationCoefficient()
{
```

```
        if( Double.isNaN( correlationCoefficient) )
            computeResults();
        return correlationCoefficient;
    }

    // @return double

    public double getIntercept()
    {
        if( Double.isNaN( intercept) )
            computeResults();
        return intercept;
    }

    // @return double

    public double getSlope()
    {
        if( Double.isNaN( slope) )
            computeResults();
        return slope;
    }

    // @param x double
    // @param y double

    public void remove( double x, double y)
    {
        sum1 -= 1;
        sumX -= x;
        sumY -= y;
        sumXX -= x * x;
        sumYY -= y * y;
        sumXY -= x * y;
        resetResults();
    }
    public void reset()
    {
        sum1 = 0;
        sumX = 0;
        sumY = 0;
        sumXX = 0;
        sumYY = 0;
        sumXY = 0;
        resetResults();
    }
```

```
private void resetResults()
{
    slope = Double.NaN;
    intercept = Double.NaN;
    correlationCoefficient = Double.NaN;
}

// @return double
// @param x double

public double value( double x)
{
    return x * getSlope() + getIntercept();
}
}
```

10.8 Least-square Fit with Polynomials

In a polynomial fit, the fit function is a polynomial of degree m . In this case, the parameters are usually numbered starting from zero; the number of free parameters is $m + 1$, and the number of degrees of freedom is $N - m - 1$. We have

$$F\left(x; p_0, p_1, \ldots, p_m\right) = \sum_{k=0}^{n} p_k x^k. \tag{10.43}$$

The partial derivative of equation 10.27 is easily computed since a polynomial is a linear function of its coefficients:

$$\frac{\partial}{\partial p_j} F\left(\mathbf{x}_i; p_1, \ldots, p_m\right) = x_i^j \quad \text{for } j = 1, \ldots, m. \tag{10.44}$$

Such a matrix is called a *Van Der Monde matrix*. The system of equations 10.28 then becomes

$$\sum_{k=0}^{m} p_k \cdot \sum_{i=1}^{N} \frac{x_i^{j+k}}{\sigma_i^2} = \sum_{i=1}^{N} \frac{x_i^j y_i}{\sigma_i^2}. \tag{10.45}$$

Equation 10.45 is of the same form as equation 10.30 where the coefficients of the matrix **M** are given by

$$M_{jk} = \sum_{i=1}^{N} \frac{x_i^{j+k}}{\sigma_i^2}, \tag{10.46}$$

and the vector **c** has for components

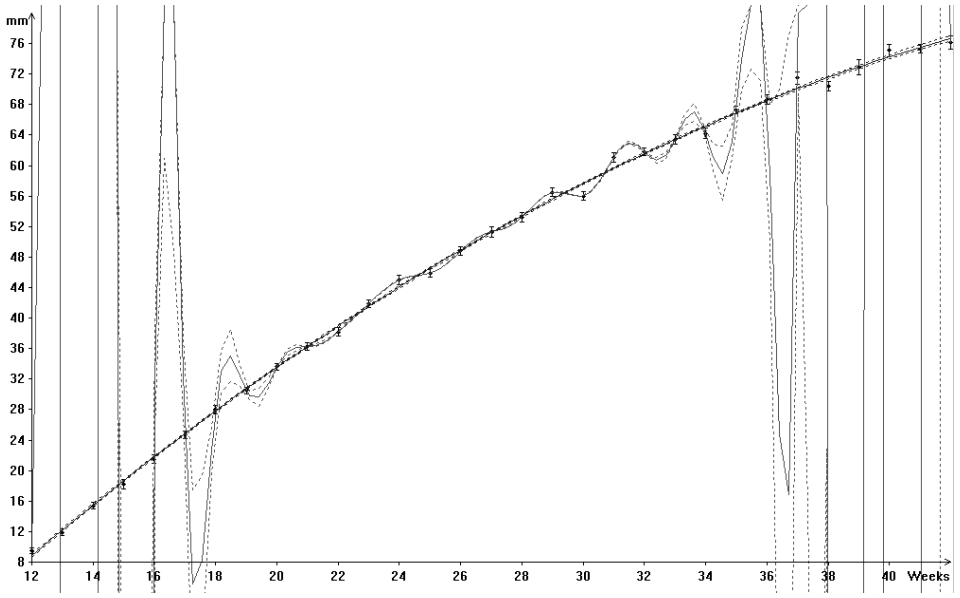

FIG. 10.5 Example of polynomial fit

$$c_j = \sum_{i=1}^{N} \frac{x_i^j y_i}{\sigma_i^2}. \tag{10.47}$$

Polynomial least-square fit provides a way to construct an ad hoc representation of a functional dependence defined by a set of points. Depending on the type of data, it can be more efficient than the interpolation methods discussed in Chapter 3. In general, the degree of the polynomial should be kept small to prevent large fluctuations between the data points.

Let us now consider a concrete example of polynomial fit.

To determine whether a fetus is developing itself normally within the womb, the dimension of the its bones are measured during an ultrasound examination of the mother-to-be. The dimensions are compared against a set of standard data measured on a control population. Such data[7] are plotted in Figure 10.5: the y-axis is the length of the femur expressed in millimeters; the x-axis represents the duration in weeks of the pregnancy based on the estimated date of conception. Each measurement has been obtained by measuring the length of different fetuses at the same gestational age. The measurements are averaged, and the error on the average is also calculated (see Section 9.1).

7. These numbers are reproduced with permission of Professor P. J. Steer from the Department of Obstetrics and Gynecology of the Chelsea and Westminster Hospital of London.

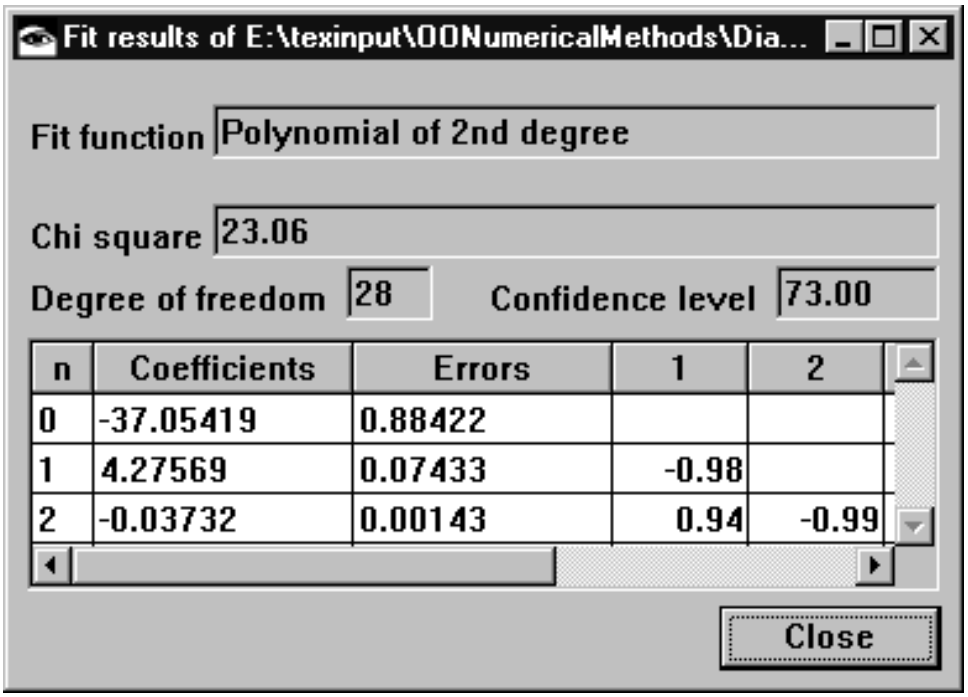

FIG. 10.6 Fit results for the fit of Figure 10.5

The obtained data do not follow a smooth curve since the data have been determined experimentally. The exact date of conception cannot be exactly determined, so some fluctuation is expected, but the major limitation of the data is to obtain a sufficient number of measurements to smooth out the natural variations between individuals. The data from Figure 10.5 have been fitted with a second order polynomial: the result is shown with a thin black curve. As one can see, the fit is excellent in spite of the fluctuation of the measurements. Figure 10.6 shows the fit results. This χ^2 confidence level is rather good. However, the correlation coefficients are quite high. This is usually the case with polynomial fits as each coefficient strongly depends on the other.

The thick gray line in Figure 10.5 shows the interpolation polynomial[8] for comparison (interpolation is discussed in Chapter 3). The interpolation polynomial gives unrealistic results because of the fluctuations of the experimental data.

Polynomial fits have something in common with interpolation: a fitted polynomial can seldom be used to extrapolate the data outside of the interval defined by

8. An attentive reader will notice that the interpolation curve does not go through some data points. This is an artifact of the plotting over a finite sample of points that do not coincide with the measured data.

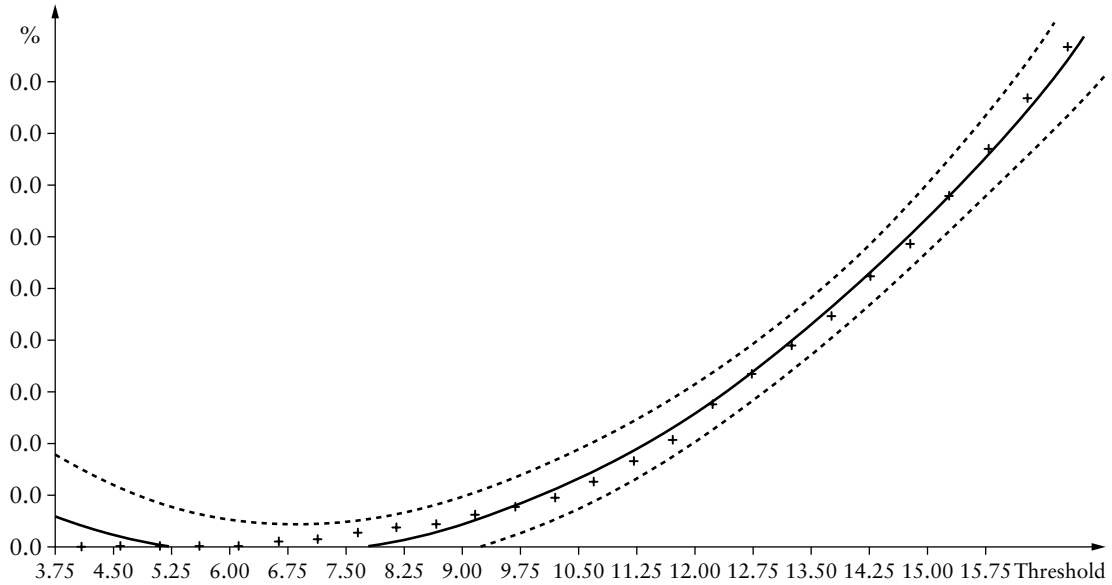

FIG. 10.7 Limitation of polynomial fits

the reference points. This is illustrated in Figure 10.7 showing a second order polynomial fit made on a series of points. This series is discussed in Section 3.1 (Figure 3.5). The reader can see that the fitted polynomial does not reproduce the behavior of the data in the lower part of the region. Nevertheless, the data are well within the estimated error. Thus, the fit results are consistent. Unfortunately, too many fit results are presented without their estimated error. This kind of information, however, is an essential part of a fit and should always be delivered along with the fitted function.

This is the idea behind what is called an *estimated polynomial* which is a polynomial whose coefficients have been determined by a least-square fit. An estimated polynomial keeps the error matrix of the fit is along with the coefficients of the fitted polynomial.

Note: The Smalltalk and Java implementations are quite different from each other. The reader is advised to read both sections and, as an exercise, may attempt to write the alternate implementation in the desired language.

10.8.1 Polynomial Least-Square Fits—Smalltalk Implementation

Listing 10.16 shows the complete implementation in Smalltalk. Code Example 10.7 shows how to perform the fit made in Figure 10.5.

Code Example 10.7

```
| fit valueStream dataHolder estimation value error|
```

```
    :
    <Accumulation of data into dataHolder>
fit := DhbPolynomialLeastSquareFit new: 2.
dataHolder pointsAndErrorsDo: [ :each | fit add: each].
estimation := fit evaluate.
value := estimation value: 20.5.
error := estimation error: 20.5.
```

The data are accumulated into a object called `dataHolder` implementing the iterator method `pointsAndErrorsDo`. The argument of the block used by this method is an instance of class `DhbWeightedPoint` described in Section 10.3.3. The iterator method acts on all experimental data stored in `dataHolder`. Next, an instance of class `DhbPolynomialLeastSquareFit` is created. The argument of the method is the degree of the polynomial; here a second-order polynomial is used. After data have been accumulated into this object, the fit is performed by sending the method `evaluate` to the fit object. This method returns a polynomial including the error matrix. The last three lines compute the predicted femur length and its error in the middle of the 20th week of pregnancy.

The Smalltalk implementation assumes the points are stored in an object implementing the iterator method `do:`. Any instance of `Collection` of its subclasses will work. Each element of the collection must be an array containing the values x_i, y_i, and $1/\sigma_i$. The class `DhbPolynomialLeastSquareFit` keeps this collection in the instance variable `pointCollection`. A second instance variable, `degreePlusOne`, keeps the number of coefficients to be estimated by the fit.

The class creation method `new:` is used to create an instance by supplying the degree of the fit polynomial as argument. `pointCollection` is set to a new instance of an `OrderedCollection`. Then, new values can be added to the fit instance with the method `add:`.

The other class creation method, `new:on:` takes two arguments: the degree of the fit polynomial and the collection of points. The fit result can be fetched directly after the creation.

The method `evaluate` solves equation 10.28 by first computing the inverse of the matrix **M** to get the error matrix. The coefficients are then obtained from the multiplication of the constant vector by the error matrix.

Listing 10.16 Smalltalk implementation of a polynomial least square fit

Class	`DhbPolynomialLeastSquareFit`
Subclass of	`Object`
Instance variable names:	`pointCollection degreePlusOne`

Class Methods

new: anInteger

```
^super new initialize: anInteger
```

new: anInteger **on:** aCollectionOfPoints

```
^super new initialize: anInteger on: aCollectionOfPoints
```

Instance Methods

accumulate: aWeightedPoint **into:** aVectorOfVectors

and: aVector

```
OfVectors and: aVector
    | t p powers |
    p := 1.0.
    powers := aVector collect: [ :each | t := p. p := p *
                                          aWeightedPoint xValue. t].
    aVector accumulate: powers * ( aWeightedPoint yValue *
                                     aWeightedPoint weight).
    1 to: aVector size do:
        [ :k |
          ( aVectorOfVectors at: k) accumulate: powers * ( ( powers
                                    at: k) * aWeightedPoint weight).
        ].
```

add: aWeightedPoint

```
^pointCollection add: aWeightedPoint
```

computeEquations

```
| rows vector |
vector := ( DhbVector new: degreePlusOne) atAllPut: 0 ; yourself.
rows := ( 1 to: degreePlusOne) collect: [ :k | ( DhbVector new:
                           degreePlusOne) atAllPut: 0 ; yourself].
pointCollection do:
    [ :each | self accumulate: each into: rows and: vector].
^Array with: ( DhbSymmetricMatrix rows: rows) with: vector
```

evaluate

```
| system errorMatrix |
system := self computeEquations.
errorMatrix := ( system at: 1) inverse.
^( DhbEstimatedPolynomial coefficients: errorMatrix * (system at:
                                                                  2))
        errorMatrix: errorMatrix;
```

```
        yourself
```

initialize: anInteger

```
^self initialize: anInteger on: OrderedCollection new
```

initialize: anInteger **on:** aCollectionOfPoints

```
pointCollection := aCollectionOfPoints.
degreePlusOne := anInteger + 1.
^self
```

Listing 10.17 show the implementation of the class DhbEstimatedPolynomial, which is a subclass of the class DhbPolynomial containing the error matrix of the fit performed to make a estimation of the polynomial's coefficients. The method error: returns the estimated error of its value based on the error matrix of the fit using equation 10.33. The convenience method valueAndError: returns an array containing the estimated value and its error in a single method call. This is suitable for plotting the resulting curve.

Listing 10.17 Smalltalk implementation of a polynomial with error

Class	DhbEstimatedPolynomial
Subclass of	DhbPolynomial
Instance variable names:	errorMatrix

Instance Methods

error: aNumber

```
| errorVector term nextTerm |
nextTerm := 1.
errorVector := ( coefficients collect: [ :each | term :=
        nextTerm. nextTerm := aNumber * nextTerm. term]) asVector.
^( errorVector * errorMatrix * errorVector) sqrt
```

errorMatrix

```
^errorMatrix
```

errorMatrix: aMatrix

```
errorMatrix := aMatrix.
```

valueAndError: aNumber

```
^Array with: ( self value: aNumber) with: ( self error: aNumber)
```

10.8.2 Polynomial Least-Square Fits—Java Implementation

Listing 10.18 shows the complete implementation in Java. Code Example 10.8 show how to perform the fit made in Figure 10.5.

Code Example 10.8

```
double[] x, y, dy;
   :
   : <Gathering measurements into the arrays x (x_i), y (y_i), and dy (σ_i)>
PolynomialLeastSquareFit fit = new PolynomialLeastSquareFit(2);
for( int i = 0; i < x.length; i++ )
   fit.accumulatePoint( x, y, w);
EstimatedPolynomial estimation = fit.evaluate();
double value = estimation.value(20.5);
double error = estimation.error(20.5);
```

The class `PolynomialLeastSquareFit` is organized like the linear regression class: data points are accumulated directly on each call as each value is not needed. One drawback is that the value of the sum of equation 10.26 must be computed by another object when the χ^2-test of the fit is required. The advantage is that the implementation of the class is very simple. The class `PolynomialLeastSquareFit` has one constructor method whose argument is the degree of the polynomial used in the fit.

The main accumulation method, `accumulateWeightedPoint`, takes the arguments x_i, y_i, and $1/\sigma_i^2$ in this order. Other constructor methods are convenience methods calling the method `accumulateWeightedPoint` to do the real work:

`accumulatePoint(double,double,double)` accumulates points with error; that is, the arguments are x_i, y_i and σ_i in this order.

`accumulateBin(double,int)` accumulates the bin contents of a histogram; the first argument is the middle of the bin and the second the bin contents; the weight is computed as described in Section 10.4.

`accumulateAverage(double,StatisticalMoments)` accumulates points whose y value is determined from an average over several measurements; the first argument is x_i; y_i and σ_i are obtained, respectively, from the average and error on average of the statistical moments.

`accumulatePoint(double,double)` accumulates a point with a weight of 1; the arguments are x_i and y_i.

The method `evaluate` returns an estimated polynomial corresponding to the fit. The system of equation 10.45 is solved using LUP decomposition since the dimension of

the matrix is usually small.[9] The calculation of the error matrix is obtained from the LUP decomposition. To avoid possible rounding errors, the off-diagonal elements are made exactly symmetrical before being passed to the constructor method of a symmetrical matrix.

Listing 10.18 Java implementation of a polynomial least square fit

```
package DhbEstimation;

import DhbFunctionEvaluation.PolynomialFunction;
import DhbMatrixAlgebra.LUPDecomposition;
import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.SymmetricMatrix;
import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbMatrixAlgebra.DhbNonSymmetricComponents;
import DhbInterfaces.PointSeriesWithErrors;
import DhbStatistics.StatisticalMoments;

// Polynomial least square fit

// @author Didier H. Besset

public class PolynomialLeastSquareFit
{
    double[][] systemMatrix;
    double[] systemConstants;

// Constructor method.

public PolynomialLeastSquareFit( int n)
{
    int n1 = n + 1;
    systemMatrix = new double[n1][n1];
    systemConstants = new double[n1];
    reset();
}

// @param x double
// @param m StatisticalMoments

public void accumulateAverage( double x, StatisticalMoments m)
{
    accumulatePoint( x, m.average(), m.errorOnAverage());
```

9. Fitted polynomials of very high degree have the same bad behavior as the interpolation polynomials.

```
}

// @param x double
// @param n int    bin content

public void accumulateBin( double x, int n)
{
    accumulateWeightedPoint( x, n, 1.0 / Math.max( 1, n));
}

// @param x double
// @param y double

public void accumulatePoint( double x, double y)
{
    accumulateWeightedPoint( x, y, 1);
}

// @param x double
// @param y double
// @param error double    standard deviation on y

public void accumulatePoint( double x, double y, double error)
{
    accumulateWeightedPoint( x, y, 1.0 / (error * error));
}

// @param x double
// @param y double
// @param w double    weight of point

public void accumulateWeightedPoint( double x, double y, double w)
{
    double xp1 = w;
    double xp2;
    for ( int i = 0; i < systemConstants.length; i++ )
    {
        systemConstants[i] += xp1 * y;
        xp2 = xp1;
        for ( int j = 0; j <= i; j++ )
        {
            systemMatrix[i][j] += xp2;
            xp2 *= x;
        }
        xp1 *= x;
    }
}
```

```
    // return DhbEstimation.EstimatedPolynomial

public EstimatedPolynomial evaluate()
{
    for ( int i = 0; i < systemConstants.length; i++ )
    {
        for ( int j = i + 1; j < systemConstants.length; j++ )
            systemMatrix[i][j] = systemMatrix[j][i];
    }
    try {
        LUPDecomposition lupSystem = new LUPDecomposition(
                                                        systemMatrix);
        double [][] components = lupSystem.inverseMatrixComponents();
        LUPDecomposition.symmetrizeComponents( components);
        return new EstimatedPolynomial(
                        lupSystem.solve( systemConstants),
                        SymmetricMatrix.fromComponents( components));
        } catch ( DhbIllegalDimension e) {}
            catch ( DhbNonSymmetricComponents ex) {};
     return null;
}
public void reset()
{
    for ( int i = 0; i < systemConstants.length; i++ )
    {
        systemConstants[i] = 0;
        for ( int j = 0; j < systemConstants.length; j++ )
            systemMatrix[i][j] = 0;
    }
}
}
```

Listing 10.19 shows the implementation of the class `EstimatedPolynomial`, which is a subclass of the class `Polynomial` containing the error matrix of the fit performed to make an estimation of the polynomial's coefficients. The class `EstimatedPolynomial` returns the estimated error of its value based on the error matrix of the fit using equation 10.33.

Listing 10.19 Java implementation of a polynomial with error

```
package DhbEstimation;

import DhbFunctionEvaluation.PolynomialFunction;
import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.SymmetricMatrix;
import DhbMatrixAlgebra.DhbIllegalDimension;
```

```
// Polynomial with error estimation

// @author Didier H. Besset

public class EstimatedPolynomial extends PolynomialFunction
{

    // Error matrix.

    SymmetricMatrix errorMatrix;

// Constructor method.
// @param coeffs double[]
// @param e double[]    error matrix

public EstimatedPolynomial(double[] coeffs, SymmetricMatrix e)
{
    super(coeffs);
    errorMatrix = e;
}

// @return double// @param x double

public double error( double x)
{
    int n = degree() + 1;
    double[] errors = new double[n];
    errors[0] = 1;
    for ( int i = 1; i < n; i++)
        errors[i] = errors[i-1] * x;
    DhbVector errorVector = new DhbVector( errors);
    double answer;
    try { answer = errorVector.product(
                                errorMatrix.product( errorVector));
        } catch (DhbIllegalDimension e) { answer = Double.NaN;};
    return Math.sqrt( answer);
}
}
```

10.9 Least-Square Fit with Nonlinear Dependence

In the case of a nonlinear function, the fit can be reduced to a linear fit and a search by successive approximations.

Let us assume that we have an approximate estimation $\mathbf{p}_0$ of the parameters $\mathbf{p}$. Let us define the vector $\Delta\mathbf{p} = \mathbf{p} - \mathbf{p}_0$. One redefines the function $F(\mathbf{x}, \mathbf{p})$ as

$$F(\mathbf{x}, \mathbf{p}) = F(\mathbf{x}, \mathbf{p}_0) + \left.\frac{\partial F(\mathbf{x}, \mathbf{p})}{\partial \mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_0} \cdot \Delta\mathbf{p}. \tag{10.48}$$

Equation 10.48 is a linear expansion[10] of the function $F(\mathbf{x}, \mathbf{p})$ around the vector $\mathbf{p}_0$ respective to the vector $\mathbf{p}$. In equation 10.48, $\left.\frac{\partial F(\mathbf{x},\mathbf{p})}{\partial \mathbf{p}}\right|_{\mathbf{p}=\mathbf{p}_0}$ is the gradient of the function $F(\mathbf{x}, \mathbf{p})$ relative to the vector $\mathbf{p}$ evaluated for $\mathbf{p} = \mathbf{p}_0$; this is a vector with the same dimension as the vector $\mathbf{p}$. Then, one minimizes the expression in equation 10.26 respective to the vector $\Delta\mathbf{p}$. This is, of course, a linear problem as described in Section 10.6. Equation 10.30 becomes

$$\mathbf{M} \cdot \Delta\mathbf{p} = \mathbf{c}, \tag{10.49}$$

where the components of the matrix $\mathbf{M}$ are now defined by

$$M_{jk} = \sum_{i=1}^{N} \frac{1}{\sigma_i^2} \cdot \left.\frac{\partial F(\mathbf{x}_i, \mathbf{p})}{\partial p_j}\right|_{\mathbf{p}=\mathbf{p}_0} \cdot \left.\frac{\partial F(\mathbf{x}_i, \mathbf{p})}{\partial p_k}\right|_{\mathbf{p}=\mathbf{p}_0} \tag{10.50}$$

and the components of the vector $\mathbf{c}$ are defined by

$$c_j = \sum_{i=1}^{N} \frac{y_i - F(\mathbf{x}_i, \mathbf{p}_0)}{\sigma_i^2} \cdot \left.\frac{\partial F(\mathbf{x}_i, \mathbf{p})}{\partial p_j}\right|_{\mathbf{p}=\mathbf{p}_0}. \tag{10.51}$$

The vector $\Delta\mathbf{p}$ is obtained by solving equation 10.49 using the algorithms described in Sections 8.2 and 8.3. Then, we can use the vector $\mathbf{p}_0 + \Delta\mathbf{p}$ as the new estimate and repeat the whole process. One can show[11] that iterating this process converges toward the vector $\bar{\mathbf{p}}$ minimizing the function $S(\mathbf{p})$ introduced in equation 10.26.

As explained in Section 10.6, the inverse of the matrix $\mathbf{M}$ is the error matrix containing the variance of each parameter and their correlation. The expression for the estimated variance on the function $F(\mathbf{x}, \mathbf{p})$ becomes

$$\text{var}\left[F(\mathbf{x}, \mathbf{p})\right] = \sum_{j=1}^{m}\sum_{k=1}^{m} M_{jk}^{-1} \cdot \frac{\partial F(\mathbf{x}_i, \bar{\mathbf{p}})}{\partial p_j} \cdot \frac{\partial F(\mathbf{x}_i, \bar{\mathbf{p}})}{\partial p_k}. \tag{10.52}$$

A careful examination of the error matrix can tell whether the fit is meaningful.

Figure 10.8 shows an example of a least square fit performed on a histogram with a probability density function. The histogram of Figure 10.8 was generated using a random generator distributed according to a Fisher-Tippett distribution (see Section D.4) with parameters $\alpha = 0$ and $\beta = 1$. Only 1,000 events have been

10. That is, the first couple of terms of a Taylor expansion of the function $F(\mathbf{x}, \mathbf{p})$ around the vector $\mathbf{p}_0$ in an m-dimensional space.

11. A mathematically oriented reader can see that this is a generalization of the Newton zero-finding algorithm (see Section 5.3)to m dimensions .

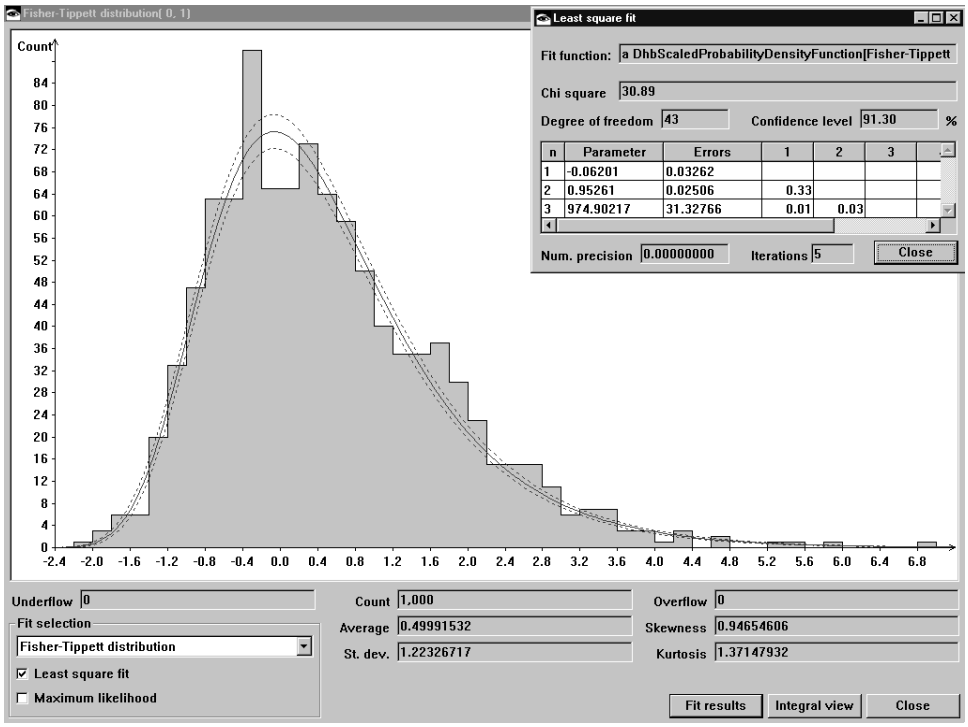

FIG. 10.8 Example of a least-square fit

accumulated into the histogram. The inset window in the upper right corner shows the fit results. The order of the parameter is α, β, and the number of generated events. The solid curve laid onto the histogram is the prediction of the fitted function; the two dotted lines indicate the error on the prediction. The reader can verify that the fit is excellent. The number of needed iterations is quite low: the convergence of the algorithm is quite good in general.

10.9.1 Nonlinear Fit—General Implementation

As we have seen, the solution of a nonlinear fit can be approximated by successive approximations. Thus, nonlinear fits are implemented with a subclass of the iterative process class described in Section 4.1. Data points must be kept in a structure maintained by the object implementing linear least square fit to be readily available at each iteration. Thus, the data points are kept in an instance variable.

The results of the iterative process are the parameters. Our implementation assumes that the supplied function contains and maintains its parameter. Thus, the instance variable corresponding to the result of the iterative process is the fit function itself. The parameters determined by the fit—the result proper—are kept within the object implementing the supplied function. In particular, the determination of the

initial values for the iterative process is the responsibility of the fit function. Thus, the method `initializeIterations` does not do anything.

In most cases, the number of parameters in a least-square fits is relatively small. Thus, LUP decomposition (described in Section 8.3) is sufficient to solve equation 10.49 at each iteration. Except for the last iteration, there is no need to compute the error matrix (the inverse of the matrix **M**). The components of the error matrix can be obtained from the LUP decomposition when the algorithm converges.

Convergence is attained when the largest of the relative variation of the components of the vector becomes smaller than a given value. In addition to the fact that we are dealing with floating-point numbers, the reason for using relative precision is that the components of the vector **p** usually have different ranges.

When the fit has been obtained, convenience methods allows retrieving the sum of equation 10.25 (`chiSquare`) and the confidence level of the fit (`confidenceLevel`). Another convenience method, `valueAndError`, computes the prediction of the fit and its estimated error using equation 10.52.

Note: The details of implementation are quite different between Smalltalk and Java. I therefore advise the reader to read both. Some are admittedly arbitrary; some are caused by the specific features of each language. A good exercise is to explain the reasons for the differences.

10.9.2 Nonlinear Fit—Smalltalk Implementation

Listing 10.20 shows the complete implementation in Smalltalk. Code Example 10.9 shows how the data of Figure 10.8 were generated (up to the plotting facilities).

Code Example 10.9

```
| genDistr hist fit |
hist := DhbHistogram new.
hist freeExtent: true.
genDistr := DhbFisherTippettDistribution shape: 0 scale: 1.
1000 timesRepeat: [ hist accumulate: genDistr random].
fit := DhbLeastSquareFit histogram: hist
           distributionClass: DhbFisherTippettDistribution.
fit evaluate.
```

The first two lines after the declaration define an instance of class `DhbHistogram` with automatic adjustment of the limits (see Section 9.3.2). The next line defines an instance of a Fisher-Tippett distribution. Then, 1,000 random numbers generated according to this distribution are accumulated into the histogram. Next, an instance of the class `DhbLeastSquareFit` is defined with the histogram for the data points and the desired class of the probability density function. The corresponding scaled probability is created within the method (see Listing 10.20). The final line performs the fit proper. After this line, the calling application can either use the fit object or extract the fit result to make predictions with the fitted distribution.

The class DhbLeastSquareFit is a subclass of DhbIterativeProcess described in Section 4.1.1. It has the following instance variables:

dataHolder is the object containing the experimental data; this object must implement the iterator method pointsAndErrorsDo:; the block supplied to the iterator method takes as argument an instance of the class DhbWeightedPoint described in Section 10.3.3.

equations contains the components of the matrix **M**.

constants contains the components of the vector **c**.

errorMatrix contains the LUP decomposition of the matrix **M**,

chiSquare contains the sum of equation 10.25.

degreeOfFreedom contains the degree of freedom of the fit.

The instance variables errorMatrix, chiSquare, and degreeOfFreedom are implemented using lazy initialization. The method finalizeIterations sets the instance variables equations and constants to nil to reclaim space at the end of the fit.

The supplied fit function—instance variable result—must implement the method valueAndGradient:, which returns an array containing the value of the function at the supplied argument and the gradient vector. This is an optimization because the gradient can be computed frequently using intermediate results coming from the computation of the function's value.

The method valueAndError: is a good example of using the vector and matrix operations described in Chapter 8.

Listing 10.20 Smalltalk implementation of a nonlinear least-square fit

Class	DhbLeastSquareFit
Subclass of	DhbIterativeProcess
Instance variable names:	dataHolder errorMatrix chiSquare equations constants degreeOfFreedom

Class Methods

histogram: aHistogram **distributionClass:**

aProbabilityDensityFunctionClass

```
^self points: aHistogram
    function: (DhbScaledProbabilityDensityFunction histogram:
                                                        aHistogram
            distributionClass: aProbabilityDensityFunctionClass)
```

points: aDataHolder **function:** aParametricFunction

```
^aParametricFunction ifNotNil: [ :dp | super new initialize:
                                          aDataHolder data: dp]
```

Instance Methods

accumulate: aWeightedPoint

```
| f g |
f := result valueAndGradient: aWeightedPoint xValue.
g := f last.
f := f first.
constants accumulate: g * ( ( aWeightedPoint yValue - f) *
                                          aWeightedPoint weight).
1 to: g size do:
    [ :k |
      ( equations at: k) accumulate: g * ( ( g at: k) *
                                          aWeightedPoint weight).
    ].
```

accumulateEquationSystem

```
dataHolder pointsAndErrorsDo: [ :each | self accumulate: each].
```

chiSquare

```
chiSquare isNil
    ifTrue: [ self computeChiSquare].
^chiSquare
```

computeChanges

```
errorMatrix := DhbLUPDecomposition direct: equations.
^errorMatrix solve: constants
```

computeChiSquare

```
chiSquare := 0.
degreeOfFreedom := self numberOfFreeParameters negated.
dataHolder pointsAndErrorsDo:
    [ :each |
      chiSquare := ( each chi2Contribution: result) + chiSquare.
      degreeOfFreedom := degreeOfFreedom + 1.
    ].
```

computeEquationSystem

```
constants atAllPut: 0.
equations do: [ :each | each atAllPut: 0].
```

```
self accumulateEquationSystem.
```

confidenceLevel

```
^( DhbChiSquareDistribution degreeOfFreedom: self
                  degreeOfFreedom) confidenceLevel: self chiSquare
```

degreeOfFreedom

```
degreeOfFreedom isNil
    ifTrue: [ self computeChiSquare].
^degreeOfFreedom
```

errorMatrix

```
^DhbSymmetricMatrix rows: errorMatrix inverseMatrixComponents
```

evaluateIteration

```
| changes maxChange|
self computeEquationSystem.
changes := self computeChanges.
result changeParametersBy: changes.
maxChange := 0.
result parameters with: changes do:
    [ :r :d | maxChange := ( d / r) abs max: maxChange].
^maxChange
```

finalizeIterations

```
equations := nil.
constants := nil.
degreeOfFreedom := nil.
chiSquare := nil
```

fitType

```
^'Least square fit'
```

initialize: aDataHolder **data:** aParametricFunction

```
dataHolder := aDataHolder.
result := aParametricFunction.
^self
```

initializeIterations

```
| n |
n := self numberOfParameters.
constants := (DhbVector new: n)
```

```
            atAllPut: 0;
            yourself.
equations := (1 to: n) collect:
                [:k |
                (DhbVector new: n)
                    atAllPut: 0;
                    yourself]
```

numberOfFreeParameters

```
^self numberOfParameters
```

numberOfParameters

```
^result parameters size
```

value: aNumber

```
^result value: aNumber
```

valueAndError: aNumber

```
| valueGradient |
valueGradient := result valueAndGradient: aNumber.
^Array with: valueGradient first
       with: ( valueGradient last * ( self errorMatrix *
                                          valueGradient last)) sqrt
```

10.9.3 Nonlinear Fit—Java Implementation

In Java, the fit function must be declared as an interface. Here we need not only to compute the functions value but also to value the gradient of the function respective to the vector of parameters; in addition, the parameters of the fit function must be changed. Thus, the interface `OneVariableFunction` is not sufficient as it only covers retrieving the function's value. A new interface `ParametrizedOneVariableFunction` (see Listing 10.21) was derived from the interface `OneVariableFunction` to handle the management of the function's parameters. This interface has the following instance methods:

`valueAndGradient` returns an array containing the value of the function for the supplied argument in the first position; the rest of the array contains the components of the function's gradient or the same argument; in other words, it returns the array $F\left(x, \mathbf{p}_0\right), \left.\frac{\partial F(x,\mathbf{p})}{\partial p_1}\right|_{\mathbf{p}=\mathbf{p}_0}, \ldots, \left.\frac{\partial F(x,\mathbf{p})}{\partial p_m}\right|_{\mathbf{p}=\mathbf{p}_0}$ for the supplied argument x (let us recall that m is the number of parameters).

`setParameters` defines the value of the function's parameters; in other words, this method is used to set the value of $\mathbf{p}$ in the fit function.

Listing 10.21 Java parametrized function interface

```
package DhbInterfaces;

// ParametrizedOneVariableFunction is an interface for mathematical
// functions of one variable depending on several parameters,
// that is functions of the form f(x;p), where p is a vector.

// @author Didier H. Besset

public interface ParametrizedOneVariableFunction
                                          extends OneVariableFunction
{

// @return double[]    array containing the parameters

double[] parameters();

// @param p double[]    assigns the parameters

void setParameters( double[] p);

// Evaluate the function and the gradient of the function with respect
// to the parameters.
// @return double[]    0: function's value, 1,2,...,n function's gradient
// @param x double

double[] valueAndGradient( double x);
}
```

Note: This implementation of the least-square fit is only[12] for functions with a real argument (a `double`). However, the mathematical derivation I gave is valid for an argument x of any type. As an exercise, the reader can generalize this implementation for a function of a vector (or array) argument.

Listing 10.22 shows the implementation of a least-square fit in Java. Code Example 10.10 shows how Figure 10.8 was generated up to the plotting facilities.

Code Example 10.10

```
FisherTippettDistribution genDistr = new
   FisherTippettDistribution( 0, 1);
Histogram histogram = new Histogram();
histogram.setGrowthAllowed();
for ( int i = 0; i < 1000; i++ )
```

12. This is not the case in Smalltalk since one does not need to define explicit types.

```
      histogram.accumulate( genDistr.random());
   WeightedPoint[] wps = new WeightedPoint[histogram.size()];
   for ( int i = 0; i < wps.length; i++)
      wps[i] = histogram.weightedPointAt(i);
   FisherTippettDistribution fitDistr =
                           new FisherTippettDistribution( histogram);
   LeastSquareFit fit = new LeastSquareFit( wps,
          new ScaledProbabilityDensityFunction( fitDistr, histogram));
   fit.evaluate();
   ScaledProbabilityDensityFunction fittedScaledDistr =
                  (ScaledProbabilityDensityFunction) fit.getResult();
```

The first line creates an instance of a Fisher-Tippett distribution with parameters $\alpha = 0$ and $\beta = 1$. The next two lines create an instance of a histogram with an automatic adjustment of the limits. Then a loop accumulates random numbers generated according to the distribution into the histogram. After data have been accumulated, an array of weighted points is created, and the bins of the histogram are collected into the array of weighted points. Next, the variable `fitDistr` is created on the histogram. This constructor method calculates rough estimates of the distribution's parameters based on the histogram's content, thus providing initial values for the fit. The last two statements perform the fit and retrieve, respectively, of the fitted function as an instance of `ScaledProbabilityDensityFunction`. A cast is necessary since the type of the object returned by the method `getResult` is defined only by an interface.

It is not necessary to retrieve the result as a scaled probability density function: after the fit has been performed, the parameters of the distribution stored in variable `fitDistr` are the fitted parameters because my implementation keeps the fit parameters in the supplied function object. In other words, the variable `fitDistr` contains the fitted probability density function.
The class `LeastSquareFit` has the following instance variables:

`result` The fit function; this object must implement the `ParametrizedOneVariableFunction` interface described at the beginning of this section.

`points` An array of weighted points containing the data, over which the fit is made.

`systemMatrix` The components of the matrix **M** calculated at each iterations.

`systemConstants` The components of the vector **c** calculated at each iterations.

`systemLUP` An LUP decomposition object (the class `LUPDecomposition`) is described in Section 8.3) needed to solve equation 10.49 at each iterations.

`errorMatrix` The error matrix of the fit, (i.e., the inverse of the matrix **M** obtained at the last iteration).

`chiSquare` The sum of equation 10.26.

`degreeOfFreedom` the degree of freedom of the fit.

The class `LeastSquareFit` has two constructor methods, each with two arguments. The arguments of the first method are respectively an array of weighted points(the class `WeightedPoint` described in Section 10.3.3) representing the data, over which the fit is performed, and the fit function. The second method is a convenience method used to define the fit of a probability density function (second argument) over a histogram (first argument). In code example 10.10 one could have replaced the generation of the weighted points and the definition of the fit function by the following line:

```
LeastSquareFit fit = new LeastSquareFit( histogram, fitDistr);
```

Many methods may seem redundant. Their presence will become clear once the reader discovers the implementation of the maximum-likelihood fit described in Section 10.24. In the maximum-likelihood fit, the weighted points are taken directly from a histogram. Thus, initializing the system (method `initializeSystem`) resetting the system at each iteration (method `resetSystem`), the number of data points (method `getDataSetSize`), and accessing each data point (method `weightedPointAt`) have been implemented with the upcoming subclassing[13] in mind.

The method `evaluateIteration` first calculates the matrix $\mathbf{M}$ and the vector $\Delta\mathbf{p}$ of equation 10.49. Then, this system of linear equations is solved using LUP decomposition (described in Section 8.3). Then, the obtained vector $\Delta\mathbf{p}$ is added to the vector $\mathbf{p}$ to get a new estimation of the parameters. This estimation is fed back into the fit function. The precision returned by the method `evaluateIteration` is the largest of the relative change of the parameter's components.

The method `accumulate` calculates each term of the series of equations 10.58 and 10.59. To optimize speed, only the lower half of the symmetrical matrix $\mathbf{M}$ is accumulated. At the end of the summation, the method `symmetrizeMatrix` is used to fill up the upper half of the matrix.

A series of methods allows retrieving information after the fit has been performed. They are all working with lazy initialization. When the fit is terminated, the method `finalizeIterations` sets the instance variables `systemMatrix` and `systemConstants` to release no longer needed storage. It also sets the values of instance variables `errorMatrix`, `chiSquare`, and `degreeOfFreedom` to undefined values to force the computation of these when their respective accessor methods are called. The reader will note that space used by the LUP decomposition is reclaimed as soon as the error matrix is computed since it is no longer needed.

Listing 10.22 Java implementation of a nonlinear least-square fit

```
package DhbEstimation;
```

13. For Smalltalkers: these methods must be declared `protected` since private methods are not inherited in Java.

```
import DhbIterations.IterativeProcess;
import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbMatrixAlgebra.DhbNonSymmetricComponents;
import DhbMatrixAlgebra.LUPDecomposition;
import DhbMatrixAlgebra.SymmetricMatrix;
import DhbStatistics.ChiSquareDistribution;
import DhbInterfaces.ParametrizedOneVariableFunction;
import DhbScientificCurves.Histogram;
import DhbStatistics.ScaledProbabilityDensityFunction;
import DhbStatistics.ProbabilityDensityFunction;

// Non-linear least square fit

// @author Didier H. Besset

public class LeastSquareFit extends IterativeProcess
{
    protected ParametrizedOneVariableFunction result;
    private WeightedPoint[] points;
    protected double[][] systemMatrix;
    protected double[] systemConstants;
    private LUPDecomposition systemLUP;
    private SymmetricMatrix errorMatrix;
    private double chiSquare;
    private int degreeOfFreedom;

// Default constructor method (internal use only)

protected LeastSquareFit()
{
}

// Constructor method
// @param n int

public LeastSquareFit(WeightedPoint[] pts,
                                ParametrizedOneVariableFunction f)
{
    points = pts;
    result = f;
    initializeSystem( result.parameters().length);
}

// Constructor method
// @param histogram Histogram
// @param distr ProbabilityDensityFunction
```

```
public LeastSquareFit(Histogram histogram,
                                    ProbabilityDensityFunction distr)
{
    points = new WeightedPoint[histogram.size()];
    for ( int i = 0; i < points.length; i++)
        points[i] = histogram.weightedPointAt(i);
    result = new ScaledProbabilityDensityFunction( distr, histogram);
    initializeSystem( result.parameters().length);
}

// @param wp DhbEstimation.WeightedPoint

protected void accumulate( WeightedPoint wp)
{
    double[] fg = result.valueAndGradient( wp.xValue());
    for( int i = 0; i < systemConstants.length; i++ )
    {
        systemConstants[i] += ( wp.yValue() - fg[0])
                              * fg[i+1] * wp.weight();
        for( int j = 0; j <= i; j++ )
            systemMatrix[i][j] += fg[i+1] * fg[j+1] * wp.weight();
    }
}

// Append the name of the fit to the supplied string buffer
// @param sb java.lang.StringBuffer

protected void appendFitName( StringBuffer sb)
{
    sb.append("Least square fit with ");
}

// Append the results of the fit to the supplied string buffer
// @param sb java.lang.StringBuffer

private void appendFitResults( StringBuffer sb)
{
    java.text.DecimalFormat fmt =
                        new java.text.DecimalFormat("###0.00000");
    java.text.DecimalFormat corFmt =
                               new java.text.DecimalFormat("0.000");
    sb.append('\n');
    sb.append("\tcompleted in ");
    sb.append( getIterations());
    sb.append(" iterations\n");
    sb.append("\tParams\tErrors\tCorrelation");
```

```
        double[][] comp = errorMatrix().toComponents();
        double[] params = result.parameters();
        double[] errors = new double[comp.length];
        char separator;
        for ( int i = 0; i < comp.length; i++ )
        {
            sb.append("\n\t");
            sb.append( fmt.format(params[i]));
            errors[i] = Math.sqrt(comp[i][i]);
            sb.append("\t+-");
            sb.append( fmt.format(errors[i]));
            separator = '\t';
            for ( int j = 0; j < i; j++ )
            {
                sb.append(separator);
                sb.append(' ');
                sb.append(corFmt.format(comp[i][j] /
                                        (errors[i] * errors[j])));
            }
        }
        appendNormalization( sb);
        sb.append("\n\tChi square =");
        sb.append( fmt.format(chiSquare()));
        sb.append("\tDegree of freedom =");
        sb.append( degreeOfFreedom());
        sb.append("\tConfidence level =");
        sb.append( corFmt.format(confidenceLevel()));
    }

    // This method does nothing (compatibility with maximum likelihood fit)
    // @param sb java.lang.StringBuffer

    protected void appendNormalization( StringBuffer sb) { }

    // @return double

    public double chiSquare()
    {
        if ( Double.isNaN( chiSquare) )
            computeChiSquare();
        return chiSquare;
    }

    // @return double[] changes on parameters

    protected double[] computeChanges()
```

```
{
    return systemLUP.solve( systemConstants);
}

// Compute the chi^2 of the fit function.

private void computeChiSquare()
{
    chiSquare = 0;
    for( int i = 0; i < getDataSetSize(); i++ )
        chiSquare += weightedPointAt(i).chi2Contribution( result);
}

// @return DhbMatrixAlgebra.SymmetricMatrix

private void computeErrorMatrix()
{
    double [][] components = systemLUP.inverseMatrixComponents();
    LUPDecomposition.symmetrizeComponents( components);
    try { errorMatrix = SymmetricMatrix.fromComponents( components);
          systemLUP = null;
        }
        catch ( DhbNonSymmetricComponents e) {}
        catch ( DhbIllegalDimension ex) {};
}
private void computeSystem()
{
    resetSystem();
    for ( int i = 0; i < getDataSetSize(); i++ )
        accumulate( weightedPointAt(i));
    symmetrizeMatrix();
}

// @return double    confidence level of the fit.

public double confidenceLevel()
{
    return (new ChiSquareDistribution(
                    degreeOfFreedom())).confidenceLevel( chiSquare());
}

// @return long    the degree of freedom of the fit.

public int degreeOfFreedom()
{
    if ( degreeOfFreedom < 0 )
```

```
        degreeOfFreedom = getDataSetSize() -
                                        result.parameters().length;
    return degreeOfFreedom;
}

// @return DhbMatrixAlgebra.SymmetricMatrix    the error matrix of the fit.

public SymmetricMatrix errorMatrix()
{
    if ( errorMatrix == null )
        computeErrorMatrix();
    return errorMatrix;
}

// @return double

public double evaluateIteration()
{
    double[] parameters = result.parameters();
    computeSystem();
    try { systemLUP = new LUPDecomposition( systemMatrix);}
        catch( DhbIllegalDimension e) {};
    double[] changes = computeChanges();
    double eps = 0;
    for ( int i = 0; i < parameters.length; i++ )
    {
        parameters[i] += changes[i];
        eps = Math.max( eps, Math.abs(
                    relativePrecision( changes[i], parameters[i])));
    }
    result.setParameters( parameters);
    return eps;
}
public void finalizeIterations()
{
    systemMatrix = null;
    systemConstants = null;
    errorMatrix = null;
    chiSquare = Double.NaN;
    degreeOfFreedom = -1;
}

// @return int    number of data points.

protected int getDataSetSize()
{
```

```
    return points.length;
}

// @return ParametrizedOneVariableFunction    the fitted function

public ParametrizedOneVariableFunction getResult()
{
    return result;
}

// @param n int

protected void initializeSystem( int n)
{
    systemConstants = new double[ n];
    systemMatrix = new double[n][n];
}
protected void resetSystem( )
{
    for( int i = 0; i < systemConstants.length; i++ )
    {
        systemConstants[i] = 0;
        for( int j = 0; j <= i; j++ )
            systemMatrix[i][j] = 0;
    }
}
private void symmetrizeMatrix( )
{
    for( int i = 0; i < systemConstants.length; i++ )
    {
        for( int j = 0; j < i; j++ )
            systemMatrix[j][i] = systemMatrix[i][j];
    }
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    appendFitName(sb);
    sb.append( getResult());
    if ( hasConverged() )
        appendFitResults(sb);
    return sb.toString();
}
```

```
// @return DhbEstimation.WeightedPoint n-th weighted data point
// @param n int

protected WeightedPoint weightedPointAt( int n)
{
    return points[n];
}
}
```

10.10 Maximum-Likelihood Fit of a Probability Density Function

In Section 9.3, histograms have been discussed as a way to represent a probability density function directly from experimental data. In this section, I shall show that the maximum-likelihood estimation can easily be applied to the data gathered in a histogram to determine the parameters of a hypothesized probability density function.

In general, the maximum-likelihood fit of a probability density function to a histogram is much faster than the corresponding least-square fit because the number of free parameters is lower, as we shall see in this section. In addition, the maximum-likelihood estimation is unbiased and is therefore a better estimation than the least-square fit estimation, especially when the histogram is sparsely populated. Thus, a maximum-likelihood fit is the preferred way of finding the parameters of a probability density function from experimental data collected in a histogram.

Let m be the number of bins in the histogram, and let n_i be the content of the ith bin. Let $P_i(\mathbf{p})$ the probability of observing a value in the ith bin. The likelihood function $L(\mathbf{p})$ is the probability of observing the particular histogram. Since the hypothesis of a probability density function does not constrain the total number of values collected into the histogram, the total number of collected values can be considered constant. As a consequence, a maximum-likelihood fit has one parameter less than a least-square fit using the same function. Since the total number is unconstrained, the probability of observing the particular histogram is given by a multinomial probability. Thus, the likelihood function can be written as

$$L(\mathbf{p}) = N! \prod_{i=1}^{m} \frac{P_i(\mathbf{p})^{n_i}}{n_i!}, \tag{10.53}$$

where $N = \sum_{i=1}^{m} n_i$ is the total number of values collected into the histogram. As we have seen in Section 10.5.1, finding the maximum of $L(\mathbf{p})$ is equivalent to finding the maximum of the function $I(\mathbf{p})$. Since N is a constant, we use a renormalized function:

$$I(\mathbf{p}) = \ln \frac{M(\mathbf{p})}{N!} = \sum_{i=1}^{m} n_i \ln P_i(\mathbf{p}) . \tag{10.54}$$

Finding the maximum of the function $I(\mathbf{p})$ is equivalent to solving the following system of nonlinear equations:

$$\frac{\partial I(\mathbf{p})}{\partial \mathbf{p}} = \sum_{i=1}^{m} \frac{n_i}{P_i(\mathbf{p})} \cdot \frac{\partial P_i(\mathbf{p})}{\partial \mathbf{p}} = 0. \tag{10.55}$$

This system can be solved with a search by successive approximations, where a system of linear equations must be solved at each step. The technique used is similar to the one described in Section 10.9. In this case, however, it is more convenient to expand the inverse of the probability density function around a previous approximation as follows:

$$\frac{1}{P_i(\mathbf{p})} = \frac{1}{P_i(\mathbf{p}_0)} - \frac{1}{P_i(\mathbf{p}_0)^2} \cdot \left. \frac{\partial P_i(\mathbf{p})}{\partial \mathbf{p}} \right|_{\mathbf{p}=\mathbf{p}_0} \cdot \Delta\mathbf{p}. \tag{10.56}$$

This expansion can only be defined over a range where the probability density function is not equal to zero. Therefore, this expansion of the maximum-likelihood estimation cannot be used on a histogram where bins with nonzero count are located on a range where the probability density function is equal to zero.[14] Contrary to a least square fit, bins with zero count do not participate to the estimation.

Now equation 10.55 becomes a system of linear equations of the type

$$\mathbf{M} \cdot \Delta\mathbf{p} = \mathbf{c}, \tag{10.57}$$

where the components of the matrix $\mathbf{M}$ are now defined by

$$M_{jk} = \sum_{i=1}^{m} \frac{n_i}{P_i(\mathbf{p}_0)^2} \cdot \left. \frac{\partial P_i(\mathbf{p}_0)}{\partial p_j} \right|_{\mathbf{p}=\mathbf{p}_0} \cdot \left. \frac{\partial P_i(\mathbf{p}_0)}{\partial p_k} \right|_{\mathbf{p}=\mathbf{p}_0} \tag{10.58}$$

and those of the vector $\mathbf{c}$ by

$$c_j = \sum_{i=1}^{m} \frac{n_i}{P_i(\mathbf{p}_0)} \cdot \left. \frac{\partial P_i(\mathbf{p}_0)}{\partial p_j} \right|_{\mathbf{p}=\mathbf{p}_0} . \tag{10.59}$$

As discussed at the beginning of this section, the maximum-likelihood estimation for a histogram cannot determine the total count in the histogram. The estimated total count, $\bar{N}$, is estimated with the following hypothesis:

$$n_i = \bar{N} P(\bar{\mathbf{p}}) , \tag{10.60}$$

14. Equation 10.53 shows that the bin over which the probability density function is zero give no information.

where $\bar{\mathbf{p}}$ is the maximum-likelihood estimation of the distribution parameters. The estimation is performed using $\bar{N}$ as the only variable. The maximum-likelihood estimation cannot be solved analytically, however, the least-square estimation can.

As we have seen in Section 10.4, the variance of the bin count is the estimated bin content. Thus, the function to minimize becomes

$$S\left(\bar{N}\right) = \sum_{i=1}^{m} \frac{\left[n_i - \bar{N}P_i\left(\bar{\mathbf{p}}\right)\right]^2}{\bar{N}P_i\left(\bar{\mathbf{p}}\right)} \tag{10.61}$$

The value of $\bar{N}$ minimizing the expression of equation 10.61 is

$$\bar{N} = \sqrt{\frac{\sum_{i=1}^{m} n_i^2/P_i\left(\bar{\mathbf{p}}\right)}{\sum_{i=1}^{m} P_i\left(\bar{\mathbf{p}}\right)}}, \tag{10.62}$$

and the estimated error on $\bar{N}$ is given by

$$\sigma_{\bar{N}} = \sqrt{\frac{\sum_{i=1}^{m} n_i^2/P_i\left(\bar{\mathbf{p}}\right)}{2\bar{N}}}. \tag{10.63}$$

After computing $\bar{N}$ using equation 10.62, the goodness of the maximum likelihood fit can be estimated by calculating the χ^2 confidence level of $S\left(\bar{N}\right)$ given by equation 10.61.

Figure 10.9 shows an example of a maximum-likelihood fit performed on the same histogram as in Figure 10.8. The inset window in the upperright corner shows the fit results in the same order as in Figure 10.8. The correlation coefficients, however, are not shown for the normalization since it is not determined as part of the fit. The solid curve laid onto the histogram is the prediction of the fitted function; the two dotted lines indicate the error on the prediction. The reader can see that the fit is as good as the least-square fit. Of course, the χ^2 test is significantly higher with a correspondingly lower confidence level. This mostly comes from the fact that a maximum likelihood fit does not use the bins with zero count. In fact, the reader can see that the count in the histogram (normalization) estimated by the maximum-likelihood fit is higher than in the case of the least-square fit.

10.10.1 Maximum-Likelihood Fit—General Implementation

A maximum-likelihood fit of a probability density function on a histogram is very similar to a least-square fit of a histogram with a scaled probability distribution. There are two major differences: first, the number of parameters is lower; second, the computation of the matrix and vectors is not the same. Otherwise, most of the structure of a least square fit can be reused.

Instead of creating special methods to compute the gradient of the fitted function using a new set of parameters, my implementation uses the same gradient calculation as the one used by the least-square fit. This approach is possible if the component

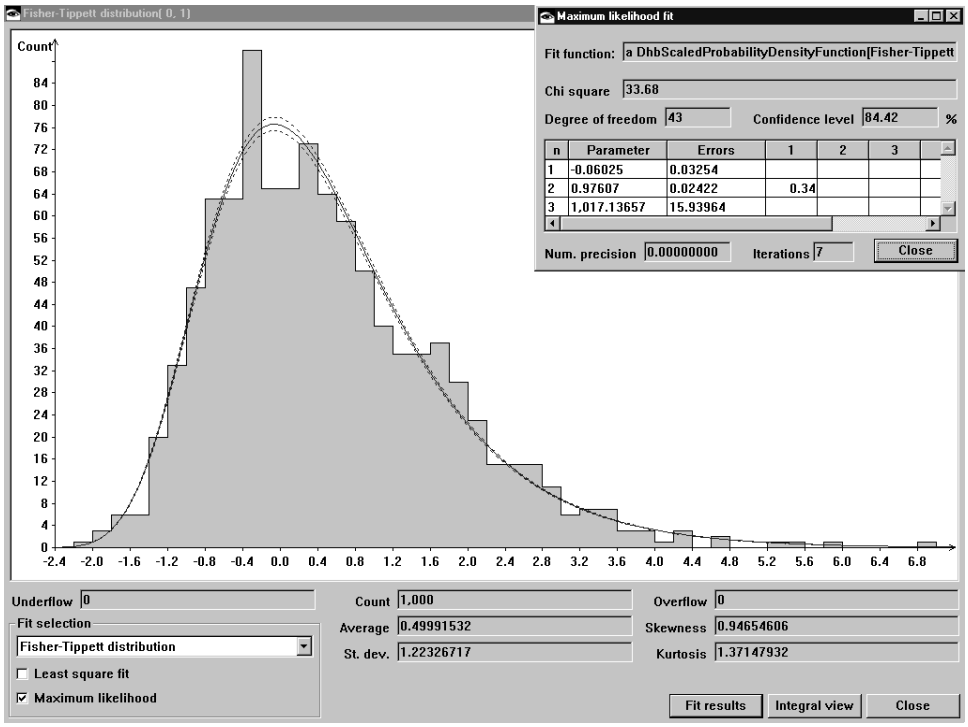

FIG. 10.9 Example of a maximum-likelihood fit

of the gradient relative to the normalization is placed at the end. Since the computation of this component does not require additional calculation, the additional time required by reusing the gradient's computation is negligible. Since the fit function is a scaled probability distribution, the current normalization is kept in an instance variable, and the normalization of the fitted function is set to 1 for the duration of the iterations. When the algorithm is completed, the estimated normalization is put back into the fit function.

The computation of the normalization (equation 10.62) and that of its error (equation 10.63) is performed in the method `finalizeIterations`.

10.10.2 Maximum-likelihood Fit—Smalltalk Implementation

Listing 10.23 shows the complete implementation in Smalltalk. Code Example 10.11 shows how Figure 10.9 was generated up to the plotting facilities.

Code Example 10.11

```
| genDistr hist fit |
hist := DhbHistogram new.
hist freeExtent: true.
```

```
genDistr := DhbFisherTippettDistribution shape: 0 scale: 1.
1000 timesRepeat: [ hist accumulate: genDistr random].
fit :=DhbMaximumLikekihoodHistogramFit histogram: hist
          distributionClass: DhbFisherTippettDistribution.
fit evaluate.
```

As the reader can see, the only difference with Code Example 10.9 is the name of the class in the statement where the instance of the fit is created.

The class `DhbMaximumLikekihoodHistogramFit` is a sublcass of the class `DhbLeastSquareFit`. It has the following additional instance variables:

`count` The estimated normalization, (i.e., $\bar{N}$)

`countVariance` The estimated variance of $\bar{N}$.

The variance is kept instead of the error because the most frequent use of this quantity is in computing the estimated error on the predicted value. In the method `valueAndError:`, this computation requires the combination of the error of the fit (i.e., equation 10.33) with the error on the normalization. An accessor method is provided for the variable `count`. The method `normalizationError` calculates the error on the normalization.

The method `accumulate:` uses the vector operations to calculate the terms of the sums in equations 10.58 and 10.59. Because of the lower number of parameters, the routine `computeChanges:` places in the vector Δ_p an additional zero element corresponding to the normalization in the case of the least-square fit.

The method `finalizeIterations` calculates the estimated value of the normalization (equation 10.61) and its variance (square of equation 10.62). After this, it sets the obtained normalization into the scaled probability distribution.

Listing 10.23 Smalltalk implementation of a maximum-likelihood fit

Class	`DhbMaximumLikekihoodHistogramFit`
Subclass of	`DhbLeastSquareFit`
Instance variable names:	`count countVariance`

Instance Methods

accumulate: `aWeightedPoint`

```
| f g temp inverseProbability|
f := result valueAndGradient: aWeightedPoint xValue.
g := f last copyFrom: 1 to: ( f last size - 1).
f := f first.
f = 0 ifTrue: [ ^nil].
inverseProbability := 1 / f.
```

```
temp := aWeightedPoint yValue * inverseProbability.
constants accumulate: g * temp.
temp := temp * inverseProbability.
1 to: g size do:
    [ :k |
      ( equations at: k) accumulate: g * ( ( g at: k) * temp).
    ].
```

computeChanges

```
^super computeChanges copyWith: 0
```

computeNormalization

```
| numerator denominator temp |
numerator := 0.
denominator := 0.
dataHolder pointsAndErrorsDo:
        [:each |
        temp := result value: each xValue.
        temp = 0
            ifFalse:
                [numerator := numerator + (each yValue squared /
                                                                temp).
                denominator := denominator + temp]].
count := ( numerator / denominator) sqrt.
countVariance := numerator / ( 4 * count).
```

finalizeIterations

```
self computeNormalization.
result setCount: count.
super finalizeIterations
```

fitType

```
^'Maximum likelihood fit'
```

initializeIterations

```
result setCount: 1.
count := dataHolder totalCount.
super initializeIterations
```

normalization

```
^count
```

normalizationError

```
^countVariance sqrt
```

numberOfFreeParameters

```
^super numberOfParameters
```

numberOfParameters

```
^super numberOfParameters - 1
```

valueAndError: aNumber

```
| valueGradient gradient gVar |
valueGradient := result valueAndGradient: aNumber.
gradient := valueGradient last copyFrom: 1 to: valueGradient last
                                                        size - 1.
gVar := gradient * (self errorMatrix * gradient) / count.
^Array with: valueGradient first
    with: ((valueGradient first / count) squared * countVariance
                                                    + gVar) sqrt
```

10.10.3 Maximum-Likelihood Fit—Java Implementation

Listing 10.24 provides the complete implementation in Java. Code Example 10.12 shows how Figure 10.9 was generated up to the plotting facilities.

Code Example 10.12

```
FisherTippettDistribution genDistr = new FisherTippettDistribution(
   0, 1);
Histogram histogram = new Histogram();
histogram.setGrowthAllowed();
for ( int i = 0; i < 1000; i++ )
   histogram.accumulate( genDistr.random());
FisherTippettDistribution fitDistr = new FisherTippettDistribution(
   histogram);
MaximumLikelihoodHistogramFit fit = new
   MaximumLikelihoodHistogramFit( histogram, fitDistr);
fit.evaluate();
```

In this example, the constructor method using a histogram and a probability density function is used. Otherwise, it is very similar to Code Example 10.10.

The class `MaximumLikelihoodHistogramFit` is a subclass of class `LeastSquareFit`. It has the following additional instance variables:

`histogram` The histogram over which the fit is performed; because of strong typing, it was not possible to reuse the variable `points`.

`count` The estimated normalization, (i.e., $\bar{N}$)

`countError` The estimated error on $\bar{N}$

The class `MaximumLikelihoodHistogramFit` has only one constructor method corresponding to the convenience constructor method of the class `LeastSquareFit`. Assignment of the instance variable is different since data are accessed directly in the histogram.

The method `accumulate` calculates the terms of the sums in equations 10.58 and 10.59. The methods `getDataSetSize`, `computeChanges`, and `weightedPointAt` overloads the corresponding methods of class `LeastSquareFit` to reflect the fact that the data points are obtained directly from the histogram.

The method `finalizeComputation` first calls the method `computeNormalization`, which calculates the estimation of the normalization and its error using equations 10.62 and 10.63. Then the obtained normalization is set into the fitted function. Unused space is released in the method of the superclass.

Listing 10.24 Java implementation of a maximum likelihood fit

```
package DhbEstimation;

import DhbInterfaces.ParametrizedOneVariableFunction;
import DhbScientificCurves.Histogram;
import DhbStatistics.ScaledProbabilityDensityFunction;
import DhbStatistics.ProbabilityDensityFunction;

// Maximum likelihood fit

// @author Didier H. Besset

public class MaximumLikelihoodHistogramFit extends LeastSquareFit
{

// Histogram containing the data

    private Histogram histogram;

// Estimated total count in histogram

    private double count;

// Estimated error on total count in histogram
```

```
    private double countError;

// Constructor method.
// @param pts Histogram
// @param f DhbInterfaces.ParametrizedOneVariableFunction

public MaximumLikelihoodHistogramFit(Histogram hist,
                                     ProbabilityDensityFunction f)
{
    histogram = hist;
    result = new ScaledProbabilityDensityFunction ( f, hist);
    initializeSystem( f.parameters().length);
}

// @param wp DhbEstimation.WeightedPoint

protected void accumulate( WeightedPoint wp)
{
    double[] fg = result.valueAndGradient( wp.xValue());
    if ( fg[0] == 0 )
        return;
    double invProb = 1  / fg[0];
    double temp = wp.yValue() * invProb;
    for( int i = 0; i < systemConstants.length; i++ )
    {
        systemConstants[i] += fg[i+1] * temp;
        for( int j = 0; j <= i; j++ )
            systemMatrix[i][j] += fg[i+1] * fg[j+1] * temp * invProb;
    }
}

// Append the name of the fit to the supplied string buffer
// @param sb java.lang.StringBuffer

protected void appendFitName( StringBuffer sb)
{
    sb.append("Maximum likelihood fit with ");
}

// Append the normalization and its error to the fit results
// @param sb java.lang.StringBuffer

protected void appendNormalization( StringBuffer sb)
{
    java.text.DecimalFormat fmt = new java.text.DecimalFormat("###0.0");
    sb.append("\n\t");
```

```
        sb.append( fmt.format( count));
        sb.append("\t+-");
        sb.append( fmt.format( countError));
    }

    // Computes the changes in the parameters:
    // since the normalization is not fitted, the change to the
    // normalization (last parameter) is set to zero.

    protected double[] computeChanges()
    {
        double[] changes = super.computeChanges();
        double[] answer = new double[changes.length+1];
        for ( int i = 0; i < changes.length; i++ )
            answer[i] = changes[i];
        answer[changes.length] = 0;
        return answer;
    }

    // Computes the estimated normalization and variance on it.

    private void computeNormalization()
    {
        double numerator = 0;
        double denominator = 0;
        double temp;
        WeightedPoint wp;
        for ( int i = 0; i < getDataSetSize(); i++ )
        {
            wp = weightedPointAt(i);
            temp = result.value( wp.xValue());
            if ( temp != 0 )
                {
                    numerator += wp.yValue() * wp.yValue() / temp;
                    denominator += temp;
                }
        }
        count = Math.sqrt( numerator / denominator);
        countError = Math.sqrt( 0.25 * numerator / count);
    }
    public void finalizeIterations()
    {
        computeNormalization();
        getDistribution().setCount( count);
        super.finalizeIterations();
    }
```

```
// @return int    number of data points.

protected int getDataSetSize()
{
    return histogram.size();
}

// @return ScaledProbabilityDensityFunction    the fitted function

public ScaledProbabilityDensityFunction getDistribution()
{
    return (ScaledProbabilityDensityFunction) getResult();
}
public void initializeIterations()
{
    getDistribution().setCount( 1);
    count = histogram.totalCount();
}

// @return DhbEstimation.WeightedPoint n-th weighted data point
// @param n int

protected WeightedPoint weightedPointAt( int n)
{
    return histogram.weightedPointAt(n);
}
}
```

CHAPTER 11

Optimization

Cours vite au but, mais gare à la chute.[1]

—Alexandre Solzhenitsyn

An *optimization problem* is a numerical problem for which the solution is characterized by the largest or smallest value of a numerical function depending on several parameters. Such function is often called the *goal function*. Many kinds of problems can be expressed in optimization—that is, finding the maximum or the minimum of a goal function. This technique has been applied to a wide variety of fields, from operation research to game playing and artificial intelligence. In Chapter 10, for example, the solution of maximum-likelihood or least square fits was obtained by finding the maximum, and minimum, respectively, of a function.

In fact, generations of high-energy physicists have used the general-purpose minimization program MINUIT[2] written by Fred James[3] of CERN to perform least-square fits and maximum-likelihood fits. To achieve generality, MINUIT uses several strategies to reach a minimum. In this chapter, we shall discuss a few techniques and conclude with a program quite similar in spirit to MINUIT. Our version, however, will not have all the features offered by MINUIT.

If the goal function can be expressed with an analytical form, the problem of optimization can be reduced into calculating the derivatives of the goal function respective to all parameters—a tedious but manageable job. In most cases, however, the goal function cannot always be expressed analytically.

The classes described in this chapter are different in Smalltalk and Java. Therefore I present two class diagrams: Figure 11.1 shows the Smalltalk class diagram, and Figure 11.2 shows the Java class diagram. The main reason for the

1. Run fast to the goal, but beware of the fall.

2. F. James and M. Roos, *MINUIT—A System for Function Minimization and Analysis of the Parameter Errors and Corrections*, Comput. Phys. Commun., 10 (1975) 343–367.

3. I take this opportunity to thank Fred for the many useful discussions we have had on the subject of minimization.

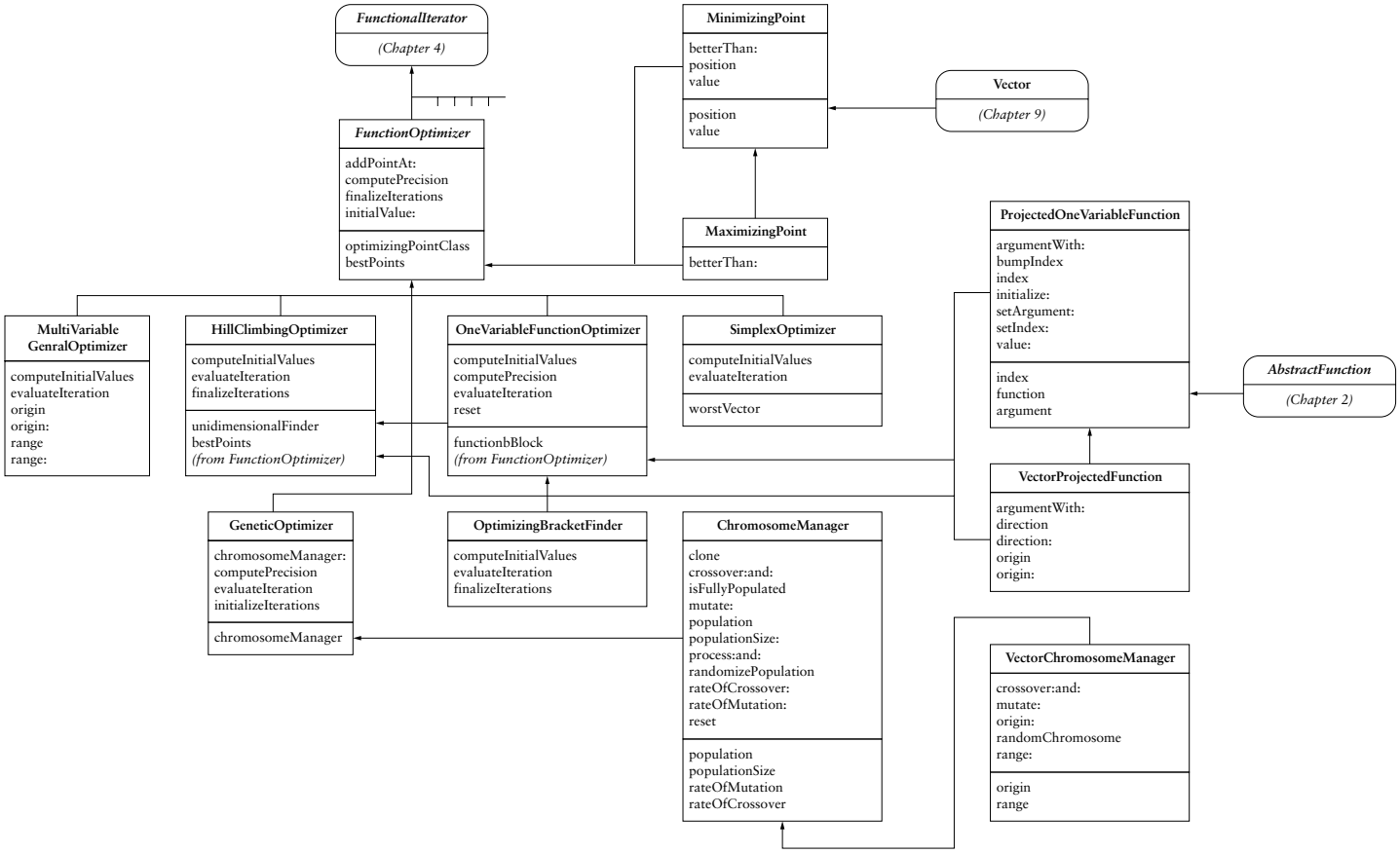

FIG. 11.1 Smalltalk classes used in optimization

art/be11f02.eps

FIG. 11.2 Java classes used in optimization

difference is the strong typing imposed in Java preventing the reuse of instance variables.

11.1 Introduction

Let us state the problem in general terms. Let $f(\mathbf{x})$ be a function of a vector $\mathbf{x}$ of dimension n. The n-dimensional space is called the *search space* of the problem. Depending on the problem, the space can be continuous or not. In this section, we shall assume that the space is continuous.

If the function is derivable, the gradient of the function respective to the vector $\mathbf{x}$ must vanish at the optimum. Finding the optimum of the function can be replaced by the problem of finding the vector $\mathbf{x}$ such that

$$\frac{df(\mathbf{x})}{d\mathbf{x}} = 0. \tag{11.1}$$

Unfortunately, equation 11.1 is not a necessary condition for an optimum. It can be a maximum, a minimum, or a saddle point—that is, a point where the function has a minimum in one projection and a maximum in another projection. Furthermore, the function may have several optima. Figure 11.3 shows an example of a function

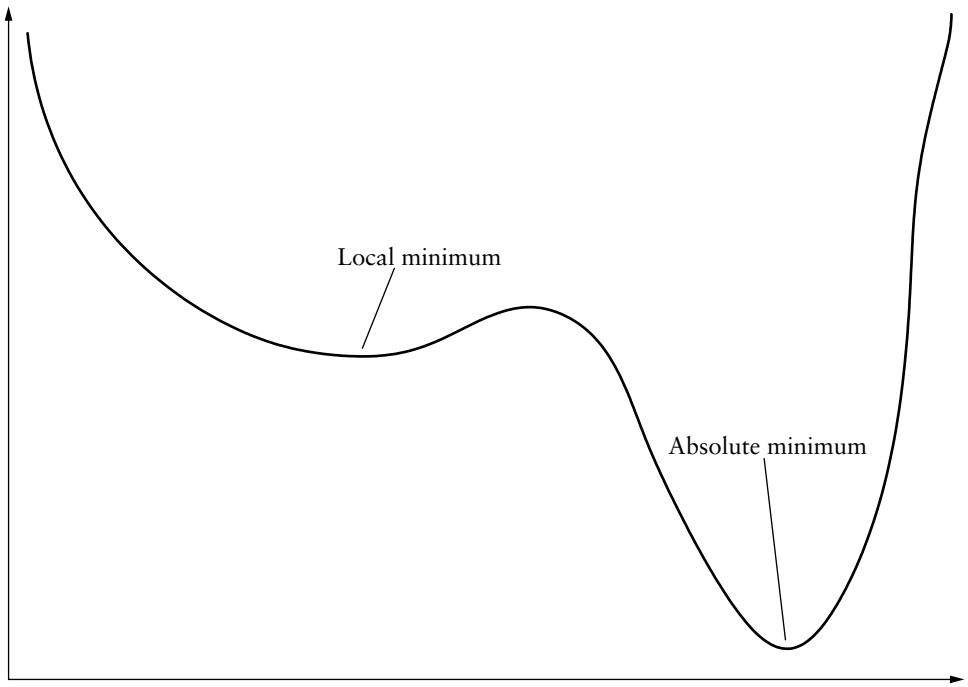

FIG. 11.3 Local and absolute optima

TABLE 11.1 Optimizing algorithms presented in this chapter

Name	Category	Derivatives
Extended Newton	greedy	yes
Powell's hill climbing	greedy	no
Simplex	greedy	no
Genetic algorithm	random based	no

having two minima. Some problems require finding the absolute optimum of the function. Thus, one must verify that the solution of equation 11.1 corresponds indeed to an optimum with the expected properties. The reader can already see at this point that searching for an optimum in the general case is a very difficult task.

All optimization algorithms can be classified into two broad categories:

- *Greedy algorithms*: These algorithms are characterized by a local search in the most promising direction. They are usually efficient and quite good at finding local optima. Among greedy algorithms, one must distinguish those requiring the evaluation of the function's derivatives.
- *Random-based algorithms*: These algorithms are using a random approach. They are not efficient; however, they are good at finding absolute optima. Simulated annealing [Press *et al.*] and genetic algorithms [Berry & Linoff] belong to this class.

Table 11.1 lists the properties of the algorithms presented in this chapter.

11.2 Extended Newton Algorithms

Extended Newton algorithms are using a generalized version of Newton's zero-finding algorithm. These algorithms assume that the function is continuous and has only one optimum in the region where the search is initiated.

Let us expand the function $f(\mathbf{x})$ around a point $\mathbf{x}^{(0)}$ near the solution. We have in components

$$f(\mathbf{x}) = f\left[\mathbf{x}^{(0)}\right] + \sum_j \left.\frac{\partial f(\mathbf{x})}{\partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} \left[x_j - x_j^{(0)}\right]. \tag{11.2}$$

Using this expansion above into equation 11.1 yields

$$\sum_j \left.\frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} \left[x_j - x_j^{(0)}\right] + \left.\frac{\partial f(\mathbf{x})}{\partial x_i}\right|_{\mathbf{x}=\mathbf{x}^{(0)}} = 0. \tag{11.3}$$

Equation 11.3 can be written as a system of linear equations of the form

$$\mathbf{M}\Delta = \mathbf{c}, \tag{11.4}$$

where $\Delta_j = x_j - x_j^{(0)}$. The components of the matrix $\mathbf{M}$—called the *Hessian matrix*—are given by

$$m_{ij} = \left.\frac{\partial^2 f\ (\mathbf{x})}{\partial x_i \partial x_j}\right|_{\mathbf{x}=\mathbf{x}^{(0)}}, \tag{11.5}$$

and the components of the vector $\mathbf{c}$ are given by

$$c_i = -\left.\frac{\partial f\ (\mathbf{x})}{\partial x_i}\right|_{\mathbf{x}=\mathbf{x}^{(0)}}. \tag{11.6}$$

As in Section 10.9, one can iterate this process by replacing $\mathbf{x}^{(0)}$ with $\mathbf{x}^{(0)} + \Delta$. This process is actually equivalent to the Newton zero-finding method (see Section 5.3). The final solution is a minimum if the matrix $\mathbf{M}$ is positive definite; otherwise, it is a maximum.

This technique is used by MINUIT in the vicinity of the goal function's optimum. It is the region where the algorithm described earlier works well. Far from the optimum, the risk of reaching a point where the matrix $\mathbf{M}$ cannot be inverted is quite high in general. In addition, the extended Newton algorithm requires that the second order derivatives of the function can be computed analytically. At least the first-order derivatives must be provided, otherwise, the cost of computation at each step becomes prohibitive. A concrete implementation of the technique is not given here. The reader can find in this book all the necessary tools to make such an implementation, which is left as an exercise for the reader. In the rest of this chapter, I shall present other methods that work without an analytical knowledge of the function.

11.3 Hill-Climbing Algorithms

Hill climbing is a generic term covering many algorithms trying to reach an optimum by determining the optimum along successive directions. The general algorithm is outlined as follows:

1. Select an initial point $\mathbf{x}_0$ and a direction $\mathbf{v}$.
2. Find $\mathbf{x}_1$, the optimum of the function along the selected direction
3. If convergence is attained, terminate the algorithm.
4. Set $\mathbf{x}_0 = \mathbf{x}_1$, select a different direction, and go back to step 2.

The simplest of these algorithms simply follows each axis in turn until a convergence is reached. More elaborate algorithms exist [Press *et al.*]. One of them is described in Section 11.6.

Hill-climbing algorithms can be applied to any continuous function, especially when the function's derivatives are not easily calculated. The core of the hill-climbing algorithm is finding the optimum along one direction. Let $\mathbf{v}$ be the direction; then finding the optimum of the vector function $f(\mathbf{x})$ along the direction $\mathbf{v}$ starting from point $\mathbf{x}_0$ is equivalent to finding the optimum of the one-variable function $g(\lambda) = f(\mathbf{x}_0 + \lambda\mathbf{v})$.

Therefore, to implement a hill-climbing algorithm, we first need to implement an algorithm able to find the optimum of a one-variable function. This is the topic of Sections 11.4 and 11.5. Before this, we need to discuss the implementation details providing a common framework to all classes considered in the rest of this chapter.

11.3.1 Optimizing—General Implementation

At this point, the reader may be a little puzzled by the use of *optimum* instead of speaking of *minimum* or *maximum*. I shall now disclose a general implementation that works for finding both a minimum or a maximum. This should not come as a surprise since, in mathematics, a minimum and a maximum are very similar—positions where the derivative of a function vanishes—and can be easily turned into each other (e.g., by negating the function).

To implement a general-purpose optimizing framework, I introduce two new classes: `MinimizingPoint` and `MaximizingPoint`, a subclass of `MinimizingPoint`. These two classes are used as STRATEGY by the optimizing algorithms. The class `MinimizingPoint` has two instance variables:

`value` The value of the goal function, (i.e., $g(\lambda)$ or $f(\mathbf{x})$)

`position` The position at which the function has been evaluated, (i.e., λ or $\mathbf{x}$).

The class `MinimizingPoint` contains most of the methods. The only method overloaded by the class `MaximizingPoint` is the method `betterThan`, which tells whether an optimizing point is better than another. The method `betterThan` can be used in all parts of the optimizing algorithms to find out which point is the optimum so far. In algorithms working in multiple dimensions, the method `betterThan` is also used to sort the points from the best to the worst. In Java, the architecture is a little more complex because of typing requirements, but the basic design concept is the same.

A convenience instance creation method allows creating instances for a given function with a given argument. The instance is then initialized with the function's value evaluated at the argument. Thus, all optimizing algorithms described here do not call the goal function explicitly.

Otherwise, the implementation of the one-dimensional optimum search uses the general framework of the iterative process. More specifically, it uses the class `FunctionalIterator` described in Section 4.2.

A final remark concerns the method `initializeIteration`. The golden search algorithm assumes that the three points λ_0, λ_1, and λ_2 have been determined. What

if they have not been? In this case, the method `initializeIteration` uses the optimum bracket finder described in Section 11.5.

11.3.2 Common Optimizing Classes—Smalltalk Implementation

In Smalltalk, we have two classes of optimizing points: `DhbMinimizingPoint` and its subclass `DhbMaximizingPoint`. These classes are shown in Listing 11.1. The class `DhbFunctionOptimizer` is in charge of handling the management of the optimizing points. This class is shown in Listing 11.2.

The class `DhbMinimizingPoint` has the following instance variables:

`position` contains the position at which the function is evaluated; this instance variable is a number if the function to optimize is a one variable function and an array or a vector if the function to evaluate is a function of many variables.

`value` contains the value of the function evaluated at the point's position.

Accessor methods corresponding to these variables are supplied. As noted in Section 11.3.1, the only method redefined by the subclass `DhbMaximizingPoint` is the method `betterThan:` used to decide whether a point is better than another.

Optimizing points are created with the convenience method `vector:function:`, which evaluates the function supplied as second argument at the position supplied as the first argument.

Listing 11.1 Smalltalk classes common to all optimizing classes

Class	`DhbMinimizingPoint`
Subclass of	`Object`
Instance variable names:	`value position`

Class Methods

new: `aVector` **value:** `aNumber`

```
^self new vector: aVector; value: aNumber; yourself
```

vector: `aVector` **function:** `aFunction`

```
^self new: aVector value: (aFunction value: aVector)
```

Instance Methods

betterThan: `anOptimizingPoint`

```
^value < anOptimizingPoint value
```

position

```
^position
```

printOn: aStream

```
position printOn: aStream.
aStream
    nextPut: $:;
    space.
value printOn: aStream
```

value

```
^value
```

value: aNumber

```
value := aNumber.
```

vector: aVector

```
position := aVector
```

Class	DhbMaximizingPoint
Subclass of	DhbMinimizingPoint

Instance Methods

betterThan: anOptimizingPoint

```
^value > anOptimizingPoint value
```

The class DhbFunctionOptimizer is in charge of handling the optimizing points and has the following instance variables:

optimizingPointClass is the class of the optimizing points used as STRATEGY by the optimizer; it is used to create instances of points at a given position for a given function.

bestPoints contains a sorted collection of optimizing points; the best point is the first and the worst point is the last; all optimizers rely on the fact that sorting is done by this sorted collection.

The method addPointAt: creates an optimizing point at the position supplied as argument and adds this point to the collection of best points. Since that collection is sorted, one is always certain to find the best result in the first position. This fact is used by the method finalizeIterations, which retrieves the result from the collection of best points.

Instances of the function optimizer are created with the two convenience methods `minimizingFuntion:` and `maximizingFuntion:` helping to define the type of optimum. An additional convenience method, `forOptimizer:`, allows creating a new optimizer with the same strategy—that is, the same class of optimizing points—and the same function as the optimizer supplied as argument. This method is used to create optimizers used in intermediate steps.

Because finding an optimum cannot be determined numerically with high precision [Press *et al.*], the class `DhbFunctionOptimizer` redefines the method `defaultPrecision` to be 100 times the default numerical precision.

Listing 11.2 Smalltalk abstract class for all optimizing classes

Class	`DhbFunctionOptimizer`
Subclass of	`DhbFunctionalIterator`
Instance variable names:	`optimizingPointClass bestPoints`

Class Methods

defaultPrecision

```
^super defaultPrecision * 100
```

forOptimizer: `aFunctionOptimizer`

```
^self new initializeForOptimizer: aFunctionOptimizer
```

maximizingFunction: `aFunction`

```
^super new initializeAsMaximizer; setFunction: aFunction
```

minimizingFunction: `aFunction`

```
^super new initializeAsMinimizer; setFunction: aFunction
```

Instance Methods

addPointAt: `aNumber`

```
bestPoints add: (optimizingPointClass vector: aNumber
                                     function: functionBlock)
```

bestPoints

```
^bestPoints
```

finalizeIterations

```
result := bestPoints first position.
```

functionBlock

```
^functionBlock
```

initialize

```
bestPoints := SortedCollection sortBlock:
                                        [ :a :b | a betterThan: b].
^super initialize
```

initializeAsMaximizer

```
optimizingPointClass := DhbMaximizingPoint.
^self initialize
```

initializeAsMinimizer

```
optimizingPointClass := DhbMinimizingPoint.
^self
```

initializeForOptimizer: `aFunctionOptimizer`

```
optimizingPointClass := aFunctionOptimizer pointClass.
functionBlock := aFunctionOptimizer functionBlock.
^self initialize
```

initialValue: `aVector`

```
result := aVector copy.
```

pointClass

```
^optimizingPointClass
```

printOn: `aStream`

```
super printOn: aStream.
bestPoints do: [ :each | aStream cr. each printOn: aStream].
```

To find an optimum along a given direction, one needs to construct an object able to transform a vector function into a one variable function. The class `DhbProjectedOneVariableFunction` and its subclass `DhbVectorProjectedFunction` provide this functionality. They are shown in Listing 11.3. The class `DhbProjectedOneVariableFunction` has the following instance variables:

`function` The goal function $f(\mathbf{x})$

`argument` The vector argument of the goal function, (i.e., the vector $\mathbf{x}$)

`index` The index of the axis along which the function is projected

The instance variables `argument` and `index` can be read and modified using direct accessor methods. The goal function is set only at creation time: the instance creation method `function:` take the goal function as argument. A convenience method `bumpIndex` allows altering the index in circular fashion.[4]

The class `DhbVectorProjectedFunction` has no additional variables. Instead, it reuses the instance variable `index` as the direction along which the function is evaluated. For clarity, the accessor methods have been renamed `direction`, `direction:`, `origin`, and `origin:`.

For both classes, the method `argumentAt:` returns the argument vector for the goal function, (i.e., the vector **x**). The method `value:` returns the value of the function $g(\lambda)$ for the supplied argument λ.

Listing 11.3 Smalltalk projected function classes

Class	`DhbProjectedOneVariableFunction`
Subclass of	`Object`
Instance variable names:	`index function argument`

Class Methods

function: `aVectorFunction`

```
^super new initialize: aVectorFunction
```

Instance Methods

argumentWith: `aNumber`

```
^argument at: index put: aNumber; yourself
```

bumpIndex

```
index isNil
    ifTrue: [ index := 1]
    ifFalse:[ index := index + 1.
              index > argument size
                ifTrue: [ index := 1].
            ].
```

---

4. We do not give the implementation of the simplest of the hill-climbing algorithms using alternatively each axis of the reference system. This implementation, which uses the method `bumpIndex`, is left as an exercise for the reader.

index

```
index isNil
    ifTrue: [ index := 1].
^index
```

initialize: aFunction

```
function := aFunction.
^self
```

setArgument: anArrayOrVector

```
argument := anArrayOrVector copy.
```

setIndex: anInteger

```
index := anInteger.
```

value: aNumber

```
^function value: ( self argumentWith: aNumber)
```

Class	DhbVectorProjectedFunction
Subclass of	DhbProjectedOneVariableFunction

Instance Methods

argumentWith: aNumber

```
^aNumber * self direction + self origin
```

direction

```
^index
```

direction: aVector

```
index := aVector.
```

origin

```
^argument
```

origin: aVector

```
argument := aVector.
```

printOn: aStream

```
self origin printOn: aStream.
```

```
aStream nextPutAll: ' ('.
self direction printOn: aStream.
aStream nextPut: $).
```

11.3.3 Common Optimizing Classes—Java Implementation

Java has two distinct hierarchies of optimizing points (see Figure 11.2). The abstract class `OptimizingPoint` has two subclasses, `MinimizingPoint` and `MaximizingPoint`. These classes are shown in Listing 11.4. The abstract class `OptimizingPoint` has two instance variables:

`position` a `double` contains the position at which the function is evaluated.

`value` a `double` contains the value of the function evaluated at the point's position.

Both instance variables have a corresponding getter accessor method but no setter method. Otherwise, the functional relation between `position` and `value` could not be ensured. As noted in Section 11.3.1, the only method redefined by the subclass `MaximizingPoint` is the method `betterThan` used to decide whether a point is better than another. Instances of optimizing points are created with a single constructor method taking as arguments the position at which the function must be evaluated and the function itself. The supplied function must implement the `OneVariableFunction` interface described in Section 2.1.2.

Listing 11.4 Java optimizing point classes

```
package DhbOptimizing;

import DhbInterfaces.OneVariableFunction;

// Point & function holder used in optimizing one-variable functions.

// @author Didier H. Besset

public abstract class OptimizingPoint
{

    // Value of the function to optimize.

    private double value;

    // Position at which the value was evaluated.

    private double position;
```

```
// Constructor method
// @param x double    position at which the goal function is evaluated.
// @param f OneVariableFunction    function to optimize.

public OptimizingPoint( double x, OneVariableFunction f)
{
    position = x;
    value = f.value(x);
}

// @return boolean    true if the receiver is "better" than
//                                  the supplied point
// @param point OptimizingPoint

public abstract boolean betterThan( OptimizingPoint entity);

// @return double    the receiver's position

public double getPosition()
{
    return position;
}

// @return double  the value of the function at the receiver's position

public double getValue()
{
    return value;
}

// (used by method toString).
// @return java.lang.String

protected abstract String printedKey();

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( value);
    sb.append(printedKey());
    sb.append( position);
    return sb.toString();
}
}
```

```
package DhbOptimizing;

import DhbInterfaces.OneVariableFunction;

// Point & function holder used in minimizing one-variable functions.

// @author Didier H. Besset

public class MinimizingPoint extends OptimizingPoint {

// Constructor method.
// @param x double
// @param f DhbInterfaces.OneVariableFunction

public MinimizingPoint(double x, OneVariableFunction f)
{
    super(x, f);
}

// @return boolean    true if the receiver is "better" than
//                                the supplied point
// @param point OptimizingPoint

public boolean betterThan(OptimizingPoint point)
{
    return getValue() < point.getValue();
}

// (used by method toString).
// @return java.lang.String

protected final String printedKey()
{
    return " min@";
}
}
```

```
package DhbOptimizing;

import DhbInterfaces.OneVariableFunction;

// Point & function holder used in maximizing one-variable functions.

// @author Didier H. Besset

public class MaximizingPoint extends OptimizingPoint {
```

```
// Constructor method.
// @param x double
// @param f DhbInterfaces.OneVariableFunction

public MaximizingPoint(double x, OneVariableFunction f)
{
    super(x, f);
}

// @return boolean    true if the receiver is "better" than
//                                    the supplied point
// @param point OptimizingPoint

public boolean betterThan(OptimizingPoint point)
{
    return getValue() > point.getValue();
}

// (used by method toString).
// @return java.lang.String

protected final String printedKey()
{
    return " max@";
}
}
```

The abstract class `OptimizingVector` has two subclasses, `MinimizingVector` and `Maximizingvector`. The only differences from the corresponding optimizing point classes are that the instance variable `position` is an array of `double` and that the function supplied as the second argument of the constructor method must implement the `ManyVariableFunction` interface. These classes and the interface `ManyVariableFunction` are shown in Listing 11.5. The need for the additional hierarchy comes primarily from the fact that we cannot handle `double` and an array of `double` as a common object since these entities are primitive types and not objects.

Listing 11.5 Java optimizing vector classes

```
package DhbInterfaces;

// ManyVariableFunction is an interface for mathematical functions
// of many variables, that is functions of the form:
//            f(X) where X is a vector.

// @author Didier H. Besset
```

```
public interface ManyVariableFunction
{

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

public double value ( double[] x);
}

package DhbOptimizing;

import DhbInterfaces.ManyVariableFunction;

// Vector & function holder used in optimizing many-variable functions.

// @author Didier H. Besset

public abstract class OptimizingVector
{

    // Value of the function to optimize.

    private double value;

    // Position at which the value was evaluated.

    private double[] position;

    // Value of the function to optimize.

    protected ManyVariableFunction f;

// Constructor method.
// @param v double[]
// @param f DhbInterfaces.OneVariableFunction

public OptimizingVector(double[] v, ManyVariableFunction func)
{
    position = v;
    f = func;
    value = f.value( position);
}

// @return boolean    true if the receiver is "better" than
//                                  the supplied point
// @param point OptimizingVector

public abstract boolean betterThan( OptimizingVector entity);
```

```
    // (used by the Simplex algorithm).
    // @param v double[]

    public void contractFrom( double[] v)
    {
        for ( int i = 0; i < position.length; i++ )
        {
            position[i] += v[i];
            position[i] *= 0.5;
        }
        value = f.value( position);
    }

    // @return double    the receiver's position

    public double[] getPosition()
    {
        return position;
    }

    // @return double    the value of the function
    //                           at the receiver's position

    public double getValue()
    {
        return value;
    }

    // (used by method toString).
    // @return java.lang.String

    protected abstract String printedKey();

    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        sb.append( value);
        sb.append(printedKey());
        for ( int i = 0; i < position.length; i++ )
        {
            sb.append( ' ');
            sb.append( position[i]);
        }
        return sb.toString();
```

```
}
}

package DhbOptimizing;

import DhbInterfaces.ManyVariableFunction;

// Vector & function holder used in minimizing many-variable functions.

// @author Didier H. Besset

public class MinimizingVector extends OptimizingVector {

// Constructor method.
// @param v double[]
// @param f DhbInterfaces.ManyVariableFunction

public MinimizingVector(double[] v,
                        DhbInterfaces.ManyVariableFunction f){
   super(v, f);
}

// @return boolean   true if the receiver is "better" than
//                   the supplied point
// @param point OptimizingVector

public boolean betterThan(OptimizingVector point)
{
    return getValue() < point.getValue();
}

// (used by method toString).
// @return java.lang.String

protected final String printedKey()
{
    return " min@";
}
}

package DhbOptimizing;

import DhbInterfaces.ManyVariableFunction;

// Vector & function holder used in maximizing many-variable functions.

// @author Didier H. Besset
```

```
public class MaximizingVector extends OptimizingVector {

// Constructor method.
// @param v double[]
// @param f DhbInterfaces.ManyVariableFunction

public MaximizingVector(double[] v,
                                DhbInterfaces.ManyVariableFunction f)
{
    super(v, f);
}

// @return boolean    true if the receiver is "better" than
//                                the supplied point
// @param point OptimizingVector

public boolean betterThan(OptimizingVector point)
{
    return getValue() > point.getValue();
}

// (used by method toString).
// @return java.lang.String

protected final String printedKey()
{
    return " max@";
}
}
```

The abstract class `OptimizingPointFactory` and its two concrete subclasses, `MinimizingPointFactory` and `MaximizingPointFactory`, are in charge of creating the optimizing points. These factory classes[5] play the role of the optimizing STRATEGY described in Section 11.3.1. The concrete classes implement the two methods `createPoint` and `createVector`. The method `createPoint` creates an optimizing point of the desired type for a supplied position (a `double`) and function (a `OneVariableFunction`). The method `createVector` creates an optimizing vector of the desired type for a supplied position (an array of `double`) and function (a `ManyVariableFunction`). A convenience method with the same name takes as first argument a `DhbVector` (see Section 8.1.2). Since the factory is able to create either

5. For people reading code in both languages: Smalltalk does not need any factory—objects of type `Class` are instance factories built into the language. Reflection in Java could have been used here to build an architecture similar to that of Smalltalk. The use of reflection, however, implies the use of casting, which, in my humble opinion, spoils the purpose of strong typing.

points or vectors, it can be reused between optimizers working with one- or many-variable functions.

Listing 11.6 Java optimizing point factory classes

```
package DhbOptimizing;

import DhbInterfaces.OneVariableFunction;
import DhbInterfaces.ManyVariableFunction;
import DhbMatrixAlgebra.DhbVector;

// Factory of point/vector & function holders for optimizing functions.

// @author Didier H. Besset

public abstract class OptimizingPointFactory {

// Constructor method.

public OptimizingPointFactory() {
    super();
}

// @return DhbOptimizing.OptimizingPoint
// @param x double
// @param f OneVariableFunction

public abstract OptimizingPoint createPoint( double x,
                                             OneVariableFunction f);

// @return DhbOptimizing.OptimizingVector
// @param v double[]
// @param f ManyVariableFunction

public abstract OptimizingVector createVector( double[] v,
                                               ManyVariableFunction f);

// @return DhbOptimizing.OptimizingVector
// @param v DhbVector
// @param f ManyVariableFunction

public OptimizingVector createVector( DhbVector v,
                                      DhbInterfaces.ManyVariableFunction f)
{
    return createVector( v.toComponents(), f);
}
```

```
}

package DhbOptimizing;

// Factory of point/vector & function holders for minimizing functions.

// @author Didier H. Besset

public class MinimizingPointFactory extends OptimizingPointFactory
{

// Constructor method.

public MinimizingPointFactory() {
    super();
}

// @return OptimizingPoint    minimizing point strategy.

public OptimizingPoint createPoint(double x,
                           DhbInterfaces.OneVariableFunction f)
{
    return new MinimizingPoint( x, f);
}

// @return OptimizingVector    an minimizing vector strategy.

public OptimizingVector createVector(double[] v,
                           DhbInterfaces.ManyVariableFunction f)
{
    return new MinimizingVector( v, f);
}
}

package DhbOptimizing;

// Factory of point/vector & function holders for maximizing functions.

// @author Didier H. Besset

public class MaximizingPointFactory extends OptimizingPointFactory {

// Constructor method.

public MaximizingPointFactory() {
    super();
}
```

```
    // @return OptimizingPoint    maximizing point strategy.

    public OptimizingPoint createPoint(double x,
                                       DhbInterfaces.OneVariableFunction f)
    {
        return new MaximizingPoint( x, f);
    }

    // @return OptimizingVector    maximizing vector strategy.

    public OptimizingVector createVector(double[] v,
                                       DhbInterfaces.ManyVariableFunction f)
    {
        return new MaximizingVector( v, f);
    }
    }
```

In the Smalltalk implementation, we have been able to bring all optimizing classes under a single abstract class. The Java architecture is similar. In the case of Java, however, the abstract class only handles optimization in *n*-dimensional space.

Listing 11.7 shows the code of class `MultiVariableOptimizer`. This class has the following instance variables:

`f` The goal function; this object must implement the `ManyVariableFunction` interface shown in Listing 11.5.

`pointFactory` The factory used to create optimizing points; this must be one concrete instance of the classes shown in Listing 11.6.

`result` An array of double containing the initial value where to start the algorithm; at the end of the algorithm, this variable contains the position of the optimum if convergence was attained;

The default constructor method provided by the abstract class take three arguments: the goal function, the optimizing point factory, and the initial value. These three arguments correspond to the three instance variables of the class.

The method `setInitialValue` allows changing the initial value in case another search is made with the same instance. The accessor method `getResult` allows one to retrieve the result.

The method `sortPoints` sorts a supplied array of optimizing points according to their STRATEGY. Since sorting is made in situ, a bubble sort algorithm is used.

Listing 11.7 Java abstract class for all optimizing classes

```
package DhbOptimizing;
```

```
import DhbInterfaces.ManyVariableFunction;
import DhbIterations.IterativeProcess;

// Abstract optimizer of many-variable functions.

// @author Didier H. Besset

 public abstract class MultiVariableOptimizer extends IterativeProcess
{

    // Value of the function to optimize.

    protected ManyVariableFunction f;

    // Best value found so far: must be set to determine the
  dimension
    // of the argument of the function.

    protected double[] result;

    // Optimizing strategy (minimum or maximum).

    protected OptimizingPointFactory pointFactory;

// Constructor method.

public MultiVariableOptimizer(ManyVariableFunction func,
        OptimizingPointFactory pointCreator, double[] initialValue)
{
    f = func;
    pointFactory = pointCreator;
    setInitialValue( initialValue);
}

// @return double[]    result of the receiver

public double[] getResult()
{
    return result;
}

// @param v double[]    educated guess for the optimum's location

public void setInitialValue( double[] v)
{
    result = v;
```

```
}

// Use bubble sort to sort the best points

protected void sortPoints( OptimizingVector[] bestPoints)
{
    OptimizingVector temp;
    int n = bestPoints.length;
    int bound = n - 1;
    int i, m;
    while ( bound >= 0 )
    {
        m = -1;
        for ( i = 0; i < bound; i++ )
        {
            if ( bestPoints[i+1].betterThan( bestPoints[i]) )
            {
                temp = bestPoints[i];
                bestPoints[i] = bestPoints[i+1];
                bestPoints[i+1] = temp;
                m = i;
            }
        }
        bound = m;
    }
}
}
```

To find an optimum along a given direction, one needs to construct an object able to transform a vector function into a one-variable function. The class `VectorProjectedFunction` providing this functionality has the following instance variables:

`f` The goal function f ($\mathbf{x}$)

`origin` The vector argument of the goal function, (i.e., the vector $\mathbf{x}_0$)

`direction` The direction along which the function is projected, (i.e., the vector $\mathbf{v}$)

The constructor method takes three arguments corresponding to the three instance variables. In addition, the instance variables `argument` and `direction` can be read and modified using direct accessor methods. For convenience, the set methods exist in two versions of the argument: an array of `double` or a `DhbVector`. This is also true for the constructor method.

The method `argumentAt:` returns the argument vector for the goal function (i.e., the vector $\mathbf{x}$). The method `value:` returns the value of the function g (λ) for the supplied argument λ.

Listing 11.8 Java projected function class

```
package DhbOptimizing;

import DhbInterfaces.OneVariableFunction;
import DhbInterfaces.ManyVariableFunction;
import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.DhbIllegalDimension;

// Projection of a many-variable function
//                      onto a one-dimensional direction.

// @author Didier H. Besset

public class VectorProjectedFunction implements OneVariableFunction
{

    // Value of the function to optimize.

    private ManyVariableFunction f;

    // Origin for function evaluation.

    private DhbVector origin;

    // Direction along which the function is evaluated.

    private DhbVector direction;

// Constructor method.
// @param func ManyVariableFunction    function to project
// @param x double[]    origin of projected function
// @param d double[]    direction of projection
// @exception NegativeArraySizeException if dimension of x or d is 0.

public VectorProjectedFunction( ManyVariableFunction func,
                                          double[] x, double[] d)
                                throws NegativeArraySizeException
{
    f = func;
    setOrigin( x);
    setDirection( d);
}

// Constructor method.
// @param func ManyVariableFunction    function to project
```

```
// @param x DhbVector    origin of projected function
// @param d DhbVector    direction of projection

public VectorProjectedFunction( ManyVariableFunction func,
                                                DhbVector x, DhbVector d)
{
    f = func;
    setOrigin( x);
    setDirection( d);
}

// @param x double[]    origin of projected function
// @exception DhbIllegalDimension
//                if dimension of x is not that of the origin.

public DhbVector argumentAt(double x) throws DhbIllegalDimension
{
    DhbVector v = direction.product( x);
    v.accumulate( origin);
    return v;
}

// @return DhbMatrixAlgebra.DhbVector    direction of the receiver

public DhbVector getDirection()
{
    return direction;
}

// @return DhbMatrixAlgebra.DhbVector    origin of the receiver

public DhbVector getOrigin()
{
    return origin;
}

// @param v DhbMatrixAlgebra.DhbVector
// @exception NegativeArraySizeException if dimension of v is 0.

public void setDirection( double[] v) throws NegativeArraySizeException
{
    direction = new DhbVector( v);
}
```

```
// @param v DhbMatrixAlgebra.DhbVector

public void setDirection( DhbVector v)
{
    direction = v;
}

// @param v DhbMatrixAlgebra.DhbVector
// @exception NegativeArraySizeException if dimension of v is 0.

public void setOrigin( double[] v) throws NegativeArraySizeException
{
    origin = new DhbVector( v);
}

// @param v DhbMatrixAlgebra.DhbVector

public void setOrigin( DhbVector v)
{
    origin = v;
}

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

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( origin);
    sb.append(" -> ");
    sb.append( direction);
    return sb.toString();
}

// @return double    value of the function
// @param x double    distance from the origin in unit of direction.

public double value(double x)
{
    try{ return f.value( argumentAt(x).toComponents());
    } catch ( DhbIllegalDimension e) { return Double.NaN;}
}
}
```

11.4 Optimizing in One Dimension

To find the optimum of a one-variable function, $g(\lambda)$, whose derivative is unknown, the most robust algorithm is an algorithm similar to the bisection algorithm described in Section 5.2.

Let us assume that we have found three points—λ_0, λ_1, and λ_2—such that $\lambda_0 < \lambda_1 < \lambda_2$ and such that $g(\lambda_1)$ is better than both $g(\lambda_0)$ and $g(\lambda_2)$. If the function g is continuous over the interval $[\lambda_0, \lambda_2]$, then we are certain that an optimum of the function is located in the interval $[\lambda_0, \lambda_2]$. As for the bisection algorithm, we shall try to find a new triplet of values with similar properties while reducing the size of the interval. A point is picked in the largest of the two intervals $[\lambda_0, \lambda_1]$ or $[\lambda_1, \lambda_2]$ and is used to reduce the initial interval.

If $\lambda_1 - \lambda_0 \leq \lambda_2 - \lambda_1$, we compute $\lambda_4 = \lambda_1 + \omega(\lambda_2 - \lambda_1)$, where ω is the golden section (i.e., $\omega = \frac{3-\sqrt{5}}{2} \approx 0.38197$) from Pythagorean lore. Choosing ω instead of $1/2$ ensures that successive intervals have the same relative scale. A complete derivation of this argument can be found in [Press *et al.*]. If λ_4 yields a better function value than λ_1, the new triplet of points becomes λ_1, λ_4, and λ_2; otherwise, the triplet becomes λ_0, λ_1, and λ_4.

If we have $\lambda_1 - \lambda_0 > \lambda_2 - \lambda_1$, we compute $\lambda_4 = \lambda_1 + \omega(\lambda_0 - \lambda_1)$. Then the new triplets can be either λ_0, λ_4, and λ_1, or λ_4, λ_1, and λ_2.

The reader can verify that the interval decreases steadily, although not as fast as in the case of bisection where the interval is halved at each iteration. Since the algorithm is using the golden section, it is called the *golden section search*.

By construction, the golden section search algorithm makes sure that the optimum is always located between the points λ_0 and λ_2. Thus, at each iteration, the quantity $\lambda_2 - \lambda_0$ gives an estimate of the error on the position of the optimum.

11.4.1 Optimizing in One Dimension—Smalltalk Implementation

Listing 11.9 shows the class `DhbOneVariableFunctionOptimizer` implementing the search for an optimum of a one-variable function using the golden section search. Code Example 11.1 shows how to use this class to find the maximum of the gamma distribution discussed in Section 9.7.

Code Example 11.1

```
| distr finder maximum |
distr := DhbGammaDistribution shape: 2 scale: 5.
finder := DhbOneVariableFunctionOptimizer maximizingFunction: distr.
maximum := finder evaluate.
```

The first line after the declarations creates a new instance of a gamma distribution with parameters $\alpha = 2$ and $\beta = 5$. The next line creates an instance of the optimum finder. The selector used to create the instance selects a search for a maximum. The last line is the familiar statement to evaluate the iterations—that is, performing

the search for the maximum—and to retrieve the result. Since no initial value was supplied, the search begins at a random location.

The class `DhbOneVariableFunctionOptimizer` is a subclass of the class `FunctionOptimizer`. It does not need any additional instance variables. The golden section is kept as a class variable and is calculated at the first time it is needed.

At each iteration, the method `nextXValue` is used to compute the next position at which the function is evaluated. This corresponding new optimizing point is added to the collection of best points. Then, the method `indexOfOuterPoint` is used to determine which point must be discarded; it is always the second point on either side of the best point. The precision of the result is estimated from the bracketing interval in the method `computePrecision`, using relative precision (of course!).

The method `addPoint:` of the superclass can be used to supply an initial bracketing interval. The method `computeInitialValues` first checks whether a valid bracketing interval has been supplied into the collection of best points. If this is not the case, a search for a bracketing interval is conducted using the class `DhbOptimizingBracketFinder` described in Section 11.5.1. The instance of the bracket finder is created with the method `forOptimizer:` so that its strategy and goal function are taken over from the golden section optimum finder.

Listing 11.9 Smalltalk golden section optimum finder

Class	`DhbOneVariableFunctionOptimizer`
Subclass of	`DhbFunctionOptimizer`
Class variable names:	`GoldenSection`

Class Methods

defaultPrecision

```
^DhbFloatingPointMachine new defaultNumericalPrecision * 10
```

goldenSection

```
GoldenSection isNil ifTrue: [GoldenSection := (3 - 5 sqrt) / 2].
^GoldenSection
```

Instance Methods

computeInitialValues

```
[ bestPoints size  > 3] whileTrue: [ bestPoints removeLast].
bestPoints size = 3
    ifTrue: [ self hasBracketingPoints
                ifFalse:[ bestPoints removeLast].
            ].
```

```
bestPoints size < 3
    ifTrue: [ ( DhbOptimizingBracketFinder forOptimizer: self)
                                                        evaluate].
```

computePrecision

```
^self precisionOf: ( ( bestPoints at: 2) position - ( bestPoints
                                                at: 3) position) abs
       relativeTo: ( bestPoints at: 1) position abs
```

evaluateIteration

```
self addPointAt: self nextXValue.
bestPoints removeAtIndex: self indexOfOuterPoint.
^self computePrecision
```

hasBracketingPoints

```
| x1 |
x1 := ( bestPoints at: 1) position.
^( ( bestPoints at: 2) position - x1) * (( bestPoints at: 3)
                                            position - x1) < 0
```

indexOfOuterPoint

```
| inferior superior x |
inferior := false.
superior := false.
x := bestPoints first position.
2 to: 4 do:
    [ :n |
      ( bestPoints at: n) position < x
            ifTrue: [ inferior
                        ifTrue: [ ^n].
                      inferior := true.
                    ]
            ifFalse:[ superior
                        ifTrue: [ ^n].
                      superior := true.
                    ].
    ].
```

nextXValue

```
| d3 d2 x1 |
x1 := ( bestPoints at: 1) position.
d2 := ( bestPoints at: 2) position - x1.
d3 := ( bestPoints at: 3) position - x1.
^( d3 abs > d2 abs ifTrue: [ d3]
```

```
                    ifFalse:[ d2]) * self class goldenSection + x1
```

reset

```
    [ bestPoints isEmpty] whileFalse: [ bestPoints removeLast].
```

11.4.2 Optimizing in One Dimension—Java Implementation

Listing 11.10 shows the class `OneVariableFunctionOptimizer` implementing the search for an optimum of a one-variable function using the golden section search in Java. Code Example 11.2 shows how to use this class to find the maximum of the gamma distribution discussed in Section 9.7.

Code Example 11.2

```
CauchyDistribution distr = new CauchyDistribution( 10, 5);
MaximizingPointFactory strategy = new MaximizingPointFactory();
OneVariableFunctionOptimizer finder =
            new OneVariableFunctionOptimizer( distr, strategy);
finder.setDesiredPrecision( 1.0e-5);
finder.evaluate();
double result = finder.getResult();
```

The first line creates an instance of a gamma distribution with parameters $\alpha = 2$ and $\beta = 5$. The next line creates an instance of a maximizing point factory. This factory is used as the strategy for the optimum finder created on the next line, the first argument of the constructor method being the function to maximize. A medium precision is given explicitly since the default provided by the class `IterativeProcess` is likely to be meaningless (see [Press *et al.*]). Since no initial value is supplied, the search begins at a random point. After a call to the method `evaluate`—common to all iterative processes—the result is retrieved on the last line.
The class `OneVariableFunctionOptimizer` is a subclass of the class `FunctionalIterator` described in Section 4.2.2. It has the following instance variables

`bestPoints` An array of three optimizing points corresponding to the values λ_0, λ_1, and λ_2 in this order; thus, the best is always the middle one.

`pointFactory` The factory used to create optimizing points (see Section 11.3.3)

The private static variable `goldenSection` holds the value of the golden section.

The method `evaluateIteration` determines which side of the bracketing interval the bisecting step will be performed. The bisection step of the algorithm is performed within the method `reducePoints`. The arguments of the method are either 0 or 2 depending on the selected side. The precision returned by the method `evaluateIteration` is of course calculated using relative precision.

The method `initializeIterations` uses the optimum bracket finder described in Section 11.5.2 to obtain the first bracketing interval. The search for the interval

is conducted from the supplied initial value. The instance of the bracket finder is created by supplying the same goal function and same strategy as the initial golden section finder.

Listing 11.10 Java implementation of the golden section optimum search

```
package DhbOptimizing;

import DhbInterfaces.OneVariableFunction;
import DhbIterations.FunctionalIterator;

// Optimizer of one-variable functions
// (uses golden search algorithm).

// @author Didier H. Besset

public class OneVariableFunctionOptimizer extends FunctionalIterator
{
    private static double goldenSection = ( 3 - Math.sqrt(5)) / 2;

    // Best points found so far.

    private OptimizingPoint[] bestPoints = null;

    // Optimizing strategy (minimum or maximum).

    private OptimizingPointFactory pointFactory;

// Constructor method
// @param func OneVariableFunction
// @param pointCreator OptimizingPointFactory    a factory to create
//                                    strategy points

public OneVariableFunctionOptimizer(OneVariableFunction func,
                                    OptimizingPointFactory pointCreator)
{
    super(func);
    pointFactory = pointCreator;
}

// @return double    the relative precision on the result

private double computePrecision()
{
    return relativePrecision( Math.abs( bestPoints[2].getPosition()
```

```
                                        - bestPoints[1].getPosition()),
                                Math.abs( bestPoints[0].getPosition()));
    }

    // @return double    current precision of result

    public double evaluateIteration()
    {
        if ( bestPoints[2].getPosition() - bestPoints[1].getPosition()
                > bestPoints[1].getPosition() - bestPoints[0].getPosition() )
            reducePoints(2);
        else
            reducePoints(0);
        result = bestPoints[1].getPosition();
        return computePrecision();
    }
    public void initializeIterations()
    {
        OptimizingBracketFinder bracketFinder =
        new OptimizingBracketFinder( f, pointFactory);
        bracketFinder.setInitialValue( result);
        bracketFinder.evaluate();
        bestPoints = bracketFinder.getBestPoints();
    }

    // Apply bisection on points 1 and n
    // @param n int    index of worst point of bisected interval

    private void reducePoints( int n)
    {
        double x = bestPoints[1].getPosition();
        x += goldenSection * ( bestPoints[n].getPosition() - x);
        OptimizingPoint newPoint = pointFactory.createPoint( x, f);
        if ( newPoint.betterThan( bestPoints[1]) )
        {
            bestPoints[2-n] = bestPoints[1];
            bestPoints[1] = newPoint;
        }
        else
            bestPoints[n] = newPoint;
    }

    // @return java.lang.String

    public String toString()
    {
```

```
        StringBuffer sb = new StringBuffer();
        sb.append( getIterations());
        sb.append( " iterations, precision = ");
        sb.append( getPrecision());
        for ( int i = 0 ; i < bestPoints.length; i++ )
        {
            sb.append( '\n');
            sb.append( bestPoints[i]);
        }
        return sb.toString();
    }
    }
```

11.5 Bracketing the Optimum in One Dimension

As we have seen in Section 11.4, the golden section algorithm requires the knowledge of a bracketing interval. This section describes a very simple algorithm to obtain a bracketing interval with certainty if the function is continuous and does indeed have an optimum of the sought type.

The algorithm goes as follows. Take two points λ_0 and λ_1. If they do not exist, pick up some random values (random generators are described in Section 9.4). Let us assume that $g(\lambda_1)$ is better than $g(\lambda_0)$.

1. Let $\lambda_2 = 3\lambda_1 - 2\lambda_0$; that is, λ_2 is twice as far from λ_1 than λ_0 and is located on the other side, toward the optimum.
2. If $g(\lambda_1)$ is better than $g(\lambda_2)$, we have a bracketing interval; the algorithm is stopped.
3. Otherwise, set $\lambda_0 = \lambda_1$ and $\lambda_1 = \lambda_2$ and go back to step 1.

The reader can see that the interval $[\lambda_0, \lambda_1]$ is increasing at each step. Thus, if the function has no optimum of the sought type, the algorithm will cause a floating overflow exception quite rapidly.
The implementation in each language has so little in common that the common section is therefore omitted.

11.5.1 Bracketing the Optimum—Smalltalk Implementation

Listing 11.11 shows the Smalltalk code of the class implementing the search algorithm for an optimizing bracket. The class `DhbOptimizingBracketFinder` is a subclass of class `DhbOneVariableFunctionOptimizer` from Section 11.9. This was a convenient, but not necessary, choice to be able to reuse most of the management and accessor methods. The methods pertaining to the algorithm are, of course, quite different.

Example of use of the optimizing bracket finder can be found in method `computeInitialValues` of class `DhbOneVariableFunctionOptimizer` (see Listing 11.9).

The method `setInitialPoints:` allows usint the collection of best points of another optimizer inside the class. This breach to the rule of hiding the implementation can be tolerated here because the class `DhbOptimizingBracketFinder` is used exclusively with the class `DhbOneVariableFunctionOptimizer`. It allows the two classes to use the same sorted collection of optimizing points. If no initial point has been supplied, it is obtained from a random generator.

The *precision* calculated in the method `evaluateIteration` is a large number, which becomes negative as soon as the condition to terminate the algorithm is met. Having a negative precision causes an iterative process as defined in chapter 4 to stop.

Listing 11.11 Smalltalk optimum bracket finder

Class	`DhbOptimizingBracketFinder`
Subclass of	`DhbOneVariableFunctionOptimizer`

Class Methods

initialPoints: `aSortedCollection` **function:** `aFunction`

```
^super new setInitialPoints: aSortedCollection; setFunction:
                                                        aFunction
```

Instance Methods

computeInitialValues

```
[bestPoints size < 2] whileTrue: [self addPointAt: Number random]
```

evaluateIteration

```
| x1 x2 |
x1 := ( bestPoints at: 1) position.
x2 := ( bestPoints at: 2) position.
self addPointAt: ( x1 * 3 - ( x2 * 2)).
precision := ( x2 - x1) * ( ( bestPoints at: 3) position - x1).
self hasConverged
    ifFalse:[ bestPoints removeLast].
^precision
```

finalizeIterations

```
result := bestPoints.
```

initializeForOptimizer: `aFunctionOptimizer`

```
super initializeForOptimizer: aFunctionOptimizer.
bestPoints := aFunctionOptimizer bestPoints.
^self
```

setInitialPoints: `aSortedCollection`

```
bestPoints := aSortedCollection.
```

11.5.2 Bracketing the Optimum—Java Implementation

Listing 11.12 shows the Smalltalk code of the class implementing the search algorithm for an optimizing bracket. The class `OptimizingBracketFinder` is a subclass of the class `FunctionalIterator` defined in Section 4.2.2. It has the same instance variables as the class `OneVariableFunctionOptimizer`.

The method `initializeIterations` gets an initial value using a random generator if no initial value has been supplied. The other points of the initial interval are obtained from a random generator. At the end of the method, the array of best points contains three optimizing points sorted in ascending order of their position.

The method `evaluateIteration` first tests whether the optimum seems to be located toward negative or positive values. After the test, the interval is expanded toward the corresponding direction by the method `moveTowardNegative` or `moveTowardPositive`.

The method `evaluateIteration` calculate a pseudoprecision: it is a large number, which becomes negative as soon as the condition to terminate the algorithm is met. Having a negative precision causes an iterative process as defined in Chapter 4 to stop.

The object using the bracket finder can retrieve the array of best points using the method `getBestPoints`. This method constitutes a breach to the rule of hiding the implementation. In this case, however, this is an acceptable breach: the class `OneVariableFunctionOptimizer` can retrieve the initial bracketing interval and use it immediately without having to evaluate again the function at these positions. After all, the bracketing interval *is* the result of the optimizing bracket search.

Listing 11.12 Java optimum bracket finder

```
package DhbOptimizing;

import java.util.Random;
import DhbInterfaces.OneVariableFunction;
import DhbIterations.FunctionalIterator;

// Finds a bracket for the optimum of a one-variable function.
```

```
// @author Didier H. Besset

public class OptimizingBracketFinder extends FunctionalIterator
{
    private OptimizingPoint[] bestPoints = null;
    private OptimizingPointFactory pointFactory;

// Constructor method
// @param func OneVariableFunction
// @param pointCreator OptimizingPointFactory    a factory to create
//                                  strategy points

public OptimizingBracketFinder(OneVariableFunction func,
                                 OptimizingPointFactory pointCreator)
{
    super(func);
    pointFactory = pointCreator;
}

// @return double    1 as long as no bracket has been found

private double computePrecision()
{
    return bestPoints[1].betterThan( bestPoints[0]) &&
                            bestPoints[1].betterThan( bestPoints[2])
                ? 0 : 1;
}

// @return double    pseudo-precision of the current search

public double evaluateIteration()
{
    if ( bestPoints[0].betterThan( bestPoints[1]) )
        moveTowardNegative();
    else if ( bestPoints[2].betterThan( bestPoints[1]) )
        moveTowardPositive();
    return computePrecision();
}

// @return OptimizingPoint[]    a triplet bracketing the optimum

public OptimizingPoint[] getBestPoints()
{
    return bestPoints;
}
```

```
    // Use random locations

    public void initializeIterations()
    {
        Random generator = new Random();
        bestPoints = new OptimizingPoint[3];
        if ( Double.isNaN( result) )
            result = generator.nextDouble();
        bestPoints[0] = pointFactory.createPoint( result, f);
        bestPoints[1] = pointFactory.createPoint( generator.nextDouble()
                                        + bestPoints[0].getPosition(), f);
        bestPoints[2] = pointFactory.createPoint( generator.nextDouble()
                                        + bestPoints[1].getPosition(), f);
    }

    // Shift the best points toward negative positions.

    private void moveTowardNegative()
    {
        OptimizingPoint newPoint = pointFactory.createPoint(
                               3 * bestPoints[0].getPosition()
                               - 2 * bestPoints[1].getPosition(), f);
        bestPoints[2] = bestPoints[1];
        bestPoints[1] = bestPoints[0];
        bestPoints[0] = newPoint;
    }

    // Shift the best points toward positive positions.

    private void moveTowardPositive()
    {
        OptimizingPoint newPoint = pointFactory.createPoint(
                                    3 * bestPoints[2].getPosition()
                                    - 2 * bestPoints[1].getPosition(), f);
        bestPoints[0] = bestPoints[1];
        bestPoints[1] = bestPoints[2];
        bestPoints[2] = newPoint;
    }
    }
```

11.6 Powell's Algorithm

Powell's algorithm is one of many hill climbing algorithms [Press *et al.*]. The idea underlying Powell's algorithm is that once an optimum has been found in one direction, the chance for the biggest improvement lies in the direction perpendicular to that direction. A mathematical formulation of this sentence can be found in [Press

et al.] and references therein. Powell's algorithm provides a way to keep track of the next best direction at each iteration step.
The original steps of Powell's algorithm are as follows:

1. Let $\mathbf{x}_0$ be the best point so far, and initialize a series of vectors $\mathbf{v}_1, \ldots, \mathbf{v}_n$ forming the system of reference (n is the dimension of the vector $\mathbf{x}_0$); in other words, the components of the vector $\mathbf{v}_k$ are all zero except for the kth components, which is 1.
2. Set $k = 1$.
3. Find the optimum of the goal function along the direction $\mathbf{v}_k$ starting from point $\mathbf{x}_{k-1}$. Let $\mathbf{x}_k$ be the position of that optimum.
4. Set $k = k + 1$. If $k \leq n$, go back to step 3.
5. For $k = 1, \ldots, n-1$, set $\mathbf{v}_k = \mathbf{v}_{k-1}$.
6. Set $\mathbf{v}_n = \frac{\mathbf{x}_n - \mathbf{x}_0}{|\mathbf{x}_n - \mathbf{x}_0|}$.
7. Find the optimum of the goal function along the direction $\mathbf{v}_n$. Let $\mathbf{x}_{n+1}$ be the position of that optimum.
8. If $|\mathbf{x}_n - \mathbf{x}_0|$ is less than the desired precision, terminate.
9. Otherwise, set $\mathbf{x}_0 = \mathbf{x}_{n+1}$ and go back to step 1.

There are actually two hill climbing algorithms within each other. The progression obtained by the inner loop is taken as the direction in which to continue the search.

Powell recommends using this algorithm on goal functions having a quadratic behavior near the optimum. It is clear that this algorithm cannot be used safely on any function. If the goal function has narrow valleys, all directions $\mathbf{v}_1, \ldots, \mathbf{v}_n$ will become colinear when the algorithm ends up in such a valley. Thus, the search is likely to end up in a position where no optimum is located. Press et al. [Press *et al.*] mention two methods avoiding such problems: one method is quite complex, and the other slows down the convergence of the algorithm.

In spite of this *caveat*, I implement Powell's algorithm in its original form. However, I recommend its use only in the vicinity of the minimum. In Section 11.9 I show how other techniques can be used to reach the vicinity of the optimum, where Powell's algorithm can be safely used to make the final determination of the optimum's position.

11.6.1 Powell's Algorithm—General Implementation

Since the class implementing the vector-projected function $g(\lambda)$ described in sections 11.3.2 and 11.3.3 keep the vector $\mathbf{x}_0$ and $\mathbf{v}$ in instance variables, there is no need to allocate explicit storage for the vectors $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{v}_1, \ldots, \mathbf{v}_n$. Instead, the class implementing Powell's algorithm keep an array of vector projected functions with the corresponding parameters. Then, the manipulation of the vector $\mathbf{x}_1, \ldots, \mathbf{x}_n$ and $\mathbf{v}_1, \ldots, \mathbf{v}_n$ is made directly on the projected function.

Since the origin of the projected function is always the starting value, $\mathbf{x}_k$, the initial value for the search of the optimum of the function g (λ) is always zero.

The method `initializeIterations` allocates a series of vector projected functions starting with the axes of the reference system. This method also creates an instance of a one-dimensional optimum finder kept in the instance variable, `unidimensionalFinder`. The goal function of the finder is alternatively assigned to each of the projected functions.

We made a slight modification to Powell's algorithm. If the norm of the vector $\mathbf{x}_n - \mathbf{x}_0$ at step 6 is smaller than the desired precision, the directions are only rotated, the assignment of step 6 is not done, and the search of step 7 is omitted.

The precision computed at the end of each iteration is the maximum of the relative change on all components between the vectors $\mathbf{x}_n$ and $\mathbf{x}_0$.

11.6.2 Powell's Algorithm—Smalltalk Implementation

Listing 11.13 shows the implementation of Powell's algorithm in Smalltalk. Code Example 11.3 shows how to find the maximum of a vector function.

Code Example 11.3

```
| fBlock educatedGuess hillClimber result |
fBlock := ¡the goal function¿
educatedGuess := ¡a vector not too far from the optimum¿

hillClimber := DhbHillClimbingOptimizer maximizingFunction: fBlock.
hillClimber initialValue: educatedGuess.
result := hillClimber evaluate.
```

The class `DhbHillClimbingOptimizer` is a subclass of class `DhbFunctionOptimizer`. It has only one additional instance variable, `unidimensionalFinder`, to hold the one-dimensional optimizer used to find an optimum of the goal function along a given direction.

The method `evaluateIteration` uses the method `inject:into:` to perform steps 2–4 of the algorithm. Similarly, step 5 of the algorithm is performed with a method `inject:into:` within the method `shiftDirection`. This mode of using the iterator method `inject:into:`—performing an action involving two consecutive elements of an indexed collection—is somewhat unusual but highly convenient [Beck]. The method `minimizeDirection:` implements step 3 of the algorithm.

Listing 11.13 Smalltalk implementation of Powell's algorithm

Class	`DhbHillClimbingOptimizer`
Subclass of	`DhbFunctionOptimizer`
Instance variable names:	`unidimensionalFinder`

Instance Methods

computeInitialValues

```
unidimensionalFinder := DhbOneVariableFunctionOptimizer
                                                forOptimizer: self.
unidimensionalFinder desiredPrecision: desiredPrecision.
bestPoints := ( 1 to: result size)
            collect: [ :n | ( DhbVectorProjectedFunction
                                function: functionBlock)
                               direction: ( ( DhbVector
                                               new: result size)
                                                   atAllPut: 0;
                                                   at: n put: 1;
                                                   yourself);
                                          yourself
                    ].
```

evaluateIteration

```
| oldResult |
precision := 1.
bestPoints inject: result
            into: [ :prev :each | ( self minimizeDirection: each
                                                   from: prev)].
self shiftDirections.
self minimizeDirection: bestPoints last.
oldResult := result.
result := bestPoints last origin.
precision := 0.
result with: oldResult do:
    [ :x0 :x1 |
      precision := ( self precisionOf: (x0 - x1) abs relativeTo:
                                          x0 abs) max: precision.
    ].
^precision
```

finalizeIterations

minimizeDirection: aVectorFunction

```
^unidimensionalFinder
    reset;
    setFunction: aVectorFunction;
    addPointAt: 0;
    addPointAt: precision;
    addPointAt: precision negated;
    evaluate
```

minimizeDirection: `aVectorFunction` **from:** `aVector`

```
Function from: aVector
    ^aVectorFunction
        origin: aVector;
        argumentWith: ( self minimizeDirection: aVectorFunction)
```

shiftDirections

```
| position delta firstDirection |
firstDirection := bestPoints first direction.
bestPoints inject: nil
                into: [ :prev :each |
                        position isNil
                            ifTrue: [ position := each origin]
                            ifFalse:[ prev direction: each
                                                    direction].
                        each
                        ].
position := bestPoints last origin - position.
delta := position norm.
delta > desiredPrecision
    ifTrue: [ bestPoints last direction: (position scaleBy: (1 /
                                                        delta))]
    ifFalse:[ bestPoints last direction: firstDirection].
```

11.6.3 Powell's Algorithm—Java Implementation

Listing 11.14 shows the implementation of Powell's algorithm in Java. Code Example 11.4 shows how to use this class to find the maximum of a vector function.

Code Example 11.4

```
ManyVariableFunction func = ¡the goal function¿
double[] educatedGuess = ¡an array of double not too far from the
optimum¿

    MaximizingPointFactory strategy = new MaximizingPointFactory();
    HillClimbingOptimizer hillClimber =
                new HillClimbingOptimizer( func, strategy);
    hillClimber.setInitialValue( educatedGuess);
    hillClimber.evaluate();
    double[] result = hillClimber.getResult();
```

The class `HillClimbingOptimizer` is a subclass of class `MultiVariableOptimizer`. It has the following instance variables:

`unidimensionalFinder` An instance of the class `OneVariableFunctionOptimizer`; this instance is used to find the optimum of the function in steps 3 and 7 of the algorithm.

`projections` An array of projected functions

Listing 11.14 Java implementation of Powell's algorithm

```
package DhbOptimizing;

import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbInterfaces.ManyVariableFunction;

// Hill climbing optimizer using Powell's algorithm.

// @author Didier H. Besset

public class HillClimbingOptimizer extends MultiVariableOptimizer
{

    // One dimensional optimizer used in each direction.

    private OneVariableFunctionOptimizer unidimensionalFinder;

    // Projected goal function on independent directions.

    private VectorProjectedFunction[] projections;

// Constructor method.
// @param func DhbInterfaces.ManyVariableFunction
// @param pointCreator DhbOptimizing.OptimizingPointFactory

public HillClimbingOptimizer( ManyVariableFunction func,
                    OptimizingPointFactory pointCreator, double[] v)
{
    super(func, pointCreator, v);
}
private void adjustLastDirection( DhbVector start)
{
    try {
        int n = projections.length - 1;
        projections[n].setOrigin( result);
        DhbVector newDirection = projections[n].getOrigin()
                                                .subtract( start);
        double norm = newDirection.norm();
```

```
            if ( norm > getDesiredPrecision() )
            {
                newDirection.scaledBy( 1 / norm);
                projections[n].setDirection( newDirection);
                unidimensionalFinder.setFunction( projections[n]);
                unidimensionalFinder.setInitialValue( 0);
                unidimensionalFinder.evaluate();
                result = projections[n].argumentAt(
                        unidimensionalFinder.getResult()).toComponents();
            }
        }  catch ( DhbIllegalDimension e){};
    }

    // @return double    relative precision of current result
    // @param x double[]    result at previous iteration

    private double computePrecision( double[] x)
    {
        double eps = 0;
        for ( int i = 0; i < result.length; i++)
            eps = Math.max( eps, relativePrecision(
                                Math.abs( result[i] - x[i]), result[i]));
        return eps;
    }
    public double evaluateIteration()
    {
        try {
            DhbVector start;
            start = new DhbVector( result);
            int n = projections.length;
            for( int i = 0; i < n; i++ )
            {
                projections[i].setOrigin( result);
                unidimensionalFinder.setFunction( projections[i]);
                unidimensionalFinder.setInitialValue( 0);
                unidimensionalFinder.evaluate();
                result = projections[i].argumentAt(
                        unidimensionalFinder.getResult()).toComponents();
            }
            rotateDirections();
            adjustLastDirection( start);
            return computePrecision( start.toComponents());
        } catch (NegativeArraySizeException e){ return Double.NaN;}
        catch ( DhbIllegalDimension e){ return Double.NaN;};
    }

    public void initializeIterations()
```

```
{
    projections = new VectorProjectedFunction[ result.length];
    double [] v = new double[ result.length];
    for ( int i = 0; i < projections.length; i++ )
        v[i] = 0;
    for ( int i = 0; i < projections.length; i++ )
        {
            v[i] = 1;
            projections[i] = new VectorProjectedFunction( f, result, v);
            v[i] = 0;
        }
    unidimensionalFinder = new OneVariableFunctionOptimizer(
                                        projections[0], pointFactory);
    unidimensionalFinder.setDesiredPrecision( getDesiredPrecision());
}
private void rotateDirections()
{
    DhbVector firstDirection = projections[0].getDirection();
    int n = projections.length;
    for ( int i = 1; i < n; i++ )
        projections[i-1].setDirection(projections[i].getDirection());
    projections[n-1].setDirection( firstDirection);
}

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

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( getIterations());
    sb.append( " iterations, precision = ");
    sb.append( getPrecision());
    sb.append( "\nResult:");
    for ( int i = 0; i < result.length; i++ )
    {
        sb.append(' ');
        sb.append( result[i]);
    }
    for ( int i = 0 ; i < projections.length; i++ )
    {
        sb.append( '\n');
        sb.append( projections[i]);
    }
    return sb.toString();
}
}
```

11.7 Simplex Algorithm

The simplex algorithm, invented by Nelder and Mead, provides an efficient way to find a good approximation of the optimum of a function starting from any place [Press *et al.*]. The only trap into which the simplex algortihm can run into is a local optimum. On the other hand, this algorithm does not converge very well in the vicinity of the optimum. Thus, it must not be used with the desired precision set to a very low value. Once the optimum has been found with the simplex algorithm, other more precise algorithms can then be used, such as the ones describes in Sections 11.1 and 11.6. MINUIT uses a combination of simplex and Newton algorithms. My implementation of general purpose optimizer uses a combination of simplex and Powell algorithms.

A simplex in an n-dimensional space is a figure formed with $n+1$ summits. For example, a simplex in a two-dimensional space is a triangle; a simplex in a three-dimensional space is a tetrahedron. Let us now discuss the algorithm for finding the optimum of a function with a simplex.

1. Pick up $n+1$ points in the search space, and evaluate the goal function at each of them. Let **A** be the summit yielding the worst function's value.
2. If the size of the simplex is smaller than the desired precision, terminate the algorithm.
3. Calculate **G**, the center of gravity of the n best points—that is, all points except **A**.
4. Calculate the location of the symmetrical point of **A** relative to **G**: $\mathbf{A}' = 2\mathbf{G} - \mathbf{A}$.
5. If $f(\mathbf{A}')$ is not the best value found so far, go to step 9.
6. Calculate the point $\mathbf{A}'' = 2\mathbf{A}' - \mathbf{G}$—that is, a point twice as far from **G** as $\mathbf{A}'$.
7. If $f(\mathbf{A}'')$ is a better value than $f(\mathbf{A}')$, build a new simplex with the point **A** replaced by the point $\mathbf{A}''$, and go to step 2.
8. Otherwise, build a new simplex with the point **A** replaced by the point $\mathbf{A}'$, and go to step 2.
9. Calculate the point $\mathbf{B} = \frac{(\mathrm{G}+\mathrm{A})}{2}$.
10. If $f(\mathbf{B})$ yields the best value found so far, build a new simplex with the point **A** replaced by the point $\mathbf{A}''$ and go to step 2.
11. Otherwise, build a new simplex obtained by dividing all edges leading to the point yielding the best value by 2, and go back to step 2.

Figure 11.4 shows the meaning of the operations involved in the algorithm in the three-dimensional case. Step 6 makes the simplex grow in the direction where the function is the best so far. Thus, the simplex becomes elongated in the expected direction of the optimum. Because of its geometrical shape, the next step is necessarily

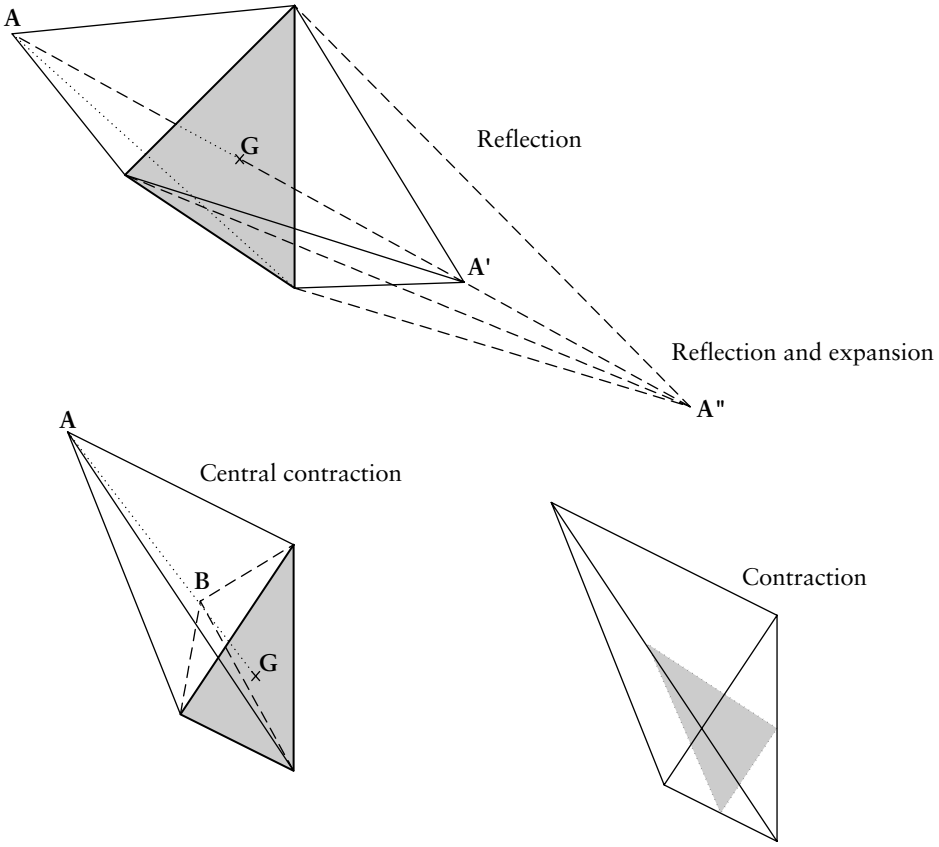

FIG. 11.4 Operations of the simplex algorithm

taken along another direction, causing an exploration of the regions surrounding the growth obtained at the preceding step. Over the iterations, the shape of the simplex adapts itself to narrow valleys where the hill climbing algorithms notoriously get into trouble. Steps 9 and 11 ensure the convergence of the algorithm when the optimum lies inside the simplex. In this mode, the simplex works very much like the golden section search or the bisection algorithms.

Finding the initial points can be done in several ways. If a good approximation of the region where the maximum might be located can be obtained, one uses that approximation as a start and generates n other points by finding the optimum of the function along each axis. Otherwise, one can generate random points and select $n + 1$ points yielding the best values to build the initial simplex. In all cases, one must make sure that the initial simplex has a nonvanishing size in all dimensions of the space. Otherwise, the algorithm will not reach the optimum.

11.7.1 Simplex Algorithm—General Implementation

The class implementing the simplex algorithm belong to the hierarchy of the iterative processes discussed in Chapter 4. The method `evaluateIteration` directly implements the steps of the algorithm as described earlier. The points **G**, **A**′, **A**″, and **B** are calculated using the vector operations described in Section 8.1.

The routine `initializeIterations` assumes that an initial value has been provided. It then finds the location of an optimum of the goal function along each axis of the reference system starting each time from the supplied initial value, unlike hill climbing algorithms. Restarting from the initial value is necessary to avoid creating a simplex with a zero volume. Such mishaps can arise when the initial value is located on an axis of symmetry of the goal function. This can happen quite frequently with educated guesses.

11.7.2 Simplex Algorithm—Smalltalk Implementation

Listing 11.15 shows the Smalltalk implementation of the simplex algorithm. Code Example 11.5 shows how to invoke the class to find the minimum of a vector function.

Code Example 11.5

```
| fBlock educatedGuess simplex result |
fBlock := <the goal function>
educatedGuess := <a vector in the search space>

 simplex := DhbSimplexOptimizer minimizingFunction: fBlock.
 simplex initialValue: educatedGuess.
 result := simplex evaluate.
```

Except for the line creating the instance of the simplex optimizer, this code example is identical to the example of Powell's hill climbing algorithm (Code Example 11.3).

The class `DhbSimplexOptimizer` is a subclass of class `DhbFunctionOptimizer`. To be able to use the iterator methods efficiently, the worst point of the simplex, **A**, is held in a separate instance variable `worstPoint`. As we do not need to know the function's value $f(\mathbf{A})$, it is kept as a vector. The remaining points of the simplex are kept in the instance variable `bestPoints` of the superclass. Since this collection is sorted automatically when points are inserted into it, there is no explicit sorting step.

Listing 11.15 Smalltalk implementation of simplex algorithm

Class	`DhbSimplexOptimizer`
Subclass of	`DhbFunctionOptimizer`
Instance variable names:	`worstVector`

Class Methods

defaultPrecision

```
^DhbFloatingPointMachine new defaultNumericalPrecision * 1000
```

Instance Methods

buildInitialSimplex

```
| projectedFunction finder partialResult |
projectedFunction := DhbProjectedOneVariableFunction
            function: functionBlock.
finder := DhbOneVariableFunctionOptimizer forOptimizer: self.
finder setFunction: projectedFunction.
[bestPoints size < (result size + 1)] whileTrue:
        [projectedFunction
            setArgument: result;
            bumpIndex.
        partialResult := finder
                    reset;
                    evaluate.
        bestPoints add: (optimizingPointClass
                    vector: (projectedFunction argumentWith:
                                                    partialResult)
                    function: functionBlock)]
```

computeInitialValues

```
bestPoints
    add: (optimizingPointClass vector: result function:
                                                    functionBlock).
self buildInitialSimplex.
worstVector := bestPoints removeLast position
```

computePrecision

```
 | functionValues bestFunctionValue |
 functionValues := bestPoints collect: [ :each | each value].
 bestFunctionValue := functionValues removeFirst.
 ^functionValues inject: 0
                into: [ :max :each | ( self precisionOf: ( each -
bestFunctionValue) abs relativeTo: bestFunctionValue abs) max: max]
```

contract

```
| bestVector oldVectors |
bestVector := bestPoints first position.
oldVectors := OrderedCollection with: worstVector.
[bestPoints size > 1] whileTrue: [oldVectors add: bestPoints
```

```
                                                    removeLast position].
    oldVectors do: [:each | self contract: each around: bestVector].
    worstVector := bestPoints removeLast position.
    ^self computePrecision
```

contract: aVector **around:** bestVector

```
    bestPoints
        add: (optimizingPointClass vector: bestVector * 0.5 +
                                                    (aVector * 0.5)
                function: functionBlock)
```

evaluateIteration

```
    | centerOfGravity newPoint nextPoint |
    centerOfGravity := (bestPoints inject: ((worstVector copy)
                        atAllPut: 0;
                        yourself)
                into: [:sum :each | each position + sum]) * (1 /
                                                    bestPoints size).
    newPoint := optimizingPointClass vector: 2 * centerOfGravity -
                                                    worstVector
                function: functionBlock.
    (newPoint betterThan: bestPoints first)
        ifTrue:
            [nextPoint := optimizingPointClass
                        vector: newPoint position * 2 -
                                                    centerOfGravity
                        function: functionBlock.
            (nextPoint betterThan: newPoint) ifTrue: [newPoint :=
                                                    nextPoint]]
        ifFalse:
            [newPoint := optimizingPointClass
                        vector: centerOfGravity * 0.666667 +
                                                (worstVector * 0.333333)
                        function: functionBlock.
            (newPoint betterThan: bestPoints first) ifFalse: [^self
                                                    contract]].
    worstVector := bestPoints removeLast position.
    bestPoints add: newPoint.
    result := bestPoints first position.
    ^self computePrecision
```

printOn: aStream

```
    super printOn: aStream.
    aStream cr.
    worstVector printOn: aStream.
```

11.7.3 Simplex Algorithm—Java Implementation

Listing 11.15 shows the Java implementation of the simplex algorithm. Code Example 11.6 shows how to invoke the class to find the minimum of a vector function.

Code Example 11.6

```
ManyVariableFunction func = <the goal function>
double[] educatedGuess = <an array of double representing one point in the
search space>

   MinimizingPointFactory strategy = new MinimizingPointFactory();
   SimplexOptimizer simplex = new SimplexOptimizer( func, strategy);
   simplex.setInitialValue( educatedGuess);
   simplex.evaluate();
   double[] result = simplex.getResult();
```

Except for the lines creating the strategy and the optimizer, this code example is identical to that of the Powell's algorithm (Code Example 11.4).

The class `SimplexOptimizer` is a subclass of class `MultiVariableOptimizer`. The additional instance variable `simplex` contains an array of optimizing vectors whose positions are the summit of the simplex.

At the end of the contraction operation, the best points must be sorted—using the method `sortPoints` of the superclass—because this transformation can alter the order of the points considerably. In all other cases, one has found the best point so far; the method `addBestPoint` is then used to discard the worst point, shift the remaining points, and add the best point in the first position of the array `simplex`.

Listing 11.16 Java implementation of simplex algorithm

```
package DhbOptimizing;

import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbMatrixAlgebra.DhbVector;

// Simplex optimizer of many-variable functions.

// @author Didier H. Besset

public class SimplexOptimizer extends MultiVariableOptimizer
{

    // Best value found so far.

    private OptimizingVector[] simplex;

// Constructor method.
```

```
// @param func DhbInterfaces.ManyVariableFunction
// @param pointCreator DhbOptimizing.OptimizingPointFactory
// @param initialValue double[]

public SimplexOptimizer(DhbInterfaces.ManyVariableFunction func,
        OptimizingPointFactory pointCreator, double[] initialValue)
{
    super(func, pointCreator, initialValue);
}

// Add a new best point to the simplex
// @param v DhbOptimizing.OptimizingVector

private void addBestPoint( OptimizingVector v)
{
    int n = simplex.length;
    while ( --n > 0 )
        simplex[n] = simplex[n-1];
    simplex[0] = v;
}

// @return boolean    true if a better point was found
// @param g DhbVector    summit whose median is contracted
// @exception DhbIllegalDimension if dimension of initial value is 0.

private boolean addContraction(DhbVector g)
                                        throws DhbIllegalDimension
{
    g.accumulate( simplex[result.length].getPosition());
    g.scaledBy( 0.5);
    OptimizingVector contractedPoint =
                                    pointFactory.createVector( g, f);
    if ( contractedPoint.betterThan( simplex[0]) )
    {
        addBestPoint( contractedPoint);
        return true;
    }
    else
        return false;
}

// @return boolean    true if a better point was found
// @exception DhbIllegalDimension if dimension of initial value is 0.

private boolean addReflection( DhbVector centerOfgravity)
                                        throws DhbIllegalDimension
```

```
{
    DhbVector reflectedVector = centerOfgravity.product( 2);
    reflectedVector.accumulateNegated(
                                simplex[result.length].getPosition());
    OptimizingVector reflectedPoint =
                    pointFactory.createVector( reflectedVector, f);
    if ( reflectedPoint.betterThan( simplex[0]) )
    {
        reflectedVector.scaledBy( 2);
        reflectedVector.accumulateNegated( centerOfgravity);
        OptimizingVector expandedPoint =
                    pointFactory.createVector( reflectedVector, f);
        if ( expandedPoint.betterThan( reflectedPoint) )
            addBestPoint( expandedPoint);
        else
            addBestPoint( reflectedPoint);
        return true;
    }
    else
        return false;
}

// @return DhbVector    center of gravity of best points of simplex,
//                              except worst one

private DhbVector centerOfGravity() throws DhbIllegalDimension
{
    DhbVector g = new DhbVector( result.length);
    for ( int i = 0; i < result.length; i++ )
        g.accumulate( simplex[i].getPosition());
    g.scaledBy( 1.0 / result.length);
    return g;
}

// @return double    maximum simplex extent in each direction

private double computePrecision()
{
    int i, j;
    double[] position = simplex[0].getPosition();
    double[] min = new double[ position.length];
    double[] max = new double[ position.length];
    for ( i = 0; i < position.length; i++ )
    {
        min[i] = position[i];
        max[i] = position[i];
```

```
        }
        for ( j = 1; j < simplex.length; j++ )
        {
            position = simplex[j].getPosition();
            for ( i = 0; i < position.length; i++ )
            {
                min[i] = Math.min( min[i], position[i]);
                max[i] = Math.max( max[i], position[i]);
            }
        }
        double eps = 0;
        for ( i = 1; i < position.length; i++ )
            eps = Math.max( eps, relativePrecision( max[i]-min[i],
                  result[i]));
        return eps;
    }

    // Reduce the simplex from the best point.

    private void contractSimplex()
    {
        double[] bestPoint = simplex[0].getPosition();
        for ( int i = 1; i < simplex.length; i++)
            simplex[i].contractFrom( bestPoint);
        sortPoints( simplex);
    }

    // Here precision is the largest extent of the simplex.

    public double evaluateIteration()
    {
        try {
            double bestValue = simplex[0].getValue();
            DhbVector g = centerOfGravity();
            if ( !addReflection( g) )
            {
                if ( !addContraction( g) )
                    contractSimplex();
            }
            result = simplex[0].getPosition();
            return computePrecision();
        } catch ( DhbIllegalDimension e) { return 1;};
    }

    // Create a Simplex by finding the optimum in each direction
    // starting from the initial value.
```

```
public void initializeIterations()
{
    double [] v = new double[ result.length];
    for ( int i = 0; i < result.length; i++ )
        v[i] = 0;
    VectorProjectedFunction projection =
                        new VectorProjectedFunction( f, result, v);
    OneVariableFunctionOptimizer unidimensionalFinder =
        new OneVariableFunctionOptimizer( projection, pointFactory);
    unidimensionalFinder.setDesiredPrecision( getDesiredPrecision());
    simplex = new OptimizingVector[result.length+1];
    try {
        for ( int i = 0; i < result.length; i++ )
            {
                v[i] = 1;
                projection.setDirection( v);
                v[i] = 0;
                unidimensionalFinder.setInitialValue( 0);
                unidimensionalFinder.evaluate();
                simplex[i] = pointFactory.createVector(
                                projection.argumentAt(
                                unidimensionalFinder.getResult()), f);
            }
    } catch ( DhbIllegalDimension e) { };
    simplex[result.length] = pointFactory.createVector( result, f);
    sortPoints( simplex);
}

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

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( simplex[0]);
    for ( int i = 1; i < simplex.length; i++ )
    {
        sb.append( '\n');
        sb.append( simplex[i]);
    }
    return sb.toString();
}
}
```

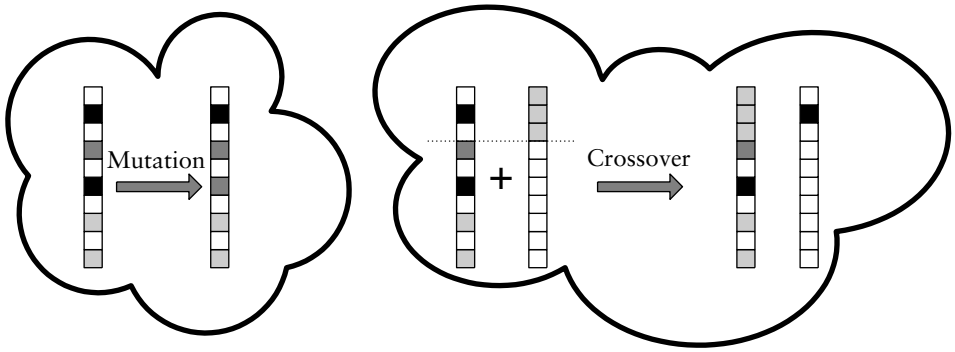

FIG. 11.5 Mutation and crossover reproduction of chromosomes

11.8 Genetic Algorithm

All optimizing algorithms discussed so far have one common flaw: they all terminate when a local optimum is encountered. In most problems, however, one wants to find the absolute optimum of the function, especially if the goal function represents some economical merit.

One academic example is the maximization of the function

$$f(\mathbf{x}) = \frac{\sin^2 |\mathbf{x}|}{|\mathbf{x}|^2}. \tag{11.7}$$

This function has an absolute maximum at $\mathbf{x} = 0$, but all algorithms discussed so far will end up inside a ring corresponding to $|\mathbf{x}| = n\pi/2$, where n is any positive odd integer.

In 1975, John Holland introduced a new type of algorithm, dubbed *genetic algorithm* because it tries to mimic the evolutionary process identified as the cause for the diversity of living species by Charles Darwin. In a genetic algorithm, the elements of the search space are considered as the chromosomes of individuals; the goal function is considered as the measure of the fitness of the individual to adapt itself to its environment [Berry & Linoff], [Koza *et al.*]. The iterations are aping (pun intended) the Darwinian principle of survival and reproduction. At each iteration, the fittest individuals survive and reproduce themselves. To bring some variability to the algorithm, mutation and crossover of chromosomes are taken into account.

Mutation occurs when one gene of a chromosome is altered at reproduction time. Crossover occurs when two chromosomes break themselves and recombine with the piece coming from the other chromosome. These processes are illustrated in Figure 11.5. The point where the chromosomes are breaking is called the *crossover point*. Which individual survives and reproduces itself, when and where mutation occurs, and when and where a crossover happens is determined randomly. This is precisely the random nature of the algorithm that gives it the ability to jump out of a local optimum to look further for the absolute optimum.

11.8.1 Mapping the Search Space on Chromosomes

To be able to implement a genetic algorithm, one must establish how to represent the genes of a chromosome. At the smallest level the genes could be the bits of the structure representing the chromosome. If the search space of the goal function does cover the domain generated by all possible permutations of the bits, this is a good approach. However, this is not always a practical solution since some bit combinations may be forbidden by the structure. For example, some of the combinations of a 64-bit word do not correspond to a valid floating-point number.

In the case of the optimization of a vector function, the simplest choice is to take the components of the vector as the genes. Genetic algorithms are used quite often to adjust the parameters of a neural network [Berry & Linoff]. In this case, the chromosomes are the coefficients of each neuron. Chromosomes can even be computer subprograms in the case of genetic programming [Koza *et al.*]. In this latter case, each individual is a computer program trying to solve a given problem.

Figure 11.6 shows a flow diagram of a general genetic algorithm. The reproduction of the individual is taken literally: a copy of the reproducing individual is copied into the next generation. The important feature of a generic algorithm is that the building of the next generation is a random process. To ensure the survival of the fittest, the selection of the parents of the individuals of the next generation is performed at random with uneven probability: the fittest individuals have a larger probability of being selected than the others. Mutation enables the algorithm to create individuals having genes corresponding to unexplored regions of the search space. Most of the times such mutants will be discarded at the next iteration but, in some cases, a mutation may uncover a better candidate. In the case of the function of equation 11.7, this would correspond to jumping from one ring to another ring closer to the function's maximum. Finally, the crossover operation mixes good genes in the hope of building a better individual out of the properties of good inidividuals. Like mutation, the crossover operation gives a stochastic behavior to the algorithm, enabling it to explore uncharted regions of the search space.

Note: Because of its stochastic nature, a genetic algorithm is the algorithm of choice when the goal function is expressed on integers.

11.8.2 Genetic Algorithm—General Implementation

The left-hand side of the diagram of Figure 11.6 is quite similar to the flow diagram of an iterative process (see Figure 4.2. Thus, the class implementing the genetic algorithm is a subclass of the iterative process class discussed in Chapter 4.

The *genetic* nature of the algorithm is located on the right hand side of the diagram of Figure 11.6. As mentioned before, the implementation of the chromosomes is highly problem-dependent. All operations located in the top portion of the mentioned area can be expressed in generic terms without any knowledge of the chromosomic implementation. To handle the lower part of the right-hand side of the diagram of Figure 11.6, a new object shall implemented, the chromosome manager.

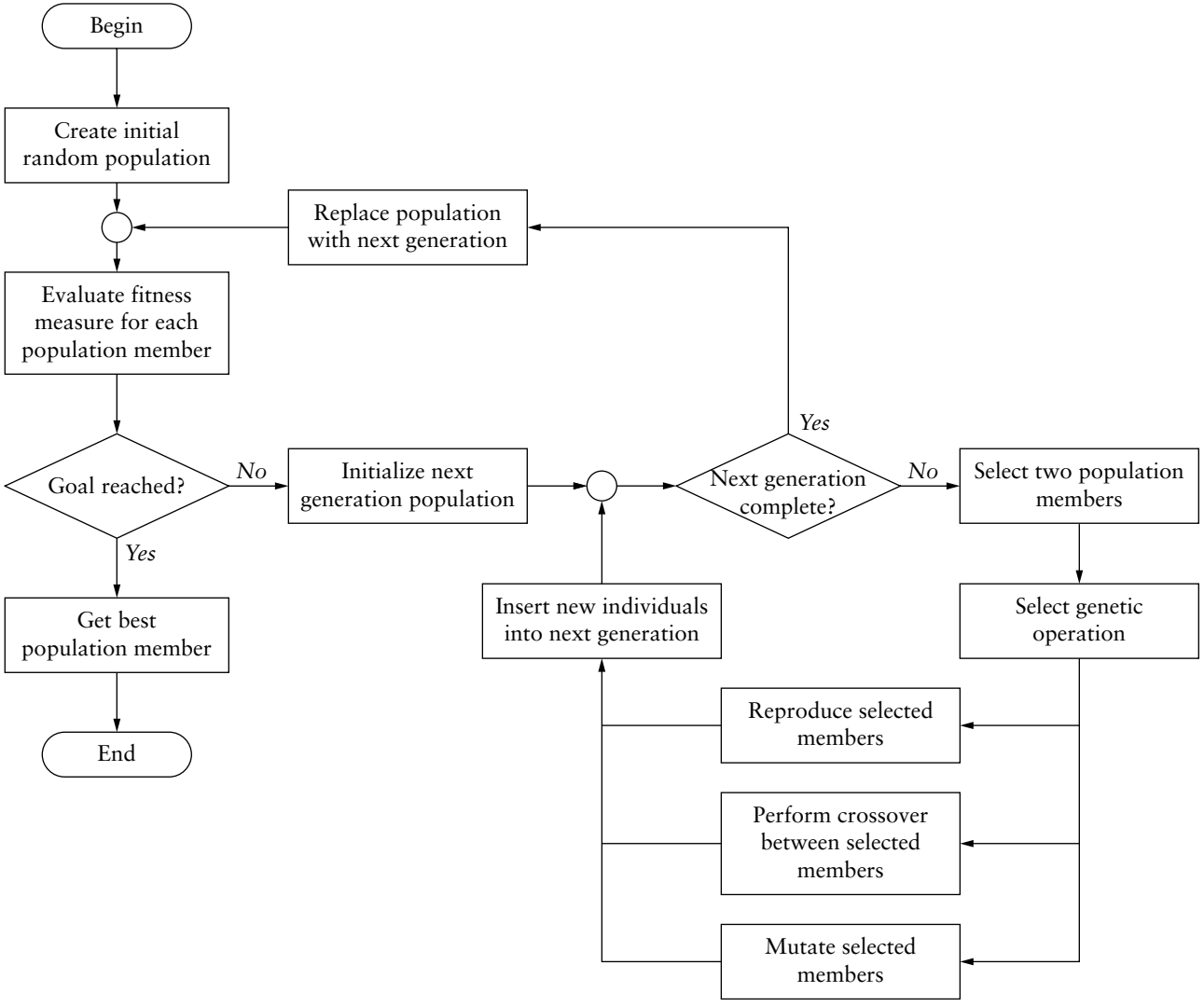

FIG. 11.6 General purpose genetic algorithm

One should also notice that the value of the function is not needed when the next generation is built. Thus, the chromosome manager does not need to have any knowledge of the goal function. The goal function comes into play when transferring the next generation to the *mature* population—that is, the population used for reproduction at the next iteration. At the maturity stage, the value of the goal function is needed to identify the fittest individuals. In our implementation, the next generation is maintained by the chromosome manager, whereas the population of mature individuals is maintained by the object in charge of the genetic algorithm that has the knowledge of the goal function.
The chromosome manager has the following instance variables:

`populationSize` contains the size of the population; one should pick up a large enough number to be able to cover the search space efficiently: the larger the dimension of the space search space, the larger must be the population size.

`rateOfMutation` contains the probability of having a mutation while reproducing.

`rateOfCrossover` contains the probability of having a crossover while reproducing.

All of these variables have getter and setter accessor methods. In addition, a convenience instance creation method is supplied to create a chromosome manager with given values for all three instance variables. The chromosome manager implements the following methods:

`isFullyPopulated` signals that a sufficient number of individuals has been generated into the population.

`process` processes a pair of individuals; this method does the selection of the genetic operation and applies it; individuals are processed by pair to always have a possibility of crossover.

`randomnizePopulation` generates a random population;

`reset` to create an empty population for the next generation.

Finally, the chromosome manager must also implement methods performing each of the genetic operations: reproduction, mutation, and crossover. The Smalltalk implementation supplies methods that returns a new individual; the Java implementation supplies methods that add a new individual to the population. The reason for this difference comes from the static typing requirements of Java.

The genetic optimizer is the object implementing the genetic algorithm proper. It is a subclass of the iterative process class described in Section 4.2. In addition to the handling of the iterations, the genetic optimizer implements the steps of the algorithm drawn on the top part of the right hand side of the diagram of Figure 11.6. It has one instance variable containing the chromosome manager with which it will interact. The instance creation method take three arguments: the function to optimize, the optimizing strategy, and the chromosome manager.

The method `initializeIteration` asks the chromosome manager to supply a random population. The method `evaluateIteration` performs the loop of the right hand side of the diagram of Figure 11.6. It selects a pair of parents on which the chromosome manager performs the genetic operation.

Selecting the genetic operation is performed with a random generator. The values of the goal function are used as weights. Let $f(p_i)$ be the value of the goal function for individual p_i, and let p_b and p_w be, respectively, the fittest and the least fit individual found so far (b stands for *best*, and w stands for *worst*). One first computes the unnormalized probability:

$$\tilde{P}_i = \frac{f(p_i) - f(p_w)}{f(p_b) - f(p_w)}. \tag{11.8}$$

This definition ensures that $\tilde{P}_i$ is always comprised between zero and 1 for any goal function. Then we can use the discrete probability

$$P_i = \frac{1}{\sum \tilde{P}_i} \tilde{P}_i. \tag{11.9}$$

The sum in equation 11.9 is taken over the entire population. An attentive reader will notice than this definition assigns a zero probability of selecting the worst individuals. This gives a slight bias to our implementation compared to the original algorithm, which can be easily compensated for by taking a sufficiently large population. The method `randomScale` calculates the P_i of equation 11.9 and returns an array containing the integrated sum

$$R_i = \sum_{k=0}^{i} P_i. \tag{11.10}$$

The array R_i is used to generate a random index to select individuals for reproduction.

The transfer between the next generation and the mature population is performed by the method `collectPoints`.

In the general case, there is no possibility to decide when to terminate the algorithm. In practice, it is possible that the population stays stable for quite a while until suddenly a new individual is found to be better than the rest. Therefore, a criterion based on the stability of the first best points is likely to beat the purpose of the algorithm—namely, to jump out of a local optimum. Some problems can define a threshold at which the goal function is considered sufficiently good. In this case, the algorithm can be stopped as soon as the value of the goal function for the fittest individual becomes better than that threshold. In the general case, however, the implementation of the genetic algorithm simply returns a constant pseudoprecision—set to 1—and runs until the maximum number of iterations becomes exhausted.

11.8.3 Genetic Algorithm—Smalltalk Implementation

Listing 11.17 shows the code of an abstract chromosome manager in Smalltalk and of a concrete implementation for vector chromosomes. The class `DhbChromosomeManager` has one instance variable in addition to the variables listed in Section 11.8.2: `population`. This variable is an instance of an `OrderedCollection` containing the individuals of the next generation being prepared.

The class `DhbVectorChromosomeManager` is a sublcass of class `DhbChromosomeManager` implementing vector chromosomes. It has two instance variables:

`origin` A vector containing the minimum possible values of the generated vectors

`range` A vector containing the range of the generated vectors

In other words, `origin` and `range` are delimiting an hypercube defining the search space.

Listing 11.17 Smalltalk chromosome: abstract and concrete

Class	`DhbChromosomeManager`
Subclass of	`Object`
Instance variable names:	`population populationSize rateOfMutation rateOfCrossover`

Class Methods

new: `anInteger` **mutation:** `aNumber1`

crossover: `aNumber2`

```
^self new populationSize: anInteger; rateOfMutation: aNumber1;
                                rateOfCrossover: aNumber2; yourself
```

Instance Methods

clone: `aChromosome`

```
^aChromosome copy
```

crossover: `aChromosome1` **and:** `aChromosome2`

```
^self subclassResponsibility
```

isFullyPopulated

```
^population size >= populationSize
```

mutate: aChromosome

```
^self subclassResponsibility
```

population

```
^population
```

populationSize: anInteger

```
populationSize := anInteger.
```

process: aChromosome1 **and:** aChromosome2

```
| roll |
roll := Number random.
roll < rateOfCrossover
    ifTrue: [population addAll: (self crossover: aChromosome1
                                            and: aChromosome2)]
    ifFalse:
        [roll < (rateOfCrossover + rateOfMutation)
            ifTrue:
                [population
                    add: (self mutate: aChromosome1);
                    add: (self mutate: aChromosome2)]
            ifFalse:
                [population
                    add: (self clone: aChromosome1);
                    add: (self clone: aChromosome2)]]
```

randomizePopulation

```
self reset.
[ self isFullyPopulated] whileFalse: [ population add: self
                                                randomChromosome].
```

rateOfCrossover: aNumber

```
(aNumber between: 0 and: 1)
    ifFalse: [self error: 'Illegal rate of cross-over'].
rateOfCrossover := aNumber
```

rateOfMutation: aNumber

```
(aNumber between: 0 and: 1)
    ifFalse: [self error: 'Illegal rate of mutation'].
rateOfMutation := aNumber
```

reset

```
population := OrderedCollection new: populationSize.
```

Class	DhbVectorChromosomeManager
Subclass of	DhbChromosomeManager
Instance variable names:	origin range

Instance Methods

crossover: aChromosome1 **and:** aChromosome2

```
| index new1 new2|
index := ( aChromosome1 size - 1) random + 2.
new1 := self clone: aChromosome1.
new1 replaceFrom: index to: new1 size with: aChromosome2
                                                startingAt: index.
new2 := self clone: aChromosome2.
new2 replaceFrom: index to: new2 size with: aChromosome1
                                                startingAt: index.
^Array with: new1 with: new2
```

mutate: aVector

```
| index |
index := aVector size random + 1.
^( aVector copy)
        at: index put: ( self randomComponent: index);
        yourself
```

origin: aVector

```
origin := aVector.
```

randomChromosome

```
^( ( 1 to: origin size) collect: [ :n | self randomComponent: n])
                                                        asVector
```

randomComponent: anInteger

```
^( range at: anInteger) random + ( origin at: anInteger)
```

range: aVector

```
range := aVector.
```

Listing 11.18 describes how the genetic optimizer is implemented in Smalltalk. Code Example 11.7 shows how to use a genetic optimizer to find the maximum of a vector function.

Code Example 11.7

```
    | fBlock optimizer manager origin range result |
fBlock := ¡the goal function ¿
origin := ¡a vector containing the minimum expected value of the component¿
range := ¡a vector containing the expected range of the component¿

    optimizer := DhbGeneticOptimizer maximizingFunction: fBlock.
    manager := DhbVectorChromosomeManager new: 100 mutation:
        0.1 crossover: 0.1.
    manager origin: origin; range: range.
    optimizer chromosomeManager: manager.
    result := optimizer evaluate.
```

After establishing the goal function and the search space, an instance of the genetic optimizer is created. The next line creates an instance of a vector chromosome manager for a population of 100 individuals (sufficient for a two- to three-dimensional space) and rates of mutation and crossover equal to 10%. The next line defines the search space into the chromosome manager. The final line performs the genetic search and returns the result.

In Smalltalk, the population of the next generation is maintained in the instance variable `population`. Each time a next generation has been established, it is transferred into a collection of best points by the method `collectPoints`. Each element of the collection `bestPoints` is an instance of a subclass of `OptimizingPoint`. The exact type of the class is determined by the search strategy. Since best points are sorted automatically, the result is always the position of the first element of `bestPoints`.

Listing 11.18 Smalltalk implementation of genetic algorithm

Class	`DhbGeneticOptimizer`
Subclass of	`DhbFunctionOptimizer`
Instance variable names:	`chromosomeManager`

Class Methods

defaultMaximumIterations

```
^500
```

defaultPrecision

```
^0
```

Instance Methods

chromosomeManager: aChromosomeManager

```
chromosomeManager := aChromosomeManager.
^self
```

collectPoints

```
| bestPoint |
bestPoints notEmpty
    ifTrue: [ bestPoint := bestPoints removeFirst].
bestPoints removeAll: bestPoints asArray.
chromosomeManager population do: [:each | self addPointAt: each].
bestPoint notNil
    ifTrue: [ bestPoints add: bestPoint].
result := bestPoints first position.
```

computePrecision

```
^1
```

evaluateIteration

```
| randomScale |
randomScale := self randomScale.
chromosomeManager reset.
[ chromosomeManager isFullyPopulated]
    whileFalse: [ self processRandomParents: randomScale].
self collectPoints.
^self computePrecision
```

initializeIterations

```
chromosomeManager randomizePopulation.
self collectPoints
```

processRandomParents: aNumberArray

```
chromosomeManager process: ( bestPoints at: ( self randomIndex:
                                        aNumberArray)) position
                    and:  ( bestPoints at: ( self randomIndex:
                                        aNumberArray)) position.
```

randomIndex: aNumberArray

```
| x n |
x := Number random.
n := 1.
aNumberArray do:
    [ :each |
      x < each
        ifTrue: [ ^n].
      n := n + 1.
    ].
^aNumberArray size
```

randomScale

```
| norm fBest fWorst answer|
fBest := bestPoints first value.
fWorst := bestPoints last value.
norm := 1 / ( fBest - fWorst).
answer := bestPoints collect: [ :each | (each value - fWorst) *
                                                              norm].
norm := 1 / ( answer inject: 0 into: [ :sum :each | each + sum]).
fBest := 0.
^answer collect: [ :each | fBest := each * norm + fBest. fBest]
```

11.8.4 Genetic Algorithm—Java Abstract Implementation

The Java implementation of the genetic algorithm is done in two parts because the type of population is not known until one uses a concrete chromosome manager. One could use casting in order to preserve anonymity of the chromosome type at the level of the genetic algorithm. I tend to avoid cast operator like the plague and use them only when all other alternatives have failed. In this case, casting is avoided by maintaining a concrete implementation of the chromosome manager in pair with a class implementing the genetic algorithm. This section describes the features of the abstract classes. The code for these classes is shown in Listings 11.19 and 11.20. The next section describes a concrete implementation for vector functions.

The class `ChromosomeManager` has an additional instance variable, called `generator`, used to keep an instance of a random generator. My implementation uses the default random generator provided by Java. The method `reset` allows the creation of an empty next generation. The current size of the next generation is obtained by calling method `getPopulationSize`. Other methods are provided to provide interaction with the genetic algorithm class. The methods `individualAt` returns the *k*th individual of the next generation, where *k* is the supplied integer argument. The methods `addCloneOf`, and `addMutationOf` add a new individual to the next generation by applying, respectively, reproduction or mutation. The method `addCrossoversOf` adds the two crossover offsprings of two individuals. Here, the reader

will see that casting becomes unavoidable by the subclasses. However, since the objects taken as arguments by the methods `addCloneOf`, `addMutationOf`, and `addCrossoversOf` are coming from the method `individualAt` and only that method, the risk of error is minimal.

Listing 11.19 Java abstract implementation of a chromosome

```
package DhbOptimizing;

import java.util.Random;

// Abstract chromosome manager.
// (genetic algorithm)
//  * @author Didier H. Besset

public abstract class ChromosomeManager
{

    // Population size.

    private int populationSize = 100;

    // Rate of mutation.

    private double rateOfMutation = 0.1;

    // Rate of crossover.

    private double rateOfCrossover = 0.1;

    // Random generator.

    private Random generator = new Random();

// Constructor method.

public ChromosomeManager()
{
    super();
}

// Constructor method.
// @param n int
// @param mRate double
// @param cRate double
```

```
public ChromosomeManager(int n, double mRate, double cRate)
{
    populationSize = n;
    rateOfMutation = mRate;
    rateOfCrossover = cRate;
}

// @param x java.lang.Object

public abstract void addCloneOf( Object x);

// @param x java.lang.Object

public abstract void addCrossoversOf( Object x, Object y);

// @param x java.lang.Object

public abstract void addMutationOf( Object x);
public abstract void addRandomChromosome();

// @return int    the current size of the population

public abstract int getCurrentPopulationSize();

// @return int    desired population size.

public int getPopulationSize( )
{
    return populationSize;
}

// @return java.lang.Object (must be casted into the proper type
//                    of chromosome)
// @param n int

public abstract Object individualAt( int n);

// @return boolean    true if the new generation is complete

public boolean isFullyPopulated()
{
    return getCurrentPopulationSize() >= populationSize;
}

// @return double    a random number (delegated to the generator)
```

```
public double nextDouble()
{
    return generator.nextDouble();
}

// @param x java.lang.Object
// @param y java.lang.Object

public void process( Object x, Object y)
{
    double roll = generator.nextDouble();
    if ( roll < rateOfCrossover )
        addCrossoversOf( x, y);
    else if ( roll < rateOfCrossover + rateOfMutation )
    {
        addMutationOf( x);
        addMutationOf( y);
    }
    else
    {
        addCloneOf( x);
        addCloneOf( y);
    }
}

// Create a population of random chromosomes.

public void randomizePopulation()
{
    reset();
    while ( !isFullyPopulated() )
        addRandomChromosome();
}

// Reset the population of the receiver.

public abstract void reset();

// @param n int    desired population size.

public void setPopulationSize( int n)
{
    populationSize = n;
}

// @param n int    desired rate of crossover
```

```
public void setRateOfCrossover( int cRate)
{
    rateOfCrossover = cRate;
}

// @param n int    desired rate of mutation

public void setRateOfMutation( int mRate)
{
    rateOfMutation = mRate;
}
}
```

The class `GeneticOptimizer` is an abstract class. The methods providing the functionality to fill up the new population from the next generation established by the chromosome manager are all abstract methods. The method `collectPoint` transfers a single individual from the next generation to the population of mature individuals.

A method `initializeIterations` with one integer argument is needed to allow the concrete class to initialize its memory for the desired population size. This allows the instance variable `populationSize` to remain private.

Listing 11.20 Java implementation of genetic algorithm

```
package DhbOptimizing;


// Abstract genetic algorithm.

// @author Didier H. Besset

public abstract class GeneticOptimizer extends MultiVariableOptimizer
{

    // Chromosome manager.

    private ChromosomeManager chromosomeManager;

// Constructor method.
// @param func DhbInterfaces.ManyVariableFunction
// @param pointCreator DhbOptimizing.OptimizingPointFactory
// @param chrManager ChromosomeManager

public GeneticOptimizer(DhbInterfaces.ManyVariableFunction func,
    OptimizingPointFactory pointCreator, ChromosomeManager chrManager)
```

```
{
    super(func, pointCreator, null);
    chromosomeManager = chrManager;
}

// @param x java.lang.Object

public abstract void collectPoint(Object x);

// Collect points for the entire population.

public void collectPoints()
{
    reset();
    for ( int i = 0; i < chromosomeManager.getPopulationSize(); i++ )
        collectPoint( chromosomeManager.individualAt(i));
}

// This method causes the receiver to exhaust the maximum number of
// iterations. It may be overloaded by a subclass (hence "protected")
// if a convergence criteria can be defined.
// @return double

protected double computePrecision()
{
    return 1;
}

// @return double

public double evaluateIteration()
{
    double[] randomScale = randomScale();
    chromosomeManager.reset();
    while ( !chromosomeManager.isFullyPopulated() )
    {
        chromosomeManager.process(
                            individualAt( randomIndex(randomScale)),
                            individualAt( randomIndex(randomScale)));
    }
    collectPoints();
    return computePrecision();
}

// @return java.lang.Object    (must be casted into the proper type)
// @param n int
```

```
public abstract Object individualAt(int n);

// Create a random population.

public void initializeIterations()
{
    initializeIterations( chromosomeManager.getPopulationSize());
    chromosomeManager.randomizePopulation();
    collectPoints();
}

// @param n int    size of the initial population

public abstract void initializeIterations( int n);

// @return int    an index generated randomly
// @param randomScale double[]    fitness scale (integral)

protected int randomIndex( double[] randomScale)
{
    double roll = chromosomeManager.nextDouble();
    if ( roll < randomScale[0] )
        return 0;
    int n = 0;
    int m = randomScale.length;
    int k;
    while ( n < m - 1 )
    {
        k = ( n + m) / 2;
        if ( roll < randomScale[k] )
            m = k;
        else
            n = k;
    }
    return m;
}

// @return double[]    integral fitness scale.

public abstract double[] randomScale();
public abstract void reset();
}
```

11.8.5 Genetic Algorithm—Java Implementation with Vectors

Listings 11.21 and 11.22 show the concrete Java classes for the chromosome manager and the genetic optimizer, respectively. They provide a concrete implementation of a genetic algorithm for finding the optimum of a vector function. Code Example 11.8 shows how to find the maximum of a vector function using these classes.

Code Example 11.8

```
ManyVariableFunction func = ¡the goal function¿
DhbVector origin = ¡a vector containing the minimum expected value of the component¿
DhbVector range = ¡a vector containing the expected range of the component¿

    MaximizingPointFactory strategy = new MaximizingPointFactory();
    VectorChromosomeManager manager =
                      new VectorChromosomeManager( 100, 0.1, 0.1);
    manager.setOrigin( origin);
    manager.setRange( range);
    VectorGeneticOptimizer finder =
               new VectorGeneticOptimizer( func, strategy, manager);
    finder.evaluate();
    double[] result = finder.getResult();
```

The line after the definition of the goal function and the hypercube defining the search space creates an instance of a maximizing vector. This will be the STRATEGY of the genetic optimizer. The next statement creates an instance of a vector chromosome manager. Right after, two statements define the search space. Then, the instance of the genetic optimizer is created. The next statement performs the genetic algorithm, and the last statement retrieves the result.

The class `VectorChromosomeManager` is a concrete subclass of the class `ChromosomeManager`. It maintains the next generation in the instance variable `population`. The instance variable `fillIndex` is used as an index when filling up the next generation.

Listing 11.21 Java implementation of a vector chromosome

```
package DhbOptimizing;

import DhbMatrixAlgebra.DhbVector;

// Chromosome manager for vector chromosomes.
// (genetic algorithm)

// @author Didier H. Besset
```

```
public class VectorChromosomeManager extends ChromosomeManager
{

    // Population.

    private DhbVector[] population;

    // Current population size.

    private int fillIndex;

    // Origin of values.

    private DhbVector origin;

    // Range of values.

    private DhbVector range;

// Default constructor method.

public VectorChromosomeManager() {
    super();
}

// Constructor method.
// @param n int
// @param mRate double
// @param cRate double

public VectorChromosomeManager(int n, double mRate, double cRate) {
    super(n, mRate, cRate);
}

// @param x DhbVector

public void addCloneOf(Object x)
{
    double[] v = ((DhbVector) x).toComponents();
    try { population[fillIndex++] = new DhbVector( v);}
        catch( NegativeArraySizeException e) {};
}

// @param x DhbVector
// @param y DhbVector
```

```
public void addCrossoversOf(Object x, Object y)
{
    double[] v = ((DhbVector) x).toComponents();
    double[] w = ((DhbVector) x).toComponents();
    int n = (int) ( nextDouble() * ( origin.dimension() - 1));
    double temp;
    for ( int i = 0; i < n; i++ )
    {
        temp = v[i];
        v[i] = w[i];
        w[i] = temp;
    }
    try { population[fillIndex++] = new DhbVector( v);
          population[fillIndex++] = new DhbVector( w);
        } catch( NegativeArraySizeException e) {};
}

// @param x DhbVector

public void addMutationOf(Object x)
{
    double[] v = ((DhbVector) x).toComponents();
    int i = (int) ( nextDouble() * origin.dimension());
    v[i] = randomComponent(i);
    try { population[fillIndex++] = new DhbVector( v);}
        catch( NegativeArraySizeException e) {};
}
public void addRandomChromosome()
{
    double[] v = new double[origin.dimension()];
    for ( int i = 0; i < origin.dimension(); i++ )
        v[i] = randomComponent(i);
    try { population[fillIndex++] = new DhbVector( v);}
        catch( NegativeArraySizeException e) {};
}

// @return int    the current size of the population

public int getCurrentPopulationSize()
{
    return fillIndex;
}

// @return Vector    vector at given index
// @param n int
```

```
public Object individualAt( int n)
{
    return population[n];
}

// @return double
// @param n int

private double randomComponent(int n)
{
    return origin.component(n) + nextDouble() * range.component(n);
}

// Allocated memory for a new generation.

public void reset()
{
    population = new DhbVector[getPopulationSize()];
    fillIndex = 0;
}

// @param x double    component of the origin of the hypercube
//    constraining the domain of definition of the function to optimize
// @exception java.lang.NegativeArraySizeException
//                when the size of the array is 0

public void setOrigin( double[] x) throws NegativeArraySizeException
{
    setOrigin(  new DhbVector( x));
}

// @param v DhbVector     origin of the hypercube
//    constraining the domain of definition of the function to optimize

public void setOrigin( DhbVector v)
{
    origin = v;
}

// @param x double    components of the lengths of the hypercube
//    constraining the domain of definition of the function to optimize
// @exception java.lang.NegativeArraySizeException
//                when the size of the array is 0

public void setRange( double[] x) throws NegativeArraySizeException
{
```

```
    setRange(  new DhbVector( x));
}

// @param v DhbVector    lengths of the hypercube
//    constraining the domain of definition of the function to optimize

public void setRange( DhbVector v)
{
    range = v;
}
}
```

The class `VectorGeneticOptimizer` is a concrete subclass of the class `GeneticOptimizer`. My implementation chooses to sort the points when they are collected. This is not really needed as the algorithm for selecting the individual does not assume that the individuals are sorted by fitness. This is quite practical, however, when following the behavior of the algorithm with the debugger. For heavy-duty usage, the sorting ought to be removed. If that is the case, the method `getResult` must be rewritten to fetch the fittest individual of the mature population.

Listing 11.22 Java implementation of genetic algorithm for vectors

```
package DhbOptimizing;

import DhbMatrixAlgebra.DhbVector;
import DhbInterfaces.ManyVariableFunction;

// Genetic optimizer of many-variable functions.

// @author Didier H. Besset

public class VectorGeneticOptimizer extends GeneticOptimizer
{

    // Best values found so far.

    private OptimizingVector[] bestPoints;

    // Number of points filled so far.

    private int fillIndex;

// Constructor method.
// @param func DhbInterfaces.ManyVariableFunction
// @param pointCreator DhbOptimizing.OptimizingPointFactory
```

```
// @param chrManager DhbOptimizing.ChromosomeManager

public VectorGeneticOptimizer( ManyVariableFunction func,
                               OptimizingPointFactory pointCreator,
                               ChromosomeManager chrManager)
{
    super(func, pointCreator, chrManager);
}

// @param x DhbVector

public void collectPoint(Object x)
{
    OptimizingVector v = pointFactory.createVector( (DhbVector) x, f);
    if ( fillIndex == 0 || bestPoints[fillIndex-1].betterThan(v) )
    {
        bestPoints[fillIndex++] = v;
        return;
    }
    int n = 0;
    int m = fillIndex - 1;
    if ( bestPoints[0].betterThan(v) )
    {
        int k;
        while ( m - n > 1 )
        {
            k = ( n + m) / 2;
            if ( v.betterThan(bestPoints[k]) )
                m = k;
            else
                n = k;
        }
        n = m;
    }
    for ( m = fillIndex; m > n; m-- )
        bestPoints[m] = bestPoints[m-1];
    bestPoints[n] = v;
    fillIndex += 1;
}

// @return double[]       best point found so far

public double[] getResult()
{
    return bestPoints[0].getPosition();
}
```

```
// @return DhbVector    vector at given index
// @param n int

public Object individualAt( int n)
{
    try { return new DhbVector( bestPoints[n].getPosition());}
        catch( NegativeArraySizeException e) { return null;};
}

// @param n int    size of the initial population

public void initializeIterations( int n)
{
    bestPoints = new OptimizingVector[n];
}

// @return double[]       fitness scale for random generation

public double[] randomScale()
{
    double[] f = new double[ bestPoints.length];
    double sum = 0;
    for ( int i = 0; i < bestPoints.length; i++ )
    {
        f[i] = bestPoints[i].getValue() + sum;
        sum += bestPoints[i].getValue();
    }
    sum = 1 / sum;
    for ( int i = 0; i < bestPoints.length; i++ )
        f[i] *= sum;
    return f;
}
public void reset()
{
    fillIndex = 0;
}

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

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( bestPoints[0]);
    for ( int i = 1; i < Math.min( bestPoints.length, 30); i++ )
    {
```

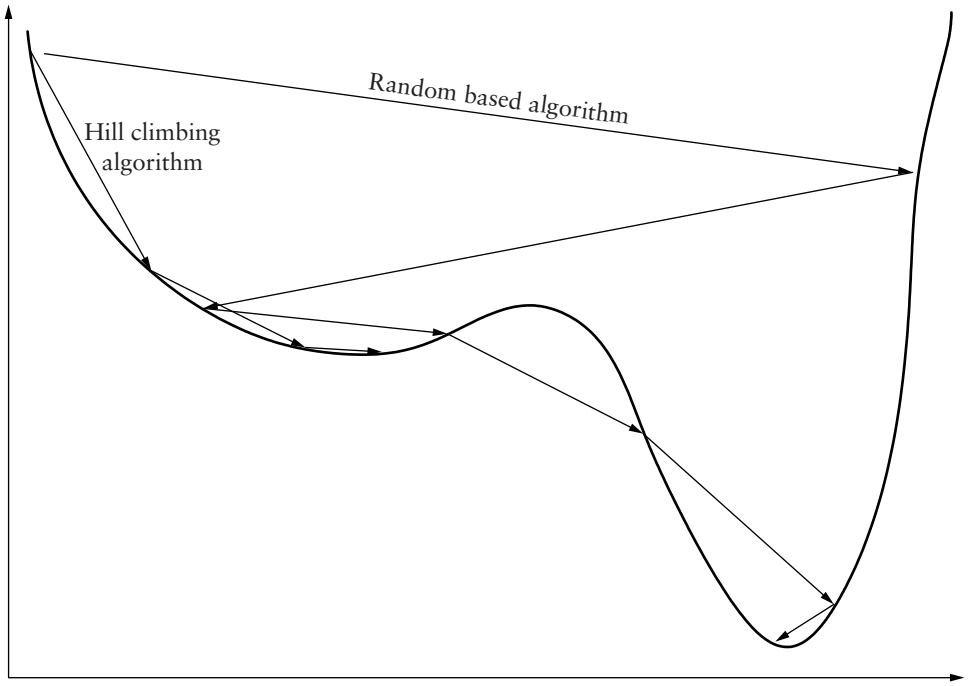

FIG. 11.7 Compared behavior of hill climbing and random based algorithms.

```
            sb.append( '\n');
            sb.append( bestPoints[i]);
        }
        return sb.toString();
    }
    }
```

11.9 Multiple Strategy Approach

As we have seen, most of the optimizing algorithms described so far have some limitation:

- Hill-climbing algorithms may get into trouble far from the optimum and may get caught into a local optimum. This is exemplified in Figure 11.7.
- The simplex algorithm may get caught into a local optimum and does not converge well near the optimum.
- Genetic algorithms do not have clear convergence criteria.

After reading this summary of the pro and cons of each algorithm, the reader may have already come to the conclusion that mixing the three algorithms together can make a very efficient strategy to find the optimum of a wide variety of functions.

One can start with a genetic optimizer for a sufficient number of iterations. This should ensure that the best points found at the end of the search do not lie too far from the absolute optimum. Then, one can use the simplex algorithm to get rapidly near the optimum. The final location of the optimum is obtained using a hill-climbing optimizer.

11.9.1 Multiple Strategy Approach—General Implementation

This multiple strategy approach, inspired from the program MINUIT, has been adapted to the use of the algorithms discussed here. The class `MultiVariableGeneralOptimizer` combines the three algorithms: genetic, simplex, and hill climbing, in this order. I could have made it a subclass of `Object`, but I decided to reuse all the management provided by the abstract optimizer class discussed in Section 11.3.1. Therefore, the general-purpose optimizer is a subclass of the abstract optimizer class although it does not really uses the framework of an iterative process. Only one additional instance variable is needed: the range used to construct the hypercube search space for the vector genetic chromosome manager. A corresponding setting method is provided: `setRange`.

The method `initializeIterations` performs search using the genetic algorithm as an option and then the simplex algorithm. Since the genetic algorithm requires a great deal of function to evaluate due to its stochastic nature, it is a good idea to give the user the choice of bypassing the use of the genetic algorithm. If no range has been defined, only the simplex algorithm is used from the supplied initial value. Otherwise, a search is made with the genetic algorithm using the initial value and the range to define the search space. Then the simplex algorithm is started from the best point found by the genetic algorithm. The precision for the simplex search is set to the square root of the precision for the final search. Less precision is required for this step because the final search will give a better precision.

The method `evaluateIteration` performs the hill climbing algorithm and returns its precision. Because the desired precision of the hill-climbing algorithm is set to that of the general purpose optimizer, there will only be a single iteration.

Listing 11.23 shows the implementation in Smalltalk. Listing 11.24 gives the code for the Java implementation. At this point, I shall abstain from commenting on the code as the reader should have no more need for such thing. . . hopefully!

Listing 11.23 Smalltalk implementation of a general optimizer

Class	`DhbMultiVariableGeneralOptimizer`
Subclass of	`DhbFunctionOptimizer`

Instance Methods

computeInitialValues

```
self range notNil
    ifTrue: [ self performGeneticOptimization].
self performSimplexOptimization.
```

evaluateIteration

```
| optimizer |
optimizer := DhbHillClimbingOptimizer forOptimizer: self.
optimizer desiredPrecision: desiredPrecision;
          maximumIterations: maximumIterations.
result := optimizer evaluate.
^optimizer precision
```

origin

```
^result
```

origin: `anArrayOrVector`

```
result := anArrayOrVector.
```

performGeneticOptimization

```
| optimizer manager |
optimizer := DhbGeneticOptimizer forOptimizer: functionBlock.
manager := DhbVectorChromosomeManager new: 100 mutation: 0.1
                                                    crossover: 0.1.
manager origin: self origin asVector; range: self range asVector.
optimizer chromosomeManager: manager.
result := optimizer evaluate.
```

performSimplexOptimization

```
| optimizer manager |
optimizer := DhbSimplexOptimizer forOptimizer: self.
optimizer desiredPrecision: desiredPrecision sqrt;
          maximumIterations: maximumIterations;
          initialValue: result asVector.
result := optimizer evaluate.
```

range

```
^self bestPoints
```

range: `anArrayOrVector`

```
bestPoints := anArrayOrVector.
```

Listing 11.24 Java implementation of a general optimizer

```
package DhbOptimizing;

// Multi-strategy optimizer of many-variable functions.

// @author Didier H. Besset

public class MultiVariableGeneralOptimizer extends MultiVariableOptimizer
{

    // Initial range for random search.

    protected double[] range;

// Constructor method.
// @param func DhbInterfaces.ManyVariableFunction
// @param pointCreator DhbOptimizing.OptimizingPointFactory
// @param initialValue double[]

public MultiVariableGeneralOptimizer(DhbInterfaces.ManyVariableFunction
        func, OptimizingPointFactory pointCreator, double[] initialValue)
{
    super(func, pointCreator, initialValue);
}
public double evaluateIteration()
{
    HillClimbingOptimizer finder = new HillClimbingOptimizer( f,
                                                pointFactory, result);
    finder.setDesiredPrecision( getDesiredPrecision());
    finder.setMaximumIterations(  getMaximumIterations());
    finder.evaluate();
    result = finder.getResult();
    return finder.getPrecision();
}
public void initializeIterations()
{
    if ( range != null )
        performGeneticOptimization();
    performSimplexOptimization();
}
```

```
private void performGeneticOptimization()
{
    VectorChromosomeManager manager = new VectorChromosomeManager();
    manager.setRange( range);
    manager.setOrigin( result);
    VectorGeneticOptimizer finder = new VectorGeneticOptimizer(
    f, pointFactory, manager);
    finder.evaluate();
    result = finder.getResult();
}
private void performSimplexOptimization()
{
    SimplexOptimizer finder = new SimplexOptimizer( f, pointFactory,
    result);
    finder.setDesiredPrecision( Math.sqrt( getDesiredPrecision()));
    finder.setMaximumIterations(  getMaximumIterations());
    finder.evaluate();
    result = finder.getResult();
}

// @param x double    component of the origin of the hypercube
//            constraining the domain of definition of the function

public void setOrigin( double[] x)
{
    result = x;
}

// @param x double    components of the lengths of the hypercube
//            constraining the domain of definition of the function

public void setRange( double[] x)
{
    range = x;
}
}
```

CHAPTER 12

Data Mining

Creusez, fouillez, bêchez, ne laissez nulle place
Où la main ne passe et repasse,[1]

—Jean de La Fontaine

Data mining is a catchy buzzword of recent introduction covering activities formerly known as data analysis. The problem is akin to what we already have seen in Chapter 10. In the case of data mining, however, the emphasis is put on large data sets. *Large* must be understood in two ways: first, each *data point* is actually made of a large number of measurements; second, the number of data points is large, even huge. The expression *data mining* was coined when large corporate databases became commonplace. The original reason for the presence of these databases was the day-to-day dealing with the business events. A few people started realizing that these databases contain huge amounts of information, mostly of a statistical nature, about the type of business. That information was waiting to be mined just like the mother lode waited for the coming of the '49ers—hence the term *data mining*.

Figure 12.1 shows the classes described in this chapter. There are two aspects to the data-mining activity: one is preparing the data to render them suitable for the processing; the second consists of extracting the information—that is, the processing of the data. Depending on the problems and on the technique used during the second step, the first step is not always needed. In any case, the preparation of the data is usually very specific to the type of data to be analyzed. Therefore, one can only make very general statements about this problem: one must watch for rounding errors; one must adjust the scale of data having widely different ranges; one must check the statistical validity of the data sample; and so forth. I shall say no more about this first aspect of data mining, but I wanted to warn the reader about this important issue.

Finally, data mining differs from estimation in that, in many cases, the type of information to be extracted is not really known in advance. Data mining tries to

1. Dig, search, excavate, do not leave a place where your hands did not go once or more.

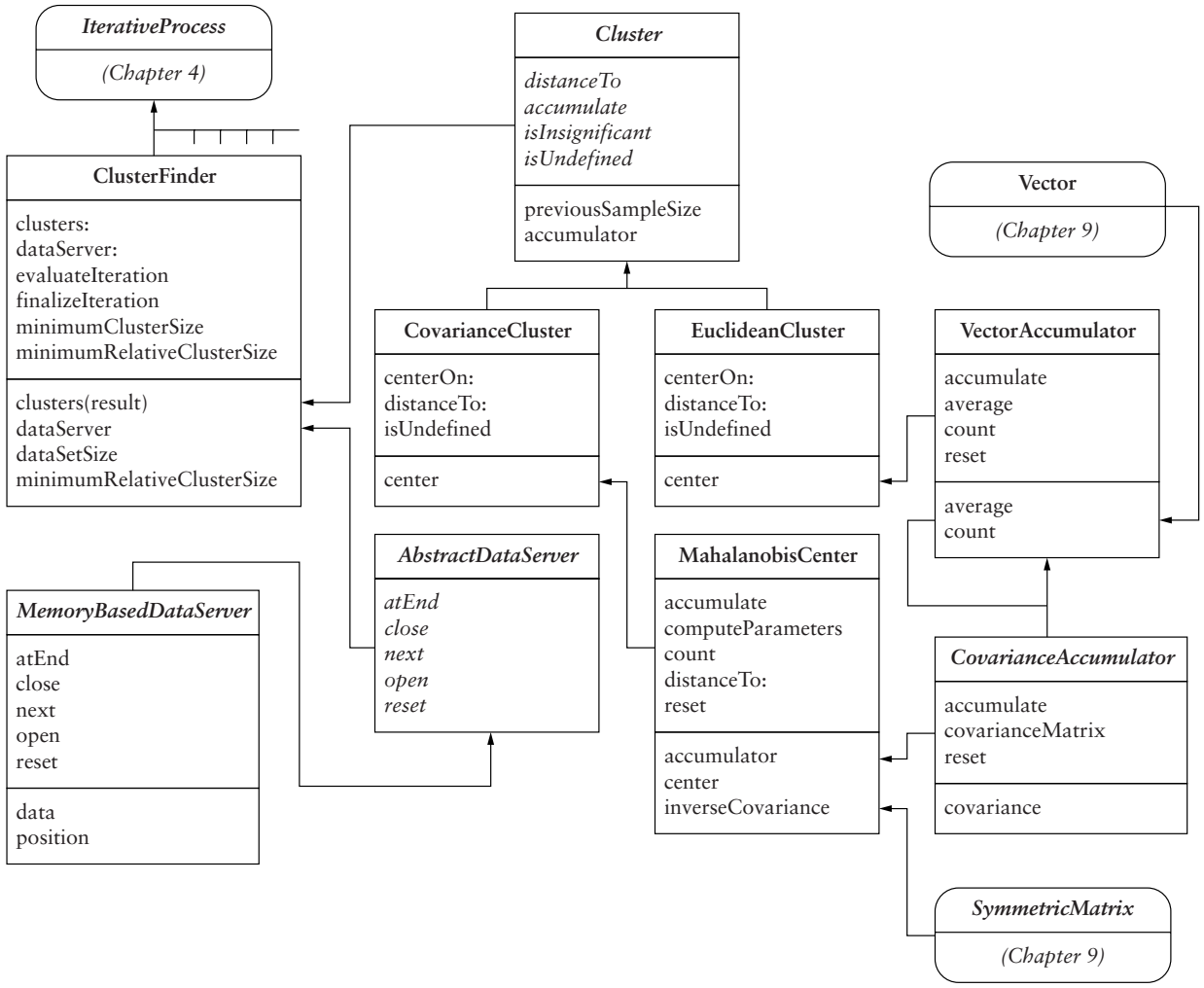

FIG. 12.1 Classes used in data mining

identify trends, common properties, and correlation between the data. The goal of data mining is usually to reduce large samples to much smaller manageable sets, which can be efficiently targeted. One example is the selection of a sample of customers suitable to respond to a new product offered by a bank or an insurance company. Mailing operations are costly; thus, any reduction of the mailing target set with a significant improvement of the probability of response can bring a significant savings. Another example is the scanning for certain types of cancer, which are expensive.[2] If one can identify a population with a high risk of cancer, the scanning only needs to be done for that population, thus keeping the cost low.

A good collection of articles about the techniques exposed in this chapter can be found in [Achtley & Bryant].

12.1 Data Server

As noted in the introduction, data mining means handling large amounts of data, most likely more than the computer memory can hold. Thus, we need an object to handle these data for all objects implementing data mining techniques.
The data server object needs to implement five functionalities:

1. opening the physical support of the data
2. getting the next data item
3. checking whether more data items are available
4. repositioning the data stream at its beginning and
5. closing the physical support of the data

Depending on the problem at hand, the responsibility of a data server can extend beyond the simple task of handling data. In particular, it could be charged wotj performing the data preparation step mentioned in the introduction. In my implementation, I give two classes. One is an abstract class from which all data servers used by the data-mining objects described in this chapter must derive. The data item returned by the method returning the nest item is a vector object whose components are the data corresponding to one measurement. The second class of data server is a concrete class implementing the data server on a collection or an array kept in the computer's memory. Such a server is handy for making tests.

Examples of uses of data servers are given in the other sections; no code examples are given here.

2. In medical domain, *expensive* is not necessarily a matter of money. It can mean that the patient faces a high risk or regards the examination as too intrusive.

12.1.1 Data Server—Smalltalk Implementation

Listing 12.1 shows the implementation of the abstract data server in Smalltalk. The implementation of the concrete class is shown in Listing 12.2. My implementation uses the same methods used by the hierarchy of the class `Stream`.

Listing 12.1 **Smalltalk abstract data server**

Class `DhbAbstractDataServer`
Subclass of `Object`

Instance Methods

atEnd

```
self subclassResponsibility
```

close

next

```
self subclassResponsibility
```

open

```
self subclassResponsibility
```

reset

```
self subclassResponsibility
```

Listing 12.2 **Smalltalk memory based data server**

Class `DhbMemoryBasedDataServer`
Subclass of `DhbAbstractDataServer`
Instance variable names: `data position`

Instance Methods

atEnd

```
^data size < position
```

data:

```
anOrderedCollection
```

```
data := anOrderedCollection.
self reset.
```

dimension

```
^data first size
```

next

```
| answer |
answer := data at: position.
position := position + 1.
^answer
```

open

```
self reset
```

reset

```
position := 1.
```

12.1.2 Data Server—Java Implementation

Listing 12.3 shows the implementation of the abstract data server in Java. The implementation of the concrete class is shown in Listing 12.4. My implementation follows the naming used in the `java.io` package. There is no method checking for the availability of more data; instead, the exception `EOFException` is thrown by the `read` method when reading beyond the end of the stream occurs.

Listing 12.3 Java abstract data server

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbVector;

// Abstract data server for data mining.

// @author Didier H. Besset

public abstract class AbstractDataServer
{

// Constructor method.

public AbstractDataServer() {
    super();
```

```
}

// Closes the stream of data.

public abstract void close();

// Opens the stream of data.

public abstract void open();

// @return DhbVector    next data point found on the stream
// @exception java.io.EOFException when no more data point can be found.

public abstract DhbVector read() throws java.io.EOFException;

// Rewind the stream of data.

public abstract void reset();
}
```

Listing 12.4 Java memory based data server

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbVector;

// Data server containing data in memory for simulation purposes.

// @author Didier H. Besset

public class MemoryBasedDataServer extends AbstractDataServer
{
    private int index;
    private DhbVector[] dataPoints;

// Constructor method (for internal use only)

protected MemoryBasedDataServer()
{
    super();
}

// @param points DhbVector[] supplied data points
// (must not be changed after creation)
```

```
public MemoryBasedDataServer( DhbVector[] points)
{
    dataPoints = points;
}

// Nothing to do

public void close()
{
}

// Nothing to do

public void open()
{
}

// @return DhbMatrixAlgebra.DhbVector    next data point
// @exception java.io.EOFException no more data.

public DhbVector read() throws java.io.EOFException
{
    if( index >= dataPoints.length )
        throw new java.io.EOFException();
    return dataPoints[index++];
}

// Data index is reset

public void reset()
{
    index = 0;
}
}
```

12.2 Covariance and Covariance Matrix

When one deals with two or more random variables, an important question to ask is whether the two variables are dependent on each other.

For example, if one collects the prices of homes and the incomes of the homeowners, one will find that inexpensive homes are mostly owned by low-income families—*mostly* but not *always*. The price of a home is thus *correlated* with the income of the homeowner. As soon as one deals with more than two variables, things stop being clear-cut. Correlations become hard to identify especially because of these

"mostly but not always" cases. Therefore, one must find a way to express mathematically how much two random variables are correlated.

Let $x_1, \ldots, x_m$ be several random variables. They can be considered as the components of an m-dimensional (random) vector $\mathbf{x}$. The probability density function of the vector $\mathbf{x}$ is denoted $P(\mathbf{x})$; it measures the probability of observing a vector within the differential volume element located at $\mathbf{x}$. The average of the vector $\mathbf{x}$ is defined in a way similar to the case of a single random variable. The ith component of the average is defined by

$$\mu_i = \int \ldots \int x_i P(\mathbf{x})\, dx_1 \ldots dx_m. \tag{12.1}$$

The covariance matrix of the random vector $\mathbf{x}$ gives a measure of the correlations between the components of the vector $\mathbf{x}$. The components of the covariance matrix are defined by

$$\varrho_{ij} = \int \ldots \int (x_i - \mu_i)\left(x_j - \mu_j\right) P(\mathbf{x})\, dx_1 \ldots dx_m. \tag{12.2}$$

As one can see, the covariance matrix is a symmetrical matrix. It is also positive definite. Furthermore, ϱ_{ii} is the variance of the ith component of the random vector.

The error matrix of a least-square or maximum-likelihood fit (discussed in Chapter 10) is the covariance matrix of the fit parameters.

If two components are independent, their covariance—that is, the corresponding element of the covariance matrix—is zero. The inverse is not true, however. For example, consider a two-dimensional vector with components $\left(zz^2\right)$ where z is a random variable. If z is distributed according to a symmetrical distribution, the covariance between the two components of the vector is zero. Yet, the components are 100% dependent on each other by construction.

The correlation coefficient between components i and j of the vector $\mathbf{x}$ is then defined by

$$\rho_{ij} = \frac{\varrho_{ij}}{\sigma_i \sigma_j}, \tag{12.3}$$

where $\sigma_i = \sqrt{\varrho_{ii}}$ is the standard deviation of the ith component of the vector $\mathbf{x}$. By definition, the correlation coefficient is comprised between -1 and 1. If the absolute value of a correlation coefficient is close to 1, then one can assert that the two corresponding components are indeed correlated.

If the random vector is determined experimentally, one calculates the estimated covariance with the following statistics:

$$\operatorname{cov}\left(x_i, x_j\right) = \frac{1}{n} \sum_{k=1}^{n} \left(x_{i,k} - \mu_i\right)\left(x_{j,k} - \mu_j\right), \tag{12.4}$$

where $x_{i,k}$ is the ith component of the kth measurement of the vector $\mathbf{x}$. Similarly, the estimated correlation coefficient of the corresponding components is defined as

$$\text{cor}\left(x_i, x_j\right) = \frac{\text{cov}\left(x_i, x_j\right)}{s_i s_j}, \tag{12.5}$$

where s_i is the estimated standard deviation of the ith component.

As for the central moment, there is a way to compute the components of the covariance matrix while they are accumulated. If $\text{cov}_n\left(x_i, x_j\right)$ denotes the estimated covariance over n measurements, one has

$$\text{cov}_{n+1}\left(x_i, x_j\right) = \frac{n}{n+1}\text{cov}_n\left(x_i, x_j\right) + n\Delta_{i,n+1}\Delta_{j,n+1}, \tag{12.6}$$

where $\Delta_{x,n+1}$ and $\Delta_{y,n+1}$ are the corrections to the averages of each variable defined in equation 9.12 of Section 9.2. The derivation of equation 12.6 is given in Appendix 13.2.

12.2.1 Using Covariance Information

A covariance matrix contains statistical information about the set of measurements over which is has been determined. There are several ways of using this information.

The first approach uses the covariance matrix directly. The best example is the analysis known as the *shopping cart analysis* [Berry & Linoff]. For example, one can observe that consumers buying cereals are also buying low-fat milk. This can give useful information on how to target special sales efficiently. Applications working in this mode can use the code of Sections 12.2.3 or 12.2.4 as is.

Another approach is to use the statistical information contained in a covariance matrix to process data coming from measurements that were not used to determine the covariance matrix. In the rest of this chapter, we shall call the set of measurements is used to determine the covariance matrix the *calibrating set*. In this second mode of using a covariance matrix, measurements are *compared* or *evaluated* against those of the calibrating set. It is clear that the quality of the information contained in the covariance matrix depends on the quality of the calibrating set. We shall assume that this is always the case. Techniques working according to this second mode are described in Sections 12.4, 12.5 and 12.7.

12.2.2 Covariance Matrix—General Implementation

The object in charge of computing the covariance matrix of a series of measurements is implemented as for central moments. Because we shall need only to compute the average of a vector, the implementation is spread over two classes, one being the subclass of the other for efficient reuse.
The class `VectorAccumulator` has two instance variables:

`count` counts the number of vectors accumulated in the object so far

`average` keeps the average of the accumulated vector

The class `CovarianceAccumulator` is a subclass of the class `VectorAccumulator`. It has one additional instance variable:

`covariance` accumulates the components of the covariance matrix; for efficiency reasons, only the lower half of the matrix is computed since it is symmetrical.

The topmost class implements equation 9.12 in the method `accumulate`. The subclass overloads this method to implement equation 12.6.

12.2.3 Covariance Matrix—Smalltalk Implementation

Listing 12.5 describes the implementation of the accumulation of a vector in Smalltalk. Listing 12.6 shows the implementation of the accumulation of the covariance matrix. Code Example 12.1 shows how to accumulate the average of a series of vectors read from a data stream.

Code Example 12.1

```
| accumulator valueStream average |
accumulator := DhbVectorAccumulator new.
valueStream open.
[ valueStream atEnd]
       whileFalse:[ accumulator accumulate: valueStream next].
valueStream close.
average := accumulator average.
```

This example is totally equivalent to Code Example 9.1. Here the method `next` of the data stream must return a vector instead of a number; all vectors must have the same dimension. The returned average is a vector of the same dimension.

Code Example 12.2 shows how to accumulate both the average and covariance matrix. The little differences from the preceding example should be self-explanatory.

Code Example 12.2

```
| accumulator valueStream average covarianceMatrix |
accumulator := DhbCovarianceAccumulator new.
valueStream open.
[ valueStream atEnd]
       whileFalse:[ accumulator accumulate: valueStream next].
valueStream close.
average := accumulator average.
covarianceMatrix := accumulator covarianceMatrix.
```

The method `accumulate` of class `DhbVectorAccumulator` answers the corrections to each component of the average vector. This allows the class `DhbCovarianceAccumulator` to reuse the results of this method. In class `DhbVectorAccumulator`, vector operations are used. The method `accumulate` of class `DhbCovarianceAccumulator` works with indices because one only computes the lower half of the matrix.

Listing 12.5 Smalltalk implementation of vector average

Class DhbVectorAccumulator
Subclass of Object
Instance variable names: count average

Class Methods

new:

```
anInteger

^self new initialize: anInteger
```

Instance Methods

accumulate: aVectorOrArray

```
| delta |
count := count + 1.
delta := average - aVectorOrArray asVector scaleBy: 1 / count.
average accumulateNegated: delta.
^delta
```

average

```
^average
```

count

```
^count
```

initialize:

```
anInteger

average := DhbVector new: anInteger.
self reset.
^self
```

printOn:

```
aStream

super printOn: aStream.
aStream space.
count printOn: aStream.
aStream space.
average printOn: aStream.
```

reset

```
count := 0.
average atAllPut: 0.
```

Listing 12.6 Smalltalk implementation of covariance matrix

Class	DhbCovarianceAccumulator
Subclass of	DhbVectorAccumulator
Instance variable names:	covariance

Instance Methods

accumulate:

anArray

```
| delta count1 r|
count1 := count.
delta := super accumulate: anArray.
r := count1 / count.
1 to: delta size
    do: [ :n |
          1 to: n do:
              [ :m |
                ( covariance at: n) at: m put: ( count1 * ( delta
   at: n) * ( delta at: m) + ( r * ( ( covariance at: n) at: m))).
              ].
        ].
```

covarianceMatrix

```
| rows n |
n := 0.
rows := covariance collect:
            [ :row | n := n + 1. row, ( ( ( n + 1) to: covariance
              size) collect: [ :m | ( covariance at: m) at: n ])].
^DhbSymmetricMatrix rows: rows
```

initialize:

anInteger

```
covariance := ( ( 1 to: anInteger) collect: [ :n | DhbVector new:
                                                    n]) asVector.
^super initialize: anInteger
```

reset

```
super reset.
covariance do: [ :each | each atAllPut: 0].
```

12.2.4 Covariance Matrix—Java Implementation

Listing 12.5 shows the implementation of the accumulation of a vector in Smalltalk. Listing 12.6 shows the implementation of the accumulation of the covariance matrix. Code Example 12.3 shows how to accumulate the average of a series of vectors read from a data stream.

Code Example 12.3

<*Creating an instance of a concrete subclass of* `AbstractDataServer` *into the variable* `dataServer`>

```
VectorAccumulator accumulator = new VectorAccumulator(dimension);
dataServer.open();
try {
    while (true) accumulator.accumulate( server.read());
    } catch ( java.io.EOFException e) {};
DhbVector averages = accumulator.averageVector();
```

This example is very similar to Code Example 9.2. Here the data to accumulate are prepared as an array of vectors. Code Example 12.4 shows how to accumulate both the average and the covariance matrix. The little differences with the preceding example should be self-explanatory.

Code Example 12.4

```
VectorAccumulator accumulator = new VectorAccumulator(dimension);
dataServer.open();
 try {
   while (true) accumulator.accumulate( server.read());
   } catch ( java.io.EOFException e) {};
DhbVector averages = accumulator.averageVector();
SymmetricMatrix covariance = accumulator.covarianceMatrix();
```

In my implementation, the method `accumulate` has been implemented without any exception. This departure from the recommended practice is justified by the fact that the handling of exceptions—even when they do not occur—increases the execution of a method. To optimize the loop, the computation of the average is duplicated in the method `accumulate` of each class.

Listing 12.7 Java implementation of vector average

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbVector;

// Statistical average for vectors

// @author Didier H. Besset

public class VectorAccumulator
{
    protected long count = 0;
    protected double[] average;

// Default constructor method.

public VectorAccumulator(int n)
{
    this(n,1);
    reset();
}

// Constructor method.
// @param n int
// @param dummy int

protected VectorAccumulator(int n, int dummy)
{
    average = new double[n];
}

// @param v double[]    values to accumulate in the receiver

public void accumulate( double[] v)
{
    count += 1;
    for ( int i = 0; i < average.length; i++ )
        average[i] -= (average[i] - v[i]) / count;
}

// @param v DhbVector    vector of values to accumulate in the
receiver

public void accumulate( DhbVector v)
{
    accumulate( v.toComponents());
```

```
}

// @return DhbVector    vector containing the average

public DhbVector averageVector()
{
    try { return new DhbVector( average);}
        catch( NegativeArraySizeException e){ return null;}
}

// @return long    number of accumulated data points.

public long getCount()
{
    return count;
}

// @param sb java.lang.StringBuffer

protected void printOn( StringBuffer sb)
{
    sb.append("Counts: "+count);
    char separator = '\n';
    for ( int i = 0; i < average.length; i++)
    {
        sb.append( separator);
        sb.append( average[i]);
        separator = ' ';
    }
}
public void reset()
{
    count = 0;
    for ( int i = 0; i < average.length; i++ )
        average[i] = 0;
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    printOn( sb);
    return sb.toString();
}
}
```

Listing 12.8 Java implementation of covariance matrix

```
package DhbDataMining;

import DhbMatrixAlgebra.SymmetricMatrix;
import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbMatrixAlgebra.DhbNonSymmetricComponents;

// Statistical average and covariance for vectors

// @author Didier H. Besset

public class CovarianceAccumulator extends VectorAccumulator
{
    private double[][] covariance;

// Constructor method
// @param n int

public CovarianceAccumulator(int n)
{
    super(n,1);
    covariance = new double[n][n];
    reset();
}

// @param v DhbVector    vector to accumulate in the receiver

public void accumulate( double[] v)
{
    long n = count;
    count += 1;
    double[] deltas = new double[average.length];
    int j;
    double r = (double) n / (double) count;
    for ( int i = 0; i < average.length; i++ )
    {
        deltas[i] = ( average[i] - v[i]) / count;
        average[i] -= deltas[i];
        for ( j = 0; j <= i; j++ )
            covariance[i][j] = r * covariance[i][j] + n * deltas[i]
                * deltas[j];
    }
}

// @return double
```

```
// @param n int
// @param m int

public double correlationCoefficient( int n, int m)
{
    return covariance[n][m] / Math.sqrt(covariance[n][n]
       * covariance[m][m]);
}

 * @return SymmetricMatrix    covariance matrix

public SymmetricMatrix covarianceMatrix()
{
    double[][] components = new double[average.length][average.length];
    int j;
    for ( int i = 0; i < average.length; i++ )
    {
        for ( j = 0; j <= i; j++)
        {
            components[i][j] = covariance[i][j];
            components[j][i] = components[i][j];
        }
    }
    try { return SymmetricMatrix.fromComponents(components);}
        catch (DhbNonSymmetricComponents e) { return null;}
        catch (DhbIllegalDimension e) { return null;}
}

// @param sb java.lang.StringBuffer

protected void printOn( StringBuffer sb)
{
    super.printOn( sb);
    for ( int i = 0; i < average.length; i++)
    {
        char separator = '\n';
        for ( int j = 0; j <= i; j++)
        {
            sb.append( separator);
            sb.append( covariance[i][j]);
            separator = ' ';
        }
    }
}
public void reset()
{
```

```
        super.reset();
        int j;
        for ( int i = 0; i < average.length; i++ )
        {
            for ( j = 0; j <= i; j++ )
                covariance[i][j] = 0;
        }
    }

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

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

    // @return double
    // @param n int

    public double variance( int n)
    {
        return covariance[n][n] * count / (count - 1);
    }
    }
```

12.3 Multidimensional Probability Distribution

To get a feeling of what the covariance matrix represents, let us now consider a vector $\mathbf{y}$ whose components are independent random variables distributed according to a normal distribution. The probability density function is given by

$$P(\mathbf{y}) = \prod_{i=1}^{m} \frac{1}{\sqrt{2\pi\sigma_i^2}} e^{-\frac{(y_i-\mu_i)^2}{2\sigma_i^2}}. \tag{12.7}$$

In this case, the covariance matrix of the vector $\mathbf{y}$ is a diagonal matrix $\tilde{\mathbf{V}}$, whose diagonal elements are the variance of the vector's components. If $\tilde{\mathbf{C}}$ is the inverse of the matrix $\tilde{\mathbf{V}}$, equation 12.7 can be rewritten as

$$P(\mathbf{y}) = \sqrt{\frac{\det \tilde{\mathbf{C}}}{(2\pi)^m}} e^{-\frac{1}{2}(\mathbf{y}-\bar{\mathbf{y}})^{\mathrm{T}}\tilde{\mathbf{C}}(\mathbf{y}-\bar{\mathbf{y}})}, \tag{12.8}$$

where $\bar{\mathbf{y}}$ is the vector whose components are $\mu_1, \ldots, \mu_m$. Let us now consider a change of coordinates $\mathbf{x} = \mathbf{O}\mathbf{y}$, where $\mathbf{O}$ is an orthogonal matrix. We have already met such transformations in Section 8.7. Because the matrix is orthogonal, the differential volume element is invariant under the change of coordinates. Thus, the probability density function of the vector $\mathbf{x}$ is

$$P(\mathbf{x}) = \sqrt{\frac{\det \mathbf{C}}{(2\pi)^m}} e^{-\frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}})^{\mathrm{T}}\mathbf{C}(\mathbf{x}-\bar{\mathbf{x}})}, \tag{12.9}$$

where the matrix $\mathbf{C}$ is equal to $\mathbf{O}^{\mathrm{T}}\tilde{\mathbf{C}}\mathbf{O}$. The vector $\bar{\mathbf{x}}$ is simply[3] equal to $\mathbf{O}\bar{\mathbf{y}}$. The covariance matrix, $\mathbf{V}$, of the vector $\mathbf{x}$ is then equal to $\mathbf{O}^{\mathrm{T}}\tilde{\mathbf{V}}\mathbf{O}$. It is also the inverse of the matrix $\mathbf{C}$. Thus, equation 12.9 can be rewritten as

$$P(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^m \det \mathbf{V}}} e^{-\frac{1}{2}(\mathbf{x}-\bar{\mathbf{x}})^{\mathrm{T}}\mathbf{V}^{-1}(\mathbf{x}-\bar{\mathbf{x}})}. \tag{12.10}$$

In the case of a normal distribution in a multidimensional space, the covariance matrix plays the same role as the variance in one dimension.

12.4 Covariance Data Reduction

Reversing the derivation of the preceding section, one can see that the eigenvalues of the covariance matrix of the vector $\mathbf{x}$ correspond to the variances of a series of independent (not necessarily normally distributed) random variables $y_1, \ldots, y_m$. Since the covariance matrix is symmetrical these eigenvalues as well as the matrix $\mathbf{O}$ describing the change of coordinates can be obtained using Jacobi's algorithm described in Section 8.7.

If some eigenvalues are much larger than the others, one can state that the information brought by the corresponding variables brings little information to the problem. Thus, one can omit the corresponding variable from the rest of the analysis.

Let $\sigma_1^2, \ldots, \sigma_m^2$ be the eigenvalues of the covariance matrix such that $\sigma_i^2 < \sigma_j^2$ for $i < j$. Let us assume that an index exists k such that $\sigma_k^2 \gg \sigma_{k-1}^2$. Then, The rest of the data analysis can be made with a vector with components $y_k, \ldots, y_m$ where the vector $\mathbf{y}$ is defined by $\mathbf{y} = \mathbf{O}^{\mathrm{T}}\mathbf{x}$.

This reduction technique has been used successfully in high-energy physics, under the name principal component analysis. The data reduction allows to extracting the relevant parameters of a complex particle detector to facilitate the quick extraction of the physical data—momentum and energy of the particle—from the observed data.

This kind of data reduction can be implemented within the framework of the data server described in Section 12.1. The concrete implementation of such a server

3. All this is a consequence of the linear property of the expectation value operator.

is straightforward. All needed objects—covariance accumulation, eigenvalues of a symmetrical matrix, vector manipulation—have been discussed in different chapters. The rest of the implementation is specific to the problem at hand and can therefore not be treated on a general basis.

12.5 Mahalanobis Distance

Mahalanobis, an Indian statistician, introduced this distance in the 30's when working on anthropometric statistics. A paper by Mahalanobis himself can be found in [Achtley & Bryant]. Readers interested by the historical dimension of the Mahalanobis distance can consult the paper by Das Gupta[4].

By definition, the exponent of equation 12.7 is distributed according to a χ^2 distribution with m degrees of freedom. This exponent remains invariant under the change of coordinates discussed in Section 12.3. Thus, the exponent of equation 12.10 is also distributed according to a χ^2 distribution with m degrees of freedom, where m is the dimension of the vector x. The Mahalanobis distance, d_{M}, is defined as the square root of the exponent, up to a factor $\sqrt{2}$. We have

$$d_M^2 = (\mathbf{x} - \bar{\mathbf{x}})^{\mathrm{T}} \, \mathbf{V}^{-1} \, (\mathbf{x} - \bar{\mathbf{x}}) \, . \tag{12.11}$$

The Mahalanobis distance is a distance using the inverse of the covariance matrix as the metric. It is a distance in the geometrical sense because the covariance matrix as well as its inverse are positive definite matrices. The metric defined by the covariance matrix provides a normalization of the data relative to their spread.

The Mahalanobis distance—or its square—can be used to measure how *close* an object is from another when these objects can be characterized by a series of numerical measurements. Using the Mahalanobis distance is done as follows

1. The covariance matrix of the measured quantities, $\mathbf{V}$, is determined over a calibrating set
2. One compute the inverse of the covariance matrix, $\mathbf{V}^{-1}$
3. The distance of a new object to the calibrating set is estimated using equation 12.11; if the distance is smaller than a given threshold value, the new object is considered as belonging to the same set

One interesting property of the Mahalanobis distance is that it is normalized. Thus, it is not necessary to normalize the data provided rounding errors in inverting the covariance matrix are kept under control. If the data are roughly distributed according to a normal distribution, the threshold for accepting whether an object belongs to the calibrating set can be determined from the χ^2 distribution.

4. Somesh Das Gupta, *The evolution of the D^2-statistics of Mahalanobis*, Indian J. Pure Appl. Math., **26**(1995), no. 6, 485-501.

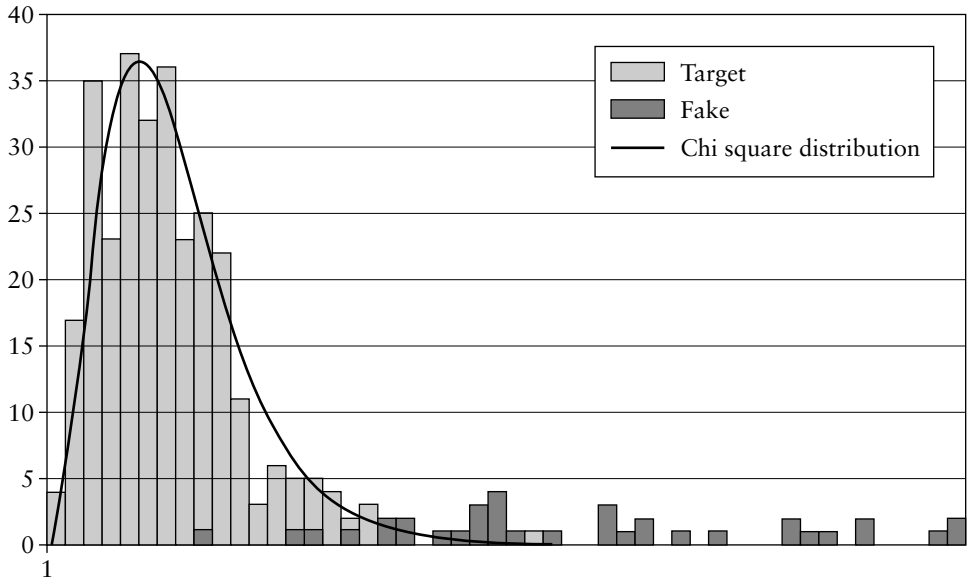

FIG. 12.2 Using the Mahalanobis distance to differentiate between good and fake coins.

12.5.1 Examples of Use

The Mahalanobis distance can be applied in all problems in which measurements must be classified.

A good example is the detection of coins in a vending machine. When a coin is inserted into the machine, a series of sensors gives several measurements, between a handful and a dozen. The detector can be calibrated using a set of good coins forming a calibration set. The coin detector can differentiate good coins from the fake coins using the Mahalanobis distance computed on the covariance matrix of the calibration set. Figure 12.2 shows an example of such data[5]. The light gray histogram is the distribution of the Mahalanobis distance of the good coins; the dark gray histogram that of the fake coins. The dotted line is the χ^2 distribution corresponding to the degrees of freedom of the problem; in other words, the distribution was not fitted. The reader can see that the curve of the χ^2 distribution reproduces the distribution of the experimental data. Figure 12.2 also shows the power of separation achieved between good and fake coins.

Another field of application is the determination of cancer cells from a biopsy. Parameters of cells—size and darkness of nucleus, granulation of the membrane—can be measured automatically and expressed in numbers. The covariance matrix

5. Data are reproduced with permission; the owner of the data wants to remain anonymous, however.

can be determined using either measurements of healthy cells or measurements of malignant cells. Identification of cancerous cells can be automatized using the Mahalanobis distance.

12.5.2 Mahalanobis Distance—General Implementation

The final goal of the object implementing the Mahalanobis distance is to compute the square Mahalanobis distance as defined in equation 12.11. This is achieved with the method `distanceTo`. The inverse of the covariance matrix as well as the average vector $\bar{\mathbf{x}}$ are contained within the object. I call the object a *Mahalanobis center* since it describes the center of the calibrating set.

To have a self-contained object, the Mahalanobis center is acting as a FACADE to the covariance accumulator of Section 12.2 for the accumulation of measurements. The methods `accumulate` and `reset` are delegated to an instance variable holding a covariance accumulator.
The Mahalanobis center has the following variables:

`accumulator` A covariance accumulator as described in Section 12.2;

`center` The vector $\bar{\mathbf{x}}$

`inverseCovariance` The inverse of the covariance matrix, (i.e., $\mathbf{V}^{-1}$)

My implementation is dictated by its future reuse in cluster analysis (see Section 12.7). There, we need to be able to accumulate measurements while using the result of a preceding accumulation. Thus, computation of the center and the inverse covariance matrix must be done explicitly with the method `computeParameters`.

There are two ways of creating a new instance. One is to specify the dimension of the vectors that will be accumulated into the object. The second supplies a vector as the tentative center. This mode is explained in Section 12.7.

12.5.3 Mahalanobis Distance—Smalltalk Implementation

Listing 12.9 shows the implementation of a Mahalanobis center in Smalltalk. Code Example 12.5 shows how to sort measurements using the Mahalanobis distance.

Code Example 12.5

```
| center calibrationServer dataServer data threshold|
center := DhbMahalanobisCenter new: 5.
<The variable calibrationServer is set up to read measurements from the cali-
brating set>

 calibrationServer open.
 [ calibrationServer atEnd]
      whileFalse: [ center accumulate: calibrationServer next].
 calibrationServer close.
 center computeParameters.
```

```
<The variable dataServer is set up to read the measurements to be
sorted between accepted and rejected; the variable threshold
must also be determined or given>

dataServer open.
[ dataServer atEnd]
     whileFalse: [ data := dataServer next.
                   ( center distanceTo: data) > threshold
                       ifTrue: [ self reject: data]
                       ifFalse:[ self accept: data].
                 ].
dataServer close.
```

The first line after the declaration creates a new instance of a Mahalanobis center for vectors of dimension 5. After setting up the server for the calibrating set, data from the calibrating set are accumulated into the Mahalanobis center. At the end the parameters of the Mahalanobis center are computed. Then, the server for the other measurements is set up. The loop calculates the distance to the Mahalanobis center for each measurement. The example supposes that the object executing this code has implemented two methods `accept` and `reject` to process accepted and rejected data.

Listing 12.9 Smalltalk Mahalanobis center

Class	`DhbMahalanobisCenter`
Subclass of	`Object`
Instance variable names:	`center inverseCovariance accumulator`

Class Methods

new:

```
anInteger

^self new initialize: anInteger
```

onVector:

```
aVector

^self new center: aVector
```

Instance Methods

accumulate:

```
aVector
```

```
accumulator accumulate: aVector.
```

center:

```
aVector
```

```
accumulator := DhbCovarianceAccumulator new: aVector size.
center := aVector.
inverseCovariance := DhbSymmetricMatrix identity: aVector size.
^self
```

computeParameters

```
center := accumulator average copy.
inverseCovariance := accumulator covarianceMatrix inverse.
```

count

```
^accumulator count
```

distanceTo:

```
aVector
```

```
| delta |
delta := aVector - center.
^delta * inverseCovariance * delta
```

initialize:

```
anInteger
```

```
accumulator := DhbCovarianceAccumulator new: anInteger.
^self
```

printOn:

```
aStream
```

```
accumulator count printOn: aStream.
aStream nextPutAll: ': '.
center printOn: aStream.
```

reset

```
accumulator reset.
```

12.5.4 Mahalanobis Distance—Java Implementation

Listing 12.9 shows the implementation of a Mahalanobis center in Smalltalk. Code Example 12.6 shows how to sort measurements using the Mahalanobis distance.

Code Example 12.6

```
MahalanobisCenter center = new MahalanobisCenter( 5);

<The variable calibrationServer is set up to read measurements
from the calibrating set>

 calibrationServer.open();
 try{ while( true)
        { center.accumulate( calibrationServer.next());};}
    catch( EOFException e){};
 calibrationServer.close();
 center.computeParameters();

<The variable dataServer is set up to read the measurements to be
sorted between accepted and rejected; the double variable
threshold
must also be determined or given>

 dataServer.open();
 try {
      while (true)
        {
            DhbVector data = dataServer.next();
            if ( center.distanceTo( data) > threshold )
                reject(data);
            else
                accept(data);
        };}
    catch( EOFException e){};
 dataServer.close();
```

The first line creates a new instance of a Mahalanobis center for vectors of dimension 5. After setting up the server for the calibrating set, data from the calibrating set are accumulated into the Mahalanobis center. At the end the parameters of the Mahalanobis center are computed. Then, the server for the other measurements is set up. The loop calculates the distance to the Mahalanobis center for each measurement. The example supposes that the object executing this code has implemented two methods `accept` and `reject` to process accepted and rejected data.

Listing 12.10 Java Mahalanobis center

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbIllegalDimension;
import DhbMatrixAlgebra.DhbNonSymmetricComponents;
import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.SymmetricMatrix;

// This object is used to compute the Mahalanobis distance
// to a set of data.

// @author Didier H. Besset

public class MahalanobisCenter
{
    private DhbVector center = null;
    private SymmetricMatrix inverseCovariance = null;
    private CovarianceAccumulator accumulator;

// Constructor method.
// @param int dimension of the receiver

public MahalanobisCenter( int dimension)
{
    accumulator = new CovarianceAccumulator( dimension);
}

// Constructor method.
// @param DhbVector center of the receiver

public MahalanobisCenter( DhbVector v)
{
    accumulator = new CovarianceAccumulator( v.dimension());
    center = v;
    inverseCovariance = SymmetricMatrix.identityMatrix(v.dimension());
}

// Accumulation is delegated to the covariance accumulator.
// @param v DhbVector    vector of values to accumulate in the receiver

public void accumulate( DhbVector v)
{
    accumulator.accumulate(v);
}
```

```
// Computes the parameters of the receiver.

public void computeParameters()
{
    center = accumulator.averageVector();
    inverseCovariance = (SymmetricMatrix)
                            accumulator.covarianceMatrix().inverse();
}

// @return double    Mahalanobis distance of the data point from the
//                                center of the receiver.
// @param dataPoint DhbVector    data point

public double distanceTo(DhbVector dataPoint)
{
    try {
        DhbVector v = dataPoint.subtract( center);
        return v.product( inverseCovariance.product(v));
        }
    catch (DhbIllegalDimension e) { return Double.NaN;}
}

// @return long    number of data points inside the receiver

public long getCount()
{
    return accumulator.getCount();
}

// Keep the center and covariance matrix.

public void reset()
{
    accumulator.reset();
}

// @return java.lang.String

public String toString() {
    return center.toString();
}
}
```

12.6 Cluster Analysis

Cluster analysis, also known as K-*cluster*, is a method to identify similarities between data. If the dimension of the data is less than or equal to 3, graphical data representation provides an easy way to identify data points having some similarity. In more than three dimensions, the human brain is unable to identify clustering clearly.

Cluster analysis has been used successfully by the U.S. army to define a new classification of uniform sizes and in banks [Berry & Linoff]. British Telecom used cluster analysis to detect a large-scale phone fraud in the early 1990's.[6]
The *K*-cluster algorithm goes as follows [Berry & Linoff]:

1. Pick up a set of centers where possible clusters may exist.
2. Place each data point into a cluster corresponding to the nearest center.
3. When all data points have been processed, compute the center of each cluster.
4. If the centers have changed, go back to step 2.

This algorithm nicely maps itself to the framework of successive approximations discussed in Section 4.1. We now will investigate the steps of the algorithm in details.

12.6.1 Algorithm Details

Picking up a set of centers corresponds to the box labeled *Compute or choose initial object* of Figure 4.3. Since one is looking for unknown structure in the data, there is little chance to make a good guess on the starting values. The most general approach is to pick up a few points at random from the existing data points to be the initial cluster's centers.

The next two steps correspond to the box labeled *Compute next object* of Figure 4.3. Here lies the gist of the *K*-cluster algorithm. For each data point, one first finds the cluster whose center is the nearest to the data point. What is the meaning of *near*? It depends on the problem. Let us just say at this point that one needs to have a way of expressing the distance between two data points. For the algorithm to converge, the distance must be a distance in the geometrical sense of the term. In geometry, a distance is a numerical function of two vectors, $d(\mathbf{x}, \mathbf{y})$. For all vectors $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$, the following conditions must be met by the function:

$$d(\mathbf{x}, \mathbf{y}) \geq 0,$$
$$d(\mathbf{x}, \mathbf{y}) = d(\mathbf{y}, \mathbf{x}),$$
$$d(\mathbf{x}, \mathbf{y}) \leq d(\mathbf{x}, \mathbf{z}) + d(\mathbf{z}, \mathbf{y}).$$

Furthermore, $d(\mathbf{x}, \mathbf{y}) = 0$ if and only if $\mathbf{x} = \mathbf{y}$. The simplest known distance function is the Euclidean distance expressed as

6. Private communication to the author.

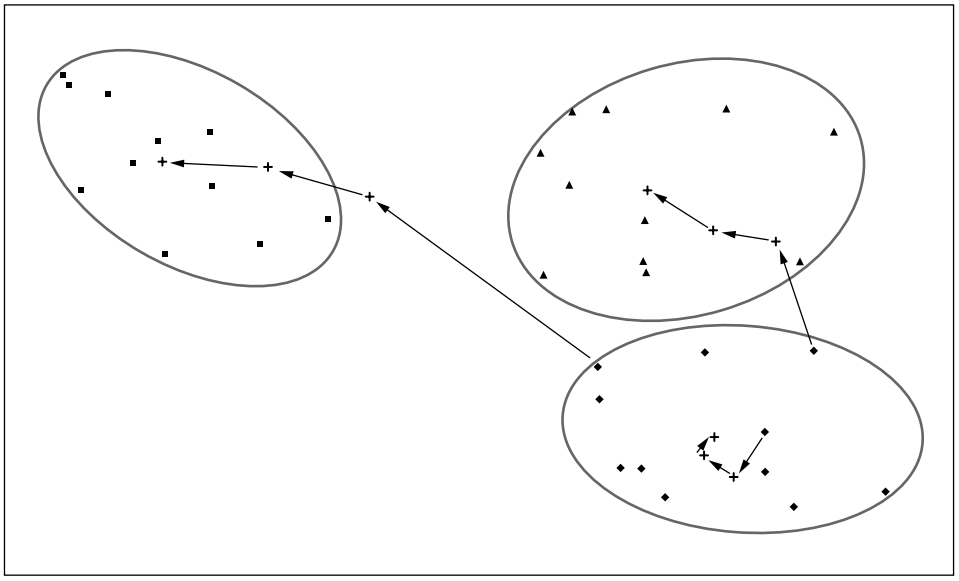

FIG. 12.3 Example of cluster algorithm

$$d_{\mathrm{E}}(\mathbf{x}, \mathbf{y}) = \sqrt{(\mathbf{x} - \mathbf{y}) \cdot (\mathbf{x} - \mathbf{y})}. \tag{12.12}$$

The square root in equation 12.12 is required for the distance function to behave linearly in a one-dimensional subspace. The Euclidean distance corresponds to the notion of distance in everyday life. In assessing the proximity of two points, the square root is not needed.

After all points have been processed and assigned to a cluster, the center of each cluster is obtained by taking the vector average of all data points belonging to that cluster.

Then, one needs to determine whether the clusters have changed since the last iteration. In the case of the *K*-cluster algorithm, it is sufficient to count the number of data points changing clusters at each iteration. When the same data point are assigned to the same clusters at the end of an iteration, the algorithm is completed. The *precision* used to stop the iterative process is an integer in this case.

Figure 12.3 shows how the algorithm works for a two-dimensional case. Data points were generated randomly centered around three separated clusters. Three random points were selected as starting points: in this example the random choice was particularly unlucky since the three starting points all belong to the same cluster. Nevertheless, convergence was attained after five iterations.[7]

7. The last iteration is not visible since the centers did not change.

Fine points

This simple example is admittedly not representative. However, this is not because of the small dimension or because of the well-separated clusters. It is because we knew in advance the number of clusters—three, in this case. If we had started with five initial clusters, we would have ended up with five clusters, according to the expression, "garbage in, garbage out," well known among programmers.

One can modify the original algorithm to prune insignificant clusters from the search. This additional step must be added between steps 2 and 3. How to characterize an insignificant cluster? This often depends on the problem at hand. One thing for sure is that clusters with zero elements should be left out. Thus, one easy method to prune insignificant clusters is to place a limit of the number of data points contained in each cluster. That limit should not be too high, however, or most clusters will never get a chance of accumulating a significant amount of data points during the first few iterations.

12.6.2 Cluster Analysis—General Implementation

Our implementation of the *K*-cluster algorithm uses two classes: `ClusterFinder` and `Cluster`.

The class `Cluster` describes the behavior of a cluster. It is an abstract class. Subclasses implement the various strategies needed by the algorithm: distance calculation and pruning of insignificant clusters. The abstract class has only one instance variable, `previousSampleSize`, keeping the count of data points accumulated in the previous iteration. A cluster must be able to return the number of data points contained in the cluster. The method `changes` gives the number of data points that have changed cluster since the last iteration. The method `isInsignificantIn` determines whether a cluster must be pruned from the process. The method `isUndefined` allows the identification of clusters whose centers have not yet been defined. This method is used by the cluster finder to initialize undefined clusters with the data points before starting the iterative process.

An instance of a subclass of `cluster` must implement the following methods to interact with the class `ClusterFinder`:

`distanceTo` The argument of this method is a vector representing the data point; the returned value is the distance between the supplied vector and the center of the cluster; any subclass of `Cluster` can implement a suitable distance function as well as its own representation of the center.

`accumulate` The argument of this method is a vector representing the data point; this method is called when the cluster finder has determined that the data point must be placed within this cluster.

`changes` This method returns the number of data points that have been added to and removed from the cluster since the last iteration; the default implementation

only calculates the difference between the number currently in the cluster and the number in the previous iteration; this simple approach works in practice[8];

`sampleSize` This method returns the number of data points actually accumulated into the cluster.

`reset` This method calculates the position of the new center at the end of an iteration; the default implementation stores the size of the previous sample.

A reader knowledgeable of patterns will think of using a STRATEGY pattern to implement the distance. I have tried this approach, but the resulting classes were much more complex for little gain. It is easier to implement subclasses of `Cluster`. Each subclass not only can implement the distance function but can also choose how to accumulate data points: accumulation or storage.

The concrete class `EuclideanCluster` calculates the square of the Euclidean distance as defined in equation 12.12. Using Euclidean distance requires that the components of the data vector have comparable scales. Cluster analysis with Euclidean clusters requires data preparation in general.

The class `ClusterFinder` implements the algorithm itself. This class is a subclass of the class for iterative processes described in Section 4.1. The result of the iterative process is an array of clusters. The class `ClusterFinder` needs the following instance variables:

`dataServer` A data server object as described in Section 12.1; this object is used to iterate on the data points.

`dataSetSize` A counter to keep track of the number of data points; this instance variable is combined with the next to provide a first-order pruning strategy.

`minimumRelativeClusterSize` The minimum relative size of a significant cluster; the minimum cluster size is computed at each iteration by taking the product of this variable with the variable `dataSetSize`.

The class `ClusterFinder` uses an instance of the data server to iterate over the data points. Before starting the search, the client application can assign the list of initial clusters (step 1 of the algorithm). By default, the minimum relative size of the cluster is set to zero. A convenience method allows creating instances for a given data server and an initial set of clusters.

The method `initializeIterations` scans all clusters and looks for undefined clusters. The center of each undefined cluster is assigned to a data point read from the data server. This assumes that the data points have been collected randomly.

The method `evaluateIteration` processes each data point: first, it finds the index of the cluster nearest to the data point; then the data point is accumulated

8. Of course, one can construct cases where this simple approach fails. Such cases, however, correspond to situations where data points are oscillating between clusters and therefore do not converge. I have never met such cases in practice.

into that cluster. After processing the data points, the clusters are processed to compute the new position of their centers and insignificant clusters are removed from the search. These tasks are performed within a method named `collectChangesAndResetClusters`. This method returns the number of data points that have changed cluster. The determination of whether a cluster is significant is delegated to each cluster with the method `isInsignificant`. Thus, any subclass of cluster can implement its own strategy. The argument of the method `isInsignificant` is the cluster finder to provide each cluster with global information if needed.

The method `finalizeIterations` closes the data server.

12.6.3 Cluster Analysis—Smalltalk Implementation

Listing 12.11 shows the implementation of the *K*-cluster algorithm in Smalltalk. Code Example 12.7 shows how to implement a search for clusters.

Code Example 12.7

```
| dataServer finder clusters|

<The variable dataServer is set up to read measurements from the calibrating set>

finder := DhbClusterFinder new: 5 server: dataServer

      type: <a concrete subclass of Cluster >.

finder minimumRelativeClusterSize: 0.1.
clusters := finder evaluate.
```

After setting up a data server to read the data point, an instance of class `DhbClusterFinder` is created. The number of desired clusters is set to five. The class of the clusters is specified. The next line sets the minimum relative size of each cluster to be kept during the iteration. Finally, the *K*-cluster algorithm is performed, and clusters are retrieved from the finder object.

The abstract class `DhbCluster` is implemented with an instance variable `accumulator`. The cluster delegates the responsibility of accumulating the data points to this variable. It is assumed that the object in `accumulator` implements the interface defined by the vector accumulators described in Section 12.2.3.

The class `DhbClusterFinder` can be created in two ways. An application can set the list of initial clusters and the data server using the methods `cluster` and `dataServer`, respectively. The convenience class creation method `new:server:type:` allows specifying the initial number of clusters, the data server, and the class of the clusters. When this method is used, the collection of clusters is created when the instance of `DhbClusterFinder` is initialized; each cluster is created in an undefined state.

Listing 12.11 Smalltalk *K*-cluster algorithm

Class DhbCluster
Subclass of Object
Instance variable names: accumulator previousSampleSize

Class Methods

new

```
^super new initialize
```

Instance Methods

accumulate: aVector

```
accumulator accumulate: aVector.
```

centerOn:

```
aVector
self subclassResponsibility
```

changes

```
^(self sampleSize - previousSampleSize) abs
```

distanceTo:

```
aVector
^self subclassResponsibility
```

initialize

```
previousSampleSize := 0.
^self
```

isInsignificantIn:

```
aClusterFinder
^self sampleSize <= aClusterFinder minimumClusterSize
```

isUndefined

```
^self subclassResponsibility
```

reset

```
previousSampleSize := self sampleSize.
self collectAccumulatorResults.
accumulator reset
```

sampleSize

```
^accumulator count
```

Class DhbClusterFinder

Subclass of DhbIterativeProcess

Instance variable names: dataServer dataSetSize minimumRelativeClusterSize

Class Methods

new:

```
anInteger
```

server:

```
aClusterDataServer
```

type:

```
aClusterClass
```

```
^super new initialize: anInteger server: aClusterDataServer type:
                                                        aClusterClass
```

Instance Methods

accumulate:

```
aVector
```

```
( result at: ( self indexOfNearestCluster: aVector)) accumulate:
                                                            aVector.
```

clusters:

```
aCollectionOfClusters
```

```
result := aCollectionOfClusters.
```

collectChangesAndResetClusters

```
| hasEmptyClusters changes |
changes := 0.
hasEmptyClusters := false.
```

```
result do:
        [:each |
        changes := each changes + changes.
        ( each isInsignificantIn: self)
            ifTrue:
                [each centerOn: nil.
                hasEmptyClusters := true]
            ifFalse: [each reset].
        ].
hasEmptyClusters
    ifTrue: [result := result reject: [:each | each
                                                    isUndefined]].
^changes / 2
```

dataServer:

```
aClusterDataServer
```

```
dataServer := aClusterDataServer.
```

evaluateIteration

```
dataServer reset.
dataSetSize := 0.
[ dataServer atEnd]
    whileFalse:[ self accumulate: dataServer next.
                 dataSetSize := dataSetSize + 1.
                ].
^self collectChangesAndResetClusters
```

finalizeIterations

```
dataServer close
```

indexOfNearestCluster:

```
aVector
```

```
| distance index |
index := 1.
distance := ( result at: 1) distanceTo: aVector.
2 to: result size do:
    [ :n | | x |
      x := ( result at: n) distanceTo: aVector.
      x < distance
        ifTrue: [ distance := x.
                  index := n.
                ].
    ].
```

```
    ^index
```

initialize:

```
    anInteger
```

server:

```
    aClusterDataServer
```

type:

```
    aClusterClass

    self dataServer: aClusterDataServer.
    self clusters: ( (1 to: anInteger) collect: [ :n | aClusterClass
                                                              new]).
    minimumRelativeClusterSize := 0.
    ^self
```

initializeIterations

```
    dataServer open.
    result
        do: [:each | each isUndefined ifTrue: [each centerOn:
                                                    dataServer next]]
```

minimumClusterSize

```
    ^(minimumRelativeClusterSize * dataSetSize) rounded
```

minimumRelativeClusterSize:

```
    aNumber

    minimumRelativeClusterSize := aNumber max: 0.
```

printOn:

```
    aStream

    aStream nextPutAll: 'Iterations: '.
    iterations printOn: aStream.
    result do: [ :each | aStream cr. each printOn: aStream].
```

Listing 12.12 shows the implementation of the concrete cluster class `DhbEuclideanCluster`. The corresponding accumulator is an instance of class `DhbVectorAccumulator`. Data points are directly accumulated into the accumulator; individual data points are not kept.

Listing 12.12 Smalltalk implementation of a Euclidean cluster

Class DhbEuclideanCluster
Subclass of DhbCluster
Instance variable names: center

Instance Methods

centerOn:

```
aVector

center := aVector.
accumulator := DhbVectorAccumulator new: aVector size.
```

collectAccumulatorResults

```
center := accumulator average copy.
```

distanceTo:

```
aVector

^( aVector - center) norm
```

isUndefined

```
^center isNil
```

printOn:

```
aStream

accumulator count printOn: aStream.
aStream nextPutAll: ': '.
center printOn: aStream.
```

12.6.4 Cluster Analysis—Java Implementation

Listing 12.13 shows the abstract implementation of the *K*-cluster algorithm in Java. Listing 12.14 describes a concrete implementation using Euclidean distance. Code Example 12.8 shows how to implement a search for Euclidean clusters.

Code Example 12.8

<*Creating an instance of a concrete subclass of* `AbstractDataServer` *into the variable* `dataServer` >

```
EuclideanCluster[] clusters = new EuclideanCluster[6];
```

```
for ( int i = 0; i < clusters.length; i++ )
       clusters[i] = new EuclideanCluster();
ClusterFinder finder =
           new ClusterFinder( clusters, server);
try { finder.setMinimumRelativeClusterSize( 0.09);}
       catch( IllegalArgumentException e){};
finder.evaluate();
EuclideanCluster[] clusters = finder.getClusters();
```

The first line after setting up the data server creates an array of six Euclidean clusters. The loop after this line populates this array with empty clusters. Next, an instance of a cluster finder is created with the array of clusters and the data server. The statement within the `try...catch` block defines the minimum relative size of significant clusters. The next statement performs the algorithm. The last statement retrieves the array of found clusters.

The constructor for the class `ClusterFinder` takes two arguments: an array of clusters and the data server. The array of clusters may contain undefined clusters. The undefined clusters are initialized in the method `initializeIterations`. The data server must be an instance of a concrete data server class as described in Section 12.1.2.

Listing 12.13 Java *K*-cluster algorithm

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbVector;

// Abstract cluster.

// @author Didier H. Besset

public abstract class Cluster
{
    protected long previousSampleSize = 0;

// Default constructor method.

public Cluster() {
}

// Constructor method.

public Cluster(DhbVector v)
{
    initialize(v);
```

```
}

// @param Object    data point

public abstract void accumulate(DhbVector dataPoint);

// @param Object    data point

public abstract double distanceTo(DhbVector dataPoint);

// @return long    number of data points taken from or added to the receiver

public long getChanges()
{
    return Math.abs( getSampleSize() - previousSampleSize);
}

// @return long    number of data points accumulated in the receiver

public abstract long getSampleSize();

// @param v DhbMatrixAlgebra.DhbVector

public abstract void initialize( DhbVector v);

// @return boolean    true if no data was accumulated in the receiver

public boolean isEmpty()
{
    return getSampleSize() == 0;
}

// @return boolean    true if the receiver should be dropped from
//                                    the cluster finder
// @param finder DhbDataMining.ClusterFinder

public boolean isInsignificantIn( ClusterFinder finder)
{
    return getSampleSize() <= finder.minimumClusterSize();
}

// @return boolean    true if the cluster is in an undefined state.

public abstract boolean isUndefined( );
public void reset()
{
```

```
    previousSampleSize = getSampleSize();
}
}
```

```
package DhbDataMining;

import DhbIterations.IterativeProcess;
import DhbMatrixAlgebra.DhbVector;

// Implements k-cluster algorithm

// @author Didier H. Besset

public class ClusterFinder extends IterativeProcess
{
    private Cluster[] clusters;
    private AbstractDataServer server;
    private double minimumRelativeClusterSize = 0;
    private long dataSetSize;

// Constructor method
// @param Cluster[] clusterArray    initial clusters
// @param server AbstractDataServer    server for the data points

public ClusterFinder(Cluster[] clusterArray,
                                    AbstractDataServer clusterServer)
{
    clusters = clusterArray;
    server = clusterServer;
}

// Constructor method
// @param numberOfCluster int    maximum number of clusters foreseen
// @param server AbstractDataServer    server for the data points

public ClusterFinder(int numberOfCluster,
                                    AbstractDataServer clusterServer)
{
    clusters = new Cluster[numberOfCluster];
    server = clusterServer;
}

// Accumulate all data points into the nearest cluster.

private void accumulateData()
{
    dataSetSize = 0;
```

```
        try {
            while (true)
            {
                DhbVector dataPoint = server.read();
                nearestCluster( dataPoint).accumulate( dataPoint);
                dataSetSize += 1;
            }
        } catch ( java.io.EOFException e) {}
    }

    // @return int    number of data points which changed clusters since
    //                                the last iteration

    private int collectChangesAndResetClusters()
    {
        int n = 0;
        int emptyClusters = 0;
        for ( int i = 0; i < clusters.length; i++ )
        {
            n += clusters[i].getChanges();
            if ( clusters[i].isInsignificantIn( this) )
            {
                emptyClusters += 1;
                clusters[i] = null;
            }
            else
                clusters[i].reset();
        }
        if ( emptyClusters > 0 )
            removeEmptyClusters( emptyClusters);
        return n / 2;
    }

    // Perform one iteration step.
    // @return double    number of data points which changed clusters
    //                              since the last iteration

    public double evaluateIteration()
    {
        server.reset();
        accumulateData();
        return collectChangesAndResetClusters();
    }

    // Closes the data point server
```

```
public void finalizeIterations()
{
    server.close();
}

// @return DhbDataMining.Cluster[] clusters contained in the receiver

public Cluster[] getClusters()
{
    return clusters;
}

// Opens the data stream and creates the initial clusters.

public void initializeIterations()
{
    server.open();
    try {
        for ( int i = 0; i < clusters.length; i++ )
            {
                if ( clusters[i].isUndefined() )
                    clusters[i].initialize(server.read());
            }
        } catch (java.io.EOFException e){};
}

// @return long    minimum cluster size to be considered in the next iteration

public long minimumClusterSize()
{
    return Math.round( minimumRelativeClusterSize * dataSetSize);
}

// @param dataPoint ClusterData
// @return Cluster    nearest to the data point

private Cluster nearestCluster( DhbVector dataPoint)
{
    int index = 0;
    int nearestIndex = index;
    double closestDistance = clusters[index].distanceTo( dataPoint);
    double distance;
    while ( ++index < clusters.length )
    {
        distance = clusters[index].distanceTo( dataPoint);
        if ( distance < closestDistance )
```

```
            {
                closestDistance = distance;
                nearestIndex = index;
            }
        }
        return clusters[nearestIndex];
    }

    // Removes empty clusters. The array of clusters is reconstructed.
    // @param n int    number of empty clusters

    private void removeEmptyClusters( int n)
    {
        Cluster[] newClusters = new Cluster[ clusters.length - n];
        int index = 0;
        for ( int i = 0; i < clusters.length; i++ )
        {
            if ( clusters[i] != null )
                newClusters[index++] = clusters[i];
        }
        clusters = newClusters;
    }

    // @return DhbDataMining.Cluster[] clusters contained in the receiver

    public void setClusters( Cluster[] clusterArray)
    {
        clusters = clusterArray;
    }

    // @param r double    the minimum relative size of a cluster to be kept
    //                                in the next iteration
    // @exception java.lang.IllegalArgumentException
    //                            argument cannot be negative.

    public void setMinimumRelativeClusterSize( double r)
                                        throws IllegalArgumentException
    {
        if ( r < 0 )
            throw new IllegalArgumentException(
                            "Relative cluster size cannot be negative");
        minimumRelativeClusterSize = r;
    }

    // @return java.lang.String
```

```
public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( "Iterations: "+getIterations());
    for ( int i = 0; i < clusters.length; i++)
    {
        sb.append( '\n');
        sb.append( clusters[i]);
    }
    return sb.toString();
}
}
```

The concrete class `EuclideanCluster` implements clusters using the Euclidean distance defined by equation 12.12. It has the following instance variables:

`center` The center of the cluster (i.e., the position from which distances to the cluster are calculated)

`accumulator` An instance of the class `VectorAccumulator` described in Section 12.2.4

The accumulation of the data points is delegated the accumulator.

Listing 12.14 Java implementation of a Euclidean cluster

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.DhbIllegalDimension;

// Cluster using Euclidean distance.

// @author Didier H. Besset

public class EuclideanCluster extends Cluster
{
    private DhbVector center;
    private VectorAccumulator accumulator;

// Default constructor method.

public EuclideanCluster()
{
    super();
}
```

```
// Constructor method.
// @param DhbVector center of the receiver

public EuclideanCluster(DhbVector dataPoint)
{
    super( dataPoint);
}

// @param dataPoint DhbVector    data point

public void accumulate(DhbVector dataPoint)
{
    accumulator.accumulate( dataPoint);
}

// @param dataPoint DhbVector    data point
// @return DhbVector    square of the Euclidian distance from the data
//                      point to the center of the receiver.

public double distanceTo( DhbVector dataPoint)
{
    try{ DhbVector v = dataPoint.subtract( center);
         return v.product( v);}
        catch(DhbIllegalDimension e) { return Double.NaN;}
}

// @return long    number of data points accumulated in the receiver

public long getSampleSize()
{
    return accumulator.getCount();
}

// @param v DhbVector    center for the receiver

public void initialize(DhbVector dataPoint)
{
    center = dataPoint;
    accumulator = new VectorAccumulator( dataPoint.dimension());
}

// @return boolean    true if the cluster is in an undefined state.

public boolean isUndefined()
{
    return center == null;
```

```
}
public void reset()
{
    super.reset();
    center = accumulator.averageVector();
    accumulator.reset();
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    sb.append( previousSampleSize);
    sb.append( ' ');
    sb.append( center);
    return sb.toString();
}
}
```

12.7 Covariance Clusters

As we have seen in Section 12.5, the Mahalanobis distance is a distance in the geometrical sense. Thus, this distance can be used by the *K*-cluster algorithm. We call clusters using the Mahalanobis distance *covariance clusters* since the metric for the distance is based on the covariance matrix.

The normalizing properties of the Mahalanobis distance makes it ideal for this task. When Euclidean distance is used, the metric remains the same in all directions. Thus, the extent of each cluster has more or less circular shapes. With the Mahalanobis distance, the covariance metric is unique for each cluster. Thus, covariance clusters can have different shapes since the metric adapts itself to the shape of each cluster. As the algorithm progresses, the metric changes dynamically.

12.7.1 Covariance Clusters—General Implementation

Covariance clusters need little implementation. All tasks are delegated to a Mahalanobis center described in Section 12.5. Listing 12.15 shows the Smalltalk implementation, and the Java implementation is shown in Listing 12.16.

Listing 12.15 Smalltalk covariance cluster

Class	`DhbCovarianceCluster`
Subclass of	`DhbCluster`
Instance variable names:	`center`

Instance Methods

centerOn:

```
aVector

accumulator := aVector ifNotNil: [ :v | DhbMahalanobisCenter
                                                    onVector: v].
```

collectAccumulatorResults

```
accumulator computeParameters.
```

distanceTo:

```
aVector

^accumulator distanceTo: aVector
```

isUndefined

```
^accumulator isNil
```

printOn:

```
aStream

accumulator printOn: aStream.
```

Listing 12.16 Java covariance cluster

```
package DhbDataMining;

import DhbMatrixAlgebra.DhbVector;
import DhbMatrixAlgebra.SymmetricMatrix;
import DhbMatrixAlgebra.DhbIllegalDimension;

// Cluster using Mahalanobis distance.

// @author Didier H. Besset

public class CovarianceCluster extends Cluster
{
    private MahalanobisCenter center = null;

// Default constructor method.

public CovarianceCluster()
{
    super();
```

```
}

// Constructor method.
// @param DhbVector center of the receiver

public CovarianceCluster(DhbVector v)
{
    super(v);
}

// Accumulation is delegated to the Mahalanobis center.

public void accumulate(DhbMatrixAlgebra.DhbVector dataPoint)
{
    center.accumulate( dataPoint);
}

// Distance computation is delegated to the Mahalanobis center.

public double distanceTo( DhbVector dataPoint)
{
    return center.distanceTo( dataPoint);
}

// @return long    number of data points accumulated in the receiver

public long getSampleSize()
{
    return center.getCount();
}

// @param v DhbVector    center for the receiver

public void initialize(DhbVector v)
{
    center = new MahalanobisCenter( v);
}

// @return boolean    true if the cluster is in an undefined state.

public boolean isUndefined()
{
    return center == null;
}
public void reset()
{
```

```
        super.reset();
        center.computeParameters();
        center.reset();
    }

    // @return java.lang.String

    public String toString() {
        return center.toString();
    }
    }
```

APPENDIX A

Decimal Floating-Point Simulation

The class `DhbDecimalFloatingNumber` is intended to demonstrate rounding problems with floating-point number representation. It models the floating-point number representation. The radix of the floating-point representation is a decimal to ease the reading of the results. It also allows people to carry some of the operations by hand. This model is almost equivalent to what is done inside a computer. One notable difference is the absence of exponent offset. Another difference is that there is no limit on the size of the exponent. Simple as it is, this model can be used to illustrate rounding problems to beginners. This class is only intended for didactical purposes.

Only the Smalltalk implementation is given here (see Listing A.1), as Java does not lend itself to operator overloading. Moreover, fraction arithmetic is not available in Java. Thus, making an equivalent class would require much more code.

Instances of the class are created with the method `new:` supplying any number as argument. For example,

```
DhbDecimalFloatingNumbernew:3.14159265358979323846264 3
```

Arithmetic operations are performed as usual. For example, the first expression of Section 1.3.2 can be coded as given in Code Example A.1.

Code Example A.1

```
| a b |
a := DhbDecimalFloatingNumber new: (2 raisedToInteger: 64).
b := DhbDecimalFloatingNumber new: 300.
a + b
```

The class has two instance variables:

`mantissa` Contains the mantissa of the number normalized to a finite number of digits.

`exponent` Contains the negative of the decimal exponent of the number.

The class has one class variable:

`Digits` Contains the maximum number of digits kept in the mantissa of all instances.

The method `normalize:` computes the mantissa and exponent of the argument and stores it into the corresponding instance variables. The method `value` allows retrieving the value of the number in numerical form. The method `printOn:` allows visualizing the representation of the number.

The number of decimal digits kept in the mantissa can be changed by the user using the method `setDigits:`. By default it is set to 15, which corresponds roughly to the precision of a 64-bit IEEE standard floating-point number.

The four arithmetic operations and the square root have been defined. The mechanism of each operation is the same. First each operand is converted to a fraction. Then, the operation is carried using normal numbers. In the case of arithmetic operation, the result is exact since the use of Smalltalk fractions guarantees absolute precision. Then, a new instance of the class `DhbDecimalFloatingNumber` is created from the result and is returned. Rounding errors will occur during this step.

Listing A.1

Smalltalk code simulating decimal floating-number arithmetic

Class	`DhbDecimalFloatingNumber`
Subclass of	`Object`
Instance variable names:	`mantissa exponent`
Class variable names:	`Digits`

Class Methods

defaultDigits

```
^15
```

defaultDigits: `anInteger`

```
Digits := anInteger.
```

digits

```
Digits isNil
    ifTrue: [ Digits := self defaultDigits].
```

```
^Digits
```

new: aNumber

```
^self new normalize: aNumber
```

resetDigits

```
Digits := nil.
```

Instance methods * aNumber

```
^self class new: ( self value * aNumber value)
```

+ aNumber

```
^self class new: ( self value + aNumber value)
```

- aNumber

```
^self class new: ( self value - aNumber value)
```

/ aNumber

```
^self class new: ( self value / aNumber value)
```

normalize: aNumber

```
exponent := (self class digits - (aNumber log: 10)) floor.
mantissa := ( aNumber * ( 10 raisedToInteger: exponent))
                                                    truncated.
^self
```

printOn: aStream

```
mantissa printOn: aStream.
aStream nextPutAll: 'xE'.
exponent negated printOn: aStream.
```

sqrt

```
^self class new:  self value sqrt
```

value

```
^mantissa / ( 10 raisedToInteger: exponent)
```

APPENDIX B

Smalltalk Primer for Java Programmers

This appendix is meant as a quick introduction to the Smalltalk language for Java programmers. The goal of the explanations is not to expose all features of Smalltalk but to give enough background so that the reader is able to read the Smalltalk code presented in this book.

A few Smalltalk entry level books are on the market, the most recent being the book by David Smith [Smith]. Kent Beck's book [Beck] is a good choice to deepen Smalltalk knowledge and the Smalltalk-specific object-oriented approach.

B.1 Syntax in a Nutshell

In Smalltalk there is no primitive type. Everything is an object. Objects are represented in the code by a symbol. Objects communicate together by sending messages to each other. As in Java, a symbol may contain any alphanumerical characters and must begin with a lowercase letter.

B.1.1 Smalltalk Expressions

A Smalltalk expression is composed of an identifier representing the object receiving the message followed by the message. The message is placed after the name of the object to which it is directed, separated by at least a blank separator (space, tabulation, or carriage return).

Objects are represented by variables in the conventional way. There are three kinds of messages:

1. unary messages
2. binary messages
3. keyword messages

TABLE B.1 Sample Smalltalk messages with their Java equivalent

Message type	Smalltalk	Java
Unary	`s1 size`	`s1.length()`
	`s1 hash`	`s1.hashCode()`
Binary	`s1 = s2`	`s1.equals( s2)`
	`s1 < s2`	`s1.compare(s2)`
Keyword	`s1 at: 3`	`s1.charAt( 3)`
	`s1 indexOfSubCollection: s2 startingAt: 5`	`s1.indexOf(s2,5)`
	`s1 copyFrom: 3 to: 5`	`s1.substring(3,5)`

Unary messages correspond to a Java method without argument. They are represented by symbols containing any alphanumerical characters and beginning with a lowercase letter.

Binary messages correspond to conventional operators in other languages, Java included. Examples are the arithmetic operators (`+`, `-`, `*`, and `/`) or the relational operators (`=`, `>`, `<`, `>=`, and `<=`). In Smalltalk, the inequality operator is noted as `~=`. Other nonalphabetical operator symbols can be used as binary messages.

Keyword messages correspond to a Java method with one or more arguments. In a Smalltalk keyword message, each argument is placed after a keyword. Each keyword is written as a symbol containing any alphanumerical character, beginning with a lowercase letter and ending with a colon (`:`).

Table B.1 shows a few examples of Smalltalk messages together with their Java equivalents. To make the examples easy to follow, the objects used are either constants or objects from the single class `String`. These objects are denoted by the symbols `s1` and `s2`. The class `String` has the advantage of being the same in both Smalltalk and Java.

B.1.2 Precedence

Arguments of binary and keyword messages may be other Smalltalk expressions. In case of combined expressions, the precedence goes as follows: unary messages are evaluated first, then binary messages, and finally keyword messages. Messages of the same type are evaluated from left to right, which gives a somewhat unconventional precedence rule for arithmetic expressions. As in any other computer language, expressions enclosed within parentheses are always evaluated first, starting with the innermost pair of parentheses.

As a consequence, keyword messages used as arguments to other messages must always be enclosed within parentheses.

TABLE B.2 Smalltalk assignment and equalities

Operator	Smalltalk	Java
Assignment	`:=`	`=`
Equality	`=`	`equals()`
Inequality	`~=`	`!equals()`
Identity	`==`	`==`
Nonidentity	`~~`	`!=`

B.1.3 Assignment, Equality, and Identity

In Smalltalk, the assignment operator is composed of an equal sign preceded by a colon (`:=`). This corresponds to the equal sign in Java.

Like Java, Smalltalk differentiates between equality and identity. The equality operator is an equal sign, corresponding to the method `equals` of Java. The inequality operator is written `~=`, corresponding to the Java `!equals`. The identity operator is a double equal sign (`==`) as in Java. The negation of the identity is written as `~~`. Table B.2 presents Smalltalk assignment and equalities and their Jave equivalents.

B.2 Class and Methods

A Smalltalk class is quite similar to a Java class. The main difference is that Smalltalk is not file oriented. Classes are not assigned to a file as in Java, instead, they reside in the Smalltalk image—that is, a copy of the memory used by the Smalltalk system when executing.

As a consequence, any class can be extended by anyone. If an application designer is missing a method from the base classes, he or she simply adds the method. This book contains numerous example of methods added to the base classes `Number` and `Integer`.

A class is declared by stating its superclass and the instance variables. There are other parameters defining a class, but I do not mention them as they are not used in this book. As in Java, the class `Object` is the topmost class.

Smalltalk instance variables are listed as symbols without types. Smalltalk is a dynamically typed language. In principle, an instance variable can hold any object. At run time, however, the type of the instance is known, which is how the virtual machine knows how to retrieve the methods supported by the object.

B.2.1 Instance methods

An instance method is very similar to a Java instance method. Of course, the syntax is quite different. At best, we shall discuss an example taken from Listing 8.15. The lines are numbered for easier reference.

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

The first line is the method declaration—that is, the declaration of the message sent when this method is executed. In this example, this is an unary message named `evaluateIteration`.

Line 2 is the declaration of the variables local to the method. Since Smalltalk is not typed, only the names of the variable are enumerated between two vertical bars. If a method does not have local variables, this line is omitted. Here the only declared variable is `indices`.

Line 3 is an assignment statement: the local variable `indices` is assigned to the result of sending the message `largestOffDiagonalIndices` to the variable `self`. `self` is the instance, which is executing the method. In other words, it is equivalent to the Java variable `this`. The statement is terminated with a period (`.`) corresponding to the semicolon used in Java.

Line 4 is a simple statement. The message `transformAt:and:` is sent to the instance executing the method. The arguments are the results of two keyword messages (`at:`) sent to the variable `indices`. In this case, the variable `indices` was set to an array with two elements. These elements are used as arguments for the message `transformAt:and:`. Here again the statement is terminated by a period.

Line 5 is the last statement of the method. The wedge (^) indicates that the expression that follows is the result of the method. In this case, the result of the method is the value of the instance variable `precision`. A return statement is not terminated.

The next example is taken from Listing 8.8. It is far from being simple but rather covers more advanced features. Together with the first example, we have covered the entire syntax of Smalltalk.

```
1   decompose
2   | n |
3   n := rows size.
4   permutation := (1 to: n) asArray.
5   1 to: ( n - 1) do:
6      [ :k |
7      self swapRow: k withRow: ( self largestPivotFrom: k);
8             pivotAt: k.
9      ].
```

The first line is the method declaration for the unary message named `decompose`.

Line 2 is the declaration of the local variable `n`.

Line 3 is an assignment statement: the local variable `n` is set to the number of rows. The variable `rows` is an instance variable of the class and is set to an array; the message `size` returns the number of elements located in the array. The statement is terminated with a period.

Line 4 is another assignment statement. It assigns the instance variable `permutation` to an array containing the series of integers 1 to *n*. The message `to:` sent to an integer answers an interval. It must be converted to an array with the message `asArray`. Here again the statement is terminated by a period.

Line 5 is the beginning of an iterator message ending at line 9. (Iterator methods are described in Section B.3. The object to which the iterator message is sent is an interval from 1 to $n - 1$. This line is equivalent to the Java statement `for (int k = 1; k < n; k++)`. The reader should notice that indices in Smalltalk begin at 1 instead of zero.

Line 6 is the beginning of the block context, argument of the `do:` message. This line is declarative and states that the variable used to evaluate the block context is named `k`.

Line 7 contains two messages sent to the variable `self`. The first message to be executed is a keyword message—`largestPivotFrom:`—with one argument `k`. The second message is a keyword message `swapRow:withRow:` with two arguments: the first argument is `k`, and the second argument is the result of the message `largestPivotFrom:`.

Unlike the preceding statements, this statement is terminated by a semicolon (`;`). In Smalltalk a semicolon is not a statement terminator. The semicolon indicates that the message on line 8, written without explicit target, is directed to the same target as the preceding message. In this case, the target of the message `pivotAt:`—a keyword message with one argument `k`—is `self`.

Line 9 is the end of the statement beginning on line 5. It is the terminating bracket of the block context beginning on line 6. This statement is terminated with a period. Because this method does not return an explicit result, the result of the method is `self`, which is the instance that was executing the method.

B.2.2 Class Methods

The biggest difference between Smalltalk and Java lies in the class methods. As a first approximation, a Java programmer can think that class methods are equivalent to static methods. Class methods, however, are methods like any other methods. In particular, class methods are fully inherited.

Here the Java programmer must bear in mind that everything is an object in Smalltalk. This point is also true for classes. A class is an object and, as such, has methods. Thus, class methods are exactly like instance methods, but they are defined on the class as opposed to the instance. In particular, the variable `self` in a class method now refers to the class itself.

Class methods are also used as class constructor methods. Unlike Java, Smalltalk allows class constructor methods with any name. The default creation method `new` is provided by the superclass of all classes. It creates a new instance of the class with all instance variables set to nil. An application designer can choose to redefine the method `new` for a given subclass. This book shows several example of this approach.

B.2.3 Block Context

In Smalltalk everything is an object—and, yes, this is the third time this important statement has been made! It holds true for Smalltalk code itself. A Smalltalk code object is specified with a special syntax: the block context. Here is an example of a block context computing polar coordinates from supplied Cartesian coordinates:

```
[ :x :y |
| radius angle |
radius := ( x squared + y squared) sqrt.
angle := y arctan: x.
Array with: radius with: angle]
```

The block context is delimited by a pair of brackets (`[]`). The first line contains the declaration of the block variables. These variables are supplied to the block context at execution time. The next line contains the declaration of the local variables—that is, variables used within the block. The next two lines perform computation needed to obtain the final result. The last expression of the block context is the value returned by the block context at execution time.

Here is an example on how the block of the previous example is used:

```
| block |
block := [ :x :y |
          | radius angle |
          radius := ( x squared + y squared) sqrt.
          angle := y arctan: x.
          Array with: radius with: angle].
block value: 2 value: 3.
```

Of course, this is a contrived example. Usually block contexts are much simpler. They are also seldom used to perform computation for the sake of computation. Functions described in Chapter 2 are good examples of using block contexts.

B.3 Iterator Methods

The most important use of block contexts is within so-called iterator methods. Quickly said, iterator methods provide the functionality of the `Enumeration` interface. Smalltalk iterator methods, however, provide far more flexibility than the `Enumeration` interface.

Iterator methods can be applied to any instance of any subclass of the class `Collection`. A Smalltalk collection is simply a container of objects, not necessarily of the same class. Depending on the particular subclass, a collection has some specific structure. They cover many types including the Java arrays and the Java classes `Vector` or `HashTable`.

There are many iterator methods, but this section describes only the one used in this book.

B.3.1 do:

The `do:` iterator corresponds to the Java `for` loop. In Smalltalk, however, there is no need to supply an index. The block context supplied as argument of the `do:` message is evaluated for each element of the collection. For example, to perform the message `sample` on each element of a collection, one simply writes

```
aCollection do: [ :each | each sample].
```

It is customary to use the variable name `each` as the argument of the block context used with an iterator method.

B.3.2 collect:

The iterator method `collect:` has no equivalent in Java. Its result is a collection of the same type[1] as the collection to which the message was sent. The elements of the new collection are the result of the context block supplied as argument to the method. For example, here is how one can construct an array containing the sum of the squares of the integers 1 to 9:

```
#(1 2 3 4 5 6 7 8 9) collect: [ :each | each squared].
```

The result of this code is the array `#(1 4 9 16 25 36 49 64 81)`.

B.3.3 inject:into:

The iterator method `inject:into:` is a great tool for mathematical evaluation. Because it is rather complex, we shall start with an example. Here is how to compute the sum of the elements of an array:

```
anArray inject: 0 into: [ :sum :each | sum + each].
```

The iterator method `inject:into:` is a keyword message. The first argument is the initial value used in the summation. The second argument is a context block with two arguments: the first argument is the result of the evaluation of the context block with the preceding element or the initial value if this is the first evaluation; the second is the element of the collection over which the iterator method is iterating. The result of the message is the value of evaluating the block on the last element of the collection.

B.4 Double Dispatching

Since the arguments of Smalltalk methods are not typed, a method is able to accept arguments of different classes as long as all possible argument types have the same

1. There are some special collections for which the type of the result is different.

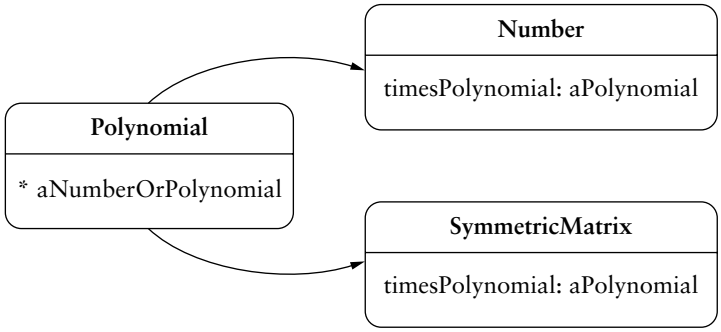

FIG. B.1 Double dispatching

behavior. However, a problem arises when the method must behave differently depending on the type of the argument.

For example, the multiplication operator defined for polynomials (see Section 2.2.3) can accept a number or another polynomial. The implementation of the method strongly differs depending on the type of argument. One possible implementation could be to test for the type of the argument and switch to the proper method accordingly. In this case the code would look as that in Code Example B.1.

Code Example B.1

```
* aNumberOrPolynomial
   ^aNumberOrPolynomial class = DhbPolynomial
       ifTrue: [ self productWithNumber: aNumberOrPolynomial]
       ifFalse:[ self productWithPolynomial: aNumberOrPolynomial]
```

This code, however, is a bad example because it is usually what beginners do, especially those who still think with a strong legacy coming from other languages.

The elegant solution is called *double dispatching* and is illustrated in Figure B.1. It merely uses the fact that testing which class an object belongs to is exactly what the virtual machine is doing when looking for the code of the method corresponding to a message received by an object. Thus, it is not necessary to repeat such a test. Instead, one delegates the execution of the method to the argument object. as shown in Code Example B.2.

Code Example B.2

```
* aNumberOrPolynomial
   ^aNumberOrPolynomial timesPolynomial: self
```

In fact, the sending of the message `timesPolynomial: self` to the method's argument ensures that the types of both operands are now completely known. One must now implement two versions of the method `timesPolynomial:`, one for instances of the class `DhbPolynomial` and one for the class `Number`. Within both

versions of the method, the programmer can be sure that the type of the argument is indeed an instance of class `DhbPolynomial`.

Figure B.1 shows this process schematically. Each box shows the class executing the method on the top and the method with the type of argument at the bottom. The arrows represent the sequence of execution.

One caveat with double dispatching is that the order of the arguments is inverted. Thus, implementing double dispatching for noncommutative operators (i.e., subtract, divide, or matrix multiplication) requires some thinking.

It is easy to understand that double dispatching is much faster that testing the type of the argument. It only requires the invocation of the operation message, whereas testing requires evaluating the test itself plus a message invocation.[2]

B.5 Multiple Dispatching

The technique of double dispatching can be extended to multiple levels. This method is required when an operation is implemented by a class having several subclasses, each subclass requiring a different behavior.

A good example of multiple dispatching is given by the addition between matrices (see Section 8.1.1). The product of two symmetrical matrices is a symmetrical matrix. In all other cases, the result is a nonsymmetrical matrix. Thus, the addition operation is delegated a first time to the proper subclass of the argument and a second time to the class of the first operand. Here we have triple dispatching. This process is shown schematically in Figure B.2.

In the case of matrix multiplication, the situation is more complex since the product is already being dispatched to distinguish among three possible arguments. Here quadruple dispatching is necessary.

2. `ifTrue:ifFalse:` *is* a message sent to the result of the testing.

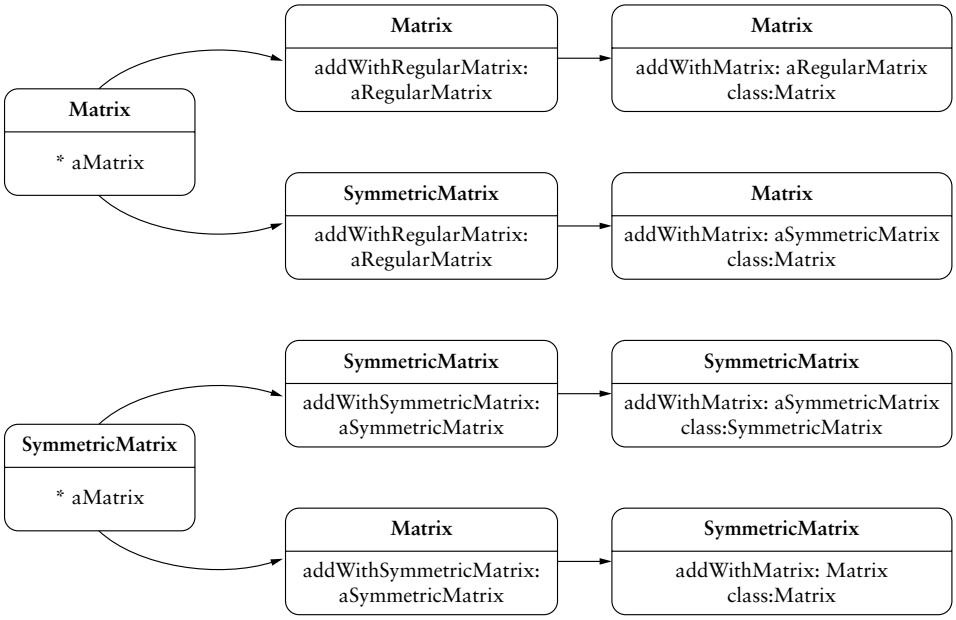

FIG. B.2 Triple dispatching

APPENDIX C

Java Primer for Smalltalk Programmers

This appendix is meant as a quick introduction to the Java language for Smalltalkers. I shall give minimum explanations to give the Smalltalker enough background to be able to read the Java code presented in this book.

Many entry level books about Java are on the market. My favorite is *Java in a Nutshell* [Flanagan]. This book teaches Java assuming that the reader is already familiar with C. A similar approach is used here. This makes sense since the syntax of Java is very close to that of C.

C.1 Remarks on the Syntax

The designer of Java elected to use the existing syntax of C. Compared to Smalltalk, this means a more complex notation. Statements are terminated by a semicolon (;). I shall not explain the syntax because it would be too long, and I shall assume that most Smalltalk programmers have already been exposed to C. Under this assumption, I shall concentrate on the object-oriented features of Java.

Like C, Java is file oriented. An application is made of several files loaded together. One file may contain several class definitions. However, a class definition must be contained within a single file. As a consequence, base classes as well as any class obtained from a third party cannot be extended. The only possibility is to extend them with a sub class when it is possible.[1]

1. Java has a qualifier `final` that is a compiler optimization. Classes with the final qualifier cannot have subclasses.

C.1.1 Classes

A Java class is defined as follows:

```
public class MyClass {}
```

This defines a public class named `MyClass` as a subclass of the class `Object` by default. As in Smalltalk, the superclass of all classes is the class `Object`. This definition of `MyClass` is not a very useful one since it has no instance variable and no method. Instance variables and methods are declared within the brackets following the class name. The keyword `public` states that this class is visible by any other class.

The class `MySubclass` is defined as a subclass of `MyClass` by the following declaration

```
public class MySubclass  extends MyClass {}
```

Instances of class are created via a *constructor* method. The constructor method has the same name as the class and is called by supplying the keyword `new` before the class name. For example, here is how to create a new instance of the class `MyClass`:

```
MyClass x = new MyClass();
```

After this statement, the local variable `x` points to the newly created instance.

Constructor methods may have parameters. Since Java differentiates methods by their names *and* by the type of its arguments, a class can have several constructor methods.

C.1.2 Instance Variables

Instance variables must be declared with a type. An optional qualifier defines the scope (or visibility) of the variable. An instance variable is declared as follows:

```
protected double var1;
```

This statement declares the variable `var1` to be of type `double`, a floating number with double precision. The keyword `protected` indicates that the variable is visible to all subclasses of this class. This is the default when no qualifier is supplied. Protected instance variables behave like instance variables in Smalltalk.

C.1.3 Method Declaration and Method Calling

A Java method is declared with a type. An optional qualifier defines the scope (or visibility) of the method. The name of the method is followed by the list of parameters enclosed in parentheses. The parentheses are always supplied, even if the list of parameters is empty. Each parameter is defined by a type and an identifier. This identifier can be used within the method code to refer to the parameters. As in Smalltalk, all parameters are pointer to the objects. The code of the method, enclosed within braces, follows the declaration.

Here is an example of a method declaration:

```
public DhbVector gradient( double x) {...}
```

This method is declared as returning an object of type `DhbVector`. The argument of the method is a `double`. The code of the method must be located within braces following the method's declaration (my example does not show the code).

One interesting aspect of Java is that one can declare two methods with the same name provided the type of the arguments is different. For example, the following two methods can also be defined within the same class where the prior `gradient` method is defined:

```
public DhbVector gradient( float x) {...}

public DhbVector gradient( double x, double[] parameters) {...}
```

At calling time, the Java virtual machine selects the proper method depending on the type of the supplied argument(s).

The code of a Java method is of procedural nature, mainly because of the use of the C syntax and constructs—`for` and `while` loops, `if` and `if...else`.

Calling a method is done by appending the name of the method to the object to which the method is sent separated with a period. For example, the prior methods can be called as follows:

```
MyClass x = new MyClass();
DhbVector v = x.gradient( Math.PI);
```

C.1.4 Objects and Nonobjects

In Smalltalk everything is an object. This is not the case in Java. It has so-called primitive types that are handled like values as in conventional programming languages. The primitive types are `char`, `int`, `long`, `float`, `double`, and `boolean`.

To use values as objects, Java provides so-called wrappers for each of the primitive type. For example, if a double needs to be passed to a method expecting an argument of type `Object`, the double must be wrapped using the `Double` class wrapper as follows:

```
double pi = 3.14159;
myMethod( new Double(pi));
```

C.1.5 Packages

A Java package provide a solution to the name space, which has been the holy grail of Smalltalk vendors for a long time. A Java package provides a way to group cooperating classes together. Classes from one package are not visible to classes of other packages. Thus, two classes in an application may have the same name as long as they are in different packages. To refer to both of these classes in one common place, the designer must use the package name followed by a period and the class name. For example, if you already have a class named `Matrix` and that you want

to use the class `Matrix` of this book, you must refer to the class of this book as: `DhbMatrixAlgebra.Matrix`.

The `import` statement allows making classes from another package visible for all methods of a class. Then, a class can be referred to by its name only. Our Java code make ample use of import statements.

C.1.6 Scope qualifiers

Java supports three types of scope qualifiers:

- `private`
- `protected`
- `public`

The meaning of the qualifiers is slightly different when applied to classes, methods, or instance variables.

A private instance variable is accessible only to the class where this instance variable is declared. This means that a private instance variable is *not visible* to the subclass of a class where the instance variable is declared. A protected instance variable is accessible to all subclasses of the class where this instance variable is declared. It is also visible to all classes located in the same package as the class defining the variable. A public instance variable is accessible by any object.

The scope qualifiers applied to a method have the same properties as those for instance variables.

A private class is accessible by all classes within the same file. A protected class is visible to all classes located in the same package. A public class is visible to any class.

The scope qualifiers are also used to optimize code execution. This has an odd consequence for Smalltalkers: a private method of a subclass called in a method from a superclass will not be executed; the private method of the superclass is executed instead. For example, consider the two classes shown in Listing C.1

Listing C.1 Inheritance of private methods

```
package InheritanceCheck;

public class MyClass {
public MyClass() {
    super();
}
public static void main(java.lang.String[] args)
{
    MyClass objectA = new MyClass();
    System.out.println("Start running MyClass.privateCommon");
```

```
        objectA.privateCommon();
        MySubclass objectB = new MySubclass();
        System.out.println("Start running MySubclass.privateCommon");
        objectB.privateCommon();
        System.out.println("Start running MyClass.protectedCommon");
        objectA.protectedCommon();
        System.out.println("Start running MySubclass.protectedCommon");
        objectB.protectedCommon();
    }
    private void privateCommon()
    {
        System.out.println("\t=> Executing private method in
                                                    superclass.");
    }
    private void privateSuper() {
        privateCommon();
    }
    protected void protectedCommon()
    {
        System.out.println("\t=> Executing protected method in
                                                    superclass.");
    }
    protected void protectedSuper() {
        protectedCommon();
    }
    }

    package InheritanceCheck;

    public class MySubclass extends MyClass {
    public MySubclass() {
        super();
    }
    public static void main(java.lang.String[] args)
    {
        MySubclass objectB = new MySubclass();
        System.out.println("Start running MySubclass.privateSub");
        objectB.privateSub();
    }
    private void privateCommon()
    {
        System.out.println("\t=> Executing private method in
                                                    subclass.");
    }
    private void privateSub() {
```

```
        privateCommon();
    }
    protected void protectedCommon()
    {
        System.out.println("\t=> Executing protected method in
                                                    subclass.");
    }
    }
```

The method `privateCommon` executed within the method `privateSuper` is always that of the class `MyClass`. Smalltalkers should carefully study the code because it works against their logic. Executing the main method of class `MyClass` yields the following result:

```
Start running MyClass.privateCommon
   => Executing private method in superclass.
Start running MySubclass.privateCommon
   => Executing private method in superclass.
Start running MyClass.protectedCommon
   => Executing protected method in superclass.
Start running MySubclass.protectedCommon
   => Executing protected method in subclass.
```

Executing the main method of class `MySubclass` yields the following result:

```
Start running MySubclass.privateSub
   => Executing private method in subclass.
```

This surprising behavior is not a bug. It is a documented feature of the Java compiler and virtual machine optimization. This apparent breach of inheritance is a consequence of code optimization.

C.1.7 Static Qualifier

The `static` qualifier defines that a variable or a method is attached to the class and not to an instance.

Smalltalkers can think of static variables as class variables. Static methods can be considered as class variables, however, they have some subtle differences compared to their Smalltalk equivalents.

C.2 Abstract Class and Interface

In Java a class or a method can be declared as `abstract`.

An abstract class is meant to be the start of a subhierarchy of classes. In Smalltalk the concept of abstract class is purely a matter of convention. In Java the abstract qualifier is enforced: an abstract class cannot have any instances.

An abstract method can only be defined within an abstract class or an interface (discussed later). It is defined as follows:

```
protected abstract double computePrecision();
```

Declaring a method as abstract instructs the Java compiler to require that any concrete subclass must implement such a method: a concrete subclass not defining concrete implementations of the abstract methods of its superclasses is flagged as an error at compiling time.

An interface is a new concept introduced by Java (at least to my knowledge). One can consider an interface as an abstract class, which cannot have any instance variable or concrete methods. Interfaces come in handy when constructing polymorphic behavior between classes that are not part of the same hierarchy. Here is an example of interface definition:

```
public interface PlanarShape {
   protected abstract double getPerimeter();
   protected abstract double getArea();
}
```

Once an interface has been declared, it can be used as a type declaration. For example, the following method can be written in a class having access to the interface just defined:

```
public double totalArea( PlanarShape[] p) {
   double totalArea = 0;
   for ( int i = 0; i < p.length; i++)
       totalArea += p.getArea();
   return totalArea;
}
```

At this point, the designer of the interface has no idea how the method `getArea` can be implemented.

A class declares itself as adhering to an interface as follows:

```
public class Square implements PlanarShape {
   private double side;
.
.
.
   double getArea() {
   return side * side;
   }
}
```

A class can adhere to several interfaces at the same time.

C.3 Exception Handling

Exception handling is built into the language. By this, I mean not only that exception handling structures exist, as in Smalltalk, but also that these structures are enforced by the compiler.

Exceptions are subclasses of the class `Exception`. As in Smalltalk, a designer can define her own type of exception. Exceptions have two constructor methods: one with no argument and one with a string argument. The string argument is used to pass additional information to the exception. This information is displayed by the debugger or on the console when the exception is not handled by the application.

In the Java jargon, an exception is *thrown* by a method and is *caught* (or better, "`catch`-ed") by a caller of that method, not necessarily the direct caller. Here is an example of a method throwing an exception:

```
public double inverse( double x) throws MyZeroDivideException {
   if ( Math.abs( x) < DhbMath.getMachinePrecision() )
       throw new MyZeroDivideException();
   return 1 / x;
}
```

The declaration of the method uses the modifier `throws MyZeroDivideException` to signal the compiler that this method may throw an exception of the specified class. The actual *throwing* of the exception is made by the statement `throw` followed by a new instance of the exception.

If a method is defined as throwing an exception, any method calling the former must catch the exception. Otherwise, the compiler issues an error. Catching the exception can be made explicitly or implicitly.

Explicit exception catching is made with the statement `try. . .catch`. Here is an example:

```
try{ double x = pivot();
     double inversePivot = inverse(x);
     } catch ( MyZeroDivideException e)
        .
        .   {<code to handle the exception>}
        .
```

To catch an exception, the statements where an exception might occur must be enclosed inside braces preceded by the keyword `try` and followed by the keyword `catch`. The keyword `catch` is followed by parentheses enclosing a declaration of the exception and a series of statements, enclosed within braces, which are handling the exception. The variable corresponding to the exception—`e` in this example—is visible inside the block handling the exception.

Implicit exception catching is made by declaring that the method is throwing the same exception. For example:

```
public double inversePivot() throws MyZeroDivideException {
   double pivot = pivot();
   return inverse( pivot);
}
```

In this case, the method `inversePivot` delegates the handling of the exception to one of its callers.

C.4 Collections and Related Topics

Smalltalkers, when first exposed to Java, surely are appalled at the scarcity of the collection classes and at the absence of decent iterator methods. Well, the sad thing is they are right! The Java designers completely omitted this aspect of programming.

Arrays do exist. However, they are primitive types, not objects. An array is declared with the keyword `new` followed by the type of the array elements and the dimension of the array enclosed in brackets. For example

```
Cluster[] clusterArray = new Cluster[10];
```

declares an array of 10 objects of the class `Cluster`. The reader should note that the type of an array is the type of the element followed by `[]`. In this example, the type of the variable `clusterArray` is `Cluster[]`. All elements of an array must be of the same type. Iterating over the elements of an array must be done using an index in a `for` or `while` loop.

To be complete, I must mention a few Java constructs that provide more flexible collection behavior. They are not used in this book, but I do not want to give Smalltalkers on a bad impression of the Java language.

The Java class `Vector` corresponds to the Smalltalk class `OrderedCollection`. The elements of a `Vector` are of type `Object`. Its elements do not need to be of the same type theoretically. In practice, however, handling a multitype `Vector` is difficult because one needs to cast the element to the proper type when extracting them from the `Vector`.

The Smalltalk class `Dictionary` corresponds to the Java class `HashTable`.

An equivalent to the Smalltalk `do:` iterator method is provided by the `Enumeration` interface to which both `Vector` and `HashTable` are adhering. Each element of the collection is retrieved under the type `Object`. However, this method forces the designer to cast each element to its proper type before using it within the iterator method.

APPENDIX D

Additional Probability Distributions

D.1 Beta Distribution

Table D.1 shows the properties of the beta distribution. If α_1 and α_1 are equal to 1, the beta distribution is identical to the uniform distribution over the interval [0, 1].

The beta distribution is an adhoc distribution, which can take many shapes. Figure D.1 shows a few characteristic shapes for several values of the parameters. The random variable is limited to the interval [0, 1]. However, any distribution of a random variable varying over a finite interval can be mapped to a beta distribution with a suitable variable transformation.

TT

D.1.1 Beta Distribution—Smalltalk Implementation

Listing D.1 shows the implementation of the beta distribution in Smalltalk.

Listing D.1 **Smalltalk implementation of the beta distribution**

Class	`DhbBetaDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`alpha1 alpha2 gamma1 gamma2 logNorm incompleteBeta-Function`

Class Methods

distributionName

```
^'Beta distribution'
```

TABLE D.1 Properties of the beta distribution

Range of random variable	$[0, 1]$
Probability density function	$P(x) = \dfrac{1}{B(\alpha_1, \alpha_2)} x^{\alpha_1 - 1}(a - x)^{\alpha_2 - 1}$
Parameters	$0 < \alpha_1 < +\infty$
	$0 < \alpha_2 < +\infty$
Distribution function	$F(x) = B(x; \alpha_1, \alpha_2)$ (see Section 7.5)
Average	$\dfrac{\alpha_1}{\alpha_1 + \alpha_2}$
Variance	$\dfrac{\alpha_1 \alpha_2}{(\alpha_1 + \alpha_2)^2(\alpha_1 + \alpha_2 + 1)}$
Skewness	$2\dfrac{\alpha_1 - \alpha_2}{\alpha_1 + \alpha_2 + 2)}$
Kurtosis	$\sqrt[3]{\dfrac{\alpha_1 + \alpha_2 + 2}{\alpha_1 \alpha_2}} \left\{ \dfrac{(\alpha_1 + \alpha_2 + 1)}{\alpha_1 \alpha_2 (\alpha_1 + \alpha_2 + 2)(\alpha_1 + \alpha_2 + 3)} \times \left[2(\alpha_1 + \alpha_2)^2 + \alpha_1 \alpha_2 (\alpha_1 + \alpha_2 - 6)\right] - 1 \right\}$

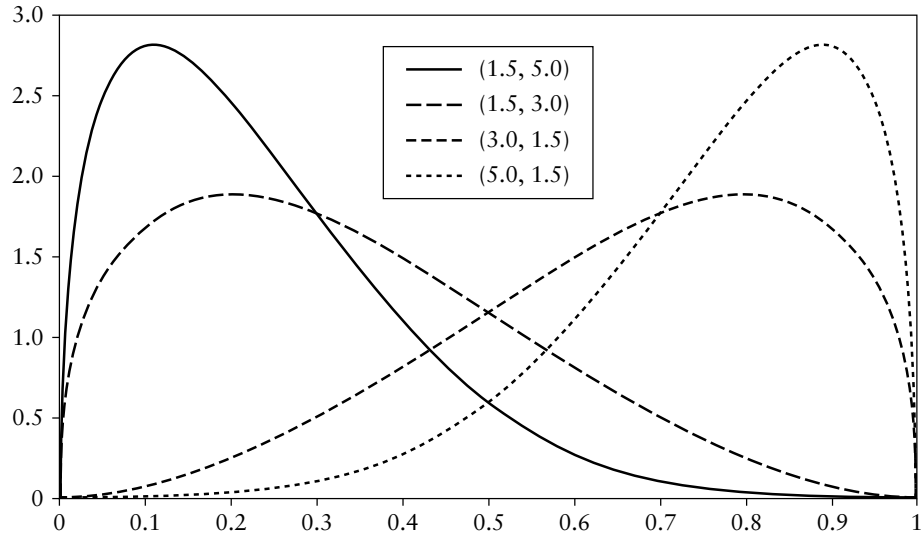

FIG. D.1 Many shapes of the beta distribution

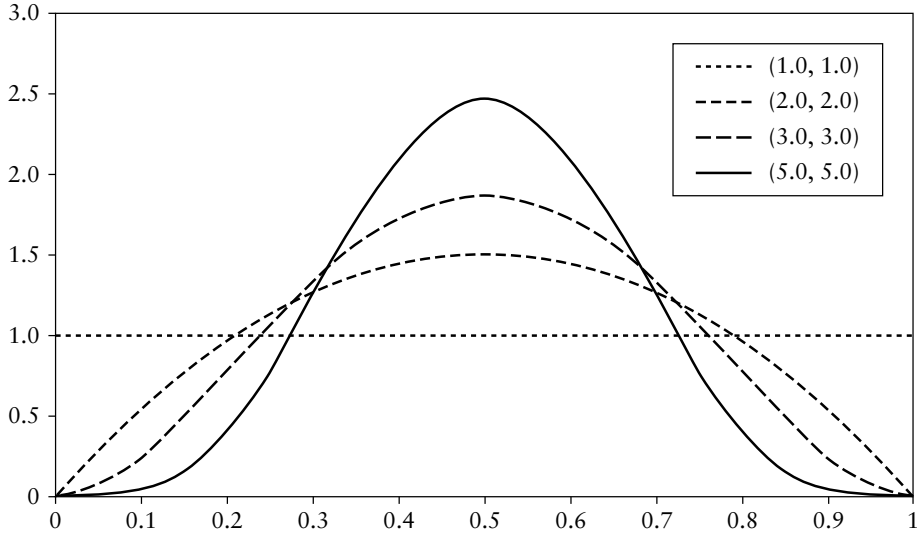
3.0
2.5
2.0
1.5
1.0
0.5
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
(1.0, 1.0)
(2.0, 2.0)
(3.0, 3.0)
(5.0, 5.0)

FIG. D.1

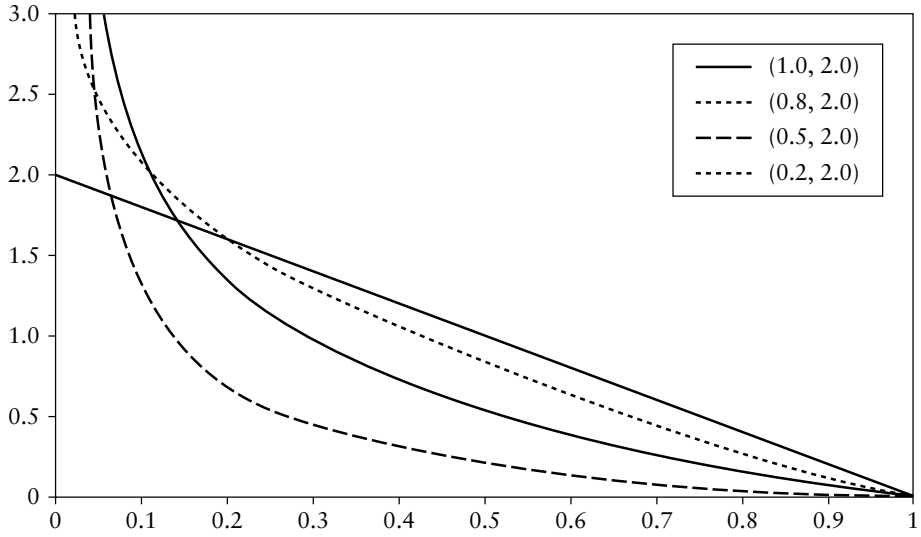
3.0
2.5
2.0
1.5
1.0
0.5
0
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
(1.0, 2.0)
(0.8, 2.0)
(0.5, 2.0)
(0.2, 2.0)

FIG. D.1

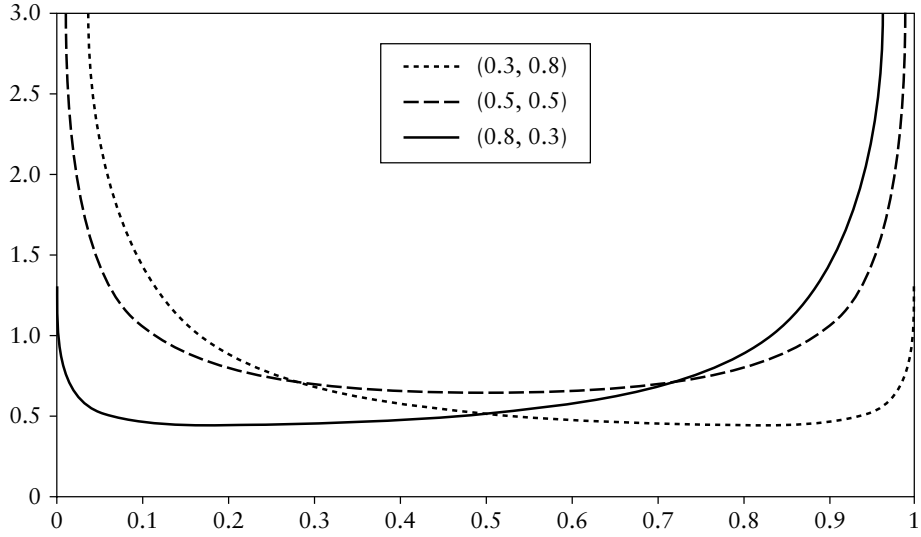

FIG. D.1

fromHistogram: aHistogram

```
| average variance a b c discr |
( aHistogram minimum < 0 or: [ aHistogram maximum > 1])
    ifTrue: [ ^nil].
average := aHistogram average.
variance := aHistogram variance.
a := ( ( 1 - average) / variance - 1) * average.
a > 0
    ifFalse:[ ^nil].
b := ( 1 / average - 1) * a.
b > 0
    ifFalse:[ ^nil].
^self shape: a shape: b
```

new

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

shape: aNumber1 **shape:** aNumber2

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

Instance Methods

average

```
^alpha1 / ( alpha1 + alpha2)
```

changeParametersBy: aVector

```
alpha1 := alpha1 + ( aVector at: 1).
alpha2 := alpha2 + ( aVector at: 2).
self computeNorm.
gamma1 := nil.
gamma2 := nil.
incompleteBetaFunction := nil.
```

computeNorm

```
logNorm := (alpha1 + alpha2) logGamma - alpha1 logGamma - alpha2
                                                              logGamma.
```

distributionValue: aNumber

```
incompleteBetaFunction isNil
    ifTrue: [ incompleteBetaFunction := DhbIncompleteBetaFunction
                                        shape: alpha1 shape: alpha2].
^incompleteBetaFunction value: aNumber
```

firstGammaDistribution

```
gamma1 isNil
    ifTrue: [ gamma1 := DhbGammaDistribution shape: alpha1 scale:
                                                                  1].
 ^gamma1
```

initialize: aNumber1 **shape:** aNumber2

```
(aNumber1 > 0 and: [aNumber2 > 0])
    ifFalse: [self error: 'Illegal distribution parameters'].
alpha1 := aNumber1.
alpha2 := aNumber2.
self computeNorm.
^self
```

kurtosis

```
^3 * ( alpha1 + alpha2 + 1) * ( (alpha1 + alpha2) squared * 2 + (
                          ( alpha1 + alpha2 - 6) * alpha1 * alpha2)
        / ( ( alpha1 + alpha2 + 2) * ( alpha1 + alpha2 + 3) *
                                              alpha1 * alpha2)) - 3
```

parameters

```
^Array with: alpha1 with: alpha2
```

random

```
| r |
r := self firstGammaDistribution random.
^r / ( self secondGammaDistribution random + r)
```

secondGammaDistribution

```
gamma2 isNil
    ifTrue: [ gamma2 := DhbGammaDistribution shape: alpha2 scale:
                                                                1].
 ^gamma2
```

skewness

```
^( alpha1 + alpha2 + 1) sqrt * 2 * ( alpha2 - alpha1) / ( (
                  alpha1 * alpha2) sqrt * ( alpha1 + alpha2 + 2))
```

value: aNumber

```
^(aNumber > 0 and: [ aNumber < 1])
    ifTrue:
        [( ( aNumber ln * (alpha1 - 1) ) + ( ( 1 - aNumber) ln *
                                    ( alpha2 - 1)) + logNorm) exp]
    ifFalse: [0]
```

variance

```
^alpha1 * alpha2 / ( ( alpha1 + alpha2) squared * ( alpha1 +
                                                    alpha2 + 1))
```

D.1.2 Beta Distribution—Java Implementation

Listing D.2 shows the implementation of the beta distribution in Java.

Listing D.2 Java implementation of the beta distribution

```
package DhbStatistics;

import DhbIterations.IncompleteBetaFunction;
import DhbFunctionEvaluation.GammaFunction;
import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Beta distribution
```

```
// @author Didier H. Besset

public final class BetaDistribution
                            extends ProbabilityDensityFunction
{

    // First shape parameter of the distribution.

    protected double alpha1;

    // Second shape parameter of the distribution.

    private double alpha2;

    // Norm of the distribution (cached for efficiency).

    private double norm;

    // Gamma distribution for alpha1 used for random generation
  (cached for efficiency).

    private GammaDistribution gamma1;

    // Gamma distribution for alpha2 used for random generation
  (cached for efficiency).

    private GammaDistribution gamma2;

    // Incomplete beta function for the distribution (cached for
  efficiency).

    private IncompleteBetaFunction incompleteBetaFunction;

// Create a new instance of the Beta distribution with given shape and scale.
// @param shape1 double first shape parameter of the distribution (alpha1).
// @param shape2 double second shape parameter of the distribution alpha2).
// @exception java.lang.IllegalArgumentException
//                 if the parameters of the distribution are illegal.

public BetaDistribution ( double shape1, double shape2)
                                    throws IllegalArgumentException
{
    if ( shape1 <= 0 )
        throw new IllegalArgumentException(
                            "First shape parameter must be positive");
    if ( shape2 <= 0 )
        throw new IllegalArgumentException(
```

```
                                "Second shape parameter must be positive");
    defineParameters( shape1, shape2);
}

// 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 BetaDistribution( Histogram h) throws IllegalArgumentException
{
    if ( h.getMinimum() < 0 || h.getMaximum() > 1)
        throw new IllegalArgumentException(
                    "Beta distribution is only defined over [0,1]");
    double average = h.average();
    double variance = h.variance();
    double a = ( ( 1 - average) / variance - 1) * average;
    if ( a <= 0 )
        throw new IllegalArgumentException("Negative shape parameter");
    double b = ( 1 /average - 1) * a;
    if ( b <= 0 )
        throw new IllegalArgumentException("Negative shape parameter");
    defineParameters( a, b);
}

// @return double average of the distribution.

public double average()
{
    return alpha1 / ( alpha1 + alpha2);
}

// Assigns new values to the parameters.
// This method assumes that the parameters have been already checked.

private void defineParameters ( double shape1, double shape2)
{
    alpha1 = shape1;
    alpha2 = shape2;
    norm = GammaFunction.logBeta( alpha1, alpha2);
    gamma1 = null;
    gamma2 = null;
    incompleteBetaFunction = null;
    return;
```

```
}
private void defineRandomGenerator ( )
{
    gamma1 = new GammaDistribution( alpha1, 1.0);
    gamma2 = new GammaDistribution( alpha2, 1.0);
    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 incompleteBetaFunction().value(x);
}

// This method was created in VisualAge.
// @return DhbIterations.IncompleteBetaFunction

private IncompleteBetaFunction incompleteBetaFunction()
{
    if ( incompleteBetaFunction == null )
        incompleteBetaFunction = new IncompleteBetaFunction( alpha1,
                                                             alpha2);
    return incompleteBetaFunction;
}

// @return double kurtosis of the distribution.

public double kurtosis( )
{
    double s = alpha1 + alpha2;
    return 3 * ( alpha1 + alpha2 + 1) * ( 2 * s * s +
                            ( alpha1 + alpha2 - 6) * alpha1 * alpha2)
                  / ( ( alpha1 + alpha2 + 2) *
                            ( alpha1 + alpha2 + 3) * alpha1 * alpha2)
                      - 3;
}

// @return java.lang.String name of the distribution.

public String name()
{
    return "Beta distribution";
```

```
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters()
{
    double[] answer = new double[2];
    answer[0] = alpha1;
    answer[1] = alpha2;
    return answer;
}

// @return double a random number distributed according to the receiver.

public double random ( )
{
    if ( gamma1 == null )
        defineRandomGenerator();
    double y1 = gamma1.random();
    return y1 / ( y1 + gamma2.random());
}

// @param a1 double

public void setAlpha1( double a1)
{
    defineParameters( a1, alpha2);
}

// @param a2 double

public void setAlpha2( double a2)
{
    defineParameters( alpha1, a2);
}

// @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( alpha1 + alpha2 + 1) * (alpha2 - alpha1)
                   / ( Math.sqrt( alpha1 * alpha2)
                                          * ( alpha1 + alpha2 + 2));
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    java.text.DecimalFormat fmt = new java.text.DecimalFormat("0.00000");
    sb.append("Beta distribution (");
    sb.append(fmt.format(alpha1));
    sb.append(',');
    sb.append(fmt.format(alpha2));
    sb.append(')');
    return sb.toString();
}

// @return double probability density function
// @param x double random variable

public double value( double x)
{
    return Math.exp( Math.log( x) * ( alpha1 - 1)
                        + Math.log( 1 - x) * ( alpha2 - 1) - norm);
}

// @return double variance of the distribution.

public double variance( )
{
    double s = alpha1 + alpha2;
    return alpha1 * alpha2 / ( s * s * ( alpha1 + alpha2 + 1));
}
}
```

D.2 Cauchy Distribution

Table D.2 shows the properties of the Cauchy distribution. Physicists use the Cauchy distribution under the name *Breit-Wigner* or *resonnance curve*. All moments of an order greater than zero are not defined as the corresponding integrals diverge.

TABLE D.2 Properties of the Cauchy distribution

Range of random variable	$]-\infty,_{\infty}[$
Probability density function	$P(x) = \frac{\beta}{\pi\left[(x-\mu)^2) + \beta^2\right]}$
Parameters	$-\infty < \mu < +\infty$
	$0 < \beta < +\infty$
Distribution function	$F(x) = \frac{1}{2} + \frac{1}{\pi}\arctan\left(\frac{x-\mu}{\beta}\right)$
Average	Undefined
Variance	Undefined
Skewness	Undefined
Kurtosis	Undefined

Figure D.2 shows the shapes taken by the Cauchy distribution for a few values of the parameters. These parameter are identical to the parameters of the normal distributions shown in figure 9.3 so that the reader can compare them.

D.2.1 Cauchy Distribution—Smalltalk Implementation

Listing D.3 shows the implementation of the Cauchy distribution in Smalltalk.

This implementation returns μ for the average although the average is not defined mathematically. Other moment related quantities are returning `nil`.

Listing D.3 Smalltalk implementation of the Cauchy distribution

Class `DhbCauchyDistribution`

Subclass of `DhbProbabilityDensity`

Instance variable names: `mu beta`

Class Methods

distributionName

```
^'Cauchy distribution'
```

fromHistogram: `aHistogram`

```
^self shape: aHistogram average
    scale: 4 * aHistogram variance
            / (Float pi * (aHistogram maximum squared +
                                    aHistogram minimum squared))
                    sqrt
```

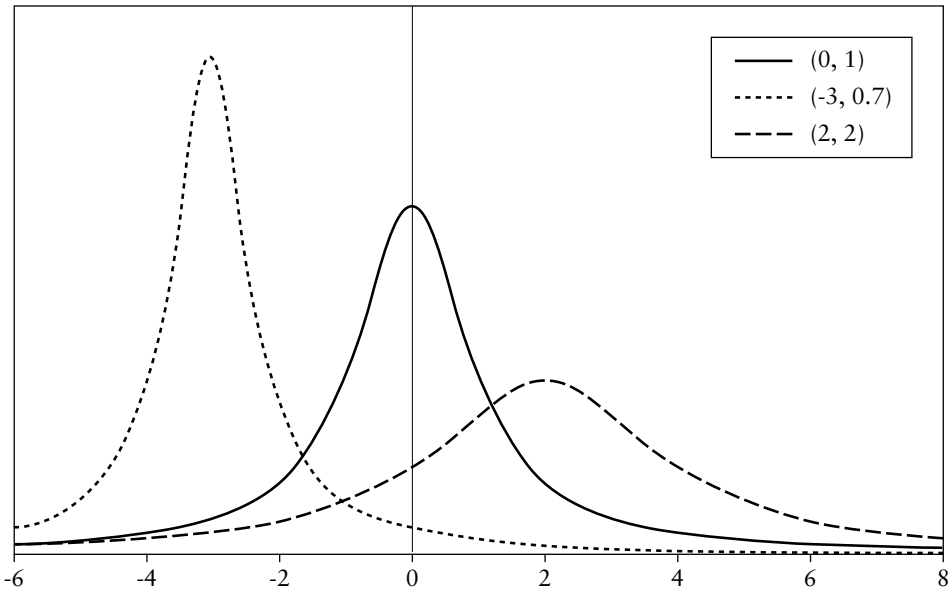

FIG. D.2 Cauchy distribution for a few parameters

new

```
^self shape: 0 scale: 1
```

shape: aNumber1 **scale:** aNumber2

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

Instance Methods

acceptanceBetween: aNumber1 **and:** aNumber2

```
^self privateAcceptanceBetween: aNumber1 and: aNumber2
```

average

```
^mu
```

changeParametersBy: aVector

```
mu := mu + ( aVector at: 1).
beta := beta + ( aVector at: 2).
```

distributionValue: aNumber

```
^(( aNumber - mu) / beta) arcTan / Float pi + (1 / 2)
```

initialize: aNumber1 **scale:** aNumber2

```
mu := aNumber1.
beta := aNumber2.
^self
```

parameters

```
^Array with: mu with: beta
```

privateInverseDistributionValue: aNumber

```
^( ( aNumber - (1 / 2)) * Float pi) tan * beta + mu
```

standardDeviation

```
^nil
```

value: aNumber

```
^beta / ( Float pi * ( beta squared + ( aNumber - mu) squared))
```

valueAndGradient: aNumber

```
| dp denominator |
dp := self value: aNumber.
denominator := 1 / ( ( aNumber - mu) squared + beta squared).
^Array with: dp
       with: ( DhbVector with: 2 * dp * ( aNumber - mu) *
                                                    denominator
                           with: dp * ( 1 / beta - ( 2 * beta *
                                                  denominator)))
```

variance

```
^nil
```

D.2.2 Cauchy Distribution—Java Implementation

Listing D.4 shows the implementation of the Cauchy distribution in Java.

This implementation returns μ for the average although the average is not defined mathematically. Other moment related quantities are returning the special value `Double.NaN`.

Listing D.4

Java implementation of the Cauchy distribution

```
package DhbStatistics;
```

```
import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Cauchy distribution

// @author Didier H. Besset

public final class CauchyDistribution
                            extends ProbabilityDensityFunction
{

    // Center of the distribution.

    private double mu;

    // Scale of the distribution.

    private double beta;


// Create an instance centered at 0 with width 1.

public CauchyDistribution ( )
{
    this( 0, 1);
}

// @param middle double middle point of the distribution.
// @param width double width of the distribution.

public CauchyDistribution ( double middle, double width)
{
    mu = middle;
    beta = width;
}

// 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 Histogram

public CauchyDistribution( Histogram h)
{
    this( h.average(),
          4 * h.variance() /Math.sqrt(Math.PI *( h.getMinimum()
                    * h.getMinimum() +h.getMaximum() * h.getMaximum()))
          );
```

```
}

// @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 Math.atan( ( x - mu) / beta) / Math.PI + 0.5;
}

// @return java.lang.String    name of the distribution

public String name ( )
{
    return "Cauchy distribution";
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters ( )
{
    double[] answer = new double[2];
    answer[0] = mu;
    answer[1] = beta;
    return answer;
}

// This method assumes that the range of the argument has been checked.
// @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)
{
    return Math.tan( (x - 0.5) * Math.PI) * beta + mu;
}
```

```
    // @param center double

    public void setBeta( double width)
    {
        beta = width;
    }

    // @param center double

    public void setMu( double center)
    {
        mu = center;
    }

    // @param p double[]    assigns the parameters

    public void setParameters( double[] params)
    {
        setMu( params[0]);
        setBeta( params[1]);
    }

    // @return NaN since the standard deviation of the distribution is
    //                                    not defined.

    public double standardDeviation( )
    {
        return Double.NaN;
    }

    // This method was created in VisualAge.
    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        java.text.DecimalFormat fmt = new java.text.DecimalFormat(
                                                        "####0.00000");
        sb.append("Cauchy distribution (");
        sb.append(fmt.format(mu));
        sb.append(',');
        sb.append(fmt.format(beta));
        sb.append(')');
        return sb.toString();
    }

    // @return double probability density function
```

TABLE D.3 Properties of the exponential distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \frac{1}{\beta} e^{-\frac{x}{\beta}}$
Parameters	$0 < \beta < +\infty$
Distribution function	$F(x) = 1 - e^{-\frac{x}{\beta}}$
Average	β
Variance	β^2
Skewness	2
Kurtosis	6

```
// @param x double random variable

public double value( double x)
{
    double dev = x - mu;
    return beta / ( Math.PI *( beta * beta + dev * dev));
}

// 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];
    answer[0] = value( x);
    double y = x - mu;
    double r = 1 / ( y * y + beta * beta);
    answer[1] = 2 * answer[0] * y * r;
    answer[2] = answer[0] * ( 1 / beta - 2 * beta * r);
    return answer;
}
}
```

D.3 Exponential Distribution

Table D.3 shows the properties of the exponential distribution.

The exponential distribution describes the distribution of the time of occurrence between independent random events with a constant probability of occurrence. It is

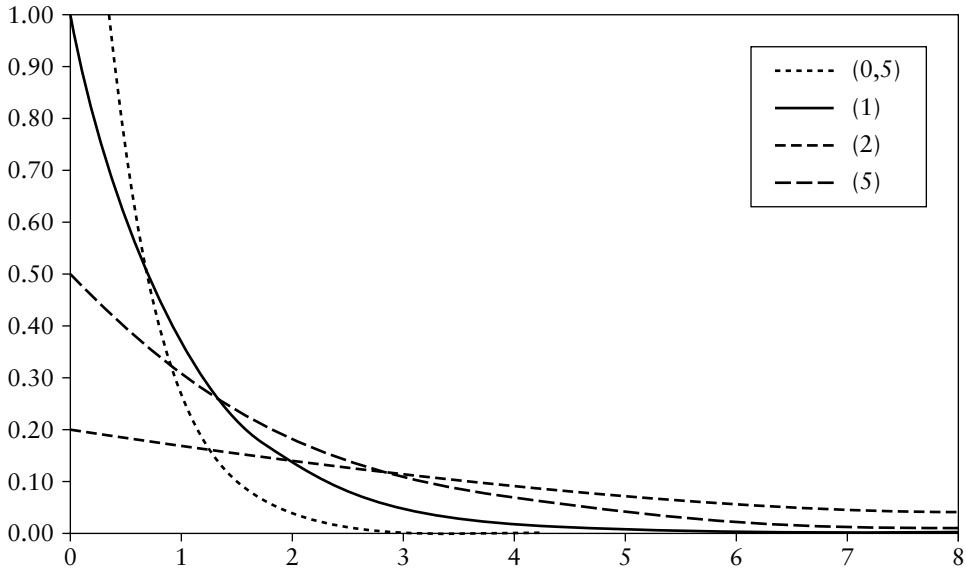

FIG. D.3 Exponential distribution for a few parameters

used in queuing theory and nuclear physics. Figure D.3 shows the shapes taken by the exponential distribution for a few values of the parameters.

D.3.1 Exponential Distribution—Smalltalk Implementation

Listing D.5 shows the implementation of the exponential distribution in Smalltalk.

Listing D.5 **Smalltalk implementation of the exponential distribution**

Class `DhbExponentialDistribution`
Subclass of `DhbProbabilityDensity`
Instance variable names: `beta`

Class Methods

distributionName

```
^'Exponential distribution'
```

fromHistogram: `aHistogram`

```
| mu |
aHistogram minimum < 0
```

```
        ifTrue: [ ^nil].
    mu := aHistogram average.
    ^mu > 0 ifTrue: [ self scale: aHistogram average]
            ifFalse:[ nil]
```

new

```
^super new initialize: 1
```

scale: aNumber

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

Instance Methods

acceptanceBetween: aNumber1 **and:** aNumber2

```
^self privateAcceptanceBetween: aNumber1 and: aNumber2
```

average

```
^beta
```

changeParametersBy: aVector

```
beta := beta + ( aVector at: 1).
```

distributionValue: aNumber

```
^[1 - ( ( aNumber / beta negated) exp)]
        when: ExAll do: [ :signal | signal exitWith: 0]
```

initialize: aNumber

```
aNumber > 0
    ifFalse: [ self error: 'Illegal distribution parameters'].
beta := aNumber.
^self
```

kurtosis

```
^6
```

parameters

```
^Array with: beta
```

privateInverseDistributionValue: aNumber

```
^(1 - aNumber) ln negated * beta
```

random

```
^DhbMitchellMooreGenerator new floatValue ln * beta negated
```

skewness

```
^2
```

standardDeviation

```
^beta
```

value: aNumber

```
^[ ( aNumber / beta) negated exp / beta]
        when: ExAll do: [ :signal | signal exitWith: 0]
```

valueAndGradient: aNumber

```
| dp |
dp := self value: aNumber.
^Array with: dp
       with: ( DhbVector with: ( aNumber / beta - 1) * dp / beta)
```

D.3.2 Exponential Distribution—Java Implementation

Listing D.6 shows the implementation of the exponential distribution in Java.

Listing D.6

Java implementation of the exponential distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Exponential distribution.

// @author Didier H. Besset

public final class ExponentialDistribution
                              extends ProbabilityDensityFunction
{

    // Exponential term.

    private double beta;
```

```
// General constructor method.
// @param exponential fall-off
// @exception java.lang.IllegalArgumentException
//                          if the fall-off is non-positive.

public ExponentialDistribution ( double fallOff)
                                        throws IllegalArgumentException
{
    if ( fallOff <= 0 )
        throw new IllegalArgumentException(
                              "Exponential fall-off must be positive");
    beta = fallOff;
}

// 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 ExponentialDistribution( Histogram h)
                                        throws IllegalArgumentException
{
    if ( h.getMinimum() < 0 )
        throw new IllegalArgumentException(
    "Exponential distribution is only defined for non-negative values");
    double average = h.average();
    if ( h.average() < 0 )
        throw new IllegalArgumentException(
        "Exponential distribution is only defined for positive scale");
    setScale( average);
}

// @return double average of the distribution.

public double average ( )
{
    return beta;
}

// 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 1 - Math.exp( -x / beta);
}

// @return double kurtosis of the distribution.

public double kurtosis ( )
{
    return 6;
}

// @return java.lang.String     name of the distribution

public String name ( )
{
    return "Exponential distribution";
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters ( )
{
    double[] answer = new double[1];
    answer[0] = beta;
    return answer;
}

// This method assumes that the range of the argument has been checked.
// @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)
{
    return -Math.log( 1 - x) * beta;
}

// @return double a random number distributed according to the receiver.

public double random( )
{
    return -beta * Math.log(generator().nextDouble());
}
```

```
    // @param p double[]    assigns the parameters

    public void setParameters( double[] params)
    {
        setScale( params[0]);
    }

    // @param falloff double

    public void setScale( double falloff)
    {
        beta = falloff;
    }

    // @return double skewness of the distribution.

    public double skewness( )
    {
        return 2;
    }

    // @return double standard deviation of the distribution

    public double standardDeviation( )
    {
        return beta;
    }

    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        java.text.DecimalFormat fmt = new java.text.DecimalFormat(
                                                              "###0.00000");
        sb.append("Exponential distribution (");
        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)
    {
```

TABLE D.4 Properties of the Fisher-Tippett distribution

Range of random variable	$]-\infty, +\infty[$
Probability density function	$P(x) = \frac{1}{\beta} e^{-\frac{x-\alpha}{\beta} - e^{\frac{x-\alpha}{\beta}}}$
Parameters	$-\infty < \alpha < +\infty$
	$0 < \beta < +\infty$
Distribution function	$F(x) = e^{-e^{-\frac{x-\alpha}{\beta}}}$
Average	$\alpha + \gamma\beta$
Variance	$\frac{\pi\beta}{\sqrt{6}}$
Skewness	1.3
Kurtosis	2.4

```
    return Math.exp( -x / beta) / beta;
}

// 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[2];
    answer[0] = value(x);
    answer[1] = ( x / beta - 1) * answer[0] / beta;
    return answer;
}
}
```

D.4 Fisher-Tippett Distribution

Table D.4 shows the properties of the Fisher-Tippett distribution. In this table, $\gamma = 0.5772156649\ldots$ is the Euler constant.

The Fisher-Tippett distribution describes the distribution of extreme values. Figure D.4 shows the shapes taken by the Fisher-Tippett distribution for a few values of the parameters. These parameter are identical to the those of the normal distributions shown in Figure 9.3 so that the reader can compare them.

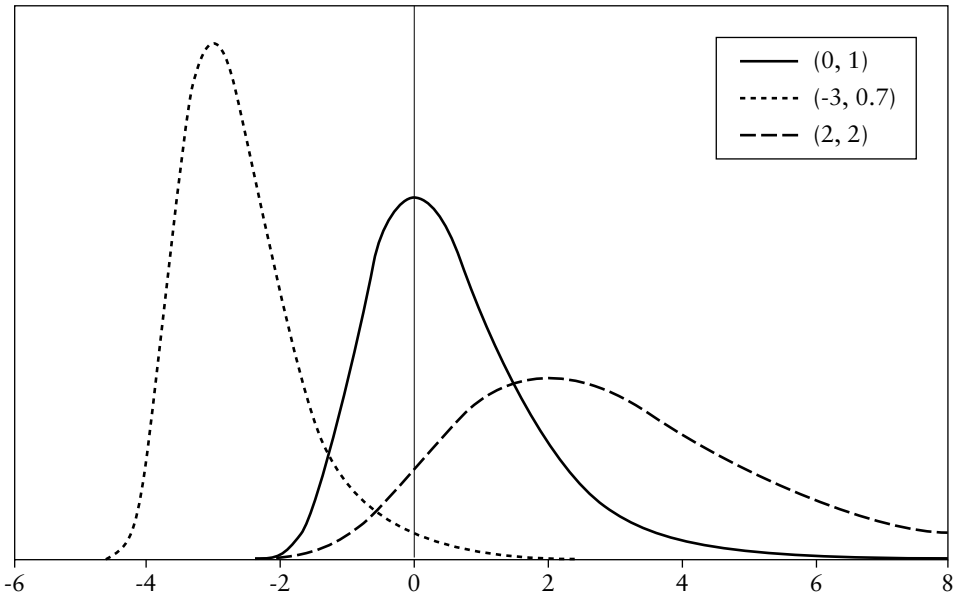

FIG. D.4 Fisher-Tippett distribution for a few parameters

D.4.1 Fisher–Tippett Distribution—Smalltalk Implementation

Listing D.7 shows the implementation of the Fisher-Tippett distribution in Smalltalk.

Listing D.7 Smalltalk implementation of the Fisher-Tippett distribution

Class `DhbFisherTippettDistribution`
Subclass of `DhbProbabilityDensity`
Instance variable names: `alpha beta`

Class Methods

distributionName

```
^'Fisher-Tippett distribution'
```

fromHistogram: `aHistogram`

```
| beta |
beta := aHistogram standardDeviation.
beta = 0 ifTrue: [^nil].
beta := beta * (6 sqrt / Float pi).
^self shape: aHistogram average - (0.5772156649 * beta) scale:
```

```
                                                                    beta
```

new

```
    ^self shape: 0 scale: 1
```

shape: aNumber1 **scale:** aNumber2

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

Instance Methods

average

```
    ^0.577256649 * beta + alpha
```

changeParametersBy: aVector

```
    alpha := alpha + ( aVector at: 1).
    beta := beta + ( aVector at: 2).
```

distributionValue: aNumber

```
    | arg |
    arg := ( aNumber - alpha) / beta.
    arg := arg < DhbFloatingPointMachine new largestExponentArgument
                                                                    negated
                    ifTrue: [ ^0]
                    ifFalse:[arg negated exp].
    ^arg > DhbFloatingPointMachine new largestExponentArgument
                                                          ifTrue: [ 1]
                        ifFalse:[ arg negated exp]
```

initialize: aNumber1 **scale:** aNumber2

```
    aNumber2 > 0
        ifFalse: [ self error: 'Illegal distribution parameters'].
    alpha := aNumber1.
    beta := aNumber2.
    ^self
```

integralFrom: aNumber1 **to:** aNumber2

```
    ^( DhbRombergIntegrator new: self from: aNumber1 to: aNumber2)
                                                              evaluate
```

integralUpTo: aNumber

```
    ^( DhbRombergIntegrator new:
            [ :x | x = 0 ifTrue: [ 0] ifFalse: [ ( self value: 1 / x)
```

```
                                                        / x squared] ]
          from: 1 / aNumber to: 0) evaluate
```

kurtosis

```
^2.4
```

parameters

```
^Array with: alpha with: beta
```

random

```
| t |
[ t := DhbMitchellMooreGenerator new floatValue ln negated.
  t > 0] whileFalse: [].
^t ln negated * beta + alpha
```

skewness

```
^1.3
```

standardDeviation

```
^Float pi * beta / ( 6 sqrt)
```

value: aNumber

```
| arg |
arg := ( aNumber - alpha) / beta.
arg := arg > DhbFloatingPointMachine new largestExponentArgument
                                                        ifTrue: [ ^0]
                    ifFalse:[arg negated exp + arg].
^arg > DhbFloatingPointMachine new largestExponentArgument
                                                         ifTrue: [ 0]
                    ifFalse:[ arg negated exp / beta]
```

valueAndGradient: aNumber

```
| dp dy y|
dp := self value: aNumber.
y := ( aNumber - alpha) / beta.
dy := ( y negated exp - 1).
^Array with: dp
       with: ( DhbVector with: dy * dp / beta negated
                       with: dp * ( y * dy + 1) / beta negated)
```

D.4.2 Fisher-Tippett Distribution—Java Implementation

Listing D.8 shows the implementation of the Fisher-Tippett distribution in Java.

Listing D.8 Java implementation of the Fisher-Tippett distribution

```
package DhbStatistics;

import DhbFunctionEvaluation.DhbMath;
import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Fisher-Tippett distribution

// @author Didier H. Besset

public final class FisherTippettDistribution
                            extends ProbabilityDensityFunction
{

// Center of the distribution.

    protected double alpha;

    // Scale parameter of the distribution.

    private double beta;

// Constructor method
// @param center double
// @param scale double
// @exception java.lang.IllegalArgumentException if the scale parameter is non-positive.

public FisherTippettDistribution( double center, double scale)
                                    throws IllegalArgumentException
{
    if ( scale <= 0 )
        throw new IllegalArgumentException("Scale parameter must
        be positive");
    alpha = center;
    beta = 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 FisherTippettDistribution( Histogram h)
                    throws IllegalAccessException
{
    double beta = h.standardDeviation();
    if ( beta < 0 )
        throw new IllegalArgumentException("Histogram has vanishing
                                                standard deviation");
    beta *= Math.sqrt(6)/Math.PI;
    defineParameters( h.average() - 0.5772156649 * beta, beta);
}

// @return double average of the distribution.

public double average()
{
    return 0.5772156649 * beta + alpha;
}

// @param center double
// @param scale double

public void defineParameters ( double center, double scale)
{
    alpha = center;
    beta = scale;
}

// 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)
{
    double y = ( x - alpha) / beta;
    if ( y < -DhbMath.getLargestExponentialArgument() )
        return 0;
    y = Math.exp( -y);
    if ( y > DhbMath.getLargestExponentialArgument() )
        return 1;
    return Math.exp( -y);
}
```

```
// @return double kurtosis of the distribution.

public double kurtosis( )
{
    return 2.4;
}

// @return java.lang.String    name of the distribution

public String name ( )
{
    return "Fisher-Tippett 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 t;
    while ( ( t = -Math.log( generator().nextDouble())) == 0 );
    return alpha - beta * Math.log( t);
}

// @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 1.3;
```

```
}

// @return double standard deviation of the distribution

public double standardDeviation( )
{
    return Math.PI * beta / Math.sqrt( 6);
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    java.text.DecimalFormat fmt = new java.text.DecimalFormat
                                                ("####0.00000");
    sb.append("Fisher-Tippett 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)
{
    double y = ( x - alpha) / beta;
    if ( y > DhbMath.getLargestExponentialArgument() )
        return 0;
    y += Math.exp( -y);
    if ( y > DhbMath.getLargestExponentialArgument() )
        return 0;
    return Math.exp( -y) / beta;
}

// 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];
```

TABLE D.5 Properties of the Laplace distribution

Range of random variable	$]-\infty, +\infty[$
Probability density function	$P(x) = \frac{1}{2\beta} e^{-\frac{\lvert x-\alpha \rvert}{\beta}}$
Parameters	$-\infty < \alpha < +\infty$
	$0 < \beta < +\infty$
Distribution function	$F(x) = \begin{cases} \frac{1}{2} e^{-\frac{\alpha - x}{\beta}} & \text{for } x < \alpha \\ 1 - \frac{1}{2} e^{-\frac{x-\alpha}{\beta}} & \text{for } x \geq \alpha \end{cases}$
Average	$\alpha + \beta$
Variance	$2\beta^2$
Skewness	0
Kurtosis	3

```
        answer[0] = value( x);
        double y = ( x - alpha) / beta;
        double dy = Math.exp( -y) - 1;
        double r = -1 / beta;
        answer[1] = dy * answer[0] * r;
        answer[2] = answer[0] * ( y * dy + 1) * r;
        return answer;
    }
    }
```

D.5 Laplace Distribution

Table D.5 shows the properties of the Laplace distribution. The Laplace distribution is an adhoc distribution made of two exponential distributions, one on each side of the peak. Figure D.5 shows the shapes taken by the Laplace distribution for a few values of the parameters. These parameter are identical to those of the normal distributions shown in Figure 9.3 so that the reader can compare them.

D.5.1 Laplace Distribution—Smalltalk Implementation

Listing D.9 shows the implementation of the Laplace distribution in Smalltalk.

Listing D.9 **Smalltalk implementation of the Laplace distribution**

Class	`DhbLaplaceDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`mu beta`

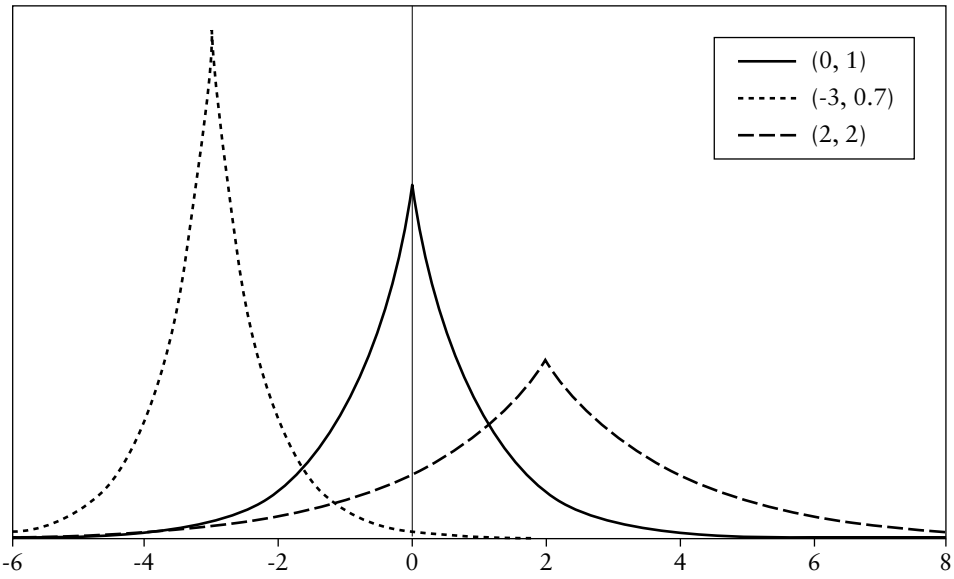

FIG. D.5 Laplace distribution for a few parameters

Class Methods

distributionName

```
^'Laplace distribution'
```

fromHistogram: `aHistogram`

```
^self shape: aHistogram average scale: (aHistogram variance / 2)
                                                                sqrt
```

new

```
^self shape: 0 scale: 1
```

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

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

Instance Methods

average

```
^mu
```

changeParametersBy: `aVector`

```
mu := mu + ( aVector at: 1).
```

```
    beta := beta + ( aVector at: 2).
```

distributionValue: aNumber

```
    ^aNumber > mu
        ifTrue: [ 1 - ( ( ( aNumber - mu) / beta) negated exp / 2)]
        ifFalse:[ ( ( ( aNumber - mu) / beta) exp / 2)]
```

initialize: aNumber1 **scale:** aNumber2

```
    mu := aNumber1.
    beta := aNumber2.
    ^self
```

kurtosis

```
    ^3
```

parameters

```
    ^Array with: mu with: beta
```

random

```
    | r |
    r := DhbMitchellMooreGenerator new floatValue ln * beta negated.
    ^DhbMitchellMooreGenerator new floatValue > 0.5
        ifTrue: [ mu + r]
        ifFalse:[ mu - r]
```

skewness

```
    ^0
```

standardDeviation

```
    ^beta * ( 2 sqrt)
```

value: aNumber

```
    ^( ( aNumber - mu) / beta) abs negated exp / ( 2 * beta)
```

valueAndGradient: aNumber

```
    | dp |
    dp := self value: aNumber.
    ^Array  with: dp
            with: ( DhbVector with: ( aNumber - mu) sign * dp / beta
                              with: ( ( ( aNumber - mu) abs / beta -
                                                          1) * dp / beta))
```

D.5.2 Laplace Distribution—Java Implementation

Listing D.10 shows the implementation of the Laplace distribution in Java.

Listing D.10 Java implementation of the Laplace distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Laplace distribution.

// @author Didier H. Besset

public final class LaplaceDistribution
                                        extends ProbabilityDensityFunction
{

    // Average of the distribution.

    private double mu;

    // Scale of the distribution.

    private double beta;

// Constructor method.
// @param center double
// @param scale double
// @exception java.lang.IllegalArgumentException
//                  when the scale parameter is non-positive

public LaplaceDistribution( double center, double scale)
                                        throws IllegalArgumentException
{
    if ( scale <= 0 )
        throw new IllegalArgumentException(
                                "Scale parameter must be positive");
    mu = center;
    beta = 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

public LaplaceDistribution( Histogram h)
{
    this( h.average(), Math.sqrt( 0.5 * h.variance()));
}

// @return double average of the distribution.

public double average()
{
    return mu;
}

// @param center double
// @param scale double

public void defineParameters ( double center, double scale)
{
    mu = center;
    beta = scale;
}

// 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 x > mu
            ? 1 - Math.exp( -( x - mu) / beta) / 2
            : Math.exp( -( x - mu) / beta) / 2;
}

// @return double kurtosis of the distribution.

public double kurtosis ( )
{
    return 3;
}

// @return java.lang.String name of the distribution.

public String name() {
    return null;
```

```
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters()
{
    double[] answer = new double[2];
    answer[0] = mu;
    answer[1] = beta;
    return answer;
}

// This method assumes that the range of the argument has been checked.
// @return double the value for which the distribution function
//                                  is equal to x.
// @param x double value of the distribution function.

public double privateInverseDistributionValue ( double x)
{
    return x < 0.5
            ? mu + beta * Math.log( 2 * x)
            : mu - beta * Math.log( 2 - 2 * x);
}

// @return double a random number distributed according to the receiver.

public double random( )
{
    double r = -beta * Math.log(generator().nextDouble());
    return generator().nextDouble() > 0.5 ? mu + r : mu - r;
}

// @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 0;
}
```

```
// @return double standard deviation of the distribution

public double standardDeviation( )
{
    return beta / Math.sqrt( 2);
}

// @return java.lang.String

public String toString()
{
    StringBuffer sb = new StringBuffer();
    java.text.DecimalFormat fmt = new java.text.DecimalFormat(
                                                    "####0.00000");
    sb.append("Laplace distribution (");
    sb.append(fmt.format(mu));
    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 Math.exp( -Math.abs( x - mu) / beta) / ( 2 * beta);
}

// 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];
    answer[0] = value( x);
    double y = x - mu;
    if ( y >= 0 )
    {
        answer[1] = answer[0] / beta;
        answer[2] = (y / beta - 1) * answer[0] / beta;
    }
    else
```

TABLE D.6 Properties of the log normal distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \frac{1}{x\sqrt{2\pi\sigma^2}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$
Parameters	$-\infty < \mu < +\infty$
	$0 < \sigma < +\infty$
Distribution function	No closed expression
Average	$e^{\mu + \frac{\sigma^2}{2}}$
Variance	$e^{2\mu+\sigma^2}\left(e^{\sigma^2} - 1\right)$
Skewness	$\sqrt{e^{\sigma^2} - 1}\left(e^{\sigma^2} + 2\right)$
Kurtosis	$e^{4\sigma^2} + 2e^{3\sigma^2} + 3e^{2\sigma^2} - 6$

```
        {
            answer[1] = - answer[0] / beta;
            answer[2] = - (y / beta + 1) * answer[0] / beta;
        }
        return answer;
    }
    }
```

D.6 Log Normal Distribution

Table D.6 shows the properties of the log normal distribution. The log normal distribution is used to describe quantities that are the product of a large number of other quantities. It is an ad hoc distribution whose shape is similar to that of gamma distributions with $\alpha > 1$. Figure D.6 shows the shapes taken by the log normal distribution for a few values of the parameters.

D.6.1 Log normal Distribution—Smalltalk Implementation

Listing D.11 shows the implementation of the log normal distribution in Smalltalk.

Listing D.11 Smalltalk implementation of the log normal distribution

Class	`DhbLogNormalDistribution`
Subclass of	`DhbProbabilityDensityWithUnknownDistribution`
Instance variable names:	`normalDistribution`

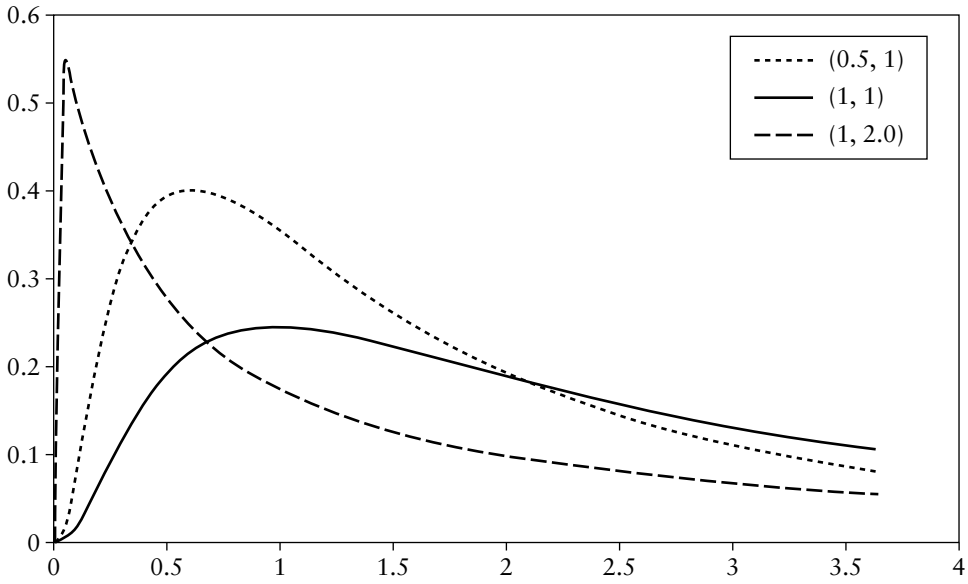

FIG. D.6 Log normal distribution for a few parameters

Class Methods

distributionName

```
^'Log normal distribution'
```

fromHistogram: `aHistogram`

```
| average variance sigma2 |
aHistogram minimum < 0
    ifTrue: [ ^nil].
average := aHistogram average.
average > 0
    ifFalse: [ ^nil].
variance := aHistogram variance.
sigma2 := ( variance / average squared + 1) ln.
sigma2 > 0
    ifFalse: [ ^nil].
^self new: ( average ln - (sigma2 * 0.5)) sigma: sigma2 sqrt
```

new

```
^self new: 0 sigma: 1
```

new: aNumber1 **sigma:** aNumber2

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

Instance Methods

average

```
^( normalDistribution variance * 0.5 + normalDistribution
                                                    average) exp
```

changeParametersBy: aVector

```
normalDistribution changeParametersBy: aVector.
```

fourthCentralMoment

```
| y x |
y := normalDistribution average exp.
x := normalDistribution variance exp.
^( y squared squared) * ( x squared)
    * ( ( ( x squared * x - 4) * ( x squared) + 6) * x - 3)
```

initialize: aNumber1 **sigma:** aNumber2

```
normalDistribution := DhbNormalDistribution new: aNumber1 sigma:
                                                        aNumber2.
^self
```

kurtosis

```
| x |
x := normalDistribution variance exp.
^( ( x + 2) * x + 3) * ( x squared) - 6
```

parameters

```
^normalDistribution parameters
```

random

```
^normalDistribution random exp
```

skewness

```
| x |
x := normalDistribution variance exp.
^(x - 1) sqrt * (x + 2)
```

thirdCentralMoment

```
| y x |
y := normalDistribution average exp.
x := normalDistribution variance exp.
^( y squared * y) * ( x raisedTo: (3/2))
    * ( ( x squared negated + 3) * x - 2)
```

value: aNumber

```
^aNumber > 0
    ifTrue: [ ( normalDistribution value: aNumber ln) / aNumber]
    ifFalse:[ 0]
```

variance

```
^( normalDistribution average * 2 + normalDistribution variance)
                        exp * ( normalDistribution variance exp - 1)
```

D.6.2 Log normal Distribution—Java Implementation

Listing D.12 shows the implementation of the log normal distribution in Java.

Listing D.12 Java implementation of the log normal distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Log normal distribution

// @author Didier H. Besset

public final class LogNormalDistribution
                    extends ProbabilityDensityWithUnknownDistribution
{

    // Normal distribution with the same parameters.

    private NormalDistribution normalDistr;

// Defines a Log Normal distribution with known parameters.
// @param mu double
// @param sigma double
// @exception java.lang.IllegalArgumentException
```

```
//                          when sigma is non-positive

public LogNormalDistribution ( double mu, double sigma)
                                    throws IllegalArgumentException
{
    normalDistr = new NormalDistribution( mu, sigma);
}

// 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 LogNormalDistribution( Histogram h)
                                    throws IllegalArgumentException
{
    if ( h.getMinimum() < 0 )
        throw new IllegalArgumentException(
    "Log normal distribution is only defined for non-negative values");
    double average = h.average();
    if ( average <= 0 )
        throw new IllegalArgumentException(
    "Log normal distribution is only defined for positive average");
    double variance = h.variance();
    double sigma2 = Math.log( variance / (average * average) + 1);
    if ( sigma2 <= 0 )
        throw new IllegalArgumentException(
    "Log normal distribution is only defined for positive sigma");
    normalDistr = new NormalDistribution(  Math.log( average),
                                                Math.sqrt( sigma2));
}

// @return double average of the distribution.

public double average ( )
{
    return Math.exp( normalDistr.variance() / 2
                                            + normalDistr.average());
}

// @return double    the lowest value of the random variable

protected double lowValue()
{
```

```
    return 0;
}

// @return java.lang.String     name of the distribution

public String name ( )
{
    return "Log normal distribution";
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters ( )
{
    return normalDistr.parameters();
}

// @return double a random number distributed according to the receiver.

public double random( )
{
    return Math.exp( normalDistr.random());
}

// @param m double

public void setMu( double mu)
{
    normalDistr.setAverage( mu);
}

// @param p double[]    assigns the parameters

public void setParameters( double[] params)
{
    setMu( params[0]);
    setSigma( params[1]);
}

// @param sigma double

public void setSigma( double sigma)
{
    normalDistr.setStandardDeviation( sigma);
}
```

```
    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        java.text.DecimalFormat fmt = new java.text.DecimalFormat(
                                                        "####0.00000");
        sb.append("Log normal distribution (");
        sb.append(fmt.format(normalDistr.average()));
        sb.append(',');
        sb.append(fmt.format(normalDistr.standardDeviation()));
        sb.append(')');
        return sb.toString();
    }

    // @return double probability density function
    // @param x double random variable

    public double value ( double x)
    {
        return x > 0 ? normalDistr.value( Math.log(x)) / x : 0;
    }

    // @return double variance of the distribution.

    public double variance ( )
    {
        double variance = normalDistr.variance();
        return Math.exp( variance + 2 * normalDistr.average())
                                          * ( Math.exp( variance) - 1);
    }
    }
```

D.7 Triangular Distribution

Table D.7 shows the properties of the triangular distribution. The triangular distribution is ad hoc distribution used when a variable is limited to an interval.

D.7.1 Triangular Distribution—Smalltalk Implementation

Listing D.13 shows the implementation of the triangular distribution in Smalltalk.

TABLE D.7 Properties of the triangular distribution

Range of random variable	$[a, b]$
Probability density function	$P(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)} & \text{if } a \le x \le c, \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{if } c \le x \le b \end{cases}$
Parameters	$-\infty < a \le c \le b < +\infty$ $a < b$
Distribution function	$F(x) = \begin{cases} \frac{(x-a)^2}{(b-a)(c-a)} & \text{if } a \le x \le c, \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{if } c \le x \le b \end{cases}$
Average	$\frac{a+b+c}{3}$
Variance	$\frac{a^2+b^2+c^2-ab-ac-bc}{18}$
Skewness	$\frac{a^3+b^3+c^3}{135} + \cdots$
Kurtosis	$\ldots$

Listing D.13 Smalltalk implementation of the triangular distribution

Class `DhbTriangularDistribution`
Subclass of `DhbProbabilityDensity`
Instance variable names: `lowLimit highLimit peak`

Class Methods

distributionName

```
^'Triangular distribution'
```

fromHistogram: `aHistogram`

```
| b c|
b := aHistogram standardDeviation * 1.73205080756888
```

new

```
^self new: (1 / 2) from: 0 to: 1
```

new: `aNumber1` **from:** `aNumber2` **to:**

```
aNumber3
^super new initialize: aNumber1 from: aNumber2 to: aNumber3
```

Instance Methods

acceptanceBetween: aNumber1 **and:** aNumber2

```
^self privateAcceptanceBetween: aNumber1 and: aNumber2
```

average

```
^(lowLimit + peak + highLimit) / 3
```

changeParametersBy: aVector

```
lowLimit := lowLimit + ( aVector at: 1).
highLimit := highLimit + ( aVector at: 2).
peak := peak + ( aVector at: 3).
```

distributionValue: aNumber

```
| norm |
^( aNumber between: lowLimit and: highLimit)
    ifTrue: [ aNumber < peak
                    ifTrue: [ norm := ( highLimit - lowLimit) * (
                                                peak - lowLimit).
                                  ( aNumber - lowLimit) squared /
                                                            norm
                                ]
                    ifFalse:[ aNumber > peak
                                    ifTrue: [ norm := ( highLimit
                                - lowLimit) * ( highLimit - peak).
                                                 1 - ( (
                                        highLimit - aNumber) squared / norm)
                                                ]
                                    ifFalse:[ ( peak - lowLimit)
                                        / ( highLimit - lowLimit)]
                                ]
                ]
    ifFalse:[ 0]
```

initialize: aNumber1 **from:** aNumber2 **to:**

aNumber3

```
( aNumber2 < aNumber3 and: [ aNumber1 between: aNumber2 and:
                                                        aNumber3])
    ifFalse: [ self error: 'Illegal distribution parameters'].
peak := aNumber1.
lowLimit := aNumber2.
highLimit := aNumber3.
^self
```

inverseAcceptanceAfterPeak: aNumber

```
^ highLimit - ( ( ( 1 - aNumber) * ( highLimit - lowLimit) * (
                                        highLimit - peak)) sqrt)
```

inverseAcceptanceBeforePeak: aNumber

```
^ ( aNumber * ( highLimit - lowLimit) * ( peak - lowLimit)) sqrt
                                                        + lowLimit
```

kurtosis

```
^(-6/10)
```

parameters

```
^Array with: lowLimit with: highLimit with: peak
```

privateInverseDistributionValue: aNumber

```
^( peak - lowLimit) / ( highLimit - lowLimit) > aNumber
        ifTrue: [ self inverseAcceptanceBeforePeak: aNumber]
        ifFalse: [ self inverseAcceptanceAfterPeak: aNumber]
```

skewness

```
  ^(((lowLimit squared * lowLimit + ( peak squared * peak) + (
                            highLimit squared * highLimit) ) / 135)
  -(((lowLimit squared * peak) + (lowLimit squared * highLimit) +
(peak squared * lowLimit) + (peak squared * highLimit) + (highLimit
squared * lowLimit) + (highLimit squared * peak))/90)
  +( 2 * lowLimit * peak * highLimit / 45)) / ( self
                              standardDeviation raisedToInteger: 3)
```

value: aNumber

```
| norm |
^( aNumber between: lowLimit and: highLimit)
    ifTrue: [ aNumber < peak
                    ifTrue: [ norm := ( highLimit - lowLimit) * (
                                                peak - lowLimit).
                                 2 * ( aNumber - lowLimit) / norm
                               ]
                    ifFalse:[ aNumber > peak
                                    ifTrue: [ norm := ( highLimit
                                - lowLimit) * ( highLimit - peak).
                                                 2 * ( highLimit
                                                 - aNumber) / norm
                                               ]
```

```
                                        ifFalse:[ 2 / ( highLimit -
                                                        lowLimit)]
                                    ]
                ]
    ifFalse:[ 0]
```

variance

```
  ^(lowLimit squared + peak squared + highLimit squared - (
lowLimit * peak) - ( lowLimit * highLimit) - ( peak * highLimit)) /
18
```

D.7.2 Triangular Distribution—Java Implementation

Listing D.14 shows the implementation of the triangular distribution in Java.

Listing D.14 Java implementation of the triangular distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Triangular distribution.

// @author Didier H. Besset

public final class TriangularDistribution
                                        extends ProbabilityDensityFunction
{

    // Low limit.

    private double a;

    // High limit.

    private double b;

    // peak location.

    private double c;


// Constructor method.
```

```
// @param low double    low limit
// @param high double    high limit
// @param peak double    peak of the distribution
// @exception java.lang.IllegalArgumentException
//                if the limits are inverted or
//                if the peak is outside the limits.

public TriangularDistribution ( double low, double high, double peak)
                                  throws IllegalArgumentException
{
    if ( low >= high )
        throw new IllegalArgumentException(
                    "Limits of distribution are equal or reversed");
    if ( peak < low || peak > high )
        throw new IllegalArgumentException(
                    "Peak of distribution lies outside the limits");
    a = low;
    b = high;
    c = peak;
}

// 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 TriangularDistribution( Histogram h)
{
    b = h.standardDeviation() * 1.73205080756888; // sqrt(12)/2
    c = h.average();
    a = c - b;
    b += c;
}

// @return double average of the distribution.

public double average()
{
    return ( a + b + c) / 3;
}

// Returns the probability of finding a random variable smaller
// than or equal to x.
// @return integral of the probability density function from a to x.
// @param x double upper limit of integral.
```

```
public double distributionValue ( double x)
{
    if ( x < a )
        return 0;
    else if ( x < c )
        return ( x - a) * ( x - a) / ( ( b - a) * ( c - a));
    else if ( x < b )
        return 1 - ( b - x) * ( b - x) / ( ( b - a) * ( b - c));
    else
        return 1;
}

// @return java.lang.String    name of the distribution

public String name()
{
    return "Triangular distribution";
}

// @return double[] an array containing the parameters of
//                                  the distribution.

public double[] parameters()
{
    double[] answer = new double[3];
    answer[0] = a;
    answer[1] = b;
    answer[2] = c;
    return answer;
}

// This method assumes that the range of the argument has been checked.
// @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)
{
    return ( x < ( c - a) / ( b - a) )
                    ? Math.sqrt( x * ( b - a) * ( c - a)) + a
                    : b - Math.sqrt( (1 - x) * (b - a) * ( b - c));
}

// @param p double[]   assigns the parameters

public void setParameters( double[] params)
```

```
{
    a = params[0];
    b = params[1];
    c = params[2];
}

// @return double probability density function
// @param x double random variable

public double value( double x)
{
    if ( x < a )
        return 0;
    else if ( x < c )
        return 2 * (x - a) / ( (b - a) * ( c - a));
    else if ( x < b )
        return 2 * (b - x) / ( (b - a) * ( b - c));
    else
        return 0;
}

// @return double variance of the distribution

public double variance()
{
    return ( a * a + b * b + c * c - a * b - a * c - b * c) / 18;
}
}
```

D.8 Uniform Distribution

Table D.8 shows the properties of the uniform distribution. The uniform distribution is another ad hoc distribution used when a variable is limited to an interval.

D.8.1 Uniform Distribution—Smalltalk Implementation

Listing D.15 shows the implementation of the uniform distribution in Smalltalk.

Listing D.15 Smalltalk implementation of the uniform distribution

Class	`DhbUniformDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`lowLimit highLimit`

TABLE D.8 Properties of the uniform distribution

Range of random variable	$[a, b]$
Probability density function	$P(x) = \frac{1}{b-a}$
Parameters	$-\infty < a < \text{¡} + \infty$
Distribution function	$F(x) = \frac{x-a}{b-a}$
Average	$\frac{a+b}{2}$
Variance	$\frac{(b-a)^2}{12}$
Skewness	0
Kurtosis	−1.2

Class Methods

distributionName

```
^'Uniform distribution'
```

from: aNumber1 **to:** aNumber2

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

fromHistogram: aHistogram

```
| b c|
b := aHistogram standardDeviation * 1.73205080756888
```

new

```
^self from: 0 to: 1
```

Instance Methods

acceptanceBetween: aNumber1 **and:** aNumber2

```
^self privateAcceptanceBetween: aNumber1 and: aNumber2
```

average

```
^( highLimit + lowLimit) / 2
```

changeParametersBy: aVector

```
lowLimit := lowLimit + ( aVector at: 1).
highLimit := highLimit + ( aVector at: 2).
```

distributionValue: aNumber

```
aNumber < lowLimit
    ifTrue: [ ^0].
^aNumber < highLimit
    ifTrue: [ (aNumber - lowLimit) / ( highLimit - lowLimit)]
    ifFalse:[ 1]
```

initialize: aNumber1 **to:** aNumber2

```
aNumber1 < aNumber2
    ifFalse: [ self error: 'Illegal distribution parameters'].
lowLimit := aNumber1.
highLimit := aNumber2.
^self
```

kurtosis

```
^-12 / 10
```

parameters

```
^Array with: lowLimit with: highLimit
```

privateInverseDistributionValue: aNumber

```
^(highLimit - lowLimit) * aNumber + lowLimit
```

skewness

```
^0
```

standardDeviation

```
^( highLimit - lowLimit) / 3.46410161513775
```

value: aNumber

```
^( aNumber between: lowLimit and: highLimit)
    ifTrue: [ 1/( highLimit - lowLimit)]
    ifFalse:[ 0]
```

variance

```
^( highLimit - lowLimit) squared / 12
```

D.8.2 Uniform Distribution—Java Implementation

Listing D.16 shows the implementation of the uniform distribution in Java.

Listing D.16 Java implementation of the uniform distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;

// Uniform distribution over a given interval.

// @author Didier H. Besset

public final class UniformDistribution
                                        extends ProbabilityDensityFunction
{

    // Low limit.

    private double a;

    // High limit.

    private double b;


// Constructor method.
// @param low double    low limit
// @param high double    high limit
// @exception java.lang.IllegalArgumentException
//                            if the limits are inverted.

public UniformDistribution ( double low, double high)
                                        throws IllegalArgumentException
{
    if ( low >= high )
        throw new IllegalArgumentException(
                    "Limits of distribution are equal or reversed");
    a = low;
    b = high;
}

// 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 UniformDistribution( Histogram h)
{
    b = h.standardDeviation() * 1.73205080756888; // sqrt(12)/2
```

```
        double c = h.average();
        a = c - b;
        b += c;
    }

    // @return double average of the distribution.

    public double average()
    {
        return ( a + b) * 0.5;
    }

    // Returns the probability of finding a random variable smaller
    // than or equal to x.
    // @return integral of the probability density function from a to x.
    // @param x double upper limit of integral.

    public double distributionValue ( double x)
    {
        if ( x < a )
            return 0;
        else if ( x < b )
            return ( x - a) / ( b - a);
        else
            return 1;
    }

    // @return double kurtosis of the distribution.

    public double kurtosis( )
    {
        return -1.2;
    }

    // @return java.lang.String     name of the distribution

    public String name ( )
    {
        return "Uniform distribution";
    }

    // @return double[] an array containing the parameters of
    //                                  the distribution.

    public double[] parameters ( )
    {
        double[] answer = new double[2];
```

```
        answer[0] = a;
        answer[1] = b;
        return answer;
    }

    // This method assumes that the range of the argument has been checked.
    // @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)
    {
        return (b - a) * x + a;
    }

    // @param p double[]    assigns the parameters

    public void setParameters( double[] params)
    {
        a = params[0];
        b = params[1];
    }

    // @return double skewness of the distribution.

    public double skewness( )
    {
        return 0;
    }

    // @return double probability density function
    // @param x double random variable

    public double value( double x)
    {
        if ( x < a )
            return 0;
        else if ( x < b )
            return 1 / (b - a);
        else
            return 0;
    }

    // @return double variance of the distribution

    public double variance()
    {
```

TABLE D.9 Properties of the Weibull distribution

Range of random variable	$[0, +\infty[$
Probability density function	$P(x) = \frac{\alpha x^{\alpha-1}}{\beta^{\alpha}} e^{-\left(\frac{x}{\beta}\right)^{\alpha}}$
Parameters	$0 < \alpha < \infty$ $0 < \beta < \infty$
Distribution function	$F(x) = 1 - e^{-\left(\frac{x}{\beta}\right)^{\alpha}}$
Average	$\frac{\beta}{\alpha}\Gamma\left(\frac{1}{\alpha}\right)$
Variance	$\frac{\beta^2}{\alpha}\left[2\Gamma\left(\frac{2}{\alpha}\right) - \frac{1}{\alpha}\Gamma\left(\frac{1}{\alpha}\right)^2\right]$

```
        double range = b - a;
        return range * range / 12;
    }
    }
```

D.9 Weibull Distribution

Table D.9 shows the properties of the Weibull distribution. The Weibull distribution is used to model the behavior of reliability. It is defined by its acceptance function. Its shape is similar to that of the gamma distribution and thus can be applied to the same types of problems. Figure D.7 shows the shapes taken by the Weibull distribution for a few values of the parameters.

Because the Weibull distribution is defined by its distribution function, the estimation of the initial values of the parameters from a histogram is made by computing the distribution function at twp positions. These positions are determined using the histogram limits and the average so that the estimation of the distribution function using the histogram has enough significance.

D.9.1 Weibull Distribution—Smalltalk Implementation

Listing D.17 shows the implementation of the Weibull distribution in Smalltalk.

Listing D.17 Smalltalk implementation of the Weibull distribution

Class	`DhbWeibullDistribution`
Subclass of	`DhbProbabilityDensity`
Instance variable names:	`alpha beta norm`

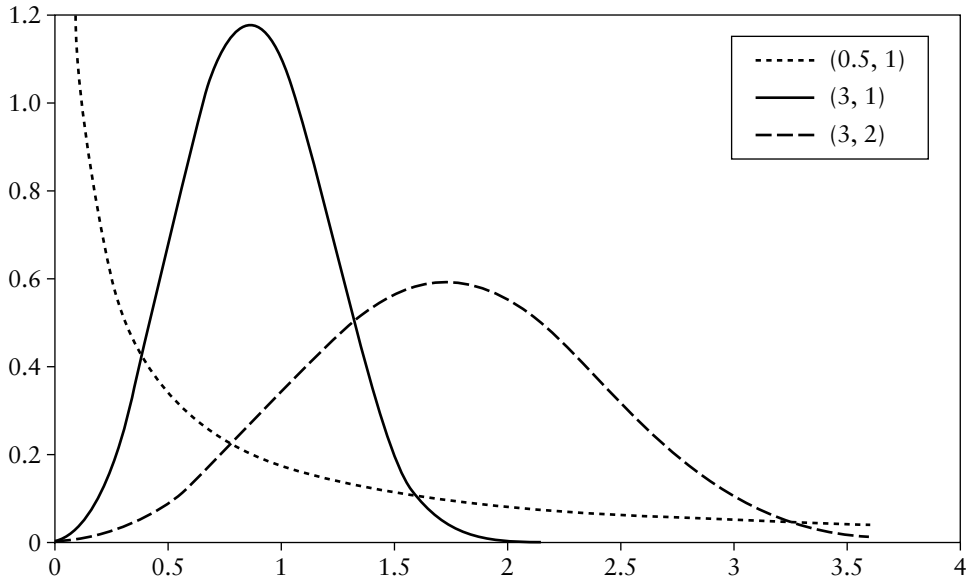

FIG. D.7 Weibull distribution for a few parameters

Class Methods

distributionName

```
^'Weibull distribution'
```

fromHistogram: aHistogram

```
| average xMin xMax accMin accMax |
aHistogram minimum < 0
    ifTrue: [ ^nil].
average := aHistogram average.
xMin := ( aHistogram minimum + average) / 2.
accMin := ( aHistogram countsUpTo: xMin) / aHistogram totalCount.
xMax := ( aHistogram maximum + average) / 2.
accMax := ( aHistogram countsUpTo: xMax) / aHistogram totalCount.
^[self solve: xMin acc: accMin upper: xMax acc: accMax]
        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
```

solve: lowX **acc:** lowAcc **upper:** highX **acc:** highAcc

```
| lowLnAcc highLnAcc deltaLnAcc lowLnX highLnX |
lowLnAcc := (1 - lowAcc) ln negated ln.
highLnAcc := (1 - highAcc) ln negated ln.
deltaLnAcc := highLnAcc - lowLnAcc.
lowLnX := lowX ln.
highLnX := highX ln.
^self shape: deltaLnAcc / (highLnX - lowLnX)
    scale: ((highLnAcc * lowLnX - (lowLnAcc * highLnX)) /
                                                    deltaLnAcc) exp
```

Instance Methods

acceptanceBetween: aNumber1 **and:** aNumber2

```
^self privateAcceptanceBetween: aNumber1 and: aNumber2
```

average

```
^(1 / alpha) gamma * beta / alpha
```

changeParametersBy: aVector

```
alpha := alpha + ( aVector at: 1).
beta := beta + ( aVector at: 2).
self computeNorm.
```

computeNorm

```
norm := alpha/ ( beta raisedTo: alpha).
```

distributionValue: aNumber

```
^aNumber > 0
    ifTrue: [ 1 - ( ( ( aNumber / beta) raisedTo: alpha) negated
                                                        exp)]
    ifFalse:[ 0]
```

initialize: aNumber1 **scale:** aNumber2

```
( aNumber1 > 0 and: [ aNumber2 > 0])
    ifFalse: [ self error: 'Illegal distribution parameters'].
alpha := aNumber1.
beta := aNumber2.
self computeNorm.
^self
```

parameters

```
^Array with: alpha with: beta
```

privateInverseDistributionValue: aNumber

```
^( (1 - aNumber) ln negated raisedTo: ( 1 / alpha)) * beta
```

value: aNumber

```
^( ( aNumber / beta) raisedTo: alpha) negated exp * ( aNumber
                                        raisedTo: ( alpha - 1)) * norm
```

variance

```
^( beta squared / alpha) * ( (2 / alpha) gamma * 2 - ( (1 / alpha
                                              ) gamma squared / alpha))
```

D.9.2 Weibull Distribution—Java Implementation

Listing D.18 shows the implementation of the Weibull distribution in Java.

Listing D.18 Java implementation of the Weibull distribution

```
package DhbStatistics;

import DhbScientificCurves.Histogram;
import DhbFunctionEvaluation.GammaFunction;
import DhbInterfaces.ParametrizedOneVariableFunction;

// Weibull distribution.

// @author Didier H. Besset

public final class WeibullDistribution
                                        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;

// Create a new instance of the Weibull 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
//                  if any of the parameters is non-positive.

public WeibullDistribution ( 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 WeibullDistribution( Histogram h)
                                    throws IllegalArgumentException
{
    if ( h.getMinimum() < 0 )
        throw new IllegalArgumentException(
    "Weibull distribution is only defined for non-negative values");
    double average = h.average();
    if ( average <= 0 )
        throw new IllegalArgumentException(
        "Weibull distribution must have a non-negative average");
    double xMin = ( h.getMinimum() + average) * 0.5;
    double accMin = Math.log( -Math.log( 1 - h.getCountsUpTo(xMin)
                                            / h.totalCount()));
    double xMax = ( h.getMaximum() + average) * 0.5;
    double accMax = Math.log( -Math.log( 1 - h.getCountsUpTo(xMax)
                                            / h.totalCount()));
    double delta = accMax - accMin;
    xMin = Math.log( xMin);
    xMax = Math.log( xMax);
```

```
        defineParameters( delta / ( xMax - xMin),
                          Math.exp( ( accMax * xMin - accMin * xMax)
                                                                / delta));
    }

    // @return double average of the distribution.

    public double average()
    {
        return GammaFunction.gamma( 1 / alpha) * beta / alpha;
    }

    // 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 = alpha / Math.pow( beta, alpha);
        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 1.0 - Math.exp(-Math.pow( x / beta, alpha));
    }

    // @return java.lang.String the name of the distribution.

    public String name()
    {
        return "Weibull 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;
    }

    // This method assumes that the range of the argument has been checked.
    // @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)
    {
        return Math.pow( -Math.log( 1 - x), 1. / alpha) * beta;
    }

    // @param p double[]    assigns the parameters

    public void setParameters( double[] params)
    {
        defineParameters( params[0], params[1]);
    }

    // This method was created in VisualAge.
    // @return java.lang.String

    public String toString()
    {
        StringBuffer sb = new StringBuffer();
        java.text.DecimalFormat fmt = new java.text.DecimalFormat(
                                                        "####0.00000");
        sb.append("Weibull 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 norm * Math.pow( x, alpha - 1)
                                * Math.exp( -Math.pow( x / beta, alpha));
    }
```

```
    // @return double variance of the distribution.

    public double variance ( )
    {
        double s = GammaFunction.gamma( 1 / alpha);
        return beta * beta *( 2 * GammaFunction.gamma( 2 / alpha)
                                                - s * s / alpha) / alpha;
    }
    }
```

CHAPTER 13

Accurate Accumulation of Expectation Values

13.1 Accurate Accumulation of Central Moments

This section shows the detailed derivation of equation 9.13 of Section 9.2. The aim of this demonstration is to express the central moment of order k estimated over a sample of $n+1$ measurements as a function of the central moments of an order lower or equal to k estimated over a sample of n measurements. The estimator of the central moment of order k is defined by:

$$\left\langle (x-\bar{x})^k \right\rangle_{n+1} = \frac{1}{n+1} \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_{n+1}\right)^k . \tag{13.1}$$

We shall now concentrate on changing the sum of equation 13.1 in such way as to bring quantities that are already computed. The sum of 13.1 is equal to

$$\begin{aligned} S &= (n+1) \left\langle (x-\bar{x})^k \right\rangle_{n+1} \\ &= \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_{n+1}\right)^k \\ &= \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_n + \langle x \rangle_n - \langle x \rangle_{n+1}\right)^k \\ &= \sum_{i=1}^{n+1} \left[\left(x_i - \langle x \rangle_n\right) + \Delta_{n+1}\right]^k , \end{aligned} \tag{13.2}$$

where we have introduced the correction defined in equation 9.12. We can now transform the expression inside the sum using binomial expansion:

$$\begin{aligned} S &= \sum_{i=1}^{n+1} \sum_{l=0}^{k} \binom{l}{k} \left(x_i - \langle x \rangle_n\right)^l \Delta_{n+1}^{k-l} \\ &= \sum_{l=0}^{k} \binom{l}{k} \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_n\right)^l \Delta_{n+1}^{k-l}. \end{aligned} \tag{13.3}$$

In the second part of equation 13.3, the two sums have been permuted. As in the case of the average, we now make the last term of the inner sum explicit. The remaining sum can then be expressed as a function of the estimators of the central moments over n measurements. The term containing the $(n+1)$th measurement can be rewritten as a function of the correction defined in equation 9.12. We have

$$\begin{aligned} S &= \sum_{l=0}^{k} \binom{l}{k} \left[\left(x_{n+1} - \langle x \rangle_n\right)^l \Delta_{n+1}^{k-l} + \Delta_{n+1}^{k-l} \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_n\right)^l \right] \\ &= \sum_{l=0}^{k} \binom{l}{k} \left[(n+1)^l \left(\frac{x_{n+1} - \langle x \rangle_n}{n+1} \right)^l \Delta_{n+1}^{k-l} + n \left\langle (x - \bar{x})^l \right\rangle_n \Delta_{n+1}^{k-l} \right] \\ &= \sum_{l=0}^{k} \binom{l}{k} \left[(-n-1)^l \Delta_{n+1}^{k} + n \left\langle (x - \bar{x})^l \right\rangle_n \Delta_{n+1}^{k-l} \right] \\ &= \sum_{l=0}^{k} \binom{l}{k} (-n-1)^l \Delta_{n+1}^{k} + \sum_{l=0}^{k} \binom{l}{k} n \left\langle (x - \bar{x})^l \right\rangle_n \Delta_{n+1}^{k-l}. \end{aligned} \tag{13.4}$$

In the last line of equation 13.4, the first term contains the binomial expansion of the following expression

$$\sum_{l=0}^{k} \binom{l}{k} (-n-1)^l = \left[1 + (-n-1)\right]^k = (-n)^k . \tag{13.5}$$

Thus,we have

$$S = \left(-n\Delta_{n+1}\right)^k + n \sum_{l=0}^{k} \binom{l}{k} \left\langle (x - \bar{x})^l \right\rangle_n \Delta_{n+1}^{k-l}. \tag{13.6}$$

In equation E.6, the first term of the sum is just Δ_{n+1}^{k}, and the second term of the sum vanishes by definition of the average $\bar{x}$. This gives us the final expression to compute the estimator of the central moment computed over $n+1$ measurements as

a function of the estimator of the central moment computed over n measurements:

$$\left\langle (x-\bar{x})^k \right\rangle_{n+1} = \frac{n}{n+1}\left\{ \left[1-(-n)^{k-1}\right]\Delta_{n+1}^k + \sum_{l=2}^{k}\binom{l}{k}\left\langle (x-\bar{x})^l \right\rangle_n \Delta_{n+1}^{k-l} \right\}. \tag{13.7}$$

Quod erat demonstrandum. . . .

13.2 Accurate Accumulation of the Covariance

This section shows the detailed derivation of equation 12.6. To simplify notation, the components x_i and x_j have been renamed x and y, respectively.

The estimator of the covariance of two random variables x and y over n measurements is defined by

$$\mathrm{cov}_n(x,y) = \left\langle (x_i - \langle x\rangle_n)\left(y_i - \langle y\rangle_n\right)\right\rangle_n = \frac{1}{n}\sum_{i=1}^{n}(x_i - \langle x\rangle_n)\left(y_i - \langle y\rangle_n\right). \tag{13.8}$$

The estimator of the covariance of x and y over $n+1$ measurements is then given b:

$$\mathrm{cov}_{n+1}(x,y) = \frac{1}{n+1}\sum_{i=1}^{n+1}(x_i - \langle x\rangle_{n+1})\left(y_i - \langle y\rangle_{n+1}\right). \tag{13.9}$$

The sum in equation E.9 can then be expressed as

$$\begin{aligned} C_{n+1} &= (n+1)\left\langle (x_i - \langle x\rangle_{n+1})\left(y_i - \langle y\rangle_{n+1}\right)\right\rangle_{n+1} \\ &= \sum_{i=1}^{n+1}(x_i - \langle x\rangle_{n+1})\left(y_i - \langle y\rangle_{n+1}\right) \\ &= \sum_{i=1}^{n+1}(x_i - \langle x\rangle_n + \langle x\rangle_n - \langle x\rangle_{n+1})\left(y_i - \langle y\rangle_n + \langle y\rangle_n - \langle y\rangle_{n+1}\right) \\ &= \sum_{i=1}^{n+1}(x_i - \langle x\rangle_n + \Delta_{x,n+1})\left(y_i - \langle y\rangle_n + \Delta_{y,n+1}\right), \end{aligned} \tag{13.10}$$

where we introduce the corrections to the estimation of the expectation value of x and y as follows:

$$\begin{cases} \Delta_{x,n+1} = & \langle x\rangle_n - \langle x\rangle_{n+1} \\ = & \frac{\langle x\rangle_n - x_{n+1}}{n+1}, \\ \Delta_{y,n+1} & = \langle y\rangle_n - \langle y\rangle_{n+1} \\ = & \frac{\langle y\rangle_n - y_{n+1}}{n+1}. \end{cases} \tag{13.11}$$

Thus, we have

$$\begin{aligned} C_{n+1} &= \sum_{i=1}^{n+1} \left[\left(x_i - \langle x \rangle_n\right) \left(y_i - \langle y \rangle_n\right) + \Delta_{y,n+1} \left(x_i - \langle x \rangle_n\right) \right. \\ &\qquad \left. + \Delta_{x,n+1} \left(y_i - \langle y \rangle_n\right) + \Delta_{x,n+1}\Delta_{y,n+1} \right] \\ &= \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_n\right) \left(y_i - \langle y \rangle_n\right) + \left(n+1\right) \Delta_{x,n+1}\Delta_{y,n+1} \\ &\qquad + \Delta_{y,n+1} \left(x_{n+1} - \langle x \rangle_n\right) + \Delta_{x,n+1} \left(y_{n+1} - \langle x \rangle_n\right). \end{aligned} \tag{13.12}$$

The last line is obtained from the definition of the expectation values $\langle x \rangle_n$ and $\langle y \rangle_n$. Using the definitions of $\Delta_{x,n+1}$ and $\Delta_{y,n+1}$, we have

$$\begin{aligned} C_{n+1} &= \sum_{i=1}^{n+1} \left(x_i - \langle x \rangle_n\right) \left(y_i - \langle y \rangle_n\right) - \left(n+1\right) \Delta_{x,n+1}\Delta_{y,n+1} \\ &= \sum_{i=1}^{n} \left(x_i - \langle x \rangle_n\right) \left(y_i - \langle y \rangle_n\right) - \left(n+1\right) \Delta_{x,n+1}\Delta_{y,n+1} \\ &\qquad + \left(x_{n+1} - \langle x \rangle_n\right) \left(y_{n+1} - \langle y \rangle_n\right) \\ &= n\mathrm{cov}_n\left(x, y\right) + n\left(n+1\right) \Delta_{x,n+1}\Delta_{y,n+1}. \end{aligned} \tag{13.13}$$

Now, we obtain the expression for the estimator of the covariance over $n+1$ measurements as a function of the estimator of the covariance over n measurements:

$$\mathrm{cov}_{n+1}\left(x, y\right) = \frac{n}{n+1}\mathrm{cov}_n\left(x, y\right) + n\Delta_{x,n+1}\Delta_{y,n+1}. \tag{13.14}$$

Note that this equation yields equation 9.14 if one put $y = x$.

References

[Abramovitz & Stegun] Milton Abramovitz and Irene A. Stegun, *Handbook of Mathematical Functions*, Dover, 1964.

[Achtley & Bryant] William R. Achtley and Edwin H. Bryant editors, *Benchmark Papers in Systematic and Evolutionary Biology*, Vol. 1, Dowden, Hutchinson & Ross, Inc., Stroudsburg, Pa.; distributed by Halsted Press [John Wiley & Sons, Inc.], New York, 1975.

[Bass] J. Bass, *Cours de Mathématiques*, Tome II, Masson, 1968.

[Beck] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Berry & Linoff] Michael J. A. Berry and Gordon Linoff, *Data Mining for Marketing, Sales and Customer Support*, Wiley, 1997.

[Cormen *et al.*] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.

[Gamma *et al.*] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[Gullberg] Jan Gullberg, *Mathematics from the Birth of the Numbers*, Norton, 1997.

[Ifrah] Georges Ifrah, *Histoire Universelle des Chiffres*, Robert Laffont, 1994.

[Knudth 1] Donald E. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, 1973.

[Knudth 2] Donald E. Knuth, *The Art of Computer Programming*, Vol. 2, Addison-Wesley, 1981.

[Knudth 3] Donald E. Knuth, *The Art of Computer Programming*, Vol. 3, Addison-Wesley, 1973.

[Koza *et al.*] John R. Koza, Forrest H, Bennett III, David Andre, and Martin A. Keane, *Genetic Programming III*, Morgan Kaufmann, 1999.

[Law & Kelton] Averill M. Law and W. David Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, 1982.

[Phillips & Taylor] G. M. Phillips and P. J. Taylor, *Theory and Applications of Numerical Analysis*, Academic Press, 1973.

[Press *et al.*] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, *Numerical Recipes for C: The Art of Scientific Computing*, Cambridge University Press, 1992.

[Alpert *et al.*] Sherman R. Alpert, Kyle Brown, and Bobby Woolf, *Design Pattern Smalltalk Companion*, Addison-Wesley, 1998.

[Smith] David N. Smith, *IBM Smalltalk: The Language*, Addison-Wesley, 1995.

[Flanagan] David Flanagan, *Java in a Nutshell*, O'Reilly, 1996.