

# **Statistics**

## **with routines in Pascal**

Engelbert Buxbaum

5th June 2021

Copyright © 2021 by Dr. Engelbert Buxbaum ([engelbert\\_buxbaum@web.de](mailto:engelbert_buxbaum@web.de)).  
This program is free software: you can redistribute it and/or modify it under the terms  
of the GNU General Public License as published by the Free Software Foundation, either  
version 3 of the License, or (at your option) any later version.  
This program is distributed in the hope that it will be useful, but WITHOUT ANY  
WARRANTY; without even the implied warranty of MERCHANTABILITY or FIT-  
NESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for  
more details.  
A copy of the GNU General Public License is included in the manual. See also [ht-  
tps://www.gnu.org/licenses/](https://www.gnu.org/licenses/).

# Contents

<b>I. Fundamental maths</b>	<b>1</b>
<b>1. Basic math routines</b>	<b>3</b>
1.1. Administrative functions . . . . .	7
1.2. Integer arithmetic . . . . .	11
1.3. Maximum and minimum of two numbers . . . . .	13
1.4. Exponents, logarithms, powers . . . . .	14
1.5. Intersection of two lines . . . . .	16
1.6. Conversion of angles between rad and degrees . . . . .	16
1.7. Hyperbolic functions . . . . .	17
1.8. Area functions . . . . .	18
1.9. Cyclometric functions . . . . .	18
1.10. Routines required for statistics . . . . .	20
1.10.1. Incomplete gamma-function . . . . .	22
1.10.2. Binomial coefficients . . . . .	24
1.11. Finance mathematics (with a German flavour) . . . . .	25
1.12. Functions already in unit <code>math</code> . . . . .	28
<b>2. Complex numbers</b>	<b>33</b>
2.1. Basic conversion and I/O routines . . . . .	37
2.2. Basic operators . . . . .	40
2.2.1. Operators on complex variables . . . . .	40
2.2.2. Operators on mixed complex and real variables . . . . .	42
2.3. Logarithms and powers . . . . .	45
2.4. Angle and cyclometric functions . . . . .	47
2.5. Hyperbolic functions . . . . .	50
2.6. Area functions . . . . .	52
2.7. Test program . . . . .	54
<b>3. Big sets</b>	<b>65</b>
<b>4. Formula compiler</b>	<b>71</b>
4.1. The source code . . . . .	72
4.1.1. Administrative routines . . . . .	74
4.1.2. Symbolic differentiation . . . . .	77
4.1.3. Simplification of Calc-programs . . . . .	79
4.1.4. Derivative of a function . . . . .	94

## Contents

4.1.5. Compile the function . . . . .	104
4.1.6. Calculate value of an expression . . . . .	111
4.1.7. Screen output of calc programs . . . . .	114
4.1.8. Test program . . . . .	117
<b>5. Vector arithmetic</b>	<b>119</b>
5.1. Generation and destruction of vectors . . . . .	122
5.2. Accesing and changing data in vectors . . . . .	124
5.3. Calculation with vectors . . . . .	125
5.3.1. Calculations using one vector and a constant number as argument	125
5.3.2. Routines that take two vectors and calculate a third . . . . .	127
5.3.3. Functions that take a Vector and produce a number . . . . .	128
5.3.4. Properties of a vector . . . . .	130
5.3.5. Vector norms . . . . .	132
5.3.6. Scaling, centering and normalisation of vectors . . . . .	137
5.3.7. Vector distance . . . . .	138
5.4. Sorting a vector . . . . .	139
<b>6. Matrix calculations</b>	<b>143</b>
6.1. Generation and destruction of matrices . . . . .	146
6.2. Accesing and changing data in a matrix . . . . .	149
6.3. Special matrix types . . . . .	154
6.3.1. Identity matrix . . . . .	154
6.3.2. The null-matrix . . . . .	155
6.3.3. The HILBERT-matrix . . . . .	156
6.3.4. The quadratic matrix . . . . .	156
6.3.5. The symmetric matrix . . . . .	156
6.3.6. Trapezoid matrices . . . . .	157
6.3.7. The diagonal matrix and function diag . . . . .	158
6.3.8. Triangular matrices . . . . .	159
6.3.9. HESSENBERG matrices . . . . .	159
6.3.10. Positive and negative definite matrices . . . . .	160
6.4. Calculation with matrices . . . . .	161
6.4.1. Matrix norms . . . . .	161
6.4.2. Determinant . . . . .	164
6.4.3. Matrix transpose . . . . .	168
6.4.4. Matrix inverse . . . . .	169
6.4.5. Matrix sums and differences . . . . .	171
6.4.6. Matrix inner and HADAMARD-SCHUR product . . . . .	172
6.4.7. Matrix division . . . . .	174
6.4.8. Symmetric and anti-symmetric matrices . . . . .	175
6.4.9. Matrix exponentiation . . . . .	176
6.5. Calculations with vectors and matrices . . . . .	177
6.5.1. Dyadic vector product . . . . .	177

6.5.2. Product of a matrix and a vector . . . . .	177
6.5.3. Change a vector to a matrix with one row . . . . .	178
6.6. Sort rows of a matrix by value of a column . . . . .	179
6.7. Solving systems of linear equations . . . . .	180
6.7.1. Private routines . . . . .	181
6.7.2. The condition number . . . . .	184
6.7.3. Over- and under-determined systems . . . . .	185
6.7.4. Example . . . . .	186
6.7.5. GAUSSian elimination . . . . .	186
6.7.6. GAUSS-elimination with partial pivoting . . . . .	187
6.7.7. LU-decomposition . . . . .	188
6.7.8. The QR-factorisation . . . . .	194
6.7.9. Matrix rank . . . . .	198
6.8. Eigenvalues and eigenvectors . . . . .	199
6.8.1. Real, symmetric matrices . . . . .	199
6.9. Singular value decomposition (SVD) . . . . .	212
6.9.1. Introduction . . . . .	212
6.9.2. Implementation . . . . .	213
6.9.3. The MOORE-PENROSE pseudoinverse . . . . .	221
<b>7. Dynamic structures for data of any type</b>	<b>225</b>
7.1. The interface . . . . .	225
7.2. The LIFO (stack) . . . . .	227
7.3. The FIFO (list, buffer, queue) . . . . .	229
7.4. Test-program . . . . .	230
<b>II. Univariate statistics</b>	<b>233</b>
<b>8. Statistical distributions</b>	<b>235</b>
8.1. F-distribution . . . . .	237
8.2. STUDENT's t-distribution . . . . .	240
8.2.1. Calculating $t$ . . . . .	243
8.3. $\chi^2$ Distribution . . . . .	245
8.3.1. The G-distribution . . . . .	249
8.4. GAUSS' normal distribution . . . . .	249
8.5. Binomial distribution . . . . .	255
8.6. The PARETO (80/20) distribution . . . . .	256
8.7. The $\beta$ -distribution . . . . .	257
8.8. The $\Gamma$ -distribution . . . . .	263
8.9. KOLMOGOROV-SMIRNOV-test . . . . .	269
<b>9. Pseudo-random numbers of various distributions</b>	<b>273</b>
9.1. LAPLACE-distribution . . . . .	274

## Contents

9.2. Binomial distribution . . . . .	274
9.3. Normal distribution . . . . .	275
9.4. Exponential and POISSON distribution . . . . .	275
9.5. BERNOULLI-distribution . . . . .	276
9.6. $\chi^2$ -, $t$ - and F- distribution . . . . .	277
9.7. PARETO-distribution . . . . .	277
<b>10. Descriptive statistics</b>	<b>279</b>
10.1. Normally distributed data . . . . .	283
10.1.1. Position . . . . .	283
10.1.2. Dispersion . . . . .	288
10.1.3. PEARSON moments . . . . .	292
10.1.4. Concentration . . . . .	293
10.2. Non-parametric measures . . . . .	297
10.3. Normalisation and standardisation of vectors . . . . .	320
10.4. Grouped data . . . . .	322
10.5. Descriptive statistics of matrices . . . . .	323
10.6. Test program . . . . .	337
<b>11. Correlation coefficients</b>	<b>343</b>
11.1. General routines . . . . .	346
11.2. Cardinal (interval and rational) data . . . . .	347
11.2.1. PEARSON's product moment correlation coefficient $r_p$ . . . . .	347
11.2.2. SPEARMAN's rank correlation coefficient $r_s$ . . . . .	354
11.2.3. Quadrant correlation $r_q$ . . . . .	356
11.3. Binary data . . . . .	358
11.3.1. MATTHEWS' correlation $r_m$ . . . . .	360
11.3.2. GOODMAN & KRUSKAL's $\lambda$ for binary/binary association . . . . .	362
11.3.3. The point biserial correlation $r_j$ for binary/cardinal association . . . . .	362
11.4. Ordinal data . . . . .	364
11.4.1. Ordinal/ordinal associations . . . . .	367
11.5. Nominal data . . . . .	373
11.5.1. Intraclass correlation for nominal/cardinal association . . . . .	373
11.5.2. CRAMÉR's V for nominal/nominal association . . . . .	374
11.5.3. GUTTMAN's coefficient of relative predictability $\lambda$ for nominal/-nominal association . . . . .	376
11.5.4. FREEMAN's measure of association $\Theta$ for nominal/ordinal association . . . . .	380
11.5.5. $\eta^2$ for nominal/cardinal association . . . . .	382
11.5.6. Latent correlation . . . . .	385
<b>12. Linear and linearising Regression</b>	<b>389</b>
12.1. Linear regression . . . . .	390
12.1.1. Requirements for linear regression . . . . .	391

12.1.2. Algorithm . . . . .	392
12.2. Weighted regression . . . . .	399
12.3. Robust linear regression . . . . .	400
12.4. Linearising regression . . . . .	402
12.5. DEMING-regression . . . . .	411
12.6. THEIL-SEN-KENDALL-estimator for noisy data . . . . .	414
12.7. The direct plot of EISENTHAL & CORNISH-BOWDEN . . . . .	417
12.8. Aberrant data points . . . . .	420
12.8.1. Leverage points . . . . .	420
12.8.2. Outliers . . . . .	421
12.9. Shrinkage . . . . .	421
12.9.1. Ridge regression . . . . .	422
12.9.2. Lasso regression . . . . .	422
<b>13. Regression of curves: The simplex algorithm</b>	<b>425</b>
13.1. How does the simplex algorithm work? . . . . .	425
13.2. Examples . . . . .	428
13.2.1. Enzyme kinetics with [S] spanning several orders of magnitude: heteroscedasticity . . . . .	428
13.2.2. Fitting of a system of differential equations to a data set . . . . .	430
13.3. Error estimation . . . . .	431
13.3.1. Confidence intervals for parameters . . . . .	431
13.3.2. Goodness of fit . . . . .	433
13.4. Code for Simplex-Regression . . . . .	435
<b>14. Circular data</b>	<b>455</b>
14.1. The interface . . . . .	455
14.2. Transformation of circular data . . . . .	459
14.3. Description of a single vector of circular data . . . . .	459
14.3.1. Position . . . . .	459
14.3.2. Spread . . . . .	462
14.3.3. Higher moments . . . . .	463
14.3.4. Confidence interval for mean direction . . . . .	467
14.4. Pseudo-random variables . . . . .	468
14.5. Statistical tests for a single circular data set . . . . .	469
14.5.1. Do samples have a preferred direction . . . . .	469
14.5.2. Symmetry around $\theta$ . . . . .	476
14.5.3. Grouped data . . . . .	476
14.6. Multi-sample tests . . . . .	477
14.6.1. WATSON-WILLIAMS-test: two vectors with circular data . . . . .	477
14.6.2. Circular KRUSKAL-WALLIS-test for equal median . . . . .	478
14.6.3. Difference between two vectors . . . . .	480
14.7. Regression and correlation . . . . .	486
14.7.1. Linear dependent, circular independent variable . . . . .	486

14.7.2. Circular dependent, linear independent variable . . . . .	494
14.7.3. Circular dependent, circular independent variable . . . . .	494
14.8. Test program . . . . .	498
<b>III. Multivariate Statistics</b>	<b>505</b>
<b>15. Definitions and concepts for multivariate statistics</b>	<b>507</b>
15.1. Tasks fore multivariate statistics . . . . .	507
15.1.1. Errors . . . . .	508
15.2. Missing data . . . . .	509
15.2.1. Classes of missing data . . . . .	509
15.2.2. Handling missing data . . . . .	510
<b>16. Principle component analysis</b>	<b>513</b>
16.1. Pricipal component analysis (PCA) <i>vs.</i> factor analysis (FA) . . . . .	513
16.2. Is a data set suitable for factor analysis and PCA? . . . . .	515
16.2.1. Number of data . . . . .	515
16.2.2. Correlation coefficients . . . . .	515
16.2.3. Multivariate normality of data . . . . .	515
16.2.4. The KAISER-MEYER-OLKIN (KMO) criterium . . . . .	516
16.2.5. BARTLETT's sphere test . . . . .	517
16.3. Mathematical basis of principle component analysis . . . . .	518
16.3.1. Calculation of component and factor scores . . . . .	522
16.4. Pascal program for PCA . . . . .	523
16.5. Example: Phylogentic trees from nucleic acid sequences . . . . .	525
16.6. The unit PCA . . . . .	527
16.7. Real world data in PCA . . . . .	542
16.7.1. If the correlation matrix is not positive definite: Shrinkage . . . . .	542
16.7.2. Effect of discrete variables on PCA . . . . .	551
16.7.3. Number of components . . . . .	552
16.8. Rotation of factors or components . . . . .	556
16.8.1. Mathematical procedure for orthogonal rotation . . . . .	560
16.8.2. Pascal code for rotation . . . . .	562
16.8.3. Examples . . . . .	573
16.8.4. Interpretation . . . . .	580
<b>17. Cluster-analysis</b>	<b>587</b>
17.1. Read the data matrix . . . . .	587
17.2. Calculate the similarity/distance matrix . . . . .	589
17.3. Hierarchic-agglomerative clustering . . . . .	594
17.4. <i>k</i> -means clustering . . . . .	600
17.4.1. Modifications . . . . .	601
17.4.2. Selection of optimal <i>k</i> . . . . .	602

17.4.3. Sourcecode for $k$ -means clustering with $k$ -fold crossvalidation . . . . .	602
---	-----

<b>IV. Appendix</b>	<b>617</b>
---------------------	------------

<b>Symbols and abbreviations</b>	<b>619</b>
----------------------------------	------------

1. Symbols used . . . . .	619
2. Abbreviations . . . . .	619

<b>GNU GENERAL PUBLIC LICENSE</b>	<b>621</b>
-----------------------------------	------------



# **Part I.**

## **Fundamental maths**



# 1. Basic math routines

## Abstract

MathFunc contains basic math functions beyond the routines already available in the Lazarus-library `math`. It also defines important mathematical and natural constants and the typed constants (changeable by the user program) `ValidFigures`, `MaxError`, `MaxIter`, `Zero`.

The interface is:

Listing 1.1: Interface of unit MathFunc

```
1 UNIT MathFunc;  
2  
3 INTERFACE  
4  
5 USES Math;  
6 // IN Lazarus no need to reinvent the wheel, otherwise uncomment important  
// routines at the end  
7  
8 CONST  
9   MathError: BOOLEAN = FALSE;    // toggle FOR error condition  
10  PowerOfTwo: ARRAY[0..30] OF LONGINT =  
11    (      1,          2,          4,          8,         16,         32,  
12      64,        128,        256,        512,       1024,       2048,  
13     4096,       8192,      16384,      32768,      65536,      131072,  
14    262144,      524288,     1048526,     2097152,     4194304,     8388608,  
15   16777216,    33554432,   67108864,   134217728,   268435456,   536870912,  
16  1073741824);  
17  
18 TYPE  
19   ByteStr = STRING[2];  
20   float    = double;           // allows global switch of precision  
21  
22 CONST  
23   ValidFigures: BYTE = 10;      // Ziffern bei der Ausgabe  
24   MaxError     : float = 1e-7;    // Konvergenz-Kriterium  
25   MaxIter      : WORD = 1000;     // falls keine Konvergenz  
26   Zero         : float = 1e-100;   // kleinste Number <> 0  
27  
28   MachineEpsilon = 2.2204460492503130E-16; // Maschinen-abhängig
```

## 1. Basic math routines

```
29  MaxRealNumber = 1.7976931348623150E+308;
30  MinRealNumber = 2.2250738585072020E-308;
31
32  Const_pi      = 3.141592653589793238462643383280; // Kreiszahl
33  Const_2pi     = 2 * Const_pi;
34  Const_e        = 2.718281828459045235360287471353; // Euler'sche Zahl
35  Const_ln2      = 0.693147180559945309417232121458; // nat. log. von 2
36  Const_ln10     = 2.302585092994045684017991454684; // nat. log. von 10
37  Const_delta    = 4.669201609102990671853203820466; // Feigenbaum-Konstante
38  Const_phi     = (1 + sqrt(5)) / 2;                      // goldener Schnitt
39  Const_Sqrt2    = sqrt(2);
40
41
42  Const_a0       = 5.29177210903e-11;                     // Bohr-radius, radius of H
   atom, m
43  Const_alpha    = 7.2973525693e-3;                      // Finestructure
   (Sommerfeld) constant
44  Const_g        = 9.80665;                            // Fallbeschleunigung Erde,
   m/s^2
45  Const_Gamma   = 6.67430e-11;                          // Gravitationskonstante, N
   m^2 / kg^2
46  Const_Na       = 6.0221367e23;                         // Avogadro-Konstante, mol^-1
47  Const_kB       = 1.380658e-23;                         // Boltzmann-Konstante, J/K
48  Const_F        = 96485.33289;                          // Faraday-Konstante, J /
   (mol V) = C/mol
49  Const_c        = 299792458;                           // Lichtgeschwindigkeit, m/s
50  Const_h       = 6.6260755e-34;                         // Plank'sches
   Wirkungsquantum, J s
51  Const_bar_h   = Const_h / Const_2pi;                  // Elektronenmasse, kg
52  Const_me       = 9.10938356e-31;                      // Protonmasse, kg
53  Const_mp       = 1.672621898e-27;                    // Neutronenmasse, kg
54  Const_mn       = 1.674927471e-27;                   // Elementarladung, A s
55  Const_Ce       = 1.60217733e-19;                     // Gaskonstante, J / (mol kg)
56  Const_R        = 8.3144598;                          // magnetische Feldkonstante, A^2 s^4 / (kg m^3) = C / (V m)
57  Const_e0       = 8.854187817e-12;
58
59  Const_m0       = 1.2566e-6;                          // magnetische Feldkonstante, kg m / (A^2 s^2) = N / A^2
60
61  Const_vCs     = 9192631770;                         // Freq 133-Cs, Hz
62  Const_Km       = 683;                                // Strahlungsausbeute 540
   THz (5550 nm), lm/W
63
64 FUNCTION FloatStr(Number: float; ValidFigures: BYTE): STRING;
65 { create a string from real number, having defined length. Minimum length
   is 8 }
66
```

```

67 FUNCTION ReadFloat(VAR Medium: TEXT): float;
68
69 FUNCTION ReadInt(VAR Medium: TEXT): LONGINT;
70
71 FUNCTION RoundTo(Number: float; RoundAfter: float): float;
72 { Bsp: RoundTo(5.4321, 0.01) = 5.43 }
73
74 FUNCTION WriteErrorMessage(TEXT: STRING): CHAR;
75
76 FUNCTION Signum(x: float): INTEGER;
77 { +1 for x > 0, -1 for x < 0 and 0 for x=0 }
78
79 FUNCTION Max(a, b: float): float;
80 { Maximal Value of two numbers }
81
82 FUNCTION Max(a, b: LONGINT): LONGINT;
83
84 FUNCTION Min(a, b: float): float;
85 { minimal value of two numbers }
86
87 FUNCTION Min(a, b: LONGINT): LONGINT;
88
89 FUNCTION Pythag(a, b: float): float;
90 { computes sqrt(sqr(a) + sqr(b)) without over/undeflow }
91
92 FUNCTION GCD(n1, n2: LONGINT): LONGINT;
93 { greatest common denominator }
94
95 FUNCTION SCM(n1, n2: LONGINT): LONGINT;
96 { smallest common multiple }
97
98 FUNCTION DecimalFraction(a: float; VAR Zaehler, Nenner: WORD): BOOLEAN;
99 { Umwandlung eines Dezimalbruches a in Fraction aus ganzen Zahlen. Funktion
   wird
   true, wenn abs(a - Zaehler / Nenner) < MaxError }
100
101
102 FUNCTION Pot(Basis: float; Exponent: LONGINT): float;
103 { Potenzfunktion mit ganzzahligem Exponenten }
104
105 FUNCTION Pot(Basis, Exponent: float): float;
106 { Potenz-Funktion, Basis und Exponent reel }
107
108 FUNCTION log(x, Basis: float): float;
109 { Allgemeine log Funktion mit beliebiger Basis }
110
111 FUNCTION xCoordinate(x1, y1, x2, y2, x3, y3, x4, y4: float): float;

```

## 1. Basic math routines

```
112 { x-coordinate of intersection between lines (x1, y1), (x2, y2) and
113   (x3, y3), (x4, y4) }

114
115 FUNCTION yCoordinate(x1, y1, x2, y2, x3, y3, x4, y4: float): float;
116 { y-coordinate of intersection }

117
118 {***** Winkel-Funktionen *****}
119
120 FUNCTION Grad(x: float): float;
121 {Radian nach Grad}

122
123 FUNCTION Rad(x: float): float;
124 { Grad nach Radian }

125
126 FUNCTION coth(x: float): float;
127 { cotangens hyperbolicus }

128
129 FUNCTION sech(x: float): float;
130 { secans hyperbolicus }

131
132 FUNCTION csch(x: float): float;
133 { cosecans hyperbolicus }

134
135 FUNCTION ArCoth(x: float): float;
136 { area cotangens hyperbolicus }

137
138 FUNCTION ArSech(x: float): float;
139 { area secans hyperbolicus }

140
141 FUNCTION ArCsch(x: float): float;
142 { area cosecans hyperbolicus }

143
144 FUNCTION Sec(x: float): float;
145 { Secans }

146
147 FUNCTION Csc(x: float): float;
148 { Cosecans }

149
150 FUNCTION ArcCot(x: float): float;
151 {arcus cotangens}

152
153 FUNCTION ArcSec(x: float): float;
154 {Arcus Secans}

155
156 FUNCTION ArcCsc(x: float): float;
157 { Arcus Cosecans }
```

```

158
159 {***** Funktionen für Statistik *****}
160
161 FUNCTION LnGamma(x: float): float;
162 { Logarithmus der Gamma-Funktion }
163
164 FUNCTION fak(i: BYTE): LONGINT;
165
166 FUNCTION LnFak(x: float): float;
167 { Log(x!) calculated via gamma-function }
168
169 FUNCTION IncompleteGamma(a, x: float): float;
170 { Numerical Recipes p. 182 }
171
172 FUNCTION BinomialCoef(n, k: LONGINT): float;
173 { Binomial-Koeffizient, Anzahl der Möglichkeiten, k Elemente aus n
  auszuwählen }
174
175 {***** Diverses *****}
176
177 FUNCTION HexByte(Number: BYTE): ByteStr;
178 { Umwandlung einer Number von Dezimal nach Hexadezimal }
179
180 FUNCTION InWorten(Number: LONGINT): STRING;
181 { Wandelt eine Number in Worte um (für Finanzmathematik). Es muß gelten:
  abs(Number) < 1.000.000 }
182
183 FUNCTION Modulo11(Number: LONGINT): CHAR;
184 { gibt den Übercode nach dem modulo-11 Verfahren zurück(eine Ziffer von 0..9
  oder den Buchstaben "X") \parencite{Mic-96}. }
185
186
187 FUNCTION TestModulo11(Number: LONGINT; TEST: CHAR): BOOLEAN;
188 { Testet ob eine Number zu ihrer Überziffer passt }
189
190
191
192 IMPLEMENTATION
193
194 VAR
195   CH: CHAR;

```

## 1.1. Administrative functions

The routines of this unit can pass error conditions to the calling program by setting an error flag **MathError**. In addition, an error message is produced:

Listing 1.2: Error handling

## 1. Basic math routines

```

1 FUNCTION WriteErrorMessage(Text: STRING): CHAR;
2
3 BEGIN
4   Write(Text);
5   ReadLn(Result);
6 END;
```

The calling program can then try to correct the error and then reset the error flag, or it can gracefully abort. User input is passed on as single character, the error message could contain a question *a la*: “Abort, ignore, continue?” All units described here use this mechanism.

Conversion of a number  $x \in \mathbb{R}$  into a string for output is straight forward. The number of decimal places can be set, **Nan** is handled:

Listing 1.3: Conversion of floating point number to string

```

1 FUNCTION FloatStr(Number: float; ValidFigures: BYTE): STRING;
2
3 VAR
4   hst: STRING;
5   i: BYTE;
6
7 BEGIN
8   IF IsNaN(Number)
9     THEN
10    BEGIN
11      hst := ' ';
12      FOR i := 1 TO (ValidFigures - 3) DO
13        hst := hst + ' ';
14      Result := hst + 'NaN';
15      EXIT;
16    END;
17   IF Abs(Number) < Zero
18     THEN Number := 0.0;
19   Str(Number: ValidFigures, hst);
20   Result := hst;
21 END;
```

The following routine reads a whole number (anything assignment compatible with longint) from a file. Characters that separate numbers are in the set **SepChar**.

Listing 1.4: Read whole numbers from file

```

1 FUNCTION ReadInt(VAR Medium: TEXT): LONGINT;
2
3 CONST
4   SepChar: SET OF CHAR = [';', ','] ; // separator between numbers
5   NumChar: SET OF CHAR = ['0'..'9', '-'];
```

```

7  VAR
8    s: STRING;
9    x: float;
10   Error: INTEGER;
11
12  BEGIN
13    s := '';
14  REPEAT
15    Read(Medium, c);
16    IF (c IN NumChar)
17      THEN s := s + c;
18  UNTIL (c IN SepChar) OR EoLn(Medium);
19  Val(s, x, Error);
20  IF Error <> 0
21    THEN
22      BEGIN
23        ch := WriteErrorMessage("'" + s + "' is not a number");
24        MathError := TRUE;
25        EXIT;
26      END;
27  Result := Round(x);
28 END;

```

Reading floating point numbers works in the same way, however, there are additional complications:

- we need to handle **NaN** (expressed as 'NA' or 'NaN')
- we need to handle numbers written as potency of 10
- in different countries, different characters are used as decimal separator
- this implies different number separators in .csv-files

Listing 1.5: Read floating point number from file

```

1  FUNCTION ReadFloat(VAR Medium: TEXT): float;
2
3  CONST
4    SepChar: SET OF CHAR = [';'];           // separator between numbers
5    NumChar: SET OF CHAR = ['0'..'9', '- ', '+ ', 'E', 'e', 'N', 'A', 'n', 'a'];
6    DecSep = [',', '.'];      // decimal separator
7
8  VAR
9    s: STRING;
10   x: float;
11   Error: INTEGER;
12

```

## 1. Basic math routines

```

13  BEGIN
14    s := '';
15  REPEAT
16    Read(Medium, c);
17    IF (c IN DecSep)
18      THEN s := s + '.';
19    ELSE IF (c IN NumChar)
20      THEN s := s + c
21    ELSE IF (c IN SepChar) OR (c = ' ')
22      THEN // DO nothing
23    ELSE // illegal character
24      BEGIN
25        s := s + c;
26        ch := WriteErrorMessage(s + ' is not a number');
27        MathError := TRUE;
28        EXIT;
29      END;
30  UNTIL (c IN SepChar) OR EoLn(Medium);
31  IF ((UpCase(s) = 'NA') OR (UpCase(s) = 'NAN'))
32  THEN
33    BEGIN
34      ReadFloat := NaN;
35      EXIT;
36    END;
37  Val(s, x, Error);
38  IF Error <> 0
39  THEN
40    BEGIN
41      ch := WriteErrorMessage(s + ' is not a number');
42      MathError := TRUE;
43      EXIT;
44    END;
45  Result := x;
46 END;

```

Rounding is accomplished to a given precision:

```

1  FUNCTION RoundTo(Number: float; RoundAfter: float): float;
2
3  BEGIN
4    Result := RoundAfter * Int(Number / RoundAfter + 0.5);
5  END;

```

For example, `Round(5.4321, 0.01)` results in 5.43.

The signum function returns  $\pm 1$  depending on the sign of  $x$ , or 0 if  $x$  is very small:

Listing 1.6: Signum function

```

1  FUNCTION Signum(x: float): INTEGER;

```

```

2
3 BEGIN
4   IF (x < -Zero)
5     THEN Result := -1
6   ELSE IF (x > Zero)
7     THEN Result := 1
8   ELSE Result := 0;
9 END;

```

“Very small” is defined by the typed constant `Zero`, which can be adjusted by the calling program according to needs.

## 1.2. Integer arithmetic

The following routines calculate the largest common denominator, the smallest common multiple and convert a decimal number into a fraction or a Hex-string:

Listing 1.7: Integer arithmetic

```

1 FUNCTION GCD(n1, n2: LONGINT): LONGINT;
2
3 VAR
4   r: LONGINT;
5
6 BEGIN
7   IF n2 <> 0
8     THEN
9       BEGIN
10      REPEAT
11        r := n1 MOD n2;
12        n1 := n2;
13        n2 := r;
14      UNTIL r = 0;
15      Result := n1;
16    END
17  ELSE
18    IF n1 = 0
19    THEN
20      Result := 0           { GCD(0,0) = 0 }
21    ELSE
22      BEGIN
23        WriteErrorMessage('division by 0');
24        MathError := TRUE;
25        EXIT;
26      END;
27 END;
28

```

## 1. Basic math routines

```
29
30 FUNCTION SCM(n1, n2: LONGINT): LONGINT;
31
32 VAR
33     n: LONGINT;
34
35 BEGIN
36     n := GCD(n1, n2);
37     IF n <> 0
38         THEN Result := n1 * n2 DIV GCD(n1, n2)
39     ELSE Result := 0;
40 END;
41
42
43 FUNCTION DecimalFraction(a: float; VAR Zaehler, Nenner: WORD): BOOLEAN;
44
45 VAR
46     temp1, temp2, Teil1, Teil2: float;
47     Iter, Num: WORD;
48
49 BEGIN
50     iter := 0;
51     Temp1 := a;
52     Teil1 := 0.0;
53     Teil2 := 1.0;
54     Zaehler := 1;
55     Nenner := 0;
56     REPEAT
57         INC(Iter);
58         Num := Trunc(Temp1);
59         temp2 := Zaehler;
60         Zaehler := Round(Num * Zaehler + Teil1);
61         Teil1 := Temp2;
62         Temp2 := Nenner;
63         Nenner := Round(Num * Nenner + Teil2);
64         Teil2 := Temp2;
65         Temp1 := 1 / (Temp1 - Num);
66     UNTIL (iter > MaxIter) OR (Abs(a - Zaehler / Nenner) < MaxError);
67     Result := Iter <= MaxIter;
68 END;
69
70
71 FUNCTION HexByte(Number: BYTE): ByteStr;
72
73 CONST
74     Ziffern: ARRAY [0..15] OF CHAR = '0123456789ABCDEF';
```

```

75
76 BEGIN
77   HexByte[1] := Ziffern[Number SHR 4];
78   HexByte[2] := Ziffern[Number AND $0F];
79 END;

```

## 1.3. Maximum and minimum of two numbers

Here we use overloading so that both integer and real variables can be used:

Listing 1.8: Maximum and minimum

```

1 FUNCTION Max(a, b: float): float;
2
3 BEGIN
4   IF (a > b)
5     THEN Result := a
6     ELSE Result := b;
7 END;
8
9 FUNCTION Min(a, b: float): float;
10
11 BEGIN
12   IF (a < b)
13     THEN Result := a
14     ELSE Result := b;
15 END;
16
17
18 FUNCTION Max(a, b: LONGINT): LONGINT;
19
20 BEGIN
21   IF (a > b)
22     THEN Result := a
23     ELSE Result := b;
24 END;
25
26 FUNCTION Min(a, b: LONGINT): LONGINT;
27
28 BEGIN
29   IF (a < b)
30     THEN Result := a
31     ELSE Result := b;
32 END;

```

1. Basic math routines

## 1.4. Exponents, logarithms, powers

$f(x, y) = x^y$  is defined in different ways depending on whether  $x, y \in \mathbb{R}$  or  $y \in \mathbb{N}$ :

$$y = 0 \ z = x^0 = 1.$$

$$x = 0 \ z = 0^y = 0, \text{ except for } y = 0.$$

$$y \in \mathbb{N}$$

$$y > 0 \ z = x \times x \times \dots \times x, \text{ } y \text{ times.}$$

$$y < 0 \ z = 1/x \times 1/x \times \dots \times 1/x, \text{ } -y \text{ times.}$$

$$y \in \mathbb{R}$$

$$x > 0 \ z = \exp(\ln(x) * y).$$

$x < 0 \ z$  undefined, raise error condition.

This is easiest done by overloading the function for the case exponent  $\in \mathbb{N}$  or  $\in \mathbb{R}$ :

Listing 1.9: Universal power function

```

1  FUNCTION Pot(Basis: float; Exponent: LONGINT): float;
2
3  VAR
4      Product: float;
5      i: LONGINT;
6
7  BEGIN
8      IF Exponent = 0
9          THEN
10             BEGIN
11                 Result := 1;
12                 EXIT;
13             END;
14     IF Abs(Basis) < Zero
15         THEN
16             BEGIN
17                 Result := 0;
18                 EXIT;
19             END;
20     IF Exponent < 0
21         THEN
22             BEGIN
23                 Basis := 1 / Basis;
24                 Exponent := -Exponent;
25             END;
26     Product := 1;
27     FOR i := 1 TO Exponent DO

```

```

28     Product := Product * Basis;
29     Result := Product;
30   END;
31
32 FUNCTION Pot(Basis, Exponent: float): float;
33
34 BEGIN
35   IF Abs(Exponent) < Zero
36     THEN
37       BEGIN
38         Result := 1;
39         EXIT;
40       END;
41   IF Abs(Basis) < Zero
42     THEN
43       BEGIN
44         Result := 0;
45         EXIT;
46       END;
47   IF Basis > Zero      // Basis > 0
48     THEN
49       Result := Exp(Ln(Basis) * Exponent)
50   ELSE
51     BEGIN
52       CH := WriteErrorMessage('Power negative base and real exponent');
53       Matherror := TRUE;
54     END;
55 END;

```

Listing 1.10: Universal log

```

1  FUNCTION log(x, Basis: float): float;
2
3 BEGIN
4   IF x > Zero
5     THEN
6       Result := Ln(x) / Ln(Basis)
7   ELSE
8     BEGIN
9       CH := WriteErrorMessage('log(x) mit x < 0');
10      EXIT;
11    END;
12 END;

```

Listing 1.11: Pythagoras

```

1  FUNCTION Pythag(a, b: float): float;
2

```

## 1. Basic math routines

```
3  VAR
4      at, bt: float;
5
6  BEGIN
7      at := Abs(a);
8      bt := Abs(b);
9      IF (at > bt)
10         THEN
11             Result := at * Sqrt(1.0 + Sqr(bt / at))
12         ELSE
13             IF (bt = 0)
14                 THEN Result := 0
15             ELSE Result := bt * Sqrt(1.0 + Sqr(at / bt));
16 END;
```

## 1.5. Intersection of two lines

If we have two lines, given by two points each, we can calculate the coordinates of their intersection. This is for example used for the “direct plot” of EISENTHAL & CORNISH-BOWDEN [1, 2].

Listing 1.12: Intersection of lines

```
1  FUNCTION xCoordinate(x1, y1, x2, y2, x3, y3, x4, y4: float): float;
2      // x-coordinate OF intersection OF between lines given by 2 points each
3
4  BEGIN
5      Result := ((x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (x3 * y4 - y3 *
6          x4)) /
7      ((x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4));
8
9  FUNCTION yCoordinate(x1, y1, x2, y2, x3, y3, x4, y4: float): float;
10
11 BEGIN
12     Result := ((x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (y3 * y4 - y3 *
13         x4)) /
14     ((x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4));
15 END;
```

## 1.6. Conversion of angles between rad and degrees

```
1  FUNCTION Grad(x: float):
2      {Bogenma in Altgrad umrechnen}
```

```

3
4   BEGIN
5     Result := x * 180 / Pi;
6   END;
7
8
9   FUNCTION Rad(x: float): float;
10  {Altgrad in Bogenma umrechnen}
11
12  BEGIN
13    Result := x * Pi / 180;
14  END;

```

## 1.7. Hyperbolic functions

```

1   FUNCTION coth(x: float): float;
2
3   VAR
4     z: float;
5
6   BEGIN
7     z := sinh(x);
8     IF (z <> 0)
9       THEN
10      Result := cosh(x) / z
11    ELSE
12      BEGIN
13        CH := WriteErrorMessage('coth with sinh(x) = 0');
14        MathError := TRUE;
15      END;
16  END;
17
18
19  FUNCTION Sech(x: float): float;
20
21  BEGIN
22    Result := Sqrt(1 - Sqr(tanh(x)));
23  END;
24
25
26  FUNCTION Csch(x: float): float;
27
28  BEGIN
29    Result := Sqrt(Sqr(coth(x)) - 1);
30  END;

```

## 1.8. Area functions

```

1  FUNCTION ArCoth(x: float): float;
2
3  BEGIN
4    IF Abs(x) > 1.0
5    THEN
6      ArCoth := 0.5 * Ln((x + 1) / (x - 1))
7    ELSE
8      BEGIN
9        CH := WriteErrorMessage('arcoth(x) with x not from [-1..1]');
10       MathError := TRUE;
11     END;
12   END;
13
14
15  FUNCTION ArSech(x: float): float;
16
17  VAR
18    y: float;
19
20  BEGIN
21    y := 1 / x + Sqrt(1 / Sqr(x) - 1);
22    Result := log10(y);
23  END;
24
25
26  FUNCTION ArCsch(x: float): float;
27
28  VAR
29    y: float;
30
31  BEGIN
32    y := 1 / x + Sqrt(1 / Sqr(x) + 1);
33    Result := log10(y);
34  END;

```

## 1.9. Cyclometric functions

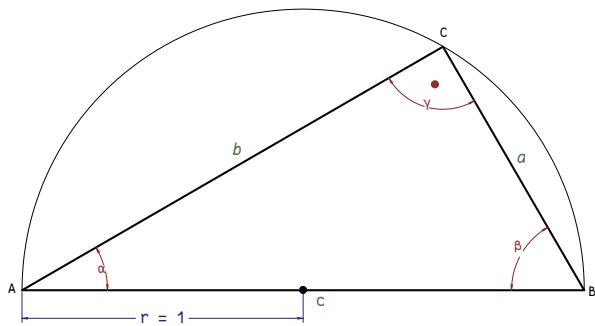
For the sake of completeness, here some functions that are defined neither in standard Pascal, nor in the Lazarus `math` library:

Listing 1.13: Cyclometric functions

```

1  FUNCTION Sec(x: float): float;
2    {Secans}

```



Function	Definition	Condition
$\sin(\alpha)$	$a/c$	$(c \neq 0)$
$\cos(\alpha)$	$b/c$	$(c \neq 0)$
$\tan(\alpha)$	$a/b = 1/\cot(\alpha)$	$(b \neq 0)$
$\cot(\alpha)$	$b/a = 1/\tan(\alpha)$	$(a \neq 0)$
$\sec(\alpha)$	$c/b = 1/\cos(\alpha)$	$(b \neq 0)$
$\csc(\alpha)$	$c/a = 1/\sin(\alpha)$	$(a \neq 0)$

Figure 1.1.: Definition of the cyclometric functions.

```

3
4  VAR
5    c: float;
6
7  BEGIN
8    c := Cos(x);
9    IF c = 0
10   THEN
11     BEGIN
12       CH := WriteErrorMessage('sec(x) with x = °90 or x = °270');
13       MathError := TRUE;
14     END
15   ELSE
16     Result := 1 / c;
17 END;
18
19
20 FUNCTION Csc(x: float): float;
21
22  VAR
23    c: float;
24
25  BEGIN
26    c := Sin(x);
27    IF c = 0
28    THEN
29     BEGIN
30       CH := WriteErrorMessage('csc(x) with x = °90 or x = °270');
31       MathError := TRUE;
32     END

```

## 1. Basic math routines

```
33     ELSE
34         Result := 1 / c;
35     END;
36
37 FUNCTION ArcCot(x: float): float;
38
39 BEGIN
40     Result := Pi / 2 - ArcTan(x);
41 END;
42
43
44 FUNCTION ArcSec(x: float): float;
45
46 BEGIN
47     IF (x = 0) OR (Abs(x) > 1)
48     THEN
49         BEGIN
50             CH := WriteErrorMessage('arcsec(x) with x not from [-1..1] or x =
51                           0');
52             MathError := TRUE;
53         END
54     ELSE
55         Result := ArcCos(1 / x);
56     END;
57
58 FUNCTION ArcCsc(x: float);
59
60 BEGIN
61     IF (x = 0) OR (Abs(x) > 1)
62     THEN
63         BEGIN
64             CH := WriteErrorMessage('arccsc(x) with x not from [-1..1] or x =
65                           0');
66             MathError := TRUE;
67         END
68     ELSE
69         Result := ArcSin(1 / x);
70     END;
```

## 1.10. Routines required for statistics

Even rather small numbers have large faculties, that can cause number overflows. The use of logarithms avoids that problem:

Listing 1.14: logarithm of important functions

```

1 FUNCTION LnGamma(x: float): float;
2
3 CONST
4   a0 = 0.083333333096;
5   a1 = -0.002777655457;
6   a2 = 0.000777830670;
7   c = 0.918938533205;
8
9 VAR
10  r: float;
11
12 BEGIN
13   r := (a0 + (a1 + a2 / Sqr(x)) / Sqr(x)) / x;
14   Result := (x - 0.5) * Ln(x) - x + c + r;
15 END;
16
17
18 FUNCTION LnFak(x: float): float;
19
20 VAR
21  z: float;
22
23 BEGIN
24   z := x + 1;
25   Result := LnGamma(z);
26 END;
```

Listing 1.15: faculties (iterative)

```

1 FUNCTION fak(i: BYTE): LONGINT;
2
3 VAR
4   j: BYTE;
5   Product: LONGINT;
6
7 BEGIN
8   Product := 1; { fak(0) und fak(1) }
9   FOR j := 2 TO i DO
10    Product := Product * j;
11   Result := Product;
12 END;
```

The incomplete  $\gamma$ -function is calculated according to [3, p. 182], using either a series representation that converges quickly for  $x < a+1$ , or a continued fraction representation that converges quickly for  $x > a+1$ .

1. Basic math routines

### 1.10.1. Incomplete gamma-function

Listing 1.16: Incomplete gamma function

```

1  FUNCTION IncompleteGamma(a, x: float): float;
2
3  VAR
4      GamSer, GamCF, gln: float;
5
6  PROCEDURE GSer(a, x: float; VAR GamSer, gln: float);
7      { series representation of incomplete gamma function, returns gamma in
8          GamSer,
9          ln(gamma) in gln. Converges quickly for x < succ(a) }
10
11
12  VAR
13      n: WORD;
14      Sum, del, ap: float;
15
16  BEGIN
17      gln := LnGamma(a);
18      IF (x <= 0)
19          THEN
20              BEGIN
21                  IF (x < 0)
22                      THEN
23                          BEGIN
24                              ch := WriteErrorMessage('Error: Incomplete gamma-function:
25                                  x < 0');
26                          EXIT;
27                      END;
28                  GamSer := 0;
29              END
30      ELSE
31          BEGIN
32              ap := a;
33              Sum := 1 / a;
34              del := Sum;
35              FOR n := 1 TO MaxIter DO
36                  BEGIN
37                      ap := ap + 1;
38                      del := del * x / ap;
39                      Sum := Sum + del;
40                      IF (Abs(del) < Abs(Sum) * MaxError)
41                          THEN
42                              BEGIN
43                                  ch := WriteErrorMessage(
44                                      'Error in incomplete gamma-function: no

```

```

        convergence ');
42     MathError := TRUE;
43     EXIT;
44   END;
45 END;
46 GamSer := Sum * Exp(-x + a * Ln(x) - gln);
47 END;
48 END;

50 PROCEDURE gcf(a, x: float; VAR GamCF, gln: float);
51 { continued fraction representation of incomplete gamma function, returns
52   gamma in GamSer, ln(gamma) in gln. Converges quickly for x > succ(a) }
53
54 VAR
55   n: WORD;
56   gOld, g, fac, b1, b0, anf, ana, an, a1, a0: float;
57
58 BEGIN
59   gln := LnGamma(a);
60   gOld := 0.0;
61   a0 := 1.0;
62   a1 := x;
63   b0 := 0.0;
64   b1 := 1.0;
65   fac := 1.0;
66   FOR n := 1 TO MaxIter DO
67     BEGIN
68       an := 1.0 * n;
69       ana := an - a;
70       a0 := (a1 + a0 * ana) * fac;
71       b0 := (b1 + b0 * ana) * fac;
72       anf := an * fac;
73       a1 := x * a0 + anf * a1;
74       b1 := x * b0 + anf * b1;
75       IF (a1 <> 0.0)
76         THEN // renormalise
77           BEGIN
78             fac := 1.0 / a1;
79             g := b1 * fac;
80             IF (Abs((g - gOld) / g) < MaxError)
81               THEN // convergence reached
82                 BEGIN
83                   GamCF := Exp(-x + a * Ln(x) - gln) * g;
84                   EXIT;
85                 END;
86             gOld := g;

```

## 1. Basic math routines

```

87          END;
88      END; // FOR
89      ch := WriteErrorMessage('Error in incomplete gamma-function: no
90                           convergence');
91      MathError := TRUE
92  END;

93 BEGIN
94 IF (x < 0) OR (a <= 0)
95 THEN
96 BEGIN
97     ch := WriteErrorMessage('Incomplete Gamma-function: arguments <=
98           0');
99     MathError := TRUE;
100    EXIT;
101 END;
102 IF (x < (a + 1.0))
103 THEN
104 BEGIN
105     GSer(a, x, GamSer, gln);
106     Result := 1.0 - GamSer;
107 END
108 ELSE
109 BEGIN
110     gcf(a, x, GamCF, gln);
111     Result := GamCF;
112 END;
113 END;

```

### 1.10.2. Binomial coefficients

The binomial coefficient is the number of ways, in which  $k$  items can be chosen out of a set of  $n$ .

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \prod_{i=1}^k \frac{n+1-i}{i} \quad (1.1)$$

The latter expression switches between multiplication and division, and hence avoids large numbers as much as possible. However, binomial coefficients – even of relatively small numbers – can be quite large, hence we use the float data type to avoid overflow.

Listing 1.17: binomial coefficient

```

1 FUNCTION BinomialCoef(n, k: LONGINT): float;
2
3 VAR i,n1 : LONGINT;
4     Res     : float;
5

```

```

6 BEGIN
7   IF n < k           // check FOR trivial cases first
8     THEN Result := 0
9   ELSE IF (n = k) OR (k = 0)
10    THEN Result := 1
11   ELSE IF k = 1
12     THEN Result := n
13   ELSE // DO the real work
14     BEGIN
15       IF (2*k) > n THEN k := n-k;
16       Res := 1;
17       n1 := Succ(n);
18       FOR i := 1 TO k DO
19         Res := Res * (n1 - i) / i;
20       Result := Res;
21     END;
22 END;

```

## 1.11. Finance mathematics (with a German flavour)

This routines converts a number into words

Listing 1.18: Finance mathematics

```

1 FUNCTION InWorten(Number: LONGINT): STRING;
2
3 CONST
4   Einer: ARRAY [1..9] OF STRING[6] =
5     ('ein', 'zwei', 'drei', 'vier', 'fünf', 'sechs', 'sieben',
6      'acht', 'neun');
7   Zehner: ARRAY [1..9] OF STRING[8] =
8     ('zehn', 'zwanzig', 'Bdreiig', 'vierzig', 'fünfzig',
9      'sechzig', 'siebzig', 'achtzig', 'neunzig');
10  Elfer: ARRAY [1..9] OF STRING[9] =
11    ('elf', 'özwlf', 'dreizehn', 'vierzehn', 'fünfzehn', 'sechzehn',
12      'siebzehn', 'achtzehn', 'neunzehn');
13
14 VAR
15   Minus: BOOLEAN;
16   ZahlStr: STRING;
17   ValueFeld: ARRAY [1..6] OF BYTE;
18
19 BEGIN
20   IF Number >= 1000000
21     THEN
22       BEGIN

```

## 1. Basic math routines

```
23          InWorten := ('üngültiger Bereich ');
24          EXIT;
25      END;
26  IF Number < 0
27  THEN
28      BEGIN
29          Minus := TRUE;
30          Number := Abs(Number);
31      END
32  ELSE
33      Minus := FALSE;
34  ValueFeld[1] := Number DIV 100000;      { Ziffern der Number isolieren }
35  Number := Number MOD 100000;
36  ValueFeld[2] := Number DIV 10000;
37  Number := Number MOD 10000;
38  ValueFeld[3] := Number DIV 1000;
39  Number := Number MOD 1000;
40  ValueFeld[4] := Number DIV 100;
41  Number := Number MOD 100;
42  ValueFeld[5] := Number DIV 10;
43  ValueFeld[6] := Number MOD 10;
44  ZahlStr := '';
45          { String aufbauen, dabei die
        Besonderheiten }
46  IF ValueFeld[1] > 0
47  THEN ZahlStr := ZahlStr + Einer[ValueFeld[1]] + 'hundert';
48 CASE ValueFeld[2] OF
49  2..9: BEGIN
50      IF ValueFeld[3] = 0
51      THEN ZahlStr := ZahlStr + Zehner[ValueFeld[2]] + 'tausend'
52      ELSE ZahlStr := ZahlStr + Einer[ValueFeld[3]] + 'und' +
53          Zehner[ValueFeld[2]] + 'tausend';
54  END;
55  1: BEGIN
56      IF ValueFeld[3] = 0
57      THEN ZahlStr := ZahlStr + 'zehntausend'
58      ELSE ZahlStr := ZahlStr + Elfer[ValueFeld[3]] + 'tausend';
59  END;
60  0: BEGIN
61      IF ValueFeld[3] = 0
62      THEN
63          IF ZahlStr <> ''
64          THEN ZahlStr := ZahlStr + 'tausend'
65          ELSE
66              ZahlStr := ZahlStr + Einer[ValueFeld[3]] + 'tausend';
67  END;
```

```

68 END; // CASE
69 IF ValueFeld[4] > 0
70 THEN ZahlStr := ZahlStr + Einer[ValueFeld[4]] + 'hundert';
71 CASE ValueFeld[5] OF
72 2..9: BEGIN
73     IF ValueFeld[6] = 0
74     THEN ZahlStr := ZahlStr + Zehner[ValueFeld[5]]
75     ELSE ZahlStr := ZahlStr + Einer[ValueFeld[6]] + 'und' +
76         Zehner[ValueFeld[5]];
77 END;
78 1: BEGIN
79     IF ValueFeld[6] = 0
80     THEN ZahlStr := ZahlStr + 'zehn'
81     ELSE ZahlStr := ZahlStr + Elfer[ValueFeld[6]];
82 END;
83 0: BEGIN
84     CASE ValueFeld[6] OF
85     0 : IF ZahlStr = '' THEN ZahlStr := 'null';
86     1 : ZahlStr := ZahlStr + 'eins';
87     2..9: ZahlStr := ZahlStr + Einer[ValueFeld[6]];
88     END;
89 END;
90 ZahlStr[1] := UpCase(Zahlstr[1]);
91 IF Minus THEN ZahlStr := 'minus ' + ZahlStr;
92 Result := ZahlStr;
93 END;

```

The modulo-11 number (a figure in 0..9 or the letter "X") is used for example to check the correctness of credit card numbers [4].

Listing 1.19: test number by modulo-11 method

```

1 FUNCTION Modulo11(Number: LONGINT): CHAR;
2
3 CONST
4     ResultArr: ARRAY[0..10] OF CHAR =
5         ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'X');
6
7 VAR
8     i, j, Sum: LONGINT;
9
10 BEGIN
11     i := 1;
12     Number := Abs(Number);
13     Sum := 0;
14     WHILE Number > 0 DO
15         BEGIN

```

## 1. Basic math routines

```
16      j := Number MOD 10;
17      Number := Number DIV 10;
18      Sum := Sum + i * j;
19      i := i * 2;
20  END;
21  j := Sum MOD 11;
22  Result := ResultArr[j];
23 END;
24
25
26 FUNCTION TestModulo11(Number: LONGINT; TEST: CHAR): BOOLEAN;
27
28 BEGIN
29     TestModulo11 := (Modulo11(Number) = TEST);
30 END;
```

## 1.12. Functions already in unit `math`

Lazarus comes with the `math` unit, which contains many mathematical functions. In other compilers, these have to be replaced by own code. Then the following lines have to be added to the interface of `MathFunc`:

```
1 function ArSinh (x: float): float;
2
3 function ArCosh (x: float): float;
4
5 function ArTanh (x: float): float;
6
7 function sinh (x: float): float;
8
9 function cosh (x: float): float;
10
11 function tanh (x: float): float;
12
13 function tan (x: float): float;
14
15 function cotan (x: float): float;
16
17 function ArcSin (x: float): float;
18
19 function ArcCos (x: float): float;
20
21 function ArcTan2 (x, y : float) : float;
22
23 function log (x, Basis : float) : float;
```

and the following lines to the implementation:

```

1  function ArSinh (x: float): float;
2
3  begin
4      Result := Ln(x + Sqrt(Sqr(x) + 1));
5  end;
6
7
8  function ArCosh (x: float): float;
9
10 begin
11     if x < 1.0
12         then
13             begin
14                 ch := WriteErrorMessage('Power of Base < 0 and broken exponent');
15                 MathError := true;
16             end
17         else
18             Result := Ln(x + Sqrt(Sqr(x) - 1));
19     end;
20
21
22  function ArTanh (x: float): float;
23
24  begin
25      if abs(x) < 1.0
26          then
27              Result := 0.5 * Ln((1 + x) / (1 - x))
28          else
29              begin
30                  ch := WriteErrorMessage('artanh(x) with x not from [-1..1]');
31                  MathError := true;
32                  exit;
33              end;
34  end;
35
36  function sinh (x: float): float;
37
38  begin
39      x := exp(x);
40      Result := 0.5 * (x - 1 / x);
41  end;
42
43
44  function cosh (x: float): float;
45
```

## 1. Basic math routines

```
46  begin
47    x := exp(x);
48    Result := 0.5 * (x + 1 / x);
49  end;
50
51
52  function tanh (x: float): float;
53
54  begin
55    Result := sinh(x) / cosh(x);
56  end;
57
58
59  function tan (x: float): float;
60
61  begin
62    if Cos(x) <> 0
63    then
64      Result := Sin(x) / (Cos(x))
65    else
66      begin
67        ch := WriteErrorMessage('tan(x) with x = °90 or x = °270');
68        MathError := true;
69      end;
70  end;
71
72
73  function ArcSin (x: float): float;
74
75  begin
76    if abs(x) > 1.0
77    then
78      begin
79        ch := WriteErrorMessage('arcsin(x) with x not from [-1..1]');
80        MathError := true;
81      end
82    else
83      if abs(x) < 1.0
84      then Result := ArcTan(x / Sqr(1 - Sqr(x)))
85      else Result := x * Pi / 2
86  end;
87
88
89  function ArcCos (x: float): float;
90
91  begin
```

```

92  if abs(x) > 1.0
93  then
94    begin
95      ch := WriteErrorMessage('arccos(x) with x not from [-1..1]');
96      MathError := true;
97    end
98  else
99    Result := Pi / 2 - ArcSin(x);
100 end;
101
102
103 function cotan (x: float): float;
104
105 VAR tmp: float;
106
107 begin
108   tmp := Int(2 * x / Pi);
109   if (tmp * Pi / 2) <> x
110   then
111     Result := 1 / tan(x)
112   else
113     begin
114       ch := WriteErrorMessage('cot(x) with x not from [0..pi]');
115       MathError := true;
116     end;
117 end;
118
119
120 function arctan2 (x, y : float) : float;
121
122 begin
123   if x > 0
124   then
125     Result := arctan(y/x)
126   else
127     if x < 0
128     then Result := arctan(y/x) + pi
129     else Result := pi/2 * signum(y);
130 end;

```

## References

- [1] A. CORNISH-BOWDEN, R. EISENTHAL: Statistical considerations in the estimation of enzyme kinetic parameters by the direct linear plot and other methods, *Biochem.*

*J.* **139**:3 (1974), 721–730 DOI: [10.1042/bj1390721](https://doi.org/10.1042/bj1390721) URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1166336/pdf/biochemj00583-0245.pdf>.

- [2] R. EISENTHAL, A. CORNISH-BOWDEN: The direct linear plot. A new graphical procedure for estimating enzyme kinetic parameters, *Biochem. J.* **139**:3 (1974), 715–720 DOI: [10.1042/bj1390715](https://doi.org/10.1042/bj1390715) URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1166335/pdf/biochemj00583-0239.pdf>.
- [3] W.H. PRESS et al.: *Numerical recipes in Pascal: The art of scientific computing* Cambridge: Cambridge University Press, 1989 ISBN: 9780521375160.
- [4] J. MICHAEL: Mit Sicherheit, *c't*:7 (1996), 264–268.

## 2. Complex numbers

### Abstract

Under standard Pascal, functions could only return simple data types. Complex numbers are – in essence – a record. However, Object Pascal allows such compound data types to be returned by functions.

The routines are based on [1, 2]. The interface is:

Listing 2.1: Interface of `Complex`

```
1 UNIT Complex;
2
3 INTERFACE
4
5 USES Math, mathfunc;
6
7 TYPE
8   ComplexTyp = RECORD
9     RealPart,
10    ImagPart: float;
11  END;
12
13 CONST Const_i      : ComplexTyp = (RealPart : 0.0; ImagPart : 1.0);
14     Const_Null : ComplexTyp = (RealPart : 0.0; ImagPart : 0.0);
15
16 VAR
17   ComplexError: BOOLEAN = FALSE;      // error-Flag
18
19 { ***** Type conversion ***** }
20
21 FUNCTION ComplexInit(RePart, ImPart: float): ComplexTyp;
22
23 FUNCTION Re(z: ComplexTyp): float;
24 { real part of a ComplexTyp number }
25
26 FUNCTION Im(z: ComplexTyp): float;
27 { imaginary part of a ComplexTyp number }
28
29 { ***** E/A- Routines ***** }
```

## 2. Complex numbers

```
31 FUNCTION ComplexToStr(z: ComplexTyp; m, n: BYTE): STRING;
32
33 { ***** Conversions ***** }
34
35 FUNCTION ComplexIsNull(z: ComplexTyp): BOOLEAN;
36
37 FUNCTION ComplexConj(z: ComplexTyp): ComplexTyp;
38
39 FUNCTION ComplexAbs(z: ComplexTyp): float;
40
41 FUNCTION ComplexArg(z: ComplexTyp): float;
42
43 FUNCTION ComplexInv(z: ComplexTyp): ComplexTyp;
44
45 FUNCTION ComplexNeg(z: ComplexTyp): ComplexTyp;
46
47 PROCEDURE Polar(z: ComplexTyp; VAR r, phi: float);
48
49 FUNCTION Rect(r, phi: float): ComplexTyp;
50
51 { ***** Calculation with ComplexTyp ***** }
52
53 OPERATOR = (z, b: ComplexTyp) : BOOLEAN;
54
55 OPERATOR + (z, b: ComplexTyp): ComplexTyp;
56
57 OPERATOR - (z, b: ComplexTyp): ComplexTyp;
58
59 OPERATOR - (z: ComplexTyp): ComplexTyp; // unary -
60
61 OPERATOR * (z, b: ComplexTyp): ComplexTyp;
62
63 OPERATOR / (z, b: ComplexTyp): ComplexTyp;
64
65 OPERATOR ** (Basis, Exponent: ComplexTyp): ComplexTyp;
66
67 { ***** Calculation with ComplexTyp and Real ***** }
68
69 OPERATOR = (z: ComplexTyp; x: float) : BOOLEAN;
70
71 OPERATOR = (x: float; z: ComplexTyp) : BOOLEAN;
72
73 OPERATOR := (x: real) : ComplexTyp;
74
75 OPERATOR := (z: ComplexTyp) : real;
```

```

77 OPERATOR + (z: ComplexTyp; x: float) : ComplexTyp;
78
79 OPERATOR + (x: float; z: ComplexTyp) : ComplexTyp;
80
81 OPERATOR - (z : ComplexTyp; r : real) : ComplexTyp;
82
83 OPERATOR - (r : real; z : ComplexTyp) : ComplexTyp;
84
85 OPERATOR * (z: ComplexTyp; x: float): ComplexTyp;
86
87 OPERATOR * (x: float; z: ComplexTyp) : ComplexTyp;
88
89 OPERATOR / (z: ComplexTyp; x: float): ComplexTyp;
90
91 OPERATOR / (x: float; z: ComplexTyp) : ComplexTyp;
92
93 OPERATOR ** (Basis : ComplexTyp; Exponent : float) : ComplexTyp;
94
95 OPERATOR ** (Basis : float; Exponent : ComplexTyp) : ComplexTyp;
96
97 { ***** Powers, roots ***** }
98
99 FUNCTION ComplexLn(z: ComplexTyp): ComplexTyp;
100 { ln(z) = ln(abs(z)) + i * arg(z) }
101
102 FUNCTION ComplexExp(z: ComplexTyp): ComplexTyp;
103 { exp(x + iy) = exp(x)*cos(y) + i*exp(x)*sin(y) }
104
105 FUNCTION ComplexPower(Basis, Exponent: ComplexTyp): ComplexTyp;
106 { b^e = exp(e * ln(b)) }
107
108 FUNCTION ComplexPower(Basis: ComplexTyp; Exponent: float): ComplexTyp;
109 { z^x = exp(x * ln(z)) }
110
111 FUNCTION ComplexPower(Basis: float; Exponent: ComplexTyp): ComplexTyp;
112 { x^z = exp(z * ln(x)) }
113
114 FUNCTION ComplexRoot(z, b: ComplexTyp): ComplexTyp;
115 { b-te Wurzel aus z = z^(1/b) }
116
117 FUNCTION ComplexRoot(z: ComplexTyp; x: float): ComplexTyp;
118 { x-te Wurzel aus z = z^(1/x) }
119
120 FUNCTION ComplexRoot(x: float; z: ComplexTyp): ComplexTyp;
121 { z-te Wurzel aus x = x^(1/z) }
122

```

## 2. Complex numbers

```

123 { ***** Cyclometric functions ***** }
124
125 FUNCTION ComplexSin(z: ComplexTyp): ComplexTyp;
126 { sin(x +- iy) = sin(x) * cosh(y) +- i * cos(x) * sinh(y) }
127
128 FUNCTION ComplexCos(z: ComplexTyp): ComplexTyp;
129 { cos(x +- iy) = cos(x) * cosh(y) +- i * sin(x) * sinh(y) }
130
131 FUNCTION ComplexTan(z: ComplexTyp): ComplexTyp;
132 { tan(x +- iy) = (sin(2x) / (cos(2x) + cosh(2x)) +- i * (sinh(2x) /
133 { cot(z) = cos(z) / sin(z) }
136
137 FUNCTION ComplexArcSin(z: ComplexTyp): ComplexTyp;
138 { arcsin(z) = 1/i * ln(i*z + sqrt(1 - sqr(z))) }
139
140 FUNCTION ComplexArcCos(z: ComplexTyp): ComplexTyp;
141 { arccos(z) = 1/i * ln(z + sqrt(sqr(z) - 1)) }
142
143 FUNCTION ComplexArcTan(z: ComplexTyp): ComplexTyp;
144 { arctan(z) = 1/2i * ln((1+iz) / (1-iz)) }
145
146 FUNCTION ComplexArcCot(z: ComplexTyp): ComplexTyp;
147 { arccot(z) = -1/2i * ln((1+iz) / (iz - 1)) }
148
149 { ***** Hyperbolic funktions ***** }
150
151 FUNCTION ComplexSinh(z: ComplexTyp): ComplexTyp;
152 { sinh(x + iy) = sinh(x)*cos(y) +- i*cosh(x)*sin(y)
153 { = (exp(z) - exp(-z)) / 2 }
154
155 FUNCTION ComplexCosh(z: ComplexTyp): ComplexTyp;
156 { cosh(x + iy) = cosh(x)*cos(y) +- i*sinh(x)*sin(y)
157 { = (exp(z) + exp(-z)) / 2 }
158
159 FUNCTION ComplexTanh(z: ComplexTyp): ComplexTyp;
160 { tanh(x + iy) = (sinh(2x)/(cosh(2x)+cos(2y)) +
161 { i*(sin(2y)/(cosh(2x)+cos(2y)))
162 { = (exp(z) - exp(-z)) / (exp(z) + exp(-z)) }
163
164 FUNCTION ComplexCoth(z: ComplexTyp): ComplexTyp;
165 { coth(z) = cosh(z) / sinh(z) }
166 FUNCTION ComplexArSinh(z: ComplexTyp): ComplexTyp;

```

```

167 { arsinh(z) = ln(z + sqrt(sqr(z) + 1)) }
168
169 FUNCTION ComplexArCosh(z: ComplexTyp): ComplexTyp;
170 { arcosh(z) = ln(z + sqrt(sqr(z) - 1)) }
171
172 FUNCTION ComplexArTanh(z: ComplexTyp): ComplexTyp;
173 { artanh(z) := -1/2 * ln((1 + z) / (1 - z)) }
174
175 FUNCTION ComplexArCoth(z: ComplexTyp): ComplexTyp;
176 { arcoth(z) := 1/2 * ln((1 + z) / (z - 1)) }
177
178 { ***** Implementation *****
179 }
180 IMPLEMENTATION
181
182 VAR ch : CHAR; // used for error handling

```

## 2.1. Basic conversion and I/O routines

Listing 2.2: Conversion and I/O routines

```

1 FUNCTION ComplexInit(RePart, ImPart: float): ComplexTyp;
2
3 BEGIN
4   Result.RealPart := RePart;
5   Result.ImagPart := ImPart;
6 END;
7
8
9 FUNCTION Re(z: ComplexTyp): float;
10
11 BEGIN
12   Result := z.RealPart;
13 END;
14
15
16 FUNCTION Im(z: ComplexTyp): float;
17
18 BEGIN
19   Result := z.ImagPart;
20 END;
21
22
23 FUNCTION ComplexIsNull(z: ComplexTyp): BOOLEAN;

```

## 2. Complex numbers

```

24
25 BEGIN
26     Result := (Abs(Re(z)) < Zero) AND (Abs(Im(z)) < Zero);
27 END;
28
29
30 FUNCTION ComplexToStr(z: ComplexTyp; m, n: BYTE): STRING;
31 {Komplexe Zahl ausgeben}
32
33 VAR
34     OutStr, ImagStr: STRING;
35     k: INTEGER;
36
37 BEGIN
38     Str(Re(z): m: n, OutStr);
39     WHILE OutStr[1] = ' ' DO
40         Delete(OutStr, 1, 1);
41     IF Im(z) >= 0
42         THEN OutStr := OutStr + '+'
43     ELSE OutStr := OutStr + '-';
44     Str(Abs(Im(z)): m: n, ImagStr);
45     WHILE ImagStr[1] = ' ' DO
46         Delete(ImagStr, 1, 1);
47     OutStr := OutStr + ImagStr + 'i';
48     FOR k := 1 TO (m - Length(OutStr)) DO
49         OutStr := ' ' + OutStr;
50     Result := OutStr;
51 END;

```

The complex conjugate of a complex number  $z = x + iy$  is  $z^* = x - iy$ :

```

1 FUNCTION ComplexConj(z: ComplexTyp): ComplexTyp;
2
3 BEGIN
4     Result := ComplexInit(Re(z), -Im(z));
5 END;

```

The absolute value of a complex number is  $|z| = |x + iy| = \sqrt{x^2 + y^2} = \sqrt{z \times z^*}$ :

```

1 FUNCTION ComplexAbs(z: ComplexTyp): float;
2
3 VAR
4     x: float;
5
6 BEGIN
7     x := (Sqr(Re(z)) + Sqr(Im(z)));
8     IF Abs(x) < Zero
9         THEN Result := 0
10    ELSE Result := Sqrt(x);

```

```

11  END;

1  FUNCTION ComplexArg(z: ComplexTyp): float;
2
3  VAR
4      i, r: float;
5
6  BEGIN
7      IF ComplexIsNull(z)
8          THEN
9              BEGIN
10                 ch := WriteErrorMessage(' Complex numbers: Argument of (0 + 0i)');
11                 ComplexError := TRUE;
12                 EXIT;
13             END;
14             i := Im(z);
15             r := Re(z);
16             IF r = 0
17                 THEN Result := Signum(i) * 0.5 * Pi
18             ELSE
19                 IF r > 0
20                     THEN Result := ArcTan(i / r)
21                     ELSE Result := ArcTan(i / r) + Signum(i) * Pi;
22     END;

1  FUNCTION ComplexInv (z : complexTyp) : ComplexTyp;
2
3  BEGIN
4      IF ComplexIsNull(z)
5          THEN
6              BEGIN
7                  ch := WriteErrorMessage('Complex Numbers: Attempt to invert (0 +
8                      0i) ');
8                  ComplexError := TRUE;
9              END
10             ELSE Result := ComplexConj(z) / Sqr(ComplexAbs(z));
11     END;

1  FUNCTION ComplexNeg(z: ComplexTyp): ComplexTyp;
2
3  BEGIN
4      Result := ComplexInit(-Re(z), -Im(z));
5  END;

1  PROCEDURE Polar(z: ComplexTyp; VAR r, phi: float);
2
3  BEGIN

```

## 2. Complex numbers

```

4   r := ComplexAbs(z);
5   IF r <> 0
6     THEN IF Abs(Im(z) / r) = 1
7       THEN phi := Pi / 2 * (Im(z) / r) / Abs(Im(z) / r)
8       ELSE phi := ArcTan(Im(z) / Re(z))
9     ELSE
10      BEGIN
11        phi := 0;
12        IF Re(z) < 0
13          THEN
14            IF Im(z) <> 0
15              THEN phi := phi + Pi * Abs(Im(z)) / Im(z)
16              ELSE phi := Pi;
17          END;
18      END;
19
20  FUNCTION Rect(r, phi: float): ComplexTyp;
21
22  BEGIN
23    Result := ComplexInit(r * Cos(phi), r * Sin(phi));
24  END;

```

## 2.2. Basic operators

### 2.2.1. Operators on complex variables

In the following routines, we use operator overloading to assign meaning to the operators  $=, :=, +, -, *, /, **$  for complex arguments [3].

Two complex numbers are equal if both their real and imaginary parts are equal:

```

1  OPERATOR = (z, b: ComplexTyp) : BOOLEAN;
2
3  BEGIN
4    Result := (Re(z) = Re(b)) AND (Im(z) = Im(b));
5  END;

```

Two numbers  $a, b \in \mathbb{C}$  are added  $a + b = x_a + y_a i + x_b + y_b i = (x_a + x_b) + (y_a + y_b) i$ :

```

1  OPERATOR + (z, b: ComplexTyp): ComplexTyp;
2
3  BEGIN
4    Result := ComplexInit(Re(z) + Re(b), Im(z) + Im(b));
5  END;

```

Similarly, for subtraction  $a - b = x_a + y_a i - x_b + y_b i = (x_a - x_b) + (y_a - y_b) i$ :

```

1  OPERATOR - (z, b: ComplexTyp): ComplexTyp;
2

```

```

3 BEGIN
4   Result := ComplexInit(Re(z) - Re(b), Im(z) - Im(b));
5 END;

```

There is also the unary minus:

```

1 OPERATOR - (z: ComplexTyp): ComplexTyp;
2
3 BEGIN
4   Result := ComplexInit(-Re(z), -Im(z));
5 END;

```

Multiplication of complex numbers  $a \times b = (x_a + y_a i) \times (x_b + y_b i) = (x_a x_b - y_a y_b) + (x_a y_b + x_b y_a) i$ .

```

1 OPERATOR * (z, b: ComplexTyp): ComplexTyp;
2
3 VAR
4   r, i: float;
5
6 BEGIN
7   r := Re(z) * Re(b) - Im(z) * Im(b);
8   i := Re(z) * Im(b) + Im(z) * Re(b);
9   Result := ComplexInit(r, i);
10 END;

```

Once the multiplication is defined, we can also define the division of a complex number by a complex number  $a/b = ab^*/|b|^2$ .

```

1 OPERATOR / (z, b: ComplexTyp): ComplexTyp;
2
3 VAR r, i : float;
4
5 BEGIN
6   IF ComplexIsNull(b)
7     THEN
8       BEGIN
9         ch := WriteErrorMessage('Complex numbers: Division by (0 + 0i)');
10        ComplexError := TRUE;
11        EXIT;
12      END;
13   r := (Re(z) * Re(b) + Im(z) * Im(b)) / (Sqr(Re(b)) + Sqr(Im(b)));
14   i := (Im(z) * Re(b) - Re(z) * Im(b)) / (Sqr(Re(b)) + Sqr(Im(b)));
15   Result := ComplexInit(r, i);
16 END;

```

And finally, we can define a power operator:

```

1 OPERATOR ** (Basis, Exponent: ComplexTyp): ComplexTyp;
2
3 BEGIN

```

## 2. Complex numbers

```

4   IF ComplexIsNull(Basis)
5     THEN Result := ComplexInit(0, 0)
6     ELSE Result := ComplexExp(Exponent * ComplexLn(Basis));
7 END;

```

### 2.2.2. Operators on mixed complex and real variables

Operations which are commutative need to be defined for both the real, complex and the complex, real case.

Comparison:

```

1  OPERATOR = (z: ComplexTyp; x: float) : BOOLEAN;
2
3  BEGIN
4    Result := (Abs(Im(z)) < Zero) AND (Re(z) = x);
5  END;
6
7  OPERATOR = (x: float; z: ComplexTyp) : BOOLEAN;
8
9  BEGIN
10   Result := (Abs(Im(z)) < Zero) AND (Re(z) = x);
11 END;
12
13 OPERATOR := (x: real) : ComplexTyp;
14
15 BEGIN
16   Result := ComplexInit(x, 0);
17 END;
18
19 OPERATOR := (z: ComplexTyp) : real;
20
21 BEGIN
22   IF Abs(Im(z)) > Zero
23     THEN
24       BEGIN
25         ch := WriteErrorMessage('Assignment of complex to real variable:
26           Im(z) <> 0');
27         ComplexError := TRUE;
28       END
29     ELSE
30       Result := Re(z);
31   END;
32
33 OPERATOR + (z: ComplexTyp; x: float) : ComplexTyp;
34
35 BEGIN

```

```

4   Result := ComplexInit(x + Re(z), Im(z));
5 END;
6
7 OPERATOR + (x: float; z: ComplexTyp) : ComplexTyp;
8
9 BEGIN
10   Result := ComplexInit(x + Re(z), Im(z));
11 END;

```

```

1 OPERATOR - (z : ComplexTyp; r : real) : ComplexTyp;
2
3 BEGIN
4   Result := ComplexInit(Re(z) - r, Im(z));
5 END;
6
7
8 OPERATOR - (r : real; z : ComplexTyp) : ComplexTyp;
9
10 BEGIN
11   Result := ComplexInit(r - Re(z), -Im(z));
12 END;

```

We can also define the multiplication of a complex number  $a$  and a real number  $b$  as  
 $a \times b = x_a + y_a i \times b = x_a b + y_a b i$ .

```

1 OPERATOR * (z: ComplexTyp; x: float) : ComplexTyp;
2
3 BEGIN
4   Result := ComplexInit(Re(z) * x, Im(z) * x);
5 END;
6
7
8 OPERATOR * (x: float; z: ComplexTyp) : ComplexTyp;
9
10 BEGIN
11   Result := ComplexInit(Re(z) * x, Im(z) * x);
12 END;

```

```

1 OPERATOR / (z: ComplexTyp; x: float): ComplexTyp;
2
3 BEGIN
4   IF x = 0
5     THEN
6       BEGIN
7         ch := WriteErrorMessage('Complex numbers: Attempt to divide by
8           zero');
8         ComplexError := TRUE;
9       END

```

## 2. Complex numbers

```

10      ELSE
11          Result := ComplexInit(Re(z) / x, Im(z) / x);
12      END;
13
14
15  OPERATOR / (x: float; z: ComplexTyp) : ComplexTyp;
16
17  VAR d : float;
18
19  BEGIN
20      IF ComplexIsNull(z)
21      THEN
22          BEGIN
23              ch := WriteErrorMessage('Complex numbers: Division by (0 + 0i)');
24              ComplexError := TRUE;
25          END
26      ELSE
27          BEGIN
28              d := Sqr(Re(z)) + Sqr(Im(z));
29              Result := ComplexInit(x*Re(z)/d, -x*Im(z)/d);
30          END;
31      END;

```

```

1  OPERATOR ** (Basis : ComplexTyp; Exponent : float) : ComplexTyp;
2
3  BEGIN
4      IF ComplexIsNull(Basis)
5      THEN Result := ComplexInit(0, 0)
6      ELSE Result := ComplexExp(ComplexLn(Basis) * Exponent);
7  END;
8
9
10 OPERATOR ** (Basis : float; Exponent : ComplexTyp) : ComplexTyp;
11
12 BEGIN
13     IF Abs(Basis) < Zero
14     THEN
15         BEGIN
16             ch := WriteErrorMessage('Complex numbers: Complex power of 0');
17             ComplexError := TRUE;
18         END
19     ELSE
20         Result := ComplexExp(Exponent * Ln(Basis));
21     END;

```

## 2.3. Logarithms and powers

$\exp(x + iy) = \exp(x) * \cos(y) + i \exp(x) * \sin(y)$ . The logarithm of a complex number is  $\ln(z) = \ln(\text{abs}(z)) + i * \arg(z)$

Listing 2.3: Logarithm and exponential function

```

1  FUNCTION ComplexExp(z: ComplexTyp): ComplexTyp;
2
3  VAR
4      i, r: float;
5
6  BEGIN
7      i := Im(z);
8      r := Exp(Re(z));
9      Result := ComplexInit(r * Cos(i), r * Sin(i));
10 END;
11
12
13 FUNCTION ComplexLn(z: ComplexTyp): ComplexTyp;
14
15 VAR r, i: float;
16
17 BEGIN
18     IF ComplexIsNull(z)
19     THEN
20         BEGIN
21             ch := WriteErrorMessage('Complex numbers: Attempt to calculate
22             ln(0+0i)');
23             ComplexError := TRUE;
24         END
25     ELSE
26         BEGIN
27             r := Ln(ComplexAbs(z));
28             i := ComplexArg(z);
29             Result := ComplexInit(r, i);
30         END;
31 END;
```

A base is raised to the power of an exponent by  $b^e = \exp(e \times \ln(b))$ . This requires different function depending on whether  $b, z \in \mathbb{R}$  or  $\mathbb{C}$ :

Listing 2.4: complex power function

```

1  FUNCTION ComplexPower(Basis, Exponent: ComplexTyp): ComplexTyp;
2
3  BEGIN
4      IF ComplexIsNull(Basis)
5          THEN Result := ComplexInit(0, 0)
```

## 2. Complex numbers

```

6      ELSE Result := ComplexExp(Exponent * ComplexLn(Basis));
7 END;
8
9
10 FUNCTION ComplexPower(Basis: ComplexTyp; Exponent: float): ComplexTyp;
11
12 BEGIN
13   IF ComplexIsNull(Basis)
14     THEN Result := ComplexInit(0, 0)
15   ELSE Result := ComplexExp(ComplexLn(Basis) * Exponent);
16 END;
17
18
19 FUNCTION ComplexPower(Basis: float; Exponent: ComplexTyp): ComplexTyp;
20
21 BEGIN
22   IF Abs(Basis) < Zero
23     THEN
24       BEGIN
25         ch := WriteErrorMessage('Complex numbers: Complex power of 0');
26         ComplexError := TRUE;
27       END
28     ELSE
29       Result := ComplexExp(Exponent * Ln(Basis));
30 END;

```

The root of a number is calculated by  $\sqrt[b]{z} = z^{1/b}$ . This requires different function depending on whether  $b, z \in \mathbb{R}$  or  $\mathbb{C}$ :

```

1 FUNCTION ComplexRoot(z, b: ComplexTyp): ComplexTyp;
2
3 BEGIN
4   IF ComplexIsNull(b)
5     THEN
6       BEGIN
7         ch := WriteErrorMessage('Complex numbers: Attempt to calculate the
8           (0 + 0i)-th root');
8         ComplexError := TRUE;
9         EXIT;
10      END
11    ELSE
12      Result := ComplexPower(z, ComplexInv(b));
13  END;
14
15
16 FUNCTION ComplexRoot(z: ComplexTyp; x: float): ComplexTyp;
17
18 BEGIN

```

```

19  IF x = 0
20  THEN
21    BEGIN
22      ch := WriteErrorMessage('Complex numbers: 0-th root');
23      ComplexError := TRUE;
24    END
25  ELSE
26    IF ComplexIsNull(z)
27    THEN Result := ComplexInit(0, 0)
28    ELSE Result := ComplexPower(z, 1 / x);
29  END;
30
31
32 FUNCTION ComplexRoot(x: float; z: ComplexTyp): ComplexTyp;
33
34 BEGIN
35  IF ComplexIsNull(z)
36  THEN
37    BEGIN
38      ch := WriteErrorMessage('Complex numbers: (0 + 0i)-th root');
39      ComplexError := TRUE;
40    END
41  ELSE IF x = 0
42    THEN Result := ComplexInit(0, 0)
43    ELSE Result := ComplexPower(x, ComplexInv(z));
44 END;

```

## 2.4. Angle and cyclometric functions

$$\sin(x \pm iy) = \sin(x) \times \cosh(y) \pm i \cos(x) \sinh(y):$$

```

1 FUNCTION ComplexSin(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   r, i: float;
5
6 BEGIN
7   r := Sin(Re(z)) * cosh(Im(z));
8   i := Cos(Re(z)) * sinh(Im(z));
9   Result := ComplexInit(r, i);
10 END;

```

$$\cos(x \pm iy) = \cos(x) * \cosh(y) \pm i * \sin(x) * \sinh(y):$$

```

1 FUNCTION ComplexCos(z: ComplexTyp): ComplexTyp;
2
3 VAR

```

## 2. Complex numbers

```

4   r, i: float;
5
6 BEGIN
7   r := Cos(Re(z)) * cosh(Im(z));
8   i := -Sin(Re(z)) * sinh(Im(z));
9   Result := ComplexInit(r, i);
10 END;

```

$$\tan(x \pm iy) = \frac{\sin(2x)}{\cos(2x)+\cosh(2y)} \pm i \frac{\sinh(2y)}{\cos(2x)+\cosh(2y)}:$$

```

1 FUNCTION ComplexTan(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   r, i: float;
5
6 BEGIN
7   r := Sin(2 * Re(z)) / (Cos(2 * Re(z)) + cosh(2 * Im(z)));
8   i := sinh(2 * Im(z)) / (Cos(2 * Re(z)) + cosh(2 * Im(z)));
9   Result := ComplexInit(r, i);
10 END;

```

$$\cot(z) = \frac{\cos(z)}{\sin(z)}:$$

```

1 FUNCTION ComplexCot(z: ComplexTyp): ComplexTyp;
2
3 BEGIN
4   IF ComplexIsNull(z)
5     THEN
6       BEGIN
7         ch := WriteErrorMessage('Complex numbers: Attempt to calculate
8           cot(0 + 0i)');
8         ComplexError := TRUE;
9       END
10  ELSE
11    Result := ComplexCos(z) / ComplexSin(z);
12 END;

```

$$\arcsin(z) = 1/i \times \ln(iz + \sqrt{1 - z^2}):$$

```

1 FUNCTION ComplexArcSin(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   c0, c1, c2, c3: ComplexTyp;
5
6 BEGIN
7   IF ComplexIsNull(z)
8     THEN
9       BEGIN

```

```

10      ch := WriteErrorMessage('Complex Numbers: Attempt to calculate
11          arcsin(0 + 0i)');
12      ComplexError := TRUE;
13      EXIT;
14  END;
15  c1 := ComplexInit(1, 0);
16  c2 := ComplexInit(0, 1);
17  c3 := ComplexInit(0, -1);
18  c0 := ComplexRoot(c1 - ComplexPower(z, 2), 2);
19  c0 := ComplexLn(c2 * z + c0);
20  Result := c3 * c0;
21 END;

```

$$\arccos(z) = \frac{1}{i} \times \ln(z + \sqrt{z^2 - 1})$$

```

1 FUNCTION ComplexArcCos(z: ComplexTyp): ComplexTyp;
2
3 VAR
4     c0, c1, c2: ComplexTyp;
5
6 BEGIN
7     IF ComplexIsNull(z)
8     THEN
9         BEGIN
10            ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11                arccos(0 + 0i)');
12            ComplexError := TRUE;
13            EXIT;
14        END;
15        c1 := ComplexInit(1.0, 0.0);
16        c2 := ComplexInit(0.0, -1.0);
17        c0 := ComplexRoot(ComplexPower(z, 2) - c1, 2);
18        Result := ComplexLn(z + c0) * c2;
19    END;

```

$$\arctan(z) = \frac{1}{2}i \times \ln\left(\frac{1+iz}{1-iz}\right)$$

```

1 FUNCTION ComplexArcTan(z: ComplexTyp): ComplexTyp;
2
3 VAR
4     c0, c1, c2, c3, a1, a2: ComplexTyp;
5
6 BEGIN
7     IF ComplexIsNull(z)
8     THEN
9         BEGIN
10            ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11                arctan(0 + 0i)');
12            ComplexError := TRUE;
13        END;

```

## 2. Complex numbers

```

12      EXIT;
13  END;
14  c1 := ComplexInit(0, 1);
15  c2 := ComplexInit(1, 0);
16  c3 := ComplexInit(0, 2);
17  c0 := c1 * z;
18  a1 := c2 + c0;
19  a2 := c2 - c0;
20  c0 := ComplexLn(a1 / a2);
21  a1 := c2 / c3;
22  Result := a1 * c0;
23 END;

```

$$\operatorname{arccot}(z) = -\frac{1}{2}i \times \ln\left(\frac{1+iz}{iz-1}\right);$$

```

1 FUNCTION ComplexArcCot(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   c0, c1, c2, c3, a1, a2: ComplexTyp;
5
6 BEGIN
7   IF ComplexIsNull(z)
8     THEN
9     BEGIN
10       ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11           arccot(0 + 0i)');
12       ComplexError := TRUE;
13       EXIT;
14     END;
15   c1 := ComplexInit(0, 1);
16   c2 := ComplexInit(1, 0);
17   c3 := ComplexInit(0, -2);
18   c0 := c1 * z;
19   a1 := c0 + c2;
20   a2 := c0 - c2;
21   c0 := ComplexLn(a1 / a2);
22   a1 := c2 / c3;
23   Result := a1 * c0;
24 END;

```

## 2.5. Hyperbolic functions

$$\sinh(x + iy) = \sinh(x)\cos(y) + i\cosh(x)\sin(y) = \frac{\exp(z) - \exp(-z)}{2};$$

```

1 FUNCTION ComplexSinh(z: ComplexTyp): ComplexTyp;
2

```

```

3  VAR
4      r, i: float;
5
6  BEGIN
7      r := sinh(Re(z)) * Cos(Im(z));
8      i := cosh(Re(z)) * Sin(Im(z));
9      Result := ComplexInit(r, i);
10 END;

```

$$\cosh(x+iy) = \cosh(x)\cos(y) \pm i\sinh(x)\sin(y) = \frac{\exp(z)+\exp(-z)}{2}:$$

```

1  FUNCTION ComplexCosh(z: ComplexTyp): ComplexTyp;
2
3  VAR
4      r, i: float;
5
6  BEGIN
7      r := cosh(Re(z)) * Cos(Im(z));
8      i := sinh(Re(z)) * Sin(Im(z));
9      Result := ComplexInit(r, i);
10 END;

```

$$\tanh(x+iy) = (\sinh(2x)/(\cosh(2x)+\cos(2y))+i(\sin(2y)/(\cosh(2x)+\cos(2y)))) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}:$$

```

1  FUNCTION ComplexTanh(z: ComplexTyp): ComplexTyp;
2
3  VAR
4      r, i: float;
5
6  BEGIN
7      r := sinh(2 * Re(z)) / (cosh(2 * Re(z)) + Cos(2 * Im(z)));
8      i := Sin(2 * Im(z)) / (cosh(2 * Re(z)) + Cos(2 * Im(z)));
9      Result := ComplexInit(r, i);
10 END;

```

$$\coth(z) = \frac{\cosh(z)}{\sinh(z)}:$$

```

1  FUNCTION ComplexCoth (z : ComplexTyp) : ComplexTyp;
2
3  BEGIN
4      IF ComplexIsNull(z)
5          THEN
6              BEGIN
7                  ch := WriteErrorMessage('Complex numbers: Attempt to calculate
8                      coth(0 + 0i)');
9                  ComplexError := TRUE;
10             END
11      ELSE
12          Result := ComplexCosh(z) / ComplexSinh(z);

```

## 2. Complex numbers

```
12 END;
```

### 2.6. Area functions

$$\text{arsinh}(z) = \ln(z + \sqrt{z^2 + 1}):$$

```

1 FUNCTION ComplexArSinh(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   c0, c1: ComplexTyp;
5
6 BEGIN
7   IF ComplexIsNull(z)
8     THEN
9     BEGIN
10      ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11        arsinh(0 + 0i)');
12      ComplexError := TRUE;
13    END
14  ELSE
15    BEGIN
16      c1 := ComplexInit(1, 0);
17      c0 := ComplexRoot(ComplexPower(z, 2) + c1, 2);
18      Result := ComplexLn(z + c0);
19    END;
20 END;
```

$$\text{arcosh}(z) = \ln(z + \sqrt{z^2 - 1}):$$

```

1 FUNCTION ComplexArCosh(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   c0, c1: ComplexTyp;
5
6 BEGIN
7   IF ComplexIsNull(z)
8     THEN
9     BEGIN
10      ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11        arcosh(0 + 0i)');
12      ComplexError := TRUE;
13    END
14  ELSE
15    BEGIN
16      c1 := ComplexInit(1, 0);
17      c0 := ComplexRoot(ComplexPower(z, 2) - c1, 2);
18      Result := ComplexLn(z + c0);
19    END;
20 END;
```

```

18      END;
19  END;

artanh(z) := - $\frac{1}{2} * \ln(\frac{1+z}{1-z})$ :

FUNCTION ComplexArTanh(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   c0, c1, c2, a1, a2: ComplexTyp;
5
6 BEGIN
7   IF ComplexIsNull(z)
8     THEN
9     BEGIN
10       ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11         artanh(0 + 0i)');
12       ComplexError := TRUE;
13     END
14   ELSE
15     BEGIN
16       c1 := ComplexInit(1, 0);
17       c2 := ComplexInit(2, 0);
18       a1 := c1 + z;
19       a2 := c1 - z;
20       c0 := ComplexLn(a1 / a2);
21       Result := c0 / c2;
22     END;
23 END;

```

```

arcoth(z) :=  $\frac{1}{2} * \ln(\frac{1+z}{z-1})$ :

FUNCTION ComplexArCoth(z: ComplexTyp): ComplexTyp;
2
3 VAR
4   c0, c1, c2, a1, a2: ComplexTyp;
5
6 BEGIN
7   IF ComplexIsNull(z)
8     THEN
9     BEGIN
10       ch := WriteErrorMessage('Complex numbers: Attempt to calculate
11         arcoth(0 + 0i)');
12       ComplexError := TRUE;
13     END
14   ELSE
15     BEGIN
16       c1 := ComplexInit(1, 0);
17       c2 := ComplexInit(2, 0);
18       a1 := z + c1;
19     END;
20 END;

```

## 2. Complex numbers

```
18      a2 := z - c1;
19      c0 := ComplexLn(a1 / a2);
20      Result := c0 / c2;
21  END;
22 END;
23
24 END. // UNIT Complex
```

## 2.7. Test program

A simple test program can be used to identify problems. It is based on applying the reciprocal function to the result of a function, this should return the original argument:

```
1 PROGRAM TestComplex;
2
3 USES Mathfunc, Complex;
4
5 VAR a, b, c, d : ComplexTyp;
6     x, y, z     : double;
7
8
9 PROCEDURE EinOperand(a, r : ComplexTyp; Operation : STRING);
10
11 BEGIN
12     Write(Operation, '(');
13     Write(ComplexToStr(a, 6, 3));
14     Write(') = ');
15     Write(ComplexToStr(r, 6, 3));
16     Writeln;
17 END;
18
19
20 PROCEDURE ZweiOperanden (a1, a2, r : ComplexTyp; Operation : STRING);
21
22 BEGIN
23     Write('(');
24     Write(ComplexToStr(a1, 6, 3));
25     Write(')', Operation, ' (');
26     Write(ComplexToStr(a2, 6, 3));
27     Write(') = ');
28     Write(ComplexToStr(r, 6, 3));
29     Writeln(')');
30 END;
31
32
```

```

33 PROCEDURE ComplexMitReel (a1, r : ComplexTyp; a2 : double; Operation :
34   STRING);
35
36 BEGIN
37   Write('(');
38   Write(ComplexToStr(a1, 6, 3));
39   Write(' ' , Operation, ' ', a2:6:3);
40   Write(' = ');
41   Write(ComplexToStr(r, 6, 3));
42   Writeln(')');
43
44
45 PROCEDURE TestAddSub;
46
47 BEGIN
48   c := a + b;
49   ZweiOperanden(a, b, c, '+');
50   d := c - b;
51   ZweiOperanden(c, b, d, '-');
52 END;
53
54
55 PROCEDURE TestMulDiv;
56
57 BEGIN
58   c := a * b;
59   ZweiOperanden(a, b, c, '*');
60   d := c / b;
61   IF ComplexError
62     THEN ComplexError := FALSE
63   ELSE ZweiOperanden(c, b, d, '/');
64 END;
65
66
67 PROCEDURE TestMulDivMitReel;
68
69 BEGIN
70   b := a * x;
71   ComplexMitReel(a, b, x, '*');
72   c := b / x;
73   IF ComplexError
74     THEN ComplexError := FALSE
75   ELSE ComplexMitReel(b, c, x, '/');
76 END;
77

```

## 2. Complex numbers

```
78
79 PROCEDURE TestPolarRect;
80
81 BEGIN
82   Polar(a, y, z);
83   c := Rect(y, z);
84   Write('Polar(');
85   Write(ComplexToStr(a, 6, 3));
86   Writeln(') = (' , y:6:3, ', ', z:6:3, ')');
87   Write('Rect(' , y:6:3, ', ', z:6:3, ') = (' );
88   Write(ComplexToStr(c, 6, 3));
89   Writeln(')');
90 END;
91
92
93 PROCEDURE TestExpLn;
94
95 BEGIN
96   b := ComplexExp(a);
97   IF ComplexError
98     THEN
99       BEGIN
100      ComplexError := FALSE;
101      EXIT;
102    END;
103   EinOperand(a, b, 'exp');
104   c := ComplexLn(b);
105   IF ComplexError
106     THEN ComplexError := FALSE
107   ELSE EinOperand(b, c, 'ln');
108 END;
109
110
111 PROCEDURE TestPotReelComplex;
112
113 BEGIN
114   b := ComplexPower(x, a);
115   IF ComplexError
116     THEN
117       BEGIN
118         ComplexError := FALSE;
119         EXIT;
120       END;
121   Write(x:6:3, '^(');
122   Write(ComplexToStr(a, 6, 3));
123   Write(') = ');
```

```

124  Write(ComplexToStr(b, 6, 3));
125  Writeln;
126  c := ComplexRoot(b, a);
127  IF ComplexError
128  THEN
129    BEGIN
130      ComplexError := FALSE;
131      EXIT;
132    END;
133  Write('(');
134  Write(ComplexToStr(b, 6, 3));
135  Write(')^1/(');
136  Write(ComplexToStr(a, 6, 3));
137  Write(') = ');
138  Write(ComplexToStr(c, 6, 3));
139  Writeln;
140 END;
141
142
143 PROCEDURE TestPotComplexReel;
144
145 BEGIN
146   b := ComplexPower(a, x);
147   IF ComplexError
148   THEN
149     BEGIN
150       ComplexError := FALSE;
151       EXIT;
152     END;
153   ComplexMitReel(a, b, x, '^');
154   c := ComplexRoot(b, x);
155   IF ComplexError
156   THEN
157     BEGIN
158       ComplexError := FALSE;
159       EXIT;
160     END;
161   ComplexMitReel(b, c, x, '^1/');
162 END;
163
164
165 PROCEDURE TestPotComplexComplex;
166
167 BEGIN
168   c := ComplexPower(a, b);
169   IF ComplexError

```

## 2. Complex numbers

```
170      THEN
171      BEGIN
172          ComplexError := FALSE;
173          EXIT;
174      END;
175      ZweiOperanden(a, b, c, '^');
176      d := ComplexRoot(c, b);
177      IF ComplexError
178      THEN
179          BEGIN
180              ComplexError := FALSE;
181              EXIT;
182          END;
183          ZweiOperanden(c, b, d, '^1/');
184      END;
185
186
187 PROCEDURE TestSin;
188
189 BEGIN
190     b := ComplexSin(a);
191     IF ComplexError
192     THEN
193         BEGIN
194             ComplexError := FALSE;
195             EXIT;
196         END;
197         EinOperand(a, b, 'sin');
198         c := ComplexArcSin(b);
199         IF ComplexError
200         THEN
201             BEGIN
202                 ComplexError := FALSE;
203                 EXIT;
204             END;
205             EinOperand(b, c, 'arcsin');
206         END;
207
208
209 PROCEDURE TestCos;
210
211 BEGIN
212     b := ComplexCos(a);
213     IF ComplexError
214     THEN
215         BEGIN
```

```

216      ComplexError := FALSE;
217      EXIT;
218  END;
219  EinOperand(a, b, 'cos');
220  c := ComplexArcCos(b);
221  IF ComplexError
222  THEN
223    BEGIN
224      ComplexError := FALSE;
225      EXIT;
226    END;
227    EinOperand(b, c, 'arccos');
228  END;
229
230
231 PROCEDURE TestTan;
232
233 BEGIN
234   b := ComplexTan(a);
235   IF ComplexError
236   THEN
237     BEGIN
238       ComplexError := FALSE;
239       EXIT;
240     END;
241   EinOperand(a, b, 'tan');
242   c := ComplexArcTan(b);
243   IF ComplexError
244   THEN
245     BEGIN
246       ComplexError := FALSE;
247       EXIT;
248     END;
249   EinOperand(b, c, 'arctan');
250 END;
251
252
253 PROCEDURE TestCot;
254
255 BEGIN
256   b := ComplexCot(a);
257   IF ComplexError
258   THEN
259     BEGIN
260       ComplexError := FALSE;
261       EXIT;

```

## 2. Complex numbers

```
262     END;
263     EinOperand(a, b, 'cot');
264     c := ComplexArcCot(b);
265     IF ComplexError
266     THEN
267         BEGIN
268             ComplexError := FALSE;
269             EXIT;
270         END;
271         EinOperand(b, c, 'arctan');
272     END;
273
274
275 PROCEDURE TestSinh;
276
277 BEGIN
278     b := ComplexSinh(a);
279     IF ComplexError
280     THEN
281         BEGIN
282             ComplexError := FALSE;
283             EXIT;
284         END;
285         EinOperand(a, b, 'sinh');
286         c := ComplexArSinh(b);
287         IF ComplexError
288         THEN
289             BEGIN
290                 ComplexError := FALSE;
291                 EXIT;
292             END;
293             EinOperand(b, c, 'arsinh');
294         END;
295
296
297 PROCEDURE TestCosh;
298
299 BEGIN
300     b := ComplexCosh(a);
301     IF ComplexError
302     THEN
303         BEGIN
304             ComplexError := FALSE;
305             EXIT;
306         END;
307         EinOperand(a, b, 'cosh');
```

```

308   c := ComplexArCosh(b);
309   IF ComplexError
310     THEN
311       BEGIN
312         ComplexError := FALSE;
313         EXIT;
314       END;
315       EinOperand(b, c, 'arcosh');
316   END;
317
318
319 PROCEDURE TestTanh;
320
321 BEGIN
322   b := ComplexTanh(a);
323   IF ComplexError
324     THEN
325       BEGIN
326         ComplexError := FALSE;
327         EXIT;
328       END;
329   EinOperand(a, b, 'tanh');
330   c := ComplexArTanh(b);
331   IF ComplexError
332     THEN
333       BEGIN
334         ComplexError := FALSE;
335         EXIT;
336       END;
337   EinOperand(b, c, 'artanh');
338 END;
339
340
341 PROCEDURE TestCoth;
342
343 BEGIN
344   b := ComplexCoth(a);
345   IF ComplexError
346     THEN
347       BEGIN
348         ComplexError := FALSE;
349         EXIT;
350       END;
351   EinOperand(a, b, 'coth');
352   c := ComplexArCoth(b);
353   IF ComplexError

```

## 2. Complex numbers

```
354     THEN
355     BEGIN
356         ComplexError := FALSE;
357         EXIT;
358     END;
359     EinOperand(b, c, 'arcoth');
360 END;
361
362
363 PROCEDURE TestAll;
364
365 VAR c : ComplexTyp;
366
367 BEGIN
368     TestAddSub; Writeln;
369     TestMulDiv; Writeln;
370     TestMulDivMitReel; Writeln;
371     TestPolarRect; Writeln;
372     TestExpln; Writeln;
373     TestPotReelComplex; Writeln;
374     TestPotComplexReel; Writeln;
375     TestPotComplexComplex; Writeln;
376     TestSin; Writeln;
377     TestCos; Writeln;
378     TestTan; Writeln;
379     TestCot; Writeln;
380     TestSinh; Writeln;
381     TestCosh; Writeln;
382     TestTanh; Writeln;
383     TestCoth; Writeln;
384     ReadLn;
385 END;
386
387
388 BEGIN
389     a := ComplexInit(0.621, 0.567);
390     b := ComplexInit(0.5, 0.4);
391     x := 1.5;
392     TestAll;
393     a := ComplexInit(1.0, -1.0);
394     x := 0.2;
395     TestAll;
396     a := ComplexInit(-0.5, 0.2);
397     x := 0.2;
398     TestAll;
399     a := ComplexInit(-1.0, -1.5);
```

```
400  x := 1.5;
401  TestAll;
402  a := ComplexInit(1.0, 0.0);
403  x := 1.5;
404  TestAll;
405  a := ComplexInit(0.0, 1.0);
406  x := 1.5;
407  TestAll;
408  a := ComplexInit(0.0, 0.0);
409  x := 1.5;
410  TestAll;
411 END.
```

## References

- [1] K. GIESELMANN: Komplexe Zahlen in PASCAL, *PASCAL Int.* **2**:3 (1987), 68–73.
- [2] K.D. THIES: *PC-XT-AT Numerik Buch* München: TeWi, 1989.
- [3] M. van CANNEYT: *Operator overloading*, In: *Free Pascal Reference guide* 2020 chap. 15 URL: <https://www.freepascal.org/docs-html/ref/refch15.html#x212-23400015>.



# 3. Big sets

## Abstract

The set data type in pascal is limited to `MaxByte` (255) members. In some applications, this is insufficient. This unit defines a set type with arbitrary size and the operations that can be performed on it.

A set is, in effect, an array `[0..pred(MaxCardinality)]` of bit. Each bit stands for one member, and is 1 if that member is in the set and 0 if it is not. However, there is no data type bit in Pascal, so we implement the array as an array of word. `MaxCardinality` should be a multiple of the `WordSize`, which is `2 byte = 16 bit` under Lazarus for Windows. Thus, for our purposes, it is reasonable to set `MaxCardinality = 2048`. Then we only need to define routines that allow access to individual bits. The interface of such a unit would be:

Listing 3.1: Interface

```
1 UNIT BigSet;
2
3 INTERFACE
4
5 USES Classes;
6
7 CONST MaxCardinality = 2048; // change AS needed
8     WordSize      = SizeOf(WORD) * 8; // IN bits
9     MaxMembers    = Pred(MaxCardinality DIV (WordSize)); // number OF words needed TO fit elements
10
11 TYPE SetType = ARRAY [0..MaxMembers] OF WORD;
12
13 PROCEDURE SetBit (VAR S : SetType; Bit : WORD);
14
15 PROCEDURE ClearBit (VAR S : SetType; Bit : WORD);
16
17 PROCEDURE ToggleBit (VAR S : SetType; Bit : WORD);
18
19 PROCEDURE ClearAllBits (VAR S : SetType);
20
21 PROCEDURE SetAllBits (VAR S : SetType);
```

### 3. Big sets

```

22
23 FUNCTION InSet (CONST S : SetType; Bit : WORD) : BOOLEAN;
24
25 PROCEDURE SetUnion (VAR S: SetType; CONST T, U : SetType);
26
27 PROCEDURE SetIntersection (VAR S: SetType; CONST T, U : SetType);
28
29 PROCEDURE SetComplement (VAR S: SetType; CONST T, U : SetType);
30
31 PROCEDURE SetSymDifference (VAR S: SetType; CONST T, U : SetType);
32
33 FUNCTION SubSet (CONST S, T : SetType) : BOOLEAN;
34
35 FUNCTION TrueSubSet (CONST S, T : SetType) : BOOLEAN;
36
37 FUNCTION EqualSet (CONST S, T : SetType) : BOOLEAN;
38
39 FUNCTION EmptySet (CONST S : SetType) : BOOLEAN;
40
41FUNCTION Cardinality (CONST S : SetType) : WORD;

```

Listing 3.2: Implementation

```

1 IMPLEMENTATION
2
3 CONST Bits : ARRAY [0..15] OF WORD = (    1,      2,      4,      8,
4                           16,      32,
5                           64,     128,
6                           256,    512,   1024,  2048,  4096,  8192,
7                           16384, 32768);
8
9
10 PROCEDURE SetBit(VAR S : SetType; Bit : WORD);
11
12 VAR n : WORD;
13
14 BEGIN
15   n := Bit DIV WordSize;           // identify the WORD that stores the bit
16   S[n] := S[n] OR Bits[Bit MOD 16];
17 END;
18
19
20 PROCEDURE ClearBit (VAR S : SetType; Bit : WORD);
21
22 VAR n : WORD;
23
24 BEGIN
25   n := Bit DIV WordSize;
26   S[n] := S[n] AND NOT Bits[Bit MOD 16];

```

```

23  END;
24
25
26 PROCEDURE ToggleBit (VAR S : SetType; Bit : WORD);
27
28 VAR n : WORD;
29
30 BEGIN
31   n := Bit DIV WordSize;
32   S[n] := S[n] XOR Bits[Bit MOD 16];
33 END;
34
35
36 FUNCTION InSet (CONST S : SetType; Bit : WORD) : BOOLEAN;
37
38 VAR n : WORD;
39
40 BEGIN
41   n := Bit DIV WordSize;
42   Result := S[n] AND Bits[Bit MOD 16] <> 0;
43 END;
44
45 PROCEDURE ClearAllBits (VAR S : SetType);
46
47 VAR n : WORD;
48
49 BEGIN
50   FOR n := 0 TO MaxMembers DO
51     S[n] := 0;
52 END;
53
54 PROCEDURE SetAllBits (VAR S : SetType);
55
56 VAR n : WORD;
57
58 BEGIN
59   FOR n := 0 TO MaxMembers DO
60     S[n] := Pred(MaxCardinality);
61 END;
62
63 PROCEDURE SetUnion (VAR S: SetType; CONST T, U : SetType);
64
65 VAR n : WORD;
66 BEGIN
67   FOR n := 0 TO Pred(MaxCardinality) DO
68     IF (InSet(T, n) OR InSet(U, n))

```

### 3. Big sets

```
69      THEN SetBit(S, n)
70      ELSE ClearBit(S, n);
71  END;
72
73 PROCEDURE SetIntersection (VAR S: SetType; CONST T, U : SetType);
74
75 VAR n : WORD;
76 BEGIN
77   FOR n := 0 TO Pred(MaxCardinality) DO
78     IF (InSet(T, n) AND InSet(U, n))
79       THEN SetBit(S, n)
80       ELSE ClearBit(S, n);
81  END;
82
83 PROCEDURE SetComplement (VAR S: SetType; CONST T, U : SetType);
84
85 VAR n : WORD;
86 BEGIN
87   FOR n := 0 TO Pred(MaxCardinality) DO
88     IF (InSet(T, n) AND NOT(InSet(U, n)))
89       THEN SetBit(S, n)
90       ELSE ClearBit(S, n);
91  END;
92
93 PROCEDURE SetSymDifference (VAR S: SetType; CONST T, U : SetType);
94
95 VAR H1, H2 : SetType;
96
97 BEGIN
98   SetComplement(H1, T, U);
99   SetComplement(H2, U, T);
100  SetUnion(S, H1, H2)
101 END;
102
103 FUNCTION SubSet (CONST S, T : SetType) : BOOLEAN;
104
105 VAR n : WORD;
106 BEGIN
107   FOR n := 0 TO Pred(MaxCardinality) DO
108     IF (InSet(S, n) AND NOT InSet(T, n)) // found one member OF S that IS
109       NOT IN T?
110     THEN
111       BEGIN
112         SubSet := FALSE;
113         EXIT;
114       END;
```

```

114 Result := TRUE; // IF none found
115 END;
116
117 FUNCTION EqualSet (CONST S, T : SetType) : BOOLEAN;
118
119 VAR n : WORD;
120 BEGIN
121   FOR n := 0 TO Pred(MaxCardinality) DO
122     IF (InSet(S, n) <> InSet(T, n)) // found one inequality?
123       THEN
124         BEGIN
125           Result := FALSE;
126           EXIT;
127         END;
128       Result := TRUE;
129   END;
130
131 FUNCTION TrueSubSet (CONST S, T : SetType) : BOOLEAN;
132
133 BEGIN
134   Result := Subset(S, T) AND NOT EqualSet(S, T);
135 END;
136
137 FUNCTION EmptySet (CONST S : SetType) : BOOLEAN;
138
139 VAR n : WORD;
140 BEGIN
141   FOR n := 0 TO Pred(MaxCardinality) DO
142     IF InSet(S, n) // found one member?
143       THEN
144         BEGIN
145           Result := FALSE;
146           EXIT;
147         END;
148       Result := TRUE;
149   END;
150
151 FUNCTION Cardinality (CONST S : SetType) : WORD;
152
153 VAR n, c : WORD;
154 BEGIN
155   c := 0;
156   FOR n := 0 TO Pred(MaxCardinality) DO
157     IF InSet(S, n) THEN INC(c); // count members
158   Result := c;
159 END;

```

### 3. Big sets

160

161      **END.**

# 4. Formula compiler

## Abstract

Formula compiler allow the user to enter a mathematical formula at run-time, which is evaluated by the program. Thus, programs can fulfill many different functions without the need of recompilation. This compiler can also perform symbolic differentiation.

As the unit in [1, 2] was originally written for CP/M, it required some modernisation, but otherwise is tried and tested.

The function `CompileExpression(Expr, VarTable, ExprPtr)` used here has the following properties:

- Each formula can contain several variables, whose names can be chosen freely as long as they start with a letter and are not identical to the reserved names (functions) of the compiler.
- The BOOLEan variable `CalcDecMode` decides whether new variables entered into a formula are added to the `VarTable` (which is then `nil` at the beginning), or whether the expression can contain only such variable as are already in the `VarTable`.
- The syntax of the expression string is the same as in `Pascal`, the expression must close with a “;”. Anything following the “;” is ignored.
- The final compilate is returned in `ExprPtr`, using reverse polish notation (UPN).

Any such `ExprPtr` can be evaluated by `CalcExpression(ExprPtr, VarTbl)`, where `VarTbl` contains the current values for all variables in the function. The variable `CalcResult` becomes `false` upon any errors during compilation or execution.

`Calc` can be made to handle additional functions:

1. The name of the function needs to be added to `Calc_symbols` and `Calc_Ids`
2. The actual algorithm is entered in the `case`-statement of `CalcExpression`.
3. The data type `Calc_operand` can be changed for example to `complex`, then all function calls need to be changed to their `complex` equivalents.

## 4.1. The source code

Listing 4.1: Interface

```

1  unit Calc;
2
3  interface
4
5  uses Crt, MathFunc;
6
7  const
8    Calc_IdLen = 10;
9    Calc_MaxVar = 10;
10   Calc_OpSize = 6;
11
12 type
13   ErrString = string[80];
14   Calc_IdStr = string[Calc_IdLen];
15   Calc_String = string[255];
16   Calc_Operand = float;
17
18   Calc_VarType = record
19     VarId: Calc_IdStr;
20     Value: Calc_Operand;
21   end;
22
23   Calc_VarTab = ^Calc_VarTable;
24   Calc_VarTable = array[0..Calc_MaxVar] of Calc_VarType;
25   Calc_Symbols = (Calc_Err, Calc_EOE, Calc_Const, Calc_Var, Calc_Pi,
26                   Calc_E, Calc_lp, Calc_rp, Calc_Neg, Calc_Add, Calc_Sub,
27                   Calc_Mul, Calc_Dvd, Calc_Div, Calc_Mod, Calc_ggT,
28                   Calc_kgV, Calc_Pow, Calc_sqr, Calc_Sqrt, Calc_Exp,
29                   Calc_Ln, Calc_Lg, Calc_Ld, Calc_Sin, Calc_Cos, Calc_Tan,
30                   Calc_Cot, Calc_ArcSin, Calc_ArcCos, Calc_ArcTan,
31                   Calc_ArcCot, Calc_Sinh, Calc_Cosh, Calc_Tanh, Calc_Coth,
32                   Calc_ArcSinh, Calc_ArcCosh, Calc_ArcTanh, Calc_ArcCoth,
33                   Calc_Abs, Calc_Deg, Calc_Rad, Calc_Rez, Calc_Fak,
34                   Calc_Sign, Calc_Int, Calc_End);
35
36   Calc_Prog = ^Calc_Instruct;
37
38   Calc_Instruct = record
39     NextInst: Calc_Prog;
40     Instruct: Calc_Symbols;
41     case Calc_Symbols of
42       Calc_Var : (VarIndex: integer);
43       Calc_Const: (Operand: Calc_Operand);

```

```

44      end;
45
46 var   CalcDecMod, CalcResult: boolean;
47
48
49 procedure CompileExpression(Expr: Calc_String; var VarTable: Calc_VarTab;
50   var ExprPtr: Calc_Prog);
51 { turn arithmetic expressions into UPN }
52
53 function CalcExpression(ExprPtr: Calc_Prog; VarTable: Calc_VarTab):
54   Calc_Operand;
55 { calculate the result of an expression }
56
57 function CalcDerivation(pptr: Calc_Prog; VarTab: Calc_VarTab;
58   Nach: Calc_IDStr): Calc_Prog;
59 { symbolic derivative of expression }
60
61 procedure CalcAOS(pptr: Calc_Prog; VarTable: Calc_VarTab);
62 { turn UPN-notation into AOS-formula and display }
63
64 procedure CalcError(ErrNum: integer; Message: ErrString);
65 {catch errors }
66
67 procedure KillExpression(var ExprPtr: Calc_Prog);
68 { delete Calc_prog if no longer needed }
69
70 function NewVarTab: Calc_VarTab;
71 { create new, empty variable table}
72
73 procedure KillVarTab(var VarTab: Calc_VarTab);
74 { delete variable table if no longer needed }
75
76 function SearchVarTab(VarTab: Calc_VarTab; ID: Calc_String): integer;
77 { search a variable in the variable table and return its index }
78
79 function AddToVarTab(VarTab: Calc_VarTab; ID: Calc_String): integer;
80 { enter a new variable into the variable table and return its index }
81
82 procedure AssignVari(VarTab: Calc_VarTab; i: integer; x: Calc_Operand);
83 { assign the i-th variable in the table the value x }
84
85 procedure AssignVar(VarTab: Calc_VarTab; ID: Calc_String; x: Calc_Operand);
86 { assign the value x to the variable ID in the variable table}
87
88 procedure HelpFormula;
89 { produces a short help text }

```

## 4. Formula compiler

```
89
90 implementation
91
92 const
93   Calc_Ids: array [Calc_Symbols] of Calc_IdStr =
94     ('ERR', ';', 'CONST', 'VAR', 'PI', 'E', '(', ')', 'NEG',
95      '+', '-', '*', '/', 'DIV', 'MOD', 'GGT', 'KGV', '^',
96      'SQR', 'SQRT', 'EXP', 'LN', 'LG', 'LD', 'SIN', 'COS',
97      'TAN', 'COT', 'ARCSIN', 'ARCCOS', 'ARCTAN', 'ARCCOT',
98      'SINH', 'COSH', 'TANH', 'COTH', 'ARCSINH', 'ARCCOSH',
99      'ARCTANH', 'ARCCOTH', 'ABS', 'DEG', 'RAD', 'REZ', 'FAK',
100     'SGN', 'INT', 'END');
```

### 4.1.1. Administrative routines

This routine catches errors, prints an error message and sets the flag `CalcResult` to false so that the calling program may handle the situation gracefully:

```
1 procedure CalcError(ErrNum: integer; Message: ErrString);
2
3   var
4     Meldung: string;
5     ch: char;
6
7   begin
8     case ErrNum of
9       1: Meldung := ' *** Run time error: Floating point overflow' + Message;
10      2: Meldung := ' *** Run time error: Division bx zero' + Message;
11      3: Meldung := ' *** Run time error: Argument error in ' + Message;
12    else
13      Meldung := ' *** Run time error: ' + Message;
14    end;
15    ch := MathFunc.WriteErrorMessage(Meldung);
16    CalcResult := False;
17  end;
```

The following routine removes a Calc-program from the heap, this will be called after failed calls to `CompileExpression`, and may be called by user programs to avoid heap overflow:

```
1 procedure KillExpression(var ExprPtr: Calc_Prog);
2
3   var
4     NextPtr: Calc_Prog;
5
6   begin
7     while ExprPtr <> nil do
```

```

8   begin
9     NextPtr := ExprPtr^.NextInst;
10    Dispose(ExprPtr);
11    ExprPtr := NextPtr;
12  end;
13  ExprPtr := nil;
14 end;

```

This routine generates a new variable table

```

1 function NewVarTab: Calc_VarTab;
2
3 var
4   VarTab: Calc_VarTab;
5
6 begin
7   Result := nil;
8   try
9     begin
10       New(VarTab);
11       VarTab^[0].Value := 0.0;
12       Result := VarTab;
13     end
14   except
15     CalcError(0, 'not enough memory');
16   end
17 end;

```

This procedure removes a no longer needed variable table from memory

```

1 procedure KillVarTab(var VarTab: Calc_VarTab);
2
3 begin
4   if vartab <> nil then
5     Dispose(VarTab);
6   VarTab := nil;
7 end;

```

This routine searches for the variable named ID in the variable table. Returns 0 if variable is not found, the index otherwise:

```

1 function SearchVarTab(VarTab: Calc_VarTab; Id: Calc_String): integer;
2
3 var
4   i: integer;
5
6 begin
7   if vartab <> nil
8   then
9     begin

```

#### 4. Formula compiler

```

10      for i := 1 to Length(id) do
11          id[i] := upcase(id[i]);
12          i := Trunc(VarTab^0.Value);
13          VarTab^0.VarId := Copy(Id, 1, Calc_IdLen);
14          while VarTab^i.VarId <> Id do
15              i := Pred(i);
16              Result := i;
17          end
18      else
19          Result := 0;
20      end;

```

This function adds the variable ID to the next free position in the variable table and returns the position if space was available. Otherwise, the return value is -1.

```

1  function AddToVarTab(VarTab: Calc_VarTab; Id: Calc_String): integer;
2
3  var
4      i: integer;
5
6  begin
7      if VarTab <> nil
8          then
9              begin
10                 for i := 1 to Length(id) do
11                     id[i] := upcase(id[i]);
12                     i := Trunc(VarTab^0.Value);
13                     if i < Calc_MaxVar
14                         then
15                             begin
16                                 i := Succ(i);
17                                 VarTab^0.Value := i;
18                                 VarTab^i.VarId := Id;
19                                 VarTab^i.Value := 0;
20                             end
21                         else
22                             i := -1;
23                         Result := i;
24                     end
25                 else
26                     Result := -1;
27             end;

```

This routine assigns the value x to the variable at position i in the variable table

```

1  procedure AssignVarI(VarTab: Calc_VarTab; i: integer; x: Calc Operand);
2
3  begin
4      if vartab <> nil

```

```

5   then
6     begin
7       if (i > 0) and (i <= Trunc(VarTab^[0].Value))
8         then VarTab^[i].Value := x
9         else CalcError(0, 'value assigned to unknown variable');
10    end
11  else
12    CalcError(0, 'value assigned to unknown variable');
13 end;

```

This routine assigns the value x to the variable ID in the variable table

```

1 procedure AssignVar(VarTab: Calc_VarTab; Id: Calc_String; x: Calc_Operand);
2
3 begin
4   AssignVari(VarTab, SearchVarTab(VarTab, Id), x);
5 end;

1 procedure HelpFormula;
2
3 begin
4   writeln('The formula is entered in Pascal-syntax and must end with a
      semicolon.');
5   writeln;
6   writeln('The compiler ''knows'' the following constants and functions,
      which ');
7   writeln('can not be redefined:');
8   writeln('Constants: e, pi                                Basic operators: +, -,
      *, /, ^');
9   writeln('Integer: div, mod, ggt, kgv                  Logarithms: ln, lg,
      ld, exp');
10  writeln('sin, cos, tan, cot and the equivalent hyperbolic and arcus
      functions');
11  writeln('Various Functions: abs, deg, rad, fakt, sgn');
12  writeln;
13 end;

```

### 4.1.2. Symbolic differentiation

The formula-string is compiled into a UPN program ( $f(x) = (7+x)*x^3 \rightarrow @7x + x3 \wedge x$ , with @ the list anchor). This means, that for symbolic differentiation the routine would find the first operator only after reading over the operands. Therefore, the UPN-program is inverted first. The derivation itself is then performed by the recursive procedure `derive`.

The result is then simplified by `CalcSimplify`, which replaces operations with constants by their result, and handles operations with the constants 1 and 0. The implementation of the commutative law is still limited. Indeed, when I showed this program to my maths-teacher, he had great fun finding equations whose differential is simple, but result in fairly

#### 4. Formula compiler

complex (yet correct!) formulas in `calcDerivation`. There still is room for improvement here.

```

1  procedure invert(pptrstart: calc_prog);
2
3  var
4      pptr, pptr1, pptr2: calc_prog;
5      max, i: integer;
6      dummy: calc_instruct;
7
8  begin
9      if pptrstart <> nil
10     then
11         begin
12             pptr := pptrstart^.nextinst;
13             max := 0;
14             while pptr^.nextinst <> nil do
15                 begin
16                     pptr := pptr^.nextinst;
17                     max := Succ(max);
18                     end;
19                     pptr := pptrstart;
20                     repeat
21                         pptr := pptr^.nextinst;
22                         pptr1 := pptr;
23                         for i := 1 to max do
24                             pptr1 := pptr1^.nextinst;
25                             dummy := pptr^;
26                             pptr^ := pptr1^;
27                             pptr1^ := dummy;
28                             pptr2 := pptr^.nextinst;
29                             pptr^.nextinst := pptr1^.nextinst;
30                             pptr1^.nextinst := pptr2;
31                             max := max - 2;
32                         until max <= 0;
33                     end; { then }
34     end;

```

```

1  function endof(pptr: calc_prog): calc_prog;
2
3  var
4      help: calc_prog;
5      op: integer;
6
7  begin
8      if pptr <> nil
9          then

```

```

10   begin
11     op := 1;
12     repeat
13       help := pptr;
14       if pptr^.instruct in [calc_var, calc_const]
15         then op := Pred(op)
16       else if not (pptr^.instruct in [calc_neg, calc_sqr..calc_fak])
17         then
18           op := Succ(op);
19           pptr := pptr^.nextinst
20       until op = 0;
21     Result := help;
22   end
23   else
24     Result := nil;
25 end;

```

### 4.1.3. Simplification of Calc-programs

```

1  procedure calcSimplify(var pptr: calc_prog);
2
3  var
4    pptr1, help1, help2      : calc_prog;
5    op                      : integer;
6    dummy                   : calc_operand;
7    arg1, arg2, SimpleError : boolean;
8    helpinstruct            : calc_symbols;
9
10
11  function Equal(pptr1, pptr2: calc_prog): boolean;
12
13  var
14    help1, help2 : calc_prog;
15    check        : boolean;
16
17  begin
18    help1 := endof(pptr1);
19    help2 := endof(pptr2);
20    if (pptr1 <> nil) and (pptr2 <> nil) and (help1 <> nil) and (help2 <>
21      nil)
22    then
23      begin
24        check := True;
25        repeat
26          check := check and (pptr1^.instruct = pptr2^.instruct);
27          case pptr1^.instruct of

```

#### 4. Formula compiler

```

27          calc_const : check := check and (pptr1^.operand =
28              pptr2^.operand);
29          calc_var   : check := check and (pptr1^.varindex =
30              pptr2^.varindex)
31      end;
32          pptr1 := pptr1^.nextinst;
33          pptr2 := pptr2^.nextinst
34      until not check or (pptr1 = help1^.nextinst) and (pptr2 =
35          help2^.nextinst);
36          Result := check;
37      end
38  else
39      Result := False;
40  end; // Equal
41
42
43
44
45
46 begin
47 try
48 begin
49     vardummy := newvartab;
50     New(a);
51     New(b);
52     New(c);
53     a^ := pptr^;
54     b^ := pptr1^;
55     if pptr2 <> nil
56         then c^ := pptr2^;
57         a^.nextinst := nil;
58         New(expr);
59         expr^.nextinst := b;
60     if pptr2 <> nil
61         then
62             begin
63                 b^.nextinst := c;
64                 c^.nextinst := a;
65             end
66         else
67             b^.nextinst := a;
68     Result := calcexpression(expr, vardummy);
69     SimpleError := SimpleError or not calcresult;

```

```

70      Dispose(a);
71      Dispose(b);
72      Dispose(c);
73      Dispose(expr);
74      killvartab(vardummy);
75  end
76 except
77 begin
78     Result := 0.0;
79     SimpleError := True;
80 end;
81 end;
82 end; // Compute
83
84
85 procedure simple(pptr: calc_prog);
86
87 var
88     pptra, pptrib : calc_prog;
89
90
91 procedure restoreptr;
92
93 begin
94     pptra := pptr^.nextinst;
95     pptrib := endof(pptra);
96     if pptrib <> nil
97         then pptrib := pptrib^.nextinst;
98 end;
99
100
101 procedure erase_entry;
102
103 begin
104     while help1 <> pptrib do
105     begin
106         help2 := help1;
107         help1 := help1^.nextinst;
108         Dispose(help2);
109     end;
110 end;
111
112
113 procedure pusha;
114
115 begin

```

#### 4. Formula compiler

```
116     pptr^ := pptra^;
117     help1 := pptr;
118     while help1^.nextinst <> pptrib do
119         help1 := help1^.nextinst;
120         help1^.nextinst := pptrib^.nextinst;
121         Dispose(pptra);
122         Dispose(pptrib);
123         restoreptr;
124     end;
125
126
127     procedure skipa;
128
129     begin
130         help1 := pptr^.nextinst;
131         erase_entry;
132         pptr^ := pptrib^;
133         Dispose(pptrib);
134         restoreptr;
135     end;
136
137
138     procedure setconst(dummy: calc_operand);
139
140     begin
141         pptr^.instruct := calc_const;
142         pptr^.operand := dummy;
143         pptr^.nextinst := pptrib^.nextinst;
144         help1 := pptra;
145         erase_entry;
146         Dispose(pptrib);
147         restoreptr;
148     end;
149
150     begin // Simple
151     if pptr <> nil
152         then
153             begin
154                 restoreptr;
155                 if pptr^.instruct in [calc_neg, calc_sqr..calc_fak]
156                     then
157                         begin
158                             simple(pptra);
159                             if pptra^.instruct = calc_const // ausrechnen !
160                                 then
161                                     begin
```

```

162     dummy := compute(pptr, pptra, nil);
163     if dummy = -0.0
164         then dummy := 0.0;
165     pptra^.instruct := calc_const;
166     pptra^.operand := dummy;
167     pptra^.nextinst := pptra^.nextinst;
168     Dispose(pptra);
169     restoreptr;
170   end;
171   if pptra^.instruct = calc_neg
172     then
173       begin
174         if pptra^.instruct = calc_neg
175           then
176             begin
177               pptra^ := pptra^.nextinst^;
178               Dispose(pptra^.nextinst);
179               Dispose(pptra);
180               restoreptr;
181             end;
182         end;
183         if (pptra^.instruct = calc_neg) and (pptra^.instruct in
184           [calc_mul.. calc_div])
185           then
186             begin
187               help1 := endof(pptra^.nextinst);
188               if pptra^.nextinst^.instruct = calc_const
189                 then
190                   begin
191                     pptra^.nextinst^.operand :=
192                       -pptra^.nextinst^.operand;
193                     pptra^ := pptra^;
194                     Dispose(pptra);
195                     restoreptr;
196                   end
197                 else
198                   if help1^.nextinst^.instruct = calc_const
199                     then
200                       begin
201                         help1^.nextinst^.operand :=
202                           -help1^.nextinst^.operand;
203                         pptra^ := pptra^;
204                         Dispose(pptra);
205                         restoreptr;
206                       end;
207                 end;
208             end;
209           end;
210         end;
211       end;
212     end;
213   end;
214 
```

#### 4. Formula compiler

```

206
207     end
208
209     else // jetzt werden Operationen mit Konstanten vereinfacht
210         if pptr^.instruct in [calc_add..calc_pow]
211             then
212                 begin
213                     simple(pptra);
214                     simple(pptrb);
215                     arg1 := pptr^.instruct = calc_const;
216                     arg2 := pptrb^.instruct = calc_const;
217                     if arg1 and arg2
218                         then
219                             begin
220                                 dummy := compute(pptr, pptrb, pptra);
221                                 pptr^.instruct := calc_const;
222                                 pptr^.operand := dummy;
223                                 pptr^.nextinst := pptrb^.nextinst;
224                                 Dispose(pptra);
225                                 Dispose(pptrb);
226                                 restoreptr;
227                             end
228                         else
229                             if arg2
230                                 then
231                                     begin
232                                         if pptrb^.operand = 0.0
233                                             then
234                                                 begin
235                                                     if pptr^.instruct in [calc_mul..
236                                         calc_div, calc_pow]
237                                             then
238                                                 setconst(0.0)
239                                             else
240                                                 if pptr^.instruct = calc_add
241                                                     then
242                                                         pusha
243                                                     else
244                                                         if pptr^.instruct = calc_sub
245                                                         then
246                                                             begin
247                                                                 pptr^.instruct :=
248                     calc_neg;
249                     help1 := endof(pptra);
250                     help1^.nextinst :=
251                         pptrb^.nextinst;
252                     Dispose(pptrb);
253                     restoreptr;

```

```

249           end;
250       end
251   else
252       if (pptrb^.operand = 1.0) and
253           (pptr^.instruct in
254               [calc_mul, calc_pow])
255           then
256               begin
257                   if pptr^.instruct = calc_mul
258                       then pusha
259                           else setconst(1.0);
260               end;
261   else
262       if arg1
263           then
264               begin
265                   if pptra^.operand = 0.0
266                       then
267                           begin
268                               if pptr^.instruct in [calc_mul,
269                                   calc_pow]
270                               then
271                                   begin
272                                       if pptr^.instruct = calc_mul
273                                           then dummy := 0.0
274                                               else dummy := 1.0;
275                                       pptr^.instruct := calc_const;
276                                       pptr^.operand := dummy;
277                                       help1 := pptrb;
278                                       op := 1;
279                                       repeat
280                                           help2 := help1;
281                                           if help1^.instruct in
282                                               [calc_add.. calc_pow]
283                                               then op := Succ(op)
284                                               else
285                                                   if help1^.instruct in
286                                                       [calc_const,
287                                                       calc_var]
288                                                       then op := Pred(op);
289                                                       help1 := help1^.nextinst;
290                                                       Dispose(help2);
291                                           until op = 0;
292                                           pptr^.nextinst := help1;
293                                           Dispose(pptra);

```

#### 4. Formula compiler

```

290                               restoreptr;
291                           end
292                       else
293                           if pptr^.instruct in [calc_add,
294                               calc_sub]
295                               then skipa;
296                           end
297                       else
298                           if (pptra^.operand = 1.0) and
299                               (pptr^.instruct in
[calc_mul..calc_div, calc_pow])
300                               then skipa;
301                           end;
302                           if (pptr^.instruct = calc_mul) and
303                               (pptr^.instruct in [calc_div,
304                               calc_dvd])
305                               then
306                                   begin
307                                       help1 := endof(pptra^.nextinst);
308                                       if (help1^.nextinst^.instruct =
309                                           calc_const)
310                                           then
311                                               begin
312                                                   pptr^ := pptra^;
313                                                   Dispose(pptra);
314                                                   Dispose(help1^.nextinst);
315                                                   help1^.nextinst := pptrib;
316                                                   restoreptr;
317                                               end
318                                           else
319                                               if help1^.nextinst^.operand =
320                                                   -1.0 then
321                                                   begin
322                                                       pptr^.instruct :=
323                                                       calc_neg;
324                                                       Dispose(help1^.nextinst);
325                                                       help1^.nextinst :=
326                                                       pptrib;
327                                                       restoreptr;
328                                                   end;
329                           if pptr^.instruct in [calc_mul..calc_div]
330                               then

```

#### 4.1. The source code

```

327      begin // Negationen vereinfachen
328      if pptr^>.instruct = calc_neg
329          then
330              if pptrb^>.instruct = calc_neg
331                  then
332                      begin
333                          pptr^>.nextinst := pptr^>.nextinst;
334                          Dispose(pptra);
335                          pptra := pptr^>.nextinst;
336                          help1 := endof(pptra);
337                          help1^>.nextinst := pptrb^>.nextinst;
338                          Dispose(pptrb);
339                          restoreptr;
340          end
341      else
342          begin
343              if ((pptrb^>.instruct =
344                  calc_const) and
345                  (pptrb^>.operand < 0.0))
346                  then
347                      begin
348                          pptr^>.nextinst := pptr^>.nextinst;
349                          Dispose(pptra);
350                          pptrb^>.operand := Abs(pptrb^>.operand);
351                          restoreptr;
352                      end;
353          end
354      else
355          if ((pptra^>.instruct =
356              calc_const) and
357              (pptra^>.operand < 0.0))
358              then
359                  if pptrb^>.instruct =
360                      calc_neg
361                      then
362                          begin
363                              pptra^>.nextinst := pptrb^>.nextinst;
364                              Dispose(pptrb);
365                              pptra^>.operand := Abs(pptra^>.operand);
366                              restoreptr;

```

#### 4. Formula compiler

```

362                                     end;
363     if (pptra^.instruct = calc_const)
364         and (pptra^.operand = -1.0)
365     then
366         begin
367             pptr^.instruct := calc_neg;
368             pptr^.nextinst := pptrib;
369             Dispose(pptra);
370             restoreptr;
371         end;
372     if ((pptrb^.instruct = calc_const)
373         and (pptrb^.operand = -1.0) and
374         (pptr^.instruct = calc_mul))
375     then
376         begin
377             help1 := endof(pptra);
378             help1^.nextinst :=
379                 pptrb^.nextinst;
380             pptr^.instruct := calc_neg;
381             Dispose(pptrb);
382             restoreptr;
383         end;
384     end;
385     if (pptr^.instruct = calc_add) and
386         (pptra^.instruct = calc_neg)
387     then
388         begin
389             pptr^.instruct := calc_sub;
390             pptr^.nextinst := pptra^.nextinst;
391             Dispose(pptra);
392             restoreptr;
393         end;
394     if (pptr^.instruct = calc_sub) and
395         (pptra^.instruct = calc_neg)
396     then
397         begin
398             pptr^.instruct := calc_add;
399             pptr^.nextinst := pptra^.nextinst;
400             Dispose(pptra);
401             restoreptr;
402         end;
403         // jetzt wird's schwierig : das
404         // Kommutativgesetz
405     if (((pptr^.instruct = calc_mul) and
406         (pptra^.instruct in
407 [calc_mul..calc_div])) or

```

#### 4.1. The source code

```

401                                     ((pptr^.instruct = calc_add) and
402                                     (pptra^.instruct in [calc_add,
403                                         calc_sub]))) and
404                                     (pptrb^.instruct = calc_const)
405                                     then
406                                         begin
407                                             help1 := endof(pptra^.nextinst);
408                                             help2 := endof(help1^.nextinst);
409                                             if pptra^.instruct in [calc_mul,
410                                         calc_add]
411                                             then
412                                                 begin
413                                                     if help1^.nextinst^.instruct
414                                                         = calc_const
415                                                         then
416                                                             begin
417                                                                 help2^.nextinst := pptra^.nextinst;
418                                                                 pptra^.nextinst := pptrb;
419                                                                 help2 := pptrb^.nextinst;
420                                                                 pptrb^.nextinst := help1^.nextinst;
421                                                                 help1^.nextinst := help2;
422                                                         end
423                                                         else
424                                                             if
425                                                               pptra^.nextinst^.instruct
426                                                               = calc_const
427                                                               then
428                                                                   begin
429                                                                       help2^.nextinst := pptrb^.nextinst;
430                                                                       pptrb^.nextinst := help1^.nextinst;
431                                                                       help1^.nextinst := pptrb;
432                                                                   end;
433                                                               end
434                                                               else
435                                                               begin
436                                                                   if help1^.nextinst^.instruct
437                                                                       = calc_const
438                                                                       then

```

#### 4. Formula compiler

```

432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
begin
    helpinstruct :=  

        pptr^.instruct;  

    pptr^.instruct :=  

        pptra^.instruct;  

    pptra^.instruct :=  

        helpinstruct;  

    pptr^.nextinst :=  

        pptra^.nextinst;  

    pptra^.nextinst :=  

        help1^.nextinst;  

    help1^.nextinst :=  

        pptra;  

end;  

end;  

restoreptr;  

simple(pptra);  

simple(pptrb);  

end  

else
if (((pptr^.instruct = calc_mul) and  

(pptrb^.instruct in  

[calc_mul..calc_div])) or  

((pptr^.instruct = calc_add) and  

(pptrb^.instruct in [calc_add,  

calc_sub]))) and  

(pptr^.instruct = calc_const)  

then
begin
    help1 := endof(pptrb^.nextinst);
    help2 := endof(help1^.nextinst);
    if pptrb^.instruct in
        [calc_add, calc_mul]
    then
        begin
            if
                pptrb^.nextinst^.instruct
                = calc_const
            then
                begin
                    pptr^.nextinst :=  

                        help1^.nextinst;
                    help1^.nextinst :=  

                        pptra;
                    pptra^.nextinst :=  

                        help2^.nextinst;
                end;
            end;
        end;
    end;
end;

```

#### 4.1. The source code

```

463           help2^.nextinst :=  
464               pptrib;  
465           end  
466       else  
467           if  
468               help1^.nextinst^.instruct  
469                   = calc_const  
470               then  
471                   begin  
472                       pptra^.nextinst  
473                           :=  
474                           pptrib^.nextinst;  
475                       pptra^.nextinst  
476                           :=  
477                           help1^.nextinst;  
478                       help1^.nextinst  
479                           := pptrib;  
480                       pptrib^.nextinst  
481                           := pptra;  
482                   end;  
483               end  
484           else  
485               begin  
486                   if  
487                       help1^.nextinst^.instruct  
488                           = calc_const  
489                       then  
490                           begin  
491                               helpinstruct :=  
492                                   pptra^.instruct;  
493                               pptra^.instruct :=  
494                                   pptrib^.instruct;  
495                               pptrib^.instruct :=  
496                                   helpinstruct;  
497                               pptra^.nextinst :=  
498                                   pptrib^.nextinst;  
499                               pptrib^.nextinst :=  
500                                   pptra;  
501                               pptra^.nextinst :=  
502                                   help1^.nextinst;  
503                               help1^.nextinst :=  
504                                   pptrib;  
505                           end;  
506                   end;  
507                   restoreptr;  
508                   simple(pptra);  


```

#### 4. Formula compiler

```

491           simple(pptrb);
492       end;
493       if pptr^.instruct = calc_mul
494           then
495       begin
496           if pptra^.instruct = calc_pow
497               then
498               begin
499                   help2 := pptra^.nextinst;
500                   help1 := endof(help2);
501                   help1 := help1^.nextinst;
502                   if (help2^.instruct =
503                       calc_const) and
504                       equal(help1, pptrb) then
505                   begin
506                       help2^.operand :=
507                           help2^.operand + 1.0;
508                       pptr^ := pptra^;
509                       Dispose(pptra);
510                       help2^.nextinst := pptrb;
511                       erase_entry;
512                       restoreptr;
513                   end;
514               end;
515           end;
516       if pptr^.instruct in [calc_add, calc_sub,
517                             calc_dvd, calc_div, calc_mul]
518           then
519               if equal(pptra, pptrb)
520                   then // sind die Operanden gleich ?
521                   begin
522                       case pptr^.instruct of
523                           calc_add: begin
524                               pptr^.instruct :=
525                                   calc_mul;
526                               help2 :=
527                                   endof(pptra);
528                               help1 :=
529                                   pptra^.nextinst;
530                               pptra^.instruct
531                                   := calc_const;
532                               pptra^.operand :=
533                                   2.0;
534                               pptra^.nextinst
535                                   :=
536                                   help2^.nextinst;
537                           calc_sub: begin
538                               pptr^.instruct :=
539                                   calc_mul;
540                               help2 :=
541                                   endof(pptra);
542                               help1 :=
543                                   pptra^.nextinst;
544                               pptra^.instruct
545                                   := calc_const;
546                               pptra^.operand :=
547                                   -2.0;
548                               pptra^.nextinst
549                                   :=
550                                   help2^.nextinst;
551                           calc_dvd: begin
552                               pptr^.instruct :=
553                                   calc_mul;
554                               help2 :=
555                                   endof(pptra);
556                               help1 :=
557                                   pptra^.nextinst;
558                               pptra^.instruct
559                                   := calc_const;
560                               pptra^.operand :=
561                                   1.0;
562                               pptra^.nextinst
563                                   :=
564                                   help2^.nextinst;
565                           calc_div: begin
566                               pptr^.instruct :=
567                                   calc_mul;
568                               help2 :=
569                                   endof(pptra);
570                               help1 :=
571                                   pptra^.nextinst;
572                               pptra^.instruct
573                                   := calc_const;
574                               pptra^.operand :=
575                                   1.0;
576                               pptra^.nextinst
577                                   :=
578                                   help2^.nextinst;
579                           calc_mul: begin
580                               pptr^.instruct :=
581                                   calc_const;
582                               help2 :=
583                                   endof(pptra);
584                               help1 :=
585                                   pptra^.nextinst;
586                               pptra^.instruct
587                                   := calc_const;
588                               pptra^.operand :=
589                                   1.0;
590                               pptra^.nextinst
591                                   :=
592                                   help2^.nextinst;
593                           calc_const: begin
594                               pptr^.instruct :=
595                                   calc_const;
596                               help2 :=
597                                   endof(pptra);
598                               help1 :=
599                                   pptra^.nextinst;
599                               pptra^.instruct
600                                   := calc_const;
601                               pptra^.operand :=
602                                   1.0;
603                               pptra^.nextinst
604                                   :=
605                                   help2^.nextinst;
606                           else begin
607                               pptr^.instruct :=
608                                   calc_const;
609                               help2 :=
610                                   endof(pptra);
611                               help1 :=
612                                   pptra^.nextinst;
613                               pptra^.instruct
614                                   := calc_const;
615                               pptra^.operand :=
616                                   1.0;
617                               pptra^.nextinst
618                                   :=
619                                   help2^.nextinst;
620                           end;
621                       end;
622                   end;
623               end;
624           end;
625       end;
626   end;
627 end;

```

```

524                     erase_entry;
525                     restoreptr;
526                     end;
527                     calc_sub: setconst(0.0);
528                     calc_div,
529                     calc_dvd: setconst(1.0);
530                     calc_mul: begin
531                         pptr^.instruct := calc_pow;
532                         help2 := endof(pptra);
533                         help1 := pptra^.nextinst;
534                         pptra^.nextinst
535                             :=
536                             help2^.nextinst;
537                             pptra^.instruct
538                                 := calc_const;
539                             pptra^.operand :=
540                             2.0;
541                             erase_entry;
542                             restoreptr;
543                             end;
544                             end; { case }
545                             end; { if equal }
546                     end; { pptr^.instruct in [calc_add..calc_pow] }
547                     end; { pptr <> nil }
548                 end; // Simple
549
550 begin // CalcSimplify
551     if pptr <> nil
552     then
553         begin
554             SimpleError := False;
555             CalcResult := True;
556             Invert(pptr);
557             pptr1 := pptr^.nextinst;
558             Simple(pptr1);
559             if SimpleError
560                 then
561                     begin
562                         KillExpression(pptr);
563                         CalcResult := False;
564                     end
565                 else
566                     invert(pptr);

```

#### 4. Formula compiler

```
563     end
564 else
565     calcresult := False;
566 end; // CalcSimplify
```

##### 4.1.4. Derivative of a function

```
1  function CalcDerivation(pptr: calc_prog; vartab: calc_vartab;
2                           nach: calc_idstr): calc_prog;
3
4  const
5      pi_durch_180 = 1.745329252e-2;
6
7  var
8      pptrstart, pptr1 : calc_prog;
9      ok              : boolean;
10
11
12  procedure Uppercase(var varid: calc_idstr);
13
14  var
15      i: integer;
16
17  begin
18      for i := 1 to Length(varid) do
19          varid[i] := Upcase(varid[i]);
20  end;
21
22
23  procedure NewConst(x: calc_operand);
24
25  var
26      pptr : calc_prog;
27
28  begin
29      try
30          begin
31              New(pptr);
32              pptr^.instruct := calc_const;
33              pptr^.operand := x;
34              pptr^.nextinst := pptrstart^.nextinst;
35              pptrstart^.nextinst := pptr;
36          end
37      except
38          calcresult := False;
39      end;
```

```

40 end;
41
42
43 procedure NewOp(id: calc_symbols);
44
45 var
46   pptr: calc_prog;
47
48 begin
49   try
50     begin
51       New(pptr);
52       pptr^.instruct := id;
53       pptr^.nextinst := pptrstart^.nextinst;
54       pptrstart^.nextinst := pptr;
55     end
56   except
57     calcresult := False;
58   end;
59 end;
60
61
62 procedure push(pptr: calc_prog);
63
64 var
65   pptr1: calc_prog;
66   op: integer;
67
68 begin
69   op := 1;
70   repeat
71     try
72       begin
73         if pptr^.instruct in [calc_add..calc_pow]
74           then op := op + 1
75         else
76           if not (pptr^.instruct in [calc_neg, calc_sqr..calc_fak])
77             then
78               op := op - 1;
79         New(pptr1);
80         pptr1^ := pptr^;
81         pptr1^.nextinst := pptrstart^.nextinst;
82         pptrstart^.nextinst := pptr1;
83         pptr := pptr^.nextinst;
84       end
85     except

```

#### 4. Formula compiler

```

86         calcresult := False
87         until (op = 0) or not calcresult;
88     end; // Push
89
90
91     procedure derive(pptr : calc_prog);
92
93     var
94         pptra, pptrib : calc_prog;
95
96     begin
97         if calcresult
98             then
99                 begin
100                     pptra := pptr^.nextinst;
101                     if (pptra <> nil)
102                         then
103                             begin
104                                 pptrib := endof(pptra);
105                                 pptrib := pptrib^.nextinst;
106                             end;
107                     case pptr^.instruct of
108                         calc_neg: begin
109                             newop(calc_neg);
110                             derive(pptra);
111                         end;
112                         calc_const,
113                         calc_div..calc_kgv,
114                         calc_fak : begin
115                             newconst(0.0);
116                         end;
117                         calc_var : begin
118                             if nach = vartab^[pptr^.varindex].varid
119                                 then newconst(1.0)
120                                 else newconst(0.0);
121                         end;
122                         calc_add : begin
123                             if calc_const in [pptra^.instruct, pptrib^.instruct]
124                                 then
125                                     if pptra^.instruct = calc_const
126                                         then derive(pptrib)
127                                         else derive(pptra)
128                                 else
129                                     begin
130                                         newop(calc_add);
131                                         derive(pptra);

```

```

132                     derive(pptrb);
133             end;
134         end;
135     calc_sub : begin
136         if calc_const in [pptra^.instruct, pptrb^.instruct]
137             then
138                 if pptra^.instruct = calc_const
139                     then derive(pptrb)
140                 else
141                     begin
142                         newop(calc_neg);
143                         derive(pptra);
144                     end
145                 else
146                     begin
147                         newop(calc_sub);
148                         derive(pptra);
149                         derive(pptrb);
150                     end;
151             end;
152     calc_mul : begin
153         if calc_const in [pptra^.instruct, pptrb^.instruct]
154             then
155                 if pptra^.instruct = calc_const
156                     then
157                         begin
158                             newop(calc_mul);
159                             push(pptra);
160                             derive(pptrb);
161                         end
162                     else
163                         begin
164                             newop(calc_mul);
165                             push(pptrb);
166                             derive(pptra);
167                         end
168                 else
169                     begin
170                         newop(calc_add);
171                         newop(calc_mul);
172                         derive(pptra);
173                         push(pptrb);
174                         newop(calc_mul);
175                         push(pptra);
176                         derive(pptrb);
177                     end;

```

#### 4. Formula compiler

```

178         end;
179 calc_dvd: begin
180     if pptra^.instruct = calc_const
181     then
182         begin
183             newop(calc_dvd);
184             push(pptra);
185             derive(pptrb);
186         end
187     else
188         begin
189             newop(calc_dvd);
190             newop(calc_sqr);
191             push(pptra);
192             newop(calc_sub);
193             newop(calc_mul);
194             derive(pptra);
195             push(pptrb);
196             newop(calc_mul);
197             push(pptra);
198             derive(pptrb);
199         end;
200     end;
201 calc_pow : begin
202     if (pptrb^.instruct = calc_const) and
203         (pptrb^.operand < 0.0)
204     then calcresult := False
205     else
206         begin
207             ok := False;
208             case pptra^.instruct of
209                 calc_const : ok := True;
210                 calc_var   : ok := nach <>
211                     vartab^[pptra^.varindex].varid
212             end;
213             if ok then
214                 begin
215                     newop(calc_mul);
216                     newop(calc_mul);
217                     newop(calc_pow);
218                     newop(calc_sub);
219                     newconst(1.0);
220                     push(pptra);
221                     push(pptrb);
222                     push(pptra);
223                     derive(pptrb);
224                 end;
225             end;
226         end;
227     end;
228 end;

```

```

222         end
223     else
224         begin
225             newop(calc_mul);
226             newop(calc_pow);
227             push(pptra);
228             push(pptrb);
229             newop(calc_add);
230             newop(calc_dvd);
231             push(pptrb);
232             newop(calc_mul);
233             push(pptra);
234             derive(pptrb);
235             newop(calc_mul);
236             derive(pptra);
237             newop(calc_ln);
238             push(pptrb);
239         end;
240     end;
241 end;
242 calc_abs : begin
243     newop(calc_mul);
244     newop(calc_sign);           // calc_sig ????
245     push(pptra);
246     derive(pptra);
247 end;
248 calc_int,
249 calc_sign: newconst(0.0);
250 calc_sqr : begin
251     newop(calc_mul);
252     newop(calc_mul);
253     push(pptra);
254     derive(pptra);
255     newconst(2.0);
256 end;
257 calc_sqrt: begin
258     newop(calc_dvd);
259     newop(calc_mul);
260     newconst(2.0);
261     newop(calc_sqrt);
262     push(pptra);
263     derive(pptra);
264 end;
265 calc_exp : begin
266     newop(calc_mul);
267     newop(calc_exp);

```

#### 4. Formula compiler

```
268         push(pptra);
269         derive(pptra);
270     end;
271 calc_ln : begin
272     newop(calc_dvd);
273     push(pptra);
274     derive(pptra);
275 end;
276 calc_lg : begin
277     newop(calc_dvd);
278     newop(calc_mul);
279     newop(calc_ln);
280     newconst(10.0);
281     push(pptra);
282     derive(pptra);
283 end;
284 calc_ld : begin
285     newop(calc_dvd);
286     newop(calc_mul);
287     newop(calc_ln);
288     newconst(2.0);
289     push(pptra);
290     derive(pptra);
291 end;
292 calc_sin : begin
293     newop(calc_mul);
294     newop(calc_cos);
295     push(pptra);
296     derive(pptra);
297 end;
298 calc_cos : begin
299     newop(calc_mul);
300     newop(calc_neg);
301     newop(calc_sin);
302     push(pptra);
303     derive(pptra);
304 end;
305 calc_tan : begin
306     newop(calc_dvd);
307     newop(calc_sqr);
308     newop(calc_cos);
309     push(pptra);
310     derive(pptra);
311 end;
312 calc_cot: begin
313     newop(calc_neg);
```

```

314         newop(calc_dvd);
315         newop(calc_sqr);
316         newop(calc_sin);
317         push(pptra);
318         derive(pptra);
319     end;
320     calc_arcsin : begin
321         newop(calc_dvd);
322         newop(calc_sqrt);
323         newop(calc_sub);
324         newop(calc_sqr);
325         push(pptra);
326         newconst(1.0);
327         derive(pptra);
328     end;
329     calc_arccos : begin
330         newop(calc_neg);
331         newop(calc_dvd);
332         newop(calc_sqrt);
333         newop(calc_sub);
334         newop(calc_sqr);
335         push(pptra);
336         newconst(1.0);
337         derive(pptra);
338     end;
339     calc_arctan: begin
340         newop(calc_dvd);
341         newop(calc_add);
342         newop(calc_sqr);
343         push(pptra);
344         newconst(1.0);
345         derive(pptra);
346     end;
347     calc_arccot : begin
348         newop(calc_neg);
349         newop(calc_dvd);
350         newop(calc_add);
351         newop(calc_sqr);
352         push(pptra);
353         newconst(1.0);
354         derive(pptra);
355     end;
356     calc_sinh : begin
357         newop(calc_mul);
358         newop(calc_cosh);
359         push(pptra);

```

#### 4. Formula compiler

```
360                     derive(pptra);
361             end;
362         calc_cosh : begin
363             newop(calc_mul);
364             newop(calc_sinh);
365             push(pptra);
366             derive(pptra);
367         end;
368         calc_tanh : begin
369             newop(calc_dvd);
370             newop(calc_sqrt);
371             newop(calc_cosh);
372             push(pptra);
373             derive(pptra);
374         end;
375         calc_coth : begin
376             newop(calc_neg);
377             newop(calc_dvd);
378             newop(calc_sqrt);
379             newop(calc_sinh);
380             push(pptra);
381             derive(pptra);
382         end;
383         calc_arcsinh : begin
384             newop(calc_dvd);
385             newop(calc_sqrt);
386             newop(calc_add);
387             newconst(1.0);
388             newop(calc_sqr);
389             push(pptra);
390             derive(pptra);
391         end;
392         calc_arccosh : begin
393             newop(calc_dvd);
394             newop(calc_sqrt);
395             newop(calc_sub);
396             newconst(1.0);
397             newop(calc_sqr);
398             push(pptra);
399             derive(pptra);
400         end;
401         calc_arctanh,
402         calc_arccoth : begin
403             newop(calc_dvd);
404             newop(calc_sub);
405             newop(calc_sqrt);
```

```

406                     push(pptr);
407                     newconst(1.0);
408                     derive(pptr);
409                     end;
410         calc_deg : begin
411             newop(calc_dvd);
412             newconst(pi_durch_180);
413             derive(pptr);
414             end;
415         calc_rad: begin
416             newop(calc_mul);
417             newconst(pi_durch_180);
418             derive(pptr);
419             end;
420         else      calcresult := False
421         end; // case
422     end; // if CalcResult
423 end; // Derive

424
425 begin // CalcDerivation
426 if pptr <> nil
427 then
428     begin
429         uppercase(nach);
430         invert(pptr);
431         pptr1 := pptr;
432         New(pptrstart);
433         pptrstart^.nextinst := nil;
434         pptr := pptr^.nextinst;
435         calcresult := True;
436         derive(pptr);
437         if calcresult then
438             begin
439                 calcSimplify(pptrstart);
440                 Result := pptrstart;
441             end
442         else
443             begin
444                 killexpression(pptrstart);
445                 Result := nil;
446                 CalcError(4, 'Derivate of function not known');
447             end;
448             invert(pptr1);
449         end
450     else
451         begin

```

#### 4. Formula compiler

```
452     Result := nil;
453     CalcResult := False;
454   end;
455 end; // CalcDerivation
```

#### 4.1.5. Compile the function

```

1  procedure CompileExpression(Expr: Calc_String; var VarTable: Calc_VarTab;
2    var ExprPtr: Calc_Prog);
3
4  var
5    VarTabFlag, ParsError, EndOfExpr : boolean;
6    ch                               : string[1];      // akt. Zeichen aus
7    String
8    LastPos, StrPos                 : integer;        // zaehlt
9    String-Position mit
10   TempIdent, Ident               : Calc_String;    // enth. aktuellen
11   Bezeichner
12   Symbol,                      // akt. Symbol des
13   Bezeichners
14   LastSymbol                   : Calc_Symbols; // vorheriges Symbol
15   Number                        : Calc_Operand; // akt. Zahl/Index aus
16   String
17   ProgPtr                       : Calc_Prog;
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
procedure Error(ErrPos: integer; ErrMsg: Calc_String);

begin
  if not ParsError
  then
    begin
      Writeln;
      Write('*** ', ' Error compiling this expression: ');
      ClrEol;
      Writeln;
      Write(' ', Expr);
      ClrEol;
      Writeln;
      Write(' ': ErrPos, '^');
      ClrEol;
      Writeln;
      Write(ErrMsg, '!');
      Clreol;
      Writeln;
      ClrEol;

```

```

35         WriteLn;
36         ClrEol;
37     end;
38     ParsError := True;
39     Symbol := Calc_Err;
40 end;
41
42
43 procedure Add_To_Queue(op: Calc_Symbols; x: Calc_Operand);
44 { add next operand to que }
45
46 var
47     UPN_Entry: Calc_Prog;
48
49 begin
50     try // is enough memory available?
51     begin
52         New(UPN_Entry);
53         with UPN_Entry^ do
54         begin
55             NextInst := nil;
56             Instruct := op;
57             case op of
58                 Calc_Var : VarIndex := Trunc(x);
59                 Calc_Const : Operand := x
60             end;
61         end;
62         ProgPtr^.NextInst := UPN_Entry;
63         ProgPtr := ProgPtr^.NextInst;
64     end
65     except
66         Error(1, 'not enough free memory');
67     end;
68 end;
69
70
71 procedure GetSymbol;
72 { get next symbol from string }
73
74
75 procedure GetChar;
76 { get next character from string }
77
78 begin
79     ch := ' ';
80     StrPos := Succ(StrPos);

```

#### 4. Formula compiler

```

81      EndOfExpr := (StrPos > Length(Expr));
82      if not EndOfExpr
83          then ch := UpCase(Expr[StrPos]);
84      end;
85
86
87      procedure GetNumber;
88 { Get the next number from string. The Turbo-Pascal val-procedure wants
89   to see only valid characters of a floating point number, NumberEnd
90   points to the first invalid character. Hence:
91   }
92
93      var
94          NumberStr: Calc_String;
95          NumberEnd, posi: integer;
96
97      begin
98          NumberStr := Copy(Expr, StrPos, 255); // everything from first figure
99          NumberStr := NumberStr + '      ';
100         posi := 1;
101         while (not (numberstr[posi] in ['e', 'E'])) and (posi <
102             length(numberstr)) do
103             posi := succ(posi);
104         if numberstr[posi] in ['e', 'E']
105             then
106                 begin
107                     if numberstr[posi + 1] in ['+', '-']
108                         then posi := succ(posi);
109                     if not (numberstr[posi + 1] in ['0'..'9'])
110                         then error(StrPos + posi, 'incomplete expression')
111                     else
112                         if ((numberstr[posi + 1] = '3') and (numberstr[posi + 2] in
113                             ['7'..'9'])) or ((numberstr[posi + 1] > '3') and
114                             (numberstr[posi + 2] in ['0'..'9']))
115                         then error(StrPos + posi, 'number out of range');
116                 end;
117         Val(NumberStr, Number, NumberEnd);
118         if NumberEnd > 0
119             then // invalid character
120                 begin // number not at the end of expression
121                     StrPos := StrPos + NumberEnd - 2;
122                     NumberStr := Copy(NumberStr, 1, Pred(NumberEnd));
123                     Val(NumberStr, Number, NumberEnd);
124                 end
125             else // worked, at the end of expression
126                 StrPos := Length(Expr);

```

```

125    end;
126
127
128    procedure searchsymtab;
129    { look up symbol in symbol table }
130
131    var
132        symok: boolean;
133
134    begin
135        symok := False;
136        symbol := calc_err;
137        while (symbol < calc_end) and not symok do
138            begin
139                symbol := Succ(symbol);
140                symok := (ident = calc_ids[symbol]);
141            end;
142            if not symok
143            then symbol := calc_err;
144    end;
145
146    begin // GetSymbol
147        LastPos := StrPos;
148        LastSymbol := Symbol;
149        Ident := '';
150        while (ch = ' ') and not EndOfExpr do
151            GetChar;
152        case ch[1] of
153            'A'..'Z': repeat      // Name of operator, function or variable
154                Ident := Concat(Ident, ch);
155                GetChar;
156                until not (ch[1] in ['A'..'Z', '0'..'9']);
157            '0'..'9', '.':
158                begin
159                    GetNumber;
160                    Ident := 'CONST';
161                    GetChar;
162                end
163            else begin          // +, -, etc.
164                Ident := ch;
165                GetChar;
166            end
167        end;
168        SearchSymtab;
169        if Symbol = Calc_Err
170            then // operator not identified

```

#### 4. Formula compiler

```

171         if Ident[1] in ['A'..'Z']
172             then Symbol := Calc_Var // use as variable
173             else
174                 if Ident <> ''
175                     then Error(LastPos, 'unknown symbol');
176             if Symbol = Calc_Var
177                 then
178                     if LastSymbol <> Calc_Var
179                         then
180                             begin
181                                 Number := SearchVarTab(VarTable, Ident);
182                                 if CalcDecMod and (Number = 0.0)
183                                     then // refuse unidentified string
184                                         Error(LastPos, 'unknown symbol')
185                                     else // or accept it as new var
186                                         if Number = 0.0
187                                             then
188                                                 begin
189                                                     Number := AddToVarTab(VarTable, Ident);
190                                                     if Number < 0.0
191                                                         then Error(LastPos, 'too many variables');
192                                                 end;
193                                             end
194                                         else
195                                             if not EndOfExpr
196                                                 then Error(LastPos, 'operator expected');
197                                         end; // GetSymbol
198
199
200     procedure Expression;
201     { expression contains several parts connected by operators }
202
203     var
204         ExprOp: Calc_Symbols;
205
206     procedure Term;
207     { expression contains several factors }
208
209     var
210         TermOp: Calc_Symbols;
211
212     procedure Factor(fparen: boolean);
213     { Factors can be variables, constants, functions or an expression in
214       brackets, and all of them can be raised to a power.
215       The parameter 'fparen' indicates whether brackets are for an
216       expression

```

```

216     or a function. In the latter case, the function needs to be
217         evaluated
218     before raising to power
219
220     }
221
222     var
223         FacOp: Calc_Symbols;
224
225     begin
226         if Symbol <> Calc_Err
227             then
228                 begin
229                     case Symbol of
230                         Calc_Var    : begin
231                             Add_To_Queue(Calc_Var, Number);
232                             GetSymbol;
233                             end;
234                         Calc_Const   : begin
235                             Add_To_Queue(Calc_Const, Number);
236                             GetSymbol;
237                             end;
238                         Calc_Pi      : begin
239                             Add_To_Queue(Calc_Const, Const_pi);
240                             GetSymbol;
241                             end;
242                         Calc_E       : begin
243                             Add_To_Queue(Calc_Const, Const_e);
244                             GetSymbol;
245                             end;
246                         Calc_lp      : begin           // expression in brackets
247                             GetSymbol;
248                             Expression;
249                             if (Symbol <> Calc_rp)
250                                 then Error(StrPos - Ord(ch[1] > ' '),
251                                         Concat(Calc_Ids[Calc_rp], ' '
252                                             expected'));
253                             GetSymbol;
254                             end;
255                         Calc_Pow     : ;           // power, dealt with later
256                         Calc_Sqr..;
257                         Calc_Fak    : begin // funktion
258                             FacOp := Symbol;
259                             GetSymbol;
260                             if (Symbol = Calc_lp)
261                                 then Factor(True)
262                             else Error(LastPos,

```

#### 4. Formula compiler

```

259                                     Concat(Calc_Ids[Calc_lp], ' expected'));
260                                     Add_To_Queue(FacOp, 0);
261                                 end
262                             else          Error(LastPos, 'here unexpected')
263                         end; // CASE
264                         if not fparen
265                             then // brackets not for function, therefore..
266                             if Symbol = Calc_Pow
267                                 then // power
268                                     if LastSymbol in [Calc_Const, Calc_Var, Calc_rp,
269                                         Calc_PI, Calc_e]
270                                         then
271                                             begin // evaluate
272                                                 GetSymbol;
273                                                 Factor(False);
274                                                 Add_To_Queue(Calc_Pow, 0);
275                                             end
276                                         else      // power here not possible
277                                             Error(Pred(StrPos), 'here unexpected');
278                                     end // IF Symbol <> Calc_Err
279                                 else
280                                     Error(StrPos - Ord(EndOfExpr), ' incomplete expression');
281                             end; // Faktor
282
283                         begin // Term
284                             Factor(False);
285                             if Symbol in [Calc_Mul..Calc_kgV]
286                                 then // term contains several factors
287                                     begin
288                                         TermOp := Symbol;
289                                         GetSymbol;
290                                         Term;
291                                         Add_To_Queue(TermOp, 0);
292                                     end;
293                             end; // Term
294
295                         begin // Expression
296                             if Symbol in [Calc_Add..Calc_Sub]
297                                 then // expression starts with + or -
298                                     begin
299                                         ExprOp := Symbol;
300                                         GetSymbol;
301                                         Term;
302                                         if ExprOp = Calc_Sub
303                                             then Add_To_Queue(Calc_Neg, 0);      // negate everything
304                                     end
305                                 end
306                             else          Error(LastPos, 'here unexpected');
307                         end
308                     end
309                 end
310             end
311         end
312     end
313 
```

```

303     else
304         Term;
305     while (Symbol in [Calc_Add..Calc_Sub]) do
306         begin                                // additional terms follow
307             ExprOp := Symbol;
308             GetSymbol;
309             Term;
310             Add_To_Queue(ExprOp, 0);
311         end;
312     end; // Expression
313
314 begin // CompileExpression
315     Symbol := Calc_Err;
316     ParsError := False;
317     EndOfExpr := False;
318     StrPos := 0;
319     ch := ' ';
320     VarTabFlag := (VarTable = nil);
321     if VarTabFlag
322         then VarTable := NewVarTab;
323     New(ExprPtr);
324     ExprPtr^.NextInst := nil;
325     ProgPtr := ExprPtr;
326     GetSymbol;
327     Expression;
328     if Symbol <> Calc_EOE
329         then Error(LastPos, ";" + ' expected');
330     CalcResult := not ParsError;
331     if ParsError
332         then
333             begin
334                 KillExpression(ExprPtr);
335                 if VarTabFlag
336                     then KillVarTab(VarTable);
337             end;
338     end; // CompileExpression

```

#### 4.1.6. Calculate value of an expression

```

1 function CalcExpression(ExprPtr: Calc_Prog; VarTable: Calc_VarTab):
2     Calc_Operand;
3 { evaluate an RPN-expression }
4
5 const
6     StackSize = 50;

```

#### 4. Formula compiler

```
7  var
8      x          : Calc_Operand;
9      StackPtr   : integer;
10     Stack      : array[1..StackSize] of Calc_Operand;
11
12
13    procedure Push;                                // pushes number onto the stack
14
15    begin
16        Stack[StackPtr] := x;
17        StackPtr := Succ(StackPtr);
18    end;
19
20
21    function Pop: Calc_Operand;                   // gets a number from stack
22
23    begin
24        StackPtr := Pred(StackPtr);
25        Result := Stack[StackPtr];
26    end;
27
28    begin // CalcExpression
29        CalcResult := True;
30        if (ExprPtr <> nil) and (VarTable <> nil)
31        then
32            begin
33                ExprPtr := ExprPtr^.NextInst;
34                StackPtr := 1;
35                x := 0.0;
36                while ExprPtr <> nil do
37                    begin
38                        with ExprPtr^ do
39                            case Instruct of
40                                Calc_Const : begin
41                                    Push;
42                                    x := Operand;
43                                    end;
44                                Calc_Var   : begin
45                                    Push;
46                                    x := VarTable^[VarIndex].Value;
47                                    end;
48                                else
49                                    case Instruct of
50                                        Calc_Neg   : x := -x;
51                                        Calc_Add   : x := Pop + x;
52                                        Calc_Sub   : x := Pop - x;
```

```

53      Calc_Mul   : x := Pop * x;
54      Calc_Dvd   : if x <> 0.0
55          then x := Pop / x
56          else CalcError(2, ' ');
57      Calc_Div   : if Trunc(x) <> 0
58          then x := Trunc(Pop) div
59              Trunc(x)
60          else CalcError(2, ' ');
61      Calc_Mod   : if Trunc(x) <> 0
62          then x := Trunc(Pop) mod
63              Trunc(x)
64          else CalcError(2, ' ');
65      Calc_ggT   : x := GCD(Trunc(Pop), Trunc(x));
66      Calc_kgV   : x := SCM(Trunc(Pop), Trunc(x));
67      Calc_Pow   : x := pot(Pop, x);
68      Calc_Sqr   : x := Sqr(x);
69      Calc_Sqrt  : if x >= 0.0
70          then x := Sqrt(x)
71          else CalcError(3, 'Sqrt(x): '
72              x <= 0');
73      Calc_Exp   : x := Exp(x);
74      Calc_Ln    : if x > 0.0
75          then x := Ln(x)
76          else CalcError(3, 'Ln(x): x '
77              <= 0');
78      Calc_Lg    : x := log(x, 10);
79      Calc_Ld    : x := log(x, 2);
80      Calc_Sin   : x := Sin(x);
81      Calc_Cos   : x := Cos(x);
82      Calc_Tan   : x := tan(x);
83      Calc_Cot   : x := cot(x);
84      Calc_ArcSin: x := arcsin(x);
85      Calc_ArcCos: x := arccos(x);
86      Calc_ArcTan: x := ArcTan(x);
87      Calc_ArcCot: x := arccot(x);
88      Calc_Sinh   : x := sinh(x);
89      Calc_Cosh   : x := cosh(x);
90      Calc_Tanh   : x := tanh(x);
91      Calc_Coth   : x := coth(x);
92      Calc_ArcSinh: x := arsinh(x);
93      Calc_ArcCosh: x := arcosh(x);
94      Calc_ArcTanh: x := artanh(x);
95      Calc_ArcCoth: x := arcoth(x);
96      Calc_Abs    : x := abs(x);
97      Calc_Deg    : x := grad(x);
98      Calc_Rad    : x := rad(x);

```

#### 4. Formula compiler

```

95          Calc_Rez   : if x <> 0
96              then x := 1 / x
97              else CalcError(3, '1 / 0');
98          Calc_Fak   : x := fak(Round(x));
99          Calc_Int   : x := Int(x);
100         Calc_Sign  : x := Signum(x);
101         else           CalcError(0, 'Function not
102                           known')
103         end; // CASE
104         end; // ELSE
105     end; // CASE
106     ExprPtr := ExprPtr^.NextInst;
107     if StackPtr > StackSize then CalcError(0, 'Stack overflow');
108     if not CalcResult then ExprPtr := nil;
109     end; // WHILE
110     Result := x;
111     end // THEN
112 else
113     CalcError(0, 'Function not known');
114 end;

```

#### 4.1.7. Screen output of calc programs

The recursive procedure `CalcAOS` converts RPN-programs into “normal” formulas for screen output.

```

1  procedure CalcAOS(pptr: Calc_Prog; VarTable: Calc_VarTab);
2
3  var
4      Value   : string[50];
5      len     : byte ABSOLUTE Value;
6      key     : char;
7      dummy   : calc_operand;
8      pptr1   : calc_prog;
9
10
11  procedure writeaos(pptr: calc_prog);
12
13  var
14      pptra, pptrib : calc_prog;
15      paren        : boolean;
16
17  begin
18      if (pptr <> nil)
19          then
20              begin

```

```

21    if keypressed
22    then
23        begin
24            key := ReadKey;
25            if key = ^s then Key := readkey;
26        end;
27        pptr_a := pptr^.nextinst;
28        if pptr_a <> nil
29        then
30            begin
31                pptr_b := endof(pptr_a);
32                pptr_b := pptr_b^.nextinst;
33            end;
34        case pptr^.instruct of
35            calc_const: begin
36                dummy := pptr^.operand;
37                Str(dummy: 0: 10, Value);
38                while Value[len] = '0' do
39                    len := Pred(len);
40                if Value[len] = '.' then len := Pred(len);
41                paren := dummy < 0.0;
42                if paren then Write('(');
43                Write(Value);
44                if paren then Write(')');
45            end;
46            calc_var : Write(vartable^[pptr^.varindex].varid);
47            calc_add..
48            calc_pow : begin
49                paren := (pptr^.instruct in [calc_mul..calc_pow])
50                    and
51                        (pptr_b^.instruct in [calc_add,
52                            calc_sub]);
53                paren := paren or (pptr^.instruct = calc_pow) and
54                    (pptr_b^.instruct in
55                        [calc_add..calc_pow]);
56                if paren then Write('(');
57                writeaos(pptr_b);
58                if paren then Write(')');
59                if pptr^.instruct in [calc_div..Calc_Kgv] then
                    Write(' ');
                    Write(calc_ids[pptr^.instruct]);
                if pptr^.instruct in [calc_div..Calc_Kgv] then
                    Write(' ');
                paren := (pptr^.instruct in [calc_mul..calc_pow])
                    and

```

#### 4. Formula compiler

```

60                                     (pptra^.instruct in [calc_add,
61                                         calc_sub]);
62                                     paren := paren or ((pptr^.instruct in
63                                         [calc_dvd..calc_pow])
64                                         and (pptra^.instruct in
65                                         [calc_add..calc_pow]));
66                                     paren := paren or ((pptr^.instruct = calc_sub) and
67                                         (pptra^.instruct in [calc_add,
68                                         calc_sub]));
69                                     if paren then Write('(');
70                                         writeaos(pptra);
71                                         if paren then Write(')');
72                                         end;
73                                     calc_neg : begin
74                                         Write('(-');
75                                         paren := pptra^.instruct in [calc_add..calc_pow];
76                                         if paren then Write('(');
77                                         writeaos(pptra);
78                                         if paren then Write(')');
79                                         Write(')');
80                                         end;
81                                     calc_sqr..
82                                     calc_fak : begin
83                                         Write(calc_ids[pptr^.instruct], '(');
84                                         writeaos(pptra);
85                                         Write(')');
86                                         end
87                                     end; // case
88                                 end; // then
89                             end; // WriteAOS
90
91 begin // CalcAOS
92     if pptr <> nil
93         then
94             begin
95                 pptr1 := pptr;
96                 invert(pptr);
97                 pptr := pptr^.nextinst;
98                 writeaos(pptr);
99                 invert(pptr1);
100                Writeln;
101            end
102        else
103            Writeln('Function not defined');
104    end; // CalcAOS
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1088
1089
1090
1091
1092
1093
1094
1095
1096
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1186
1187
1188
1189
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1208
1209
1210
1211
1212
1213
1214
1215
1216
1216
1217
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1227
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1246
1247
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1257
1258
1259
1260
1261
1262
1263
1264
1265
1265
1266
1267
1268
1268
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1278
1278
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1288
1288
1289
1290
1291
1292
1293
1294
1294
1295
1296
1297
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1306
1307
1308
1309
1309
1310
1311
1312
1313
1314
1315
1315
1316
1317
1318
1318
1319
1320
1321
1322
1323
1324
1325
1325
1326
1327
1328
1328
1329
1330
1331
1332
1333
1334
1335
1335
1336
1337
1338
1338
1339
1340
1341
1342
1343
1344
1345
1345
1346
1347
1348
1348
1349
1350
1351
1352
1353
1354
1355
1355
1356
1357
1358
1358
1359
1360
1361
1362
1363
1364
1364
1365
1366
1367
1367
1368
1369
1370
1371
1372
1373
1373
1374
1375
1376
1376
1377
1378
1379
1379
1380
1381
1382
1383
1384
1385
1385
1386
1387
1388
1388
1389
1390
1391
1392
1393
1393
1394
1395
1396
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1404
1405
1406
1407
1407
1408
1409
1410
1411
1412
1412
1413
1414
1415
1415
1416
1417
1418
1418
1419
1420
1421
1422
1423
1423
1424
1425
1426
1426
1427
1428
1429
1429
1430
1431
1432
1433
1434
1434
1435
1436
1437
1437
1438
1439
1440
1441
1442
1442
1443
1444
1445
1445
1446
1447
1448
1448
1449
1450
1451
1452
1453
1453
1454
1455
1456
1456
1457
1458
1459
1459
1460
1461
1462
1463
1463
1464
1465
1466
1466
1467
1468
1469
1469
1470
1471
1472
1472
1473
1474
1475
1475
1476
1477
1478
1478
1479
1480
1481
1482
1483
1483
1484
1485
1486
1486
1487
1488
1489
1489
1490
1491
1492
1493
1493
1494
1495
1496
1496
1497
1498
1499
1499
1500
1501
1502
1503
1503
1504
1505
1506
1506
1507
1508
1509
1509
1510
1511
1512
1512
1513
1514
1515
1515
1516
1517
1518
1518
1519
1520
1521
1522
1522
1523
1524
1525
1525
1526
1527
1528
1528
1529
1530
1531
1532
1532
1533
1534
1535
1535
1536
1537
1538
1538
1539
1540
1541
1542
1542
1543
1544
1545
1545
1546
1547
1548
1548
1549
1550
1551
1552
1552
1553
1554
1555
1555
1556
1557
1558
1558
1559
1560
1561
1562
1562
1563
1564
1565
1565
1566
1567
1568
1568
1569
1570
1571
1572
1572
1573
1574
1575
1575
1576
1577
1578
1578
1579
1580
1581
1582
1582
1583
1584
1585
1585
1586
1587
1588
1588
1589
1590
1591
1592
1592
1593
1594
1595
1595
1596
1597
1598
1598
1599
1599
1600
1601
1602
1602
1603
1604
1605
1605
1606
1607
1608
1608
1609
1610
1611
1611
1612
1613
1614
1614
1615
1616
1617
1617
1618
1619
1619
1620
1621
1622
1622
1623
1624
1625
1625
1626
1627
1627
1628
1629
1629
1630
1631
1632
1632
1633
1634
1635
1635
1636
1637
1637
1638
1639
1639
1640
1641
1642
1642
1643
1644
1644
1645
1646
1646
1647
1648
1648
1649
1650
1651
1651
1652
1653
1653
1654
1655
1655
1656
1657
1657
1658
1659
1659
1660
1661
1661
1662
1663
1663
1664
1665
1665
1666
1667
1667
1668
1669
1669
1670
1671
1671
1672
1673
1673
1674
1675
1675
1676
1677
1677
1678
1679
1679
1680
1681
1681
1682
1683
1683
1684
1685
1685
1686
1687
1687
1688
1689
1689
1690
1691
1691
1692
1693
1693
1694
1695
1695
1696
1697
1697
1698
1699
1699
1700
1701
1701
1702
1703
1703
1704
1705
1705
1706
1707
1707
1708
1709
1709
1710
1711
1711
1712
1713
1713
1714
1715
1715
1716
1717
1717
1718
1719
1719
1720
1721
1721
1722
1723
1723
1724
1725
1725
1726
1727
1727
1728
1729
1729
1730
1731
1731
1732
1733
1733
1734
1735
1735
1736
1737
1737
1738
1739
1739
1740
1741
1741
1742
1743
1743
1744
1745
1745
1746
1747
1747
1748
1749
1749
1750
1751
1751
1752
1753
1753
1754
1755
1755
1756
1757
1757
1758
1759
1759
1760
1761
1761
1762
1763
1763
1764
1765
1765
1766
1767
1767
1768
1769
1769
1770
1771
1771
1772
1773
1773
1774
1775
1775
1776
1777
1777
1778
1779
1779
1780
1781
1781
1782
1783
1783
1784
1785
1785
1786
1787
1787
1788
1789
1789
1790
1791
1791
1792
1793
1793
1794
1795
1795
1796
1797
1797
1798
1799
1799
1800
1801
1801
1802
1803
1803
1804
1805
1805
1806
1807
1807
1808
1809
1809
1810
1811
1811
1812
1813
1813
1814
1815
1815
1816
1817
1817
1818
1819
1819
1820
1821
1821
1822
1823
1823
1824
1825
1825
1826
1827
1827
1828
1829
1829
1830
1831
1831
1832
1833
1833
1834
1835
1835
1836
1837
1837
1838
1839
1839
1840
1841
1841
1842
1843
1843
1844
1845
1845
1846
1847
1847
1848
1849
1849
1850
1851
1851
1852
1853
1853
1854
1855
1855
1856
1857
1857
1858
1859
1859
1860
1861
1861
1862
1863
1863
1864
1865
1865
1866
1867
1867
1868
1869
1869
1870
1871
1871
1872
1873
1873
1874
1875
1875
1876
1877
1877
1878
1879
1879
1880
1881
1881
1882
1883
1883
1884
1885
1885
1886
1887
1887
1888
1889
1889
1890
1891
1891
1892
1893
1893
1894
1895
1895
1896
1897
1897
1898
1899
1899
1900
1901
1901
1902
1903
1903
1904
1905
1905
1906
1907
1907
1908
1909
1909
1910
1911
1911
1912
1913
1913
1914
1915
1915
1916
1917
1917
1918
1919
1919
1920
1921
1921
1922
1923
1923
1924
1925
1925
1926
1927
1927
1928
1929
1929
1930
1931
1931
1932
1933
1933
1934
1935
1935
1936
1937
1937
1938
1939
1939
1940
1941
1941
1942
1943
1943
1944
1945
1945
1946
1947
1947
1948
1949
1949
1950
1951
1951
1952
1953
1953
1954
1955
1955
1956
1957
1957
1958
1959
1959
1960
1961
1961
1962
1963
1963
1964
1965
1965
1966
1967
1967
1968
1969
1969
1970
1971
1971
1972
1973
1973
1974
1975
1975
1976
1977
1977
1978
1979
1979
1980
1981
1981
1982
1983
1983
1984
1985
1985
1986
1987
1987
1988
1989
1989
1990
1991
1991
1992
1993
1993
1994
1995
1995
1996
1997
1997
1998
1999
1999
2000
2001
2001
2002
2003
2003
2004
2005
2005
2006
2007
2007
2008
2009
2009
2010
2011
2011
2012
2013
2013
2014
2015
2015
2016
2017
2017
2018
2019
2019
2020
2021
2021
2022
2023
2023
2024
2025
2025
2026
2027
2027
2028
2029
2029
2030
2031
2031
2032
2033
2033
2034
2035
2035
2036
2037
2037
2038
2039
2039
2040
2041
2041
2042
2043
2043
2044
2045
2045
2046
2047
2047
20
```

```

102
103 end.           // Unit Calc

```

#### 4.1.8. Test program

```

1 PROGRAM CalcDemo;
2
3 USES Calc, Vector;
4
5 VAR a, b, dx          : REAL;
6     dummy              : REAL;
7     Formula            : Calc_String;
8     FormProg           : Calc_Prog;
9     x                  : Calc_VarTab;
10    f, xv, yv, x1, x2 : VectorTyp;
11    xmin, xmax, ymin, ymax : REAL;
12    n                  : WORD;
13    i                  : INTEGER;
14    c                  : STRING[1];
15
16 BEGIN
17     x := NewVarTab;
18     dummy := AddToVarTab(x, 'X');
19     CalcDecMod := TRUE;
20
21     REPEAT
22         WriteLn;
23         WriteLn;
24         WriteLn('Please enter function to be evaluated: ');
25         Write('f(x) = '); ReadLn(Formula); WriteLn;
26         CompileExpression(Formula, x, FormProg);
27
28     IF CalcResult
29     THEN
30         BEGIN
31             WriteLn('Expression ', Formula, '" compiled correctly...!');
32             WriteLn;
33             Write('Evaluate f(x) between a = '); Read(a);
34             Write('                                and b = '); Read(b);
35             Write('      with step width dx = '); ReadLn(dx);
36             WriteLn;
37             i := Round((abs(a-b)/dx));
38             FOR n := 1 TO i+1 DO
39                 BEGIN
40                     SetVectorElement(xv, n, a + dx);
41                     AssignVar(x, 'X', a);
42                     SetVectorElement(yv, n, CalcExpression(FormProg, x));
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
479
479
480
481
482
483
484
485
486
487
488
489
489
490
491
492
493
494
495
496
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
579
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
619
619
620
621
622
623
624
625
626
627
628
629
629
630
631
632
633
634
635
636
637
638
639
639
640
641
642
643
644
645
646
647
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
778
779
780
781
782
783
784
785
786
787
787
788
789
789
790
791
792
793
794
795
796
797
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
838
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
877
878
879
880
881
882
883
884
885
886
887
887
888
889
889
890
891
892
893
894
895
896
897
897
898
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
917
918
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
948
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
977
978
979
980
981
982
983
984
985
986
987
987
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1011
1012
1013
1014
1015
1016
1017
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1089
1090
1091
1092
1093
1094
1095
1095
1096
1097
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1117
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1166
1167
1168
1169
1170
1171
1172
1173
1173
1174
1175
1176
1177
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1189
1190
1191
1192
1193
1194
1195
1195
1196
1197
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1207
1207
1208
1209
1210
1211
1212
1213
1214
1215
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1285
1286
1287
1288
1289
1289
1290
1291
1292
1293
1294
1295
1295
1296
1297
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1307
1307
1308
1309
1310
1311
1312
1313
1314
1315
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1385
1386
1387
1388
1389
1389
1390
1391
1392
1393
1394
1395
1395
1396
1397
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1406
1407
1408
1409
1410
1411
1412
1413
1414
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1485
1486
1487
1488
1489
1489
1490
1491
1492
1493
1494
1495
1495
1496
1497
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1506
1507
1508
1509
1510
1511
1512
1513
1514
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1585
1586
1587
1588
1589
1589
1590
1591
1592
1593
1594
1595
1595
1596
1597
1598
1599
1599
1600
1601
1602
1603
1604
1605
1605
1606
1607
1608
1609
1610
1611
1612
1613
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1633
1634
1635
1636
1637
1638
1639
1639
1640
1641
1642
1643
1644
1645
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1663
1664
1665
1666
1667
1668
1669
1669
1670
1671
1672
1673
1674
1675
1675
1676
1677
1678
1679
1680
1681
1682
1683
1683
1684
1685
1686
1687
1688
1689
1689
1690
1691
1692
1693
1694
1694
1695
1696
1697
1698
1699
1699
1700
1701
1702
1703
1704
1704
1705
1706
1707
1708
1709
1710
1711
1712
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1722
1723
1724
1725
1726
1727
1728
1729
1729
1730
1731
1732
1733
1734
1735
1735
1736
1737
1738
1739
1740
1741
1742
1743
1743
1744
1745
1746
1747
1748
1749
1749
1750
1751
1752
1753
1754
1755
1755
1756
1757
1758
1759
1760
1761
1762
1763
1763
1764
1765
1766
1767
1768
1769
1769
1770
1771
1772
1773
1774
1775
1775
1776
1777
1778
1779
1780
1781
1782
1782
1783
1784
1785
1786
1787
1788
1788
1789
1790
1791
1792
1793
1794
1794
1795
1796
1797
1798
1799
1799
1800
1801
1802
1803
1803
1804
1805
1806
1807
1808
1809
1810
1811
1811
1812
1813
1814
1815
1816
1817
1818
1819
1819
1820
1821
1822
1823
1824
1825
1825
1826
1827
1828
1829
1830
1831
1832
1833
1833
1834
1835
1836
1837
1838
1839
1839
1840
1841
1842
1843
1844
1844
1845
1846
1847
1848
1849
1849
1850
1851
1852
1853
1854
1854
1855
1856
1857
1858
1859
1860
1861
1861
1862
1863
1864
1865
1866
1867
1868
1868
1869
1870
1871
1872
1873
1874
1874
1875
1876
1877
1878
1879
1880
1880
1881
1882
1883
1884
1885
1885
1886
1887
1888
1889
1889
1890
1891
1892
1893
1893
1894
1895
1896
1897
1898
1898
1899
1900
1901
1902
1902
1903
1904
1905
1906
1907
1908
1908
1909
1910
1911
1912
1913
1914
1914
1915
1916
1917
1918
1919
1920
1920
1921
1922
1923
1924
1925
1925
1926
1927
1928
1929
1930
1931
1931
1932
1933
1934
1935
1936
1936
1937
1938
1939
1940
1941
1942
1942
1943
1944
1945
1946
1947
1948
1948
1949
1950
1951
1952
1953
1954
1954
1955
1956
1957
1958
1959
1960
1960
1961
1962
1963
1964
1965
1966
1967
1967
1968
1969
1970
1971
1972
1973
1973
1974
1975
1976
1977
1978
1979
1979
1980
1981
1982
1983
1984
1984
1985
1986
1987
1988
1989
1989
1990
1991
1992
1993
1993
1994
1995
1996
1997
1998
1998
1999
2000
2001
2002
2002
2003
2004
2005
2006
2007
2007
2008
2009
2010
2011
2012
2013
2013
2014
2015
2016
2017
2018
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2048
2049
2050
2051
2052
2053
2054
2055
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2078
2079
2080
2081
2082
2083
2084
2085
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2148
2149
2150
2151
2152
2153
2154
2155
2
```

```

42           a := a + dx;
43           END;
44           KillExpression(FormProg);
45       END;
46   UNTIL Formula = '';
47   KillVarTab(x);
48   WriteLn('Demo finished...');
49 END.

```

## References

- [1] K. GIESELMANN, M. CEOL: CALC – ein mathematischer Compiler, *PASCAL Int.8* (1987), 52–60.
- [2] P. ENGELS: CALC, DISCUSS und die Turbo-4.0 Graphikbibliothek, *Toolbox:3* (1989), 55–61.

# 5. Vector arithmetic

## Abstract

Vectors and arrays are probably the most common data structures. However, in Pascal their use is not comfortable, as their size needs to be determined at compile-time. This makes it difficult to write reusable code. In WIRTH's Pascal definition [1] there were **conformant arrays** for this purpose, however, these were not implemented in most Pascal dialects, including Object Pascal.

One can define arrays large enough for all purposes, and then use only whatever part is actually required. However, this is memory-expensive. Under 16-bit operating systems like DOS the data segment was limited to 64 kB, equivalent to about 8000 elements. Even under modern 64 Bit operating systems and compilers, this problem isn't solved, because the stack (required for passing of variables to functions and procedures) still has this limit.

In addition, with this method the size of the array has to be passed with the data structure, a possible source of errors. Thus, it would be useful to have size information in the structure itself:

```
1 TYPE VectorStruc = RECORD
2   Columns : WORD;
3   Data : ARRAY [1..MaxVector] of float;
4 END;
```

and then use pointers to pass data to functions, in order to save stack space and – in 16 Bit systems – to have the data on the heap to avoid space problems as much as possible:

```
1 VectorTyp = ^VectorStruc;
```

This leads to the following interface

Listing 5.1: Interface of unit Vector

```
1 UNIT Vector;
2
3 INTERFACE
4
5 USES Math, mathfunc;
6
7 CONST
```

## 5. Vector arithmetic

```
8     MaxVector = 8000;           // PowerOfTwo[16] DIV SizeOf(float);
9     VectorError: BOOLEAN = FALSE; // toggle IF error occurs.
10
11
12 TYPE
13     VectorStruc = RECORD
14         Length: WORD;
15         Data: ARRAY [1..MaxVector] OF float;
16     END;
17     VectorTyp = ^VectorStruc;
18
19 PROCEDURE CreateVector(VAR Vec: VectorTyp; Length: WORD; Value: float);
20
21 PROCEDURE DestroyVector(VAR Vec: VectorTyp);
22
23 PROCEDURE ReadVector(VAR Vec: VectorTyp; MedStr: STRING);
24
25 PROCEDURE WriteVector(MedStr: STRING; CONST Vec: VectorTyp; ValidFigures:
26     BYTE);
27
28 FUNCTION VectorLength(CONST Vec: VectorTyp): WORD;
29
30 FUNCTION GetVectorElement(CONST Vec: VectorTyp; n: WORD): float;
31
32 PROCEDURE SetVectorElement(VAR Vec: VectorTyp; n: WORD; c: float);
33
34 PROCEDURE CopyVector(CONST Source: VectorTyp; VAR Dest: VectorTyp);
35
36 PROCEDURE LoadConstant(VAR Vec: VectorTyp; C: float);
37
38 PROCEDURE AddConstant(VAR A: VectorTyp; C: float);
39
40 PROCEDURE SubConstant(VAR A: VectorTyp; C: float);
41
42 PROCEDURE MulConstant(VAR A: VectorTyp; C: float);
43
44 PROCEDURE DivConstant(VAR A: VectorTyp; C: float);
45
46 PROCEDURE VectorAdd(CONST A, B: VectorTyp; VAR C: VectorTyp);
47
48 PROCEDURE VectorSub(CONST A, B: VectorTyp; VAR C: VectorTyp);
49
50 FUNCTION VectorInnerProduct(CONST A, B: VectorTyp): float;
51
52 FUNCTION TotalSum(CONST Vec: VectorTyp): float;
```

```

53 FUNCTION NeumaierSum(CONST A: VectorTyp): float;
54
55 FUNCTION TotalProduct(CONST Vec: VectorTyp): float;
56
57 FUNCTION ActualElements(CONST A: VectorTyp): WORD;
58
59 FUNCTION VectorSignum(Vec: VectorTyp): INTEGER;
60
61 FUNCTION FindLargest(Vec: VectorTyp): float;
62 { returns largest element of vector }
63
64 FUNCTION FindSmallest(Vec: VectorTyp): float;
65
66 FUNCTION FindLargestAbsolute(Vec: VectorTyp): float;
67
68 FUNCTION FindSmallestAbsolute(Vec: VectorTyp): float;
69
70 PROCEDURE Scale(VAR Vec: VectorTyp; min, max: float);
71
72 PROCEDURE centre(VAR Vec: VectorTyp);
73
74 FUNCTION VectorAngle(CONST A, B: VectorTyp): float;
75
76 { ----- Vector norms, normalisation ----- }
77
78 FUNCTION VectorEuklidianNorm(CONST Vec: VectorTyp): float;
79
80 FUNCTION VectorMaximumNorm(CONST Vec: VectorTyp): float;
81
82 FUNCTION VectorAbsoluteSumNorm(CONST Vec: VectorTyp): float;
83
84 PROCEDURE NormaliseVector(VAR Vec: VectorTyp; Norm: float);
85
86 { ----- Elemente sortieren ----- }
87
88 PROCEDURE ShellSort(VAR t: VectorTyp);
89
90 { ----- Distance ----- }
91
92 FUNCTION SquaredEuklidianDistance(CONST A, B: VectorTyp;
93
94 { ----- }
95
96 IMPLEMENTATION
97
98 VAR

```

## 5. Vector arithmetic

```
99     CH: CHAR;
```

The mechanism of error handling is the same as discussed for unit `MathFunc`.

### 5.1. Generation and destruction of vectors

This unit is written for data handling in sciences. By convention, we will use elements  $1 \dots n$  in this unit, rather than  $0 \dots n - 1$ , as some other programmers (who come from pure math) do.

Vectors contain `Length` times the data space required to store a float number plus the size of the word variable `Length`. As a matter of fact, 6 additional bytes are also required. This amount of memory needs to be reserved when a vector is created, and released again when it is destroyed. If not enough memory is available, the attempt is aborted and an error message is produced. The calling routine is informed about the error situation by the flag `VectorError`:

Listing 5.2: generation and destruction of vectors

```
1 PROCEDURE CreateVector(VAR Vec: VectorTyp; Length: WORD; Value: float);
2
3 VAR
4     i: WORD;
5
6 BEGIN
7     TRY
8         GetMem(Vec, Length * SizeOf(float) + SizeOf(WORD) + 6);
9     EXCEPT
10        CH := WriteErrorMessage(' Not enough memory to create vector');
11        VectorError := true;
12    EXIT;
13 END;
14 Vec^.Length := Length;
15 FOR i := 1 TO Length DO
16     Vec^.Data[i] := Value;
17 END;
18
19 PROCEDURE DestroyVector(VAR Vec: VectorTyp);
20
21 VAR
22     x: WORD;
23
24 BEGIN
25     x := Vec^.Length * SizeOf(float) + SizeOf(WORD) + 6;
26     FreeMem(Vec, x);
27 END;
28
29
```

```

30 PROCEDURE ReadVector(VAR Vec: VectorTyp; MedStr: STRING);
31
32 VAR
33   j, l: WORD;
34   Medium: TEXT;
35
36 BEGIN
37   Assign(Medium, MedStr);
38   Reset(Medium);
39   IF IOResult <> 0 THEN
40     BEGIN
41       CH := WriteErrorMessage(' File with vector not found');
42       VectorError := true;
43       EXIT;
44     END;
45   ReadLn(Medium, l);
46   FOR j := 1 TO l DO
47     BEGIN
48       IF EoF(Medium) THEN
49         BEGIN
50           CH := WriteErrorMessage(' Unknown file format');
51           VectorError := true;
52           EXIT;
53         END;
54       ReadLn(Medium, Vec^.Data[j]);
55       IF IOResult <> 0 THEN
56         BEGIN
57           CH := WriteErrorMessage(' Unknown file format');
58           VectorError := true;
59           EXIT;
60         END;
61     END;
62   Close(Medium);
63 END;
64
65
66 PROCEDURE WriteVector(MedStr: STRING; CONST Vec: VectorTyp; ValidFigures:
67   BYTE);
68
69 VAR
70   j: WORD;
71   Medium: TEXT;
72
73 BEGIN
74   Assign(Medium, MedStr);
75   Rewrite(Medium);

```

## 5. Vector arithmetic

```
75  Writeln(Medium, Vec^.Length);
76  FOR j := 1 TO Vec^.Length DO
77    Write(Medium, FloatStr(Vec^.Data[j], ValidFigures), ' ');
78  Close(Medium);
79 END;
```

It is not possible to copy these vectors by  $a = b$ , rather, the following procedure needs to be used:

```
1 PROCEDURE CopyVector(CONST Source: VectorTyp; VAR Dest: VectorTyp);
2
3 VAR
4   i, n: WORD;
5
6 BEGIN
7   n := VectorLength(Source);
8   TRY
9     GetMem(Dest, n * SizeOf(Float) + SizeOf(WORD) + 6);
10  EXCEPT
11    CH := WriteErrorMessage('Copy vector: Not enough memory to create
12      copy');
13    VectorError := true;
14    EXIT;
15  END;
16  CreateVector(Dest, n, 0.0);
17  FOR i := 1 TO n DO
18    SetVectorElement(Dest, i, GetVectorElement(Source, i));
```

## 5.2. Accesing and changing data in vectors

The following routines provide opaque access to the information in vectors, that is, they work irrespective on how the vector is implemented. Access from outside of this unit should always use these routines. To save memory and execution time, unchanged vectors are given the key word `const` in the definition of routines, this prevents copying of these structures (call by reference instead of call by value). In older Pascal dialects use `var` for the same purpose:

Listing 5.3: Accesing and changing data in vectors

```
1 FUNCTION VectorLength(CONST Vec: VectorTyp): WORD;
2
3 BEGIN
4   Result := Vec^.Length;
5 END;
```

```

8  FUNCTION GetVectorElement(CONST Vec: VectorTyp; n: WORD): float;
9
10 VAR
11   s1, s2: STRING;
12
13 BEGIN
14   IF ((n <= VectorLength(Vec)) AND (n > 0))
15   THEN
16     Result := Vec^.Data[n]
17   ELSE
18     BEGIN
19       Str(n: 4, s1);
20       Str(VectorLength(Vec): 4, s2);
21       CH := WriteErrorMessage(' Attempt to read non-existend vector
22         element No ' +
23         s1 + ' of ' + s2);
24     END;
25
26
27 PROCEDURE SetVectorElement(VAR Vec: VectorTyp; n: WORD; C: float);
28
29 BEGIN
30   IF ((n <= VectorLength(Vec)) AND (n > 0)) THEN
31     Vec^.Data[n] := C
32   ELSE
33     BEGIN
34       CH := WriteErrorMessage(' Attempt to write to non-existend vector
35         element');
36       VectorError := true;
37     END;
38 END;

```

## 5.3. Calculation with vectors

### 5.3.1. Calculations using one vector and a constant number as argument

Listing 5.4: Calculations using one vector and a constant number as argument

```

1  PROCEDURE LoadConstant(VAR Vec: VectorTyp; C: float);
2
3  VAR
4    i, n: WORD;
5

```

## 5. Vector arithmetic

```
6  BEGIN
7      n := VectorLength(Vec);
8      IF (n = 0) THEN EXIT;
9      FOR i := 1 TO n DO
10         SetVectorElement(Vec, i, c);
11     END;
12
13
14 PROCEDURE AddConstant(VAR A: VectorTyp; C: float);
15
16 VAR
17   i, n: WORD;
18
19 BEGIN
20   n := VectorLength(A);
21   IF (n = 0) THEN EXIT;
22   FOR i := 1 TO n DO
23     SetVectorElement(A, i, GetVectorElement(A, i) + c);
24   END;
25
26
27 PROCEDURE SubConstant(VAR A: VectorTyp; C: float);
28
29 VAR
30   i, n: WORD;
31
32 BEGIN
33   n := VectorLength(A);
34   IF (n = 0) THEN EXIT;
35   FOR i := 1 TO n DO
36     SetVectorElement(A, i, GetVectorElement(A, i) - c);
37   END;
38
39
40
41 PROCEDURE MulConstant(VAR A: VectorTyp; C: float);
42
43 VAR
44   i, n: WORD;
45
46 BEGIN
47   n := VectorLength(A);
48   IF (n = 0) THEN EXIT;
49   FOR i := 1 TO n DO
50     SetVectorElement(A, i, GetVectorElement(A, i) * c);
51
```

```

52 END;
53
54
55 PROCEDURE DivConstant(VAR A: VectorTyp; C: float);
56
57 VAR
58   i, n: WORD;
59
60 BEGIN
61   n := VectorLength(A);
62   IF (n = 0) THEN EXIT;
63   FOR i := 1 TO n DO
64     SetVectorElement(A, i, GetVectorElement(A, i) / c);
65 END;

```

### 5.3.2. Routines that take two vectors and calculate a third

#### Addition and subtraction of vectors

```

1  PROCEDURE VectorAdd(CONST A, B: VectorTyp; VAR C: VectorTyp);
2
3  VAR
4    i, n: WORD;
5
6  BEGIN
7    n := VectorLength(A);
8    IF (n = 0) OR (n <> VectorLength(B))
9      THEN
10        BEGIN
11          CH := WriteErrorMessage(' Addition of vectors: vectors of different
12            lengths');
13          VectorError := true;
14          EXIT;
15        END;
16        CreateVector(C, n, 0.0);
17        FOR i := 1 TO n DO
18          SetVectorElement(C, i, GetVectorElement(A, i) + GetVectorElement(B, i));
19
20
21 PROCEDURE VectorSub(CONST A, B: VectorTyp; VAR C: VectorTyp);
22
23 VAR
24   i, n: WORD;
25
26 BEGIN

```

## 5. Vector arithmetic

```

27   n := VectorLength(A);
28   IF (n = 0) OR (n <> VectorLength(B))
29     THEN
30       BEGIN
31         CH := WriteErrorMessage(' Subtraction of vectors: vectors of
32           different lengths');
33         VectorError := true;
34         EXIT;
35       END;
36       CreateVector(C, n, 0.0);
37       FOR i := 1 TO n DO
38         SetVectorElement(C, i, GetVectorElement(A, i) - GetVectorElement(B, i));
39     END;

```

### 5.3.3. Functions that take a Vector and produce a number

#### Sum over all vector elements

The sum of all elements of  $\mathbf{x}$  ( $S = \sum_{i=1}^n x_i$ ) is used to calculate the arithmetic mean of the elements  $\bar{x} = S/n$ , that is:

Listing 5.5: Grand total of vector elements

```

1  FUNCTION TotalSum(CONST Vec: VectorTyp): float;
2
3  VAR
4    i, n: WORD;
5    x: float;
6
7  BEGIN
8    n := VectorLength(Vec);
9    IF (n = 0)
10      THEN
11        BEGIN
12          TotalSum := 0;
13          EXIT;
14        END;
15    x := 0;
16    FOR i := 1 TO n DO
17      IF IsNaN(GetVectorElement(Vec, i))
18        THEN
19          ELSE x := x + GetVectorElement(Vec, i);
20    Result := x;
21  END;

```

The problem with this naïve implementation is that with floating point arithmetic there may be a catastrophic accumulation of error when the summands have very different orders of magnitude. For example, the sum of  $(1.0, +1 \times 10^{100}, 1.0, -1 \times 10^{100})$

obviously should be 2.0, but above routine will yield 0.0 when performed in IEEE double precision. NEUMAIER's compensated summation algorithm calculates and sums the rounding error of each step, the sum is added to the final result. This reduces the relative error from  $\mathbf{O}(n\epsilon)$  to  $\mathbf{O}(\epsilon)$ , unless  $n$  becomes significant compared to  $1/\epsilon$  ( $1 \times 10^{16}$  in IEEE double precision arithmetic).

Listing 5.6: Compensated summation

```

1  FUNCTION NeumaierSum(CONST A: VectorTyp): float;
2
3  VAR
4      Sum, c, t, x: float;
5      i, j: WORD;
6
7  BEGIN
8      IF (VectorLength(A) = 0)
9          THEN
10             BEGIN
11                 NeumaierSum := 0;
12                 EXIT;
13             END;
14     i := 1;
15     REPEAT      // search FOR the first non-NaN element
16         Sum := GetVectorElement(A, i);
17         INC(i);
18     UNTIL NOT (IsNaN(Sum));
19     c := 0.0;
20     FOR j := i TO VectorLength(A) DO
21         IF IsNaN(GetVectorElement(A, j))
22             THEN
23             ELSE
24                 BEGIN
25                     x := GetVectorElement(A, j);
26                     t := Sum + x;
27                     IF Abs(Sum) >= Abs(x)
28                         THEN c := c + ((Sum - t) + x)    // LOW-order digits OF A(j) are
29                                         lost
30                     ELSE c := c + ((x - t) + Sum);   // LOW-order digits OF Sum are
31                                         lost
32                     Sum := t;
33                 END;
34     Result := Sum + c;
35 END;
```

### Product over all vector elements

Similarly, the overall product  $P = \prod_{i=1}^n x_i$  is used for the harmonic mean  $\tilde{x} = \sqrt[n]{P}$ :

## 5. Vector arithmetic

Listing 5.7: Product of all vector elements

```

1 FUNCTION TotalProduct(CONST Vec: VectorTyp): float;
2
3 VAR
4   i, n: WORD;
5   x: float;
6
7 BEGIN
8   n := VectorLength(Vec);
9   IF (n = 0)
10  THEN
11    BEGIN
12      TotalProduct := 0;
13      EXIT;
14    END;
15   x := 0;
16   FOR i := 1 TO n DO
17     IF IsNaN(Vec^.Data[i])
18     THEN
19     ELSE x := x + Ln(GetVectorElement(Vec, i));
20   TotalProduct := Exp(x);
21 END;

```

For vectors with many large elements or very large length, it can be useful to calculate the total product from the sum of the logarithms of vector elements to avoid floating point overflow. NEUMAIER correction is not required in such cases as the logarithms will be of similar magnitude.

### 5.3.4. Properties of a vector

A vector is positive, if  $x_i \geq 0$ , it is strictly positive if  $x_i > 0 \forall i \in 1 \dots n$ . **VectorSignum** returns 1 for strictly positive, 0 for positive and -1 for vectors containing negative elements.

Listing 5.8: Signum of vector

```

1 FUNCTION VectorSignum(Vec: VectorTyp): INTEGER;
2
3 VAR
4   x: float;
5   i: WORD;
6
7 BEGIN
8   VectorSignum := 1;
9   FOR i := 1 TO VectorLength(Vec) DO
10  BEGIN
11    x := GetVectorElement(Vec, i);

```

```

12      IF IsNaN(x)
13      THEN
14      ELSE
15          CASE signum(x) OF
16              -1: BEGIN
17                  VectorSignum := -1; // one negative element IS enough
18                  EXIT;
19              END;
20              0: BEGIN
21                  Result := 0; // continue IN CASE a negative element comes
22                  later
23                  END;
24                  1: BEGIN
25                      // do nothing
26                      END;
27          END; { case }
28      END; { for }
29  END;

```

Listing 5.9: Smallest and largest absolute value

```

1 FUNCTION FindLargestAbsolute(Vec: VectorTyp): float;
2
3 VAR
4     i: WORD;
5     a: float;
6
7 BEGIN
8     a := GetVectorElement(Vec, 1);
9     FOR i := 2 TO VectorLength(Vec) DO
10         IF Abs(a) < Abs(GetVectorElement(Vec, i))
11             THEN a := GetVectorElement(Vec, i);
12     Result := a;
13 END;
14
15
16 FUNCTION FindSmallestAbsolute(Vec: VectorTyp): float;
17
18 VAR
19     i: WORD;
20     a: float;
21
22 BEGIN
23     a := GetVectorElement(Vec, 1);
24     FOR i := 2 TO VectorLength(Vec) DO
25         IF Abs(a) > Abs(GetVectorElement(Vec, i))
26             THEN a := GetVectorElement(Vec, i);

```

## 5. Vector arithmetic

```

27   Result := a;
28 END;
29
30
31 FUNCTION FindLargest(Vec: VectorTyp): float;
32
33 VAR
34   i: WORD;
35   a: float;
36
37 BEGIN
38   a := GetVectorElement(Vec, 1);
39   FOR i := 2 TO VectorLength(Vec) DO
40     IF a < GetVectorElement(Vec, i)
41       THEN a := GetVectorElement(Vec, i);
42   Result := a;
43 END;
44
45
46 FUNCTION FindSmallest(Vec: VectorTyp): float;
47
48 VAR
49   i: WORD;
50   a: float;
51
52 BEGIN
53   a := GetVectorElement(Vec, 1);
54   FOR i := 2 TO VectorLength(Vec) DO
55     IF a > GetVectorElement(Vec, i)
56       THEN a := GetVectorElement(Vec, i);
57   Result := a;
58 END;

```

### 5.3.5. Vector norms

#### EUKLIDian norm of a vector

The length or EUKLIDIAN norm of a vector is  $\ell_2 = \|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^n x_i^2}$ :

Listing 5.10: EUKLIDian norm

```

1 FUNCTION VectorEuklidianNorm(CONST Vec: VectorTyp): float;
2
3 VAR
4   i, n: WORD;
5   x: VectorTyp;
6

```

```

7  BEGIN
8    n := VectorLength(Vec);
9    IF (n = 0)
10   THEN
11     BEGIN
12       Result := 0;
13       EXIT;
14     END;
15   CreateVector(x, n, 0.0);
16   IF VectorError THEN EXIT;
17   FOR i := 1 TO Vec^.Length DO
18     SetVectorElement(x, i, Sqr(GetVectorElement(Vec, i)));
19   VectorEuklidianNorm := Sqrt(NeumaierSum(x)); // avoid rounding error
20   during summation
21   DestroyVector(x);
22 END;

```

### TSCHEBYSCHEW-norm

The maximum (TSCHEBYSCHEW-, chess board-) norm  $\|\mathbf{x}\|_\infty = \max_{i=1\dots n}(|x_i|)$  is the absolute largest element of  $\mathbf{x}$ :

Listing 5.11: Maximum norm

```

1  FUNCTION VectorMaximumNorm(CONST Vec: VectorTyp): float;
2
3  VAR
4    Max, Element: float;
5    i: WORD;
6
7  BEGIN
8    IF (VectorLength(Vec) = 0)
9    THEN
10   BEGIN
11     Result := 0;
12     EXIT;
13   END;
14   Max := -1e300;
15   FOR i := 1 TO VectorLength(Vec) DO
16     BEGIN
17       Element := Abs(GetVectorElement(Vec, i));
18       IF Element > Max THEN Max := Element;
19     END;
20   Result := Max;
21 END;

```

## 5. Vector arithmetic

### Manhattan-norm

In the absolute sum (taxicab or Manhattan-) norm  $\ell_1 = \|\mathbf{x}\|_1 = \sum_{i=1}^n |\mathbf{x}_i|$  the distance between two points is the sum of the absolute differences of their Cartesian coordinates. The name alludes to the grid layout of most streets on the island of Manhattan. A taxicab driving between two intersections drives the distance corresponding to the length of the  $\ell_1$ -norm (see fig. 5.1).

Listing 5.12: Absolute sum norm

```

1  FUNCTION VectorAbsoluteSumNorm(CONST Vec: VectorTyp): float;
2
3  VAR
4      i: WORD;
5      Sum: float;
6
7  BEGIN
8      IF (VectorLength(Vec) = 0)
9          THEN
10         BEGIN
11             Result := 0;
12             EXIT;
13         END;
14     Sum := 0;
15     FOR i := 1 TO VectorLength(Vec) DO
16         Sum := Sum + Abs(GetVectorElement(Vec, i));
17     Result := Sum;
18 END;
```

### Vector normalisation

These are special cases of  $p$ -norms,  $\|\mathbf{x}\|_p = \sqrt[p]{\sum_{i=1}^n |\mathbf{x}_i|^p}$  for  $p = 1, 2, \infty$ , respectively. Vectors can be normalised by dividing all elements by a norm:

Listing 5.13: Normalisation of vector

```

1  PROCEDURE NormaliseVector(VAR Vec: VectorTyp; Norm: float);
2
3  BEGIN
4      DivConstant(Vec, Norm);
5  END;
```

### Inner (dot) product of a vector

The inner product of two vectors  $\mathbf{a}, \mathbf{b}$  is a real number  $\mathbf{a} \odot \mathbf{b} = \sum_{i=1}^n (\mathbf{a}_i \mathbf{b}_i)$ , the signed area of the parallelogram formed by the vectors. The sign depends on the angle between the vectors (clock- or counter-clockwise). We use NEUMAIER-addition to prevent the accumulation of error:

# Manhattan distance

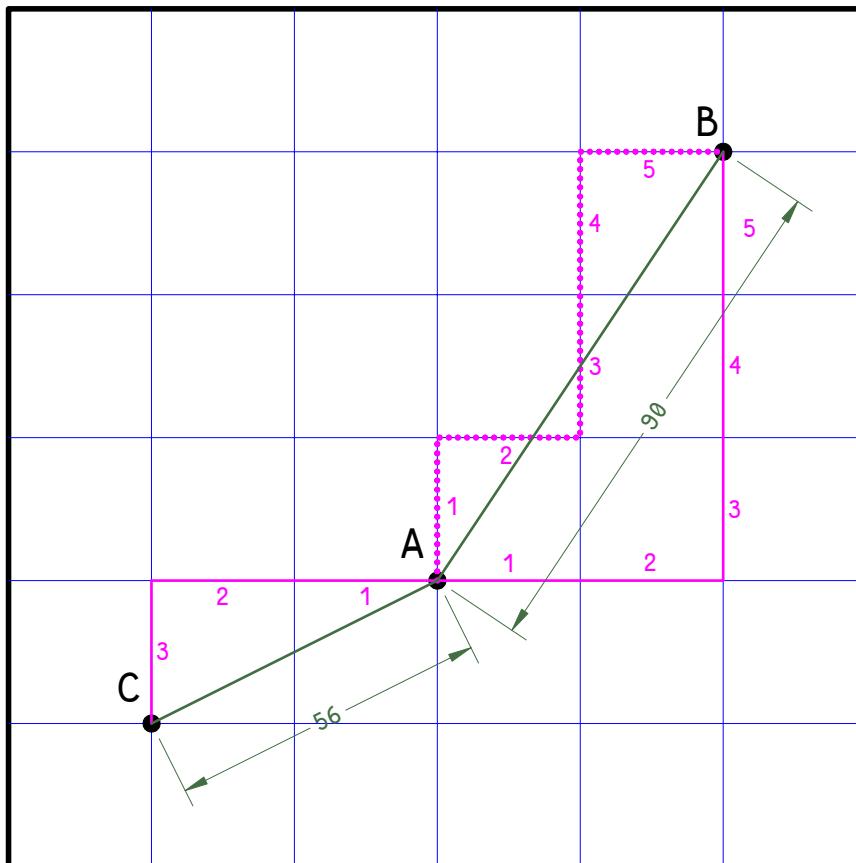


Figure 5.1.: Manhattan- (aka taxicab-) norm. The streets of Manhattan form a grid, and the distance between two points (say, A and B) is the number of blocks a taxi needs to drive (*purple*). As long as we don't take detours, the number of blocks is independent of the route (*full and dotted line*). By comparison, the EUKLIDIAN norm (*green*) measures the distance "straight as the crow flies".

## 5. Vector arithmetic

Listing 5.14: Inner product of vectors

```

1  FUNCTION VectorInnerProduct(CONST A, B: VectorTyp): float;
2
3  VAR
4      i, n1, n2: WORD;
5      x: VectorTyp;
6
7  BEGIN
8      n1 := VectorLength(A);
9      n2 := VectorLength(B);
10     IF (n1 <> n2)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage('Vector error: Inner Product of vectors of
14                     unequal length');
15                 VectorError := true;
16                 EXIT;
17             END;
18     IF (n1 = 0)
19         THEN
20             BEGIN
21                 Result := 0;
22                 EXIT;
23             END;
24     CreateVector(x, n1, 0.0);
25     FOR i := 1 TO n1 DO
26         SetVectorElement(x, i, GetVectorElement(A, i) * GetVectorElement(B, i));
27     Result := NeumaierSum(x);
28     DestroyVector(x);
29 END;
```

### Angle between two vectors

The angle between two vectors is then calculated by  $\alpha = \frac{ab}{|a||b|}$ :

Listing 5.15: Angle between vectors

```

1  FUNCTION VectorAngle(CONST A, B: VectorTyp): float;
2
3  VAR
4      n1, n2: WORD;
5
6  BEGIN
7      n1 := VectorLength(A);
8      n2 := VectorLength(B);
9      IF (n1 <> n2)
10         THEN
```

```

11   BEGIN
12     CH := WriteErrorMessage('Vector-Error: Angle between vectors of
13       unequal dimension');
14     VectorError := TRUE;
15     EXIT;
16   END;
17   IF (n1 = 0)
18     THEN Result := 0
19   ELSE Result := ArcCos(VectorInnerProduct(A, B) /
20     (VectorEuklidianNorm(A) * VectorEuklidianNorm(B)));
21 END;

```

TotalSum and TotalProduct calculate the sum or product over all elements of the vector. Any `Nan` is handled by ignoring that value. Therefore, to calculate the arithmetic or harmonic mean, one cannot simply use `VectorLength()` to determine the number of actual elements:

Listing 5.16: Number of non-NaN elements

```

1 FUNCTION ActualElements(CONST A: VectorTyp): WORD;
2
3 VAR
4   i, j: WORD;
5
6 BEGIN
7   j := 0;
8   FOR i := 1 TO VectorLength(A) DO
9     IF IsNaN(GetVectorElement(A, i))
10    THEN
11      ELSE INC(j);
12   Result := j;
13 END;

```

### 5.3.6. Scaling, centering and normalisation of vectors

To scale a vector to the range [0...1] we can use

$$y = \frac{x - \min(x)}{\max(x) - \min(x)} \quad (5.1)$$

If the data contain outliers, it can be useful to use, say, the 5th and 95th percentile instead of the minimum and maximum. For this reason, we give `min` and `max` as parameters of the function rather than determining it within:

Listing 5.17: Scaling a vector

```

1 PROCEDURE Scale(VAR Vec: VectorTyp; min, max: float);
2
3 VAR

```

## 5. Vector arithmetic

```

4   i: WORD;
5   Range: float;
6
7 BEGIN
8   Range := max - min;
9   FOR i := 1 TO VectorLength(Vec) DO
10    SetVectorElement(Vec, i, (GetVectorElement(Vec, i) - min) / Range);
11 END;

```

A data vector is centered by subtracting the arithmetic mean, so that the new mean becomes zero:

$$\mathbf{y} = \mathbf{x} - \bar{\mathbf{x}} \quad (5.2)$$

Listing 5.18: Centering a vector

```

1 PROCEDURE Centre(VAR Vec: VectorTyp);
2
3 VAR
4   Mean: float;
5   i: WORD;
6
7 BEGIN
8   Mean := NeumaierSum(Vec) / ActualElements(Vec);
9   FOR i := 1 TO VectorLength(Vec) DO
10    SetVectorElement(Vec, i, GetVectorElement(Vec, i) - Mean);
11 END;

```

### 5.3.7. Vector distance

The distance between two row vectors (observations)  $\vec{a}$ ,  $\vec{b}$  is the sum of the squared differences  $\sum_{i=1}^n (a_i - b_i)^2$ :

```

1 FUNCTION SquaredEuklidianDistance(CONST A, B: VectorTyp;
2   IgnoreFirst: BOOLEAN): float;
3
4 VAR
5   p, i, start: WORD;
6   c: CHAR;
7   ai, bi, Sum: float;
8
9 BEGIN
10  p := VectorLength(A);
11  IF VectorLength(B) <> p
12  THEN
13    BEGIN
14      VectorError := TRUE;
15      c := WriteErrorMessage('Euklidian distance of two vectors: vectors
        have unequal length');

```

```

16      EXIT;
17  END;
18 Sum := 0;
19 IF IgnoreFirst
20 THEN Start := 2 // first column study number
21 ELSE Start := 1; // first column data
22 FOR i := Start TO p DO
23 BEGIN
24     ai := GetVectorElement(A, i);
25     bi := GetVectorElement(B, i);
26     IF IsNaN(ai) OR IsNaN(bi)
27     THEN
28         BEGIN
29             VectorError := TRUE;
30             c := WriteErrorMessage(
31                 'Euklidian distance of two vectors: vectors contain NaN');
32             EXIT;
33         END;
34     Sum := Sum + Sqr(ai - bi);
35 END;
36 Result := Sum;
37 END;

```

## 5.4. Sorting a vector

There are several different sorting algorithms, with different efficiencies with respect to memory requirement and computation speed. For medium numbers of elements (several hundred to a few thousand), the SHELL sort is often the best compromise, even if values are partially sorted already. This routine, modified from [2], can handle `NaN` elements and sorts them as highest values:

Listing 5.19: Shell sort

```

1 PROCEDURE ShellSort(VAR t: VectorTyp);
2
3 LABEL
4   10;
5
6 VAR
7   i, j, k, l, m, nn, NaNs, n, LdN: INTEGER;
8   tmp, s: float;
9
10 BEGIN
11   NaNs := 0;
12   s := -1e300;
13   n := VectorLength(t);

```

## 5. Vector arithmetic

```

14   FOR i := 1 TO n DO
15     BEGIN
16       IF IsNaN(GetVectorElement(t, i)) // check FOR NaN data
17         THEN INC(NaNs)
18       ELSE IF (GetVectorElement(t, i)) > s
19         THEN s := GetVectorElement(t, i); // AND find largest
20           element OF data vector
21     END;
22   s := 10 * s;
23   IF (NaNs > 0)
24     THEN
25       FOR i := 1 TO n DO
26         IF IsNaN(GetVectorElement(t, i))
27           THEN SetVectorElement(t, i, s);
28 // replace all NaN WITH very large number so they Move TO END OF vector
29   LdN := Trunc(Ln(n) / Const_ln2);
30   m := n;
31   FOR nn := 1 TO LdN DO
32     BEGIN
33       m := m DIV 2;
34       k := n - m;
35       FOR j := 1 TO k DO
36         BEGIN
37           i := j;
38           l := i + m;
39           IF (GetVectorElement(t, l) < GetVectorElement(t, i))
40             THEN
41               BEGIN
42                 tmp := GetVectorElement(t, i);
43                 SetVectorElement(t, i, GetVectorElement(t, l));
44                 SetVectorElement(t, l, tmp);
45                 i := i - m;
46                 IF i >= 1 THEN GOTO 10;
47               END;
48           END;
49       IF (NaNs > 0) THEN
50         FOR i := Succ(n - NaNs) TO n DO // change the top NaNs elements back
51           TO NaN
52           SetVectorElement(t, i, NaN);
53     END;
54
55 END.
```

## References

- [1] K. JENSEN, N. WIRTH: *PASCAL - User Manual and Report* 1st ed. Lecture Notes in Computer Science Berlin Heidelberg: Springer-Verlag, 1974 ISBN: 978-3-662-21554-8.
- [2] J.-P. MOREAU: *Sorting an array with the Shell method* Web-site, accessed 2019-03-30 2013 URL: [http://jean-pierre.moreau.pagesperso-orange.fr/Pascal/sort2\\_pas.txt](http://jean-pierre.moreau.pagesperso-orange.fr/Pascal/sort2_pas.txt).



# 6. Matrix calculations

## Abstract

Matrices are vectors that have vectors as members. They are used to assign a vector of measured values (columns) to a vector of test subjects (rows). Thus, matrices are used as data structures in multivariate statistics.

The routines in this unit are based on [1–7]. For the implementation of two-dimensional arrays we use the same method as for vectors. We could use an `array[1..MaxN, 1..MaxP]`, however, it is more memory efficient to use a “flat”, one-dimensional array. The conversion of a two-dimensional position  $i, j$  into a one-dimensional is done in `SetVectorElement` and `GetVectorElement`.

```
1 type MatrixStruc = record
2   Columns, Rows: word;
3   Data: array[1..MaxVector] of float;
4 end
5 MatrixTyp = ^MatrixStruc;
```

The method of error handling is the same too, except that we use the typed constant `MatrixError` as error flag. Therefore, the interface is

Listing 6.1: Interface of unit Matrix

```
1 unit Matrix;
2
3 INTERFACE
4
5 USES Math, MathFunc, Vector;
6
7 CONST
8   MatrixError: BOOLEAN = FALSE; { toggle for error condition }
9
10 TYPE
11   MatrixStruc = RECORD
12     Columns, Rows: WORD;
13     Data: ARRAY[1..MaxVector] OF float;
14   END;
15   MatrixTyp = ^MatrixStruc;
16
17 FUNCTION WriteErrorMessage(TEXT: STRING): CHAR;
18
```

## 6. Matrix calculations

```
19 PROCEDURE CreateMatrix(VAR Mat: MatrixTyp; Rows, Columns: WORD; Value:
20   float);
21
22 PROCEDURE DestroyMatrix(VAR Mat: MatrixTyp);
23
24 PROCEDURE ReadMatrix(MedStr: STRING; VAR A: MatrixTyp);
25
26 PROCEDURE WriteMatrix(MedStr: STRING; CONST A: MatrixTyp; ValidFigures:
27   BYTE);
28
29 FUNCTION GetMatrixElement(CONST A: MatrixTyp; Row, Column: WORD): float;
30
31 PROCEDURE SetMatrixElement(VAR A: MatrixTyp; Row, Column: WORD; Value:
32   float);
33
34 PROCEDURE CreateIdentityMatrix(VAR A: MatrixTyp; n: WORD);
35
36 PROCEDURE CreateNullMatrix(VAR A: MatrixTyp; n: WORD);
37
38 PROCEDURE CreateHilbertMatrix(VAR H: MatrixTyp; n: WORD);
39
40 FUNCTION MatrixRows(CONST A: MatrixTyp): WORD;
41 { *****
42 }
43
44 PROCEDURE GetRow(CONST A: MatrixTyp; Z: WORD; VAR Vek: VectorTyp);
45
46 PROCEDURE GetColumn(CONST A: MatrixTyp; S: WORD; VAR Vek: VectorTyp);
47
48 PROCEDURE ExchangeColumns(VAR A: MatrixTyp; Column1, Column2: WORD);
49
50 PROCEDURE ExchangeRows(VAR A: MatrixTyp; Row1, Row2: WORD);
51
52 PROCEDURE SetRow(VAR A: MatrixTyp; CONST Vek: VectorTyp; Z: WORD);
53
54 PROCEDURE SetColumn(VAR A: MatrixTyp; CONST Vek: VectorTyp; S: WORD);
55
56 PROCEDURE CopyMatrix(CONST Source: MatrixTyp; VAR Dest: MatrixTyp);
57 { *****
58 }
59 PROCEDURE MatrixAdd(CONST A, B: MatrixTyp; VAR Res: MatrixTyp);
```

```

60
61 PROCEDURE SkalarMultiplikation(VAR A: MatrixTyp; x: float);
62
63 PROCEDURE MatrixInnerProduct(CONST A, B: MatrixTyp; VAR C: MatrixTyp);
64
65 PROCEDURE HadamardSchurProduct(CONST A, B: MatrixTyp; VAR C: MatrixTyp);
66
67 PROCEDURE MatrixDivision(CONST A, B: MatrixTyp; VAR C: MatrixTyp);
68
69 PROCEDURE CenterData(VAR A: MatrixTyp);
70
71 PROCEDURE ChangeMatrixNorm(VAR A: MatrixTyp; Norm: float);
72
73 FUNCTION Determinante(CONST A: MatrixTyp): float;
74
75 FUNCTION MatrixTrace(CONST A: MatrixTyp): float;
76
77 FUNCTION FrobeniusNorm(CONST A: MatrixTyp): float;
78
79 FUNCTION FrobeniusSkalarProduct(CONST A, B: MatrixTyp): float;
80
81 PROCEDURE ERoMultAdd(VAR A: MatrixTyp; Faktor: float; Row1, Row2: WORD);
82
83 PROCEDURE InverseMatrix(VAR A: MatrixTyp);
84
85 PROCEDURE MatrixTranspose(CONST A: MatrixTyp; VAR B: MatrixTyp);
86
87 PROCEDURE AntiSym(CONST A: MatrixTyp; VAR Symmetric, Antisymmetric:
88   MatrixTyp);
89
90 PROCEDURE Diag(VAR Matrix: MatrixTyp);
91
92 PROCEDURE LeadingPrincipleMinors(VAR A: MatrixTyp; VAR V: VectorTyp);
93
94 PROCEDURE NegativeMatrix(VAR A: MatrixTyp);
95 { ***** spezielle Matrizen *****
96 }
97
98 FUNCTION MatrixSquare(CONST A: MatrixTyp): BOOLEAN;
99
100 FUNCTION MatrixSymmetric(VAR A: MatrixTyp): BOOLEAN;
101
102 FUNCTION MatrixLeftTrapezoid(VAR A: MatrixTyp): BOOLEAN;
103 FUNCTION MatrixRightTrapezoid(VAR A: MatrixTyp): BOOLEAN;

```

## 6. Matrix calculations

```
104
105 FUNCTION MatrixDiagonal(VAR A: MatrixTyp): BOOLEAN;
106
107 FUNCTION MatrixUpperTriangular(VAR A: MatrixTyp): BOOLEAN;
108
109 FUNCTION MatrixLowerTriangular(VAR A: MatrixTyp): BOOLEAN;
110
111 FUNCTION MatrixUpperHessenberg(VAR A: MatrixTyp): BOOLEAN;
112
113 FUNCTION MatrixLowerHessenberg(VAR A: MatrixTyp): BOOLEAN;
114
115 FUNCTION MatrixTridiagonal(VAR A: MatrixTyp): BOOLEAN;
116
117 FUNCTION MatrixPositivDefinite(CONST A: MatrixTyp): BOOLEAN;
118
119 FUNCTION NullMatrix(VAR A: MatrixTyp): BOOLEAN;
120
121 { **** Vector/Matrix ****
122 }
123
124 PROCEDURE DyadicVectorProduct(CONST A, B: VectorTyp; VAR C: MatrixTyp);
125
126 PROCEDURE MultMatrixVector(CONST Mat: MatrixTyp; CONST Vek: VectorTyp;
127   VAR Result: VectorTyp);
128
129 PROCEDURE CangeVectorToMatrix(CONST Vek: VectorTyp; VAR Mat: MatrixTyp);
130
131 { **** sorting **** }
132
133 PROCEDURE ShellSortMatrix(VAR t: MatrixTyp; Column: WORD);
134   { sorts the rows of a matrix by column "Column" }
135
136
137 IMPLEMENTATION
138
139 VAR
  CH: CHAR;
```

### 6.1. Generation and destruction of matrices

Listing 6.2: Create a new matrix

```
1 PROCEDURE CreateMatrix(VAR Mat: MatrixTyp; Rows, Columns: WORD; Value:
2   float);
```

```

3  VAR
4    i: WORD;
5    x: longword;
6
7  BEGIN
8    x := Rows * Columns * SizeOf(float) + 2 * SizeOf(WORD) + 4;
9    TRY
10      GetMem(Mat, x);
11    except
12      CH := WriteErrorMessage(' Not enough memory to create matrix');
13      MatrixError := true;
14      EXIT;
15    END;
16    Mat^.Columns := Columns;
17    Mat^.Rows := Rows;
18    FOR i := 1 TO (Rows * Columns) DO
19      Mat^.Data[i] := Value;
20  END;

```

Listing 6.3: Remove a matrix and free its memory

```

1  PROCEDURE DestroyMatrix(VAR Mat: MatrixTyp);
2
3  VAR
4    x: longword;
5
6  BEGIN
7    x := MatrixRows(Mat) * MatrixColumns(Mat) * SizeOf(float) + 2 *
8      SizeOf(WORD) + 4;
9    FreeMem(Mat, x);
9  END;

```

Matrices can be read from, and written to, files. The file name is in `MedStr`. Note that '`CON`' and '`PRN`' write to the console or the printer, respectively.

Listing 6.4: Read a matrix from file

```

1  PROCEDURE ReadMatrix(MedStr: STRING; VAR A: MatrixTyp);
2
3  VAR
4    i, j, n, p: WORD;
5    x: float;
6    Medium: TEXT;
7
8  BEGIN
9    Assign(Medium, MedStr);
10   Reset(Medium);
11   IF IOResult <> 0
12     THEN

```

## 6. Matrix calculations

```
13      BEGIN
14          CH := WriteErrorMessage(' Reading a matrix: File not found');
15          MatrixError := true;
16          EXIT;
17      END;
18      ReadLn(Medium, n);
19      ReadLn(Medium, p);
20      IF ((n * p > MaxVector))
21      THEN
22          BEGIN
23              CH := WriteErrorMessage(' Reading a matrix: Matrix too big');
24              EXIT;
25          END;
26      CreateMatrix(A, n, p, 0.0);
27      FOR i := 1 TO n DO
28          BEGIN
29              IF EoF(Medium)
30              THEN
31                  BEGIN
32                      CH := WriteErrorMessage(' Reading a matrix: Unknown file
33                          format');
34                      MatrixError := true;
35                      EXIT;
36                  END;
37              FOR j := 1 TO p DO
38                  BEGIN
39                      IF EoLn(Medium)
40                      THEN
41                          BEGIN
42                              CH := WriteErrorMessage(' Reading a matrix: Unknown file
43                                  format');
44                              MatrixError := true;
45                              EXIT;
46                          END;
47                      Read(Medium, x);
48                      IF IOResult <> 0
49                      THEN
50                          BEGIN
51                              CH := WriteErrorMessage(' Reading a matrix: Unknown file
52                                  format');
53                              MatrixError := true;
54                              EXIT;
55                          END;
56                      SetMatrixElement(A, i, j, x);
57                  END; { for j }
58      ReadLn(Medium);
```

```

56     END; { for i }
57     Close(Medium);
58 END;

```

Listing 6.5: Write a matrix to a file

```

1 PROCEDURE WriteMatrix(MedStr: STRING; CONST A: MatrixTyp; ValidFigures:
2   BYTE);
3
4 VAR
5   i, j: WORD;
6   Medium: TEXT;
7
8 BEGIN
9   Assign(Medium, MedStr);
10  Rewrite(Medium);
11  Writeln(Medium, A^.Rows);
12  Writeln(Medium, A^.Columns);
13  IF NOT (IOResult = 0)
14    THEN
15      BEGIN
16        CH := WriteErrorMessage(' Writing a matrix: Illegal File
17          operation');
18        MatrixError := true;
19        EXIT;
20      END;
21  FOR i := 1 TO MatrixRows(A) DO
22    BEGIN
23      FOR j := 1 TO MatrixColumns(A) DO
24        Write(Medium, FloatStr(GetMatrixElement(A, i, j), ValidFigures), ' );
25        Writeln(Medium);
26    END;
27  Close(Medium);
28 END;

```

## 6.2. Accesing and changing data in a matrix

```

1 FUNCTION MatrixRows(CONST A: MatrixTyp): WORD;
2
3 BEGIN
4   MatrixRows := A^.Rows;
5 END;
6
7
8 FUNCTION MatrixColumns(CONST A: MatrixTyp): WORD;

```

## 6. Matrix calculations

```
9
10 BEGIN
11     MatrixColumns := A^.Columns;
12 END;
13
14
15 FUNCTION GetMatrixElement(CONST A: MatrixTyp; Row, Column: WORD): float;
16
17 VAR
18     n, p: WORD;
19
20 BEGIN
21     n := MatrixRows(A);
22     p := MatrixColumns(A);
23     IF (Row <= n) AND (Column <= p)
24         THEN
25             GetMatrixElement := A^.Data[Pred(Row) * p + Column]
26     ELSE
27         BEGIN
28             MatrixError := true;
29             CH := WriteErrorMessage(' Attempt to read a non-existent matrix
30                         element');
31         END;
32
33
34 PROCEDURE SetMatrixElement(VAR A: MatrixTyp; Row, Column: WORD; Value:
35                         float);
36
37 VAR
38     n, p: WORD;
39
40 BEGIN
41     n := MatrixRows(A);
42     p := MatrixColumns(A);
43     IF (Row <= n) AND (Column <= p)
44         THEN
45             A^.Data[Pred(Row) * p + Column] := Value
46     ELSE
47         BEGIN
48             MatrixError := true;
49             CH := WriteErrorMessage(' Attempt to write to a non-existent matrix
50                         element');
51         END;
52 END;
```

Just as with vectors, we cannot simply use  $\mathcal{A} = \mathcal{B}$  to copy a matrix. Copying should

be done with the following routine.

Listing 6.6: copy a matrix

```

1 PROCEDURE CopyMatrix(CONST Source: MatrixTyp; VAR Dest: MatrixTyp);
2
3 VAR
4   i, j, p, n: WORD;
5
6 BEGIN
7   n := MatrixRows(Source);
8   p := MatrixColumns(Source);
9   CreateMatrix(Dest, n, p, 0.0);
10  IF MatrixError THEN EXIT;
11  FOR i := 1 TO n DO
12    FOR j := 1 TO p DO
13      SetMatrixElement(Dest, i, j, GetMatrixElement(Source, i, j));
14 END;
```

Listing 6.7: Row operations

```

1 PROCEDURE GetRow(CONST A: MatrixTyp; Z: WORD; VAR Vek: VectorTyp);
2
3 VAR
4   j, m, n: WORD;
5
6 BEGIN
7   m := MatrixRows(A);
8   n := MatrixColumns(A);
9   IF Z > m
10  THEN
11    BEGIN
12      CH := WriteErrorMessage(' Matrix-Error: accessing a non-existent
13          row');
14      MatrixError := TRUE;
15      EXIT;
16    END;
17   CreateVector(Vek, n, 0.0);
18   FOR j := 1 TO n DO
19     SetVectorElement(Vek, j, GetMatrixElement(A, z, j));
20 END;
```

```

21 PROCEDURE SetRow(VAR A: MatrixTyp; CONST Vek: VectorTyp; Z: WORD);
22
23 VAR
24   j, n, p: WORD;
25
26 BEGIN
```

## 6. Matrix calculations

```

27   n := MatrixRows(A);
28   p := MatrixColumns(A);
29   IF (z > n) OR (p <> VectorLength(Vek))
30   THEN
31     BEGIN
32       CH := WriteErrorMessage('Matrix-error: set row with illegal
33           parameter');
34       MatrixError := TRUE;
35     EXIT;
36     END;
37   FOR j := 1 TO p DO
38     SetMatrixElement(A, z, j, GetVectorElement(Vek, j));
39   END;
40
41 PROCEDURE ExchangeRows(VAR A: MatrixTyp; Row1, Row2: WORD);
42
43 VAR
44   Dummy: float;
45   j, n, p: WORD;
46
47 BEGIN
48   n := MatrixRows(A);
49   p := MatrixColumns(A);
50   IF ((Row1 > n) OR (Row2 > n))
51   THEN
52     BEGIN
53       CH := WriteErrorMessage('Matrix-error: accessing non-existant row');
54       MatrixError := TRUE;
55     EXIT;
56     END;
57   FOR j := 1 TO p DO
58     BEGIN
59       Dummy := GetMatrixElement(A, Row1, j);
60       SetMatrixElement(A, Row1, j, GetMatrixElement(A, Row2, j));
61       SetMatrixElement(A, Row2, j, Dummy);
62     END;
63 END; { procedure ExchangeRows }
```

Listing 6.8: Column operations

```

1 PROCEDURE GetColumn(CONST A: MatrixTyp; S: WORD; VAR Vek: VectorTyp);
2
3 VAR
4   i, n, p: WORD;
5
6 BEGIN
```

```

7   n := MatrixRows(A);
8   p := MatrixColumns(A);
9   IF (S > p)
10  THEN
11  BEGIN
12    CH := WriteErrorMessage(' Matrix-error: accessing non-existent
13      column');
14    MatrixError := TRUE;
15    EXIT;
16  END;
17  CreateVector(Vek, n, 0.0);
18  FOR i := 1 TO n DO
19    SetVectorElement(Vek, i, GetMatrixElement(A, i, S));
20
21
22 PROCEDURE SetColumn(VAR A: MatrixTyp; CONST Vek: VectorTyp; S: WORD);
23
24 VAR
25   i, n, p: WORD;
26
27 BEGIN
28   n := MatrixRows(A);
29   p := MatrixColumns(A);
30   IF (S > p) OR (n <> VectorLength(Vek))
31   THEN
32   BEGIN
33     CH := WriteErrorMessage(' Matrix-error: setting non-existent
34      column');
35     MatrixError := TRUE;
36     EXIT;
37   END;
38   FOR i := 1 TO n DO
39     SetMatrixElement(A, i, S, GetVectorElement(Vek, i));
40
41
42 PROCEDURE ExchangeColumns(VAR A: MatrixTyp; Column1, Column2: WORD);
43 { Columns n und m vertauschen }
44
45 VAR
46   i, n, p: WORD;
47   Dummy: float;
48
49 BEGIN
50   n := MatrixRows(A);

```

## 6. Matrix calculations

```
51  p := MatrixColumns(A);
52  IF ((Column1 > p) OR (Column2 > p))
53  THEN
54    BEGIN
55      CH := WriteErrorMessage(' Matrix-error: accessing non-existent
56      column');
57      MatrixError := TRUE;
58      EXIT;
59    END;
60  FOR i := 1 TO n DO
61    BEGIN
62      Dummy := GetMatrixElement(A, i, Column1);
63      SetMatrixElement(A, i, Column1, GetMatrixElement(A, i, Column2));
64      SetMatrixElement(A, i, Column2, Dummy);
65    END;
66 END;
```

The following routine inverts the sign of all elements of a matrix:

Listing 6.9: invert sign of all matrix elements

```
1 PROCEDURE NegativeMatrix(VAR A: MatrixTyp);
2
3 VAR
4   i, j: WORD;
5
6 BEGIN
7   FOR i := 1 TO MatrixRows(A) DO
8     FOR j := 1 TO MatrixColumns(A) DO
9       SetMatrixElement(A, i, j, -GetMatrixElement(A, i, j));
10  END;
```

## 6.3. Special matrix types

Some matrices have special properties that simplify calculations with them or allow a particular method to be used.

### 6.3.1. Identity matrix

In an identity matrix all diagonal elements are unity, all off-diagonal elements zero. It is the neutral element of matrix multiplication.

Listing 6.10: identity matrix

```
1 PROCEDURE CreateIdentityMatrix(VAR A: MatrixTyp; n: WORD);
2 { n*n identity matrix }
3
```

```

4  VAR
5    i: WORD;
6
7  BEGIN
8    CreateMatrix(A, n, n, 0.0);
9    IF MatrixError THEN EXIT;
10   FOR i := 1 TO n DO
11     SetMatrixElement(A, i, i, 1.0);
12 END;

```

### 6.3.2. The null-matrix

A matrix  $\mathcal{A}$  is called null-matrix if all its elements are the neutral element of addition, zero.

Listing 6.11: null matrix

```

1  PROCEDURE CreateNullMatrix(VAR A: MatrixTyp; n: WORD);
2
3  BEGIN
4    CreateMatrix(A, n, n, 0.0);
5  END;

```

The following function tests, if a matrix is a null-matrix. Again, we set all elements to zero precisely, that are absolute smaller than the constant Zero.

Listing 6.12: are all matrix matrix elements zero?

```

1  FUNCTION NullMatrix(VAR A: MatrixTyp): BOOLEAN;
2
3  VAR
4    i, j: WORD;
5
6  BEGIN
7    Result := TRUE;
8    FOR i := 1 TO MatrixRows(A) DO
9      FOR j := 1 TO MatrixColumns(A) DO
10        IF Abs(GetMatrixElement(A, i, j)) > Zero
11          THEN
12            BEGIN
13              Result := FALSE;
14              EXIT;
15            END
16        ELSE
17          SetMatrixElement(A, i, j, 0.0);
18    END;

```

## 6. Matrix calculations

### 6.3.3. The HILBERT-matrix

The HILBERT-matrix  $\mathcal{H}_{n \times n}$  with  $h_{ij} = \frac{1}{i+j-1}$  is often used as a test system for matrix inversion and solving linear equations, as it is quite ill-conditioned.

Listing 6.13: Hilbert matrix

```

1 PROCEDURE CreateHilbertMatrix(VAR H: MatrixTyp; n: WORD);
2
3 VAR
4   i, j: WORD;
5
6 BEGIN
7   CreateMatrix(H, n, n, 0.0);
8   FOR i := 1 TO n DO
9     FOR j := 1 TO n DO
10      SetMatrixElement(H, i, j, 1 / Pred(i + j));
11 END;
```

### 6.3.4. The quadratic matrix

A matrix is quadratic if it has the same number of lines and columns:

Listing 6.14: is matrix square?

```

1 FUNCTION MatrixSquare(CONST A: MatrixTyp): BOOLEAN;
2
3 BEGIN
4   Result := MatrixRows(A) = MatrixColumns(A);
5 END;
```

### 6.3.5. The symmetric matrix

A Matrix is symmetric if it is quadratic and  $x_{ij} = x_{ji} \forall i, j \in 1 \dots n$ . Since we cannot test for identity of real numbers, we test for the difference being smaller than the typed constant `Zero`. In that case, we set the upper diagonal to the lower, to avoid numeric difficulties in later calculations.

```

1 FUNCTION MatrixSymmetric(VAR A: MatrixTyp): BOOLEAN;
2
3 VAR
4   i, j, Dimen: WORD;
5   x, y: float;
6
7 BEGIN
8   Dimen := MatrixRows(A);
9   IF NOT (MatrixSquare(A))
10  THEN
```

```

11   BEGIN
12     MatrixSymmetric := FALSE;
13     EXIT;
14   END;
15   FOR i := 1 TO Dimen DO
16     FOR j := 1 TO Dimen DO
17       BEGIN
18         x := GetMatrixElement(A, i, j);
19         y := GetMatrixElement(A, j, i);
20         IF Abs(x - y) > Zero
21           THEN
22             BEGIN
23               MatrixSymmetric := FALSE;
24               EXIT;
25             END
26           ELSE // make sure they are identical
27             SetMatrixElement(A, i, j, GetMatrixElement(A, j, i));
28       END;
29     MatrixSymmetric := TRUE;
30   END;

```

### 6.3.6. Trapezoid matrices

A matrix is left trapezoid, if all elements below the diagonal are zero. A matrix is right trapezoid if all elements above the diagonal are zero. Again, if these elements are smaller than `Zero`, we set them to 0.0:

Listing 6.15: is matrix left trapezoid

```

1 FUNCTION MatrixLeftTrapezoid(VAR A: MatrixTyp): BOOLEAN;
2
3 VAR
4   i, j: WORD;
5
6 BEGIN
7   Result := FALSE;
8   FOR i := 1 TO MatrixRows(A) DO
9     FOR j := 1 TO Pred(i) DO
10      IF Abs(GetMatrixElement(A, i, j)) > Zero
11        THEN EXIT
12        ELSE SetMatrixElement(A, i, j, 0.0);
13   Result := TRUE;
14 END;
15
16
17 FUNCTION MatrixRightTrapezoid(VAR A: MatrixTyp): BOOLEAN;
18

```

## 6. Matrix calculations

```

19  VAR
20    i, j: WORD;
21
22  BEGIN
23    Result := FALSE;
24    FOR i := 1 TO MatrixRows(A) DO
25      FOR j := Succ(i) TO MatrixColumns(A) DO
26        IF Abs(GetMatrixElement(A, i, j)) > Zero
27          THEN EXIT
28        ELSE SetMatrixElement(A, i, j, 0.0); // make sure
29    Result := TRUE;
30  END;

```

### 6.3.7. The diagonal matrix and function diag

If only the diagonal elements are different from zero, we call a matrix diagonal:

Listing 6.16: is matrix diagonal

```

1  FUNCTION MatrixDiagonal(VAR A: MatrixTyp): BOOLEAN;
2
3  BEGIN
4    Result := MatrixRightTrapezoid(A) AND MatrixLeftTrapezoid(A);
5  END;

```

The procedure `Diag` sets all of-diagonal elements of a matrix to zero:

Listing 6.17: set all of-diagonal elements to zero

```

1  PROCEDURE Diag(VAR Matrix: MatrixTyp);
2
3  VAR
4    n, p, i, j: WORD;
5
6  BEGIN
7    n := MatrixRows(Matrix);
8    p := MatrixColumns(Matrix);
9    IF (p <> n)
10      THEN
11        BEGIN
12          CH := WriteErrorMessage('Matrix-error: Diag of a non-square
13                         matrix');
14          MatrixError := TRUE;
15          EXIT;
16        END;
17    FOR i := 1 TO n DO
18      FOR j := Succ(i) TO p DO
19        BEGIN

```

```

19     SetMatrixElement(Matrix, i, j, 0.0);
20     SetMatrixElement(Matrix, j, i, 0.0);
21   END;
22 END;

```

### 6.3.8. Triangular matrices

Square matrices which are trapezoid are called triangular:

Listing 6.18: is matrix triangular?

```

1 FUNCTION MatrixUpperTriangular(VAR A: MatrixTyp): BOOLEAN;
2
3 BEGIN
4   Result := MatrixLeftTrapezoid(A) AND MatrixSquare(A);
5 END;
6
7
8 FUNCTION MatrixLowerTriangular(VAR A: MatrixTyp): BOOLEAN;
9
10 BEGIN
11   Result := MatrixRightTrapezoid(A) AND MatrixSquare(A);
12 END;

```

### 6.3.9. HESSENBERG matrices

The elements  $a_{ij}, j = i + 1$  are called superdiagonal. A matrix has the upper HESSENBERG form if it is quadratic and all elements above the superdiagonal are zero. The elements  $a_{ij}, j = i - 1$  are called infradiagonal. A matrix has the lower HESSENBERG form if it is quadratic and all elements below the infradiagonal are zero. If a matrix has both upper and lower HESSENBERG form, it is called tridiagonal.

Listing 6.19: has matrix the upper Hessenberg form?

```

1 FUNCTION MatrixUpperHessenberg(VAR A: MatrixTyp): BOOLEAN;
2
3 VAR
4   i, j: WORD;
5
6 BEGIN
7   Result := FALSE;
8   IF NOT (MatrixSquare(A)) THEN EXIT;
9   FOR i := 1 TO MatrixRows(A) DO
10     FOR j := 1 TO i - 2 DO
11       IF Abs(GetMatrixElement(A, i, j)) > Null
12         THEN EXIT
13       ELSE SetMatrixElement(A, i, j, 0.0);

```

## 6. Matrix calculations

```

14   Result := TRUE;
15 END;
16
17
18 FUNCTION MatrixLowerHessenberg(VAR A: MatrixTyp): BOOLEAN;
19
20 VAR
21   i, j: WORD;
22
23 BEGIN
24   Result := FALSE;
25   IF NOT (MatrixSquare(A)) THEN EXIT;
26   FOR i := 1 TO MatrixRows(A) DO
27     FOR j := i + 2 TO MatrixColumns(A) DO
28       IF Abs(GetMatrixElement(A, i, j)) > Null
29         THEN EXIT
30       ELSE SetMatrixElement(A, i, j, 0.0);
31   Result := TRUE;
32 END;
33
34
35 FUNCTION MatrixTridiagonal(VAR A: MatrixTyp): BOOLEAN;
36
37 BEGIN
38   Result := MatrixUpperHessenberg(A) AND MatrixLowerHessenberg(A);
39 END;

```

### 6.3.10. Positive and negative definite matrices

A square matrix is positive definite, if all diagonal elements are  $> 0$ , the largest element is on the diagonal and  $a_{ij}^2 < a_{ii} * a_{jj}$ . For such matrices, the GAUSS-algorithm without pivot search can be used. Their eigenvalues are all  $> 0$ ; positive semi-definite matrix have one or more eigenvalues that are  $= 0$ , their rank is equal to the number of eigenvalues  $> 0$ .

Listing 6.20: is matrix positive definite?

```

1 FUNCTION MatrixPositivDefinite(CONST A: MatrixTyp): BOOLEAN;
2
3 VAR
4   i, j: WORD;
5   Akt, Max: float;
6
7 BEGIN
8   Result := FALSE;
9   Max := 0.0;
10  IF NOT (MatrixSquare(A)) THEN EXIT;

```

```

11  FOR i := 1 TO MatrixRows(A) DO
12    BEGIN
13      Akt := GetMatrixElement(A, i, i);
14      IF Akt <= 0 THEN EXIT;
15      IF Akt > Max THEN Max := Akt;
16    END;
17  FOR i := 1 TO MatrixRows(A) DO
18    BEGIN
19      FOR j := 1 TO Pred(i) DO
20        BEGIN
21          Akt := GetMatrixElement(A, i, j);
22          IF Akt > Max
23            THEN EXIT;
24          IF Sqr(Akt) >= GetMatrixElement(A, i, i) * GetMatrixElement(A, j,
25              j)
26            THEN EXIT;
27        END;
28      FOR j := Succ(i) TO MatrixColumns(A) DO
29        BEGIN
30          Akt := GetMatrixElement(A, i, j);
31          IF Akt > Max
32            THEN EXIT;
33          IF Sqr(Akt) >= GetMatrixElement(A, i, i) * GetMatrixElement(A, j, j)
34            THEN EXIT;
35        END;
36      Result := TRUE;
37    END;

```

## 6.4. Calculation with matrices

### 6.4.1. Matrix norms

The trace of a quadratic matrix is the sum of its diagonal elements:

$$\text{tr}(\mathcal{A}_{n \times n}) = \sum_{i=1}^n a_{ii} \quad (6.1)$$

The following rules apply:

$$\text{tr}(\mathcal{A} + \mathcal{B}) = \text{tr}(\mathcal{A}) + \text{tr}(\mathcal{B}) \quad (6.2)$$

$$\text{tr}(\mathcal{A}\mathcal{B}) = \text{tr}(\mathcal{B}\mathcal{A}) \quad (6.3)$$

$$\text{tr}(\mathcal{A}^T \mathcal{A}) = \text{tr}(\mathcal{A}\mathcal{A}^T) = \sum_{i=1}^n \sum_{j=1}^p a_{ij}^2 \quad (6.4)$$

## 6. Matrix calculations

Listing 6.21: Trace of a matrix

```

1  FUNCTION MatrixTrace(CONST A: MatrixTyp): float;
2
3  VAR
4      i, n, p: WORD;
5      Sum: float;
6
7  BEGIN
8      n := MatrixRows(A);
9      p := MatrixColumns(A);
10     IF (n <> p)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage(' Matrix-error: trace of a matrix that is
14                     not square');
15                 MatrixError := TRUE;
16                 EXIT;
17             END;
18     Sum := 0;
19     FOR i := 1 TO n DO
20         Sum := Sum + GetMatrixElement(A, i, i);
21     MatrixTrace := Sum;
22 END;
```

Matrix-norms **induced by vector-norms** for a (not necessarily quadratic) matrix  $\mathcal{A}_{n \times p}$  are:

$$\|\mathbf{A}\|_{\infty} = \max_{1 \leq i \leq n} \sum_{j=1}^p |a_{ij}| \quad (6.5)$$

$$\|\mathbf{A}\|_1 = \max_{1 \leq j \leq p} \sum_{i=1}^n |a_{ij}| \quad (6.6)$$

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^n \sum_{j=1}^p |a_{ij}|^2} \quad (6.7)$$

the maximum absolute row sum, maximum absolute column sum and FROBENIUS-norm, respectively. For example

$$\begin{array}{ccc|c} -3 & 5 & 7 & 15 \\ 2 & 6 & 4 & 12 \\ 0 & 2 & 8 & 10 \\ \hline 5 & 13 & 19 & \end{array}$$

$\|\mathbf{A}\|_1 = \max(5, 13, 19) = 19$  and  $\|\mathbf{A}\|_{\infty} = \max(15, 12, 10) = 15$ . The trace  $\text{tr}(\mathcal{A}) = -3 + 6 + 8 = 11$ . The FROBENIUS-Norm is

$$\|\mathbf{A}\|_F = \sqrt{(-3^2 + 5^2 + 7^2) + (2^2 + 6^2 + 4^2) + (0^2 + 2^2 + 8^2)} = \sqrt{207} = 14.4 \quad (6.8)$$

**Entry-wise norms** treat a matrix  $\mathcal{A}_{n \times p}$  as vector of length  $np$ . Then

$$\|\mathcal{A}\|_q = \sqrt[q]{\sum_{i=1}^{np} |\alpha_i|^q} \quad (6.9)$$

with the special cases  $q = 2$  the FROBENIUS-norm and  $q = \infty$  the maximum norm.

The **Schatten-norms** are defined as follows:

$$\|\mathcal{A}\|_q = \sqrt[q]{\sum_{i=1}^{\min(n,p)} \sigma_i^q} \quad (6.10)$$

with  $\sigma_i$  the i-th singular value (*vide infra*) of  $\mathcal{A}$ .

```

1  FUNCTION FrobeniusNorm(CONST A: MatrixTyp): float;
2
3  VAR
4      i, j, n, p: WORD;
5      Sum: extended;
6
7  BEGIN
8      n := MatrixRows(A);
9      p := MatrixColumns(A);
10     Sum := 0.0;
11     FOR i := 1 TO n DO
12         FOR j := 1 TO p DO
13             Sum := Sum + Sqr(GetMatrixElement(A, i, j));
14     FrobeniusNorm := Sqrt(Sum);
15 END;
```

The FROBENIUS scalar product of two matrices  $\|\mathcal{A}, \mathcal{B}\| = \text{trace}(\mathcal{A}^T \mathcal{B})$ . It follows that  $\|\mathcal{A}\|_F = \sqrt{\|\mathcal{A}^T, \mathcal{A}\|}$ :

Listing 6.22: Frobenius skalar product

```

1  FUNCTION FrobeniusSkalarProduct(CONST A, B: MatrixTyp): float;
2
3  VAR
4      CH: CHAR;
5      C, D: MatrixTyp;
6      n, p: WORD;
7
8  BEGIN
9      n := MatrixRows(A);
10     p := MatrixColumns(A);
11     IF NOT ((n = MatrixRows(B)) AND (p = MatrixColumns(B)))
12     THEN
13         BEGIN
```

## 6. Matrix calculations

```

14      CH := WriteErrorMessage(
15          ' Matrix-error: Frobenius-Skalar product of incompatible
16          matrices');
17      MatrixError := TRUE;
18      EXIT;
19  END;
20  MatrixTranspose(A, C);           // C = A^T
21  MatrixInnerProduct(C, B, D);    // D = A^T B
22  FrobeniusSkalarProduct := MatrixTrace(D);
23  DestroyMatrix(C);
24  DestroyMatrix(D);
25 END;

```

### 6.4.2. Determinant

To calculate the determinant, we will need a little helper function, that adds a multiple of row 1 to row 2. It is also required for the solution of systems of equations.

Listing 6.23: add multiple of row 1 to row 2

```

1 PROCEDURE ERoMultAdd(VAR A: MatrixTyp; Faktor: float; Row1, Row2: WORD);
2
3 VAR
4     j: WORD;
5
6 BEGIN
7     FOR j := 1 TO MatrixColumns(A) DO
8         SetMatrixElement(A, Row2, j, GetmatrixElement(A, Row2, j) +
9             GetMatrixElement(A, Row1, j) * Faktor);
10 END;

```

The determinant of a square matrix  $|\mathcal{A}_{n \times n}|$  is defined as the sum of all  $n!$  possible products of  $n$  elements such that

1. each product contains one element from every row and every column
2. the factors in each product are written so that the column subscripts appear in order of magnitude and each product is then preceded by a plus or minus sign according to whether the number of inversions in the row subscripts is even or odd.
3. An inversion occurs whenever a larger number precedes a smaller one.

For a  $3 \times 3$  matrix, this is

$$|\mathcal{A}| = \det \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \quad (6.11)$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{31}a_{22}a_{13} - a_{32}a_{23}a_{11} - a_{33}a_{12}a_{21} \quad (6.12)$$

For larger matrices, this becomes unwieldy.

The following special cases exist:

- For diagonal matrices, the determinant is the product of the diagonal elements  $|\mathcal{X}| = \prod_{i=1}^n (\mathfrak{x}_i)$ .
  - If a square matrix is singular, then its determinant is  $|\mathcal{X}| = 0$ , if it is non-singular, the determinant  $|\mathcal{X}| \neq 0$ .
  - If a matrix is **positive definite**, then its determinant is positive.
  - $|\mathcal{X}^T| = |\mathcal{X}|$ ,  $|\mathcal{X}^{-1}| = |\mathcal{X}|^{-1}$

Listing 6.24: Determinant of a matrix

```

1  FUNCTION Determinante(CONST A: MatrixTyp): float;
2
3  VAR
4      PartialDeter, Multiplier: float;
5      Row, ReferenceRow: WORD;
6      DetEqualsMaxError: BOOLEAN;
7      Copy: MatrixTyp;
8
9  PROCEDURE Pivot(ReferenceRow: WORD; VAR PartialDeter: float;
10   VAR DetEqualsMaxError: BOOLEAN);
11  {- This procedure searches the ReferenceRow column of the matrix Data
12    for
13    the first non-MaxError element below the diagonal. If it finds one,
14    then
15    the procedure switches rows so that the non-MaxError element is on
16    the
17    diagonal. Switching rows changes the determinant by a factor of -1;
18    this change is returned in PartialDeter. If it doesn't find one, the
19    matrix is singular and the Determinant is MaxError
20    (DetEqualsMaxError = true
21    is returned. -}
22
23  VAR
24      NewRow: INTEGER;
25
26  BEGIN
27      DetEqualsMaxError := TRUE;
28      NewRow := ReferenceRow;
29      WHILE DetEqualsMaxError AND (NewRow < MatrixRows(Copy)) DO
30          BEGIN { Try to find a row with a non-MaxError }
31              NewRow := Succ(NewRow);
32              IF Abs(GetMatrixElement(Copy, NewRow, ReferenceRow)) > MaxError
33                  THEN
34
35  END.

```

## 6. Matrix calculations

```

30      BEGIN
31          ExchangeRows(Copy, NewRow, ReferenceRow); { Switch these two
32          rows }
33          DetEqualsMaxError := FALSE;
34          PartialDeter := -PartialDeter; { Switching rows changes }
35      END; { the determinant by a factor of -1 }
36  END; { procedure Pivot }

37
38 BEGIN { Determinante }
39 IF MatrixRows(A) = 1
40 THEN
41 BEGIN
42     Result := GetMatrixElement(A, 1, 1);
43     EXIT;
44 END;
45 CopyMatrix(A, Copy);
46 IF MatrixError THEN EXIT;
47 DetEqualsMaxError := FALSE;
48 PartialDeter := 1;
49 ReferenceRow := 0;
50 { Make the matrix upper triangular }
51 WHILE NOT (DetEqualsMaxError) AND (ReferenceRow < Pred(MatrixRows(Copy)))
52 DO
53 BEGIN
54     INC(ReferenceRow);
55     { If diagonal element is MaxError then switch rows }
56     IF Abs(GetMatrixElement(Copy, ReferenceRow, ReferenceRow)) < MaxError
57         THEN Pivot(ReferenceRow, PartialDeter, DetEqualsMaxError);
58     IF NOT (DetEqualsMaxError)
59         THEN
60             FOR Row := Succ(ReferenceRow) TO MatrixRows(Copy) DO
61                 { Make the ReferenceRow element of this row MaxError }
62                 IF Abs(GetMatrixElement(Copy, Row, ReferenceRow)) > MaxError
63                     THEN
64                         BEGIN
65                             Multiplier :=
66                                 -GetMatrixElement(Copy, Row, ReferenceRow) /
67                                 GetMatrixElement(Copy, ReferenceRow, ReferenceRow);
68                             EROmultAdd(Copy, Multiplier, ReferenceRow, Row);
69                         END;
70                         { Multiply the diagonal Term into PartialDeter }
71                         PartialDeter := PartialDeter * GetMatrixElement(Copy, ReferenceRow,
72                                         ReferenceRow);
73 END; { while }
74 IF DetEqualsMaxError

```

```

73   THEN
74     Result := 0
75   ELSE
76     Result := PartialDeter * GetMatrixElement(Copy, MatrixRows(Copy),
77       MatrixColumns(Copy));
78     DestroyMatrix(Copy);
79 END; { function Determinante }
```

Example:

$$\det \begin{pmatrix} 1.00 & 2.00 & 0.00 & -1.00 \\ -1.00 & 4.00 & 3.00 & -0.50 \\ 2.00 & 2.00 & 1.00 & -3.00 \\ 0.00 & 0.00 & 3.00 & -4.00 \end{pmatrix} = 21.0 \quad (6.13)$$

### Leading principle minors

The leading principle minors of a symmetric matrix  $\mathcal{A}_{n \times n}$  are the determinants of the sub-matrices with  $1, 2, \dots, n$  rows and columns, starting with the upper left corner.  $\mathcal{A}$  is positive definite if all principal minors are positive, it is negative definite if all even principle minors are positive, all odd negative (SYLVESTER-criterion).

Listing 6.25: Leading principle minors

```

1 PROCEDURE LeadingPrincipleMinors(VAR A: MatrixTyp; VAR V: VectorTyp);
2 // Implementierung nicht elegant, bei Bgroen Matrizen Wiederholung vermeiden
3
4 VAR
5   n, i, j, k: WORD;
6   B: MatrixTyp;
7
8 BEGIN
9   IF NOT MatrixSymmetric(A)
10  THEN
11    BEGIN
12      CH := WriteErrorMessage( 'Matrix-error: leading principle minors of
13        a non-square matrix');
14      MatrixError := TRUE;
15      EXIT;
16    END;
17   n := MatrixRows(A);
18   CreateVector(V, n, 0.0);
19   FOR i := 1 TO n DO
20     BEGIN
21       CreateMatrix(B, i, i, 0.0);
22       FOR j := 1 TO i DO
23         FOR k := 1 TO i DO
24           SetMatrixElement(B, j, k, GetMatrixElement(A, j, k));
```

## 6. Matrix calculations

```
24     SetVectorElement(V, i, Determinante(B));
25     DestroyMatrix(B);
26   END;
27   FOR i := 1 TO n DO
28     IF Abs(GetVectorElement(V, i)) < Zero THEN SetVectorElement(V, i, 0);
29   END;
```

The following routine changes the norm of a matrix:

```
1 PROCEDURE ChangeMatrixNorm(VAR A: MatrixTyp; Norm: float);
2
3 VAR
4   SP: float;
5   i, j: WORD;
6
7 BEGIN
8   IF Abs(Norm) < MaxError
9     THEN
10    BEGIN
11      CH := WriteErrorMessage(' Norm of a matrix must be greater than 0');
12      MatrixError := true;
13      EXIT;
14    END;
15   SP := 0;
16   FOR i := 1 TO MatrixRows(A) DO
17     FOR j := 1 TO MatrixColumns(A) DO
18       SP := SP + Sqr(GetMatrixElement(A, i, j));
19   IF Abs(SP) < MaxError
20     THEN
21     BEGIN
22       CH := WriteErrorMessage(' Norm of a matrix: no solution');
23       MatrixError := true;
24       EXIT;
25     END;
26   Norm := Norm / Sqrt(SP);
27   FOR i := 1 TO MatrixRows(A) DO
28     FOR j := 1 TO MatrixColumns(A) DO
29       SetMatrixElement(A, i, j, GetMatrixElement(A, i, j) * norm);
30 END;
```

### 6.4.3. Matrix transpose

The transpose of a matrix  $A^T$  is generated by turning the matrix by 90°, so that rows turn into columns and *vice versa*. In the following routine, the input matrix and its transpose must be different:

```
1 PROCEDURE MatrixTranspose(CONST A: MatrixTyp; VAR B: MatrixTyp);
2
```

```

3  VAR
4    i, j, n, p: WORD;
5
6  BEGIN
7    n := MatrixRows(A);
8    p := MatrixColumns(A);
9    CreateMatrix(B, p, n, 0.0);
10   FOR i := 1 TO n DO
11     FOR j := 1 TO p DO
12       SetMatrixElement(B, j, i, GetMatrixElement(A, i, j));
13 END;

```

For matrices with complex elements, the **conjugate (HERMITEian) transpose**  $\mathcal{A}^*$  is obtained from the transpose by calculating the complex conjugate of each element. For real matrices,  $\mathcal{A}^* = \mathcal{A}^T$ . Matrices, for which  $\mathcal{A}^* \mathcal{A} = \mathcal{A} \mathcal{A}^*$  are called **normal**.

### Orthogonal vectors and matrices

Two vectors are said to be orthogonal (perpendicular) if they have the same size and  $a^T b = 0$ . If  $\sqrt{a^T a}$  is the length of a vector, then  $c = \frac{a}{\sqrt{a^T a}}$  is said to be normalised and  $c^T c = 1$ .

A matrix  $C$  is orthogonal if all its columns  $c_1, c_2, \dots, c_p$  are normalised and orthogonal to each other. Then  $C^T C = C C^T = I$ , thus the rows are also normalised and orthogonal.

If  $z = Cx$  and  $C$  is orthogonal, then

$$z^T z = (Cx)^T (Cx) = x^T C^T C x = x^T I x = x^T x \quad (6.14)$$

and  $x$  and  $z$  have the same length, but the axes are rotated.

#### 6.4.4. Matrix inverse

The inverse of a matrix  $\mathcal{A}^{-1}$  is defined as  $\mathcal{A} \mathcal{A}^{-1} = I$  the identity matrix (diagonal matrix with all diagonal elements equal 1.0), it is equivalent to the reciprocal of a number:

Listing 6.26: Inverse of a matrix

```

1  PROCEDURE InverseMatrix(VAR A: MatrixTyp);
2
3  VAR
4    i, j, m, n, p: WORD;
5    NoExch: 0..MaxVector;
6    Exch: ARRAY [1..MaxVector, 1..2] OF INTEGER;
7
8
9  PROCEDURE Transform(m: WORD);
10
11  VAR

```

## 6. Matrix calculations

```
12      B: MatrixTyp;
13      i, j: WORD;
14      Pivot: float;
15
16      BEGIN
17          CopyMatrix(A, B);
18          Pivot := GetMatrixElement(B, m, m);
19          IF Abs(Pivot) <= MaxError THEN
20              BEGIN
21                  CH := WriteErrorMessage('Matrix error: inversion of singular matrix');
22                  MatrixError := true;
23                  EXIT;
24              END;
25          FOR i := 1 TO n DO
26              FOR j := 1 TO n DO
27                  IF i <> m
28                      THEN
29                          IF j <> m
30                              THEN
31                                  SetMatrixElement(B, i, j, GetMatrixElement(A, i, j) -
32                                      GetMatrixElement(A, i, m) * GetMatrixElement(A, m, j) /
33                                      Pivot)
34                          ELSE
35                              SetMatrixElement(B, i, j, GetMatrixElement(A, i, j) / Pivot)
36                          ELSE IF j <> m
37                              THEN SetMatrixElement(B, i, j, -GetMatrixElement(A, i, j)
38                                     / Pivot)
39                          ELSE SetMatrixElement(B, i, j, 1 / Pivot);
40          DestroyMatrix(A);
41          CopyMatrix(B, A);
42          DestroyMatrix(B);
43      END; { Transform }
44
45      BEGIN { InverseMatrix }
46          NoExch := 0;
47          m := MatrixRows(A);
48          n := MatrixColumns(A);
49          IF m <> n
50              THEN
51                  BEGIN
52                      CH := WriteErrorMessage(' Matrix error: inversion of non-quadratic
53                         matrix');
54                      MatrixError := true;
55                      EXIT;
56                  END;
57          FOR i := 1 TO m DO
```

```

55   BEGIN
56     p := i;
57     FOR j := Succ(i) TO n DO
58       IF Abs(GetMatrixElement(A, j, i)) > Abs(p)
59         THEN p := j;
60     IF p <> i
61       THEN
62         BEGIN
63           INC(NoExch);
64           Exch[NoExch, 1] := i;
65           Exch[NoExch, 2] := p;
66           ExchangeRows(A, i, p);
67         END;
68         Transform(i);
69         IF MatrixError THEN EXIT;
70       END;
71     FOR i := NoExch DOWNTO 1 DO
72       ExchangeColumns(A, Exch[i, 2], Exch[i, 1]);
73   END; { InverseMatrix }

```

Example:

$$\begin{pmatrix} 1.00 & 2.00 & 0.00 & -1.00 \\ -1.00 & 4.00 & 3.00 & -0.50 \\ 2.00 & 2.00 & 1.00 & -3.00 \\ 0.00 & 0.00 & 3.00 & -4.00 \end{pmatrix}^{-1} = \begin{pmatrix} -1.95 & 0.19 & 1.57 & -0.71 \\ 0.76 & 0.05 & -0.36 & 0.07 \\ -1.90 & 0.38 & 1.14 & -0.43 \\ -1.43 & 0.29 & 0.86 & -0.57 \end{pmatrix} \quad (6.15)$$

## 6.4.5. Matrix sums and differences

Matrices, if they have the same size, are added by adding their elements  $c_{ij} = a_{ij} + b_{ij}$ :

Listing 6.27: Sum of two matrices

```

1 PROCEDURE MatrixAdd(CONST A, B: MatrixTyp; VAR Res: MatrixTyp);
2 { Elementweise Addition zweier Matrizen. }
3
4 VAR
5   i, j, Rows, Columns: WORD;
6
7 BEGIN
8   Rows := MatrixRows(A);
9   Columns := MatrixColumns(A);
10  IF ((Rows <> MatrixRows(B)) OR (Columns <> MatrixColumns(B)))
11    THEN
12      BEGIN
13        CH := WriteErrorMessage(
14          ' Matrix error: addition of matrices not of the same size');
15        MatrixError := true;

```

## 6. Matrix calculations

```

16     EXIT;
17   END;
18   CreateMatrix(Res, Rows, Columns, 0);
19   FOR i := 1 TO Rows DO
20     FOR j := 1 TO Columns DO
21       SetMatrixElement(Res, i, j, GetMatrixElement(A, i, j) +
22                     GetMatrixElement(B, i, j));
23   END;

```

A matrix is multiplied with a scalar by multiplying all elements with the scalar:

```

1 PROCEDURE SkalarMultiplikation(VAR A: MatrixTyp; x: float);
2 { Elementweise Multiplikation einer Matrix mit einem Skalar }
3
4 VAR
5   i, j: WORD;
6
7 BEGIN
8   FOR i := 1 TO MatrixRows(A) DO
9     FOR j := 1 TO MatrixColumns(A) DO
10      SetMatrixElement(A, i, j, GetMatrixElement(A, i, j) * x);
11 END;

```

### 6.4.6. Matrix inner and HADAMARD-SCHUR product

The inner product of a matrix is defined by the inner products of all rows of the first matrix with all columns of the second. Hence the number of columns of the first and the number of rows of the second matrix need to be identical ( $k$ ):  $\mathcal{A}_{n \times p} \mathcal{B}_{p \times q} = \mathcal{C}_{n \times q}$ ,  $c_{ij} = \mathbf{a}_i \cdot \mathbf{b}_j = a_{i1}b_{1j} + a_{i2}b_{2j} \dots a_{ip}b_{pj} = \sum_k a_{ik}b_{kj}$ . Calculation via the vector dot product has the advantage that the summation is performed by NEUMAIER-process, preventing error accumulation. Note that for square matrices of equal size both  $\mathcal{AB}$  and  $\mathcal{BA}$  are defined, but  $\mathcal{AB} \neq \mathcal{BA}$ .

Listing 6.28: inner product of two matrices

```

1 PROCEDURE MatrixInnerProduct(CONST A, B: MatrixTyp; VAR C: MatrixTyp);
2
3 VAR
4   i, j: WORD;
5   Col, Row: VectorTyp;
6
7 BEGIN
8   IF MatrixColumns(A) <> MatrixRows(B)
9     THEN
10    BEGIN
11      CH := WriteErrorMessage(' Matrix error: multiplication of A.Columns
12        <> B.Rows');
13      MatrixError := true;

```

```

13      EXIT;
14  END;
15 CreateMatrix(C, MatrixRows(A), MatrixColumns(B), 0.0);
16 FOR i := 1 TO MatrixRows(A) DO
17   FOR j := 1 TO MatrixColumns(B) DO
18     BEGIN
19       GetRow(A, i, Row);
20       GetColumn(B, j, Col);
21       SetMatrixElement(C, i, j, VectorInnerProduct(Row, Col));
22       DestroyVector(Row);
23       DestroyVector(Col);
24     END;
25 END;

```

Much rarer used than the inner product of two matrices is the HADAMARD-SCHUR-(element-wise) product:

Listing 6.29: Hadamard-Schur product of two matrices

```

1 PROCEDURE HadamardSchurProduct(CONST A, B: MatrixTyp; VAR C: MatrixTyp);
2
3 VAR
4   i, j, n, p: WORD;
5
6 BEGIN
7   n := MatrixRows(A);
8   IF (n <> MatrixRows(B))
9     THEN
10    BEGIN
11      CH := WriteErrorMessage('Hadamard-Schur multiplication: A.Rows <>
12          B.Rows');
13      MatrixError := true;
14      EXIT;
15    END;
16   p := MatrixColumns(A);
17   IF (p <> MatrixColumns(B))
18     THEN
19       BEGIN
20         CH := WriteErrorMessage('Hadamard-Schur multiplication: A.Columns
21             <> B.Columns');
22         MatrixError := true;
23         EXIT;
24       END;
25   CreateMatrix(C, n, p, 0.0);
26   FOR i := 1 TO n DO
27     FOR j := 1 TO p DO
28       SetMatrixElement(C, i, j, GetMatrixElement(A, i, j) *
29                     GetMatrixElement(B, i, j));

```

## 6. Matrix calculations

27   **END;**

### 6.4.7. Matrix division

Listing 6.30: division of two matrices

```

1  PROCEDURE MatrixDivision(CONST A, B: MatrixTyp; VAR C: MatrixTyp);
2
3  VAR
4      Rows, Columns, i, j, k: WORD;
5      M, N: MatrixTyp;
6
7  BEGIN
8      Rows := MatrixRows(A);
9      Columns := MatrixColumns(A);
10     IF (Rows <> Columns) OR (Columns <> MatrixRows(B))
11     THEN
12         BEGIN
13             CH := WriteErrorMessage(
14                 ' Matrix division: A.Columns <> B.Rows or A not quadratic');
15             MatrixError := TRUE;
16             EXIT;
17         END;
18     CopyMatrix(A, M);
19     CopyMatrix(B, N);
20     IF MatrixError THEN EXIT;
21     FOR j := 1 TO Rows DO
22         BEGIN
23             IF GetMatrixElement(M, j, j) = 0
24             THEN
25                 FOR k := j TO Rows DO
26                     IF GetMatrixElement(M, k, j) <> 0
27                     THEN
28                         BEGIN
29                             ExchangeRows(M, j, k);
30                             ExchangeRows(N, j, k);
31                         END;
32                     ELSE IF k = Rows
33                     THEN
34                         BEGIN
35                             CH := WriteErrorMessage(' Matrix division: no
36                                         solution');
37                             MatrixError := TRUE;
38                             EXIT;
39                         END;
40             FOR i := Succ(j) TO Rows DO

```

```

40      SetMatrixElement(M, i, j, GetMatrixElement(M, i, j) /
41          GetMatrixElement(M, j, j));
42  FOR i := Succ(j) TO Rows DO
43    BEGIN
44      FOR k := Succ(j) TO Rows DO
45        SetMatrixElement(M, i, k, GetMatrixElement(M, i, k) -
46            GetMatrixElement(M, j, k) * GetMatrixElement(M, i, j));
47      FOR k := 1 TO MatrixColumns(N) DO
48        SetMatrixElement(N, i, k, GetMatrixElement(N, i, k) -
49            GetMatrixElement(N, j, k) * GetMatrixElement(M, i, j));
50    END;
51  END;
52  FOR i := Rows DOWNTO 1 DO
53    FOR k := 1 TO MatrixColumns(N) DO
54      BEGIN
55        FOR j := Succ(i) TO Columns DO
56          SetMatrixElement(N, i, k, GetMatrixElement(N, i, k) -
57              GetMatrixElement(N, j, k) * GetMatrixElement(M, i, j));
58        SetMatrixElement(N, i, k, GetMatrixElement(N, i, k) /
59            GetMatrixElement(M, i, i));
60      END;
61  CopyMatrix(N, C);
62  DestroyMatrix(M);
63  DestroyMatrix(N);
64 END;

```

```

1 PROCEDURE CenterData(VAR A: MatrixTyp);
2
3 VAR
4   Data: VectorTyp;
5   j, Columns: WORD;
6
7 BEGIN
8   Columns := MatrixColumns(A);
9   FOR j := 1 TO Columns DO
10    BEGIN
11      GetColumn(A, j, Data);
12      Centre(Data);
13      SetColumn(A, Data, j);
14      DestroyVector(Data);
15    END;
16 END;

```

### 6.4.8. Symmetric and anti-symmetric matrices

The following routine splits a matrix into a symmetric and an anti-symmetric:

## 6. Matrix calculations

Listing 6.31: split matrix in symmetric and anti-symmetric

```

1 PROCEDURE AntiSym(CONST A: MatrixTyp; VAR Symmetric, Antisymmetric:
2   MatrixTyp);
3
4 VAR
5   Dimen, i, j: WORD;
6
7 BEGIN
8   Dimen := MatrixRows(A);
9   IF Dimen <> MatrixColumns(A)
10  THEN
11    BEGIN
12      CH := WriteErrorMessage('Antisymmetrical of an unsymmetric Matrix');
13      MatrixError := true;
14      EXIT;
15    END;
16    FOR i := 1 TO Dimen DO
17      FOR j := 1 TO Dimen DO
18        BEGIN
19          SetMatrixElement(Antisymmetric, i, j, 0.5 *
20            (GetMatrixElement(A, i, j) + GetMatrixElement(A, j, i)));
21          SetMatrixElement(Symmetric, i, j, GetMatrixElement(A, i, j) -
22            GetMatrixElement(Antisymmetric, i, j));
23        END;
24  END;

```

### 6.4.9. Matrix exponentiation

The matrix exponential is defined for square matrices  $\mathcal{X} \in \mathbb{R}_{n \times n}$  or  $\mathbb{C}_{n \times n}$  by a TAYLOR-series:

$$e^{\mathcal{X}} = \sum_{k=0}^{\infty} \frac{\mathcal{X}^k}{k!} = \mathcal{I} + \mathcal{X} + \frac{\mathcal{X}^2}{2} + \dots \quad (6.16)$$

, which always converges.

The matrix exponential shares a number of properties with the exponential on real numbers:

- $e^{\mathbf{0}} = \mathcal{I}$
- $e^{a\mathcal{X}} e^{b\mathcal{X}} = e^{(a+b)\mathcal{X}} \forall a, b \in \mathbb{C}$
- $e^{\mathcal{X}} e^{-\mathcal{X}} = e^{(1-1)\mathcal{X}} = e^{\mathbf{0}} = \mathcal{I}$
- $(e^{\mathcal{X}})^{-1} = e^{-\mathcal{X}}$
- $e^{\mathcal{X}+\mathcal{Y}} = e^{\mathcal{X}} e^{\mathcal{Y}}$  for commuting matrices ( $\mathcal{X}\mathcal{Y} = \mathcal{Y}\mathcal{X}$ ).

- $e^{\mathcal{X}^T} = (e^{\mathcal{X}})^T$
- $\det(e^{\mathcal{X}}) = e^{\text{tr}(\mathcal{X})}$
- $e^{\text{diag}(x_1, x_2, \dots, x_n)} = \text{diag}(e^{x_1}, e^{x_2}, \dots, e^{x_n})$
- $e^{\mathcal{X}}$  is always invertible.

One applications of the matrix exponential is solving systems of ordinary differential equations, for example starting value problems:

$$\frac{d}{dt}y(t) = \mathcal{X}y(t), \quad y(t_0) = y_0 \quad (6.17)$$

for square matrix  $\mathcal{X}$  is given by

$$y(t) = e^{(t-t_0)\mathcal{X}}y_0 \quad (6.18)$$

For algorithms for matrix exponentials see [8–10].

## 6.5. Calculations with vectors and matrices

### 6.5.1. Dyadic vector product

The dyadic vector product multiplies two vectors, resulting in a matrix:

$$\mathcal{C}_{n \times p} = \mathbf{a}_n \otimes \mathbf{b}_p \quad \mathbf{c}_{i,j} = \mathbf{a}_i \times \mathbf{b}_j \quad (6.19)$$

```

1 PROCEDURE DyadicVectorProduct(CONST A, B: VectorTyp; VAR C: MatrixTyp);
2
3 VAR
4   Row, Column: WORD;
5
6 BEGIN
7   CreateMatrix(C, A^.Length, B^.Length, 0.0);
8   FOR Row := 1 TO A^.Length DO
9     FOR Column := 1 TO B^.Length DO
10       SetMatrixElement(C, Row, Column, GetVectorElement(A, Row) *
11                     GetVectorElement(B, Column));
12 END;
```

### 6.5.2. Product of a matrix and a vector

$$\mathcal{C}_n = \mathcal{A}_{n \times p} \star \mathbf{b}_p \quad \mathbf{c}_i = \sum_{j=1}^p \mathbf{a}_{ij} \times \mathbf{b}_j \quad (6.20)$$

## 6. Matrix calculations

Listing 6.32: multiply matrix and vector

```
1 PROCEDURE MultMatrixVector(CONST Mat: MatrixTyp; CONST Vek: VectorTyp;
2   VAR Result: VectorTyp);
3
4 VAR
5   Row, Column: WORD;
6   Sum: float;
7
8 BEGIN
9   IF NOT (MatrixColumns(Mat) = VectorLength(Vek))
10  THEN
11    BEGIN
12      CH := WriteErrorMessage(' Matrix and vector have different number
13        of rows');
14      MatrixError := true;
15      EXIT;
16    END;
17    CreateVector(Result, MatrixRows(Mat), 0.0);
18    FOR Row := 1 TO MatrixRows(Mat) DO
19      BEGIN
20        Sum := 0;
21        FOR Column := 1 TO MatrixColumns(Mat) DO
22          Sum := Sum + GetMatrixElement(Mat, Row, Column) +
23            GetVectorElement(Vek, Column);
24        SetVectorElement(Result, Row, Sum);
25      END;
26  END;
```

### 6.5.3. Change a vector to a matrix with one row

Listing 6.33: change vector to one-dimensional matrix

```
1 PROCEDURE CangeVectorToMatrix(CONST Vek: VectorTyp; VAR Mat: MatrixTyp);
2
3 VAR
4   j: WORD;
5
6 BEGIN
7   CreateMatrix(Mat, VectorLength(Vek), 1, 0.0);
8   FOR j := 1 TO VectorLength(Vek) DO
9     SetMatrixElement(Mat, j, 1, GetVectorElement(Vek, j));
10  END;
```

## 6.6. Sort rows of a matrix by value of a column

It is often necessary to sort all rows of a matrix so that a particular column is sorted. We use basically the same routine as for sorting vectors (see 5.4 on page 139).

Listing 6.34: sort matrix

```

1 PROCEDURE ShellSortMatrix(VAR t: MatrixTyp; Column: WORD);
2
3 LABEL
4   10;
5
6 VAR
7   i, j, k, l, m, nn, NaNs, n, LdN: INTEGER;
8   s: float;
9   tmp1, tmp2: VectorTyp;
10
11 BEGIN
12   NaNs := 0;
13   s := -MaxRealNumber;
14   n := MatrixRows(t);
15   FOR i := 1 TO n DO
16     BEGIN
17       IF IsNaN(GetMatrixElement(t, i, Column)) // check forR NaN data
18         THEN INC(NaNs)
19         ELSE IF (GetMatrixElement(t, i, Column)) > s
20           THEN
21             s := GetMatrixElement(t, i, Column); // and find largest
22               element of data vector
23     END;
24   s := 10 * s;
25   IF (NaNs > 0)
26     THEN
27       FOR i := 1 TO n DO
28         IF IsNaN(GetMatrixElement(t, i, Column))
29           THEN SetMatrixElement(t, i, Column, s);
30 // replace all NaN WITH very large number so they Move TO END OF vector
31   LdN := Trunc(Ln(n) / Const_ln2);
32   m := n;
33   FOR nn := 1 TO LdN DO
34     BEGIN
35       m := m DIV 2;
36       k := n - m;
37       FOR j := 1 TO k DO
38         BEGIN
39           i := j;
10           l := i + m;
```

## 6. Matrix calculations

```

40          IF (GetMatrixElement(t, l, Column) < GetMatrixElement(t, i,
41                                         Column))
42          THEN
43              BEGIN
44                  GetRow(t, i, tmp1);
45                  GetRow(t, l, tmp2);
46                  SetRow(t, tmp2, i);
47                  SetRow(t, tmp1, l);
48                  DestroyVector(tmp1);
49                  DestroyVector(tmp2);
50                  i := i - m;
51                  IF i >= 1 THEN GOTO 10;
52              END;
53          END;
54      IF (NaNs > 0)
55      THEN
56          FOR i := Succ(n - NaNs) TO n DO // change the top NaNs elements back
57              TO NaN
58                  SetMatrixElement(t, i, Column, NaN);
59  END; { unit Matrix }
```

## 6.7. Solving systems of linear equations

The system

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1p}x_p = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2p}x_p = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{np}x_p = b_n \end{cases} \quad (6.21)$$

with known coefficients  $\mathcal{A}$ , known right hand side  $b_i$  and unknown solutions  $x$  can be written in matrix terminology:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1p} \\ a_{21} & a_{22} & \dots & a_{2p} \\ \dots \\ a_{n1} & a_{n2} & \dots & a_{np} \end{pmatrix} \times \begin{pmatrix} x_1 & x_2 & \dots & x_p \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}, \quad \mathcal{A}_{n \times p} \times \mathfrak{X}_p = \mathfrak{B}_n \quad (6.22)$$

There should be as many equations as unknown ( $n = p$ ), thus  $\mathcal{A}$  is square [1–7].

Listing 6.35: Interface of SystemSolve

```

1 UNIT SystemSolve;
2
```

```

3 INTERFACE
4
5 USES Math, MathFunc, Vector, Matrix;
6
7 CONST
8   SystemError: StrArrayType = ('ok', 'dimension error', 'matrix singular',
9     'MaxIter < 1', 'no conversion', 'evaluation short-circuited',
10    '', 'matrix not symmetric', 'not enough memory', '', '', '');
11
12 FUNCTION MakeUpperTriangular(VAR Koeff : MatrixTyp; VAR Right : VectorTyp)
13   : BYTE;
14
15 FUNCTION HadamardConditionNumber (VAR Mat : MatrixTyp) : double;
16
17 FUNCTION GaussElimination(CONST Coefficients: MatrixTyp;
18   VAR RightSide, Solution: VectorTyp): BYTE;
19
20 FUNCTION PartialPivoting(CONST Coefficients: MatrixTyp;
21   VAR RightSide, Solution: VectorTyp): BYTE;
22
23 FUNCTION LU_Decompose(CONST Coefficients: MatrixTyp;
24   VAR Decomp, Permute: MatrixTyp): BYTE;
25
26 FUNCTION LU_Solve(CONST Decomp, Permute: MatrixTyp; CONST RightSide:
27   VectorTyp;
28   VAR Solution: VectorTyp): BYTE;
29
30 IMPLEMENTATION
31
32 VAR CH  : CHAR;

```

## 6.7.1. Private routines

```

1 FUNCTION MakeUpperTriangular(VAR Koeff : MatrixTyp; VAR Right : VectorTyp)
2   : BYTE;
3
4 VAR Multiplier           : double;
5   Row, ReferenceRow, Dimen : WORD;
6
7 FUNCTION Pivot(ReferenceRow: INTEGER) : BYTE;
8   { This procedure searches the ReferenceRow column of the
     Coefficients
     matrix for the first non-MaxError element below the diagonal. If
     it

```

## 6. Matrix calculations

```
9      finds one, then the procedure switches rows so that the
10     non-MaxError
11     element is on the diagonal. It also switches the corresponding
12     elements in the Constants vector. If it doesn't find one, the
13     matrix is singular and no solution exists (Error = 2 is
14     returned). }
15
16
17  VAR NewRow : INTEGER;
18      Dummy   : double;
19
20  BEGIN
21      Result := 2;           { No solution exists }
22      NewRow := ReferenceRow;
23      WHILE (Result > 0) AND (NewRow < Dimen) DO
24          BEGIN { Try to find a row with a non-MaxError diagonal element }
25              NewRow := Succ(NewRow);
26              IF Abs(GetMatrixElement(Koeff, NewRow, ReferenceRow)) > MaxError
27                  THEN
28                      BEGIN
29                          ExchangeRows(Koeff, NewRow, ReferenceRow);
30                          Dummy := GetVectorElement(Right, NewRow);
31                          SetVectorElement(Right, NewRow, GetVectorElement(Right,
32                                              ReferenceRow));
33                          SetVectorElement(Right, ReferenceRow, Dummy);
34                          Result := 0;    { Solution may exist }
35                      END;
36          END;
37      END; { procedure Pivot }

38
39  BEGIN { procedure UpperTriangular }
40      Dimen := MatrixRows(Koeff);
41      ReferenceRow := 0;
42      Result := 0;
43      WHILE ((Result = 0) AND (ReferenceRow < Pred(Dimen))) DO
44          BEGIN
45              INC(ReferenceRow); { Check to see if the main diagonal element is
46                           MaxError }
47              IF Abs(GetMatrixElement(Koeff, ReferenceRow, ReferenceRow)) < MaxError
48                  THEN Result := Pivot(ReferenceRow);
49              IF Result = 0
50                  THEN
51                      FOR Row := Succ(ReferenceRow) TO Dimen DO
52                          { Make the ReferenceRow element of this row MaxError }
53                          IF Abs(GetMatrixElement(Koeff, Row, ReferenceRow)) > MaxError
54                          THEN
55                              BEGIN
```

```

51      Multiplier := -GetMatrixElement(Koeff, Row, ReferenceRow)
52          / GetMatrixElement(Koeff, ReferenceRow, ReferenceRow);
53      EROmultAdd(Koeff, Multiplier, ReferenceRow, Row);
54      SetVectorElement(Right, Row, GetVectorElement(Right, Row)
55          + Multiplier * GetVectorElement(Right, ReferenceRow));
56  END;
57  END; { while }
58 IF Abs(GetMatrixElement(Koeff, MatrixRows(Koeff), MatrixColumns(Koeff)))
59     < MaxError
60 THEN Result := 2; { No solution }
61 END; { procedure UpperTriangular }

1 PROCEDURE BackwardSubst (VAR Koeff : MatrixTyp; VAR Right, Solution :
2     VectorTyp);
3 { This procedure applies backwards substitution to the upper triangular
4     Coefficients matrix and Constants vector. The resulting vector is the
5     solution to the set of equations and is stored in the vector Solution. }
6
7 VAR Term, Row, Dimen : WORD;
8     Sum           : double;

9 BEGIN
10    Dimen := MatrixRows(Koeff);
11    FOR Term := Dimen DOWNTO 1 DO
12        BEGIN
13            Sum := 0;
14            FOR Row := Succ(Term) TO Dimen DO
15                Sum := Sum + GetMatrixElement(Koeff, Term, Row) *
16                    GetVectorElement(Solution, Row);
17            SetVectorElement(Solution, Term, (GetVectorElement(Right, Term) -
18                Sum) / GetMatrixElement(Koeff, Term, Term));
19        END;
20    END; { procedure Backwardssub }

```

Listing 6.36: Test if matrix is suitable

```

1 FUNCTION Initial(CONST Coefficients : MatrixTyp; VAR RightSide, Solution :
2     VectorTyp) : BYTE;
3
4 VAR Dimen : WORD;
5
6 BEGIN
7     Result := 0;
8     Dimen := MatrixRows(Coefficients);
9     IF ((Dimen < 1) OR (Dimen >> MatrixColumns(Coefficients)) OR (Dimen >>
10         VectorLength(RightSide)))
11     THEN

```

## 6. Matrix calculations

```

10      Result := 1
11  ELSE
12    IF Dimen = 1
13      THEN
14        IF Abs(GetMatrixElement(Coefficients, 1, 1)) < MaxError
15          THEN Result := 2
16          ELSE SetVectorElement(Solution, 1, GetVectorElement(RightSide,
17                        1) / GetMatrixElement(Coefficients, 1, 1));
18  END; { procedure Initial }
```

### 6.7.2. The condition number

The matrix  $\mathcal{A}$  may be well conditioned, that is, small changes in the elements  $\mathbf{b}$  will produce only small changes in the solution  $\mathbf{x}$ . Ill conditioned matrices, on the other hand, produce large changes in the solution for small changes in  $\mathbf{b}$ . If  $\mathbf{e}$  is the error in  $\mathbf{b}$  and  $\mathcal{A}$  is not singular, then the error of  $\mathcal{A}^{-1}\mathbf{b}$  will be  $\mathcal{A}^{-1}\mathbf{e}$  and the relative error will be

$$\kappa(\mathcal{A}) = \frac{\|\mathcal{A}^{-1}\mathbf{E}\|}{\|\mathbf{E}\|} = \left( \frac{\|\mathcal{A}^{-1}\mathbf{E}\|}{\|\mathbf{E}\|} \right) \left( \frac{\|\mathbf{B}\|}{\|\mathcal{A}^{-1}\mathbf{B}\|} \right) = \|\mathcal{A}^{-1}\| \times \|\mathcal{A}\| \quad (6.23)$$

$\kappa$  is called the **condition number** of  $\mathcal{A}$ . Most commonly, the FROBENIUS-norm is used for its calculation. Note that the condition number is a property of the matrix  $\mathcal{A}$ , not the algorithm used for its solution or the numerical precision of the computer.

Listing 6.37: Condition Number

```

1  FUNCTION HadamardConditionNumber (VAR Mat : MatrixTyp) : double;
2
3  VAR n, i, j           : WORD;
4      res                : BYTE;
5      DecomP, Permute     : MatrixTyp;
6      temp, cond          : double;
7
8  BEGIN
9      n := MatrixRows(Mat);
10     IF NOT(MatrixSymmetric(Mat))
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage(' Hadamard condition number of a
14                               non-symmetric matrix');
15                 Result := NaN;
16                 EXIT;
17             END;
18     res := LU_Decompose(Mat, DecomP, Permute);
19     IF MatrixError
20         THEN
```

```

20      BEGIN
21          Result := NaN;
22          EXIT;
23      END;
24  IF res = 2
25  THEN
26      Result := 0      // singular matrix
27  ELSE
28      BEGIN
29          cond := 1.0;
30          FOR i := 1 TO n DO
31              BEGIN
32                  temp := 0.0;
33                  FOR j := 1 TO n DO
34                      temp := temp + Sqr(GetMatrixElement(Mat, i, j));
35                  cond := cond * GetMatrixElement(Decomp, i, i) / Sqrt(temp);
36              END;
37          Result := Abs(cond);
38      END;
39      DestroyMatrix(Decomp);
40      DestroyMatrix(Permute);
41  END;

```

### 6.7.3. Over- and under-determined systems

If there are more (independent) equations  $n$  than unknowns  $p$  (overdetermined system) we have an error-minimisation (linear least squares) problem. If there are fewer equations than unknowns (or some equations are linear combinations), then there will be  $p - n$  solution vectors (underdetermined system). These situations can be handled with singular value decomposition.

The routines for system solving return the following error codes:

- 0 everything ok
- 1 dimension error (number of vars  $\neq$  number of equations)
- 2 matrix singular
- 3 MaxIter < 0
- 4 MaxIter exceeded
- 5 evaluation short-circuited
- 6
- 7 matrix not symmetric
- 8 not enough memory

### 6.7.4. Example

$$\begin{pmatrix} 1.00 & 2.00 & 0.00 & -1.00 \\ -1.00 & 4.00 & 3.00 & -0.50 \\ 2.00 & 2.00 & 1.00 & -3.00 \\ 0.00 & 0.00 & 3.00 & -4.00 \end{pmatrix} \times \begin{pmatrix} -1.00 & 2.00 & 3.00 & -7.00 \end{pmatrix} = \begin{pmatrix} 10.0 \\ 21.5 \\ 26.0 \\ 37.0 \end{pmatrix} \quad (6.24)$$

### 6.7.5. GAUSSian elimination

The GAUSSian elimination method is used to solve systems of equations. The matrix is converted into the upper triangular form, then the solution is calculated by back-substitution ( $x_m, x_{m-1} \dots x_1$ ). The system has no solution if one or more diagonal elements of the triangular matrix are zero (singular matrix). The method is fast, but sensitive to rounding errors when elements of the diagonal are small compared to elements below them in the same column.

```

1  FUNCTION GaussElimination(CONST Coefficients : MatrixTyp;
2      VAR RightSide, Solution : VectorTyp) : BYTE;
3
4  VAR Error : BYTE;
5      Dimen : WORD;
6      Koeff : MatrixTyp;
7      Right : VectorTyp;
8
9  BEGIN { procedure Gaussian_Elimination }
10     Dimen := MatrixRows(Coefficients);
11     Result := Initial(Coefficients, RightSide, Solution);
12     IF result <> 0 THEN EXIT;
13     CreateVector(Right, Dimen, 0.0);
14     CreateVector(Solution, Dimen, 0.0);
15     CopyVector(RightSide, Right);
16     IF VectorError
17         THEN
18             BEGIN
19                 VectorError := FALSE;
20                 Result := 8;
21                 EXIT;
22             END;
23     CopyMatrix(Coefficients, Koeff);
24     IF MatrixError
25         THEN
26             BEGIN
27                 MatrixError := FALSE;
28                 Result := 8;
29                 EXIT;
30             END;

```

```

31 IF MatrixRows(Koeff) > 1
32 THEN
33 BEGIN
34     Error := MakeUpperTriangular(Koeff, Right);
35     IF Error = 0 THEN BackwardSubst(Koeff, Right, Solution);
36 END;
37 Result := Error;
38 DestroyVector(Right);
39 DestroyMatrix(Koeff);
40 END; { procedure Gaussian_Elimination }
```

### 6.7.6. GAUSS-elimination with partial pivoting

In this method lines of matrix and right side are exchanged to ensure that all diagonal elements are larger than the elements below them. This method is more stable, but slower than the simple GAUSS elimination.

```

1 FUNCTION PartialPivoting(CONST Coefficients: MatrixTyp;
2     VAR RightSide, Solution: VectorTyp): BYTE;
3
4 VAR Error : BYTE;
5     Dimen : WORD;
6     Koeff : MatrixTyp;
7     Right : VectorTyp;
8
9 BEGIN { procedure PartialPivoting }
10    Dimen := MatrixRows(Coefficients);
11    Result := Initial(Coefficients, RightSide, Solution);
12    IF result <> 0 THEN EXIT;
13    CopyMatrix(Coefficients, Koeff);
14    IF MatrixError
15    THEN
16        BEGIN
17            MatrixError := FALSE;
18            Result := 8;
19            EXIT;
20        END;
21    CreateVector(Solution, Dimen, 0.0);
22    CopyVector(RightSide, Right);
23    IF VectorError
24    THEN
25        BEGIN
26            VectorError := FALSE;
27            Result := 8;
28            EXIT;
29        END;
```

## 6. Matrix calculations

```

30  IF Dimen > 1
31  THEN
32  BEGIN
33      Error := MakeUpperTriangular(Koeff, Right);
34      IF Error = 0 THEN BackwardSubst(Koeff, Right, Solution);
35  END;
36  Result := Error;
37  DestroyMatrix(Koeff);
38  DestroyVector(Right);
39 END; { Partial_Pivoting }
```

### 6.7.7. LU-decomposition

The matrix  $\mathcal{A}$  is decomposed into a upper triangular matrix  $\mathcal{U}$  and a lower triangular matrix  $\mathcal{L}$ . Then

$$\mathcal{L} \times \mathcal{U} = \mathcal{A} \quad (6.25)$$

and

$$\mathcal{A} \times \mathbf{x} = (\mathcal{L} \times \mathcal{U}) \times \mathbf{x} = \mathcal{L} \times (\mathcal{U} \times \mathbf{x}) = \mathcal{B} \quad (6.26)$$

Taking advantage of the triangular shape of  $\mathcal{L}$  and  $\mathcal{U}$ , we first solve for  $\mathcal{L}\mathbf{y} = \mathcal{B}$ , then for  $\mathcal{U}\mathbf{x} = \mathbf{y}$ . This algorithm is only  $O(n^2)$ , and once the decomposition has been performed, it can be used to solve for several right hand sides, if desired.

Listing 6.38: LU-decomposition

```

1  FUNCTION LU_Decompose(CONST Coefficients : MatrixTyp; VAR Decomp, Permute :
   MatrixTyp): BYTE;
2
3  VAR Error: BYTE;
4      Koeff, Upper, Lower: MatrixTyp;
5      Dimen: WORD;
6
7
8  FUNCTION RowColumnMult(VAR Lower, Upper: MatrixTyp; Row, Column: WORD):
   double;
9      { Function return: dot product of row Row of Lower and column Column of
   Upper }
10
11  VAR Term : INTEGER;
12      Sum : double;
13
14  BEGIN
15      Sum := 0;
16      FOR Term := 1 TO Pred(Row) DO
17          Sum := Sum + GetMatrixElement(Lower, Row, Term) *
             GetMatrixElement(Upper, Term, Column);
18      Result := Sum;
```

```

19  END; { RowColumnMult }

20

21

22 PROCEDURE Pivot(ReferenceRow: WORD; VAR Error: BYTE);
23   { This procedure searches the ReferenceRow column of the Coefficients
24     matrix for the element in the Row below the main diagonal which
25     produces the largest value of Coefficients[Row, ReferenceRow] - Sum
26     (for K=1 to pred(ReferenceRow) of Upper[Row, k] - Lower[k,
27       ReferenceRow]
28     If it finds one, then the procedure switches rows so that this element
29     is on the main diagonal. The procedure also switches the corresponding
30     elements in the Permute matrix and the Lower matrix. If the largest
31     value of the above expression is MaxError, then the matrix is
32       singular and
33     no solution exists (Error = 2 is returned).    }

34

35

36 BEGIN { procedure Pivot }
37   { First, find the row with the largest TestMax }
38   PivotRow := ReferenceRow;
39   ColumnMax := Abs(GetMatrixElement(Koeff, ReferenceRow, ReferenceRow) -
40     RowColumnMult(Lower, Upper, ReferenceRow, ReferenceRow));
41   FOR Row := Succ(ReferenceRow) TO Dimen DO
42     BEGIN
43       TestMax := Abs(GetMatrixElement(Koeff, Row, ReferenceRow) -
44         RowColumnMult(Lower, Upper, Row, ReferenceRow));
45       IF TestMax > ColumnMax
46         THEN
47           BEGIN
48             PivotRow := Row;
49             ColumnMax := TestMax;
50           END;
51       IF PivotRow <> ReferenceRow
52         THEN { Second, switch these two rows }
53           BEGIN
54             ExchangeRows(Koeff, PivotRow, ReferenceRow);
55             ExchangeRows(Lower, PivotRow, ReferenceRow);
56             ExchangeRows(Permute, PivotRow, ReferenceRow);
57           END
58       ELSE { If ColumnMax is MaxError, no solution exists }
59         IF ColumnMax < MaxError
60           THEN Error := 2;
61   END; { procedure Pivot }

```

## 6. Matrix calculations

```
62
63
64 PROCEDURE Decompose(VAR Error: BYTE);
65 { This procedure decomposes the Coefficients matrix into two triangular
66 matrices, a lower and an upper one. The lower and upper matrices are
67 combined into one matrix, Decomp. The permutation matrix, Permute,
68 records the effects of partial pivoting. }
69
70 VAR Term, Index: INTEGER;
71
72 PROCEDURE Initialize;
73 { This procedure initializes Lower and Upper to the MaxError
74 matrix
75 and Permute to the identity matrix. }
76
77 BEGIN
78     CreateMatrix(Upper, Dimen, Dimen, 0.0);
79     CreateMatrix(Lower, Dimen, Dimen, 0.0);
80     CreateIdentityMatrix(Permute, Dimen);
81     IF MatrixError
82     THEN
83         BEGIN
84             MatrixError := FALSE;
85             Error := 8;
86         END;
87     END; { procedure Initialize }
88
89 BEGIN { Decompose }
90     Initialize;
91     { partial pivoting on row 1 }
92     Pivot(1, Error);
93     IF Error = 0
94     THEN
95         BEGIN
96             SetMatrixElement(Lower, 1, 1, 1);
97             SetMatrixElement(Upper, 1, 1, GetMatrixElement(Koeff, 1, 1));
98             FOR Term := 1 TO Dimen DO
99                 BEGIN
100                     SetMatrixElement(Lower, Term, 1, GetMatrixElement(Koeff,
101                         Term, 1) / GetMatrixElement(Upper, 1, 1));
102                     SetMatrixElement(Upper, 1, Term, GetMatrixElement(Koeff, 1,
103                         Term) / GetMatrixElement(Lower, 1, 1));
104                 END;
105             Term := 1;
106             WHILE (Error = 0) AND (Term < Pred(Dimen)) DO
```

```

105   BEGIN
106     Term := Succ(Term); { perform partial pivoting on row Term }
107     Pivot(Term, Error);
108     SetMatrixElement(Lower, Term, Term, 1);
109     SetMatrixElement(Upper, Term, Term,
110       GetMatrixElement(Koeff, Term, Term) - RowColumnMult(Lower,
111         Upper, term, term));
112     IF Abs(GetMatrixElement(Upper, Term, Term)) < MaxError
113       THEN
114         Error := 2 { no solutions }
115       ELSE
116         FOR Index := Succ(Term) TO Dimen DO
117           BEGIN
118             SetMatrixElement(Upper, Term, Index,
119               GetMatrixElement(Koeff, Term, Index) -
120                 RowColumnMult(Lower, Upper, Term, Index));
121             SetMatrixElement(Lower, Index, Term,
122               (GetMatrixElement(Koeff, Index, Term) -
123                 RowColumnMult(Lower, Upper, Index, Term)) /
124                   GetMatrixElement(Upper, Term, Term));
125           END;
126         END;
127         SetMatrixElement(Lower, Dimen, Dimen, 1);
128         SetMatrixElement(Upper, Dimen, Dimen, GetMatrixElement(Koeff, Dimen,
129           Dimen) - RowColumnMult(Lower, Upper, Dimen, Dimen));
130         IF Abs(GetMatrixElement(Upper, Dimen, Dimen)) < MaxError THEN Error :=
131           2;
132         { Combine the upper and lower triangular matrices into one }
133         CopyMatrix(Upper, Decomp);
134         FOR Term := 2 TO Dimen DO
135           FOR Index := 1 TO Pred(Term) DO
136             SetMatrixElement(Decomp, Term, Index, GetMatrixElement(Lower, Term,
137               Index));
138             DestroyMatrix(Upper);
139             DestroyMatrix(Lower);
140           END; { procedure Decompose }

141           BEGIN { LU_Decompose }
142             Dimen := MatrixRows(Coefficients);
143             CopyMatrix(Coefficients, Koeff);
144             IF MatrixError
145               THEN
146                 BEGIN
147                   MatrixError := FALSE;
148                   Result := 8;
149                   EXIT;

```

## 6. Matrix calculations

```

143      END;
144  IF Dimen < 1
145    THEN
146      Error := 1
147  ELSE
148    BEGIN
149      Error := 0;
150      IF Dimen = 1
151        THEN
152          BEGIN
153            CopyMatrix(Koeff, Decom);
154            SetMatrixElement(Permute, 1, 1, 1);
155          END
156        ELSE
157          Decompose(Error);
158        END;
159      Result := Error;
160      DestroyMatrix(Koeff);
161  END; { LU_Decompose }
```

Listing 6.39: Solve system

```

1  FUNCTION LU_Solve(CONST Decom, Permute : MatrixTyp; CONST RightSide :
2   VectorTyp;
3   VAR Solution : VectorTyp): BYTE;
4
5   VAR Error      : BYTE;
6   Dimen       : WORD;
7   DEC, Per : MatrixTyp;
8   Right       : VectorTyp;
9
10  PROCEDURE FindSolution;
11  { The Decom matrix contains a lower and upper triangular matrix. This
12    procedure performs a two step backwards substitution to compute the
13    solution to the system of equations. First, forward substitution is
14    applied to the lower triangular matrix and Constants vector yielding
15    PartialSolution. Then backwards substitution is applied to the Upper
16    matrix and the PartialSolution vector yielding Solution. }
17
18  VAR PartialSolution : Vectortyp;
19  Term, Index       : WORD;
20  Sum               : double;
21
22  BEGIN { FindSolution }
23  { First solve the lower triangular matrix }
24  CreateVector(PartialSolution, Dimen, 0.0);
```

```

25 SetVectorElement(PartialSolution, 1, GetVectorElement(Right, 1));
26 FOR Term := 2 TO Dimen DO
27 BEGIN
28   Sum := 0;
29   FOR Index := 1 TO Pred(Term) DO
30     IF Term = Index
31     THEN
32       Sum := Sum + GetVectorElement(PartialSolution, Index)
33     ELSE
34       Sum := Sum + GetMatrixElement(DEC, Term, Index) *
35         GetVectorElement(PartialSolution, Index);
36   SetVectorElement(PartialSolution, Term, GetVectorElement(Right,
37     Term) - Sum);
38 END;
39 { Then solve the upper triangular matrix }
40 SetVectorElement(Solution, Dimen,
41   GetVectorElement(PartialSolution, Dimen) / GetMatrixElement(DEC,
42     Dimen, Dimen));
43 FOR Term := Pred(Dimen) DOWNTO 1 DO
44 BEGIN
45   Sum := 0;
46   FOR Index := Succ(Term) TO Dimen DO
47     Sum := Sum + GetMatrixElement(DEC, Term, Index) *
48       GetVectorElement(Solution, Index);
49   SetVectorElement(Solution, Term, (GetVectorElement(PartialSolution,
50     Term) - Sum) / GetMatrixElement(DEC, Term, Term));
51 END;
52 DestroyVector(PartialSolution);
53 END; { procedure FindSolution }

54
55
56 PROCEDURE PermuteRightSide;
57
58 VAR Row, Column : WORD;
59   Entry : double;
60   TempConstants : VectorTyp;
61
62 BEGIN
63   CreateVector(TempConstants, Dimen, 0.0);
64   FOR Row := 1 TO Dimen DO
65     BEGIN
66       Entry := 0;
67       FOR Column := 1 TO Dimen DO
68         Entry := Entry + GetMatrixElement(Per, Row, Column) *
69           GetVectorElement(Right, Column);
70       SetVectorElement(TempConstants, Row, Entry);
71     END;
72   END;
73 END;

```

## 6. Matrix calculations

```

64      END;
65      CopyVector(TempConstants, Right);
66      DestroyVector(TempConstants);
67  END; { PermuteRightSide }

68
69 BEGIN { Solve_LU_Decomposition }
70     Dimen := MatrixRows(Decomp);
71     CopyMatrix(Decomp, DEC);
72     CopyMatrix(Permute, Per);
73     CreateVector(Solution, Dimen, 0.0);
74     IF MatrixError
75     THEN
76         BEGIN
77             MatrixError := FALSE;
78             Result := 8;
79             EXIT;
80         END;
81     CopyVector(RightSide, Right);
82     IF VectorError
83     THEN
84         BEGIN
85             VectorError := FALSE;
86             Result := 8;
87             EXIT;
88         END;
89     IF Dimen < 1
90     THEN
91         Error := 1
92     ELSE
93         BEGIN
94             Error := 0;
95             PermuteRightSide;
96             FindSolution;
97         END;
98     Result := Error;
99     DestroyMatrix(DEC);
100    DestroyMatrix(Per);
101 END; { procedure LU_Solve }
```

### 6.7.8. The QR-factorisation

The QR-factorisation decomposes a  $\mathcal{A}_{n \times p}$ -matrix ( $n \geq p$ ) into the product of an unitary (orthogonal if  $\mathcal{A}$  is square, real and of full rank) matrix  $\mathcal{Q}_{n \times n}$  (i.e.,  $\mathcal{Q}\mathcal{Q}^T = \mathcal{Q}^T\mathcal{Q} = \mathcal{I}$ ) and an upper triangular matrix  $\mathcal{R}_{n \times p}$ :

$$\mathcal{A} = \mathcal{Q}\mathcal{R} \quad (6.27)$$

The bottom  $(n - p)$  rows of  $\mathcal{R}$  consist entirely of zeroes. Then

$$\mathcal{A} = \mathcal{Q}\mathcal{R} = \mathcal{Q} \begin{pmatrix} \hat{\mathcal{R}} \\ \mathbf{0} \end{pmatrix} \quad (6.28)$$

The upper triangular matrix  $\hat{\mathcal{R}}_{p \times p}$  is regular with diagonal elements  $\hat{\mathcal{R}}_{ii} \neq 0$  if  $\text{rk}(\mathcal{A}) = p$  (see below).

The factorisation is used to calculate linear least-square and eigenvalue problems. Also, since  $\det(\mathcal{A}) = \det(\mathcal{Q}) * \det(\mathcal{R})$  and  $\det(\mathcal{Q}) = \pm 1$ ,  $|\det(\mathcal{A})| = |\det(\mathcal{R})| = |\prod_{i=1}^p r_{ii}| = |\prod_{i=1}^p \sigma_{ii}|$  (or  $|\prod_{i=1}^p \lambda_{ii}|$  if  $\mathcal{A}$  is square). Thus, QR-factorisation can be used to cheaply calculate the products of singular or eigenvalues of  $\mathcal{A}$ . Because  $\mathcal{Q}\mathcal{R}$  have better condition numbers than  $\mathcal{A}$ , this factorisation is also used for linear inverse problems.

It is also possible to work with a lower triangular matrix  $\mathcal{L}$ , or to exchange the matrices (RQ-, QL- and LQ-factorisation).

There are three important methods for QR-factorisation:

GRAM–SCHMIDT **process** easy to implement, but numerically unstable

HOUSEHOLDER **reflection** medium complexity, stable, not parallelisable

GIVENS **rotation** complex to implement, stable, parallelisable

Since we need a stable algorithm, but haven't worried about parallelisation with any of the other routines, we'll use HOUSEHOLDER reflection.

### HOUSEHOLDER **reflection**

Multiplication of a vector, for example the  $k$ -th column vector of  $\mathcal{A}$ ,  $\mathbf{a}_{:,k}$ , with a HOUSEHOLDER matrix

$$\mathcal{H} = \mathcal{I} - \frac{2}{\mathbf{u}^T \odot \mathbf{u}} \mathbf{u} \otimes \mathbf{u}^T \quad (6.29)$$

reflects this vector by a plane defined by its normal  $\mathbf{u}$ . Note that  $\mathbf{u} \otimes \mathbf{u}^T$  is the outer (dyadic) product, and hence a matrix, while  $\mathbf{u}^T \odot \mathbf{u}$  is the inner (dot) product of the vectors and hence a scalar.  $\frac{2}{\mathbf{u}^T \odot \mathbf{u}} = \frac{1}{\|\mathbf{u}\|_2(\|\mathbf{u}\|_2+1)}$ .

For the first column vector of  $\mathcal{A}$  we use  $\mathbf{u} = \mathbf{a}_{:,1} - \alpha \mathbf{e}$  with the first standard basis vector of length  $n$ ,  $\mathbf{e} = (1, 0, \dots, 0)^T$  (only the first element is 1, the rest 0) and  $\alpha = \|\mathbf{a}_{:,1}\|_2$  the EUKLIDIAN norm to set all elements after the first to zero. To avoid cancellation errors, we modify this equation by the opposite sign of the pivot element  $\mathcal{A}_{k,k}$ :

$$\mathbf{u} = \mathbf{a}_{:,1} - \text{sgn}(\mathbf{a}_{1,1})\alpha \mathbf{e}_1 \quad (6.30)$$

Then we normalise this vector by its first element:

$$\mathbf{v} = \frac{\mathbf{u}}{\mathbf{u}_1} \quad (6.31)$$

## 6. Matrix calculations

Then the first HOUSEHOLDER-matrix becomes:

$${}^1\mathcal{H} = \mathcal{I} - \frac{2}{\mathbf{v}^T \odot \mathbf{v}} \mathbf{v} \otimes \mathbf{v}^T = \mathcal{I} - \beta \mathbf{v} \otimes \mathbf{v}^T \quad (6.32)$$

The first step zeros all subdiagonal elements of the first column:

$${}^1\mathcal{H}\mathcal{A} = \begin{pmatrix} r_{1,1} & r_{1,2} & \dots & r_{1,p} \\ 0 & * & \dots & * \\ \vdots & * & \dots & * \\ 0 & * & \dots & * \end{pmatrix} \quad (6.33)$$

This process is repeated in the second step on the submatrix  $\mathcal{A}'$  (the elements marked with a star above), resulting in a HOUSEHOLDER-matrix  ${}^2\mathcal{H}$ . Since we really want to apply this matrix on  $\mathcal{A}$  rather than  $\mathcal{A}'$ , we need to embed into an identity matrix  $\mathcal{I}_{n \times p}$ :

$${}^2\mathcal{H} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & {}^2\mathcal{H}' & & \\ \vdots & & & \\ 0 & & & \end{pmatrix} \quad (6.34)$$

Thus, in  $\min(n-1, p)$  operations we can remove all subdiagonal elements of  $\mathcal{A}$ , turning it into  $\mathcal{R}$ :

$$\mathcal{R} = {}^n\mathcal{H} \dots {}^3\mathcal{H} {}^2\mathcal{H} {}^1\mathcal{H} \mathcal{A} \quad (6.35)$$

$$\mathcal{Q} = {}^1\mathcal{H} {}^2\mathcal{H} {}^3\mathcal{H} \dots {}^n\mathcal{H} \quad (6.36)$$

This algorithm is **O** ( $n^3$ ) and described in detail on [11, 12].

Example: the HILBERT-matrix with  $n = 4$  gives

$$\mathcal{A} = \begin{pmatrix} 1.0000 & 0.5000 & 0.3333 & 0.2500 \\ 0.5000 & 0.3333 & 0.2500 & 0.2000 \\ 0.3333 & 0.2500 & 0.2000 & 0.1667 \\ 0.2500 & 0.2000 & 0.1667 & 0.1429 \end{pmatrix} \quad (6.37)$$

$$\mathcal{R} = \begin{pmatrix} 1.193 & 0.6705 & 0.4749 & 0.3698 \\ 0.0000 & -0.1185 & -0.1257 & -0.1175 \\ 0.0000 & 0.0000 & 0.0062 & 0.0096 \\ 0.0000 & 0.0000 & 0.0000 & -0.0002 \end{pmatrix} \quad (6.38)$$

$$\mathcal{Q} = \begin{pmatrix} 0.8381 & 0.5226 & -0.1540 & 0.0261 \\ 0.4191 & -0.4417 & 0.7278 & -0.3157 \\ 0.2794 & -0.5288 & -0.1395 & 0.7892 \\ 0.2095 & -0.5021 & -0.6536 & -0.5261 \end{pmatrix} \quad (6.39)$$

The maximum difference between the original HILBERT matrix and the product  $\mathcal{QR}$  is  $1 \times 10^{-16}$  when the calculation is performed in double precision.

Listing 6.40: QR-factorisation by Householder

```

1  FUNCTION Householder (CONST A : MatrixTyp; VAR Q, R : MatrixTyp) : BYTE;
2
3  VAR Rows, Columns,
4      i, j, k           : WORD;
5      c                 : CHAR;
6      alpha, beta       : double;
7      x, u, e           : VectorTyp;
8      Identity, Product,
9      H, Hs             : MatrixTyp;
10
11 BEGIN
12     Rows := MatrixRows(A);
13     Columns := MatrixColumns(A);
14     IF Rows < Columns
15         THEN
16             BEGIN
17                 c := WriteErrorMessage('QR decomposition of matrix with more
18                               columns than rows');
19                 MatrixError := TRUE;
20                 Householder := 1; // dimension error
21                 EXIT;
22             END;
23             CreateIdentityMatrix(Q, Rows); // neutral element of
24             // matrix multiplication
25             IF MatrixError THEN EXIT;
26             CopyMatrix(A, R); // so that A IS unchanged
27             IF MatrixError THEN EXIT;
28             FOR j := 1 TO min(Pred(Rows), Columns) DO // calculate j-th
29                 Householder Matrix
30                 BEGIN
31                     CreateVector(x, Succ(Rows-j), 0.0);
32                     FOR i := j TO Rows DO
33                         BEGIN
34                             k := Succ(i-j);
35                             SetVectorElement(x, k, GetMatrixElement(R, i, j));
36                         END;
37                         alpha := -Signum(GetVectorElement(x, 1)) * VectorEuklidianNorm(x);
38                         CreateVector(e, Succ(Rows-j), 0.0);
39                         SetVectorElement(e, 1, alpha);
40                         VectorAdd(x, e, u); // u = x + sgn(x_1) *
41                         // ||x||_2 * (1, 0, ..., 0)
42                         DestroyVector(x);
43                         DestroyVector(e);
44                         DivConstant(u, Abs(GetVectorElement(u, 1))); // scale u by Abs(first
45                         element) -> v

```

## 6. Matrix calculations

```

41      beta := 2 / VectorInnerProduct(u, u);           // \beta = 2 / (v \odot v)
42      DyadicVectorProduct(u, u, Product);           // v \otimes v
43      DestroyVector(u);
44      SkalarMultiplikation(Product, -beta);         // beta * (v\otimes v)
45      CreateIdentityMatrix(Identity, Succ(Rows-j));
46      MatrixAdd(Identity, Product, hs);             // Hs = I - beta * (v\otimes v)
47      DestroyMatrix(Identity);
48      DestroyMatrix(Product);
49      CreateIdentityMatrix(H, Rows);
50      FOR i := j TO Rows DO                         // Copy Hs into lower
51          right corner OF H
52          FOR k := j TO Columns DO
53              SetMatrixElement(H, i, k, GetMatrixElement(Hs, i-Pred(j),
54                                              k-Pred(j)));
55          MatrixInnerProduct(H, R, Product);           // R = HR
56          DestroyMatrix(R);
57          CopyMatrix(Product, R);
58          DestroyMatrix(Product);
59          MatrixInnerProduct(Q, H, Product);           // Q = QH
60          DestroyMatrix(Q);
61          CopyMatrix(Product, Q);
62          DestroyMatrix(Product);
63      END;
64      Result := 0;                                  // everything ok
65
66 END.    // SystemSolve

```

### 6.7.9. Matrix rank

The column rank of a matrix  $\text{rk}(\mathcal{A}_{n \times p})$ ,  $n \geq p$  is the number of linearly independent columns, the row rank the number of independent rows. Both are always equal, and at most the smaller of  $n$  and  $p$ :  $\text{rk}(\mathcal{A}_{n \times p}) \leq \min(n, p)$ . For example,

$$\text{rk} \begin{pmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{pmatrix} = 2 \quad (6.40)$$

because  $\alpha_{3.} = \alpha_{1.} - \alpha_{2.}$ . The rank can be determined from the number of singular values larger than a certain threshold, if there is a noticeable gap in the singular value spectrum of  $\mathcal{M}$ . The matrix rank is also the number of positive eigenvalues of a square matrix.

An alternative method to determine rank is to perform a QR- (or LU-) decomposition of the matrix and then identify the number of rows with row-sums  $> 0$ . The procedure

is sensitive to rounding errors, **strong rank-revealing QR- (or LU-) decomposition** algorithms need to be used. If  $\text{rk}(\mathcal{A}_{n \times p}) \approx p$  we speak of a high-rank matrix, if  $\text{rk}(\mathcal{A}_{n \times p}) \approx 1$  of a low rank matrix. Intermediate cases with  $\text{rk}(\mathcal{A}_{n \times p}) \approx p/2$  cause numerical trouble. Algorithms are given in [13, 14], the R-function `qr(matrix)$rank` is *not* rank-revealing!

**Aside: Rank of rectangular matrices** If the rank of a rectangular matrix  $\text{rk}(\mathcal{A}_{n \times p}) = p$ , the matrix would be of full rank, but will have linearly dependent rows if  $n > p$ . Then there must exist a column vector  $\mathbf{c}_p$  such that  $\mathcal{A}\mathbf{c} = \mathbf{0}$ . Although both  $\mathcal{A} \neq \mathbf{0}$  and  $\mathbf{c} \neq \mathbf{0}$ , their product is! Similarly, it is possible to construct rectangular matrices  $\mathcal{A}, \mathcal{B}, \mathcal{C}$  such that  $\mathcal{A}\mathcal{B} = \mathcal{C}\mathcal{B}$  even though  $\mathcal{A} \neq \mathcal{C}$ . Thus, in matrix algebra, it is not possible to cancel matrices from both sides of an equation, except in special cases.

## 6.8. Eigenvalues and eigenvectors

The routines in this unit are based on [1–7]. A square matrix  $\mathcal{A}_{n \times n}$  has  $n$  eigenvalues  $\lambda_i$  that solve the polynomial

$$p(\lambda) = \det(\mathcal{A} - \lambda_i \mathcal{I}) \quad (6.41)$$

If  $\lambda_i$  is an eigenvalue of  $\mathcal{A}$  and  $\mathbf{e} \neq 0$  has the property  $\mathcal{A}\mathbf{e} = \lambda_i\mathbf{e} \rightarrow (\mathcal{A} - \lambda_i \mathcal{I})\mathbf{e} = 0$ , then  $\mathbf{e}$  is called **right eigenvector** of  $\mathcal{A}$ . There also exists a **left eigenvector**  $\mathbf{y}$  so that  $\mathbf{y}^T \mathcal{A} = \lambda \mathbf{y}^T$ . If  $\mathcal{A}$  is real and symmetric, then right and left eigenvectors are identical. In the following, we will use “eigenvector” for “right eigenvector”.

If  $\mathbf{e}$  is an eigenvector of  $\mathcal{A}$ , then  $\alpha\mathbf{e}$  is also an eigenvector. It is therefore common to normalise eigenvectors to  $\|\mathbf{e}\| = 1$ .

### 6.8.1. Real, symmetric matrices

Eigenvalues of a real, symmetric matrix  $\mathcal{A}$  are real, and the eigenvectors corresponding to the eigenvalues are orthogonal – that is,  $\mathbf{e}_i^T \mathbf{e}_j = 0 \quad \forall i \neq j$  – and orthogonal – that is,  $\mathbf{e}_i^T \mathbf{e}_i = 1 \quad \forall i \in \{1..n\}$ . Therefore, for a real, symmetric matrix  $\mathcal{A}_{n \times n}$  there exists a matrix of eigenvectors (in columns)  $\mathcal{E}_{n \times n}$  so that  $\mathcal{E}^{-1} \mathcal{A} \mathcal{E} = \Lambda_{n \times n}$ , the diagonal matrix of eigenvalues. Hence, the matrix  $\mathcal{E} = (e_1, e_2, \dots, e_n)$  containing the normalised eigenvectors in columns is also orthogonal, that is,  $\mathcal{C} \mathcal{C}^T = \mathcal{E}^T \mathcal{E} = \mathcal{I}$ .

$\mathcal{A} = \mathcal{E} \Lambda \mathcal{E}^T$  is called the spectral decomposition of the matrix  $\mathcal{A}$ . From this, it can be shown that  $\mathcal{E}^T \mathcal{A} \mathcal{E} = \Lambda$ .

The **square root matrix**  $\Lambda^{1/2}$  is the diagonal matrix of the square roots of the sorted eigenvalues. Since  $\mathcal{A}^{1/2} = \mathcal{E} \Lambda^{1/2} \mathcal{E}^T$ , it follows that  $\mathcal{A}^{1/2} \mathcal{A}^{1/2} = (\mathcal{A}^{1/2})^2 = \mathcal{A}$  and  $\Lambda^{1/2}$  is the square root of a real, symmetric matrix  $\mathcal{A}$ . The eigenvalues of  $\mathcal{A}^2$  are the squares of the eigenvalues of  $\mathcal{A}$ . If  $\mathcal{A}$  is also non-singular, then the eigenvalues of  $\mathcal{A}^{-1}$  are the reciprocals of the eigenvalues of  $\mathcal{A}$ . The eigenvectors of both  $\mathcal{A}^2$  and  $\mathcal{A}^{-1}$  are the same as those of  $\mathcal{A}$ .

## 6. Matrix calculations

The routines return the following error codes:

- 0 ok
- 1 dimension error
- 2 tolerance  $\leq 0$
- 3 MaxIter < 1
- 4 no convergence
- 5 singular matrix
- 6 matrix not square
- 7 matrix not symmetrical
- 8
- 9 last two roots not real
- 10 not enough memory

Example:

$$\mathcal{A} = \begin{pmatrix} 1.0 & 2.0 & -3.0 & -1.0 \\ 2.0 & 1.0 & -1.0 & -3.0 \\ -3.0 & -1.0 & 1.0 & 2.0 \\ -1.0 & -3.0 & 2.0 & 1.0 \end{pmatrix} \quad \mathcal{E} = \begin{pmatrix} 1.0 & -1.0 & 1.0 & 1.0 \\ 1.0 & 1.0 & 1.0 & -1.0 \\ -1.0 & 1.0 & 1.0 & 1.0 \\ -1.0 & -1.0 & 1.0 & -1.0 \end{pmatrix} \quad \Lambda = \begin{pmatrix} 7.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & -3.0 \end{pmatrix} \quad (6.42)$$

```

1 UNIT EigenValues;
2
3 INTERFACE
4
5 USES MathFunc, Vector, Matrix, SystemSolve;
6
7 CONST EigenError: StrArrayType = ('ok', 'dimension error', 'tolerance < 0',
8     'MaxIter < 1', 'no convergence', 'matrix singular', 'matrix not square',
9     'matrix not symmetrical', 'matrix not positive definite',
10    'last two roots not real',
11    'not enough memory', 'Null-matrix');
12
13 TYPE IntVec = ARRAY[1..MaxIndex] OF WORD; // iterations FOR eigenvalue
14   calculations
15
16 { **** EigenValues
17   **** }
18
19 FUNCTION DominantEigenValue(CONST Mat: MatrixTyp; VAR EigenVector:
20   VectorTyp;
21   VAR EigenValue: double): BYTE;
22
23 FUNCTION NextEigenValue(CONST Mat: MatrixTyp; VAR EigenVector: VectorTyp;
24   VAR EigenValue: double): BYTE;

```

```

24 FUNCTION Jacobi(VAR Mat : MatrixTyp; VAR Eigenvalues : VectorTyp;
25   VAR Eigenvectors : MatrixTyp; VAR Iter : WORD) : BYTE;
26
27 PROCEDURE SortEigenValues(VAR EigenValues: VectorTyp; VAR EigenVectors:
28   MatrixTyp);
29 { Sorts eigenvalues largest first and puts eigenvectors in the same order }
30
31 IMPLEMENTATION
32
33 VAR Iter: WORD;
34   CH : CHAR;

```

### Private routines required for eigenanalysis

Convergenz becomes true when the difference between all elements of OldApprox and NewApprox is smaller than MaxError.

```

1 FUNCTION Convergenz(OldApprox, NewApprox : VectorTyp): BOOLEAN;
2
3   VAR Index : WORD;
4     Found : BOOLEAN;
5
6   BEGIN
7     Index := 0;
8     Found := TRUE;
9     WHILE Found AND (Index < VectorLength(OldApprox)) DO
10    BEGIN
11      INC(Index);
12      IF Abs(GetVectorElement(OldApprox, Index) -
13        GetVectorElement(NewApprox, Index)) > MaxError
14      THEN found := FALSE;
15    END;
16    Result := found;
17  END;

```

TestDataAndInit performs sanity checks and prepares vectors for iteration.

```

1 FUNCTION TestDataAndInit(Mat: MatrixTyp; VAR EigenVector, NewApprox,
2   OldApprox: VectorTyp;
3   VAR EigenValue: double): BYTE;
4
5   VAR Error: BYTE;
6
7   BEGIN
8     Error := 0;
9     IF MatrixRows(Mat) < 1 THEN Error := 1;
10    IF MaxIter < 1 THEN Error := 3;
11    LoadConstant(EigenVector, 1);

```

## 6. Matrix calculations

```

11  IF Error = 0
12  THEN
13  BEGIN
14    Iter := 0;
15    CopyVector(EigenVector, OldApprox);
16    CopyVector(EigenVector, NewApprox);
17  END;
18  IF MatrixRows(Mat) = 1
19  THEN
20  BEGIN
21    EigenValue := GetMatrixElement(Mat, 1, 1);
22    SetVectorElement(EigenVector, 1, 1);
23  END;
24  Result := Error;
25 END; { TestDataAndInit }
```

### Dominant eigenvalue and corresponding eigenvector by inverse vector iteration

The power method approximates the dominant eigenvalue of a matrix. EigenVector contains an estimate of the eigenvector of the largest eigenvalue (in the simplest case, all values = 1). The dominant eigenvalue is the eigenvalue of largest absolute magnitude. Given a square matrix Mat and an arbitrary vector OldApprox, the vector NewApprox is constructed by the matrix operation NewApprox = Mat - OldApprox. NewApprox is divided by its largest element ApproxEigenval, thereby normalizing NewApprox. If NewApprox is the same as OldApprox then ApproxEigenval is the dominant eigenvalue and NewApprox is the associated eigenvector of the matrix Mat. If NewApprox is not the same as OldApprox then OldApprox is set equal to NewApprox and the operation repeats until a solution is reached. AITKEN's  $\delta^2$  acceleration is used to speed the convergence from linear to quadratic.

Application of this method on the inverse of a matrix gives the smallest eigenvalue.

```

1  FUNCTION DominantEigenValue(CONST Mat : MatrixTyp; VAR EigenVector :
2   VectorTyp;
3   VAR EigenValue : double) : BYTE;
4
5  TYPE Hilfsmatrix = ARRAY [0..2, 1..MaxIndex] OF double; { Spart
6   Speicherplatz }
7
8  VAR Remainder, Index, n      : WORD;
9  Error                  : BYTE;
10 ApproxEigenValue, Denom  : double;
11 AitkenVector            : Hilfsmatrix;
12 Gefunden                : BOOLEAN;
13 NewApprox, OldApprox     : VectorTyp;
14
15 BEGIN
```

```

14  n := MatrixRows(Mat);
15  CreateVector(NewApprox, n, 0.0);
16  CreateVector(OldApprox, n, 0.0);
17  IF VectorError
18    THEN
19      BEGIN
20          VectorError := FALSE;
21          Result := 10;
22          EXIT;
23      END;
24  Error := TestDataAndInit(Mat, EigenVector, NewApprox, OldApprox,
25                           ApproxEigenValue);
25  IF Error <> 0
26    THEN
27      BEGIN
28          Result := Error;
29          EXIT;
30      END;
31  IF n = 1
32    THEN
33      BEGIN
34          Result := 0;
35          EigenValue := ApproxEigenValue;
36          EXIT;
37      END;
38  Gefunden := FALSE;
39  FillChar(AitkenVector, SizeOf(AitkenVector), 0);
40  ApproxEigenValue := FindLargest(OldApprox);
41  DivConstant(OldApprox, ApproxEigenValue);
42  WHILE (Iter < MaxIter) AND NOT Gefunden DO
43    BEGIN
44        INC(Iter);
45        Remainder := Iter MOD 3;
46        IF Remainder = 0
47            THEN { Aitken's Beschleunigung }
48            BEGIN
49                FOR Index := 1 TO n DO
50                    SetVectorElement(OldApprox, Index, AitkenVector[0, Index]);
51                FOR Index := 1 TO n DO
52                    BEGIN
53                        Denom := AitkenVector[2, Index] - 2 *
54                        AitkenVector[1, Index] + AitkenVector[0, Index];
55                        IF Abs(Denom) > MaxError
56                            THEN
57                                SetVectorElement(OldApprox, Index, AitkenVector[0,
Index] -

```

## 6. Matrix calculations

```

58                     Sqr(AitkenVector[1, Index] - AitkenVector[0, Index])
59                     / Denom;
60     END; { for }
61     END; {if Remainder }
62     MultMatrixVector(Mat, OldApprox, NewApprox);
63     ApproxEigenValue := FindLargest(NewApprox);
64     IF Abs(ApproxEigenValue) < MaxError
65     THEN
66     BEGIN
67         ApproxEigenValue := 0;
68         Gefunden := TRUE;
69     END
70     ELSE
71     BEGIN
72         DivConstant(NewApprox, ApproxEigenValue);
73         Gefunden := Convergenz(OldApprox, NewApprox);
74         CopyVector(NewApprox, OldApprox);
75     END;
76     FOR Index := 1 TO n DO
77         AitkenVector[Remainder, Index] := GetVectorElement(NewApprox,
78                                         Index);
79     END; { while }
80     EigenValue := ApproxEigenValue;
81     CopyVector(OldApprox, EigenVector);
82     IF Iter >= MaxIter THEN Error := 4;
83     Result := Error;
84     DestroyVector(NewApprox);
85     DestroyVector(OldApprox);
86 END; { DominantEigenValue }
```

### Eigenvalue closest to user-supplied value

Converges onto the eigenvalue closest to a user-supplied value. This method converges fast and can be used to improve results from DominantEigenValue. The linear system  $(\text{Mat} - \text{ClosestVal} - I)\text{OldApprox} = \text{NewApprox}$  is solved via LU decomposition. The vector  $\text{NewApprox}$  is divided by its largest element  $\text{ApproxEigenval}$  (thereby normalizing the vector). If  $\text{NewApprox}$  is identical to  $\text{OldApprox}$  then  $(1/\text{ApproxEigenval} + \text{ClosestVal})$  is the eigenvalue of  $A$  closest to  $\text{ClosestVal}$  and  $\text{OldApprox}$  is the associated eigenvector. If  $\text{NewApprox}$  is not identical to  $\text{OldApprox}$  then  $\text{OldApprox}$  is set equal to  $\text{NewApprox}$  and the process repeats until a solution is reached.

```

1  FUNCTION NextEigenValue(CONST Mat : MatrixTyp; VAR EigenVector : VectorTyp;
2   VAR EigenValue : double) : BYTE;
3
4   VAR i, n           : WORD;
5   Error             : BYTE;
```

```

6      ok, Gefunden          : BOOLEAN;
7      mu                      : double;
8      NewApprox, OldApprox   : VectorTyp;
9      A, Decomp, Permute     : MatrixTyp;
10
11 BEGIN
12   n := MatrixRows(A);
13   CopyMatrix(Mat, A);
14   CreateMatrix(Decomp, n, n, 0.0);
15   CreateMatrix(Permute, n, n, 0.0);
16   CreateVector(NewApprox, n, 0.0);
17   CreateVector(OldApprox, n, 0.0);
18   IF VectorError OR MatrixError
19   THEN
20     BEGIN
21       VectorError := FALSE;
22       MatrixError := FALSE;
23       Result := 8;
24       EXIT;
25     END;
26   Error := TestDataAndInit(A, EigenVector, NewApprox, OldApprox,
27                           EigenValue);
27   IF Error <> 0
28   THEN
29     BEGIN
30       Result := Error;
31       EXIT;
32     END;
33   IF n = 1
34   THEN
35     BEGIN
36       Result := 0;
37       EXIT;
38     END;
39   FOR i := 1 TO n DO
40   BEGIN
41     SetMatrixElement(A, i, i, GetMatrixElement(A, i, i) - EigenValue);
42     SetVectorElement(OldApprox, i, 1);
43   END;
44   IF (LU_Decompose(A, Decomp, Permute) <> 0) OR
45   (LU_Solve(Decomp, Permute, OldApprox, NewApprox) <> 0)
46 { AusValueung kurzgeschlossen! }
47   THEN
48     BEGIN
49       Result := 5;
50       EXIT;

```

## 6. Matrix calculations

```

51      END;
52  REPEAT
53    INC(Iter);
54    mu := 0;
55    FOR i := 1 TO n DO
56      IF (Abs(GetVectorElement(NewApprox, i)) > Abs(mu))
57        THEN mu := SetVectorElement(NewApprox, i);
58    Gefunden := TRUE;
59    DivConstant(NewApprox, mu);           { y normalisieren }
60    Gefunden := Convergenz(NewApprox, OldApprox);
61    CopyVector(NewApprox, OldApprox);
62    IF NOT Gefunden
63      THEN
64        IF Iter >= MaxIter
65          THEN Error := 4
66          ELSE ok := LU_Solve(Decomp, Permute, OldApprox, NewApprox) = 0;
67    UNTIL Gefunden OR (Error <> 0);
68    IF Gefunden
69      THEN Result := 0
70      ELSE Result := Error;
71    EigenValue := EigenValue + 1 / mu;
72    CopyVector(NewApprox, EigenVector);
73    DestroyVector(NewApprox);
74    DestroyVector(OldApprox);
75  END; { NextEigenValue }
```

### All eigenvalues and -vectors of a symmetric matrix by the cyclic JACOBI method

The cyclic Jacobi method is an iterative technique for approximating the complete eigensystem of a symmetric matrix to a given tolerance. The method consists of multiplying the matrix, Mat, by a series of rotation matrices,  $r@-[i]$ . The rotation matrices are chosen so that the elements of the upper triangular part of Mat are systematically annihilated. That is,  $r@-[1]$  is chosen so that Mat[1, 1] becomes less than MaxError;  $r@-[2]$  is chosen so that Mat[1, 2] becomes less than MaxError; etc. Since each operation will probably change the value of elements annihilated in previous operations, the method is iterative. Eventually, the matrix will be diagonal. The eigenvalues will be the elements along the diagonal of the matrix and the eigenvectors will be the rows of the matrix created by the product of all the rotation matrices  $r@-[i]$ . The input matrix is overwritten during the procedure, therefore work with a copy!

```

1  FUNCTION Jacobi(VAR Mat : MatrixTyp; VAR Eigenvalues : VectorTyp;
2                    VAR Eigenvectors : MatrixTyp; VAR Iter : WORD) : BYTE;
3
4  VAR Row, Column, Diag, Dimen      : WORD;
5        SinTheta, CosTheta, SumSquareDiag : double;
6        Done                      : BOOLEAN;
```

```

7
8 FUNCTION TestData(Dimen: WORD; VAR Mat: MatrixTyp; MaxIter: WORD): BYTE;
9   {- This procedure tests the input data for errors. -}
10
11 BEGIN
12   IF (Dimen < 1)
13     THEN
14       BEGIN
15         Result := 1;
16         EXIT;
17       END;
18   IF NOT (MatrixSquare(Mat))
19     THEN
20       BEGIN
21         Result := 6;
22         EXIT;
23       END;
24   IF MaxIter < 1
25     THEN
26       BEGIN
27         Result := 3;
28         EXIT;
29       END;
30   IF NOT (MatrixSymmetric(Mat))
31     THEN
32       BEGIN
33         Result := 7;
34         EXIT;
35       END;
36   Result := 0;
37 END; { procedure TestData }

38
39
40 PROCEDURE CalculateRotation(RowRow, RowCol, ColCol : double; VAR
41   SinTheta, CosTheta: double);
42
43 { This procedure calculates the sine and cosine of the angle Theta
44   through which
45   to rotate the matrix Mat. Given the tangent of 2-Theta, the tangent of
46   Theta
47   can be calculated with the quadratic formula. The cosine and sine are
48   easily
49   calculable from the tangent. The rotation must be such that the Row,
50   Column
51   element is MaxError. RowRow is the Row,Row element; RowCol is the
52   Row,Column element;
53 }
```

## 6. Matrix calculations

```
47     ColCol is the Column,Column element of Mat. }
48
49     VAR TangentTwoTheta, TangentTheta, Dummy: double;
50
51     BEGIN
52         IF Abs(RowRow - ColCol) > MaxError
53             THEN
54                 BEGIN
55                     TangentTwoTheta := (RowRow - ColCol) / (2 * RowCol);
56                     Dummy := Sqr(Sqr(TangentTwoTheta) + 1);
57                     IF TangentTwoTheta < 0
58                         THEN TangentTheta := -TangentTwoTheta - Dummy
59                         ELSE TangentTheta := -TangentTwoTheta + Dummy;
60                     CosTheta := 1 / Sqr(1 + Sqr(TangentTheta));
61                     SinTheta := CosTheta * TangentTheta;
62                 END
63             ELSE
64                 BEGIN
65                     CosTheta := Sqr(1 / 2);
66                     IF RowCol < 0
67                         THEN SinTheta := -Sqr(1 / 2)
68                         ELSE SinTheta := Sqr(1 / 2);
69                 END;
70             END; { procedure CalculateRotation }

71
72     PROCEDURE RotateMatrix(SinTheta, CosTheta: double; Row: WORD; Col: WORD;
73         VAR Mat: MatrixTyp);
74     { This procedure rotates the matrix Mat through an angle Theta. The
      rotation
      matrix is the identity matrix except for the Row,Row; Row,Col;
      Col,Col; and
      Col,Row elements. The rotation will make the Row,Col element of Mat to
      be
      MaxError. }

75
76     VAR CosSqr, SinSqr, SinCos, MatRowRow, MatColCol,
77         MatRowCol, MatRowIndex, MatColIndex           : double;
78         Index                                         : WORD;

79     BEGIN
80         CosSqr := Sqr(CosTheta);
81         SinSqr := Sqr(SinTheta);
82         SinCos := SinTheta * CosTheta;
83         MatRowRow := GetMatrixElement(Mat, Row, Row) * CosSqr + 2 *
84             GetMatrixElement(Mat, Row, Col) * SinCos + GetMatrixElement(Mat, Col,
85             Col) * SinSqr;
```

```

89      MatColCol := GetMatrixElement(Mat, Row, Row) * SinSqr - 2 *
90      GetMatrixElement(Mat, Row, Col) * SinCos + GetMatrixElement(Mat, Col,
91      Col) * CosSqr;
92      MatRowCol := (GetMatrixElement(Mat, Col, Col) -
93      GetMatrixElement(Mat, Row, Row)) * SinCos + GetMatrixElement(Mat,
94      Row, Col) * (CosSqr - SinSqr);
95      FOR Index := 1 TO Dimen DO
96      IF NOT (Index IN [Row, Col])
97      THEN
98      BEGIN
99          MatRowIndex := GetMatrixElement(Mat, Row, Index) * CosTheta +
100         GetMatrixElement(Mat, Col, Index) * SinTheta;
101         MatColIndex := -GetMatrixElement(Mat, Row, Index) *
102         SinTheta + GetMatrixElement(Mat, Col, Index) * CosTheta;
103         SetMatrixElement(Mat, Row, Index, MatRowIndex);
104         SetMatrixElement(Mat, Index, Row, MatRowIndex);
105         SetMatrixElement(Mat, Col, Index, MatColIndex);
106         SetMatrixElement(Mat, Index, Col, MatColIndex);
107     END;
108     SetMatrixElement(Mat, Row, Row, MatRowRow);
109     SetMatrixElement(Mat, Col, Col, MatColCol);
110     SetMatrixElement(Mat, Row, Col, MatRowCol);
111     SetMatrixElement(Mat, Col, Row, MatRowCol);
112 END; { procedure RotateMatrix }

113 PROCEDURE RotateEigenvectors(SinTheta, CosTheta: double; Row, Col: WORD;
114 VAR Eigenvectors: MatrixTyp);
115 {- This procedure rotates the Eigenvectors matrix through an angle Theta.
116   The
117   rotation matrix is the identity matrix except for the Row,Row; Row,Col;
118   Col,Col;
119   and Col,Row elements. The Eigenvectors matrix will be the product of
120   all the
121   rotation matrices which operate on Mat.  }
122 VAR EigenvectorsRowIndex, EigenvectorsColIndex : double;
123           Index : WORD;
124
125 BEGIN
126 { Transform eigenvector matrix }
127 FOR Index := 1 TO Dimen DO
128 BEGIN
129     EigenvectorsRowIndex := CosTheta * GetMatrixElement(Eigenvectors,
130     Row, Index) +
131     SinTheta * GetMatrixElement(Eigenvectors, Col, Index);

```

## 6. Matrix calculations

```
128     EigenvectorsColIndex := -SinTheta * GetMatrixElement(Eigenvectors,
129         Row, Index) +
130         CosTheta * GetMatrixElement(Eigenvectors, Col, Index);
131     SetMatrixElement(Eigenvectors, Row, Index, EigenvectorsRowIndex);
132     SetMatrixElement(Eigenvectors, Col, Index, EigenvectorsColIndex);
133 END;
134
135 PROCEDURE NormalizeEigenvectors(VAR Eigenvectors : MatrixTyp);
136 {- This procedure normalizes the eigenvectors so that the largest element
137   in each vector is one. -}
138
139 VAR Row      : WORD;
140     Largest   : double;
141     CurrentRow : VectorTyp;
142
143 BEGIN { procedure NormalizeEigenvectors }
144 FOR Row := 1 TO Dimen DO
145 BEGIN
146     GetRow(EigenVectors, Row, CurrentRow);
147     Largest := FindLargest(CurrentRow);
148     DivConstant(CurrentRow, Largest);
149     SetRow(EigenVectors, CurrentRow, Row);
150     DestroyVector(CurrentRow); // was generated IN GetRow
151 END;
152
153 BEGIN { procedure Jacobi }
154 Dimen := MatrixRows(Mat);
155 Result := TestData(Dimen, Mat, MaxIter);
156 IF (Result <> 0) THEN EXIT; // matrix NOT suitable FOR Jacobi
157 Iter := 0;
158 CreateIdentityMatrix(EigenVectors, Dimen);
159 CreateVector(Eigenvalues, Dimen, 0);
160 REPEAT
161     Iter := Succ(Iter);
162     SumSquareDiag := 0;
163     FOR Diag := 1 TO Dimen DO
164         SumSquareDiag := SumSquareDiag + Sqr(GetMatrixElement(Mat, Diag,
165             Diag));
166     Done := TRUE;
167     FOR Row := 1 TO Pred(Dimen) DO
168         FOR Column := Succ(Row) TO Dimen DO
169             IF (Abs(GetMatrixElement(Mat, Row, Column)) > MaxError *
SumSquareDiag)
THEN
```

```

170      BEGIN
171          Done := FALSE;
172          CalculateRotation(GetMatrixElement(Mat, Row, Row),
173              GetMatrixElement(Mat, Row, Column),
174              GetMatrixElement(Mat, Column, Column),
175              SinTheta, CosTheta);
176          RotateMatrix(SinTheta, CosTheta, Row, Column, Mat);
177          RotateEigenvectors(SinTheta, CosTheta, Row, Column,
178              Eigenvectors);
179      END;
180      UNTIL Done OR (Iter > MaxIter);
181      FOR Diag := 1 TO Dimen DO
182          SetVectorElement(Eigenvalues, Diag, GetMatrixElement(Mat, Diag, Diag));
183      IF Iter > MaxIter THEN Result := 4;
184  END; { procedure Jacobi }

```

## Sort eigenvalues

Jacobi returns the eigenvalues not in numerical order, as would often be required for further processing (*i.e.*, PCA). Eigenvalues are returned largest first, the eigenvectors are moved so that their order corresponds to the new order of eigenvalues. The eigenvector is also transformed from row-wise to column-wise.

```

1 PROCEDURE SortEigenValues(VAR EigenValues: VectorTyp; VAR EigenVectors:
2     MatrixTyp);
3
4 VAR n, nMax, i, j : WORD;
5     Max           : double;
6     ValuesInter   : VectorTyp;
7     VectorsInter  : MatrixTyp;
8
9 BEGIN
10    n := VectorLength(EigenValues);
11    CopyVector(EigenValues, ValuesInter);
12    CopyMatrix(EigenVectors, VectorsInter);
13    FOR i := 1 TO n DO
14        BEGIN
15            Max := -MaxRealNumber;           // größten Eigenvalue identifizieren
16            nMax := 0;
17            FOR j := 1 TO n DO
18                IF (GetVectorElement(ValuesInter, j) > Max)
19                THEN
20                    BEGIN
21                        nMax := j;
22                        Max := GetVectorElement(ValuesInter, j);
23                    END;
24                SetVectorElement(EigenValues, i, Max);    // sortierte Kopie erstellen

```

## 6. Matrix calculations

```

24      FOR j := 1 TO n DO
25          SetMatrixElement(EigenVectors, j, i, GetMatrixElement(VectorsInter,
26                           nMax, j));
27          SetVectorElement(ValuesInter, nMax, -MaxRealNumber);
28          // auf sehr kleinen Wert setzen
29      END;
30      DestroyVector(ValuesInter);
31      DestroyMatrix(VectorsInter);
32
33
34END. // Eigenvalues

```

## 6.9. Singular value decomposition (SVD)

### 6.9.1. Introduction

A matrix  $\mathcal{A}_{n \times p}$  ( $n > p$ ) of rank  $k$  can be decomposed according to

$$\mathcal{A}_{n \times p} = \mathcal{U}_{n \times k} \Sigma_{k \times k} \mathcal{V}_{p \times k}^T \quad (6.43)$$

where both  $\mathcal{U}$  and  $\mathcal{V}$  are orthogonal, that is

$$\mathcal{U}^T \mathcal{U} = \mathcal{V} \mathcal{V}^T = \mathcal{I} \quad (6.44)$$

Since  $\mathcal{U}$  is not square, it is orthogonal only in one direction as  $\mathcal{U} \mathcal{U}^T$  is undefined.  $\Sigma$  is a diagonal matrix of singular values (positive square roots of eigenvalues), usually sorted such that  $\sigma_{i+1} \leq \sigma_i$ . The columns of  $\mathcal{U}$  contain the  **$k$  left singular vectors** (normalised eigenvectors of  $\mathcal{A} \mathcal{A}^T$ ) and the  $k$  columns of  $\mathcal{V}$  the **right singular vectors** (normalised eigenvalues of  $\mathcal{A}^T \mathcal{A}$ ) of  $\mathcal{A}$ .

Because of orthogonality, if  $n > p$  the eigenvalue equations

$$\mathcal{A} \mathcal{A}^T \mathcal{U} = \mathcal{U} \Sigma^2 \wedge \mathcal{A}^T \mathcal{A} \mathcal{V} = \mathcal{V} \Sigma^2 \quad (6.45)$$

can be used to find  $\mathcal{V}$  and  $\Sigma^2$  from  $\mathcal{A}^T \mathcal{A}$ , and then  $\mathcal{U} = \mathcal{A} \mathcal{V} \Sigma^{-1}$ . In the case of  $p > n$  calculate the decomposition  $\mathcal{A}^T = \mathcal{V} \Sigma \mathcal{U}^T$ , then swap  $\mathcal{U}$  and  $\mathcal{V}$ . There will be at most  $p$  non-zero singular values in this case.

Singular values and eigenvalues of a matrix are related by  $\sigma_i = \sqrt{\lambda_i}$ .

SVD can be used for

**Noise suppression** Those singular values that are much smaller than the largest can be replaced by zero and then for all  $q \leq p$  non-zero singular values noise-reduced values for the data  $\hat{\mathbf{a}}_{ij} = \sum_{k=1}^q \mathbf{u}_{ik} \sigma_{kk} \mathbf{v}_{jk}$  can be calculated. This can also be used for compression, for example of images.

**Solving linear equations**  $\mathcal{A}\mathbf{x} = \mathbf{b}$  can be rewritten  $\mathbf{x} = \mathcal{V}\Sigma^{-1}\mathcal{U}^T\mathbf{b} = \frac{\mathbf{v}_{ij}}{s_{jj}} \sum_k \mathbf{u}_{ki} \mathbf{b}_k$ . This method is more stable than QR decomposition or Gaussian elimination, if very small  $\sigma$  are first set to zero.

**Inversion of non-square matrices** (pseudo inverse)  $\mathcal{A}^T\mathcal{A}\mathbf{x} = \mathcal{A}^T\mathbf{b}$

SVD is the generalisation of eigenanalysis for non-square matrices:

- Columns of  $\mathcal{V}$  are eigenvectors of  $\mathcal{A}^T\mathcal{A}$ .
- Columns of  $\mathcal{U}$  are eigenvectors of  $\mathcal{A}\mathcal{A}^T$ .
- Diagonals of  $\Sigma$  are square roots of the eigenvalues of both  $\mathcal{A}^T\mathcal{A}$  and  $\mathcal{A}\mathcal{A}^T$ .

## 6.9.2. Implementation

The following routine [1] calculates the SVD of  $\mathcal{A}$ , where  $\mathcal{U}$  overwrites  $\mathcal{A}$ , which therefore has to be copied before calling this routine.  $\mathbf{w}$  is the vector of diagonal elements of  $\Sigma$ .  $\mathcal{V}$  (*not*  $\mathcal{V}^T$ ) is also returned.

Listing 6.41: Interface of unit SingularValue

```

1  UNIT SingularValue;
2
3  INTERFACE
4
5  USES MathFunc, Vector, Matrix;
6
7  PROCEDURE svdcmp(VAR a: MatrixTyp; VAR w: VectorTyp; VAR v: MatrixTyp);
8
9  PROCEDURE svdSynth(VAR A: MatrixTyp; CONST W: VectorTyp; CONST U, V:
10   MatrixTyp);
11 { Berechnet A aus U, W, und V (nicht V^T). }
12
13 IMPLEMENTATION
14
15 VAR CH: CHAR;
```

Listing 6.42: Singular value decomposition

```

1  PROCEDURE svdcmp(VAR a: MatrixTyp; VAR w: VectorTyp; VAR v: MatrixTyp);
2 { Berechnet die Singular Value Decomposition A = U * W * V^T zur
3  m*n Matrix A. Dabei überschreibt U die Matrix A, welche bei Bedarf
4  vor Aufruf der Prozedur kopiert werden muss. Die n*n Diagonalmatrix W
5  wird zurückgegeben als Vector der Diagonal-Elemente. Wichtig: In
6  V steht die n*n Matrix V, nicht V^T. V und W werden in der Routine
7  initialisiert.
8  Literatur: Press et al., Numerical Methods in Pascal, Cambridge 1989 }
```

## 6. Matrix calculations

```
8
9  LABEL 1, 2, 3;
10
11 CONST MaxIter = 50;
12
13 VAR m, n, nm, mnmin, l, k, j, jj, its, i : INTEGER;
14   z, y, x, scale, s, h, g, f, c, anorm, Number : double;
15   rv1 : VectorTyp;
16
17
18 FUNCTION sign(a, b: double): double;
19
20 BEGIN
21   IF (b >= 0.0)
22     THEN Result := Abs(a)
23   ELSE Result := -Abs(a);
24 END;
25
26 BEGIN
27 { Householder reduction to bidiagonal form }
28   m := MatrixRows(A);
29   n := MatrixColumns(A);
30   CreateMatrix(V, n, n, 0.0);
31   CreateVector(W, n, 0.0);
32   CreateVector(rv1, n, 0.0);
33   g := 0.0;
34   scale := 0.0;
35   anorm := 0.0;
36   FOR i := 1 TO n DO
37     BEGIN
38       l := i + 1;
39       SetVectorElement(rv1, i, scale * g);
40       g := 0.0;
41       s := 0.0;
42       scale := 0.0;
43       IF (i <= m)
44         THEN
45           BEGIN
46             FOR k := i TO m DO
47               scale := scale + Abs(GetMatrixElement(A, k, i));
48             IF (scale <> 0.0)
49               THEN
50                 BEGIN
51                   FOR k := i TO m DO
52                     BEGIN
53                       Number := GetMatrixElement(A, k, i) / scale;
```

```

54      SetMatrixElement(A, k, i, Number);
55      s := s + Sqr(Number);
56  END;
57  f := GetMatrixElement(A, i, i);
58  g := -sign(Sqrt(s), f);
59  h := f * g - s;
60  SetMatrixElement(A, i, i, f - g);
61  IF (i <> n)
62  THEN
63    FOR j := l TO n DO
64    BEGIN
65      s := 0.0;
66      FOR k := i TO m DO
67        s := s + GetMatrixElement(A, k, i) *
68          GetMatrixElement(A, k, j);
69      f := s / h;
70      FOR k := i TO m DO
71        SetMatrixElement(A, k, j, GetMatrixElement(A,
72          k, j) +
73          f * GetMatrixElement(A, k, i));
74      END;
75    FOR k := i TO m DO
76      SetMatrixElement(A, k, i, scale * GetMatrixElement(A,
77        k, i));
78    END;
79  SetVectorElement(W, i, scale * g);
80  g := 0.0;
81  s := 0.0;
82  scale := 0.0;
83  IF ((i <= m) AND (i <> n))
84  THEN
85    BEGIN
86      FOR k := l TO n DO
87        scale := scale + Abs(GetMatrixElement(A, i, k));
88      IF (scale <> 0.0)
89      THEN
90        BEGIN
91          FOR k := l TO n DO
92            BEGIN
93              Number := GetMatrixElement(A, i, k) / scale;
94              SetMatrixElement(A, i, k, Number);
95              s := s + Sqr(Number);
96            END;

```

## 6. Matrix calculations

```
97          h := f * g - s;
98          SetMatrixElement(A, i, l, f - g);
99          FOR k := l TO n DO
100             SetVectorElement(rv1, k, GetMatrixElement(A, i, k) / h);
101             IF (i <> m)
102               THEN
103                 FOR j := l TO m DO
104                   BEGIN
105                     s := 0.0;
106                     FOR k := l TO n DO
107                       s := s + GetMatrixElement(A, j, k) *
108                           GetMatrixElement(A, i, k);
109                     FOR k := l TO n DO
110                       SetMatrixElement(A, j, k, GetMatrixElement(A,
111                                         j, k) +
112                                         s * GetVectorElement(rv1, k));
113                     END;
114                   END;
115                 END;
116                 anorm := max(anorm, (Abs(GetVectorElement(W, i)) +
117                               Abs(GetVectorElement(rv1, i))));
118               END;
119               { Accumulation of right-hand transform }
120               FOR i := n DOWNTO 1 DO
121                 BEGIN
122                   IF (i < n)
123                     THEN
124                       BEGIN
125                         IF (g <> 0.0)
126                           THEN
127                             BEGIN
128                               FOR j := l TO n DO
129                                 SetMatrixElement(V, j, i, GetMatrixElement(A, i, j) /
130                                     GetMatrixElement(A, i, l) / g);
131                               FOR j := l TO n DO
132                                 BEGIN
133                                   s := 0.0;
134                                   FOR k := l TO n DO
135                                     s := s + GetMatrixElement(A, i, k) *
136                                       GetMatrixElement(V, k, j);
137                                   FOR k := l TO n DO
138                                     SetMatrixElement(V, k, j, GetMatrixElement(V, k, j)
139                                     +
```

```

137           s * GetMatrixElement(V, k, i));
138       END;
139   END;
140   FOR j := l TO n DO
141     BEGIN
142       SetMatrixElement(V, i, j, 0.0);
143       SetMatrixElement(V, j, i, 0.0);
144     END;
145   END;
146   SetMatrixElement(V, i, i, 1.0);
147   g := GetVectorElement(rv1, i);
148   l := i;
149 END;
{ Accumulation of left-hand transformation }
150 IF m < n
151 THEN mnmin := m
152 ELSE mnmin := n;
153 FOR i := mnmin DOWNTO 1 DO
154   BEGIN
155     l := i + 1;
156     g := GetVectorElement(W, i);
157     IF (i < n)
158       THEN FOR j := l TO n DO SetMatrixElement(A, i, j, 0.0);
159     IF (g <> 0.0)
160       THEN
161         BEGIN
162           g := 1.0 / g;
163           IF (i <> n)
164             THEN
165               BEGIN
166                 FOR j := l TO n DO
167                   BEGIN
168                     s := 0.0;
169                     FOR k := l TO m DO
170                       s := s + GetMatrixElement(A, k, i) *
171                           GetMatrixElement(A, k, j);
172                     f := (s / GetMatrixElement(A, i, i)) * g;
173                     FOR k := i TO m DO
174                       SetMatrixElement(A, k, j, GetMatrixElement(A, k, j) +
175                         f *
176                           GetMatrixElement(A, k, i));
177                     END;
178                   FOR j := i TO m DO
179                     SetMatrixElement(A, j, i, GetMatrixElement(A, j, i) * g);
180                   END;
181                 ELSE
182                   FOR j := i TO m DO SetMatrixElement(A, j, i, 0.0);

```

## 6. Matrix calculations

```

181      SetMatrixElement(A, i, i, GetMatrixElement(A, i, i) + 1.0);
182  END;
183 { Diagonalisation of bidiagonal form }
184 FOR k := n DOWNTO 1 DO           { loop over singular values }
185 BEGIN
186   FOR its := 1 TO MaxIter DO      { loop over allowed iterations }
187   BEGIN
188     FOR l := k DOWNTO 1 DO        { test for splitting }
189     BEGIN
190       nm := l - 1;
191       IF ((Abs(GetVectorElement(rv1, l)) + anorm) = anorm)
192         THEN GOTO 2;
193       IF ((Abs(GetVectorElement(W, nm)) + anorm) = anorm)
194         THEN GOTO 1;
195     END;
196   1:  c := 0.0;                  { Cancellation of rv1[l] if l > 1 }
197   s := 1.0;
198   FOR i := l TO k DO
199   BEGIN
200     f := s * GetVectorElement(rv1, i);
201     IF ((Abs(f) + anorm) <> anorm)
202       THEN
203       BEGIN
204         g := GetVectorElement(W, i);
205         h := Pythag(f, g);
206         SetVectorElement(W, i, h);
207         h := 1.0 / h;
208         c := (g * h);
209         s := -(f * h);
210         FOR j := 1 TO m DO
211         BEGIN
212           y := GetMatrixElement(A, j, nm);
213           z := GetMatrixElement(A, j, i);
214           SetMatrixElement(A, j, nm, (y * c) + (z * s));
215           SetmatrixElement(A, j, i, -(y * s) + (z * c));
216         END;
217       END;
218     END;
219   2:  z := GetVectorElement(W, k);
220   IF (l = k)
221     THEN
222     BEGIN          { Convergence }
223       IF (z < 0.0)    { Singular Value is made non-negative }
224       THEN
225       BEGIN
226         SetVectorElement(W, k, -z);

```

```

227      FOR j := 1 TO n DO
228          SetMatrixElement(V, j, k, -GetMatrixElement(V, j,
229                           k));
230      END;
231      GOTO 3;
232  END;
233  IF (its = MaxIter)
234  THEN
235      BEGIN
236          Writeln('no convergence in ', Maxiter: 2, ' SVDCMP
237              iterations');
238          ReadLn;
239      END;
240      x := GetVectorElement(W, l); { Shift from bottom 2 by 2 minor
241          }
242      nm := k - 1;
243      y := GetVectorElement(W, nm);
244      g := GetVectorElement(rv1, nm);
245      h := GetVectorElement(rv1, k);
246      f := ((y - z) * (y + z) + (g - h) * (g + h)) / (2.0 * h * y);
247      g := Pythag(f, 1.0);
248      f := ((x - z) * (x + z) + h * ((y / (f + sign(g, f))) - h)) / x;
249      { Next QR-transformation }
250      c := 1.0;
251      s := 1.0;
252      FOR j := l TO nm DO
253          BEGIN
254              i := j + 1;
255              g := GetVectorElement(rv1, i);
256              y := GetVectorElement(W, i);
257              h := s * g;
258              g := c * g;
259              z := Pythag(f, h);
260              SetVectorElement(rv1, j, z);
261              c := f / z;
262              s := h / z;
263              f := (x * c) + (g * s);
264              g := -(x * s) + (g * c);
265              h := y * s;
266              y := y * c;
267              FOR jj := 1 TO n DO
268                  BEGIN
269                      x := GetMatrixElement(V, jj, j);
270                      z := GetMatrixElement(V, jj, i);
271                      SetMatrixElement(V, jj, j, (x * c) + (z * s));
272                      SetMatrixElement(V, jj, i, -(x * s) + (z * c));

```

## 6. Matrix calculations

```

270           END;
271           z := Pythag(f, h);
272           SetVectorElement(W, j, z); { Rotation can be arbitrary if
273           z=0 }
274           IF (z <> 0.0)
275           THEN
276               BEGIN
277                   z := 1.0 / z;
278                   c := f * z;
279                   s := h * z;
280               END;
281               f := (c * g) + (s * y);
282               x := -(s * g) + (c * y);
283               FOR jj := 1 TO m DO
284                   BEGIN
285                       y := GetMatrixElement(A, jj, j);
286                       z := GetMatrixElement(A, jj, i);
287                       SetMatrixElement(A, jj, j, (y * c) + (z * s));
288                       SetMatrixElement(A, jj, i, -(y * s) + (z * c));
289                   END;
290               END;
291               SetVectorElement(rv1, l, 0.0);
292               SetVectorElement(rv1, k, f);
293               SetVectorElement(W, k, x);
294           END;
295       END;
296       DestroyVector(rv1);
297   END;

```

The following routine calculates  $\mathcal{A}$  from  $\mathcal{U}$ ,  $\mathcal{V}$  and the diagonal elements of  $\Sigma$  in  $\mathbf{w}$

Listing 6.43: Singular value synthesis

```

1 PROCEDURE svdSynth(VAR A : MatrixTyp; CONST W : VectorTyp; CONST U, V :
2                           MatrixTyp);
3
4 VAR j, k, l, n, p: INTEGER;
5
6 BEGIN
7     n := MatrixRows(U);
8     p := MatrixColumns(U);
9     IF VectorLength(W) <> p
10    THEN
11        BEGIN
12            CH := Matrix.WriteString('Singular value synthesis: Dimension
13                           error');
14        EXIT;
15    END;
16 END;

```

```

13      END;
14  IF ((MatrixRows(V) <> p) OR (MatrixColumns(V) <> p))
15  THEN
16    BEGIN
17      CH := Matrix.WriteErrorMessage('Singular value synthesis: Dimension
18          error');
19      EXIT;
20    END;
21  CreateMatrix(A, n, p, 0.0);
22  FOR k := 1 TO n DO
23    FOR l := 1 TO p DO
24      BEGIN
25        FOR j := 1 TO p DO
26          SetMatrixElement(A, k, l, GetMatrixElement(A, k, l) +
27              GetMatrixElement(U, k, j) * GetVectorElement(W, j) *
28              GetMatrixElement(V, l, j));
29      END;
30  END;
31 END. // SingularValue

```

### 6.9.3. The MOORE-PENROSE pseudoinverse

The pseudoinverse  $\mathcal{X}^+$  of a real or complex matrix  $\mathcal{X}_{n \times p}$  satisfies the MOORE-PENROSE conditions:

- $\mathcal{X}\mathcal{X}^+$  needs not to be the general identity matrix, but it maps all column vectors of  $\mathcal{X}$  to themselves:  $\mathcal{X}\mathcal{X}^+\mathcal{X} = \mathcal{X}$ .
- $\mathcal{X}^+\mathcal{X}\mathcal{X}^+ = \mathcal{X}^+$
- the transpose  $(\mathcal{X}\mathcal{X}^+)^T$  (the HERMITEian (conjugate) transpose  $(\mathcal{X}\mathcal{X}^+)^H$  for complex matrices) is equal to  $\mathcal{X}\mathcal{X}^+$ .
- similarly,  $(\mathcal{X}^+\mathcal{X})^T = \mathcal{X}^+\mathcal{X}$

Pseudoinverses have the following properties:

- $\mathcal{X}^+ \exists \forall \mathcal{X}$  and is unique.
- if  $\mathcal{X}$  is invertible, then  $\mathcal{X}^{-1} = \mathcal{X}^+$ .
- if  $\mathcal{X} = \mathbf{0}$ , then  $\mathcal{X}^+ = \mathcal{X}^T$ .
- $(\mathcal{X}^+)^+ = \mathcal{X}$
- Pseudoinversion commutes with transposition, conjugation, and taking the conjugate transpose.

- $(a\mathcal{X})^+ = a^{-1}\mathcal{X}^+ \quad \forall \quad a \neq 0.$

For a scalar  $x$  the pseudoinverse is

$$x^+ = \begin{cases} 0, & \text{if } x = 0 \\ x^{-1}, & \text{otherwise} \end{cases} \quad (6.46)$$

For a vector  $\mathbf{x}$  the pseudoinverse is

$$\mathbf{x}^+ = \begin{cases} \mathbf{0}^T, & \text{if } \mathbf{x} = \mathbf{0} \\ \frac{\mathbf{x}^T}{\mathbf{x}^T \mathbf{x}}, & \text{otherwise} \end{cases} \quad (6.47)$$

For matrices, when all columns of  $\mathcal{X}$  are linearly independent, so that  $(\mathcal{X}^T \mathcal{X})$  is invertible, then the left inverse of  $\mathcal{X}$  is calculated as  $\mathcal{X}^+ = (\mathcal{X}^T \mathcal{X})^{-1} \mathcal{X}^T$ . If, on the other hand, all rows of  $\mathcal{X}$  are linearly independent ( $(\mathcal{X} \mathcal{X}^T)$  is invertible), then the right inverse of  $\mathcal{X}$  is calculated by  $\mathcal{X}^+ = \mathcal{X}^T (\mathcal{X} \mathcal{X}^T)^{-1}$ .

As a special case, if  $\mathcal{X}$  has orthonormal columns  $(\mathcal{X}^T \mathcal{X}) = \mathcal{I}$  or orthonormal rows  $(\mathcal{X} \mathcal{X}^T) = \mathcal{I}$  then  $\mathcal{X}^+ = \mathcal{X}^T$ .

If  $\mathcal{X}$  is an orthogonal projection matrix ( $\mathcal{X} = \mathcal{X}^T \wedge \mathcal{X}^2 = \mathcal{X}$ ) then  $\mathcal{X}^+ = \mathcal{X}$ .

The pseudoinverse of  $(\mathcal{X} \mathcal{X}^T)$  and  $(\mathcal{X}^T \mathcal{X})$ , often needed in statistics, can be calculated by QR-decomposition:

$$\mathcal{X}^T \mathcal{X} = (\mathcal{Q} \mathcal{R})^T (\mathcal{Q} \mathcal{R}) = \mathcal{R}^T \mathcal{Q}^T \mathcal{Q} \mathcal{R} = \mathcal{R}^T \mathcal{R} \quad (6.48)$$

The pseudoinverse of  $\mathcal{X}$  can be calculated numerically from the singular value decomposition  $\text{svd}(\mathcal{X}_{n \times p}) = \mathcal{U}_{n \times n} \hat{\Sigma} \mathcal{V}_{p \times p}^T$  with  $\hat{\Sigma} = \text{diag}(\sigma_1 \dots \sigma_p)$  and  $\sigma_1 \geq \dots \sigma_r > \sigma_{r+1} = \sigma_{r+2} = \dots = \sigma_p = 0$ . Then  $\mathcal{X}_{p \times n}^+ = \mathcal{V} \hat{\Sigma}^+ \mathcal{U}^T$ .

The pseudoinverse of a rectangular diagonal matrix like  $\Sigma$  is calculated by taking the reciprocal of each non-zero diagonal element. Non-zero means larger than a given value, often  $\epsilon \max(n, p) \max(\Sigma)$ , all values smaller than this are set to zero. Then the matrix is transposed.

In R, the **MASS** package provides the MOORE-PENROSE pseudoinverse through the **ginv** function, calculated via the svd. Alternatively, the **pracma** package provides the **pinv** function.

## References

- [1] W.H. PRESS et al.: *Numerical recipes in Pascal: The art of scientific computing* Cambridge: Cambridge University Press, 1989 ISBN: 9780521375160.
- [2] L.V. ATKINSON, P.J. HARLEY: *An introduction to numerical methods with Pascal* International Computer science series London: Addison-Wesley, 1983 ISBN: 0201137887.

- [3] BORLAND INTERNATIONAL INC.: *Turbo-Pascal Mathe-Toolbox* Computer program 1986 URL: <http://old.macintosh.garden/apps/NMT.zip>.
- [4] G. ENGELN-MÜLLGES, F. REUTER: *Formelsammlung zur Numerischen Mathematik mit Turbo-Pascal Programmen* 2nd ed. Mannheim: Wissenschaftsverlag, 1987 ISBN: 3411031565.
- [5] R. SEDGEWICK: *Algorithms* 1st ed. Reading (Mass.): Addison-Wesley, 1983 ISBN: 9780201066722.
- [6] G.W. STEWART: *Introduction to Matrix Computation* New York: Academic Press, 1973 ISBN: 9780126703504.
- [7] A.C. RENCHER: *Methods of Multivariate Analysis* 2nd ed. New York: John Wiley & Sons, 2002.
- [8] C. MOLER, C. van LOAN: Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later, *SIAM Rev.* **45**:1 (2003), 3–49 DOI: [10.1137/S00361444024180](https://doi.org/10.1137/S00361444024180).
- [9] C. LI, X. ZHU, C. GU: Matrix Padé-Type Method for Computing the Matrix Exponential, *Appl. Math.* **2**:2 (2011), 247–253 DOI: [10.4236/am.2011.22028](https://doi.org/10.4236/am.2011.22028).
- [10] P. BOCHEV, S. MARKOV: A self-validating numerical method for the matrix exponential, *Computing* **43** (1989), 59–72 DOI: [10.1007/BF02243806](https://doi.org/10.1007/BF02243806).
- [11] ROSETTACODE.ORG: *QR decomposition* Web-site, accessed 2018-08-18 2018 URL: [https://rosettacode.org/mw/index.php?title=QR\\_decomposition&oldid=265204](https://rosettacode.org/mw/index.php?title=QR_decomposition&oldid=265204).
- [12] G. LUBE: *QR-Zerlegung für lineare Ausgleichsprobleme* Web-site, accessed 2018-08-16 2004 URL: <http://lp.uni-goettingen.de/get/text/1029>.
- [13] P. G. MARTINSSON: Blocked rank-revealing QR factorizations: How randomized sampling can be used to avoid single-vector pivoting, *ArXiv e-prints* (May 2015) eprint: [1505.08115](https://arxiv.org/abs/1505.08115) URL: <http://adsabs.harvard.edu/abs/2015arXiv150508115M>.
- [14] M. GU, S.C. EISENSTAT: Efficient algorithms for computing a strong rank-revealing QR factorization, *SIAM J. Sci. Comput.* **17**:4 (1996), 848–869 DOI: [10.1137/0917055](https://doi.org/10.1137/0917055).



# 7. Dynamic structures for data of any type

## Abstract

This unit provides abstract LIFOs and FIFOs that can be used with any data type.

This unit provides abstract data structures that can be used to store data of any type (`procedure Store(var I; S : word);`). The variable `I` is untyped (more correctly: can have any type). Untyped parameters of functions and procedures were not part of the original Pascal-definitions, they were introduced in Turbo-Pascal 4. From the point of view of this unit, `I` is simply an array [`S`] of byte, where `S` is the number of bytes required (obtained with `SizeOf()`). This unit was modified from [1]. An implementation of abstract binary trees has been published in [2], but isn't needed for this project.

## 7.1. The interface

Listing 7.1: Interface of unit Dynam

```
1 UNIT Dynam;  
2  
3 INTERFACE
```

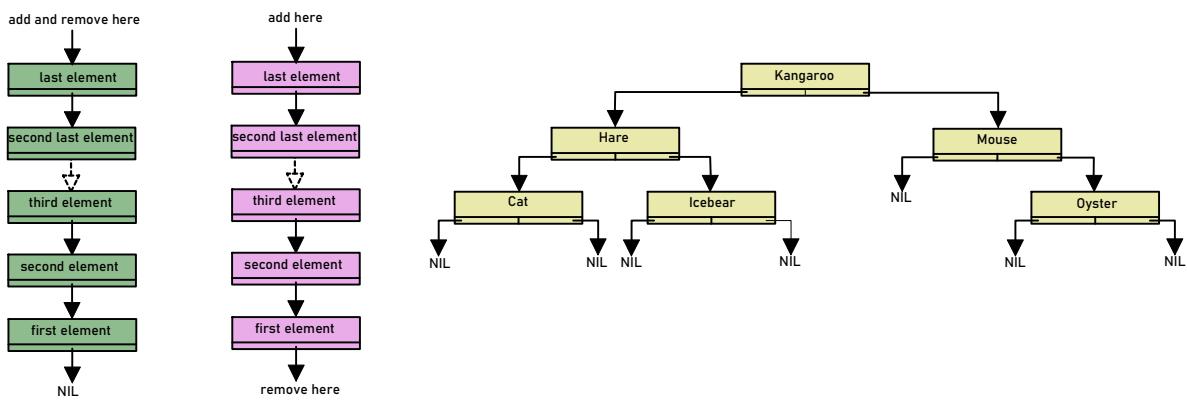


Figure 7.1.: Dynamical data structures: stack (green), queue (pink) and binary tree (yellow). For details see text.

## 7. Dynamic structures for data of any type

```
4
5 USES MathFunc; // here used only for error handling
6
7 CONST DynamError : BOOLEAN = FALSE;
8     MaxElem      = 2000;
9
10 TYPE
11     Space = ARRAY[0..MaxElem] OF BYTE;
12
13 { ***** LiFo ***** }
14
15 LList = ^LListNode;
16
17 LListNode = RECORD
18     Size: WORD;
19     Info: ^Space;
20     Next: LList;
21 END;
22
23 LIFO = LList;
24
25 PROCEDURE InitLIFO(VAR L: Lifo);
26 {einen leeren Stapel initialisieren}
27
28 FUNCTION EmptyLIFO(L: LIFO): BOOLEAN;
29 {Ueprfen, ob der Stapel L leer ist}
30
31 PROCEDURE PUSH(VAR L: LIFO; VAR I; S: WORD);
32 {einen Wert auf den Stapel legen}
33
34 PROCEDURE POP(VAR L: Lifo; VAR I; S: WORD);
35 {einen Wert aus dem Stapel nehmen}
36
37 { ***** FiFo ***** }
38
39 TYPE
40     FList = ^FListNode;
41
42 FListNode = RECORD
43     Size: WORD;
44     Info: ^Space;
45     Next: FList;
46 END;
47
48 FIFO = RECORD
49     First, last: FList;
```

```

50  END;
51
52 PROCEDURE InitFIFO(VAR F: FIFO);
53 {legt eine leere Schlange an}
54
55 FUNCTION EmptyFIFO(F: FIFO): BOOLEAN;
56 {true, wenn die Schlange leer ist}
57
58 PROCEDURE Put(VAR F: FIFO; VAR I; S: WORD);
59 {einen Wert I mit Größe S in die Schlange schieben}
60
61 PROCEDURE Get(VAR F: FIFO; VAR I; S: WORD);
62 {einen Wert I der Größe S aus der Schlange holen}
63
64 {*****}
65
66 IMPLEMENTATION
67
68 VAR ch : char; // for error handling

```

## 7.2. The LIFO (stack)

A LIFO resembles a stack of dinner plates, new plates are added (`Push`) to the top and also required plates are removed (`Pop`) from the top (see fig. 7.1, *left*). The only two other operations allowed are the creation of a new stack (`InitLIFO`), and freeing the space of an empty stack (`EmptyLIFO`).

The items in the stack are linked by pointers. The pointer to the stack points to the top element, which contains a pointer to the next element and so on. When an element is removed from the stack, the pointer to the next element becomes the pointer to the stack; if, on the other hand, an element is added the pointer to the former top becomes its pointer to the next element and the new pointer to the stack points to the new element. At the end of the stack the pointer from the last element (the one added first) is `NIL`. When the last element has been reached, the pointer to the stack therefore becomes `NIL`.

Listing 7.2: LIFO

```

1 PROCEDURE InitLIFO(VAR L: LIFO);
2
3
4 BEGIN
5   L := NIL;
6 END; (* InitLIFO *)
7
8
9 FUNCTION EmptyLIFO(L: LIFO): BOOLEAN;
10

```

## 7. Dynamic structures for data of any type

```
11 BEGIN
12     Result := L = NIL;
13 END; (* EmptyLIFO *)
14
15
16 PROCEDURE PUSH(VAR L: LIFO; VAR I; S: WORD);
17
18 VAR
19     p: LLList;
20
21 BEGIN
22     NEW(p);
23     WITH p^ DO
24         BEGIN
25             Size := S;
26             Next := L;
27             GetMem(Info, Size);
28             Move(I, Info^, Size);
29         END; (* WITH *)
30         L := p;
31     END; (* Push *)
32
33
34 PROCEDURE POP(VAR L: LIFO; VAR I; S: WORD);
35
36 VAR
37     p: LLList;
38
39 BEGIN
40     p := L;
41     WITH p^ DO
42         BEGIN
43             IF Size <> S
44                 THEN
45                     BEGIN
46                         Writeln('TYPE-MISMATCH-ERROR');
47                         HALT;
48                     END;
49                     Move(Info^, I, Size);
50                     FreeMem(Info, Size);
51                     L := Next;
52                 END; (* WITH *)
53                 DISPOSE(p);
54             END; (* Pop *)
```

## 7.3. The FIFO (list, buffer, queue)

If the new mail reaching an office would be added to a LIFO, the mail at the bottom may never be answered. Therefore, in a queue, the elements are added from one end (**Put**) and removed from the other (**Get**). Such a list has two pointers, one for the beginning and the other for the end. The pointer from the element added first is **NIL** (see fig. 7.1, *middle*).

Listing 7.3: FIFO

```

1 PROCEDURE InitFiFo(VAR F: FiFo);
2
3 BEGIN
4   WITH F DO
5     BEGIN
6       First := NIL;
7       last := NIL;
8     END;
9   END;
10
11
12 FUNCTION EmptyFiFo(F: FiFo): BOOLEAN;
13
14 BEGIN
15   Result := F.First = NIL;
16 END;
17
18
19 PROCEDURE Put(VAR F: FiFo; VAR i; s: WORD);
20
21 VAR p: FList;
22
23 BEGIN
24   NEW(p);
25   WITH p^ DO
26     BEGIN
27       Size := s;
28       Next := NIL;
29       GetMem(Info, Size);
30       Move(I, Info^, Size);
31     END;
32   IF EmptyFiFo(F)
33     THEN
34     BEGIN
35       F.First := p;
36       F.last := p;
37     END

```

## 7. Dynamic structures for data of any type

```

38     ELSE
39         BEGIN
40             F.last^.Next := p;
41             F.last := p;
42         END;
43     END;
44
45
46 PROCEDURE Get(VAR F: FiFo; VAR i; s: WORD);
47
48 VAR
49     p: FList;
50
51 BEGIN
52     p := F.First;
53     WITH p^ DO
54         BEGIN
55             IF Size <> s
56             THEN
57                 BEGIN
58                     Writeln('Type-Mismatch-Error');
59                     HALT;
60                 END;
61             Move(Info^, I, Size);
62             FreeMem(Info, Size);
63             F.First := Next;
64         END;
65     DISPOSE(p);
66 END;

```

## 7.4. Test-program

The following program tests the data structures provided by `Dynam` and demonstrates how they can be used.

Listing 7.4: Test program

```

1 PROGRAM TestDynam;
2
3 USES dynam, mathfunc, crt;
4
5
6 PROCEDURE LifoFifoTesten;
7
8 VAR s           : STRING[20];
9     LifoListe   : Lifo;

```

```

10    FifoListe : Fifo;
11    i         : WORD;
12
13BEGIN
14  ClrScr;
15  Writeln('Demoprogramm für die Arbeit mit Fifo und Lifo-Strukturen');
16  Writeln;
17  InitLifo(LifoListe);
18  InitFifo(FifoListe);
19  Writeln('Bitte geben Sie jetzt 10 kurze Strings ein (z.B. Namen)');
20  Writeln;
21  Writeln('EINGABE', 'AUSGABE von LIFO': 30, 'AUSGABE von FIFO': 30);
22  Writeln;
23  FOR i := 1 TO 10 DO
24    BEGIN
25      ReadLn(s);
26      PUSH(LifoListe, s, SizeOf(s));
27      Put(FifoListe, s, SizeOf(s))
28    END;
29  GotoXY(1, 7);
30  WHILE NOT EmptyLifo(LifoListe) DO
31    BEGIN
32      GotoXY(30, WhereY);
33      POP(LifoListe, s, SizeOf(s));
34      Writeln(s)
35    END;
36  GotoXY(1, 7);
37  WHILE NOT EmptyFifo(FifoListe) DO
38    BEGIN
39      GotoXY(60, WhereY);
40      Get(FifoListe, s, SizeOf(s));
41      Writeln(s)
42    END;
43  Writeln;
44  Writeln('weiter mit RETURN:');
45  ReadLn;
46END; {LifoFifoTesten}
47
48TYPE KeyType = WORD;
49
50BEGIN // Test program
51  LifoFifoTesten;
52END.

```

## References

- [1] A. SCHWARZ, U. KERN, I. KURZ, eds.: *Turbo-Pascal, das Allerbeste* Chip Special Turbo Pascal Würzburg: Vogel-Verlag, 1989 ISBN: 3-8023-1008-X.
- [2] ARBEE: *Binary Tree implementation* Computer forum contribution 2012 URL: <https://forum.lazarus.freepascal.org/index.php?action=dlattach;topic=18242.0;attach=4643>.

## **Part II.**

# **Univariate statistics**



# 8. Statistical distributions

## Abstract

This unit calculates the most important distributions and their integrals. The routines are based on [1–5]. All probabilities are given in mathematical form, that is in the range of 0...1.

The unit has the following interface:

Listing 8.1: Interface of `Stat`

```
1 UNIT Stat;  
2  
3 INTERFACE  
4  
5 USES Math, mathfunc;  
6  
7 CONST  
8   StatError: BOOLEAN = FALSE; // toggle FOR error condition  
9  
10 { ***** F-distribution ***** }  
11  
12 FUNCTION Integral_F(F: float; f1, f2: LONGINT): float;  
13  
14 FUNCTION Finv(Alpha, Dfn, Dfe: float) : float;  
15  
16 { ***** t-distribution ***** }  
17  
18 FUNCTION Integral_t(t: float; f: LONGINT): float;  
19  
20 FUNCTION SignificanceLimit_t(p: float; f: LONGINT): float;  
21  
22 PROCEDURE CompareLiterature(VAR t: float; Average, Sta, Literatur: float;  
23           VAR f: LONGINT; n: LONGINT);  
24  
25 PROCEDURE Unequalsigma(VAR t: float; Average1, Average2, sta1, sta2: float;  
26           VAR f: LONGINT; n1, n2: LONGINT);  
27  
28 { ***** X2-distribution ***** }  
29  
30 FUNCTION IntegralChi(ChiSqr: float; f: WORD): float;
```

## 8. Statistical distributions

```
31
32 FUNCTION SignificanceLimit_Chi2 (p : float; dgf : WORD) : float;
33
34 FUNCTION SigChi( Chisq , Df : float ) : float;
35
36 { ***** normal distribution ***** }
37
38 FUNCTION IntegralGauss(z: float): float;
39
40 FUNCTION SignificanceLimitGauss(WantedSecurity: float): float;
41
42 FUNCTION NormalStandardValue (P : float) : float;
43
44 FUNCTION Erf( Z : float ) : float;
45
46 FUNCTION SigNorm (X : float) : float;
47
48 FUNCTION NinvCoarse( P : float ) : float;
49
50 FUNCTION Ninv( P : float ) : float;
51
52 FUNCTION CDNorm (X : float) : float;
53
54 { ***** binomial distribution ***** }
55
56 FUNCTION BinomialFrequency (Trials, Success: LONGINT; P: float): float;
57
58 FUNCTION BinomialIntegral (Trials, Value: LONGINT; P: float): float;
59
60 { ***** Pareto distribution ***** }
61
62 FUNCTION IntegralPareto (x, Minimum, Shape: float): float;
63
64 FUNCTION PDFPareto (x, Minimum, Shape: float): float;
65
66 { ***** beta-distribution ***** }
67
68 FUNCTION CDBeta (X, Alpha, Beta : float; Dprec, MaxIter : INTEGER;
69           VAR Cprec : float; VAR Iter, Ifault : INTEGER ) : float;
70
71 FUNCTION BetaInv (P, Alpha, Beta : float; MaxIter, Dprec: INTEGER;
72           VAR Iter : INTEGER; VAR Cprec : float; VAR Ierr: INTEGER)
73           : float;
74 { ***** gamma-distribution ***** }
```

```

76 FUNCTION ALGama ( Arg : float ) : float;
77
78 FUNCTION GammaInt (Y, P : float; Dprec, MaxIter : INTEGER;
79                      VAR Cprec : float; VAR Iter, Ifault : INTEGER) : float;
80
81 FUNCTION KolmogorovSmirnovIntegral (alam: float): float;
82
83
84 IMPLEMENTATION
85
86 CONST Xln2sp = 0.918938533204673;           // LogE(Sqrt(2 * CONST_PI))
87     MaxPrec = 16;                            // Max. precision
88     LnTenInv = 1/Const_ln10;                 // 1 / LN(10)
89
90 var ch : char; // for error reporting

```

## 8.1. F-distribution

Integration of the F-distribution gives the probability for the 0-hypothesis that two series of measurements have the same standard deviations. Both series must be normally distributed and free of outliers.

Listing 8.2: Routines for the F-Distribution

```

1 FUNCTION Integral_F(F: float; f1, f2: LONGINT): float;
2
3 VAR
4     S, X, A, B, V: float;
5
6
7 FUNCTION R0(M, N: LONGINT; V: float): float;
8
9 VAR
10    G, Sum: float;
11    k, TEST: LONGINT;
12
13 BEGIN
14    k := 1;
15    G := 1;
16    Sum := G;
17    TEST := M DIV 2;
18    WHILE NOT (k = TEST) DO
19        BEGIN
20            k := Succ(k);
21            G := G * V * (N + 2 * k - 4) / (2 * k - 2);
22            Sum := Sum + G;

```

## 8. Statistical distributions

```
23     END;
24     Integral_F := Sum;
25 END;
26
27
28 FUNCTION R1(M: LONGINT; V: float): float;
29
30 VAR
31   G, Sum: float;
32   k, TEST: LONGINT;
33
34 BEGIN
35   k := 1;
36   G := Sqrt(V);
37   Sum := G;
38   TEST := (M - 1) DIV 2;
39   WHILE NOT (k = TEST) DO
40     BEGIN
41       k := Succ(k);
42       G := G * V * (2 * k - 2) / (2 * k - 1);
43       Sum := Sum + G;
44     END;
45   R1 := Sum;
46 END;
47
48
49 FUNCTION R2(f1, f2: LONGINT; V: float): float;
50
51 VAR
52   G, Sum: float;
53   k, TEST: LONGINT;
54
55 BEGIN
56   k := 1;
57   G := 1;
58   Sum := G;
59   TEST := (f1 - 1) DIV 2;
60   WHILE NOT (k = TEST) DO
61     BEGIN
62       k := Succ(k);
63       G := G * V * (f2 + 2 * k - 3) / (2 * k - 1);
64       Sum := Sum + G;
65     END;
66   R2 := Sum;
67 END;
68
```

```

69
70 FUNCTION Q(f2: LONGINT): float;
71
72 VAR
73   k: LONGINT;
74   Product: float;
75
76 BEGIN
77   Product := 1;
78   FOR k := 1 TO ((f2 - 1) DIV 2) DO
79     Product := Product * (k / (k - 0.5));
80   Q := (2 / Pi) * Product;
81 END;
82
83 BEGIN
84   IF (f1 MOD 2) = 0
85     THEN
86       BEGIN
87         X := f2 / (f2 + f1 * F);
88         S := 1 - (R0(f1, f2, 1 - x) * Pot(X, f2 / 2));
89       END
90   ELSE IF (f2 MOD 2) = 0
91     THEN
92       BEGIN
93         X := f2 / (f2 + f1 * F);
94         S := Pot(1 - x, f1 / 2) * R0(f2, f1, X);
95       END
96   ELSE IF (f1 = 1) AND (f2 = 1)
97     THEN
98       S := (2 / Pi) * ArcTan(Sqrt(F))
99   ELSE IF (f1 = 1) AND (f2 > 1)
100    THEN
101      BEGIN
102        X := Sqrt(F / f2);
103        V := 1 / (1 + Sqr(X));
104        S := (2 / Pi) * (ArcTan(X) + X * R1(f2, V) * Sqrt(1
105          / (1 + Sqr(X))));
106      END
107   ELSE IF (f1 > 1) AND (f2 = 1)
108     THEN
109       BEGIN
110         X := Sqrt(1 / (f1 * F));
111         V := 1 / (1 + Sqr(X));
112         S := 1 - (2 / Pi) * (ArcTan(X) + X * R1(f1,
V) *
Sqrt(1 / (1 + Sqr(X))));
```

## 8. Statistical distributions

```

113          END
114      ELSE           {f1 und f2 ungerade und > 1}
115      BEGIN
116          X := Sqrt(F * f1 / f2);
117          V := 1 / (1 + Sqr(X));
118          A := (2 / Pi) * (ArcTan(X) + X * R1(f2, V) *
119                          Sqr(1 / (1 + Sqr(X))));
120          V := Sqr(X) / (1 + Sqr(X));
121          B := Q(f2) * X * Pot(1 / (1 + Sqr(X)), (f2 +
122              1) / 2) *
123              R2(f1, f2, V);
124          S := A - B;
125      END;
126      Result := 1 - S;
127  END;    // Integral_F
128
129 FUNCTION Finv( Alpha, Dfn, Dfe: float ) : float;
130
131 CONST MaxIter = 100;
132     Dprec = 10;
133
134 VAR Fin, Cprec : float;
135     Iter, Ierr : INTEGER;
136
137 BEGIN
138     Fin := -1.0;
139     IF ((Dfn > 0.0) AND (Dfe > 0.0))
140     THEN
141         IF ((Alpha >= 0.0) AND (Alpha <= 1.0))
142         THEN
143             BEGIN
144                 Fin := BetaInv(1.0 - Alpha, Dfn/2.0, Dfe/2.0, MaxIter, Dprec,
145                               Iter, Cprec, Ierr);
146                 IF ((Fin >= 0.0) AND (Fin < 1.0) AND (Ierr = 0))
147                 THEN Fin := Fin * Dfe / (Dfn * (1.0 - Fin));
148             END;
149     Finv := Fin;
150 END (* Finv *);

```

## 8.2. STUDENT's t-distribution

Integration of the  $t$ -distribution returns the error probability for a given  $t$  and degree of freedom  $f$ :

Listing 8.3: Routines for t-distribution

```

1  FUNCTION Integral_t(t: float; f: LONGINT): float;
2
3  VAR
4      x, u, v, w: float;
5
6
7  FUNCTION R(f, v, w: float): float;
8
9  VAR
10     g, Sum: float;
11     i: LONGINT;
12
13 BEGIN
14     i := 2;
15     Sum := w;
16     g := w;
17     REPEAT
18         g := g * v * (i / (i + 1));
19         Sum := Sum + g;
20         i := i + 2;
21     UNTIL i > (f - 3);
22     R := Sum;
23 END;
24
25
26 FUNCTION Q(f, v: float): float;
27
28 VAR
29     g, Sum: float;
30     i: LONGINT;
31
32 BEGIN
33     Sum := 1;
34     i := 1;
35     g := 1;
36     REPEAT
37         g := g * v * (i / (i + 1));
38         Sum := Sum + g;
39         i := i + 2;
40     UNTIL i > (f - 3);
41     Q := Sum;
42 END;
43
44 BEGIN
45     x := t / Sqrt(f);
46     IF f = 1

```

## 8. Statistical distributions

```

47   THEN
48     u := 2 / Pi * ArcTan(x)
49   ELSE
50     BEGIN
51       v := 1 / (1 + Sqr(x));
52       IF Odd(f)
53         THEN
54           BEGIN
55             w := Sqrt(v);
56             u := 2 / Pi * (R(f, v, w) * Sqrt(1 - v) + ArcTan(x));
57           END
58         ELSE
59           u := Q(f, v) * Sqrt(1 - v);
60       END;
61     Result := 1 - u;
62   END; // Integral_t

```

`SignificanceLimit_t` returns that  $t$ -value where the error probability becomes lower than a given value:

```

1  FUNCTION SignificanceLimit_t(p: float; f: LONGINT): float;
2
3 CONST
4   a0 = 2.515517;      a1 = 0.802853;      a2 = 0.010328;
5   b1 = 1.432788;      b2 = 0.189269;      b3 = 0.001308;
6
7 VAR
8   s, q, n, z, d, t1, t2, t4, a, b, c, x: float;
9
10 BEGIN
11   s := 1 - p;
12   q := p / 2;
13   n := Sqrt(Ln(1 / Sqr(q)));
14   z := n - (a0 + n * (a1 + a2 * n)) / (1 + n * (b1 + n * (b2 + b3 * n)));
15   d := Ln(z);
16   t1 := tan(Pi * s / 2);
17   t2 := Sqrt(2 * (1 / (1 - Sqr(s)) - 1));
18   t4 := 2 * Sqrt(1 / (2 * Cos(ArcCos(1 - 2 * Sqr(s)) / 3) - 1) - 1);
19   a := (Ln(t1) - 6 * Ln(t2) + 8 * Ln(t4) - 3 * d) / 0.375;
20   b := 2 * Ln(t1) - 4 * Ln(t2) + 2 * d - 3 * a / 2;
21   c := Ln(t1) - d - a - b;
22   x := 1 / f;
23   Result := Exp(d + x * (c + x * (b + a * x)));
24 END; // SignificanceLimit_t

```

### 8.2.1. Calculating *t*

The *t*-test is used to compare two values. The samples should be either normally distributed or large enough that the central limit theorem applies ( $> 30$ ). Their variance is unknown, otherwise a GAUSS-test would be more appropriate. That is, instead of

$$Z = \sqrt{n} \frac{\bar{x} - \mu_0}{\sigma}$$

, which is normally distributed, we use

$$T = \sqrt{n} \frac{\bar{x} - \mu_0}{S}$$

, which is *t*-distributed, as test-statistics. The distribution was developed by WILLIAM SEALY GOSSET, who published under the pseudonym STUDENT [6].

#### One-sample *t*-test

If there is only one data vector  $\mathbf{x}$ , whose average is to be compared with a population mean, literature value or desired value  $\mu_0$ , then  $H_0 : \bar{x} = \mu_0$  (two-sided test, or larger or smaller for a one-sided test):

$$\begin{aligned} v &= n - 1 \\ \hat{t} &= \sqrt{n} \frac{\bar{x} - \mu_0}{s} \end{aligned} \tag{8.1}$$

The test is

**right-sided**  $H_0 : \bar{x} \leq \mu_0$ , rejected with probability  $\alpha$  if  $\hat{t} > t_{1-\alpha, v}$

**two-sided**  $H_0 : \bar{x} = \mu_0$ , rejected with probability  $\alpha$  if  $|\hat{t}| > t_{1-\frac{\alpha}{2}, v}$

**left-sided**  $H_0 : \bar{x} \geq \mu_0$ , rejected with probability  $\alpha$  if  $\hat{t} < t_{1-\alpha, v}$

Listing 8.4: *t* for comparing sample with population

```

1 PROCEDURE CompareLiterature(VAR t: float; Average, sta, literatur: float;
2   VAR f: LONGINT; n: LONGINT);
3
4 BEGIN
5   t := (Abs(Average - literatur) / sta) * Sqrt(n);
6   f := n - 1;
7 END;
```

#### Paired two samples

If there are two samples, but those are paired (*e.g.*, right and left arm of the same person, or values of the same person obtained before and after treatment), then we can test the average of their differences against zero, this is more powerful than doing a two-sample test.

## 8. Statistical distributions

### The two-sample $t$ -test with equal variances

The two-sample  $t$ -test is used to compare the difference of two averages  $\bar{x}_1, \bar{x}_2$  with the null-hypothesis

**right-sided**  $H_0 : \bar{x}_1 - \bar{x}_2 \leq \omega_0$

**two-sided**  $H_0 : \bar{x}_1 - \bar{x}_2 = \omega_0$

**left-sided**  $H_0 : \bar{x}_1 - \bar{x}_2 \geq \omega_0$

, where  $\omega_0$  is the assumed difference between the averages (usually zero).

In case both samples are independent, but have identical variance,  $t$  and  $v$  are calculated by

$$\begin{aligned} v &= n_1 + n_2 - 2 \\ s^* &= \sqrt{\frac{(n_1 - 1)s_1^2(n_2 - 1)s_2^2}{v}} \\ t &= \frac{\bar{x}_1 - \bar{x}_2 - \omega_0}{s^* \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} = \sqrt{\frac{n_1 n_2}{n_1 + n_2}} \frac{\bar{x}_1 - \bar{x}_2 - \omega_0}{s^*} \end{aligned} \quad (8.2)$$

with  $s^*$  the pooled (weighted) standard deviation of both samples.

Listing 8.5: t for independent samples of equal variance

```

1 PROCEDURE Equalsigma(VAR t: float; Average1, Average2, sta1, sta2: float;
2   VAR f: LONGINT; n1, n2: LONGINT);
3
4 VAR
5   sta: float;
6
7 BEGIN
8   f := n1 + n2 - 2;
9   sta := Sqr((n1 - 1) * Sqr(sta1) + (n2 - 1) * Sqr(sta2)) / f;
10  t := (Abs(Average1 - Average2) / sta) * Sqr(n1 * n2 / (n1 + n2));
11 END;
```

### The two-sample $t$ -test with unequal variances (WELCH-test)

In case the samples are independent, and have unequal variance, then:

$$\begin{aligned} v &= \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1-1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2-1}} \\ s^* &= \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \\ t &= \frac{\bar{x}_1 - \bar{x}_2 - \omega_0}{s^*} \end{aligned} \quad (8.3)$$

Listing 8.6: t for independent samples of unequal variance

```

1 PROCEDURE UnequalSigma(VAR t: float; Average1, Average2, sta1, sta2: float;
2   VAR f: LONGINT; n1, n2: LONGINT);
3
4 BEGIN
5   t := Abs(Average1 - Average2) / (Sqr(Sqr(sta1) / n1) + Sqr(Sqr(sta2) /
6     n2));
7   f := Round((Sqr(Sqr(sta1) / n1) + Sqr(sta2) / n2) /
8     (Sqr(Sqr(sta1) / n1) / (n1 + 1) + Sqr(Sqr(sta2) / n2) / (n2 - 1))) - 2;
9 END;
```

## 8.3. $\chi^2$ Distribution

$\chi^2$  is defined as

$$\chi^2 = \sum_{i=1}^n \frac{(O_i - E_i)^2}{E_i} \quad (8.4)$$

where O and E are the observed and expected values, under the condition that  $E_i > 0 \forall i$ , and that  $\sum_{i=1}^n O_i = \sum_{i=1}^n E_i = n$  [7].

The integral of the  $\chi^2$ -distribution is calculated by [1, Eqn 26.4.6] which, however, causes an overflow for  $\chi^2 > 550$  or so. Therefore, for  $\chi^2 > 500$  the integral is calculated as described in [4, p. 526]. The latter method, however, doesn't converge if  $P_0 > 20\%$  or so.

Listing 8.7:  $\chi^2$  Distribution

```

1 UNCTION IntegralChi(chisqr: float; f: WORD): float;
2
3  VAR
4    s1, s2, s3: extended;
5    c: CHAR;
```

## 8. Statistical distributions

```

6
7     FUNCTION Reihe1(chi: extended; f: WORD): float;
8
9     VAR
10    Fraction, Sum, Summand: extended;
11    k, i: LONGINT;
12
13    BEGIN
14      Sum := 1;
15      k := 1;
16      REPEAT
17        Fraction := 1;
18        FOR i := 1 TO k DO
19          Fraction := Fraction * (f + 2 * i);
20          Summand := Exp(Ln(chi) * 2 * k) / Fraction;
21          Sum := Sum + Summand;
22          INC(k);
23        UNTIL (Summand < MaxError);           {Ü gewnschte Genauigkeit}
24        Result := Sum;
25    END;
26
27    BEGIN
28      IF Abs(ChiSqr) < Zero                // chi^2 ~ 0
29      THEN
30      BEGIN
31        Result := 0.0;
32        EXIT;
33      END;
34      IF ChiSqr < 0                         // should NOT happen
35      THEN
36      BEGIN
37        Result := NaN;
38        c := WriteErrorMessage('Integral of chi^2: chi^2 < 0');
39        StatError := TRUE;
40        EXIT;
41      END;
42      IF ChiSqr < 500
43      THEN
44      BEGIN                                // formula 26.4.6 OF Abramowitz & Stegun
45        s1 := f / 2 * Ln(chisqr / 2);
46        s2 := Ln(Reihe1(Sqrt(chisqr), f));
47        s3 := ((-chisqr / 2) - LnGamma((f + 2) / 2));
48        Result := 1 - Exp(s1 + s2 + s3);
49      END
50      ELSE
51      BEGIN                                // page 526 OF Numerical Methods

```

```

52     Result := IncompleteGamma(0.5 * f, 0.5 * ChiSqr);
53     IF MathError
54         THEN // no convergence
55             BEGIN
56                 Result := NaN;
57                 MathError := FALSE;
58                 StatError := TRUE;
59                 c := WriteErrorMessage('Integral Chi-Sqr: no convergence');
60             END;
61     END;
62 END;

```

The following function calculates the error probability given  $\chi^2$  and  $v$  by transforming the input data to match the requirements of the  $\Gamma$ -distribution. Function `GammaInt` then provides the corresponding cumulative incomplete  $\Gamma$ -probability. An error in the input arguments results in a returned probability of -1.

Listing 8.8: Alternative  $\chi^2$  integral

```

1 FUNCTION SigChi( Chisq , Df : float ) : float;
2
3 CONST MaxIter = 200;
4     Dprec    = 12;
5
6 VAR Iter, Ierr : INTEGER;
7     Cprec      : float;
8
9 BEGIN
10    SigChi := 1.0 - GammaInt(Chisq / 2.0, Df / 2.0, Dprec, MaxIter, Cprec,
11                           Iter, Ierr);
12    IF ( Ierr <> 0 ) THEN SigChi := -1.0;
13 END (* SigChi *);

```

### The inverse $\chi^2$ distribution

Sometimes we have a desired probability for a given degree of freedom, and want to know how high the corresponding  $\chi^2$  needs to be. [8] gives three approximation, quadratic, cubic and improved cubic. Of these, the cubic is the most accurate for  $v \leq 10$ , and the improved cubic for  $v > 10$ :

$$\chi^2(P|v) = \begin{cases} v \left[ 1 - \frac{2}{9v} + x_p \sqrt{\frac{2}{9v}} \right]^3 & \forall v \leq 10 \\ v \left[ 1 - \frac{2}{9v} + \left( x_p - \frac{60h(x_p)}{v} \right) \sqrt{\frac{2}{9v}} \right]^3 & \forall v > 10 \end{cases} \quad (8.5)$$

The paper has a table of  $h(x_p)$ -values, but as can be seen in fig. 8.1, these can be fitted very well with a polynom of 3rd order.  $x_p$  is such that the standardised normal probability function  $P(x_p) = 1 - P$ .

## 8. Statistical distributions

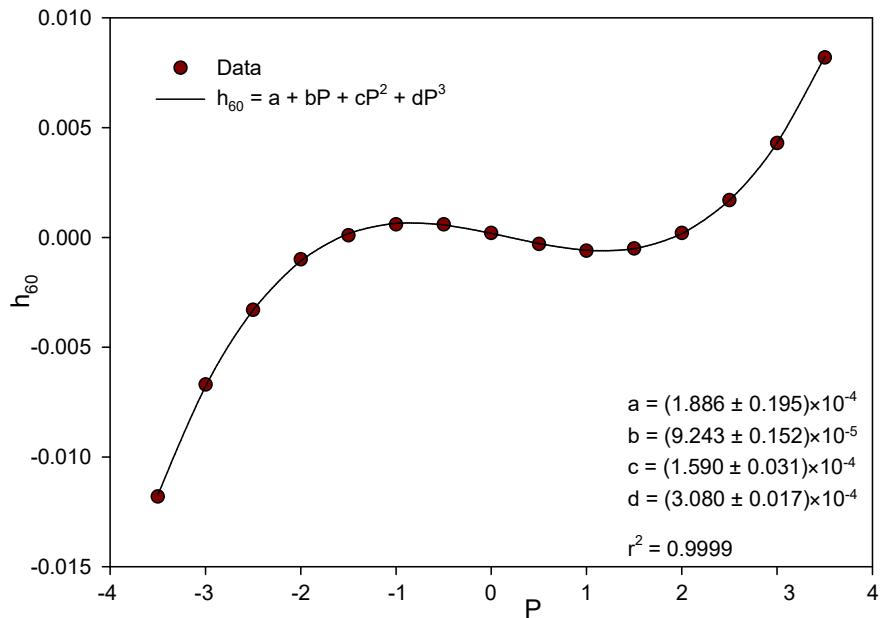


Figure 8.1.:  $h_{60}$ -values as function of  $P$ .

Listing 8.9:  $\chi^2$  required

```

1  FUNCTION SignificanceLimit_Chi2 (p : float; dgf : WORD) : float;
2
3  FUNCTION H (x : float) : float;
4
5  CONST a = 1.886e-4; b = 9.243e-5; c = 1.590e-4; d = 3.080e-4;
6
7  BEGIN
8      Result := a + b*x + c*x*x + d*x*x*x;
9  END;
10
11 VAR x, y : float;
12
13 BEGIN
14     y := -NormalStandardValue(p);
15     IF dgf <= 10
16     THEN x := (1 - 2/(9*dgf) + y
17                 * Sqrt(2/(9*dgf)));
18     ELSE x := (1 - 2/(9*dgf) + (y - (h(y)*60/dgf)) * Sqrt(2/(9*dgf)));
19     Result := dgf * x*x*x;
20 END;
```

### 8.3.1. The G-distribution

The G-distribution [9] is related to the  $\chi^2$ -distribution and defined as

$$G = 2 \sum_{i=1}^n O_i \ln \left( \frac{O_i}{E_i} \right) \quad (8.6)$$

under the same conditions as for  $\chi^2$ . G follows a  $\chi^2$ -distribution with the same degrees of freedom, however, when used for small  $n$ , G approximates better than PEARSON's  $\chi^2$ .

## 8.4. GAUSS' normal distribution

A continuous random variable  $x \in \mathbb{R}$  is GAUSSIAN normal distributed with mean  $\mu$  and variance  $\sigma^2$  ( $x \sim N(\mu, \sigma^2)$ ), if  $x$  has the following probability density:

$$f(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \left( -\frac{(x-\mu)^2}{2\sigma^2} \right) \quad (8.7)$$

This function has a bell-shaped, symmetrical graph where  $\mu$  is the centre of symmetry and also the median and modulus.  $\mu \pm \sigma$  are the inflection points. The **standard normal distribution** has  $\mu = 0, \sigma^2 = 1$ .

The integral of the GAUSS-distribution is:

$$F(x|\mu, \sigma^2) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^x \exp \left[ -\frac{1}{2} \left( \frac{t-\mu}{\sigma} \right)^2 \right] dt \quad (8.8)$$

Then for integration between

$z1 \dots z2$

$$\begin{aligned} z1 < 0 \text{ and } z2 < 0 \quad w &= 0.5 * (\text{GI}(\text{abs}(z1)) - \text{GI}(\text{abs}(z2))) \\ z1 > 0 \text{ and } z2 > 0 \quad w &= 0.5 * (\text{GI}(z1) + \text{GI}(z2)) \\ z1 < 0 \text{ and } z2 > 0 \quad w &= 0.5 * (\text{GI}(\text{abs}(z1))) + \text{GI}(z2) \end{aligned}$$

$-\infty \dots z$

$$\begin{aligned} z > 0 \quad w &= 0.5 + 0.5 * \text{GI}(z) \\ z < 0 \quad w &= 0.5 - 0.5 * \text{GI}(\text{abs}(z)) \end{aligned}$$

$z \dots \infty$

$$\begin{aligned} z > 0 \quad w &= 0.5 - 0.5 * \text{GI}(z) \\ z < 0 \quad w &= 0.5 + 0.5 * \text{GI}(\text{abs}(z)) \end{aligned}$$

The **central limit theorem** of LINDEBERG & LÉVY states, that the sum of (infinitely) many random variables is normally distributed, as long as none of the individual variables exerts a dominating influence on  $\sigma$ . Since variables in nature are often determined by many interacting causes, the normal distribution is very common.

## 8. Statistical distributions

```

1  FUNCTION IntegralGauss(z: float): float;
2
3  VAR
4      f: float;
5
6
7  FUNCTION Reihe(z: float): float;
8
9  VAR
10     Sum, G: float;
11     i: INTEGER;
12
13 BEGIN
14     Sum := z;
15     i := 1;
16     G := z;
17     REPEAT
18         G := G * Sqr(z) / (2 * i + 1);
19         INC(i);
20         Sum := Sum + G;
21     UNTIL G < MaxError;
22     Result := Sum;
23 END;
24
25 BEGIN
26     f := Exp(-Sqr(z) / 2) / Sqrt(2 * Pi);
27     IntegralGauss := 2 * f * Reihe(z);
28 END;

```

The next function calculates the factor with which the standard deviation needs to be multiplied so that any measured value is with a wanted probability in  $-nz \dots +nz$ :

```

1  FUNCTION SignificanceLimitGauss(WantedSecurity: float): float;
2  // -z ... z
3
4  CONST
5      a0 = 2.515517;      a1 = 0.802853;      a2 = 0.010328;
6      b1 = 1.432788;      b2 = 0.189269;      b3 = 0.001308;
7
8  VAR
9      p, q, n, z0, f, p0, t: float;
10
11
12  FUNCTION Reihe(z: float): float;
13
14  VAR
15      Sum, G: float;
16      i: INTEGER;

```

```

17
18 BEGIN
19   Sum := z;
20   i := 1;
21   G := z;
22   REPEAT
23     G := G * Sqr(z) / (2 * i + 1);
24     INC(i);
25     Sum := Sum + G;
26   UNTIL G < MaxError;
27   Result := Sum;
28 END;
29
30 BEGIN
31   q := (1 - WantedSecurity) / 2;
32   p := 1 - q;
33   n := Sqrt(Ln(1 / Sqr(q)));
34   z0 := n - (a0 + n * (a1 + a2 * n)) / (1 + n * (b1 + n * (b2 + b3 * n)));
35   f := Exp(-Sqr(z0) / 2) / Sqrt(2 * Pi);
36   p0 := 0.5 + f * Reihe(z0);
37   t := (p - p0) / f;
38   Result := z0 + t + z0 * Sqr(t) / 2 + (2 * Sqr(z0) + 1) / 6 * pot(t, 3);
39 END;

```

The next function calculates the factor with which the standard deviation needs to be multiplied so that any measured value is with a wanted probability in  $-\infty \dots +nz$ . It is based on [10].

Listing 8.10: Normal standard value

```

1 FUNCTION NormalStandardValue (P : float) : float;
2
3 CONST a0 = 0.389422403767615;      a1 = -1.699385796345221; a2 =
4   1.246899760652504;
5   b0 = 0.155331081623168;      b1 = -0.839293158122257;
6   c0 = 16.896201479841517652; c1 = -2.793522347562718412;
7   c2 = -8.731478129786263127; c3 = -1.000182518730158122;
8   d0 = 7.173787663925508066;  d1 = 8.759693508958633869;
9
10 VAR q, r : float;
11   c : CHAR;
12
13 BEGIN
14   IF (P < 1e-298) OR (P > (1-1e-298))
15   THEN
16     BEGIN
17       c := WriteErrorMessage('Normal standard value: Value outside
18         range');

```

## 8. Statistical distributions

```

17      Result := NaN;
18      StatError := TRUE;
19      EXIT;
20  END;
21  IF P < 0.0465
22  THEN // lower tail
23  BEGIN
24      r := Sqrt(Ln(1/(P*P)));
25      Result := (c0 + c1*r + c2*r*r + c3*r*r*r) / (d0 + d1*r + r*r);
26  END
27 ELSE
28  IF P < 0.9535
29  THEN // central region
30  BEGIN
31      q := p-0.5;
32      r := q*q;
33      Result := q * (a0 + a1*r + a2*r*r) / (b0 + b1*r + r*r);
34  END
35 ELSE // upper tail
36  BEGIN
37      q := 1-p;
38      r := Sqrt(Ln(1/(q*q)));
39      Result := -(c0 + c1*r + c2*r*r + c3*r*r*r) / (d0 + d1*r + r*r);
40  END;
41 END;

```

The GAUSSian error function is

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-r^2} dr \approx \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)n!} \quad (8.9)$$

, which for  $x \in \mathbb{R}$  returns real numbers in  $[-1 \dots +1]$ . The generalisation to complex arguments is of no interest here. The conjugated error function is

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-r^2} dr = 1 - \text{erf}(x) \quad (8.10)$$

Listing 8.11: Error function

```

1 FUNCTION Erf (Z : float) : float;
2
3 CONST A: ARRAY[1..14] OF float =
4     ( 1.1283791670955 ,      0.34197505591854 ,
5       0.86290601455206E-1 ,
6       0.12382023274723E-1 ,    0.11986242418302E-2 ,
7       0.76537302607825E-4 ,
8       0.25365482058342E-5 ,   -0.99999707603738 ,      -1.4731794832805 ,
9
10      );
11
12 VAR Z: float;
13
14 PROCEDURE Calc;
15
16      BEGIN
17          Z := 0;
18          FOR I := 1 TO 14 DO
19              Z := Z + A[I] * Power(-1, I) * Power(Z, 2*I+1) / (2*I+1)/I!;
20      END;
21
22      Erf := 1 - Z;
23  END;
24
25 END;

```

```

7      -1.0573449601594 ,      -0.44078839213875 ,      -0.10684197950781 ,
8      -0.12636031836273E-1 ,      -0.1149393366616E-8 );
9
10     B: ARRAY[1..12] OF REAL =
11      ( -0.36359916427762 ,      0.52205830591727E-1 ,
12      -0.30613035688519E-2 ,
13      -0.46856639020338E-4 ,      0.15601995561434E-4 ,
14      -0.62143556409287E-6 ,
15      2.6015349994799 ,      2.9929556755308 ,      1.9684584582884 ,
16      0.79250795276064 ,      0.18937020051337 ,
17      0.22396882835053E-1 );
18
19 VAR U, X, S: float;
20
21 BEGIN
22   X := ABS(Z);                      (* Get absolute value of argument *)
23   S := Signum(Z);                  (* Remember sign of argument *)
24   IF (S = 0.0)                     (* Check for zero argument *)
25     THEN
26       Result := 0.0
27   ELSE                                (* Check for large argument *)
28     IF (X >= 5.5)
29     THEN
30       Result := S
31     ELSE                                (* Arg in approximation range *)
32     BEGIN
33       U := X * X;
34       IF (X <= 1.5)
35         THEN                                (* Approx. for arg <= 1.5 *)
36         Result := (X * EXP(-U) * (A[1] + U * (A[2] + U *
37                                         (A[3] + U * (A[4] + U * (A[5] + U *
38                                         (A[6] + U * A[7])))))) / (
39                                         1.0 + U * (B[1] + U * (B[2] + U *
40                                         (B[3] + U * (B[4] + U * (B[5] + U *
41                                         B[6])))))) * S
42     ELSE                                (* Approx. for arg > 1.5 *)
43       Result := (EXP(-U) * (A[8] + X * (A[9] + X * (A[10] + X *
44                                         (A[11] + X * (A[12] + X * (A[13] + X * A[14]
45                                         )))))) / (
46                                         1.0 + X * (B[7] + X * (B[8] + X * (B[9] + X *
47                                         (B[10] + X * (B[11] + X * B[12])))))) + 1.0) *
48       S;
49
50   END;
51 END;      (* Erf *)

```

The error function is related to the normal distribution with standard deviation  $\sigma$  and

## 8. Statistical distributions

expected value  $\mu$

$$F(x) = \frac{1}{2} \left[ 1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \right] \quad (8.11)$$

Listing 8.12: Tail probability of normal distribution

```

1  FUNCTION SigNorm (X : float) : float;
2
3  BEGIN
4      IF X >= 0.0
5          THEN Result := 1.0 - (1.0 + Erf(X / Const_Sqrt2)) / 2.0
6      ELSE Result := 1.0 - (1.0 - Erf(-X / Const_Sqrt2)) / 2.0;
7  END (* SigNorm *);
```

Listing 8.13: Cumulative normal distribution probability

```

1  FUNCTION CDNORM (X : float) : float;
2
3  BEGIN
4      IF X >= 0.0
5          THEN Result := ( 1.0 + Erf( X / Const_Sqrt2 ) ) / 2.0
6      ELSE Result := ( 1.0 - Erf( -X / Const_Sqrt2 ) ) / 2.0;
7  END;
```

An alternative way to calculate the percentage point of the normal distribution with about 6 valid figures is

Listing 8.14: percentage point of normal distribution

```

1  FUNCTION NinvCoarse (P : float) : float;
2
3  CONST Lim = 1.0E-20;
4
5  VAR Y : float;
6      Pr: float;
7      Nv: float;
8
9  CONST PN : ARRAY [1..5] OF float =
10     ( -0.322232431088 , -1.0 , -0.342242088547 ,
11       -0.0204231210245 , -0.453642210148E-4 );
12
13     QN : ARRAY [1..5] OF float =
14     ( 0.0993484626060 , 0.588581570495 , 0.531103462366 ,
15       0.103537752850 , 0.38560700634E-2 );
16
17  BEGIN (* NinvCoarse *)
18      Result := 0.0;
19      IF (P > 0.5)
20          THEN Pr := 1.0 - P
```

```

21     ELSE Pr := P;
22 IF (( Pr >= Lim ) AND ( Pr <> 0.5 ))
23 THEN
24 BEGIN
25     Y := SQRT ( LN( 1.0 / Pr / Pr ) );
26     Nv := Y + (((Y * PN[ 5 ] + PN[ 4 ]) * Y + PN[ 3 ]) * Y
27                 + PN[ 2 ]) * Y + PN[ 1 ] ) /
28             (((Y * QN[ 5 ] + QN[ 4 ]) * Y + QN[ 3 ]) * Y
29                 + QN[ 2 ]) * Y + QN[ 1 ] );
30     IF (P < 0.5)
31     THEN Result := -Nv
32     ELSE Result := Nv;
33 END;
34 END (* NinvCoarse *);

```

This result can be made precise to 12 valid figures by using a TAYLOR series on the approximation error:

Listing 8.15: percentage point of normal distribution

```

1 FUNCTION Ninv( P : float ) : float;
2
3 VAR Xp, P1, Z, X3, X2, X1, Phi: float;
4
5 BEGIN
6     Xp := Ninv(P);
7     P1 := SigNorm(Xp);
8     Phi := SQRT(1.0 / (2.0 * PI)) * EXP(-(Xp * Xp) / 2.0);
9     Z := (P - P1) / Phi;
10    X3 := (2.0 * (Xp * Xp) + 1.0) * Z / 3.0;
11    X2 := (X3 + Xp) * Z / 2.0;
12    X1 := ((X2 + 1.0) * Z);
13    Result := Xp + X1;
14 END (* Ninv *);

```

## 8.5. Binomial distribution

The binomial frequency is the probability to have `Success` successes out of `Trials` attempts, when the success probability per attempt is `P`

Listing 8.16: Routines for the binomial distribution

```

1 FUNCTION BinomialFrequency(Trials, Success: LONGINT; P: float): float;
2
3 VAR
4     TrialsOverSuccess: float;
5
6 BEGIN

```

## 8. Statistical distributions

```

7   TrialsOverSuccess := BinomialCoef(Trials, Success);
8   Result := TrialsOverSuccess * pot(P, Success) *
9     pot(1 - P, Trials - Success);
10  END;

```

The integral is the probability that the number of successes is less or equal than `Value`, when the number of trials is `Trials` and the probability of success in each trial is `P`.

```

1  FUNCTION BinomialIntegral(Trials, Value: LONGINT; P: float): float;
2
3  VAR
4    Sum: float;
5    i: WORD;
6
7  BEGIN
8    Sum := 0.0;
9    FOR i := 0 TO Value DO
10      Sum := Sum + BinomialFrequency(Trials, i, P);
11    Result := Sum;
12  END;

```

## 8.6. The PARETO (80/20) distribution

The PARETO distribution [11] describes a continuous function on the interval  $[x_{\min}, \infty]$ , the cumulative distribution function of the PARETO-distribution with Minimum  $x_{\min} \geq 0$ , Shape  $k > 1$  is

$$f(x) = \begin{cases} \frac{kx_{\min}^k}{x^{k+1}} & x \geq x_{\min} \\ 0 & x < x_{\min} \end{cases} \quad (8.12)$$

This distribution can describe parameters that cover several orders of magnitude and depend on many random factors. The PARETO-principle says that the lowest 20 % of the independent variable account for 80 % of the dependent variable (law of diminishing return), then  $k \approx 1.16$ .

Listing 8.17: The PARETO distribution

```

1  FUNCTION IntegralPareto(x, Minimum, Shape: float): float;
2
3  BEGIN
4    Result := 1 - Pot((Minimum / x), Shape);
5  END;
6
7
8  FUNCTION ProbabilityDistributionFunctionPareto(x, Minimum, Shape: float):
9    float;
10 BEGIN

```

```

11  Result := Shape * pot(Minimum, Shape) / pot(x, Shape + 1);
12  END;
13
14
15  END. // Stat

```

## 8.7. The $\beta$ -distribution

Listing 8.18:  $\beta$ distribution

```

1  FUNCTION CDBeta (X, Alpha, Beta : float; Dprec, MaxIter : INTEGER;
2                  VAR Cprec      : float; VAR Iter       : INTEGER;
3                  VAR Ifault     : INTEGER ) : float;
4
5  VAR Epsz, A, B, C,
6      F, Fx, Apb, Zm,
7      Alo, Ahi, Aev, Aod,
8      Blo, Bhi, Bod, Bev,
9      Zm1, D1           : float;
10     Ntries          : INTEGER;
11     Qswap, Qdoit, Qconv : BOOLEAN;
12
13  LABEL 20;
14
15  BEGIN
16    (* Initialize *)
17    IF Dprec > MaxPrec
18      THEN Dprec := MaxPrec
19    ELSE IF Dprec <= 0
20      THEN Dprec := 1;
21    Cprec := Dprec;
22    Epsz := pot(10, -Dprec);
23    X := X;
24    A := Alpha;
25    B := Beta;
26    QSwap := FALSE;
27    CDBeta := -1.0;
28    Qdoit := TRUE;
29    (* Check arguments. Error if: X <= 0 or A <= 0 or B <= 0      *)
30    Ifault := 1;
31    IF (X <= 0.0) OR ((A <= 0.0) OR (B <= 0.0)) OR ( X >= 1.0 )
32      THEN
33        BEGIN
34          ch := WriteErrorMessage('CDBeta: illegal arguments');
35          StatError := TRUE;

```

## 8. Statistical distributions

```

36         EXIT;
37     END;
38     CDBeta := 1.0;
39     Ifault := 0;
40     (* IF X > A / (A + B) then swap A, B for more efficient eval. *)
41     IF (X > (A / (A + B)))
42     THEN
43     BEGIN
44         X      := 1.0 - X;
45         A      := Beta;
46         B      := Alpha;
47         QSwap := TRUE;
48     END;
49     (* Check for extreme values *)
50     IF ((X = A) OR (X = B)) THEN GOTO 20;
51     IF (A = ((B * X) / (1.0 - X))) THEN GOTO 20;
52     IF (ABS(A - (X * (A + B))) <= Epsz) THEN GOTO 20;
53     C := ALGama(A + B) + A * LN(X) + B * LN(1.0 - X) - ALGama(A) -
54         ALGama(B) - LN(A - X * (A + B));
55     IF ((C < -36.0) AND QSwap) OR (C < -180.0)
56     THEN
57     BEGIN
58         ch := WriteErrorMessage('CDBeta: extreme values outside computable
59                               range');
60         StatError := TRUE;
61         EXIT;
62     END;
63     CDBeta := 0.0;
64     (* Set up continued fraction expansion evaluation. *)
65     20:
66     Apb := A + B;
67     Zm := 0.0;
68     Alo := 0.0;
69     Bod := 1.0;
70     Bev := 1.0;
71     Bhi := 1.0;
72     Blo := 1.0;
73     Ahi := EXP(ALGama(Apb) + A * LN(X) + B * LN(1.0 - X) -
74                 ALGama(A + 1.0) - ALGama(B));
75     F := Ahi;
76     Iter := 0;
77     (* Continued fraction loop begins here. Evaluation continues until
78     maximum iterations are exceeded, or convergence achieved.      *)
79     Qconv := FALSE;
80     REPEAT
81         Fx := F;

```

```

81      Zm1 := Zm;
82      Zm := Zm + 1.0;
83      D1 := A + Zm + Zm1;
84      Aev := -(A + Zm1) * (Apb + Zm1) * X / D1 / (D1 - 1.0);
85      Aod := Zm * (B - Zm) * X / D1 / (D1 + 1.0);
86      Alo := Bev * Ahi + Aev * Alo;
87      Blo := Bev * Bhi + Aev * Blo;
88      Ahi := Bod * Alo + Aod * Ahi;
89      Bhi := Bod * Blo + Aod * Bhi;
90      IF ABS(Bhi) < MinRealNumber THEN Bhi := 0.0;
91      IF (Bhi <> 0.0)
92          THEN
93              BEGIN
94                  F := Ahi / Bhi;
95                  Qconv := (ABS((F - Fx) / F) < Epsz);
96              END;
97              Iter := Iter + 1;
98      UNTIL ( (Iter > MaxIter) OR Qconv );
(* Arrive here when convergence achieved, or maximum iterations exceeded.
*)
100     IF (Qswap)
101         THEN CDBeta := 1.0 - F
102         ELSE CDBeta := F;
(* Calculate precision of result *)
103     IF ABS(F - Fx) <> 0.0
104         THEN Cprec := -Log(ABS(F - Fx), 10)
105         ELSE Cprec := MaxPrec;
106     END;    (* CDBeta *)
107
108
109
110    FUNCTION BetaInv (P, Alpha, Beta : float; MaxIter, Dprec: INTEGER;
111                      VAR Iter : INTEGER; VAR Cprec : float; VAR Ierr: INTEGER)
112                      : float;
113
113    VAR Eps, Xim1, Xi, Xip1, Fim1, Fi,
114        W, Cmplbt, Adj, Sq, R, S, T,
115        G, A, B, PP, H, A1, B1, Eprec : float;
116        Done : BOOLEAN;
117        Jter : INTEGER;
118
119    LABEL 10, 30;
120
121    BEGIN
122        Ierr := 1;
123        Result := P;
(* Check validity of arguments *)

```

## 8. Statistical distributions

```

125   IF ((Alpha <= 0.0) OR (Beta <= 0.0)) OR ((P > 1.0) OR (P < 0.0))
126     THEN
127       BEGIN
128         ch := WriteErrorMessage('BetaInv: illegal parameters');
129         StatError := TRUE;
130         Result := -1.0;
131         EXIT;
132       END;
133 (* Check for P = 0 or 1 *)
134   IF ((P = 0.0) OR (P = 1.0))
135     THEN
136       BEGIN
137         Iter := 0;
138         Cprec := MaxPrec;
139         EXIT;
140       END;
141 (* Set precision *)
142   IF Dprec > MaxPrec
143     THEN Dprec := MaxPrec
144   ELSE IF Dprec <= 0
145     THEN Dprec := 1;
146   Cprec := Dprec;
147   Eps := pot(10, -2 * Dprec);
148 (* Flip params if needed so that P for evaluation is <= .5 *)
149   IF (P > 0.5)
150     THEN
151       BEGIN
152         A := Beta;
153         B := Alpha;
154         PP := 1.0 - P;
155       END
156     ELSE
157       BEGIN
158         A := Alpha;
159         B := Beta;
160         PP := P;
161       END;
162 (* Generate initial approximation. Several different ones used, depending
upon parameter values. *)
163   Ierr := 0;
164   Cmplbt := ALGama(A) + ALGama(B) - ALGama(A + B);
165   Fi := Ninv(1.0 - PP);
166   IF ((A > 1.0) AND (B > 1.0))
167     THEN
168       BEGIN
169         R := (Fi * Fi - 3.0) / 6.0;

```

```

170      S := 1.0 / (A + A - 1.0);
171      T := 1.0 / (B + B - 1.0);
172      H := 2.0 / (S + T);
173      W := Fi * SQRT(H + R) / H - (T - S) * (R + 5.0 / 6.0 - 2.0 / (3.0 *
174          H));
175      Xi := A / (A + B * EXP(W + W));
176  END
177 ELSE
178 BEGIN
179     R := B + B;
180     T := 1.0 / (9.0 * B);
181     T := R * Pow((1.0 - T + Fi * SQRT(T)), 3);
182     IF (T <= 0.0)
183         THEN
184             Xi := 1.0 - EXP((LN((1.0 - PP) * B) + Cmplbt) / B)
185     ELSE
186         BEGIN
187             T := (4.0 * A + R - 2.0) / T;
188             IF (T <= 1.0)
189                 THEN Xi := EXP((LN(PP * A) + Cmplbt) / PP)
190                 ELSE Xi := 1.0 - 2.0 / (T + 1.0);
191         END;
192     END;
193 (* Force initial estimate to reasonable range. *)
194 IF (Xi < 0.0001) THEN Xi := 0.0001;
195 IF (Xi > 0.9999) THEN Xi := 0.9999;
196 (* Set up Newton-Raphson loop *)
197 A1 := 1.0 - A;
198 B1 := 1.0 - B;
199 Fim1 := 0.0;
200 Sq := 1.0;
201 Xim1 := 1.0;
202 Iter := 0;
203 Done := FALSE;
204 (* Begin Newton-Raphson loop *)
205 REPEAT
206     Iter := Iter + 1;
207     Done := Done OR (Iter > MaxIter);
208     Fi := CDBeta(Xi, A, B, Dprec+1, MaxIter, Eprec, Jter, Ierr);
209     IF (Ierr <> 0)
210         THEN
211             BEGIN
212                 Ierr := 2;
213                 Done := TRUE;
214             END
215     ELSE
216 
```

## 8. Statistical distributions

```

215      BEGIN
216          Fi := (Fi - PP) * EXP(Cmplbt + A1 * LN(Xi) + B1 * LN(1.0 - Xi));
217          IF ((Fi * Fim1) <= 0.0) THEN Xim1 := Sq;
218          G := 1.0;
219 10:    REPEAT
220          Adj := G * Fi;
221          Sq := Adj * Adj;
222          IF (Sq >= Xim1) THEN G := G / 3.0;
223          UNTIL (Sq < Xim1);
224          Xip1 := Xi - Adj;
225          IF ((Xip1 < 0.0) OR (Xip1 > 1.0))
226          THEN
227              BEGIN
228                  G := G / 3.0;
229                  GOTO 10;
230              END;
231          IF (Xim1 <= Eps) THEN GOTO 30;
232          IF (Fi * Fi <= Eps) THEN GOTO 30;
233          IF ((Xip1 = 0.0) OR (Xip1 = 1.0))
234          THEN
235              BEGIN
236                  G := G / 3.0;
237                  GOTO 10;
238              END;
239          IF (Xip1 <> Xi)
240          THEN
241              BEGIN
242                  Xi := Xip1;
243                  Fim1 := Fi;
244              END
245          ELSE
246              Done := TRUE;
247          End;
248          UNTIL( Done );
249 30:
250          Result := Xi;
251          IF (P > 0.5) THEN Result := 1.0 - Xi;
252          IF ABS(Xi - Xim1) <> 0.0
253          THEN
254              Cprec := -log(ABS(Xi - Xim1), 10)
255          ELSE
256              Cprec := MaxPrec;
257      END (* BetaInv *);

```

## 8.8. The $\Gamma$ -distribution

EULER's  $\Gamma$ -function for  $n \in \mathbb{N}$  is

$$\Gamma(n) = (n-1)! = \int_0^\infty t^{n-1} e^{-t} dt \quad (8.13)$$

with  $t = -\log(n)$ . This function can be expanded to  $\mathbb{R}$  and  $\mathbb{C}$ :

$$\Gamma(x) = \lim_{n \rightarrow \infty} \frac{n! n^x}{x(x+1)(x+2)\dots(x+n)} = \int_0^\infty t^{x-1} e^{-t} dt \quad \forall \quad \operatorname{Re}(x) > 0 \quad (8.14)$$

The upper incomplete  $\gamma$ -function is

$$\gamma(a, x) = \int_0^x t^{a-1} e^{-t} dt \quad (8.15)$$

, for the lower incomplete  $\gamma$ -function the integration limits are  $x$  and  $\infty$ .

The  $\Gamma$ -distribution  $G(x|\beta, \alpha)$  has the density and distribution function

$$f(x) = \begin{cases} \frac{b^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x} & x > 0 \\ 0 & x \leq 0 \end{cases} \quad (8.16)$$

$$F(x) = \begin{cases} \gamma(\alpha, \beta x) & x \geq 0 \\ 0 & x < 0 \end{cases} \quad (8.17)$$

with the inverse scale parameter  $\beta > 0$  and the shape parameter  $\alpha > 0$ .

Listing 8.19:  $\gamma$ distribution

```

1 FUNCTION ALGama( Arg : float ) : float;
2
3 VAR
4   Rarg, Alinc, Scale, Top, Bot, Frac, Algval : float;
5   I, Iapprox, Iof, Ilo, Ihi : INTEGER;
6   Qminus, Qdoit : BOOLEAN;
7
8
9 CONST P : ARRAY [1..29] OF float =
10    ( 4.12084318584770E+00 , 8.56898206283132E+01 ,
11      2.43175243524421E+02 , -2.61721858385614E+02 ,
12      -9.22261372880152E+02 , -5.17638349802321E+02 ,
13      -7.74106407133295E+01 , -2.20884399721618E+00 ,
14      5.15505761764082E+00 , 3.77510679797217E+02 ,
15      5.26898325591498E+03 , 1.95536055406304E+04 ,
16      1.20431738098716E+04 , -2.06482942053253E+04 ,
17      -1.50863022876672E+04 , -1.51383183411507E+03 ,
18      -1.03770165173298E+04 , -9.82710228142049E+05 ,
19      -1.97183011586092E+07 , -8.73167543823839E+07 ,

```

## 8. Statistical distributions

```

20          1.11938535429986E+08 , 4.81807710277363E+08 ,
21          -2.44832176903288E+08 , -2.40798698017337E+08 ,
22          8.06588089900001E-04 , -5.94997310888900E-04 ,
23          7.93650067542790E-04 , -2.7777777688189E-03 ,
24          8.3333333333330E-02 );
25
26  Q : ARRAY [1..24] OF float =
27  (
28    1.00000000000000E+00 , 4.56467718758591E+01 ,
29    3.77837248482394E+02 , 9.51323597679706E+02 ,
30    8.46075536202078E+02 , 2.62308347026946E+02 ,
31    2.44351966250631E+01 , 4.09779292109262E-01 ,
32    1.00000000000000E+00 , 1.28909318901296E+02 ,
33    3.03990304143943E+03 , 2.20295621441566E+04 ,
34    5.71202553960250E+04 , 5.26228638384119E+04 ,
35    1.44020903717009E+04 , 6.98327414057351E+02 ,
36    1.00000000000000E+00 , -2.01527519550048E+03 ,
37    -3.11406284734067E+05 , -1.04857758304994E+07 ,
38    -1.11925411626332E+08 , -4.04435928291436E+08 ,
39    -4.35370714804374E+08 , -7.90261111418763E+07 );
40
41 BEGIN
42   Algval := MaxRealNumber; // Initialize
43   Scale := 1.0;
44   Alinc := 0.0;
45   Frac := 0.0;
46   Rarg := Arg;
47   Iof := 1;
48   Qminus := FALSE;
49   Qdoit := TRUE;
50   IF (Rarg < 0.0)           // Adjust for negative argument
51   THEN
52     BEGIN
53       Qminus := TRUE;
54       Rarg := -Rarg;
55       Top := Int( Rarg );
56       Bot := 1.0;
57       IF((INT(Top / 2.0) * 2.0) = 0.0) THEN Bot := -1.0;
58       Top := Rarg - Top;
59       IF (Top = 0.0)
60       THEN
61         Qdoit := FALSE
62       ELSE
63         BEGIN
64           Frac := Bot * CONST_PI / SIN( Top * CONST_PI );
65           Rarg := Rarg + 1.0;
66           Frac := LN( ABS( Frac ) );

```

```

66      END;
67  END;
68 { Choose approximation interval based upon argument range }
69 IF (Rarg = 0.0)
70 THEN
71   Qdoit := FALSE
72 ELSE
73   IF (Rarg <= 0.5)
74     THEN
75       BEGIN
76         Alinc := -LN( Rarg );
77         Scale := Rarg;
78         Rarg := Rarg + 1.0;
79         IF (Scale < MachineEpsilon)
80           THEN
81             BEGIN
82               Algval := Alinc;
83               Qdoit := FALSE;
84             END;
85       END;
86     ELSE
87       IF (Rarg <= 1.5)
88         THEN
89           Scale := Rarg - 1.0
90         ELSE
91           IF (Rarg <= 4.0)
92             THEN
93               BEGIN
94                 Scale := Rarg - 2.0;
95                 Iof := 9;
96               END;
97             ELSE
98               IF (Rarg <= 12.0)
99                 THEN
100                  Iof := 17
101                ELSE
102                  IF (Rarg <= MaxRealNumber)
103                    THEN
104                      BEGIN
105                        Alinc := ( Rarg - 0.5 ) * LN( Rarg ) - Rarg +
106                           Xln2sp;
107                        Scale := 1.0 / Rarg;
108                        Rarg := Scale * Scale;
109                        Top := P[ 25 ];
110                        for i := 26 to 29 do Top := Top * Rarg + P[ I ];
111                        Algval := Scale * Top + Alinc;

```

## 8. Statistical distributions

```

111                      Qdoit := FALSE;
112                      END;
113 { Common evaluation code for Arg <= 12. Horner's method is used, which
114   seems to
115   give better accuracy than continued fractions. }
116 IF Qdoit
117 THEN
118 BEGIN
119   Ilo    := Iof + 1;
120   Ihi    := Iof + 7;
121   Top    := P[ Iof ];
122   Bot    := Q[ Iof ];
123   FOR I := Ilo TO Ihi DO
124     BEGIN
125       Top    := Top * Rarg + P[ I ];
126       Bot    := Bot * Rarg + Q[ I ];
127     END;
128   Algval := Scale * ( Top / Bot ) + Alinc;
129 END;
130 IF (Qminus) THEN Algval := Frac - Algval;
131 Result := Algval;
132 END; (* ALGama *)
133
134 FUNCTION GammaInt (Y, P : float; Dprec, MaxIter : INTEGER;
135                      VAR Cprec : float; VAR Iter, Ifault : INTEGER) : float;
136
137 CONST Oflo      = 1.0E+37;
138 MinExp     = -87.0;
139
140 VAR F, C, A, B, Term, An, Gin, Rn, Dif, Eps : float;
141   Pn          : ARRAY[1..6] OF float;
142   Done        : BOOLEAN;
143
144 BEGIN
145 (* Check arguments *)
146   Ifault := 1;
147   Result := 1.0;
148   IF ((Y <= 0.0) OR (P <= 0.0))
149   THEN
150     BEGIN
151       ch := WriteErrorMessage('Gamma integralv: illegal parameters');
152       StatError := TRUE;
153       EXIT;
154     END;
155 (* Check value of F *)

```

```

156  Ifault := 0;
157  F := P * LN(Y) - ALGama(P + 1.0) - Y;
158  IF (F < MinExp)
159    THEN
160      BEGIN
161        ch := WriteErrorMessage('Gamma integralv: extreme value not
162                               computable');
163        StatError := TRUE;
164        EXIT;
165      END;
166  F := EXP(F);
167  IF (F = 0.0)
168    THEN
169      BEGIN
170        ch := WriteErrorMessage('Gamma integralv: extreme value not
171                               computable');
172        StatError := TRUE;
173        EXIT;
174      END;
175  (* Set precision *)
176  IF Dprec > MaxPrec
177    THEN Dprec := MaxPrec
178  ELSE IF Dprec <= 0 THEN Dprec := 1;
179  Cprec := Dprec;
180  Eps := Pot(10, -Dprec);
181  (* Choose infinite series or continued fraction.      *)
182  IF ((Y > 1.0) AND (Y >= P))
183    THEN
184      BEGIN (* Continued Fraction *)
185        A := 1.0 - P;
186        B := A + Y + 1.0;
187        Term := 0.0;
188        Pn[1] := 1.0;
189        Pn[2] := Y;
190        Pn[3] := Y + 1.0;
191        Pn[4] := Y * B;
192        Gin := Pn[3] / Pn[4];
193        Done := FALSE;
194        Iter := 0;
195        REPEAT
196          Iter := Iter + 1;
197          A := A + 1.0;
198          B := B + 2.0;
199          Term := Term + 1.0;
          An := A * Term;
          Pn[5] := B * Pn[3] - An * Pn[1];

```

## 8. Statistical distributions

```

200      Pn[6] := B * Pn[4] - An * Pn[2];
201      IF (Pn[6] <> 0.0)
202          THEN
203              BEGIN
204                  Rn := Pn[ 5 ] / Pn[ 6 ];
205                  Dif := ABS(Gin - Rn);
206                  IF (Dif <= Eps)
207                      THEN
208                          IF (Dif <= (Eps * Rn)) THEN Done := TRUE;
209                          Gin := Rn;
210                  END;
211                  Pn[1] := Pn[3];
212                  Pn[2] := Pn[4];
213                  Pn[3] := Pn[5];
214                  Pn[4] := Pn[6];
215                  IF (ABS(Pn[5]) >= Oflo)
216                      THEN
217                          BEGIN
218                              Pn[1] := Pn[1] / Oflo;
219                              Pn[2] := Pn[2] / Oflo;
220                              Pn[3] := Pn[3] / Oflo;
221                              Pn[4] := Pn[4] / Oflo;
222                          END;
223                      UNTIL (Iter > MaxIter) OR Done;
224                      Gin := 1.0 - (F * Gin * P);
225                      Result := Gin;
226                      (* Calculate precision of result *)
227                      IF Dif <> 0.0
228                          THEN Cprec := -Log(Dif, 10)
229                          ELSE Cprec := MaxPrec;
230                      END      (* Continued Fraction *)
231      ELSE
232          BEGIN (* Infinite series *)
233              Iter := 0;
234              Term := 1.0;
235              C := 1.0;
236              A := P;
237              Done := FALSE;
238              REPEAT
239                  A := A + 1.0;
240                  Term := Term * Y / A;
241                  C := C + Term;
242                  Iter := Iter + 1;
243              UNTIL (Iter > MaxIter) OR ((Term / C) <= Eps);
244              Result := C * F;
245              (* Calculate precision of result *)

```

```

246      Cprec := -Log(Term / C, 10);
247      END   (* Infinite series *);
248  END;    (* GammaIn *)

```

## 8.9. KOLMOGOROV-SMIRNOV-test

The KOLMOGOROV-SMIRNOV-test is used to test **H<sub>0</sub>**: **Two random variables have the same distribution** ( $F_1(x) = F_2(x)$ ) against **H<sub>1</sub>**: **The two random variables have different distributions** ( $F_1(x) \neq F_2(x)$ ). The second variable can be either experimental (two sample test), or can come from a theoretical distribution (one sample test). This can be used, for example, to test whether a sample is normally distributed; and therefore, whether tests relying on this distribution are applicable.

For the test, both samples are sorted in ascending order. Then  $d_n = \|F_1 - F_2\| = \sup|F_1(x) - F_2(x)|$ , where the supremum of a set is defined as the smallest number that is larger than all members of the set. For example, for the set of real numbers  $< 2$ , two is the supremum: the set contains no number larger than 2, and no number smaller than 2 is larger than all members of the set.

In other words, the cumulative distribution of both functions are plotted, and  $d_n$  is the largest distance between both curves. Since the test is non-parametric, it works better for small samples than the  $\chi^2$ -test, and it works independent of the actual distribution of the variables. It was developed for continuous variables, but also works for discrete or rank-scaled variables. It is, however, less sensitive than parametric tests would be.

The following routine calculates the probability for the 0-hypothesis in the KOLMOGOROV-SMIRNOV-test. `alam` is

- $d\sqrt{n}$  for the 1-sample test
- $d\sqrt{\frac{n_1*n_2}{(n_1+n_2)}}$  for the 2-sample test

Listing 8.20: KOLMOGOROV-SMIRNOV-test

```

1  FUNCTION KolmogorovSmirnovIntegral(alam: float): float;
2
3  CONST
4      eps1 = 0.001;
5      eps2 = 1.0e-8;
6
7  VAR
8      a2, fac, sum, term, termbf: float;
9      j: INTEGER;
10
11 BEGIN
12     a2 := -2.0 * alam * alam;
13     fac := 2.0;

```

```

14   sum := 0.0;
15   termbf := 0.0;
16   FOR j := 1 TO 100 DO
17     BEGIN
18       term := fac * Exp(a2 * Sqr(j));
19       sum := sum + term;
20       IF ((Abs(term) < (eps1 * termbf)) OR (Abs(term) < (eps2 * sum)))
21         THEN
22           BEGIN // convergence reached
23             Result := sum;
24             EXIT;
25           END
26         ELSE
27           BEGIN
28             fac := -fac;
29             termbf := Abs(term);
30           END;
31         END;
32   Result := 1.0;
33 END;

```

## References

- [1] M. ABRAMOWITZ, I.A. STEGUN, eds.: *Handbook of Mathematical Functions* 9th printing New York: Dover, 1964.
- [2] S. NOACK: *Statistische Auswertung von Mess- und Versuchsdaten mit Taschenrechner und Tischcomputer: Anleitungen und Beispiele aus dem Laborbereich* Berlin / New York: De Gruyter, 1980 ISBN: 9783110072631.
- [3] H.P. KINDER, G. OSINS, J. TIMM: *Statistik für Biologen* Braunschweig: Vieweg+Teubner, 1982 ISBN: 9783528033439.
- [4] W.H. PRESS et al.: *Numerical recipes in Pascal: The art of scientific computing* Cambridge: Cambridge University Press, 1989 ISBN: 9780521375160.
- [5] IBM-PC Special Interest GROUP: *PC-SIG Library* Computer software as CD-image 1994 URL: [https://archive.org/download/PC-Sig\\_Library\\_13th\\_Edition\\_PC-SIG\\_1994/PC-Sig%20Library%20%2813th%20Edition%29%20%28PC-SIG%29%20%281994%29.ISO](https://archive.org/download/PC-Sig_Library_13th_Edition_PC-SIG_1994/PC-Sig%20Library%20%2813th%20Edition%29%20%28PC-SIG%29%20%281994%29.ISO).
- [6] STUDENT: The Probable Error of a Mean, *Biometrika* **6**:1 (1908), 1–25 DOI: [10.2307/2331554](https://doi.org/10.2307/2331554).
- [7] F. R. HELMERT: Über die Wahrscheinlichkeit der Potenzsummen der Beobachtungsfehler und über einige damit im Zusammenhange stehende Fragen, *Z. Math. Physik* **21**:3 (1876), 192–218.

- [8] K.-C. LIANG: *Numerical Approximation to the Inverse Function of the Cumulative Chi-Square, t, and F Distribution*, MA thesis Logan, UT: Utah State University, 1968 URL: <https://digitalcommons.usu.edu/gradreports/1109>.
- [9] R.R. SOKAL, F.J. ROHLF: *Biometry: The Principles and Practice of Statistics in Biological Research* 2nd ed. New York: Freeman, 1981 ISBN: 978-0-7167-2411-7.
- [10] P.M. VOUTIER: *A New Approximation to the Normal Distribution Quantile Function* 2010 arXiv: [1002.0567 \[stat.CO\]](https://arxiv.org/abs/1002.0567).
- [11] V. PARETO: *Cours d'Économie Politique* vol. 2 Lousanne, Paris: Rouge, Pichon, 1897 URL: <https://web.archive.org/web/20160403032646/http://www.institutcoppel.org/wp-content/uploads/2012/05/Cours-d%C3%A9conomie-politique-Tome-II-Vilfredo-Pareto.pdf>.



# 9. Pseudo-random numbers of various distributions

## Abstract

This unit calculates pseudo-random numbers of various distributions

The interface is

Listing 9.1: Interface of unit RandomNumbers

```
1 UNIT Zufall;
2
3 INTERFACE
4
5 USES MathFunc, dos;
6
7 FUNCTION RandomLaplace(LowerLimit, UpperLimit: float): float;
8
9 FUNCTION RandomLaplace(LowerLimit, UpperLimit: LONGINT): LONGINT;
10
11 FUNCTION RandomBinomial(p: double; n: WORD): WORD;
12
13 FUNCTION RandomNormal(Average, SigmaSqr: double): double;
14
15 FUNCTION RandomExponential(Average: double): double;
16
17 FUNCTION RandomPoisson(Average: double): WORD;
18
19 FUNCTION RandomGamma(Form: WORD; Average: double): double;
20
21 FUNCTION RandomBernoulli(P: double): BYTE;
22
23 FUNCTION RandomGeometric(P: double): WORD;
24
25 FUNCTION RandomPareto(Minimum, Shape: double): double;
26
27 FUNCTION RandomChiSqr(DegFreedom: WORD): double;
28
29 FUNCTION RandomT(DegFreedom: WORD): double;
30
```

## 9. Pseudo-random numbers of various distributions

```
31 FUNCTION RandomF(f1, f2: WORD): double;  
32  
33 IMPLEMENTATION
```

### 9.1. LAPLACE-distribution

RandomLaplace generates discrete, linearly distributed random numbers from [LowerLimit..UpperLimit]. There are two versions of this routine, one for floating point and the other LONGINT.

Listing 9.2: Laplace-distributed random numbers

```
1 FUNCTION RandomLaplace(LowerLimit, UpperLimit: float): float;  
2  
3 VAR  
4     dummy: float;  
5  
6 BEGIN  
7     IF LowerLimit > UpperLimit  
8     THEN  
9         BEGIN  
10            Dummy := LowerLimit;  
11            LowerLimit := UpperLimit;  
12            UpperLimit := Dummy;  
13        END;  
14        Result := LowerLimit + (UpperLimit - LowerLimit) * Random;  
15    END;
```

### 9.2. Binomial distribution

RandomBinomial returns the number of hits in n Trials with individual probability p.

Listing 9.3: Linear

```
1 FUNCTION RandomBinomial(p: double; n: WORD): WORD;  
2  
3 VAR  
4     i, Sum: WORD;  
5  
6 BEGIN  
7     Sum := 0;  
8     FOR i := 1 TO n DO  
9         IF Random <= p THEN INC(Sum);  
10        Result := Sum;  
11    END;
```

## 9.3. Normal distribution

`RandomNormal` produces normally distributed random numbers.

Listing 9.4: Normal

```

1 FUNCTION RandomNormal(Average, SigmaSqr: double): double;
2
3 BEGIN
4   Result := Sqrt(-2 * Ln(Random)) * Sin(2 * Pi * Random) *
5     SigmaSqr + Average;
6 END;
```

## 9.4. Exponential and POISSON distribution

Exponentially distributed random numbers simulate the time in between random events of a given average rate (number of events per unit time is then Poisson-distributed). Poisson-distributed random numbers simulate rare events per unit time (the time between events is then exponentially distributed). The  $\Gamma$ -distribution is a generalisation of exponential distribution, it allows the simulation of events with modus  $> 0$ , e.g., the life time of products until failure.

Listing 9.5: Exponential Poisson and Gamma

```

1 FUNCTION RandomExponential(Average: double): double;
2
3 BEGIN
4   Result := -Ln(Random) * Average;
5 END;
6
7
8 FUNCTION RandomPoisson(Average: double): WORD;
9
10 VAR
11   Count: WORD;
12   ProbZero, Product: double;
13
14 BEGIN
15   Count := 0;
16   Product := Random;
17   ProbZero := Exp(-Average);
18   WHILE Product > ProbZero DO
19     BEGIN
20       INC(Count);
21       Product := Product * Random;
22     END;
23   Result := Count;
```

## 9. Pseudo-random numbers of various distributions

```
24 END;  
25  
26  
27 FUNCTION RandomGamma(Form: WORD; Average: double): double;  
28  
29 VAR  
30     Zaehler: WORD;  
31     Product: double;  
32  
33 BEGIN  
34     Product := 1;  
35     FOR Zaehler := 1 TO Form DO  
36         Product := Product * Random;  
37     Result := -Average * Ln(Product);  
38 END;
```

## 9.5. BERNOULLI-distribution

Bernoulli-distributed random numbers are either 0 or 1, they simulate for example the result of a coin toss. The geometric distribution simulates the number of trials required to get a 1 in Bernoulli-distributed experiments.

Listing 9.6: Geometric and Bernoulli

```
1 FUNCTION RandomBernoulli(P: double): BYTE;  
2  
3 BEGIN  
4     Result := Ord(Random < P);  
5 END;  
6  
7  
8 FUNCTION RandomGeometric(P: double): WORD;  
9  
10 VAR  
11     Count: WORD;  
12  
13 BEGIN  
14     Count := 1;  
15     WHILE Random > P DO  
16         INC(Count);  
17     Result := Count;  
18 END;
```

## 9.6. $\chi^2$ -, $t$ - and F- distribution

The following routines return random numbers that are  $\chi^2$ ,  $t$  or F-distributed.

Listing 9.7: Distribution for decision statistics

```

1  FUNCTION RandomChiSqr(DegFreedom: WORD): double;
2
3  VAR
4      Zaehler: WORD;
5      Sum: double;
6
7  BEGIN
8      Sum := 0;
9      FOR Zaehler := 1 TO DegFreedom DO
10         Sum := Sum + Sqr(RandomNormal(0, 1));
11     Result := Sum;
12 END;
13
14
15 FUNCTION RandomT(DegFreedom: WORD): double;
16
17 VAR
18     Zaehler: WORD;
19     Sum: double;
20
21 BEGIN
22     Sum := 0;
23     FOR Zaehler := 1 TO DegFreedom DO
24         Sum := Sum + Sqr(RandomNormal(0, 1));
25     Result := RandomNormal(0, 1) * Sqrt(DegFreedom / Sum);
26 END;
27
28
29 FUNCTION RandomF(f1, f2: WORD): double;
30
31 BEGIN
32     Result := (RandomChiSqr(f1) / f1) / (RandomChiSqr(f2) / f2);
33 END;

```

## 9.7. PARETO-distribution

PARETO-distributed random numbers give the damage amount that is not exceeded with a probability ( $\text{Random} \times 100\%$ ). Minimum is the minimal damage per claim ( $>= 0$ ) and Shape the exponent of the PARETO-distribution ( $> 1$ ). PARETO-distributions (also called 80/20 distributions) also describe efforts of diminishing returns, for example when 20 %

## 9. Pseudo-random numbers of various distributions

of the total effort required achieve already 80 % of the result. Then higher and higher efforts are required to achieve less and less improvements.

Listing 9.8: Pareto

```
1 FUNCTION RandomPareto(Minimum, Shape: double): double;  
2  
3 BEGIN  
4     Result := Minimum / Pow((1 - Random), 1 / Shape);  
5 END;  
6  
7 END. // Zufall
```

# 10. Descriptive statistics

## Abstract

Descriptive statistics summarizes data vectors by parameters of position, dispersion, shape and concentration. This can be done assuming the data follow a given distribution (most commonly GAUSS' normal distribution), or it can be done parameter-free.

Good introductions to elementary statistics are [1, 2].

The **breakdown point** of a statistics is the number of data that can be replaced with arbitrary values before a measure can collapse to zero or explode to infinity. The arithmetic mean, for example, has a breakdown point of zero, because a single  $x_i = \infty \rightarrow \bar{x} = \infty$ . Similarly, for any  $x_i = 0 \rightarrow \bar{x} = 0, \tilde{x} = 0$ . For the median, the breakdown point is 50 %. This is the highest value possible, because with higher contamination it would no longer be possible to distinguish the distribution of the data from the distribution of the contamination. If the  $y\%$  highest and lowest values are trimmed (removed) from the data, the breakdown point of the trimmed arithmetic mean becomes  $y\%$ .

The **influence function** describes how a measure is affected by changing (or removing) a single data point (empirical influence function) or by a slight change in the distribution of the data. Ideally, the influence function should be bounded and differentiable (without discontinuities).

The **efficiency** of a parameter or procedure measures how many data points are required to achieve a given purpose. The **relative efficiency** compares the efficiency of two procedures by dividing the number of data required by one of them by the number of data required by the procedure considered the “best possible”. Because the efficiency of some procedures depends on sample size, the **asymptotic relative efficiency** measures the relative efficiency as the sample size grows toward infinity.

The unit **descript** has the following interface

Listing 10.1: Interface of descript

```
1UNIT Deskript;
2
3{ descriptive statistics on vectors and matrices }
4
5INTERFACE
6
7USES Math, mathfunc, Vector, Matrix;
8
9CONST
```

## 10. Descriptive statistics

```
10  DeskriptError: BOOLEAN = FALSE;
11
12{ **** Position **** }
13
14FUNCTION ArithmeticMean(Data: VectorTyp): float;
15
16FUNCTION GeometricMean(Data: VectorTyp): float;
17
18FUNCTION HarmonicMean(Data: VectorTyp): float;
19
20FUNCTION GeneralMean(Data, Gewichte: VectorTyp; Exponent: float): float;
21
22{ **** Scale **** }
23
24FUNCTION Gini(Data: VectorTyp; Mean: float): float;
25
26FUNCTION Covariance(CONST X, Y: VectorTyp): float;
27
28FUNCTION MeanDeviationFromMean(Data: VectorTyp): float;
29
30FUNCTION MedianDeviationFromMean(Data: VectorTyp): float;
31
32FUNCTION Variance(Data: VectorTyp): float;
33
34FUNCTION StandardDeviation(VAR Data: VectorTyp): double;
35
36FUNCTION CoefficientOfVariation(Mean, v: float; n: WORD): float;
37
38{ **** Moments **** }
39
40FUNCTION Mue(CONST Data: VectorTyp; Mean: float; k: WORD): float;
41
42FUNCTION Skewness(CONST Data: VectorTyp; Mean, StaDev: float): float;
43
44FUNCTION ExcessKurtosis(CONST Data: VectorTyp; Mean, StaDev: float): float;
45
46{ **** Grouped data **** }
47
48FUNCTION WeightedMean(Means, Vars, Lengths: VectorTyp): float;
49
50FUNCTION WeightedStandardDeviation(Means, Vars, Lengths: VectorTyp): float;
51
52{ **** Concentration **** }
53
54FUNCTION LorenzMuenzner(Data: VectorTyp; VAR xVektor, yVektor: VectorTyp):
      float;
```

```

55
56FUNCTION HerfindahlIndex(Data: VectorTyp): float;
57
58{ ***** non-parametric position ***** }
59
60FUNCTION HiMed(CONST SortedData: VectorTyp): float;
61
62FUNCTION LoMed(CONST SortedData: VectorTyp): float;
63
64FUNCTION Median(VAR Data: VectorTyp): float;
65
66FUNCTION WeightedHiMed(CONST SortedData: MatrixTyp): float;
67
68FUNCTION WeightedLoMed(CONST SortedData: MatrixTyp): float;
69
70FUNCTION WeightedMedian(VAR Data: MatrixTyp): float;
71
72FUNCTION Quantile(VAR Data: VectorTyp; q: float): float;
73
74FUNCTION TriMedian(Data: VectorTyp): float;
75
76FUNCTION HodgesLehmann(Data: VectorTyp): float;
77
78FUNCTION NaiveHodgesLehmann(Data: VectorTyp): float;
79
80{ ***** non-parametric scale ***** }
81
82FUNCTION InterQuantilDistance(Q1, Q3: float): float;
83
84FUNCTION MAD(Data: VectorTyp): double;
85
86FUNCTION StandardErrorOfMedian(Data: VectorTyp): float;
87
88FUNCTION QuantileDispersionCoefficient(Q1, Q3: float): float;
89
90FUNCTION Sn(VAR Data: VectorTyp): float;
91
92FUNCTION NaiveSn(VAR Data: VectorTyp): float;
93
94FUNCTION Qn(VAR Data: VectorTyp): float;
95
96FUNCTION NaiveQn(VAR Data: VectorTyp): float;
97
98{ ***** non-parametric moments ***** }
99
100FUNCTION Ell2(CONST SortedData: VectorTyp): float;

```

## 10. Descriptive statistics

```
101
102FUNCTION Ell3(CONST SortedData: VectorTyp): float;
103
104FUNCTION Ell4(CONST SortedData: VectorTyp): float;
105
106FUNCTION QuartileCoefficientOfSkewness(Q1, Q2, Q3: float): float;
107
108FUNCTION CentilCoeffKurtosis(Data: VectorTyp): float;
109
110{ ***** Standardise and normalise vector ***** }
111
112PROCEDURE MeanNormalise(VAR Data: VectorTyp);
113{ subtract arithmetic mean from all elements }
114
115PROCEDURE Z_Standardise(VAR Data: VectorTyp);
116{ subtract mean and divide by standard deviation }
117
118PROCEDURE RobustStandardise(VAR Data: VectorTyp);
119{ subtract median and divide by Qn }
120
121{ ***** Description of matrices ***** }
122
123PROCEDURE VarCovarMatrix(CONST Data: MatrixTyp; VAR VarCovar: MatrixTyp);
124
125PROCEDURE VarCov(CONST Data: MatrixTyp; VAR VarCovar: MatrixTyp);
126
127PROCEDURE MeanVector(CONST Data: MatrixTyp; VAR Mean: VectorTyp);
128
129PROCEDURE StaVector(CONST Data: MatrixTyp; VAR Sta: VectorTyp);
130
131{ ***** Standardise and normalise matrix columns ***** }
132
133PROCEDURE CentreMatrix(VAR A: MatrixTyp);
134
135PROCEDURE StandardiseMatrix(VAR A: MatrixTyp);
136
137PROCEDURE RobustStandardiseMatrix(VAR A: MatrixTyp);
138
139{ ***** Distances and outliers ***** }
140
141PROCEDURE MahalanobisDistance(CONST Data: MatrixTyp; VAR Dm: VectorTyp);
142
143PROCEDURE RobustDistance(CONST Data: MatrixTyp; VAR Dr: VectorTyp);
144
145
146IMPLEMENTATION
```

```

147
148 VAR
149   CH: CHAR;

```

## 10.1. Normally distributed data

### 10.1.1. Position

If we have a vector of data, we may want to characterise them by giving a value that is “typical” for all of them, in the sense that if we did not have a particular value, we could use this typical value as an estimate without too much error. This typical value is called position. The position of a data vector is given by its mean. The most commonly used mean is the arithmetic mean

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n} \quad (10.1)$$

For normally distributed data, the arithmetic mean is the maximum likelihood estimator for missing values, which is one reason for its popularity. In other words, for the arithmetic mean  $\sum_{i=1}^n (x_i - \bar{x}) = 0$  and  $\sum_{i=1}^n (x_i - \bar{x})^2$  is minimal.

When calculating the sum over all elements, we use the NEUMAIER-sum to prevent catastrophic loss of precision if the data are of very different magnitude. Because the function `NeumaierSum` ignores NaN, we have to use the function `ActualElements` rather than `VectorLength` for the denominator:

Listing 10.2: arithmetic mean

```

1 FUNCTION ArithmeticMean(Data: VectorTyp): float;
2
3 VAR
4   i: WORD;
5
6 BEGIN
7   i := VectorLength(Data);
8   IF i > 0
9     THEN
10    Result := NeumaierSum(Data) / ActualElements(Data)
11   ELSE
12     BEGIN
13       ch := WriteErrorMessage('Arithmetic mean from vector of length 0');
14       DeskriptError := TRUE;
15     END;
16 END;

```

The geometric mean is defined as

$$\bar{x} = \sqrt[n]{\prod_{i=1}^n x_i} = \exp\left(\frac{1}{n} \sum_{i=1}^n \ln(x_i)\right) \forall x_i > 0 \quad (10.2)$$

## 10. Descriptive statistics

If at least one of the  $x_i$  is zero, the geometric mean becomes zero. Compared to the arithmetic mean,  $\bar{x} \leq \tilde{x}$ . The logarithm of the geometric mean is the arithmetic mean of the logarithms of the data, the geometric mean is the length of the sides of a hypercube that has the same volume as the hyper-brick with the side-lengths  $x_i$ . For implementation in a computer, the logarithmic form of the algorithm is preferred, as it is less likely to produce over- or under-flows. Even if the order of magnitude of the data were vastly different (in which case an average wouldn't make sense), the order of the logarithms is comparable. Therefore the use of a NEUMAIER-sum is not required.

Listing 10.3: Geometric mean

```

1  FUNCTION GeometricMean(Data: VectorTyp): float;
2
3  VAR
4      Sum, x: float;
5      n, i: WORD;
6
7  BEGIN
8      n := VectorLength(Data);
9      IF (n = 0)
10         THEN
11             BEGIN
12                 ch := WriteErrorMessage('Geometric mean from vector of length 0');
13                 DeskriptError := TRUE;
14                 EXIT;
15             END;
16     CASE VectorSignum(Data) OF
17         -1: BEGIN
18             ch := WriteErrorMessage('Geometric mean from negative data');
19             DeskriptError := TRUE;
20             EXIT;
21         end;
22         0: Result := 0; // product IS 0 if any of the factors is 0
23         1: BEGIN
24             n := 0;
25             Sum := 0;
26             FOR i := 1 TO VectorLength(Data) DO
27                 BEGIN
28                     x := GetVectorElement(Data, i);
29                     IF IsNaN(x)
30                         THEN
31                         ELSE
32                             BEGIN
33                                 INC(n);
34                                 Sum := Sum + Ln(x);
35                             END;
36             Result := Exp(Sum / n);

```

```

37      END;
38  END;
39 END; { case }
40 END;

```

The harmonic mean is defined as

$$\tilde{x} = \frac{n}{\sum_{i=1}^n x_i^{-1}} \quad \forall x_i > 0 \quad (10.3)$$

The reciprocal of the harmonic mean is the arithmetic mean of the reciprocals of the data. If at least one of the  $x_i$  is zero, the harmonic mean is defined as zero because  $\lim_{x_i \rightarrow 0} \tilde{x} = 0$ . The harmonic mean is smaller or equal than the geometric mean.

Listing 10.4: Harmonic mean

```

1 FUNCTION HarmonicMean(Data: VectorTyp): float;
2
3 VAR
4   Sum, x: float;
5   i, j, n: WORD;
6
7 BEGIN
8   n := VectorLength(Data);
9   IF (n = 0)
10  THEN
11    BEGIN
12      CH := WriteErrorMessage('Harmonic mean from vector of length 0');
13      DeskriptError := TRUE;
14      EXIT;
15    END;
16  CASE VectorSignum(Data) OF
17    -1: BEGIN
18      WriteErrorMessage('Harmonic mean from negative data');
19      DeskriptError := TRUE;
20      EXIT;
21    end;
22    0: Result := 0;
23    1: BEGIN
24      Sum := 0;
25      j := 0;
26      n := VectorLength(Data);
27      FOR i := 1 TO n DO
28        BEGIN
29          x := GetVectorElement(Data, i);
30          IF IsNaN(x)
31            THEN // ignore
32          ELSE
33            BEGIN

```

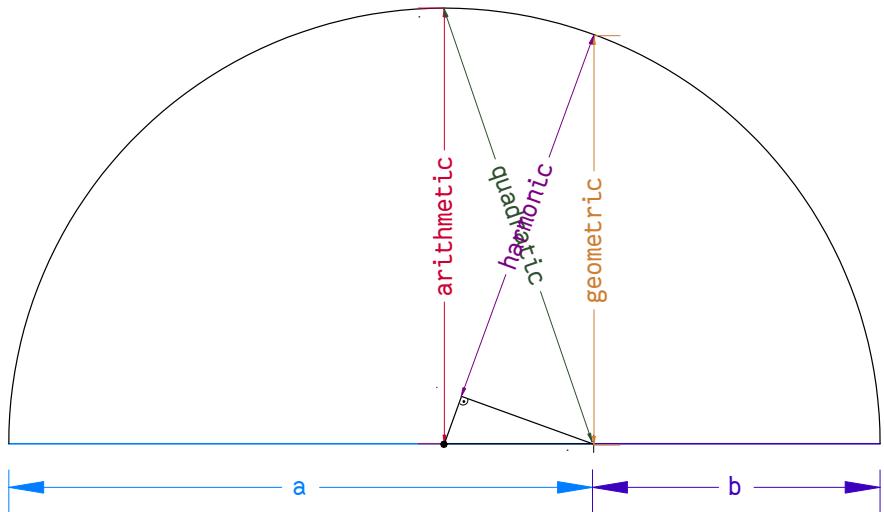


Figure 10.1.: Geometric interpretation of the various means of two data,  $a$  and  $b$ .

```

34             Sum := Sum + 1 / x;
35             INC(j);
36         END;
37     END;
38     Result := j / Sum;
39     END;
40 END; // CASE
41 END;

```

The quadratic mean (EUKLIDIAN mean, root mean square RMS) is defined as

$$\text{RMS} = \sqrt[2]{\frac{\sum_{i=1}^n x_i^2}{n}} \quad (10.4)$$

The cubic mean is defined analogously. The RMS is technically important to calculate the effective voltage and current in alternating power.

The weighted HÖLDER (general) mean is defined for  $x_i \in \mathbb{R} \geq 0$ , a whole number  $z \neq 0$  and a weight vector  $w$  with  $\sum_{i=1}^n w_i = 1$

$$M_w^z(x) = \sqrt[z]{\sum_{i=1}^n w_i x_i^z} \quad (10.5)$$

where in the unweighted case all  $w_i = 1/n$ . Depending on the choice of  $z$  we get different means:

$$\lim_{z \rightarrow -\infty} \text{minimum } M^{-\infty}(x) = \min(x_i)$$

$$z = -1 \text{ harmonic mean } M^{-1}(x) = \frac{n}{\sum_{i=1}^n x_i^{-1}}$$

$$\lim_{z \rightarrow 0} \text{geometric mean } M^0(\mathbf{x}) = \sqrt[n]{\prod_{i=1}^n x_i}$$

$$z = 1 \text{ arithmetic mean } M^1(\mathbf{x}) = \frac{\sum_{i=1}^n x_i}{n}$$

$$z = 2 \text{ quadratic mean } M^2(\mathbf{x}) = \sqrt{\frac{\sum_{i=1}^n x_i^2}{n}}$$

$$z = 3 \text{ cubic mean } M^3(\mathbf{x}) = \sqrt[3]{\frac{\sum_{i=1}^n x_i^3}{n}}$$

...

$$\lim_{z \rightarrow \infty} \text{maximum } M^\infty(\mathbf{x}) = \max(x_i)$$

Listing 10.5: HÖLDER mean

```

1  FUNCTION GeneralMean(Data, Gewichte: VectorTyp; Exponent: float): float;
2
3  VAR
4      i, n: WORD;
5      Sum: float;
6
7  BEGIN
8      Sum := 0;
9      n := VectorLength(Data);
10     IF (n = 0)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage('General mean from vector of length 0');
14                 DeskriptError := TRUE;
15                 EXIT;
16             END;
17     IF (n <> VectorLength(Gewichte))
18         THEN
19             BEGIN
20                 CH := WriteErrorMessage( 'General mean: unequal vector lengths of
21                     data and weights');
22                 DeskriptError := TRUE;
23                 EXIT;
24             END;
25     FOR i := 1 TO n DO
26         Sum := Sum + GetVectorElement(Gewichte, i) * pot(GetVectorElement(Data,
27             i), Exponent);
28     Result := pot(Sum, 1 / Exponent);
29 END;
```

The **f-mean** is a generalisation of the HÖLDER-mean

$$M_w^f = f^{-1} \left( \sum_{i=1}^n w_i f(x_i) \right) \quad (10.6)$$

## 10. Descriptive statistics

where in case of the HÖLDER-mean  $f(\mathbf{x}) = \mathbf{x}^z$ . For example, if  $\mathbf{x}$  contains the returns of a capital from year 1 to  $n$ , then the f-mean with  $f(\mathbf{x}) = \ln(\mathbf{x} + 1)$  is the average return.

### 10.1.2. Dispersion

As discussed above, we can describe a data vector by a parameter of location, that is, a mean. However, in addition to the location, we also would like information about how much the data spread around that mean, *i.e.*, how much error are we likely to make, if we use the mean to fill in a missing value. This is measured by parameters of dispersion.

#### Variance and Covariance

The covariance of two data vectors  $\mathbf{x}, \mathbf{y}$  is defined as

$$s_{\mathbf{x}, \mathbf{y}}^2 = 1/n \sum_{i=1}^n (\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{y}_i - \bar{\mathbf{y}}) \quad (10.7)$$

The variance of a vector is the covariance with itself,  $\text{Covariance}(\mathbf{x}, \mathbf{x})$ .

To perform this calculation, two loops over the data are required, once to calculate the averages and once to calculate the variance. Here we use an alternative way that calculates both in a single loop:

$$\begin{aligned} s_1^2 &= 0.0 & (10.8) \\ \bar{\mathbf{x}}_1 &= \mathbf{x}_1 \\ \bar{\mathbf{y}}_1 &= \mathbf{y}_1 \\ s_i^2 &= s_{i-1} \times \frac{i-2}{i-1} + \frac{(\mathbf{x}_i - \bar{\mathbf{x}}_{i-1})(\mathbf{y}_i - \bar{\mathbf{y}}_{i-1})}{i}, \quad i = 2 \dots n \\ \bar{\mathbf{x}}_i &= \bar{\mathbf{x}}_{i-1} + \frac{\mathbf{x}_i - \bar{\mathbf{x}}_{i-1}}{i} \\ \bar{\mathbf{y}}_i &= \bar{\mathbf{y}}_{i-1} + \frac{\mathbf{y}_i - \bar{\mathbf{y}}_{i-1}}{i} \end{aligned}$$

Listing 10.6: Covariance of two vectors

```

1  FUNCTION Covariance(CONST X, Y: VectorTyp): float;
2
3  VAR
4    n, Count, i, j: WORD;
5    CH: CHAR;
6    xi, yi, xAv, yAv, s: float;
7
8  BEGIN
9    n := VectorLength(X);
10   IF (n = 0)
11     THEN

```

```

12   BEGIN
13     CH := WriteErrorMessage('Covariance from vector of length 0');
14     DeskriptError := TRUE;
15     EXIT;
16   END;
17   IF (n <> VectorLength(Y))
18   THEN
19     BEGIN
20       CH := WriteErrorMessage('Covariance of vectors of unequal length ');
21       DeskriptError := TRUE;
22       EXIT;
23     END;
24   j := 0;
25   REPEAT                                // find first complete pair OF data
26     INC(j);
27     xi := GetVectorElement(X, j);
28     yi := GetVectorElement(Y, j);
29   UNTIL NOT (IsNaN(xi) OR IsNaN(yi));
30   xAv := xi;
31   yAv := yi;
32   s := 0.0;
33   Count := 1;
34   FOR i := Succ(j) TO n DO
35     BEGIN
36       xi := GetVectorElement(X, i);
37       yi := GetVectorElement(Y, i);
38       IF (IsNaN(xi) OR IsNaN(yi))
39       THEN
40       ELSE
41         BEGIN
42           INC(Count);
43           s := s * ((Count - 2) / (Count - 1)) + (xi - xAv) * (yi - yAv)
44             / Count;
45           xAv := xAv + (xi - xAv) / Count;
46           yAv := yAv + (yi - yAv) / Count;
47         END;
48     END;
49   Result := s;
50
51
52 FUNCTION Variance(Data: VectorTyp): float;
53
54 BEGIN
55   Result := Covariance(Data, Data);
56 END;

```

## 10. Descriptive statistics

The standard deviation  $\sigma$  of a data vector is the square root of its variance. It has the same unit as the mean, which makes them comparable. If the data are normally distributed, 68.3 % of the data will be within the range  $\pm\sigma$ , 95.4 % within  $\pm 2\sigma$  and 99.7 % within  $\pm 3\sigma$ .

```

1  function StandardDeviation(var Data: VectorTyp): float;
2
3  begin
4      Result := sqrt(Covariance(Data, Data));
5  end;
```

The standard error of the mean is  $\sigma_{\bar{x}} = \frac{\sqrt{\sigma^2}}{\sqrt{n}}$ , thus, increasing  $n$  will decrease this error, but with diminishing return.

```

1  FUNCTION StandardErrorOfMean(v: float; n: WORD): float;
2
3  BEGIN
4      IF n > 0
5          THEN
6              StandardErrorOfMean := Sqrt(v) / Sqrt(n)
7          ELSE
8              BEGIN
9                  WriteErrorMessage(' Standard error of mean for n = 0');
10                 DeskriptError := TRUE;
11             END;
12     END;
```

The coefficient of variation (unitised risk) is the standard deviation divided by the absolute value of the mean:  $CV = \frac{\sqrt{\sigma^2}}{|\mu|}$ , often multiplied with 100 %. It is used to express the precision of an assay. As a number without unit it is independent of the unit in which the data were measured. Warning: If the data are not measured on a rational scale, the mean may be zero (*e.g.*, temperatures on the C scale near freezing) and the CV is affected by small fluctuations of the mean.

```

1  FUNCTION CoefficientOfVariation(Mean, v: float; n: WORD): float;
2
3  BEGIN
4      IF Abs(mean) > Zero
5          THEN
6              Result := Sqrt(v) / Abs(Mean)
7          ELSE
8              BEGIN
9                  WriteErrorMessage(' Coefficient of variation with mean 0');
10                 DeskriptError := TRUE;
11             end;
12     END;
```

## Other measures of dispersion

Alternatively, one can calculate the distance of all data points from the mean, and then determine the mean or median of those:

```

1  FUNCTION MeanDeviationFromMean(Data: VectorTyp): float;
2
3  VAR
4      Mean, Sum: float;
5      i, j, n: WORD;
6
7  BEGIN
8      n := VectorLength(Data);
9      IF (n = 0)
10         THEN
11             BEGIN
12                 CH := WriteErrorMessage('Mean deviation of mean from vector of
13                     length 0');
14                 DeskriptError := TRUE;
15                 EXIT;
16             END;
17             Mean := ArithmeticMean(Data);
18             Sum := 0;
19             FOR i := 1 TO n DO
20                 Sum := Sum + Abs(Mean - GetVectorElement(Data, i));
21             Result := Sum / n;
22         END;
23
24  FUNCTION MedianDeviationFromMean(Data: VectorTyp): float;
25
26  VAR
27      Mean: float;
28      Abweichungen: VectorTyp;
29      i: WORD;
30
31  BEGIN
32      CreateVector(Abweichungen, VectorLength(Data), 0.0);
33      Mean := ArithmeticMean(Data);
34      CreateVector(Abweichungen, VectorLength(Data), 0.0);
35      FOR i := 1 TO VectorLength(Data) DO
36          SetVectorElement(Abweichungen, i, Abs(Mean - GetVectorElement(Data,
37              i)));
38      ShellSort(Abweichungen);
39      MedianDeviationFromMean := Median(Abweichungen);
40      DestroyVector(Abweichungen);
41  END;
```

### 10.1.3. PEARSON moments

Moments describe the shape of a distribution curve. The 0th moment is the total probability, that is, the integral of the distribution curve and hence always unity. The 1st moment is the expected value, that is, the mean. The 2nd moment is the variance. The 3rd moment is the skewness (lopsidedness, asymmetry), the 4th moment the kurtosis (heaviness of the tail). Higher moments can be defined, but are not commonly used. The moment is calculated from

$$\mu_k = \frac{1}{n} \sum_{i=1}^n (\bar{x}_i - \bar{x})^k \quad (10.9)$$

They are then standardised by dividing them by  $\sigma^k$ .

```

1  FUNCTION Mue(CONST Data: VectorTyp; Mean: float; k: WORD): float;
2
3  VAR
4      Sum: float;
5      i, n: WORD;
6
7  BEGIN
8      Sum := 0;
9      n := VectorLength(Data);
10     IF (n = 0)
11     THEN
12         BEGIN
13             CH := WriteErrorMessage('Mue from vector of length 0');
14             DeskriptError := TRUE;
15             EXIT;
16         END;
17     FOR i := 1 TO n DO
18         Sum := Sum + pot(GetVectorElement(Data, i) - Mean, k);
19     Result := Sum / n;
20 END;
```

Skewness,  $\mu_3/\sigma^3$ , is a measure of asymmetry of a distribution, and can be

**negative** tail is on the left side, left-skewed = right-leaning

**zero** the tails balance, but the distribution is not necessarily symmetrical

**positive** tail is on the right side, right-skewed = left-leaning

, at least for simple unimodal distributions.

```

1  FUNCTION Skewness(CONST Data: VectorTyp; Mean, StaDev: float): float;
2
3  BEGIN
4      Result := Mue(Data, Mean, 3) / pot(StaDev, 3);
5  END;
```

Alternative measures of skewness have been suggested by PEARSON:

$$\frac{\bar{x} - \text{mode}(x)}{\sigma}, \frac{3\bar{x} - \bar{x}}{\sigma} \quad (10.10)$$

The kurtosis (from Gr. curved, arching) is the 4th standardised PEARSON moment  $\mu_4/\sigma^4$ . Data within one standard deviation of the mean contribute little to its value, because raising the z-standardised datum to the 4th power will make them close to zero. Only data away from the peak – that is, outliers – contribute to the kurtosis. The kurtosis parameter is a measure of the combined weight of the tails, relative to the centre of the distribution. The normal distribution has a kurtosis of 3. Thus, we get the **excess kurtosis** by subtracting 3 from the moment. The result can be

**negative**, then the distribution is called platykurtic and has fewer and less extreme outliers

**zero**, then the distribution is called mesokurtic and has about as many outliers

**positive**, then the distribution is called leptokurtic and has more, and more extreme outliers

than the normal distribution.

```

1 FUNCTION ExcessKurtosis(CONST Data: VectorTyp; Mean, StaDev: float): float;
2
3 BEGIN
4     Result := (Mue(Data, Mean, 4) / pot(StaDev, 4)) - 3.0;
5 END;
```

#### 10.1.4. Concentration

##### LORENZ-MÜNZNER-curve and GINI-coefficient

The LORENZ-MÜNZNER curve of an data vector  $x$  sorted in non-decreasing order (see fig. 10.2) is given by

$$L(j) = \sum_{i=1}^j \frac{x_i}{n\mu} = \sum_{i=1}^j l_j \quad (10.11)$$

and is the cumulative fraction of the data for every position  $j$ . If  $x_i = \mu \forall i$  (all data are the same), then a plot of  $L(j)$  vs  $j/n$  is a straight line through the origin with slope one (line of equality, red). In the other extreme, if all  $x_i$  except one are zero, then the LORENZ-MÜNZNER curve consist of two straight lines (purple), one from  $(0, 0)$  to  $(1, 0)$  and the second perpendicular to  $(1, 1)$ . Together with the line of equality, they form a triangle with area 0.5.

Real data will form a curve between those extremes, and the GINI-coefficient [3] is the area between this curve and the line of equality, divided by the area of the triangle:  $G = a/(a+b)$ , because  $a+b = 0.5$ , this is equal to  $G = 2a$ . If  $x_i \geq 0 \forall i$ , the GINI-coefficient

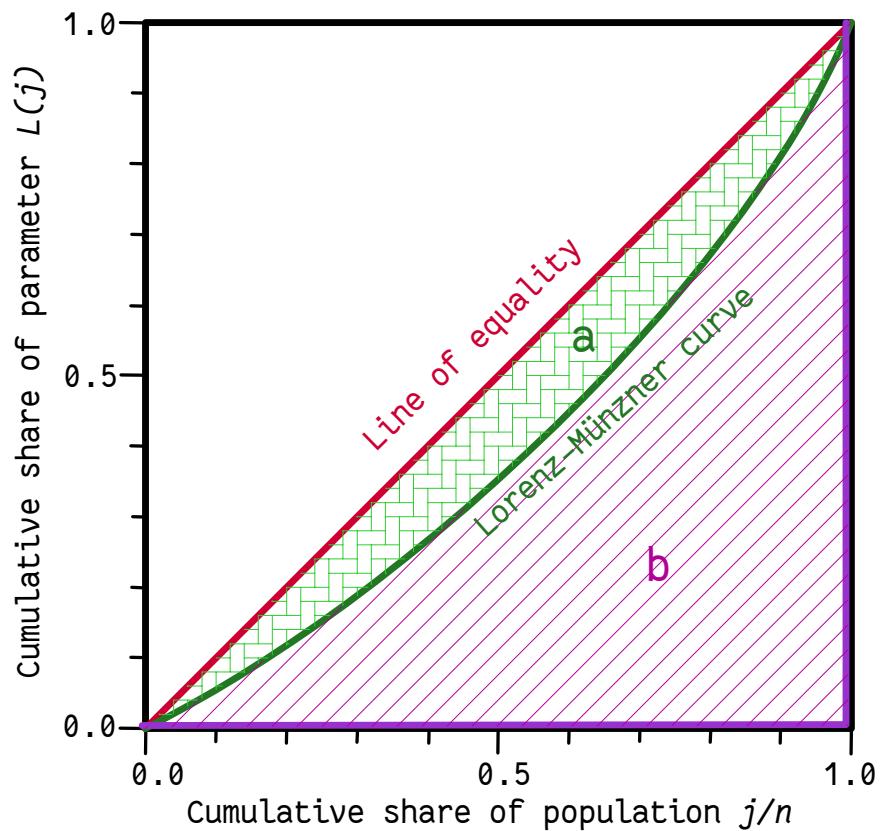


Figure 10.2.: LORENZ-MÜNZNER-plot. For details see text.

### 10.1. Normally distributed data

ranges from 0...1. If the parameter is distributed equally, and all data are on the line of equality, the GINI-coefficient is zero, in case of maximal inequality it becomes 1. Thus

$$G = 2a = \sum_{i=1}^n l_i \left( \frac{i-1}{n} + \frac{i}{n} \right) - \frac{1}{2} = \sum_{i=1}^n l_i \frac{2i-n-1}{2n} \quad (10.12)$$

```

1  FUNCTION LorenzMuenzner(Data: VectorTyp; VAR xVektor, yVektor: VectorTyp):
2      float;
3
4  VAR
5      n, i: WORD;
6      u, v, Sum, Gesamt, SumV: float;
7
8  BEGIN
9      n := VectorLength(Data);
10     IF (n = 0)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage('Lorenz-Muenzner from vector of length 0');
14                 DeskriptError := TRUE;
15                 EXIT;
16             END;
17     CreateVector(xVektor, n, 0.0);
18     CreateVector(yVektor, n, 0.0);
19     Gesamt := NeumaierSum(Data);
20     v := 0;
21     Sum := 0;
22     SumV := 0;
23     FOR i := 1 TO n DO
24         BEGIN
25             u := i / n;
26             Sum := Sum + GetVectorElement(Data, i);
27             v := Sum / Gesamt;
28             SetVectorElement(xVektor, i, u);
29             SetVectorElement(yVektor, i, v);
30             SumV := SumV + v;
31         END;
32     Result := (Succ(n) - 2 * SumV) / (Pred(n));
33 END;
```

Alternatively, one can define the GINI-Coefficient as half the sum of the absolute difference of all pairs of items  $|\bar{x}_i - \bar{x}_j|$  normalised by the average  $\bar{x}$ . This does not require sorting of the data vector:

$$G = \frac{\sum_{i=1}^n \sum_{j=1}^n |\bar{x}_i - \bar{x}_j|}{2 \sum_{i=1}^n \bar{x}_i} = \frac{\sum_{i=1}^n \sum_{j=1}^n |\bar{x}_i - \bar{x}_j|}{2n^2 \bar{x}} \quad (10.13)$$

## 10. Descriptive statistics

```

1  FUNCTION Gini(Data: VectorTyp; Mean: float): float;
2      { mittlere Abweichung aller Data voneinander }
3
4  VAR
5      i, j, n: WORD;
6      Sum: float;
7
8  BEGIN
9      Sum := 0;
10     n := VectorLength(Data);
11     IF (n = 0)
12         THEN
13             BEGIN
14                 CH := WriteErrorMessage('Gini coefficient from vector of length 0');
15                 DeskriptError := TRUE;
16                 EXIT;
17             END;
18     FOR i := 1 TO Pred(n) DO
19         FOR j := Succ(i) TO n DO
20             Sum := Sum + Abs(GetVectorElement(Data, i) - GetVectorElement(Data,
21                                             j));
22     Result := Sum / (2 * n * n * Mean);
23 END;
```

### The HERFINDAHL-HIRSCHMAN-Index

This measure of concentration is often used by anti-trust authorities to calculate the concentration of producers in a market. It is defined as

$$H = \sum_{i=1}^n \left( \frac{x_i}{\sum_{j=1}^n x_j} \right)^2 \quad (10.14)$$

, that is, the sum of the squared relative market shares, and reaches from  $1/n$  to 1.  $1/H$  is the effective number of competitors in the market. The index is equivalent to the SIMPSON **diversity index** in ecology or the **effective number of parties index** in politics. The squaring gives additional weight to the larger companies. In economy, we consider markets with  $H <$

**0.01** highly competitive

**0.15** unconcentrated

**0.25** moderately concentrated

**beyond** highly concentrated

One can normalise H to the range 0...1 by  $H^* = \frac{H-1/n}{1-1/n}$ .

```

1 FUNCTION HerfindahlIndex(Data: VectorTyp): float;
2
3 VAR
4   n, i: WORD;
5   Sum1, Sum2: float;
6
7 BEGIN
8   n := VectorLength(Data);
9   IF (n = 0)
10  THEN
11    BEGIN
12      CH := WriteErrorMessage('Herfindahl index from vector of length 0');
13      DescriptError := TRUE;
14      EXIT;
15    END;
16   Sum1 := NeumaierSum(Data);
17   Sum2 := 0;
18   FOR i := 1 TO n DO
19     Sum2 := Sum2 + Sqr(GetVectorElement(Data, i) / Sum1);
20   Result := Sum2;
21 END;

```

## 10.2. Non-parametric measures

### Location

**Median and related measures** The `LowMed` of a sorted data vector is the low median (the  $[(n+1) \text{ div } 2]$ -th smallest element  $\mathbf{x}_{((n+1) \text{ div } 2)}$ ) and `HiMed` the high median (the  $[(n \text{ div } 2)+1]$ -th smallest element  $\mathbf{x}_{((n \text{ div } 2)+1)}$ ). For odd  $n$ , low and high median are identical, for even  $n$  they are different. The common median  $\mathbf{\tilde{x}}$  is the average between `LoMed` and `HiMed`. It is equivalent to the 2nd quartile  $Q_2$  of the data. The  $k$ -th smallest element of a vector  $\mathbf{x}$  is called its  $k$ -th order statistics,  $\mathbf{x}_{(k)}$ .

Listing 10.7: Median of a data vector

```

1 FUNCTION HiMed(CONST SortedData: VectorTyp): float;
2
3 VAR
4   n: WORD;
5
6 BEGIN
7   n := VectorLength(SortedData);
8   IF (n = 0)
9   THEN
10  BEGIN
11    CH := WriteErrorMessage('Hi median from vector of length 0');

```

## 10. Descriptive statistics

```

12         DeskriptError := TRUE;
13     END
14 ELSE
15     Result := GetVectorElement(SortedData, Succ(n DIV 2));
16 END;
17
18 FUNCTION LoMed(CONST SortedData: VectorTyp): float;
19
20 VAR
21     n: WORD;
22
23 BEGIN
24     n := VectorLength(SortedData);
25     IF (n = 0)
26     THEN
27         BEGIN
28             CH := WriteErrorMessage('Low median from vector of length 0');
29             DeskriptError := TRUE;
30         END
31     ELSE
32         Result := GetVectorElement(SortedData, Succ(n) DIV 2);
33     END;
34
35 FUNCTION Median(VAR Data: VectorTyp): float;
36
37 VAR
38     n: WORD;
39     Sorted: VectorTyp;
40
41 BEGIN
42     n := VectorLength(Data);
43     IF (n = 0)
44     THEN
45         BEGIN
46             CH := WriteErrorMessage('Median from vector of length 0');
47             DeskriptError := TRUE;
48             EXIT;
49         END;
50     ShellSort(Data);
51     IF Odd(n)
52     THEN Median := LoMed(Data) // LOW AND Hi median are identical
53     ELSE Median := (LoMed(Data) + HiMed(Data)) / 2;
54 END;

```

The median is a special case of a quantile, namely 50 %, also called  $Q_2$  for second quartile. Other important quantiles are  $Q_1$  and  $Q_3$ , which correspond to the 25 % and 75 % quantiles, respectively. However, for the calculations of quantiles, we do not calcu-

late the average between neighbouring elements. Rather, we take the higher of the two elements.

Listing 10.8: Quantiles of a data vector

```

1 FUNCTION Quantile(VAR Data: VectorTyp; q: float): float;
2
3 VAR
4   n: WORD;
5
6 BEGIN
7   n := VectorLength(Data);
8   IF (n = 0)
9     THEN
10    BEGIN
11      CH := WriteErrorMessage('Quantile from vector of length 0');
12      DeskriptError := TRUE;
13      EXIT;
14    END;
15    ShellSort(Data);
16    Result := GetVectorElement(Data, Round(n * q + 0.5)); // always Round up
17 END;
```

The trimedian is a measure of location that is particularly robust against outliers. It is defined as  $\frac{Q_1+2Q_2+Q_3}{4}$ .

```

1 FUNCTION TriMedian(Data: VectorTyp): float;
2
3 VAR
4   n: WORD;
5
6 BEGIN
7   n := VectorLength(Data);
8   IF (n = 0) THEN
9     BEGIN
10      CH := WriteErrorMessage('Trimedian from vector of length 0');
11      DeskriptError := TRUE;
12      EXIT;
13    END;
14    Result := (Quantile(Data, 0.25) + 2 * Quantile(Data, 0.5) +
15               Quantile(Data, 0.75)) / 4;
16 END;
```

If the individual data have different reliabilities, then it is possible to assign weights to them so that  $\sum_{i=1}^n w_i = 1$ . Then, for data sorted in ascending order of  $x$ , the **LoMed** is the last element for which  $\sum_{i=1}^n w_i \leq 0.5$ , and the **HiMed** the first element for which this sum is  $\geq 0.5$ . Again, the median is the average of the two. Here, we assume that the data are in the first, and the weights in the second column of a matrix.

## 10. Descriptive statistics

```
1  FUNCTION WeightedHiMed(CONST SortedData: MatrixTyp): float;
2
3  VAR
4      i: WORD;
5      Sum: float;
6
7  BEGIN
8      Sum := 0.0;
9      i := 0;
10     REPEAT
11         INC(i);
12         sum := sum + GetMatrixElement(SortedData, i, 2);
13     UNTIL (sum >= 0.5);
14     WeightedHiMed := GetMatrixElement(SortedData, i, 1);
15 END;
16
17 FUNCTION WeightedLoMed(CONST SortedData: MatrixTyp): float;
18
19 VAR
20     i: WORD;
21     Sum: float;
22
23 BEGIN
24     Sum := 0.0;
25     i := 0;
26     REPEAT
27         INC(i);
28         sum := sum + GetMatrixElement(SortedData, i, 2);
29         IF (sum > 0.5) THEN DEC(i);
30     UNTIL (sum >= 0.5);
31     WeightedLoMed := GetMatrixElement(SortedData, i, 1);
32 END;
33
34 FUNCTION WeightedMedian(VAR Data: MatrixTyp): float;
35
36 VAR
37     n: WORD;
38
39 BEGIN
40     n := MatrixRows(Data);
41     IF (n = 0)
42         THEN
43             BEGIN
44                 CH := WriteErrorMessage('Weighted median from vector of length 0');
45                 DeskriptError := TRUE;
46                 EXIT;
47             END;
48     END;
```

```

47      END;
48      ShellSortMatrix(Data, 1);
49      WeightedMedian := (WeightedLoMed(Data) + WeightedHiMed(Data)) / 2.0;
50  END;

```

**Other measures of location** The median has a discontinuous influence function. Instead, one can use the HODGES-LEHMANN-estimator [4], which is the median of  $\frac{x_i+x_j}{2}$ ,  $i < j$ , that is, the median over all  $\binom{n}{2}$  pairwise averages of the data vector. The naive implementation would be

```

1  FUNCTION NaiveHodgesLehmann(Data: VectorTyp): float;
2
3  VAR
4      n, i, j, k: WORD;
5      x, y: float;
6      Averages: VectorTyp;
7
8  BEGIN
9      n := VectorLength(Data);
10     IF (n = 0)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage('Hodges-Lehmann estimator from vector of
14                     length 0');
15                 DeskriptError := TRUE;
16                 EXIT;
17             END;
18     n := Round(BinomialCoef(n, 2));
19     CreateVector(Averages, n, 0.0);
20     k := 0;
21     FOR i := 1 TO VectorLength(Data) DO
22         BEGIN
23             x := GetVectorElement(Data, i);
24             FOR j := 1 TO Pred(i) DO
25                 BEGIN
26                     INC(k);
27                     y := GetVectorElement(Data, j);
28                     SetVectorElement(Averages, k, (x + y) / 2);
29                 END;
30             Result := Median(Averages);
31             DestroyVector(Averages);
32         END;

```

The problem with this implementation is, that  $\binom{n}{2} \leq \text{MaxVector} = 10\,000$  is fulfilled only for  $n \leq 141$ . To find a more memory-efficient implementation, we first consider that all averages must be from the range  $\max(\mathbf{x}) - \min(\mathbf{x})$ . If we divide this range into

## 10. Descriptive statistics

a moderate number `NrBins` of subranges of equal size we can count the frequencies of averages in those subranges. The bin number `bins` is thus

$$\text{Bin}(\text{Average}) = \frac{\text{Average} - \min(\mathbf{x})}{\text{Step}}, \quad \text{Step} = \frac{\max(\mathbf{x}) - \min(\mathbf{x})}{\text{NrBins}} \quad (10.15)$$

To get to the median, we simply add the frequencies starting from subrange 0 until the sum reaches  $\binom{n}{2}/2$ . There is some quantisation error, but if the bin size is chosen small enough, this shouldn't matter. What does "small enough" mean? For data classification, the number of classes is commonly chosen as  $\sqrt{n}$ , so that the information loss and clarity are balanced and, in addition, most classes are occupied. Thus, the optimal number of classes would be  $\sqrt{\binom{n}{2}}$ , a maximum of 10000 bins would then be sufficient for a data vector of length 14142. The relative difference between the results of both implementations with synthetic data is in the order  $1 \times 10^{-3}$ .

```

1  FUNCTION HodgesLehmann(Data: VectorTyp): float;
2
3  VAR
4      min, max, Step, Sum, x, y: float;
5      Counts: VectorTyp;
6      i, j, n, NrBins, Bin: WORD;
7
8  BEGIN
9      n := VectorLength(Data);
10     IF (n = 0)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage('Hodges-Lehmann estimator from vector of
14                     length 0');
15                 DeskriptError := TRUE;
16                 EXIT;
17             END;
18             NrBins := Round(Sqrt(BinomialCoef(n, 2)));
19             min := FindSmallest(Data);
20             max := FindLargest(Data);
21             IF (min = max)
22                 THEN
23                     BEGIN
24                         CH := WriteErrorMessage('Hodges-Lehmann estimator: all data are the
25                             same');
26                         DeskriptError := TRUE;
27                         EXIT;
28                     END;
29             Step := (max - min) / NrBins;
30             CreateVector(Counts, Succ(NrBins), 0.0); // because vector starts at 1,
31                             rather than 0

```

```

29  FOR i := 1 TO n DO                                // count frequency OF grouped
   averages
  BEGIN
    x := GetVectorElement(Data, i);
  FOR j := 1 TO Pred(i) DO
    BEGIN
      y := GetVectorElement(Data, j);
      Bin := Succ(Round(((x + y) / 2 - min) / Step));
      SetVectorElement(Counts, Bin, GetVectorElement(Counts, Bin) +
        1.0);
    END;
  END;
  Sum := 0.0;
  i := 0;
  REPEAT                                         // identify bin OF median
    INC(i);
    Sum := Sum + GetVectorElement(Counts, i);
  UNTIL sum >= (BinomialCoef(n, 2) / 2);
  HodgesLehmann := Pred(i) * Step + min;
  DestroyVector(Counts);
END;

```

## Dispersion

The interquartile distance ( $0.5(Q_3 - Q_1)$ ) can be used as probable range of  $\bar{x}$  in the same sense as standard deviation. Although simple to calculate, the breakdown point of the interquartile distance is 25 %.

```

1 FUNCTION InterQuantilDistance(Q1, Q3: float): float;
2
3 BEGIN
4   Result := 0.5 * (Q3 - Q1);
5 END;

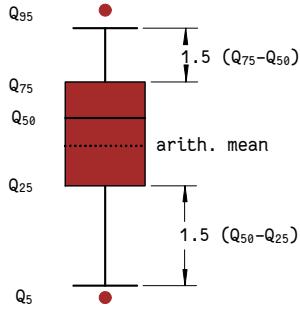
```

```

1 FUNCTION QuantileDispersionCoefficient(Q1, Q3: float): float;
2
3 BEGIN
4   QuantileDispersionCoefficient := (Q3 - Q1) / (Q3 + Q1);
5 END;

```

## 10. Descriptive statistics



The quartiles are used also to create **box-and-whisker plots**, where the box encompasses the interquartile range and the whiskers have the length of  $1.5(Q_{50} - Q_{25})$  and  $1.5(Q_{75} - Q_{50})$ , respectively. Data outside the whisker range are considered outliers. They can be placed individually, or the  $Q_5$  and  $Q_{95}$ -points are marked instead. Marking the arithmetic mean by a dotted line is optional, but gives a good idea about the skew of the data. This plot is very good for comparing data vectors obtained under different experimental conditions for position and spread.

The coefficient of quartile deviation  $\frac{Q_3 - Q_1}{Q_3 + Q_1}$  is another measure of dispersion.

```

1  FUNCTION StandardErrorOfMedian(Data: VectorTyp): float;
2
3  VAR
4      i, j, n: WORD;
5
6  BEGIN
7      n := VectorLength(Data);
8      IF (n = 0)
9          THEN
10             BEGIN
11                 CH := WriteErrorMessage('Weighted low median from vector of length
12                     0');
13                 DeskriptError := TRUE;
14                 EXIT;
15             END;
16             i := Round(n / 2 - Sqrt(3 * n) / 2);
17             j := Round(n / 2 + Sqrt(3 * n) / 2);
18             Result := (GetVectorElement(Data, j) - GetVectorElement(Data, i)) /
19                     3.4641;
20         END;

```

The median absolute deviation from the median (MAD) [5] is the median of  $|\mathbf{x}_i - \bar{\mathbf{x}}|$ , a robust measure of dispersion with a breakdown point of 50 % of the data. It is thus more useful than the interquartile distance. To make the MAD consistent with other measures of dispersion, it needs to be multiplied by a scaling parameter, to make it consistent with the standard deviation  $\sigma$ , this parameter is  $\frac{1}{\sqrt{2}} \text{qnorm}\left(\frac{5}{8}\right) = 1.4826$ . To detect outliers in the data vector, one can flag those  $\mathbf{x}_i$ , whose  $\frac{|\mathbf{x}_i - \bar{\mathbf{x}}|}{\text{MAD}(\mathbf{x})}$  exceeds a cutoff, for example 2.5 or 3.0. However, this assumes a distribution symmetrical around the median.

Listing 10.9: median absolute deviation from median (MAD)

```

1  FUNCTION MAD(Data: VectorTyp): double;
2
3  CONST
4      Scale = 1.4826; // 1 / (sqrt(2) * qnorm(5/8))
5
6  VAR
7      m, Sum, cn: float;
8      n, i: WORD;
9      Dev: VectorTyp;
10
11 BEGIN
12     n := VectorLength(Data);
13     IF (n = 0)
14         THEN
15             BEGIN
16                 CH := WriteErrorMessage('MAD from vector of length 0');
17                 DescriptError := TRUE;
18                 EXIT;
19             END;
20     m := Median(Data);
21     CreateVector(Dev, n, 0.0);
22     FOR i := 1 TO n DO
23         SetVectorElement(Dev, i, Abs(GetVectorElement(Data, i) - m));
24     CASE n OF
25         2: cn := 1.196;
26         3: cn := 1.495;
27         4: cn := 1.363;
28         5: cn := 1.206;
29         6: cn := 1.200;
30         7: cn := 1.140;
31         8: cn := 1.129;
32         9: cn := 1.107
33     ELSE
34         cn := n / (n - 0.8);
35     END; { case }
36     MAD := cn * Scale * Median(Dev);
37     DestroyVector(Dev);
38 END;

```

To calculate **MAD** as parameter of dispersion, one first has to calculate  $\bar{x}$  as parameter of position.

Dispersion parameters corresponding to the HODGES-LEHMANN-estimator of location are  $S_n$  and  $Q_n$  [5]:

$$S_n = c_n \times 1.1926 \times \text{lomed}_{i=1 \dots n} \left( \text{himed}_{j \neq i} |\bar{x}_i - \bar{x}_j| \right) \quad (10.16)$$

$$Q_n = c_n \times 2.2219 \times [|x_i - x_j|, i < j]_{(k)} \quad (10.17)$$

## 10. Descriptive statistics

The constant factor is for consistency at normal distribution,  $c_n$  is a finite sample correction factor. For a maximal breakdown point,  $k = \binom{n}{s}/4$ . For both  $S_n$  and  $Q_n$ , the breakdown point is  $(n \text{ div } 2) \text{ div } n$ , the best possible value [6]. GAUSSIAN asymptotic efficiency is 58 % for  $S_n$ , and even 82 % for  $Q_n$ . If we divide  $Q_n$  by  $\sqrt{n}$ , we get a robust standard error.

Listing 10.10: naive  $S_n$  and  $Q_n$ :  $\mathbf{O}(n^2)$

```

1  FUNCTION NaiveSn(VAR Data: VectorTyp): float;
2
3  VAR
4      i, j, n: WORD;
5      a1, a2: VectorTyp;
6      xi, cn: float;
7
8  BEGIN
9      n := VectorLength(Data);
10     IF (n = 0)
11         THEN
12             BEGIN
13                 CH := WriteErrorMessage('Sn from vector of length 0');
14                 DeskriptError := TRUE;
15                 EXIT;
16             END;
17     CreateVector(a1, n, 0.0);
18     CreateVector(a2, n, 0.0);
19     FOR i := 1 TO n DO
20         BEGIN
21             xi := GetVectorElement(Data, i);
22             FOR j := 1 TO n DO
23                 SetVectorElement(a1, j, Abs(xi - GetVectorElement(Data, j)));
24             ShellSort(a1);
25             SetVectorElement(a2, i, GetVectorElement(a1, Succ(n DIV 2))); // high
26                 median
27         END;
28     DestroyVector(a1);
29     ShellSort(a2);
30     CASE n OF
31         2: cn := 0.743;
32         3: cn := 1.851;
33         4: cn := 0.954;
34         5: cn := 1.351;
35         6: cn := 0.993;
36         7: cn := 1.198;
37         8: cn := 1.005;
38         9: cn := 1.131
39     ELSE IF Odd(n)

```

```

39         THEN cn := n / (n - 0.9)
40     ELSE cn := 1;
41 END; { case }
42 Result := cn * 1.1926 * GetVectorElement(a2, Succ(n) DIV 2); // LOW
43     median
44
45
46
47 FUNCTION NaiveQn(VAR Data: VectorTyp): float;
48
49 VAR
50     i, j, n, k: WORD;
51     a1: VectorTyp;
52     xi, dn: float;
53
54 BEGIN
55     n := VectorLength(Data);
56     IF (n = 0)
57     THEN
58         BEGIN
59             CH := WriteErrorMessage('Qn from vector of length 0');
60             DescriptError := TRUE;
61             EXIT;
62         END;
63     CreateVector(a1, (n - 1) * n DIV 2, 0.0);
64     k := 0;
65     FOR i := 1 TO n DO
66         BEGIN
67             xi := GetVectorElement(Data, i);
68             FOR j := Succ(i) TO n DO
69                 BEGIN
70                     INC(k);
71                     SetVectorElement(a1, k, Abs(xi - GetVectorElement(Data, j)));
72                 END;
73         END;
74     ShellSort(a1);
75     CASE n OF    // finite sample correction factor
76         2: dn := 0.399;
77         3: dn := 0.994;
78         4: dn := 0.512;
79         5: dn := 0.844;
80         6: dn := 0.611;
81         7: dn := 0.857;
82         8: dn := 0.669;
83         9: dn := 0.872;

```

## 10. Descriptive statistics

```

84     ELSE
85         IF Odd(n)      // n >= 10
86             THEN dn := n / (n + 1.4)
87             ELSE dn := n / (n + 3.8);
88     END; { case }
89     xi := GetVectorElement(a1, Round(BinomialCoef(n, 2)) DIV 4);
90     Result := dn * 2.2219 * xi;
91     DestroyVector(a1);
92 END;

```

These algorithms are  $\mathbf{O}(n^2)$ , and the naive Qn implementation is limited to  $n = 141$  by available memory. However, they can be made  $\mathbf{O}(n \log(n))$  with  $\mathbf{O}(n)$  storage space requirement [6].

Listing 10.11: final Sn:  $\mathbf{O}(n \log(n))$

```

1  FUNCTION Sn(VAR Data: VectorTyp): float;
2
3  VAR
4      n, i, j, l, nA, nB, rightA, rightB, leftA, leftB, tryA, tryB, diff,
5      Amin, Amax, even, half: WORD;
6      cn, medA, medB: float;
7      a2: VectorTyp;
8
9  BEGIN
10     n := VectorLength(Data);
11     IF (n = 0)
12         THEN
13             BEGIN
14                 CH := WriteErrorMessage('Sn from vector of length 0');
15                 DeskriptError := TRUE;
16                 EXIT;
17             END;
18     ShellSort(Data);
19     CreateVector(a2, n, 0.0);
20     SetVectorElement(a2, 1, GetVectorElement(Data, Succ(n DIV 2)) -
21         GetVectorElement(Data, 1));
22     FOR i := 2 TO (Succ(n) DIV 2) DO // a2(i) = lomed_{j>i} | xi - xj |, i =
23         2...Succ(n)/2
24         BEGIN
25             nA := Pred(i);
26             nB := n - i;
27             diff := nB - nA;
28             leftA := 1;
29             leftB := 1;
30             rightA := nB;
31             rightB := nB;
32             Amin := Succ(diff DIV 2);

```

```

32      Amax := diff DIV 2 + nA;
33      WHILE (leftA < rightA) DO
34          BEGIN
35              l := Succ(rightA - leftA);
36              even := 1 - (l MOD 2);
37              half := Pred(l) DIV 2;
38              tryA := leftA + half;
39              tryB := leftB + half;
40              IF (tryA < Amin)
41                  THEN
42                      BEGIN
43                          rightB := tryB;
44                          leftA := tryA + even;
45                      END
46              ELSE IF (tryA > Amax)
47                  THEN
48                      BEGIN
49                          rightA := tryA;
50                          leftB := tryB + even;
51                      END
52              ELSE
53                  BEGIN
54                      medA := GetVectorElement(Data, i) -
55                          GetVectorElement(Data, (i - tryA + Amin - 1));
56                      medB := GetVectorElement(Data, tryB + i) -
57                          GetVectorElement(Data, i);
58                      IF (medA >= medB)
59                          THEN
60                              BEGIN
61                                  rightA := tryA;
62                                  leftB := tryB + even;
63                              END
64                      ELSE
65                          BEGIN
66                              rightB := tryB;
67                              leftA := tryA + even;
68                          END;
69      END; { while }
70      IF (leftA > Amax)
71          THEN
72              SetVectorElement(a2, i, GetVectorElement(Data, leftB + i) -
73                               GetVectorElement(Data, i))
74      ELSE
75          BEGIN
76              medA := GetVectorElement(Data, i) - GetVectorElement(Data,

```

## 10. Descriptive statistics

```

76             Pred(i - leftA + Amin));
77             medB := GetVectorElement(Data, leftB + i) -
78                 GetVectorElement(Data, i);
79             SetVectorElement(a2, i, min(medA, medB));
80         END;
81     END; { for }
82 FOR i := Succ(Succ(n) DIV 2) TO Pred(n) DO      // same, but i =
83     Succ(Succ(n)/2) ... n-1
84 BEGIN
85     nA := n - i;
86     nB := Pred(i);
87     diff := nB - nA;
88     leftA := 1;
89     leftB := 1;
90     rightA := nB;
91     rightB := nB;
92     Amin := Succ(diff DIV 2);
93     Amax := diff DIV 2 + nA;
94     WHILE (leftA < rightA) DO
95         BEGIN
96             l := Succ(rightA - leftA);
97             even := 1 - (l MOD 2); // 0 OR 1
98             half := Pred(l) DIV 2;
99             tryA := leftA + half;
100            tryB := leftB + half;
101            IF (tryA < Amin)
102                THEN
103                    BEGIN
104                        rightB := tryB;
105                        leftA := tryA + even;
106                    END
107            ELSE IF (tryA > Amax)
108                THEN
109                    BEGIN
110                        rightA := tryA;
111                        leftB := tryB + even;
112                    END
113            ELSE
114                BEGIN
115                    medA := GetVectorElement(Data, Succ(i + tryA -
116                        Amin)) -
117                            GetVectorElement(Data, i);
118                    medB := GetVectorElement(Data, i) -
119                        GetVectorElement(Data, i - tryB);
120                    IF (medA >= medB)
121                        THEN
122

```

```

119      BEGIN
120          rightA := tryA;
121          leftB := tryB + even;
122      END
123      ELSE
124          BEGIN
125              rightB := tryB;
126              leftA := tryA + even;
127          END;
128      END; { while }
129  IF (leftA > Amax)
130  THEN
131      SetVectorElement(a2, i, GetVectorElement(Data, i) -
132                      GetVectorElement(Data, i - leftB))
133  ELSE
134      BEGIN
135          medA := GetVectorElement(Data, Succ(i + leftA - Amin)) -
136                      GetVectorElement(Data, i);
137          medB := GetVectorElement(Data, i) - GetVectorElement(Data, i -
138                      leftB);
139          SetVectorElement(a2, i, min(medA, medB));
140      END;
141  END; { for }
142  SetVectorElement(a2, n, GetVectorElement(Data, n) -
143                  GetVectorElement(Data, Succ(n) DIV 2));
144 CASE n OF           // finite sample correction factor
145     2: cn := 0.743;
146     3: cn := 1.851;
147     4: cn := 0.954;
148     5: cn := 1.351;
149     6: cn := 0.993;
150     7: cn := 1.198;
151     8: cn := 1.005;
152     9: cn := 1.131
153 ELSE
154     IF Odd(n)
155         THEN cn := n / (n - 0.9)
156         ELSE cn := 1;
157 END; { case }
158 ShellSort(a2);
159 Result := cn * 1.1926 * GetVectorElement(a2, Succ(n) DIV 2);
160 DestroyVector(a2);
161 END; { Sn }

```

Listing 10.12: Qn: doesn't give same result as naive Qn

## 10. Descriptive statistics

```

1  FUNCTION Qn(VAR Data: VectorTyp): float;
2
3  TYPE
4      IntVec = ARRAY [1..MaxVector] OF WORD;
5
6  VAR
7      work, weight: VectorTyp;
8      Left, Right, Q, P: IntVec;
9      n, h, i, j, jj, k, knew, kcand, jhelp, nL, nR, sumQ, sumP: WORD;
10     found: BOOLEAN;
11     trial, dn: float;
12
13     FUNCTION WeightedHiMedian(SortedData, weight: VectorTyp): float;
14
15  VAR
16      acand, iwcand: VectorTyp;
17      wrest, wleft, wmid, wright, i, n, nn: WORD;
18      wtotal: float;
19
20  BEGIN
21      n := VectorLength(SortedData);
22      nn := n;
23      wtotal := 0;
24      CreateVector(acand, n, 0.0);
25      CreateVector(iwcand, n, 0.0);
26      FOR i := 1 TO nn DO
27          wtotal := wtotal + GetVectorElement(weight, i);
28      wrest := 0;
29      WHILE TRUE DO      // infinite LOOP EXIT'ed when result known
30          BEGIN
31              Trial := GetVectorElement(SortedData, succ(nn div 2));
32              wleft := 0;
33              wmid := 0;
34              wright := 0;
35              FOR i := 1 TO nn DO
36                  IF (GetVectorElement(SortedData, i) < trial)
37                      THEN wleft := wleft + trunc(GetVectorElement(weight, i))
38                  ELSE IF (GetVectorElement(SortedData, i) > trial)
39                      THEN wright := wright + trunc(GetVectorElement(weight,
40                                         i));
41                  ELSE wmid := wmid + trunc(GetVectorElement(weight, i));
42          IF ((2 * wrest + 2 * wleft) > wtotal)
43              THEN
44                  BEGIN
45                      kcand := 0;
46                      FOR i := 1 TO nn DO

```

```

46      IF (GetVectorElement(SortedData, i) < trial)
47      THEN
48          BEGIN
49              Inc(kcand);
50              SetVectorElement(acand, kcand,
51                  GetVectorElement(SortedData, i));
52              SetVectorElement(iwcand, kcand,
53                  GetVectorElement(weight, i));
54          END;
55          nn := kcand;
56      END
57      ELSE
58          BEGIN
59              IF ((2 * wrest + 2 * wleft + 2 * wmid) > wtotal) // Endpunkt
60                  wird nicht erreicht
61              THEN
62                  BEGIN
63                      WeightedHiMedian := trial;
64                      exit;
65                  END
66              ELSE
67                  BEGIN
68                      kcand := 0;
69                      FOR i := 1 TO nn DO
70                          IF (GetVectorElement(SortedData, i) > trial)
71                          THEN
72                              BEGIN
73                                  Inc(kcand);
74                                  SetVectorElement(acand, kcand,
75                                      GetVectorElement(SortedData, i));
76                                  SetVectorElement(iwcand, kcand,
77                                      GetVectorElement(weight, i));
78                              END;
79                          nn := kcand;
80                          wrest := wrest + wleft + wmid;
81                      END; { else }
82                  END; { else }
83                  FOR i := 1 TO nn DO
84                      BEGIN
85                          SetVectorElement(SortedData, i, GetVectorElement(acand, i));
86                          SetVectorElement(weight, i, GetVectorElement(iwcand, i));
87                      END;
88                  END; { while }
89                  DestroyVector(acand);
90                  DestroyVector(iwcand);
91              END; { WeightedHiMedian }

```

## 10. Descriptive statistics

```

89
90 BEGIN
91     n := VectorLength(Data);
92     IF (n = 0)
93     THEN
94         BEGIN
95             ch := WriteErrorMessage('Qn from vector OF Length 0');
96             DeskriptError := TRUE;
97             EXIT;
98         END;
99         h := succ(n div 2);
100        k := h * pred(h) div 2;
101        ShellSort(Data);
102        CreateVector(work, n, 0.0);
103        CreateVector(weight, n, 0.0);
104        FOR i := 1 TO n DO
105            BEGIN
106                left[i] := n - i + 2;
107                right[i] := n;
108            END;
109            jhelp := n * succ(n) div 2;
110            knew := k + jhelp;
111            nL := jhelp;
112            nR := sqr(n);
113            found := False;
114            WHILE ((nR - nL > n) and (not found)) DO
115                BEGIN
116                    j := 1;
117                    FOR i := 2 TO n DO
118                        IF (left[i] <= right[i])
119                        THEN
120                            BEGIN
121                                SetVectorElement(weight, j, succ(right[i] - left[i]));
122                                jhelp := left[i] + trunc(GetVectorElement(weight, j)) div 2;
123                                SetVectorElement(work, j, GetVectorElement(Data, i) -
124                                    GetVectorElement(Data, succ(n) - jhelp));
125                                Inc(j);
126                            END;
127                            trial := WeightedHiMedian(work, weight);
128                            j := 0;
129                            FOR i := n DOWNTO 1 DO
130                                BEGIN
131                                    WHILE ((j < n) AND (GetVectorElement(Data, i) -
132                                        GetVectorElement(Data, n - j)
133                                            < trial)) DO
134                                    Inc(j);

```

```

134      P[i] := j;
135  END;
136  j := succ(n);
137  FOR i := 1 TO n DO
138    BEGIN
139      WHILE (GetVectorElement(Data, i) - GetVectorElement(Data, n - j +
140          2) > trial) DO
141        Dec(j);
142        Q[i] := j;
143      End;
144      sumP := 0;
145      sumQ := 0;
146      FOR i := 1 TO n DO
147        BEGIN
148          sumP := sumP + P[i];
149          sumQ := sumQ + pred(Q[i]);
150        END;
151      IF (knew <= sumP)      // problem about here
152      THEN
153        BEGIN
154          FOR i := 1 TO n DO
155            right[i] := P[i];
156          nR := sumP;
157        END
158      ELSE IF (knew > sumQ)
159      THEN
160        BEGIN
161          FOR i := 1 TO n DO
162            left[i] := Q[i];
163          nL := sumQ;
164        END
165      ELSE
166        BEGIN
167          Qn := trial;
168          found := True;
169        END;
170    END; { WHILE }
171  IF NOT (found)
172  THEN
173    BEGIN
174      j := 1;
175      FOR i := 2 TO n DO
176        IF (left[i] <= right[i])
177        THEN
178          FOR jj := left[i] TO right[i] DO
179            BEGIN

```

## 10. Descriptive statistics

```

179          SetVectorElement(work, j, GetVectorElement(Data, i) -
180                         GetVectorElement(Data, succ(n - jj)));
181          Inc(j); // !!!! grows beyond n !!!!
182      END;
183  END;
184 CASE n OF // finite sample correction factor
185   2: dn := 0.399;
186   3: dn := 0.994;
187   4: dn := 0.512;
188   5: dn := 0.844;
189   6: dn := 0.611;
190   7: dn := 0.857;
191   8: dn := 0.669;
192   9: dn := 0.872;
193 ELSE
194   IF odd(n)
195     THEN dn := n / (n + 1.4)
196     ELSE dn := n / (n + 3.8);
197 END; { case }
198 ShellSort(work);
199 Result := dn * 2.2219 * GetVectorElement(work, knew - nL);
200 DestroyVector(work);
201 DestroyVector(weight);
202 END;

```

### L-moments

The non-parametric equivalent of the PEARSON-moments are the L-moments. For an increasing ordered sample vector the  $r$ th L-moment is

$$\lambda_r = r^{-1} \binom{n}{r}^{-1} \sum_{x_1 < \dots < x_j < \dots < x_r} (-1)^{r-j} \binom{r-1}{j} x_j \quad (10.18)$$

This means that

$$\begin{aligned}
 \ell_1 &= \binom{n}{1}^{-1} \sum_{i=1}^n \mathfrak{x}_i = n^{-1} \sum_{i=1}^n \mathfrak{x}_i \\
 \ell_2 &= \frac{1}{2} \binom{n}{2}^{-1} \sum_{i=1}^n \left[ \binom{i-1}{1} - \binom{n-i}{1} \right] \mathfrak{x}_{(i)} \\
 &= \frac{1}{(n-1)n} \sum_{i=1}^n (2i - n - 1) \mathfrak{x}_{(i)} \\
 \ell_3 &= \frac{1}{3} \binom{n}{3}^{-1} \sum_{i=1}^n \left[ \binom{i-1}{2} - 2 \binom{i-1}{1} \binom{n-i}{1} + \binom{n-i}{2} \right] \mathfrak{x}_{(i)} \\
 &= \frac{2}{n^3 - 3n^2 + 2n} \sum_{i=1}^n \left[ 3i^2 - 3in - 3i + \frac{n^2}{2} + \frac{3n}{2} + 1 \right] \mathfrak{x}_{(i)} \\
 \ell_4 &= \frac{1}{4} \binom{n}{4}^{-1} \sum_{i=1}^n \left[ \binom{i-1}{3} - 3 \binom{i-1}{2} \binom{n-i}{1} + 3 \binom{i-1}{1} \binom{n-i}{2} - \binom{n-i}{3} \right] \mathfrak{x}_{(i)} \\
 &= \frac{6}{n^4 - 6n^3 + 11n^2 - 6n} \sum_{i=1}^n \left[ \frac{10i^3}{3} - 5i^2n - 5i^2 + 2in^2 + 5in + \frac{11i}{3} - \frac{n^3}{6} - n^2 - \frac{11n}{6} - i \right] \mathfrak{x}_{(i)}
 \end{aligned} \tag{10.19}$$

where  $\mathfrak{x}_{(i)}$  is the  $i$ th order statistics ( $i$ th smallest value) of the data vector. The first two L-moments have conventional names:

$\ell_1$  L-mean (identical to arithmetic mean)

$\ell_2$  L-dispersion (half the mean absolute difference)

L-moments, when standardised, are called

$\tau_2 = \ell_2/\ell_1$  coefficient of L-variation (identical to GINI-coefficient)

$\tau_3 = \ell_3/\ell_2$  L-skewness

$\tau_4 = \ell_4/\ell_2$  L-kurtosis

L-moments are more robust against outliers than conventional moments, and may exist for distributions where conventional moments cannot be defined (they require only a finite mean). For **Trimmed L-moments**, extreme values are left out during calculation for additional robustness.

Listing 10.13: L-moments

```

1 FUNCTION Ell2(CONST SortedData: VectorTyp): float;
2
3 VAR
4     sum, factor: float;

```

## 10. Descriptive statistics

```

5   i, n: WORD;
6
7 BEGIN
8   n := VectorLength(SortedData);
9   IF (n = 0)
10  THEN
11    BEGIN
12      CH := WriteErrorMessage('Ell2 from vector OF Length 0');
13      DeskriptError := TRUE;
14      EXIT;
15    END;
16   sum := 0;
17   FOR i := 1 TO n DO
18    BEGIN
19      factor := Pred(2 * i - n);
20      sum := sum + factor * GetVectorElement(SortedData, i);
21    END;
22   Factor := 1 / (n * n - n);
23   Result := Factor * Sum;
24 END;
25
26 FUNCTION Ell3(CONST SortedData: VectorTyp): float;
27
28 VAR
29   sum, factor: float;
30   i, n: WORD;
31
32 BEGIN
33   n := VectorLength(SortedData);
34   IF (n = 0)
35   THEN
36    BEGIN
37      CH := WriteErrorMessage('Ell3 from vector OF Length 0');
38      DeskriptError := TRUE;
39      EXIT;
40    END;
41   sum := 0;
42   FOR i := 1 TO n DO
43    BEGIN
44      factor := 3 * i * i - 3 * i * n - 3 * i + n * n / 2 + 3 * n / 2 + 1;
45      sum := sum + factor * GetVectorElement(SortedData, i);
46    END;
47   Factor := 2 / (n * n * n - 3 * n * n + 2 * n);
48   Result := Factor * Sum;
49 END;
50

```

```

51
52 FUNCTION Ell4(CONST SortedData: VectorTyp): float;
53
54 VAR
55   sum, factor: float;
56   i, n: WORD;
57
58 BEGIN
59   n := VectorLength(SortedData);
60   IF (n = 0)
61     THEN
62       BEGIN
63         CH := WriteErrorMessage('Ell4 from vector OF Length 0');
64         DescriptError := TRUE;
65         EXIT;
66       END;
67   sum := 0;
68   FOR i := 1 TO n DO
69     BEGIN
70       factor := 10 * i * i * i / 3 - 5 * i * i * n - 5 * i * i + 2 * i * n
71       * n + 5 * i * n +
72       11 * i / 3 - n * n * n / 6 - n * n - 11 * n / 6 - i;
73       sum := sum + factor * GetVectorElement(SortedData, i);
74     END;
75   Factor := 6 / (n * n * n * n - 6 * n * n * n + 11 * n * n - 6 * n);
76   Result := Factor * Sum;
77 END;

```

**Other moments** BOWLEY's measure of skewness [7] is  $Q_1 - 2Q_2 + Q_3$ , in effect the difference between the upper and lower part of the box in a box-and-whisker plot. It is an absolute measure of skewness in the units of the data. It can therefore not be used to compare the skewness of two vectors that, say, measured weight and height of individuals. For that purpose, the dimensionless relative skewness is used  $\gamma_B = \frac{Q_1 - 2Q_2 + Q_3}{(Q_3 - Q_1)}$ , which is bounded between [-1...1]. KELLY's definition uses the 10 and 90 % quantiles rather than  $Q_2$  and  $Q_3$  as limits, thereby using more of the data. Fundamentally, of course, one can set the limits quite arbitrarily [8].

```

1 FUNCTION QuartileCoefficientOfSkewness(Q1, Q2, Q3: float): float;
2
3 BEGIN
4   Result := (Q1 - 2 * Q2 + Q3) / (Q3 - Q1);
5 END;

```

The percentile coefficient of kurtosis compares the width of the curves for the central 50 % of the data with the width of the middle 80 %. Again, it is possible to use even wider ranges, but then the result should be scaled so that the kurtosis of the normal

## 10. Descriptive statistics

distribution is unity (1-99 % s = 1/3.49).

```

1  FUNCTION CentileCoeffKurtosis(Data: VectorTyp): float;
2
3  VAR
4      Q1, Q3, Q10, Q90: float;
5      n: WORD;
6
7  BEGIN
8      n := VectorLength(Data);
9      IF (n = 0)
10         THEN
11             BEGIN
12                 CH := WriteErrorMessage('Centile coefficient OF kurtosis from
13                     vector OF Length 0');
14                 DeskriptError := TRUE;
15                 EXIT;
16             END;
17             Q1 := Quantile(Data, 0.25);
18             Q3 := Quantile(Data, 0.75);
19             Q10 := Quantile(Data, 0.10);
20             Q90 := Quantile(Data, 0.90);
21             Result := 0.5 * (Q3 - Q1) / (Q90 - Q10);
22         END;

```

### 10.3. Normalisation and standardisation of vectors

The following routine centers a vector to mean zero by subtracting from each element the average:

$$c_i = x_i - \bar{x} \quad (10.20)$$

```

1  PROCEDURE MeanNormalise(VAR Data: VectorTyp);
2
3  VAR
4      Mean: double;
5      i, n: WORD;
6
7  BEGIN
8      n := VectorLength(Data);
9      IF (n = 0)
10         THEN
11             BEGIN
12                 CH := WriteErrorMessage('Normalisation OF vector OF Length 0');
13                 DeskriptError := TRUE;
14                 EXIT;
15             END;

```

```

16  Mean := ArithmeticMean(Data);
17  FOR i := 1 TO VectorLength(Data) DO
18      SetVectorElement(Data, i, (GetVectorElement(Data, i) - Mean));
19  END;

```

$z$ -standardising data works like centring, except that the data are also normalised to a standard deviation of 1.0:

$$\bar{z}_i = \frac{x_i - \bar{x}}{s(x)} \quad (10.21)$$

```

1 PROCEDURE Z_Standardise(VAR Data: VectorTyp);
2
3 VAR
4     Mean, s: double;
5     i, n: WORD;
6
7 BEGIN
8     n := VectorLength(Data);
9     IF (n = 0)
10        THEN
11            BEGIN
12                CH := WriteErrorMessage('Standardisation OF vector OF Length 0');
13                DescriptError := TRUE;
14                EXIT;
15            END;
16     Mean := ArithmeticMean(Data);
17     s := StandardDeviation(Data);
18     FOR i := 1 TO n DO
19         SetVectorElement(Data, i, (GetVectorElement(Data, i) - Mean) / s);
20 END;

```

Robust standardisation is performed essentially as  $z$ -standardisation, however, instead of the mean the median is subtracted from the data, and the data are divided by  $Q_n$  rather than the standard deviation. Note that the routine for calculating the median sorts the data, which would be a problem if used on the columns of a matrix. Therefore, the calculation of median and  $Q_n$  are performed on a copy of the data vector.

```

1 PROCEDURE RobustStandardise(VAR Data: VectorTyp);
2
3 VAR
4     Mean, s: double;
5     i, n: WORD;
6     Sorted: VectorTyp;
7
8 BEGIN
9     n := VectorLength(Data);
10    IF (n = 0)
11        THEN

```

## 10. Descriptive statistics

```

12      BEGIN
13          CH := WriteErrorMessage('Standardisation OF a vector OF Length 0');
14          DeskriptError := TRUE;
15          EXIT;
16      END;
17      CopyVector(Data, Sorted);
18      Mean := Median(Sorted); // so that order OF elements IS NOT changed
19      s := NaiveQn(Sorted);
20      FOR i := 1 TO n DO
21          SetVectorElement(Data, i, (GetVectorElement(Data, i) - Mean) / s);
22      DestroyVector(Sorted);
23  END;

```

## 10.4. Grouped data

```

1  FUNCTION WeightedMean(Means, Vars, Lengths: VectorTyp): float;
2
3  VAR
4      Sum1, Sum2: float;
5      i, n: WORD;
6
7  BEGIN
8      Sum1 := 0;
9      Sum2 := 0;
10     n := VectorLength(Means);
11     IF (n = 0)
12         THEN
13             BEGIN
14                 CH := WriteErrorMessage('Weighted mean from vector of length 0');
15                 DeskriptError := TRUE;
16                 EXIT;
17             END;
18     FOR i := 1 TO n DO
19         BEGIN
20             Sum1 := Sum1 + (GetVectorElement(Lengths, i) *
21                             GetVectorElement(Means, i) /
22                             GetVectorElement(Vars, i));
23             Sum2 := Sum2 + (GetVectorElement(Lengths, i) / GetVectorElement(Vars,
24                                         i));
25         END;
26     Result := Sum1 / Sum2;
27  END;
28
29  FUNCTION WeightedStandardDeviation(Means, Vars, Lengths: VectorTyp): float;
30

```

```

3  VAR
4      Sum1, Sum2: float;
5      i, n: WORD;
6
7  BEGIN
8      Sum1 := 0;
9      Sum2 := 0;
10     n := VectorLength(Means);
11     IF (n = 0)
12         THEN
13             BEGIN
14                 CH := WriteErrorMessage('Weighted standard deviation from vector of
15                     length 0');
16                 DeskriptError := TRUE;
17                 EXIT;
18             END;
19     FOR i := 1 TO n DO
20         BEGIN
21             Sum1 := Sum1 + (GetVectorElement(Lengths, i) - 1) *
22                 GetVectorElement(Vars, i);
23             Sum2 := Sum2 + GetVectorElement(Lengths, i);
24         END;
25     Result := Sum1 / (Sum2 - n);
26 END;

```

## 10.5. Descriptive statistics of matrices

### Variance-covariance matrix

The following routine to calculate the variance-covariance matrix  $\mathcal{S}$  is robust against missing data, but does not guarantee a positive-definite result. It calculates the matrix by calculating the covariance of all pairs of data vectors:

Listing 10.14: Variance-covariance matrix

```

1  PROCEDURE VarCovarMatrix(CONST Data: MatrixTyp; VAR VarCovar: MatrixTyp);
2
3  VAR
4      Rows, Columns, i, j: WORD;
5      x, y: VectorTyp;
6
7  BEGIN
8      Rows := MatrixRows(Data);
9      Columns := MatrixColumns(Data);
10     CreateMatrix(VarCovar, Columns, Columns, 0.0);
11     FOR i := 1 TO Columns DO

```

## 10. Descriptive statistics

```

12   BEGIN
13     GetColumn(Data, i, x);
14     SetMatrixElement(VarCovar, i, i, Covariance(x, x));
15     FOR j := Succ(i) TO Columns DO
16       BEGIN
17         GetColumn(Data, j, y);
18         SetMatrixElement(VarCovar, i, j, Covariance(x, y));
19         SetMatrixElement(VarCovar, j, i, GetMatrixElement(VarCovar, i,
20           j));
20         DestroyVector(y);
21       END;
22     DestroyVector(x);
23   END;
24 END;
```

The following routine also calculates  $\mathcal{S}$ , but uses a matrix approach. This ensures a positive definite result, but cannot handle missing data.

Listing 10.15: Variance-covariance matrix

```

1 PROCEDURE VarCov(CONST Data: MatrixTyp; VAR VarCovar: MatrixTyp);
2
3 VAR
4   Columns, Rows: WORD;
5   Ones, I1, Dev, DevT: MatrixTyp;
6
7 BEGIN
8   Rows := MatrixRows(Data);
9   Columns := MatrixColumns(Data);
10  CreateMatrix(Ones, Rows, Rows, 1.0);
11  MatrixInnerProduct(Ones, Data, I1);
12  SkalarMultiplikation(I1, 1 / Rows);
13  NegativeMatrix(I1);
14  MatrixAdd(Data, I1, Dev);           // deviation scores
15  DestroyMatrix(I1);
16  DestroyMatrix(Ones);
17  MatrixTranspose(Dev, DevT);
18  MatrixInnerProduct(DevT, Dev, VarCovar); // deviation score sum OF squares
19  SkalarMultiplikation(VarCovar, 1 / Pred(Rows));
20  DestroyMatrix(Dev);
21  DestroyMatrix(DevT);
22 END;
```

### Mean and standard deviation of columns

Listing 10.16: Column means of a matrix

```
1 PROCEDURE MeanVector(CONST Data: MatrixTyp; VAR Mean: VectorTyp);
```

```

2
3 VAR
4   Columns, i, j: WORD;
5   x: VectorTyp;
6
7 BEGIN
8   Columns := MatrixColumns(Data);
9   CreateVector(Mean, Columns, 0.0);
10  FOR j := 1 TO Columns DO
11    BEGIN
12      GetColumn(Data, j, x);
13      SetVectorElement(Mean, j, NeumaierSum(x) / ActualElements(x)); // arithmetic mean
14      DestroyVector(x);
15    END;
16  END;

```

Listing 10.17: Column standard deviation of a matrix

```

1 PROCEDURE StaVector(CONST Data: MatrixTyp; VAR Sta: VectorTyp);
2
3 VAR
4   Columns, Rows, i, j: WORD;
5   x: VectorTyp;
6
7 BEGIN
8   Rows := MatrixRows(Data);
9   Columns := MatrixColumns(Data);
10  CreateVector(Sta, Columns, 0.0);
11  FOR j := 1 TO Columns DO
12    BEGIN
13      GetColumn(Data, j, x);
14      SetVectorElement(Sta, j, StandardDeviation(x));
15      DestroyVector(x);
16    END;
17  END;

```

## Matrix standardisation and normalisation

```

1 PROCEDURE CentreMatrix(VAR A: MatrixTyp);
2
3 VAR
4   Data: VectorTyp;
5   j, Columns: WORD;
6
7 BEGIN

```

## 10. Descriptive statistics

```
8   Columns := MatrixColumns(A);
9   FOR j := 1 TO Columns DO
10    BEGIN
11      GetColumn(A, j, Data);
12      Centre(Data);
13      SetColumn(A, Data, j);
14      DestroyVector(Data);
15    END;
16  END;

17
18 PROCEDURE StandardiseMatrix(VAR A: MatrixTyp);
19
20 VAR
21   Data: VectorTyp;
22   j, Columns: WORD;
23
24 BEGIN
25   Columns := MatrixColumns(A);
26   FOR j := 1 TO Columns DO
27     BEGIN
28       GetColumn(A, j, Data);
29       Z_Standardise(Data);
30       SetColumn(A, Data, j);
31       DestroyVector(Data);
32     END;
33   END;

34
35 PROCEDURE RobustStandardiseMatrix(VAR A: MatrixTyp);
36
37 VAR
38   Data: VectorTyp;
39   j, Columns: WORD;
40
41 BEGIN
42   Columns := MatrixColumns(A);
43   FOR j := 1 TO Columns DO
44     BEGIN
45       GetColumn(A, j, Data);
46       RobustStandardise(Data);
47       SetColumn(A, Data, j);
48       DestroyVector(Data);
49     END;
50   END;
```

### The MAHALANOBIS-distance $D_m$

In multi-dimensional (uncorrelated) data, the distance of a point from the centre of the data cloud could be calculated as EUKLIDIAN distance. However, there are several problems:

- If the vectors have different units, what would be the unit of the distance? We would be comparing apples and oranges.
- If the variables have different scales, the larger variables would influence the result more than the smaller.
- If the variables have different standard deviations, the less reliable ones should receive smaller weight.

If instead of the variables themselves we used their  $z$ -scores for distance calculation, these objections would vanish, they are dimensionless, scaled in standard deviations, and more variable factors are given less weight [9]:

$$d_w = \sqrt{z_1^2 + z_2^2 + \dots + z_p^2} = \sqrt{\left(\frac{x_1 - \mu_1}{\sigma_1}\right)^2 + \left(\frac{x_2 - \mu_2}{\sigma_2}\right)^2 + \dots + \left(\frac{x_p - \mu_p}{\sigma_p}\right)^2} \quad (10.22)$$

If we use the identity matrix instead of the variance-covariance matrix, we get the EUKLIDIAN distance.

Squaring both sides and then dividing by  $d_w^2$  gives the equation of an ellipsoid around  $\mu$ , the vector of their column means:

$$1 = \frac{(x_1 - \mu_1)^2}{\sigma_1^2 d_w^2} + \frac{(x_2 - \mu_2)^2}{\sigma_2^2 d_w^2} + \dots + \frac{(x_p - \mu_p)^2}{\sigma_p^2 d_w^2} \quad (10.23)$$

All points on such an **probability density contour** are equally “close” to  $\mu$ . Vectorisation and allowing the variables to be correlated gives the MAHALANOBIS-distance  $D_m$  [10]:

$$D_m(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu}) \mathcal{S}^{-1} (\mathbf{x} - \boldsymbol{\mu})^\top} \quad (10.24)$$

Note that in some textbooks  $\mathbf{x}$  is defined as column vector, then the transposition needs to be done with the first rather than the third term.

The probability for a  $D_m^2$  follows the  $\chi^2$ -distribution with  $p$  degrees of freedom: All points satisfying  $D_m^2 \leq \chi^2(\alpha)$  have a probability  $1 - \alpha$ .

The problem with the MAHALANOBIS-distance for identification of outliers is that both the mean vector  $\mu$  and the variance-covariance matrix  $\mathcal{S}$  are influenced by outliers. Instead of the vector of arithmetic means one may use the vector of medians to define a more robust centroid. Another method is to calculate the MAHALANOBIS-distance of each point from the centroid by using  $\mu$  and  $\mathcal{S}$  calculated from the other  $n-1$  cases. Thus, if the datum under test were an outlier, it would not influence their calculation. However,

## 10. Descriptive statistics

for very large numbers of cases, the calculation of  $n$  inverses of  $n$  variance-covariance matrices may be impractical.

The MAHALANOBIS-distance is asymptotically distributed as  $\chi_d^2$ .  $\mathcal{S}$  is the sample covariance matrix, in this context also called shape matrix. If  $\beta$  denotes a constant probability level with  $0 \leq \beta \leq 1$ , then the probability of a random variable  $z \sim \chi_d^2$  being greater or equal to  $\chi_{\beta}^2$  is smaller than

$$P(z \leq \chi_{\beta}^2) = 1 - \beta = \alpha \quad (10.25)$$

where  $\alpha$  is called the significance level. Then the cutoff for the MAHALANOBIS-distance becomes  $L_{\beta} = \sqrt{\chi_{d,\beta}^2}$ . Any vector  $\mathbf{x}_i$  with  $D_i \geq \sqrt{\chi_{d;1-\alpha}^2}$  is a suspected outlier. The R-function `drawMahal` from the `chemometrics` package can draw ellipses of constant D into a data set.

**Example:** The data on age, length, weight and lead content of a fish (*Mercurio peces*) population were obtained from <https://www.r-bloggers.com/mahalanobis-distance-with-r-exercise/>, accessed 2018-11-19.

$$\mathbf{x} = \begin{pmatrix} 28 & 31 & 130.0 & 68.12 \\ 24 & 28 & 143.0 & 127.89 \\ 28 & 20 & 136.0 & 89.03 \\ 32 & 34 & 130.5 & 78.28 \\ 22 & 15 & 125.0 & 134.08 \\ 26 & 37 & 147.5 & 135.31 \\ 24 & 19 & 135.0 & 130.48 \\ 28 & 22 & 125.0 & 86.48 \\ 24 & 26 & 127.0 & 129.47 \\ 30 & 21 & 139.0 & 82.43 \\ 22 & 20 & 121.5 & 127.41 \\ 30 & 38 & 150.5 & 71.21 \\ 24 & 17 & 120.0 & 132.06 \\ 26 & 20 & 125.0 & 90.85 \end{pmatrix} \quad (10.26)$$

Then

$$\mu = (26.3 \ 24.9 \ 132.5 \ 105.9) \quad \mathcal{S} = \begin{pmatrix} 9.76 & 12.81 & 12.08 & -72.15 \\ 12.81 & 56.90 & 49.12 & -70.62 \\ 12.08 & 49.12 & 92.81 & -46.07 \\ -72.15 & -70.62 & -46.07 & 714.00 \end{pmatrix} \quad (10.27)$$

$$\mathcal{S}^{-1} = \begin{pmatrix} 0.5837 & -0.0418 & -0.0275 & 0.0531 \\ -0.0418 & 0.0390 & -0.0159 & -0.0014 \\ -0.0275 & -0.0159 & 0.0213 & -0.0030 \\ 0.0531 & -0.0014 & -0.0030 & 0.0064 \end{pmatrix} \quad (10.28)$$

Centralisation of the data matrix yields

$$(\mathbf{x} - \boldsymbol{\mu}) = \begin{pmatrix} 1.71 & 6.14 & -2.50 & -37.82 \\ -2.29 & 3.14 & 10.50 & 21.95 \\ 1.71 & -4.86 & 3.50 & -16.91 \\ 5.71 & 9.14 & -2.00 & -27.66 \\ -4.29 & -9.86 & -7.50 & 28.14 \\ -0.29 & 12.14 & 15.00 & 29.37 \\ -2.29 & -5.86 & 2.50 & 24.54 \\ 1.71 & -2.86 & -7.50 & -19.46 \\ -2.29 & 1.14 & -5.50 & 23.53 \\ 3.71 & -3.86 & 6.50 & -23.51 \\ -4.29 & -4.86 & -11.00 & 21.47 \\ 3.71 & 13.14 & 18.00 & -34.73 \\ -2.29 & -7.86 & -12.50 & 26.12 \\ -0.29 & -4.86 & -7.50 & -15.09 \end{pmatrix} \quad (10.29)$$

and the product  $(\mathbf{x} - \boldsymbol{\mu})\mathcal{S}^{-1}$  becomes

$$\begin{pmatrix} -1.195 & 0.260 & -0.085 & -0.153 \\ -0.589 & 0.021 & 0.171 & -0.016 \\ 0.210 & -0.293 & 0.155 & -0.021 \\ 1.541 & 0.188 & -0.263 & 0.119 \\ -0.390 & -0.125 & 0.031 & -0.010 \\ 0.472 & 0.206 & 0.047 & 0.112 \\ 0.144 & -0.206 & 0.136 & 0.037 \\ 0.294 & -0.037 & -0.103 & -0.008 \\ 0.018 & 0.195 & -0.142 & 0.045 \\ 0.903 & -0.376 & 0.167 & 0.032 \\ -0.856 & 0.135 & -0.103 & -0.050 \\ -0.720 & 0.120 & 0.176 & -0.098 \\ 0.724 & -0.049 & -0.156 & 0.095 \\ -0.558 & -0.037 & -0.030 & -0.083 \end{pmatrix} \quad (10.30)$$

Multiplying this result with  $(\mathbf{x} - \boldsymbol{\mu})^T$  yields

$$(x - \mu)S^{-1}(x - \mu)^T = \begin{pmatrix} 5.57 & -0.71 & -1.01 & -0.03 & -1.12 & -2.28 & -2.77 & 0.84 & -0.11 & -2.38 & 1.50 & 2.78 & -2 \\ -0.71 & 2.87 & -0.24 & -3.08 & 0.60 & 2.53 & 1.27 & -2.04 & 0.06 & -0.79 & 0.21 & 1.71 & -1 \\ -1.01 & -0.25 & 2.69 & -1.19 & 0.22 & -1.92 & 1.10 & 0.45 & -2.17 & 3.42 & -1.64 & 0.46 & -0 \\ -0.04 & -3.09 & -1.20 & 7.76 & -3.15 & 1.38 & -2.37 & 1.76 & 0.92 & 0.50 & -2.09 & -0.66 & 1 \\ -1.12 & 0.60 & 0.22 & -3.14 & 2.38 & -1.24 & 1.45 & -0.34 & 0.34 & -0.52 & 1.72 & -2.17 & 1 \\ -2.29 & 2.52 & -1.93 & 1.38 & -1.25 & 6.36 & 0.58 & -2.32 & 1.54 & -1.38 & -1.13 & 1.40 & -0 \\ -2.77 & 1.27 & 1.10 & -2.37 & 1.45 & 0.59 & 2.13 & -0.91 & -0.44 & 1.34 & -0.31 & -1.03 & 0 \\ 0.83 & -2.05 & 0.45 & 1.77 & -0.34 & -2.32 & -0.91 & 1.54 & -0.33 & 0.74 & -0.11 & -0.98 & 0 \\ -0.12 & 0.06 & -2.17 & 0.93 & 0.33 & 1.54 & -0.44 & -0.33 & 2.02 & -2.66 & 1.50 & -1.50 & 1 \\ -2.39 & -0.79 & 3.42 & 0.50 & -0.52 & -1.38 & 1.34 & 0.74 & -2.66 & 5.14 & -3.20 & 0.31 & -0 \\ 1.50 & 0.21 & -1.64 & -2.08 & 1.72 & -1.13 & -0.31 & -0.11 & 1.51 & -3.20 & 3.08 & -1.53 & 0 \\ 2.78 & 1.71 & 0.46 & -0.65 & -2.17 & 1.41 & -1.02 & -0.98 & -1.49 & 0.32 & -1.53 & 5.47 & -4 \\ -2.26 & -1.37 & -0.68 & 1.38 & 1.21 & -0.35 & 0.56 & 0.70 & 1.38 & -0.37 & 0.88 & -4.06 & 3 \\ 2.04 & -0.98 & 0.53 & -1.17 & 0.65 & -3.18 & -0.62 & 0.99 & -0.56 & -0.17 & 1.12 & -0.21 & -0 \end{pmatrix} \quad (10.31)$$

The diagonal elements are  $D_m^2(x)$ , the square root then is  $D_m(x)$ :

$$D_m^2(x) = \begin{pmatrix} 5.57 \\ 2.87 \\ 2.69 \\ 7.76 \\ 2.38 \\ 6.36 \\ 2.13 \\ 1.54 \\ 2.02 \\ 5.14 \\ 3.08 \\ 5.47 \\ 3.15 \\ 1.82 \end{pmatrix} \quad D_m(x) = \begin{pmatrix} 2.36 \\ 1.69 \\ 1.64 \\ 2.79 \\ 1.54 \\ 2.52 \\ 1.46 \\ 1.24 \\ 1.42 \\ 2.27 \\ 1.76 \\ 2.34 \\ 1.78 \\ 1.35 \end{pmatrix} \quad (10.33)$$

Listing 10.18: MAHALANOBIS distance

```

1 PROCEDURE MahalanobisDistance(CONST Data: MatrixTyp; VAR Dm: VectorTyp);
2
3 VAR
4   S, T, C, Inter, Res: MatrixTyp;
5   Mean: VectorTyp;
6   i, j, Rows, Columns: WORD;
7   Sum: float;

```

```

8
9 BEGIN
10   Rows := MatrixRows(Data);
11   Columns := MatrixColumns(Data);
12   CreateVector(Dm, Rows, 0.0);
13   CreateMatrix(C, Rows, Columns, 0.0);
14   MeanVector(Data, Mean);
15   VarCovarMatrix(Data, S);
16   InverseMatrix(S);
17   FOR i := 1 TO Rows DO                                // ( $\bar{x} - \mu$ )
18     FOR j := 1 TO Columns DO
19       SetMatrixElement(C, i, j, GetMatrixElement(Data, i, j) -
20                         GetVectorElement(Mean, j));
21   MatrixTranspose(C, T);                                // ( $\bar{x} - \mu$ ) $^T$ 
22   MatrixInnerProduct(C, S, Inter);                    // ( $\bar{x} - \mu$ )  $S^{-1}$ 
23   MatrixInnerProduct(Inter, T, Res);                  // ( $\bar{x} - \mu$ )  $S^{-1}$  ( $\bar{x} - \mu$ ) $^T$ 
24   FOR j := 1 TO Rows DO                          // Sqrt OF diagonal elements
25     SetVectorElement(Dm, j, Sqrt(GetMatrixElement(Res, j, j)));
26   DestroyMatrix(S);
27   DestroyMatrix(T);
28   DestroyMatrix(C);
29   DestroyMatrix(Inter);
30   DestroyMatrix(Res);
31   DestroyVector(Mean);
32 END;

```

### Penalised MAHALANOBIS distance

Calculating the MAHALANOBIS distance requires the calculation of  $\mathcal{S}^{-1}(\mathcal{X})$ . For a data matrix  $\mathcal{X}_{n \times p}$  with  $n \geq p$  and rank  $r = \text{rk}(\mathcal{S}_{p \times p}) < p$  the variance-covariance matrix will not be invertible. This happens if some variables are linear combinations of others, that is, there are actually fewer variables present than  $p$ . Then singular value decomposition yields

$$\mathcal{S}(\mathcal{X}) = \mathcal{X}^T \mathcal{X} \quad (10.34)$$

$$\text{svd}(\mathcal{X}^T \mathcal{X}) = \mathcal{V} \Sigma^2 \mathcal{V}^T = \mathcal{V} \mathcal{D} \mathcal{V}^T \quad (\mathcal{D} = \Sigma^2) \quad (10.35)$$

where  $\mathcal{V}$  and  $\Sigma$  are identical to those of  $\text{svd}(\mathcal{X})$ . Thus, in case of a singular variance-covariance matrix  $\mathcal{S}$  the following procedure can be used:

1. center data columns on  $\bar{\mathbf{x}}$  to calculate the penalised MAHALANOBIS distance. However, it is better to omit this step to calculate the EUKLIDIAN distance when data values around 0 contribute least to the classification problem. Then centralising will make nearest-neighbour solutions worse.
2. Calculate the variance-covariance  $\mathcal{S}(\mathcal{X}) = \mathcal{X}^T \mathcal{X}$ .

## 10. Descriptive statistics

3. calculate either the  $\text{svd}(\mathcal{X})$  or, if the table is too large, the  $\text{svd}(\mathcal{S})$  (both should be identical).

4. identify the first  $r$  significant singular values and calculate  $\mathfrak{d}_i = \sigma_i^2$

5. calculate

$$\mathcal{X}_{\text{trans}} = \mathcal{V}_{n \times r} \text{diag} \left( \sqrt{\frac{\mathfrak{d}_i}{\mathfrak{d}_i \sigma}} \right)$$

, that is, data weighted by the sqrt-term.

6. Calculate the EUKLIDIAN distance of data points in the transformed matrix.

Alternatively, one could use the MOORE-PENROSE pseudoinverse  $\mathcal{S}^+$  to calculate a MAHALANOBIS-like distance (see section 6.9.3 on page 221).

### Robust distance for outlier identification

The ordinary, sample-based MAHALANOBIS-distance as defined in 10.22 uses the arithmetic mean and the standard deviation for scaling. Both are sensitive to outliers. Thus, in a data set containing outliers, the 95 % probability ellipse becomes very large, and contains many actual outliers. These are therefore not identified, a problem known as **masking**.

One possible solution would be to use robust estimates for position (median, trimedian, HODGES-LEHMANN-estimator) and dispersion ([MAD](#),  $Q_n$ ,  $S_n$ ). A plot of robust distance *vs* standard MAHALANOBIS distance can then be used to identify outliers (**distance-distance plot**, see fig. 10.3).

Listing 10.19: Robust distance

```

1 PROCEDURE RobustDistance(CONST Data: MatrixTyp; VAR Dr: VectorTyp);
2
3 VAR
4   i, j, n, p: WORD;
5   Position, Scale, x: VectorTyp;
6   dist, Sum: float;
7
8 BEGIN
9   n := MatrixRows(Data);
10  p := MatrixColumns(Data);
11  CreateVector(Position, p, 0.0);
12  CreateVector(Scale, p, 0.0);
13  CreateVector(Dr, n, 0.0);
14  FOR j := 1 TO p DO
15    BEGIN
16      GetColumn(Data, j, x);
17      SetVectorElement(Position, j, HodgesLehmann(x));
18      SetVectorElement(Scale, j, NaiveQn(x));

```

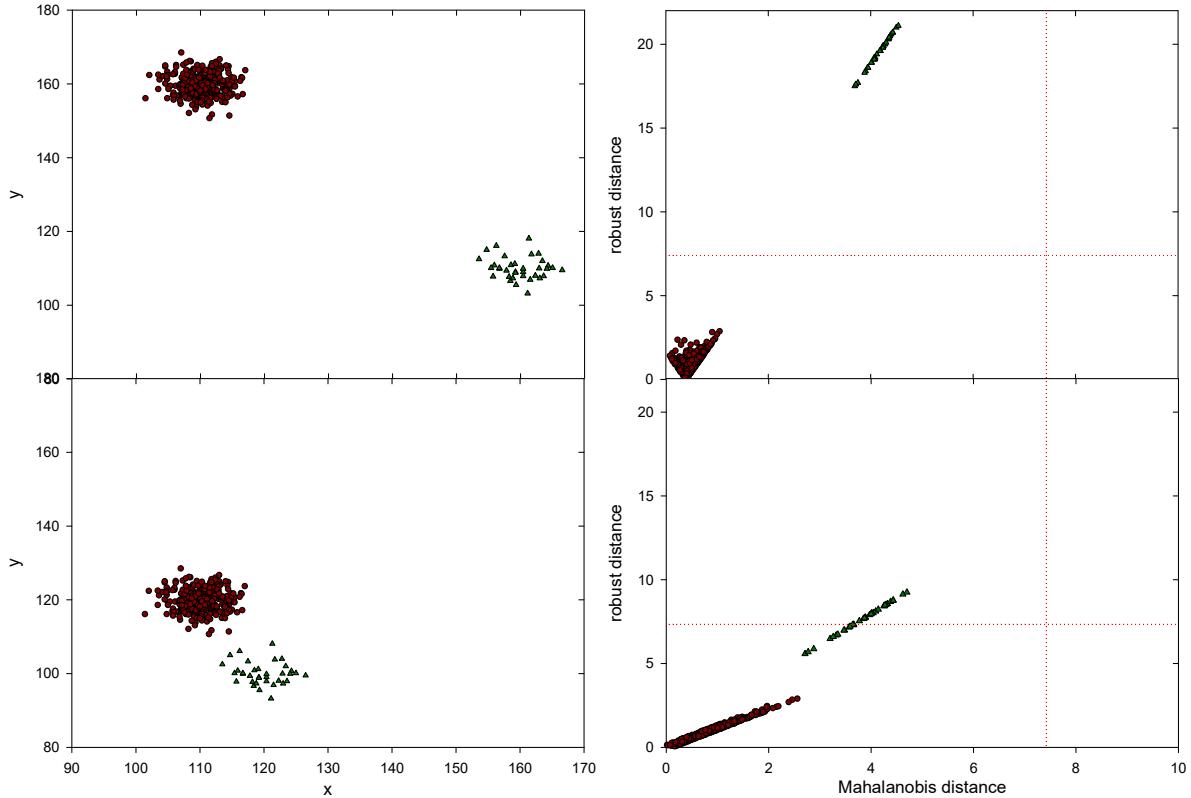


Figure 10.3.: Distance-distance plot. *Top left:* An artificial data set of 350 items of two variables with  $\bar{x} = 110 \pm 3$ ,  $\bar{y} = 160 \pm 3$  (*dark red*) is polluted with 10 % outliers with  $\bar{x} = 160 \pm 3$ ,  $\bar{y} = 110 \pm 3$  (*green*). Both are clearly separated (*left*). The robust distances calculated with the HODGES-LEHMANN-estimator and  $Q_n$ , but not the MAHALANOBIS distances, for the outlier are clearly beyond the critical value of  $\chi^2_{2,0.975} = 7.4$  (*red, dotted line*). *Bottom:* The coordinates are now  $\bar{x} = 110 \pm 3$ ,  $\bar{y} = 120 \pm 3$  and  $\bar{x} = 120 \pm 3$ ,  $\bar{y} = 110 \pm 3$ . Many of the robust distances of the outlier are still beyond the critical value.

## 10. Descriptive statistics

```

19      DestroyVector(x);
20  END;
21  FOR i := 1 TO n DO
22    BEGIN
23      sum := 0;
24      FOR j := 1 TO p DO
25        BEGIN
26          dist := (GetMatrixElement(Data, i, j) -
27                    GetVectorElement(Position, j)) / GetVectorElement(Scale, j);
28          Sum := Sum + Sqr(dist);
29        END;
30        SetVectorElement(Dr, i, Sqrt(Sum));
31    END;
32    DestroyVector(Position);
33    DestroyVector(Scale);
34 END;

```

### The minimum covariance determinant (MCD)

If the data are not normally distributed, both location and the covariance (shape) matrix need to be calculated in a robust way, otherwise outliers will affect both the mean and covariance matrix in such a way that outliers are missed (masking). This is particularly true, if the number of outliers approaches or exceeds  $\frac{n}{(p+1)}$ . If  $n/2 \leq h < n, h > p$  is selected (which requires at least  $n > 2p$ , better  $5p$ ), one can look for the  $h$  data points whose covariance matrix has the smallest determinant (minimum covariance determinant, MCD). The estimate of location is then the arithmetic mean of these  $h$  data [11, 12]. For maximal robustness,  $h = (n + p + 1)\text{div}2$ , then  $\alpha = \lim_{n \rightarrow \infty} \frac{h(n)}{n} = 0.5$ . However, the efficiency is quite low especially when  $p$  is small, this can be combatted by setting  $\alpha$  to higher values like 0.75.

In order to increase efficiency, it is possible to weigh the data, for example with

$$w(i) = \begin{cases} 1 \leftarrow d_i^2 \leq \chi_{p,0.975}^2 \\ 0 \leftarrow d_i^2 > \chi_{p,0.975}^2 \end{cases} \quad (10.36)$$

Then

$$\begin{aligned} \hat{\mu}_{\text{MCD}} &= \frac{\sum_{i=1}^n w(d_i^2) \mathbf{x}_i}{\sum_{i=1}^n w(d_i^2)} \\ \hat{\Sigma}_{\text{MCD}} &= c_1 n^{-1} \sum_{i=1}^n w(d_i^2) (\mathbf{x}_i - \hat{\mu}_{\text{MCD}})(\mathbf{x}_i - \hat{\mu}_{\text{MCD}})' \end{aligned} \quad (10.37)$$

$$c_1 = \frac{\alpha}{F_{\chi_{p+2}^2}(q_\alpha)} \quad (10.38)$$

where  $(q_\alpha)$  is the  $\alpha$ -quantile of the  $\chi_p^2$ -distribution. A robust correlation matrix can be

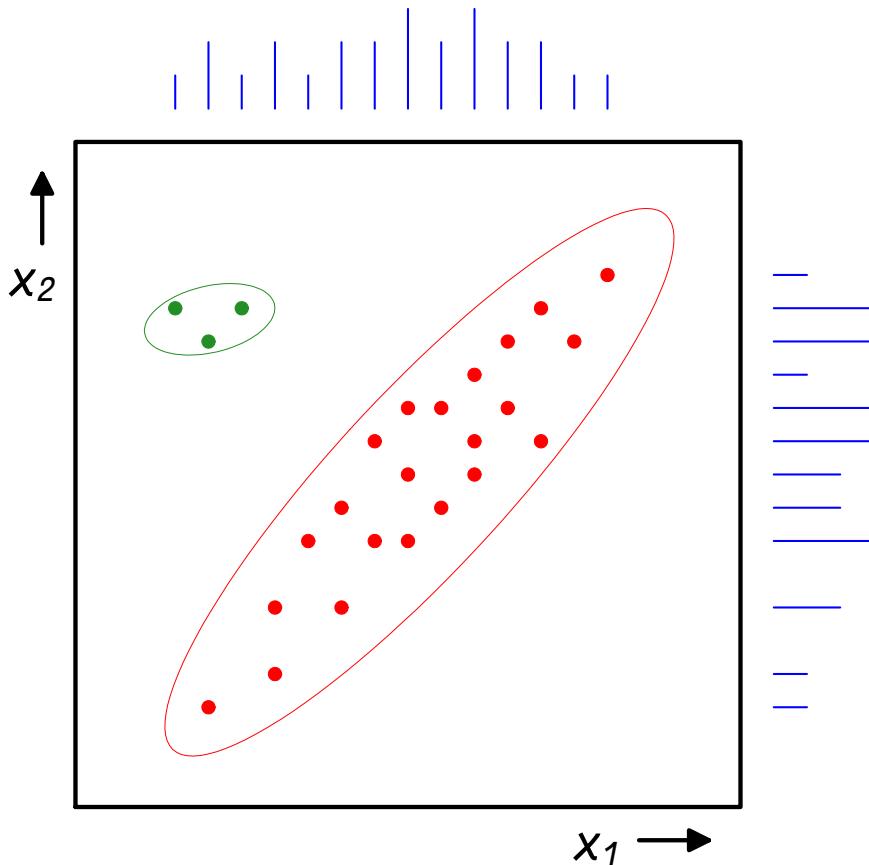


Figure 10.4.: Two variables may individually look normally distributed. However, the data set still may belong to two distinct populations, that is, the distribution is not multivariate normal.

calculated from  $r_{ij} = \frac{s_{ij}}{\sqrt{s_{ii}s_{jj}}}$ . Weighing does not affect the breakdown value, as long as the weight function goes to zero for large distances.

Another problem occurs if the data distribution is not multivariate normal. For example, if the data really contain two distinct populations, the centroid calculated will be located between the populations and have no meaning (see fig. 10.4). Of course,  $\mathcal{S}$  would also be meaningless. Methods to detect violation of multivariate normality are discussed in section 16.2.3 on page 515.

### The multivariate median

A data vector can be sorted, and hence its median determined. For a matrix, this is not possible. Several definitions of the multivariate median are used [13], of which the conceptually most useful are:

**vector of medians** is appropriate only if the variables in the data set are orthogonal (uncorrelated, say, after PCA).

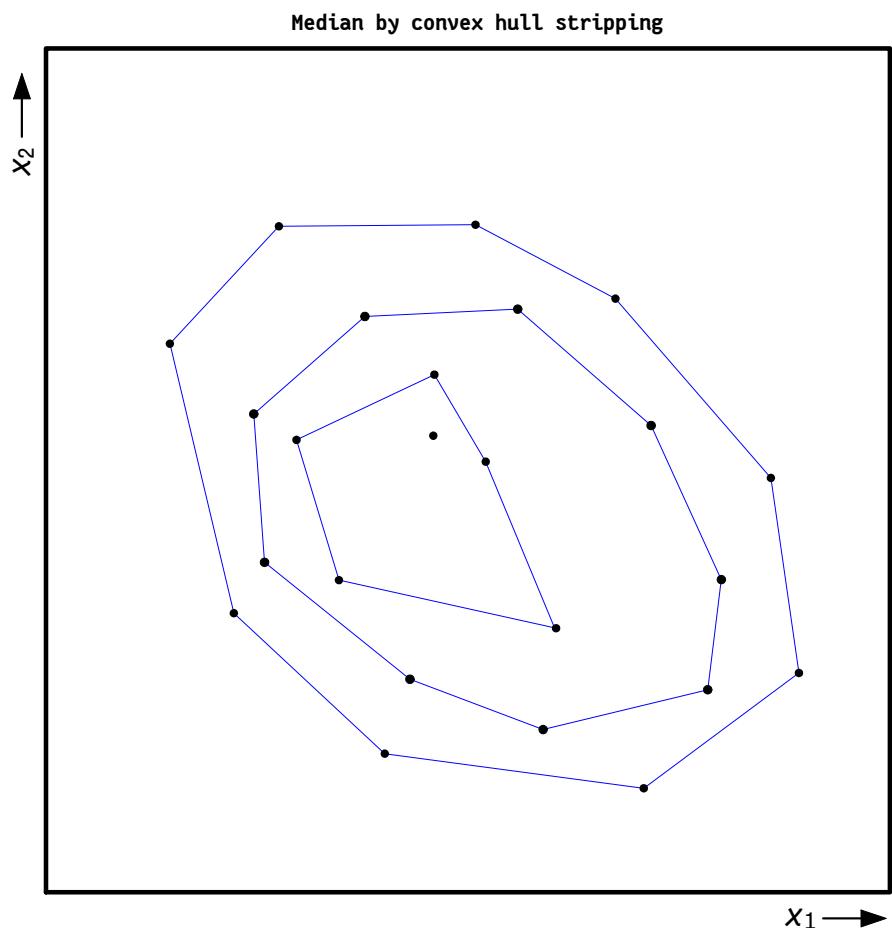


Figure 10.5.: Two-dimensional example of convex hull stripping. The outermost data are iteratively removed, until only the data of the innermost hull remain, here a single data point. The centroid of this innermost hull is the multivariate median. For details see text.

$\ell_1$  **Median** is the point of the data cloud that minimises the sum of the distance to all data points  $\sum_{i=1}^n \|\mathbf{x}_i - \hat{\mu}\|$  (position of warehouse that optimally supplies all customers). It has a breakdown-point of 50 % and reduces to the standard median in the 1D-case. Unfortunately, there is no efficient algorithm for finding it.

**convex hull stripping** For calculation of the univariate median, we recursively remove the largest and smallest values, until we are left with either one value (which is the median) or two (then the median is in between). This idea can be extended to multiple dimensions, by finding the hull of the data cloud. The hull is a convex polygon that encloses all data points with maximal area and minimal circumference. These points are removed, the new hull is calculated and removed and so on, until all remaining points are on the innermost hull. If this innermost hull consists of a single point, this point is the median, otherwise, the centroid is used (see fig. 10.5). However, the breakdown point is  $1/n$  under unfavourable circumstances (if all data are on the hull). The median is also not a continuous function of the data in the matrix  $\mathcal{X}$ .

**Simplicial depth median** If, in the 1D case, we construct all possible pairs of data points, then the median is the point which is enclosed by the largest number of these intervals, it is *deep inside* the data cloud. This can be extended to  $p$  dimensions by replacing the intervals with simplexes  $S(\mathbf{x}_1, \dots, \mathbf{x}_{p+1})$ . Then

$$SDF(\mu) = \binom{n}{p+1}^{-1} \sum_{1 \leq i_1 < \dots < i_{p+1} \leq n} 1[\mu \in S(\mathbf{x}_1, \dots, \mathbf{x}_{p+1})] \quad (10.39)$$

Then the simplicial depth median is the point  $\hat{\mu}$ , that maximises  $SDF(\mu)$ .

## 10.6. Test program

```

1 PROGRAM TestDescriptor;
2
3 USES math, MathFunc, Vector, Matrix, Zufall, Deskript;
4
5 CONST ProbSize = 100;
6     Vars      = 2;
7
8 VAR Data, Weights, Dm, Dr          : VectorTyp;
9     i, j                  : WORD;
10    MinEle, MaxEle, sum, mean,
11    std, q1, q2, q3, e2, e3, e4 : float;
12    MData, DataWithWeights     : MatrixTyp;
13
14 BEGIN
15     CreateVector(Data, ProbSize, 0.0);

```

## 10. Descriptive statistics

```

16 CreateVector(Weights, ProbSize, 1/ProbSize);
17 FOR i := 1 TO ProbSize DO
18     SetVectorElement(Data, i, 100 + RandomNormal(0, 10));
19
20 Mean := ArithmeticMean(Data);
21 Writeln('Arithmetic mean: ', FloatStr(Mean, 10));
22 Writeln('Geometric mean: ', FloatStr(GeometricMean(Data), 10));
23 Writeln('Harmonic mean: ', FloatStr(HarmonicMean(Data), 10));
24 Writeln;
25 Writeln('Power mean, k=1: ', FloatStr(GeneralMean(Data, Weights, 1), 10));
26 Writeln('Power mean, k=0.001: ', FloatStr(GeneralMean(Data, Weights,
27     0.001), 10));
28 Writeln('Power mean, k=-1: ', FloatStr(GeneralMean(Data, Weights, -1),
29     10));
30 Writeln;
31 std := StandardDeviation(Data);
32 Writeln('Standard deviation', FloatStr(std, 10));
33 Writeln('excess kurtosis', FloatStr(ExcessKurtosis(Data, mean, std), 10));
34 Writeln('Skewness', FloatStr(Skewness(Data, mean, std), 10));
35 Writeln;
36 Writeln('Gini coefficient: ', FloatStr(Gini(Data, Mean), 10));
37 Writeln('Herfindahl-Index: ', FloatStr(HerfindahlIndex(Data), 10));
38 Writeln;
39 q2 := Median(Data);
40 q1 := Quantile(Data, 0.25);
41 q3 := Quantile(Data, 0.75);
42 Writeln('Median: ', FloatStr(q2, 10));
43 Writeln('Trimedian: ', FloatStr(Trimedian(Data), 10)); // data have been
44     sorted by Median
45 Writeln('Inter-quartile distance: ', FloatStr(InterQuantilDistance(q1,
46     q3), 10));
47 Writeln('MAD: ', FloatStr(MAD(Data), 10));
48 Writeln('std. error of median: ', FloatStr(StandardErrorOfMedian(Data),
49     10));
50 Writeln;
51 Writeln('Naive Hodges-Lehmann: ', FloatStr(NaiveHodgesLehmann(Data), 10),
52 ' Hodges-Lehmann estimator: ', FloatStr(HodgesLehmann(Data), 10)); //
53     data have been sorted by Median
54 Writeln('Naive Sn: ', FloatStr(NaiveSn(Data), 10), ' Sn: ',
55     FloatStr(Sn(Data), 10));
56 Writeln('Naive Qn: ', FloatStr(NaiveQn(Data), 10));
57 // Writeln('Qn: ', FloatStr(Qn(Data), 10));
58
59 FOR i := 1 TO ProbSize DO SetVectorElement(Weights, i,
60     RandomNormal(1/ProbSize, 0.01));
61 MaxEle := FindLargest(Weights);

```

```

54 MinEle := FindSmallest(Weights);
55 Scale(Weights, MinEle, MaxEle); // weights IN 0..1
56 Sum := TotalSum(Weights);
57 FOR i := 1 TO ProbSize DO
58   SetVectorElement(Weights, i, GetVectorElement(Weights, i)/Sum); //
      total weight = 1
59 CreateMatrix(DataWithWeights, ProbSize, 2, 0.0);
60 SetColumn(DataWithWeights, Data, 1);
61 SetColumn(DataWithWeights, Weights, 2);
62 Writeln;
63 Write('Weighted median: ', FloatStr(WeightedMedian(DataWithWeights), 10));
64 Writeln(' weighted LoMed: ', FloatStr(WeightedLoMed(DataWithWeights),
      10),
      ', weighted HiMed: ', FloatStr(WeightedHiMed(DataWithWeights),
      10));
65 Writeln;
66 e2 := ell2(Data);
67 e3 := ell3(Data);
68 e4 := ell4(Data);
69 Writeln('ell2: ', FloatStr(e2, 10), ', ell3: ', FloatStr(e3, 10), ',
      ell4: ', FloatStr(e4, 10));
70 Writeln('tau2: ', FloatStr(e2/mean, 10), ', tau3: ', FloatStr(e3/e2, 10),
      ', tau4: ', FloatStr(e4/e2, 10));
71 Writeln('Quartile coefficient of skewness: ',
      FloatStr(QuartileCoefficientofSkewness(q1, q2, q3), 10));
72 Writeln('Centile coefficient of kurtosis: ',
      FloatStr(CentilCoeffKurtosis(Data), 10));
73 ReadLn;
74 DestroyVector(Data);
75 DestroyMatrix(DataWithWeights);
76
77 CreateMatrix(MData, ProbSize, Vars, 0.0);
78 FOR i := 1 TO (ProbSize DIV 10) DO          // 10% outliers
79   BEGIN
80     SetMatrixElement(MData, i, 1, 115 + RandomNormal(0, 3));
81     SetMatrixElement(MData, i, 2, 110 + RandomNormal(0, 3));
82   END;
83 FOR i := Succ(ProbSize DIV 10) TO ProbSize DO // 90% inliers
84   BEGIN
85     SetMatrixElement(MData, i, 1, 110 + RandomNormal(0, 3));
86     SetMatrixElement(MData, i, 2, 115 + RandomNormal(0, 3));
87   END;
88 RobustDistance (MData, Dr);
89 MahalanobisDistance (MData, Dm);
90 FOR i := 1 TO ProbSize DO
91   Writeln(i:3, ' ', FloatStr(GetMatrixElement(MData, i, 1), 11), ' ');

```

```

93     FloatStr(GetMatrixElement(MData, i, 2), 11), ' ',
94     FloatStr(GetVectorElement(Dm, i), 10), ' ',
95     FloatStr(GetVectorElement(Dr, i), 10));
96
97     ReadLn;
98     DestroyVector(Dr);
99     DestroyVector(Dm);
100    DestroyMatrix(MData);
101 END.
```

## References

- [1] E. KREYSZIG: *Statistische Methoden und ihre Anwendungen* 7th ed. Göttingen: Vandenhoeck & Ruprecht, 1979 ISBN: 9783525407172.
- [2] L.C. FREEMAN: *Elementary applied statistics* New York, London, Sidney: John Wiley & Sons, 1965.
- [3] C. GINI: On the measure of concentration with special reference to income and statistics, *Colorado College Publication, General Series* **208** (1936), 73–79.
- [4] J.L. HODGES, E.L. LEHMANN: Estimates of Location Based on Rank Tests, *Ann. Math. Stat.* **34**:2 (1963), 598–611 URL: <http://www.jstor.org/stable/2238406>.
- [5] P.J. ROUSSEEUW, C. CROUX: Alternatives to the Median Absolute Deviation, *J. Am. Stat. Assoc.* **88**:424 (1993), 1273–1283 DOI: [10.1080/01621459.1993.10476408](https://doi.org/10.1080/01621459.1993.10476408) URL: <https://www.jstor.org/stable/2291267>.
- [6] C. CROUX, P.J. ROUSSEEUW: *Time-Efficient Algorithms for Two Highly Robust Estimators of Scale*, In: *Computational Statistics* Y. DODGE, J. WHITTAKER (editor) Berlin, Heidelberg: Springer, 1992 chap. 58, 411–428 DOI: [https://www.researchgate.net/publication/228595593\\_Time-Efficient\\_Algorithms\\_for\\_Two\\_Highly\\_Robust\\_Estimators\\_of\\_Scale](https://www.researchgate.net/publication/228595593_Time-Efficient_Algorithms_for_Two_Highly_Robust_Estimators_of_Scale).
- [7] A.L. BOWLEY: *Elements of Statistics* London: P.S. King & Son, 1901 ISBN: 9781397213594.
- [8] G.U. YULE: *Introduction to the Theory of Statistics* 9th ed. London: Griffin, 1929.
- [9] R. WARREN, R.E. SMITH, A.K. CYBENKO: *Use of Mahalanobis distance for detecting outliers and outlier clusters in markedly non-normal data: A vehicular traffic example* Web-site, accessed 2018-11-19 AFRL-RH-WP-TR-2011-0070 United States Air Force Research Laboratory, 2011 URL: <http://www.dtic.mil/dtic/tr/fulltext/u2/a545834.pdf>.
- [10] P.C. MAHALANOBIS: On the Generalized Distance in Statistics, *Proc. Natl. Inst. Sci. India* **12** (1936), 49–55 URL: [https://insa.nic.in/writereaddata/UpLoadedFiles/PINSA/Vol02\\_1936\\_1\\_Art05.pdf](https://insa.nic.in/writereaddata/UpLoadedFiles/PINSA/Vol02_1936_1_Art05.pdf).

- [11] M. HUBERT, M. DEBRUYNE: Minimum covariance determinant, *WIREs Comput. Stat.* **2**:1 (2010), 36–43 DOI: [10.1002/wics.61](https://doi.org/10.1002/wics.61) URL: <https://wis.kuleuven.be/stat/robust/papers/2010/wire-mcd.pdf>.
- [12] P.J. ROUSSEEUW, A.M. LEROY: *Robust Regression and Outlier Detection* vol. 589 Wiley series in probability and mathematical statistics John Wiley & Sons, 1987 ISBN: 0471852333 URL: [https://books.google.de/books?id=woAH\\_73s-MwC&printsec=frontcover&hl=de&source=gbs\\_ge\\_summary\\_r&cad=0#v=onepage&q&f=false](https://books.google.de/books?id=woAH_73s-MwC&printsec=frontcover&hl=de&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false).
- [13] C.G. SMALL: A Survey of Multidimensional Medians, *Int. Stat. Rev.* **58**:3 (1990), 263–277 DOI: [10.2307/1403809](https://doi.org/10.2307/1403809).



# 11. Correlation coefficients

## Abstract

If  $k$  variables are observed in an experiment, then we get a  $k \times k$  matrix  $\mathcal{R}$  of correlation coefficients  $r$  between these variables. The diagonal elements are always 1.0, as each variable correlates perfectly with itself. Also, for all  $i, j$   $r_{i,j} = r_{j,i}$ , i.e.,  $\mathcal{R}$  is symmetrical. There are different ways to calculate the correlation coefficients, depending on the scale level of the data.

The unit calculates various types of correlation coefficients. Missing data should be encoded `NaN`. Nominal, binary and ordinal values are given as integer numbers, but in vectors and fields of type double (alternatively, the type definitions in units Vector and Matrix would have to be overloaded).

The record `Significance` contains the appropriate test variable, degrees of freedom and probability for  $H_0 : r = 0$  vs  $H_1 : r \neq 0$

Listing 11.1: Interface for unit Correlations

```
1 INTERFACE
2
3 USES Math, MathFunc, Vector, Stat, Deskript;
4
5 CONST
6   MaxCases = 1360;
7   MaxSteps = 100; // maximal # OF different values FOR ordinal AND nominal
                      data
8   CorrelationsError: BOOLEAN = FALSE;
9
10 TYPE
11   ContStruc = RECORD
```

Table 11.1.: Interpretation of correlation coefficients.

$ r $	meaning
.00 to .19	little to no relationship
.20 to .39	weak relationship
.40 to .59	moderate relationship
.60 to .79	strong relationship
.80 to 1.0	very strong relationship

## 11. Correlation coefficients

```
12  Table: ARRAY [0..MaxSteps, 0..MaxSteps] OF WORD;
13  RowSums, ColumnSums: ARRAY [0..MaxSteps] OF WORD;
14  r, c, n: WORD;
15  // number OF rows, columns AND legal (no NaN) comparisons
16 END;
17 ContTable = ^ContStruc;
18
19 Confusion = RECORD
20   a, b, c, d: WORD;
21 END;
22
23 SignificanceType = RECORD
24   TestValue, Freedom, P0: float;
25 END;
26
27 PROCEDURE CorrelationSignificance(CONST Vector1, Vector2: VectorTyp;
28   r: float; VAR Significance: SignificanceType);
29
30 FUNCTION PearsonProductMomentCorrelation(CONST Vector1, Vector2: VectorTyp;
31   VAR Significance: SignificanceType): float;
32
33 FUNCTION WeightedPearson(CONST Vector1, Vector2, weight: VectorTyp;
34   VAR Significance: SignificanceType): float;
35
36 PROCEDURE Rank(VAR Data: VectorTyp);
37
38 FUNCTION SpearmanRankCorrelation(CONST Vector1, Vector2: VectorTyp;
39   VAR Significance: SignificanceType): float;
40
41 FUNCTION QuadrantCorrelation(CONST Vector1, Vector2: VectorTyp;
42   VAR Significance: SignificanceType): float;
43
44 FUNCTION PointBiserialCorrelation(CONST BinaryVector, CardinalVector:
45   VectorTyp;
46   VAR Significance: SignificanceType): float;
47
48 FUNCTION ConfusionTable(CONST Vector1, Vector2: VectorTyp): Confusion;
49
50 FUNCTION McConaugheyCorrelation(c: confusion): float;
51
52 FUNCTION MatthewsCorrelation(c: Confusion): float;
53
54 FUNCTION Rk(Cont: ContTable): float;
55
56 PROCEDURE Contingency(Data1, Data2: VectorTyp; VAR Cont: ContTable);
```

```

57 PROCEDURE WriteContingency(CONST Cont: ContTable; MedStr: STRING;
58   ValidFigures: BYTE);
59
60 PROCEDURE DestroyContingency(VAR Cont: ContTable);
61
62 FUNCTION Chi2(CONST Contingency: ContTable;
63   VAR Significance: SignificanceType): float;
64
65 FUNCTION Phi2(CONST Contingency: ContTable;
66   VAR Significance: SignificanceType): float;
67
68 FUNCTION CramersVTilde(chi2: float; r, c, n: WORD): float;
69
70 FUNCTION Lambda(CONST Contingency: ContTable): float;
71
72 FUNCTION lambda_r(CONST Contingency: ContTable): float;
73
74 FUNCTION lambda_c(CONST Contingency: ContTable): float;
75
76 FUNCTION OrdinalCorrelations(CONST Data1, Data2: VectorTyp; Formula: CHAR;
77   VAR Significance: SignificanceType): float;
78
79 FUNCTION theta(CONST Contingency: ContTable): float;
80
81 FUNCTION eta_sqr(CONST NominalVector, CardinalVector: VectorTyp;
82   VAR Significance: SignificanceType): float;
83 { **** calculations with r ****
84
85 FUNCTION AverageCorrelations(CONST Correlations: VectorTyp): float;
86
87 FUNCTION Angle(r: float): float;
88
89
90 IMPLEMENTATION

```

One thing to remember about all correlation coefficients: **Correlation does not prove causality!** This is known as the *cum hoc, ergo propter hoc* fallacy. If a correlation is found between two variables  $x, y$ , then there are several possible explanations:

**direct causation**  $x$  causes  $y$

**reverse causation**  $y$  causes  $x$

**cyclic causation**  $x$  causes  $y$  and  $y$  causes  $x$

**indirect causation**  $x$  causes  $a$ , which causes  $y$

## 11. Correlation coefficients

**common cause** Both  $x, y$  are caused by another (**confounding**) factor. Many such pseudo-correlations have been published, including between polio incidence and ice cream sales (polio virus is spread by the fecal-oral route in swimming pools, hence mostly in warm weather when ice cream sales also increase) or human birth rate and number of storks ([1], both decline with industrialisation).

**coincidence** the correlation is spurious and will vanish if new data become available.

Correlation coefficients may be symmetrical ( $-1 \leq r \leq +1$ ) or unsymmetrical ( $0 \leq r^* \leq +1$ ). A symmetrical coefficient can be converted to unsymmetrical by  $r^* = 0.5(r+1)$  and *vice versa* by  $r = 2r^* - 1$ .

### 11.1. General routines

Listing 11.2: Test for equal vector length

```
1  FUNCTION TestDataVectorLength(Data1, Data2: VectorTyp): BOOLEAN;
2
3  BEGIN
4      IF (VectorLength(Data1) <> VectorLength(Data2))
5          THEN
6              BEGIN
7                  c := WriteErrorMessage(' Correlation: vectors of unequal length');
8                  Result := FALSE;
9              END
10         ELSE
11             Result := TRUE;
12     END;
```

Listing 11.3: Average of several correlation coefficients

```
1  FUNCTION AverageCorrelations(CONST Correlations: VectorTyp): float;
2
3  VAR
4      i, j, k: WORD;
5      SumR: float;
6
7  BEGIN
8      k := VectorLength(Correlations);
9      SumR := 0;
10     j := 0;
11     FOR i := 1 TO k DO
12         IF IsNaN(GetVectorElement(Correlations, i))
13             THEN // ignore
14         ELSE
15             BEGIN
```

```

16      SumR := SumR + tanh(GetVectorElement(Correlations, i));
17      Inc(j);
18  END;
19  Result := arctanh(SumR / j);
20 END;

```

## 11.2. Cardinal (interval and rational) data

### 11.2.1. PEARSON's product moment correlation coefficient $r_p$

In common usage, when we speak of a correlation coefficient without further specification, then PEARSON's product moment coefficient  $r_p$  [2] is meant. However, it is defined only for data on a interval or rational scale, it has also been used for binary data. For two data vectors  $\mathbf{x}$  and  $\mathbf{y}$ , both of length  $n$ ,  $r_p$  is

$$r_p = \frac{\sum_{i=1}^n (\mathbf{x}_i - \bar{\mathbf{x}}) \sum_{i=1}^n (\mathbf{y}_i - \bar{\mathbf{y}})}{\sqrt{\sum_{i=1}^n (\mathbf{x}_i - \bar{\mathbf{x}})^2 \sum_{i=1}^n (\mathbf{y}_i - \bar{\mathbf{y}})^2}} = \frac{n \sum_{i=1}^n (\mathbf{x}_i \mathbf{y}_i) - \sum_{i=1}^n \mathbf{x}_i \sum_{i=1}^n \mathbf{y}_i}{\sqrt{[n \sum_{i=1}^n \mathbf{x}_i^2 - (\sum_{i=1}^n \mathbf{x}_i)^2][n \sum_{i=1}^n \mathbf{y}_i^2 - (\sum_{i=1}^n \mathbf{y}_i)^2]}} \quad (11.1)$$

Computationally, the second way to calculate  $r_p$  is quicker, as it doesn't require the calculation of averages, that is, the loop over all  $\mathbf{x}$ ,  $\mathbf{y}$  has to be done only once, not twice.

Listing 11.4: PEARSON product moment correlation

```

1 FUNCTION PearsonProductMomentCorrelation(CONST Vector1, Vector2: VectorTyp;
2   VAR Significance: SignificanceType): float;
3
4 VAR
5   i, j: WORD;
6   SumXY, SumX, SumY, SumX2, SumY2, varX, varY, covXY, x, y, r: float;
7
8 BEGIN
9   IF NOT (TestDataVectorLength(Vector1, Vector2)) THEN EXIT;
10  SumXY := 0;
11  SumX := 0;
12  SumY := 0;
13  SumX2 := 0;
14  SumY2 := 0;
15  j := 0;
16  FOR i := 1 TO VectorLength(Vector1) DO
17    BEGIN
18      x := GetVectorElement(Vector1, i);
19      y := GetVectorElement(Vector2, i);
20      IF (IsNaN(x) OR IsNaN(y))
21        THEN // ignore

```

## 11. Correlation coefficients

```

22      ELSE
23      BEGIN
24          SumXY := SumXY + x * y;
25          SumX := SumX + x;
26          SumY := SumY + y;
27          SumX2 := SumX2 + x * x;
28          SumY2 := SumY2 + y * y;
29          INC(j); // actual number OF valid comparisons
30      END; { else }
31      END; { for }
32      varX := j * SumX2 - SumX * SumX;
33      varY := j * SumY2 - SumY * SumY;
34      covXY := j * SumXY - SumX * SumY;
35      IF (varX * varY) = 0
36          THEN r := 0 // ???
37      ELSE r := covXY / Sqrt(varX * varY);
38      CorrelationSignificance(Vector1, Vector2, r, Significance);
39      Result := r;
40  END; { PearsonProductMoment }
```

For factor analysis the data should be  $z$ -standardised ( $\bar{x} = 0, \sigma = 1$ ) before calculating the correlation coefficient, especially if they have widely varying scales. Otherwise, factor analysis may pick the variable that has the highest variance as the direction of the greatest variability. Then

$$z_i = \frac{x_i - \bar{x}}{\sigma} \quad (11.2)$$

It is possible to weigh data points by importance (e.g., relative standard deviations if they are means) with a weight vector  $w$ :

$$\bar{x} = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i} \quad \bar{y} = \frac{\sum_{i=1}^n w_i y_i}{\sum_{i=1}^n w_i} \quad (11.3)$$

$$s_{x,y} = \frac{\sum_{i=1}^n w_i (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n w_i} \quad s_x^2 = \frac{\sum_{i=1}^n w_i (x_i - \bar{x})^2}{w_i} \quad s_y^2 = \frac{\sum_{i=1}^n w_i (y_i - \bar{y})^2}{w_i} \quad (11.4)$$

$$r(x, y, w) = \frac{s_{x,y}}{\sqrt{s_x^2 s_y^2}} \quad (11.5)$$

Listing 11.5: Weighted PEARSON product moment correlation

```

1  FUNCTION WeightedPearson(CONST Vector1, Vector2, weight: VectorTyp;
2      VAR Significance: SignificanceType): float;
3
4  VAR
5      i, n: WORD;
6      BarX, BarY, SumWX, SumWY, SumW, r, SumWXd, SumWYd, SumWXYd, CovXY,
```

```

7  VarX, VarY, x, y, w: float;
8
9  BEGIN
10 IF NOT (TestDataVectorLength(Vector1, Vector2)) THEN EXIT;
11 n := VectorLength(Vector1);
12 SumWX := 0;
13 SumWY := 0;
14 SumW := 0;
15 SumWXd := 0;
16 SumWYd := 0;
17 SumWXYd := 0;
18 FOR i := 1 TO n DO
19   BEGIN
20     x := GetVectorElement(Vector1, i);
21     y := GetVectorElement(Vector2, i);
22     w := GetVectorElement(Weight, i);
23     IF (IsNaN(x) OR IsNaN(y) OR IsNaN(w))
24       THEN // ignore
25     ELSE
26       BEGIN
27         SumWX := SumWX + w * x;
28         SumWY := SumWY + w * y;
29         SumW := SumW + w;
30       END; { else }
31   END; { for }
32 BarX := SumWX / SumW;      // calculate averages
33 BarY := SumWY / SumW;
34 FOR i := 1 TO n DO
35   BEGIN
36     x := GetVectorElement(Vector1, i);
37     y := GetVectorElement(Vector2, i);
38     w := GetVectorElement(Weight, i);
39     IF (IsNaN(x) OR IsNaN(y) OR IsNaN(w))
40       THEN // ignore
41     ELSE
42       BEGIN
43         SumWXd := SumWXd + w * (x - BarX) * (x - BarX);
44         SumWYd := SumWYd + w * (y - BarY) * (y - BarY);
45         SumWXYd := SumWXYd + w * (x - BarX) * (y - BarY);
46       END; { else }
47   END; { for }
48 CovXY := SumWXYd / SumW;
49 VarX := SumWXd / SumW;
50 VarY := SumWYd / SumW;
51 IF (varX * varY) = 0
52   THEN r := 0 // ???

```

## 11. Correlation coefficients

```

53     ELSE r := CovXY / Sqrt(VarX * VarY);
54     CorrelationSignificance(Vector1, Vector2, r, Significance);
55     Result := r;
56 END; { WeightedPearson }

```

The standard error of  $r_p$  is

$$\sigma = \frac{0.6325}{\sqrt{n-1}} \quad (11.6)$$

To test for the 0-hypothesis  $r_p = 0$  even for small data sets use

$$t = r_p \sqrt{\frac{n-2}{1-r_p^2}} \quad v = n-2 \quad (11.7)$$

for  $r < 1$ .

Listing 11.6: Probability for r

```

1 PROCEDURE CorrelationSignificance(CONST Vector1, Vector2: VectorTyp;
2   r: float; VAR Significance: SignificanceType);
3
4 VAR
5   i, n, m: WORD;
6
7 BEGIN
8   n := VectorLength(Vector1);
9   IF NOT (TestDataVectorLength(Vector1, Vector2))
10    THEN EXIT;
11   m := 0;
12   FOR i := 1 TO n DO
13     IF (IsNaN(GetVectorElement(Vector1, i)) OR
14       IsNaN(GetVectorElement(Vector2, i)))
15     THEN // ignore
16     ELSE INC(m);
17   Significance.TestValue := (m - 2) / (1 - r * r);
18   IF signum(Significance.TestValue) >= 0
19   THEN Significance.TestValue := r * Sqrt(Significance.TestValue)
20   ELSE Significance.TestValue := 0;
21   Significance.Freedom := m - 2;
22   Significance.P0 := Integral_t(Significance.TestValue,
23     Round(Significance.Freedom));
24 END;

```

To compare the difference of two  $r$ -values for significance (if for both data sets  $n \geq 10$ ) FISHER's  $z$ -transformation is used for both coefficients:

$$z_i = 0.5 \ln \left( \frac{1+r_i}{1-r_i} \right) \quad (11.8)$$

## 11.2. Cardinal (interval and rational) data

The two-sided significance of the difference is

$$P_0 = \operatorname{erfc} \left( \frac{|z_1 - z_2|}{\sqrt{2} \sqrt{\frac{1}{n_1-3} + \frac{1}{n_2-3}}} \right) \quad (11.9)$$

To compare an observed  $r_o$  with a hypothetical value  $r_h$ , use

$$z_h = 0.5 * \left[ \ln \left( \frac{1+r_h}{1-r_h} \right) + \frac{r_h}{n-1} \right] \quad (11.10)$$

$$P(0) = \operatorname{erfc} \left( \frac{|z - z_h| \sqrt{n-3}}{\sqrt{2}} \right) \quad (11.11)$$

$n$  correlation coefficients can be averaged only after transformation:

$$\bar{r} = \tanh^{-1} \left( \frac{\sum_{i=1}^n \tanh(r_i)}{n} \right) \quad (11.12)$$

### Geometric and PRE interpretation of PEARSON's correlation coefficient

For  $n$  samples of a random variable  $\mathcal{Y} = (\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n)$  the expected value of  $\mathcal{Y}$  is [3]

$$\mu = E(\mathcal{Y}) = \sum_{j=1}^p \mathbf{y}_j p_j \quad (11.13)$$

with  $p_i$  the probability of the  $i$ -th element. Then the arithmetic mean  $\bar{\mathbf{y}}$  is an estimator of this expected value:

$$\bar{\mathbf{y}} = \frac{1}{n} \sum_{i=1}^n \mathbf{y}_i \quad (11.14)$$

and the variance is a measure of scatter around the expected value:

$$\sigma(\mathcal{Y}) = E[(\mathcal{Y} - \mu)^2] = \sum_{i=1}^n (\mathbf{y}_i - \mu)^2 p_i \quad (11.15)$$

and its estimator is

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (\mathbf{y}_i - \bar{\mathbf{y}})^2 \quad (11.16)$$

and its square root is the standard deviation  $s$  (which has the same unit as  $\mathcal{Y}$ ). If one had to guess a  $\mathbf{y}_i$ , then the best estimator would be  $\bar{\mathbf{y}}$  and  $s$  would be the probable error.

Then a measure of the relationship of two variables  $\mathcal{X}, \mathcal{Y}$  is their covariance

$$\sigma_{\mathcal{X}, \mathcal{Y}} = E[(\mathcal{X} - \mu_{\mathcal{X}})(\mathcal{Y} - \mu_{\mathcal{Y}})] \quad (11.17)$$

## 11. Correlation coefficients

which when normalised to unity is PEARSON's correlation coefficient  $r_p$ :

$$r_p = \frac{\sigma_{\mathcal{X}, \mathcal{Y}}}{\sigma_{\mathcal{X}} \times \sigma_{\mathcal{Y}}} = \frac{E[(\mathcal{X} - \mu_{\mathcal{X}})(\mathcal{Y} - \mu_{\mathcal{Y}})]}{E[(\mathcal{X} - \mu_{\mathcal{X}})^2] \times E[(\mathcal{Y} - \mu_{\mathcal{Y}})^2]} = \frac{\sum_{i=1}^n [(\mathfrak{x}_i - \bar{\mathfrak{x}})(\mathfrak{y}_i - \bar{\mathfrak{y}})]}{\sqrt{\sum_{i=1}^n (\mathfrak{x}_i - \bar{\mathfrak{x}})^2} \sqrt{\sum_{i=1}^n (\mathfrak{y}_i - \bar{\mathfrak{y}})^2}} \quad (11.18)$$

The dispersion of a random variable is  $\tilde{\mathfrak{x}}_i = \mathfrak{x}_i - \bar{\mathfrak{x}}$ . Using this we can write

$$r_p = \frac{\sum_{i=1}^n \tilde{\mathfrak{x}}_i \tilde{\mathfrak{y}}_i}{\sqrt{\sum_{i=1}^n \tilde{\mathfrak{x}}_i^2} \times \sqrt{\sum_{i=1}^n \tilde{\mathfrak{y}}_i^2}} \quad (11.19)$$

A variable  $\mathcal{X}$  can be interpreted as a vector in  $p$ -dimensional space. The EUKLIDIAN norm (length) of this vector is

$$|\mathcal{X}| = \sqrt{\sum_{i=1}^n \mathfrak{x}_i^2} \quad (11.20)$$

The dot product of two such vectors of equal length is

$$\mathcal{X} \bullet \mathcal{Y} = \sum_{i=1}^n \mathfrak{x}_i \mathfrak{y}_i = |\mathcal{X}| \times |\mathcal{Y}| \times \cos(\mathcal{X} \mathcal{Y}) \quad (11.21)$$

with  $\cos(\mathcal{X} \mathcal{Y})$  the cosine of the angle between the vectors, which in other words is equal to

$$\cos(\mathcal{X} \mathcal{Y}) = \frac{\mathcal{X} \bullet \mathcal{Y}}{|\mathcal{X}| \times |\mathcal{Y}|} \quad (11.22)$$

Using eqn. 11.19, we see that

$$r_p = \frac{\sum_{i=1}^n \tilde{\mathfrak{x}}_i \tilde{\mathfrak{y}}_i}{\sqrt{\sum_{i=1}^n \tilde{\mathfrak{x}}_i^2} \times \sqrt{\sum_{i=1}^n \tilde{\mathfrak{y}}_i^2}} = \frac{\mathcal{X} \bullet \mathcal{Y}}{|\mathcal{X}| \times |\mathcal{Y}|} = \cos(\mathcal{X} \mathcal{Y}) \quad (11.23)$$

the correlation coefficient  $r_p$  is equal to the cosine of the angle between the data vectors  $\mathcal{X}, \mathcal{Y}$ . If  $r_p$  is zero, the angle between the vectors is 90 and the data are independent. If  $r_p$  is unity, the angle is 0 and the vectors are parallel and the data are perfectly correlated.

Listing 11.7: Convert correlation to angle

```

1  FUNCTION Angle (r : float) : float;
2
3  BEGIN
4      Result := ArcCos(r) * 180 / Const_pi;
5  END;
```

Now consider the situation that we had two random variables  $\mathcal{X}, \mathcal{Y}$  which are related by a linear regression line

$$\hat{\mathcal{Y}} = a + b \times \mathcal{X} \quad (11.24)$$

Then the correlation coefficient is

$$r_p = \frac{\sum_{i=1}^n [(\mathfrak{y}_i - \bar{\mathfrak{y}}) \times (\hat{\mathfrak{y}}_i - \bar{\mathfrak{y}})]}{\sqrt{\sum_{i=1}^n (\mathfrak{y}_i - \bar{\mathfrak{y}})^2} \times \sqrt{\sum_{i=1}^n (\hat{\mathfrak{y}}_i - \bar{\mathfrak{y}})^2}} \quad (11.25)$$

which again is equal to the cosine of two vectors, representing the dispersion of  $\mathcal{Y}$  and  $\hat{\mathcal{Y}}$ , respectively. Eqn. 11.25 can also be written as

$$r_p = \frac{\sqrt{\sum_{i=1}^n (\hat{y}_i - \bar{y})^2}}{\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}} \quad (11.26)$$

which is the ratio of the length of the two vectors.

If  $SS_{\text{total}} = \sum (y_i - \bar{y})^2$ ,  $SS_{\text{explained}} = \sum (\hat{y}_i - \bar{y})^2$  and  $SS_{\text{residual}} = \sum (y_i - \hat{y}_i)^2$ , then  $r^2 = 1 - \frac{SS_{\text{residual}}}{SS_{\text{total}}}$  the **coefficient of determination**, that is, the part of the variance in  $y$  that is explained by the regression equation. Thus,  $r_p^2$  is a **proportional reduction in error (PRE)** measure, which is determined by how much error is reduced if for any  $x_i$  one uses  $\hat{y}_i$  instead of  $\bar{y}$  to predict  $y_i$  [4].

Although some of the other correlation coefficients discussed below have a PRE interpretation, the angle interpretation is specific to  $r_p$ .

### Partial correlation

If the correlation between two variables  $X, Y$  is caused by the effect of a third  $Z$  on both, then this can be detected with the partial correlation coefficient

$$r_{XY|Z} = \frac{r_{XY} - r_{XZ} \times r_{YZ}}{\sqrt{(1 - r_{XZ}^2)(1 - r_{YZ}^2)}} \quad (11.27)$$

The partial correlation can also be used to determine if a particular variable out of a set of independent variables contributes to the explanatory power of that set towards a dependent variable. It simply compares the sum of squared errors  $SS_{\text{res}} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$  for the full and the reduced model (that is, model without the variable under test):

$$r_{\text{par}}^2 = \frac{SS_{\text{res,red}} - SS_{\text{res,full}}}{SS_{\text{res,red}}} = \frac{r_{\text{full}}^2 - r_{\text{red}}^2}{1 - r_{\text{red}}^2} \quad (11.28)$$

### Limitations of PEARSON's $r_p$

PEARSON's  $r_p$  is not a robust estimate of the association between two variables. It is affected by outliers, heteroscedasticity, curvature, the magnitude of residuals and range restriction [5, fig. 2]. Many such pitfalls can be detected by plotting the data!

Relatively large  $n (> 100)$  is required for the sample  $r$  to be a reliable estimate for the population  $r$ , especially when the “true” correlation is small. If  $n < 30$  and the measured  $r > 0.6$ , then a correction should be applied to the correlation coefficient [6], because the sample  $r$  tends to underestimate the population association:

$$r^* = r \left( 1 + \frac{1 - r^2}{2(n - 3)} \right) \quad (11.29)$$

, but that is usually ignored.

## 11. Correlation coefficients

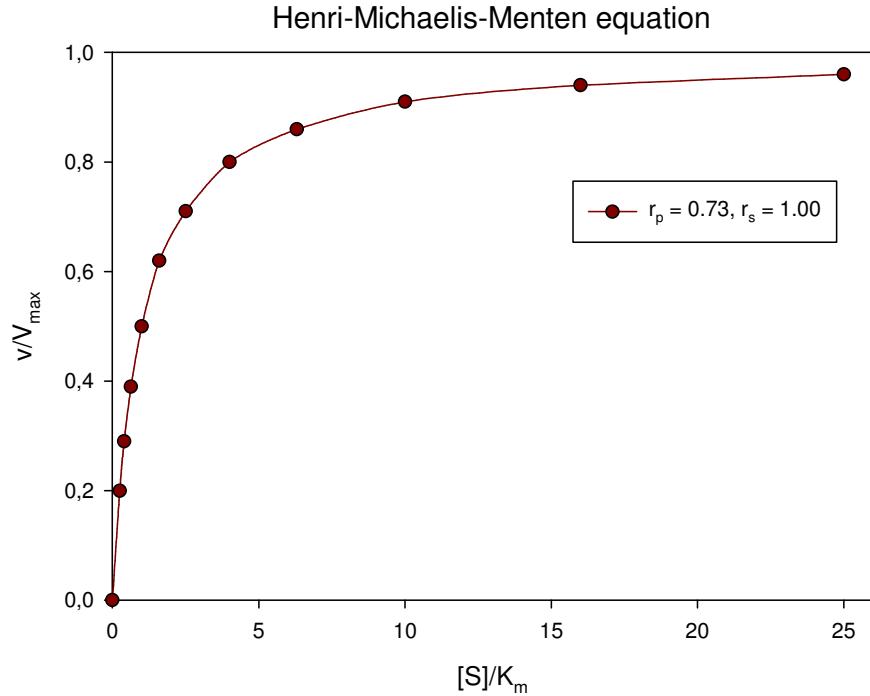


Figure 11.1.: The synthetic data in this plot follow the HENRI-MICHAELIS-MENTEN-law, and the rank correlation coefficient  $r_s$  is 1.0. However, because of the curvature of the data, the product moment correlation  $r_p$  is only 0.73.

### 11.2.2. SPEARMAN's rank correlation coefficient $r_s$

$r_s$  ([7], often called  $\rho$ ) was developed for ordinal scaled variables, however, it can also be used for interval and rational scales. In that case, we work with the rank of each  $x_i, y_i$  rather than their values. This has the disadvantage of some loss of information, but the advantage that  $r_s$  works for non-linear relationships between  $x$  and  $y$ , provided only that  $y$  is a monotonically increasing function of  $x$  (see fig. 11.1). For the rank correlation of a linear and a circular variable see chapter 14.

Listing 11.8: Ranking of a variable

```

1 PROCEDURE Rank(VAR Data: VectorTyp);
2
3 LABEL
4   done;
5
6 VAR
7   i, j, k, Valid, n: WORD;
8   r, x, y: float;
9   Sorted: VectorTyp;
10
11 BEGIN
12   n := VectorLength(Data);
```

```

13 CopyVector(Data, Sorted);
14 ShellSort(Sorted);           // create a sorted Copy OF data  (any NaNs
15   TO highest #)
16 Valid := n;
17 FOR i := 1 TO n DO
18   IF IsNaN(GetVectorElement(Sorted, i))
19     THEN DEC(Valid);          // determine number OF non-NaN data points
20 i := 1;
21 REPEAT
22   x := GetVectorElement(Sorted, i);
23   IF IsNaN(x)
24     THEN
25     ELSE
26       BEGIN
27         j := i;
28       REPEAT           // find number OF Elements that are equal TO x
29         y := GetVectorElement(Sorted, Succ(j));
30         IF IsNaN(y)
31           THEN GOTO Done // leave counting LOOP, no more valid data
32           ELSE IF (x = y)
33             THEN INC(j);
34           UNTIL ((x < y) OR (j = n));
35 Done: r := (1.0 * i + j) / 2;    // rank IS average OF lowest +
36   highest element
37   FOR k := 1 TO Valid DO        // change all elements OF Data equal
38     TO x -> rank
39     BEGIN
40       y := GetVectorElement(Data, k);
41       IF IsNaN(y)
42         THEN
43         ELSE IF (y = x)
44           THEN SetVectorElement(Data, k, r);
45         END;
46       i := Succ(j);
47     END;
48   UNTIL (i >= Valid) OR IsNaN(x);
49 DestroyVector(Sorted);
50 END;

```

The distribution of  $r_s$  does not depend on the distribution of  $\mathbf{x}, \mathbf{y}$ , so the tests described for  $r_p$  can always be used ( $r_s$  is non-parametric). The calculation of  $r_s$  is performed exactly as described for  $r_p$ .

If there are no ties (all x-values different from each other and all y-values different from each other), then there is a computationally simpler form to calculate  $r_s$ :

$$r_s = 1 - \frac{6 \sum_{i=1}^n (\mathbf{x}_i - \mathbf{y}_i)^2}{n^3 - n} \quad (11.30)$$

## 11. Correlation coefficients

$r_s$  has no PRE interpretation [8].

Listing 11.9: SPEARMAN's rank correlation coefficient

```

1  FUNCTION SpearmanRankCorrelation(CONST Vector1, Vector2: VectorTyp;
2    VAR Significance: SignificanceType): float;
3
4  VAR
5    Ranked1, Ranked2: VectorTyp;
6
7  BEGIN
8    IF NOT (TestDataVectorLength(Vector1, Vector2)) THEN EXIT;
9    CopyVector(Vector1, Ranked1);
10   CopyVector(Vector2, Ranked2);
11   Rank(Ranked1);
12   Rank(Ranked2);
13   Result := PearsonProductMomentCorrelation(Ranked1,
14     Ranked2, Significance);
15   DestroyVector(Ranked1);
16   DestroyVector(Ranked2);
17 END;
```

### Significance testing of $r_s$

A non-zero  $r_s$  can be tested for significance by

$$t = r_s \times \sqrt{\frac{n-2}{1-r_s^2}} \quad (11.31)$$

with  $n - 2$  degrees of freedom.

### 11.2.3. Quadrant correlation $r_q$

The quadrant correlation is measure of correlation robust against outliers. It is calculated from the number of observations in the four quadrants determined by the median pair. The number of observations in quadrant I and III ( $n_+$ ) and in quadrants II and IV ( $n_-$ ) are determined:

$$r_q = \frac{n_+ - n_-}{n_+ + n_-} = \frac{1}{N} \times \sum_{i=1}^n \text{sgn}(\bar{x}_i - \tilde{x}) \times \text{sgn}(\bar{y}_i - \tilde{y}) \quad (11.32)$$

with  $\bar{x}, \bar{y}$  the medians of  $\mathfrak{X}, \mathfrak{Y}$ .

### Median test for significance of $r_q$

for the test  $H_0 : r_q = 0$  vs  $H_1 : r_q \neq 0$  calculate (with  $n_e = (n_+ + n_-)/2$ ,  $n_e > 5$ ):

$$\chi^2 \approx \frac{(n_+ - n_e)^2 + (n_- - n_e)^2}{n_e} \quad (11.33)$$

with one degree of freedom.

Listing 11.10: Quadrant correlation and its significance

```

1  FUNCTION QuadrantCorrelation(CONST Vector1, Vector2: VectorTyp;
2    VAR Significance: SignificanceType): float;
3
4  VAR
5    Sorted1, Sorted2          : VectorTyp;
6    n, i, n_plus, n_minus     : WORD;
7    Median1, Median2, x, y, n_e : float;
8    c                         : char;
9
10 BEGIN
11  IF NOT (TestDataVectorLength(Vector1, Vector2)) THEN EXIT;
12  n := VectorLength(Vector1);
13  CopyVector(Vector1, Sorted1);
14  CopyVector(Vector2, Sorted2);
15  Median1 := Median(Sorted1);
16  Median2 := Median(Sorted2);
17  DestroyVector(Sorted1);
18  DestroyVector(Sorted2);
19  n_plus := 0;
20  n_minus := 0;
21  FOR i := 1 TO n DO
22    BEGIN
23      x := GetVectorElement(Vector1, i);
24      y := GetVectorElement(Vector2, i);
25      IF IsNaN(x) OR IsNaN(y)
26        THEN // ignore NaNs
27      ELSE IF ((x = Median1) OR (y = Median2))
28        THEN // ignore data ON coordinate axis OF system WITH origin
29          Median1/Median2
30      ELSE IF (((x > Median1) AND (y > Median2)) OR ((x < Median1)
31          AND (y < Median2)))
32        THEN INC(n_plus) // concordance: quadrant I OR III
33        ELSE INC(n_minus); // discordance: quadrant II OR IV
34    END;
35  Result := (n_plus - n_minus) / (n_plus + n_minus);
36  n_e := 1.0 * (n_plus + n_minus) / 2;
37  IF (n_e < 5)
38    THEN
39      BEGIN
40        c := WriteErrorMessage('Quadrant correlation: cannot calculate
41          significance because n_e < 5');
42        CorrelationsError := TRUE;
43        EXIT;
44      END;

```

## 11. Correlation coefficients

```

42  Significance.TestValue := ((n_plus - n_e) * (n_plus - n_e) +
43      (n_minus - n_e) * (n_minus - n_e)) / n_e;
44  Significance.Freedom := 1;
45  Significance.P0 := IntegralChi(Significance.TestValue,
46      Round(Significance.Freedom));
46 END;

```

### 11.3. Binary data

Association and similarity coefficients for binary data have been reviewed in [9]. Given

observed	predicted	
1	0	
1	$TP = a$	$FN = b$
0	$FP = c$	$TN = d$

the association measure  $r$  should meet the following obligatory conditions:

1. For an association measure  $r_{i,j} \leq r_{i,i}$  for all  $i, j$  and if  $r_{i,j} > r_{k,l}$  then items  $i, j$  are more similar than items  $k, l$ .
2.  $\min(r)$  should be at  $a = d = 0$  and  $\max(r)$  at  $b = c = 0$ .
3. Symmetry:  $r_{i,j} = r_{j,i}$  for all  $i, j$ .
4. Discrimination between positive and negative association:  $r(a > a') > r(a < a')$  with  $a' = \frac{(a+b)(a+c)}{n}$ .
5.  $r$  should be linear with  $\sqrt{\chi^2}$  for both subsets  $ad - bc < 0$  and  $ad - bc \geq 0$  (note that  $\chi^2$  violates condition 4).

and ideally the following non-obligatory conditions:

- A Range of  $r$  should be either  $-1 \dots +1$  or  $0 \dots +1$ .
- B Absolute association:  $r(b = c = 0) > r(b = 0 \vee c = 0)$  (with  $\vee$  meaning exclusive disjunction (xor)).
- C Nil association at  $a = 0$ :  $r(a = 0) = \min(r)$  (stricter than 2) above).
- D Linearity:  $r(a+1) - r(a) = r(a+2) - r(a+1)$ .
- E No sensitivity to  $a = 0$ :  $r(a = 0, b, c, d), r(a = 1, b - 1, c - 1, d + 1), r(a = 2, b - 2, c - 2, d + 2) \dots$  should be smooth
- F Homogeneous distribution of  $r$  in permutation samples
- G For random samples from a population with known  $a, b, c, d$ ,  $r$  should show little variability even in small samples.

H simplicity of calculation, low computer time

All conditions are met by JACCARD [10] ( $\frac{a}{a+b+c} = (A \cap B)/(A \cup B)$ ), RUSSEL & RAO [11] ( $\frac{a}{a+b+c+d}$ ) (both range 0 ... +1) and McCONAUGHEY [12] ( $\frac{a^2-bc}{(a+b)(a+c)}$ , range -1 ... +1). However, neither of these coefficients has a PRE interpretation.

Listing 11.11: confusion table

```

1  FUNCTION ConfusionTable(CONST Vector1, Vector2: VectorTyp): Confusion;
2
3  VAR
4      i, n: WORD;          // contingency table values
5
6  BEGIN
7      IF NOT (TestDataVectorLength(Vector1, Vector2)) THEN EXIT;
8      n := VectorLength(Vector1);
9      WITH ConfusionTable DO
10         BEGIN
11             a := 0;
12             b := 0;
13             c := 0;
14             d := 0;
15             FOR i := 1 TO n DO
16                 IF (IsNaN(GetVectorElement(Vector1, i)) OR
17                     IsNaN(GetVectorElement(Vector2, i)))
18                     THEN // ignore
19                     ELSE IF (GetVectorElement(Vector1, i) = 1)    // count contingency
20                         table elements
21                         THEN IF (GetVectorElement(Vector2, i) = 1)
22                             THEN INC(a) // joint presence
23                             ELSE INC(b) // discordance
24                         ELSE IF (GetVectorElement(Vector2, i) = 1)
25                             THEN INC(c) // discordance
26                             ELSE INC(d); // joint absence
27         END;
28     END;

```

Listing 11.12: McCONAUGHEY's correlation

```

1  FUNCTION McConaugheyCorrelation(c: confusion): float;
2
3  BEGIN
4      WITH c DO
5          BEGIN
6              Result := (1.0 * (a + b) * (a + c));
7              IF (Result = 0)
8                  THEN // shouldn't happen, but result will be 0

```

## 11. Correlation coefficients

```

9      ELSE Result := (1.0 * a * a - 1.0 * b * c) / Result;
10     END;
11 END;
```

### 11.3.1. MATTHEWS' correlation $r_m$

MATTHEWS' correlation [13] is used mainly to characterise confusion tables, as it gives balanced results even with very different class sizes:

$$r_m = \frac{ad - cb}{\sqrt{(a+c)(a+b)(d+c)(d+b)}} = \sqrt{\frac{\chi^2}{n}} \quad (11.34)$$

Should the denominator become zero,  $r_m$  is set to zero too, which is the correct limiting value. In [9]  $r_m = A_{30}$  (tetrachoric correlation), its root is called  $A_{31}$  and  $A_{27} = \chi^2 = nr_m^2$ ,  $0 \leq r_m \leq \infty$ . For clustering, these coefficients are not well suited because

$A_{27}$  violates conditions 2, 4, D, E

$A_{30}$  violates conditions C, F

$A_{31}$  violates conditions 4, D

Listing 11.13: MATTHEWS' correlation

```

1 FUNCTION MatthewsCorrelation(c: Confusion): float;
2
3 BEGIN
4   WITH c DO
5     BEGIN
6       Result := 1.0 * (a + c) * (a + b) * (d + c) * (d + b);
7       // calculate denominator first
8       IF Result <= 0
9         THEN // shouldn't happen, but result will be 0, the correct
10            limiting value
11       ELSE Result := (a * d - c * b) / Sqrt(Result);
12     END;
13 END;
```

Recall the equation for PEARSON's  $r_p = \frac{\text{Cov}(x,y)}{\sqrt{\text{Cov}(x,x)} \sqrt{\text{Cov}(y,y)}} = \frac{\sum_{i=1}^n (\bar{x}_i - \bar{x})(\bar{y}_i - \bar{y})}{\sqrt{(\bar{x}_i - \bar{x})^2} \sqrt{(\bar{y}_i - \bar{y})^2}}$ . The elements of the confusion matrix can be shown to be  $a = \sum_{i=1}^n \bar{x}_i \bar{y}_i$ ,  $b = \sum_{i=1}^n \bar{x}_i (1 - \bar{y}_i)$ ,  $c = \sum_{i=1}^n (1 - \bar{x}_i) \bar{y}_i$ ,  $d = \sum_{i=1}^n (1 - \bar{x}_i)(1 - \bar{y}_i)$ , from which the equation for  $r_m$  can be derived. Thus,  $r_m$  has PRE- and angle-interpretation.

$r_m$  has been generalised to  $k \times k$  confusion matrices [14], for which we use the nomenclature of contingency tables in fig. 11.2 on page 365:

$$r_k = \frac{\text{tr}(\mathcal{T}) \times n - \sum_{i=1}^k (\mathfrak{c}_i \mathfrak{r}_i)}{\sqrt{(n^2 - \sum_{i=1}^k \mathfrak{c}_i^2)(n^2 - \sum_{i=1}^k \mathfrak{r}_i^2)}} \quad (11.35)$$

where  $\text{tr}(T)$  (sum of diagonal elements) is the number of correct predictions, the row-sums  $r_i$  give the number of times that class  $i$  occurred, the column sums  $c_i$  the number of times that class  $i$  was predicted and  $n$  is the total number of observations. The problem with  $r_k$  compared to  $r_m$  is that although the maximum value remains +1, the minimal value is somewhere between -1 and 0.

Listing 11.14: GORODKIN's multi-class correlation  $r_k$ 

```

1  FUNCTION Rk(Cont: ContTable): float;
2
3  VAR
4      k, i                      : WORD;
5      Sum1, Sum2, Sum3, Sum4 : float;
6      c                         : char;
7
8  BEGIN
9      k := Cont^.r;
10     IF k <> Cont^.c
11         THEN
12             BEGIN
13                 c := WriteErrorMessage('Gorodkin''s Rk: Contingency table not
14                     square ');
15                 CorrelationsError := TRUE;
16                 EXIT;
17             END;
18     Sum1 := 0;
19     Sum2 := 0;
20     Sum3 := 0;
21     Sum4 := 0;
22     FOR i := 1 TO k DO
23         BEGIN
24             Sum1 := Sum1 + (Cont^.RowSums[i] * Cont^.ColumnSums[i]);
25             Sum2 := Sum2 + (Cont^.ColumnSums[i] * Cont^.ColumnSums[i]);
26             Sum3 := Sum3 + (Cont^.RowSums[i] * Cont^.RowSums[i]);
27             Sum4 := Sum4 + Cont^.Table[i, i];
28         END;
29     Result := (Sum4 * Cont^.n - Sum1) / Sqrt((Cont^.n * Cont^.n - Sum2) *
30     (Cont^.n * Cont^.n - Sum3));
31 END;
```

In principle, a correlation can be calculated even if the number of classes in the observation  $k$  is not equal to the number of classes in the prediction  $l$ , resulting in a non-square confusion matrix  $C_{k \times l}$ ,  $l \neq k$ . In such a confusion matrix,  $c_{ij}$  would be the number of elements of observed class  $i \in 1 \dots k$  and predicted class  $j \in 1 \dots l$ .

We can then calculate TP, TN, FP and FN for the  $i$ -th row (or column), from the number of samples where  $i$  is

## 11. Correlation coefficients

	observed class	predicted class
TP <sub>i</sub> (=a)	✓	✓
FP <sub>i</sub> (=c)	✗	✓
TN <sub>i</sub> (=d)	✗	✗
FN <sub>i</sub> (=b)	✓	✗

these are then used to calculate the row- (or column)-wise  $r_i$ .

### 11.3.2. GOODMAN & KRUSKAL's $\lambda$ for binary/binary association

Binary data can be viewed as a special case of nominal, with only two classes. Therefore, the symmetrical version  $\lambda$  [15] (see below) can be used also for binary/binary associations.  $\lambda$  has a PRE interpretation.

### 11.3.3. The point biserial correlation $r_j$ for binary/cardinal association

$r_j$  is simply the product moment correlation between the rational data and binary data taken as numerical values (usually 0 and 1). As such,  $r_j$  has both PRE and angle interpretation [4]. There is, however, a numerically more elegant way to calculate it:

$$r_j = \frac{\bar{X}_1 - \bar{X}_0}{s} \sqrt{\frac{n_1 n_o}{N^2}} \quad (11.36)$$

with  $n_1, n_0$  the number of cases who got 1 and 0, respectively,  $N = n_0 + n_1$  the total number of cases,  $\bar{X}_1, \bar{X}_0$  the group averages of cases who got 1 and 0,  $\bar{X}$  the overall average and with

$$s = \sqrt{\frac{1}{N} \sum_{i=1}^N (X_i - \bar{X})^2} \quad (11.37)$$

the standard deviation of the total average  $\bar{X}$ .

$r_j$  is used in test theory of binary exams to calculate how well student's results on a particular question mirror their overall performance in the test. When used for this purpose,  $s$  needs to be calculated without the question under test, at least if the number of questions is small.

**Significance of  $r_j$**  A t-test can be used to test  $H_0 : r_j = 0$  vs  $H_1 : r_j \neq 0$

$$t \approx r_j \sqrt{\frac{N-2}{1-r_j^2}}, \quad f = N-2 \quad (11.38)$$

Listing 11.15: PointBiserial correlation and its significance

```

1  FUNCTION PointBiserialCorrelation(CONST BinaryVector, CardinalVector:
2      VectorTyp;
3      VAR Significance: SignificanceType): float;
4
5
6
7
8
9  BEGIN
10     IF NOT (TestDataVectorLength(BinaryVector, CardinalVector)) THEN EXIT;
11     n := VectorLength(CardinalVector);
12     TotalI := 0;
13     SumX := 0;
14     FOR i := 1 TO n DO          // calculate average x
15         IF NOT (IsNaN(GetVectorElement(CardinalVector, i)) OR
16             IsNaN(GetVectorElement(BinaryVector, i)))
17         THEN
18             BEGIN
19                 INC(TotalI);
20                 SumX := SumX + GetVectorElement(CardinalVector, i);
21             END;
22             BarX := SumX / TotalI;
23             SumX := 0;
24             FOR i := 1 TO n DO          // calculate standard deviation OF x
25                 IF NOT (IsNaN(GetVectorElement(CardinalVector, i)) OR
26                     IsNaN(GetVectorElement(BinaryVector, i)))
27                 THEN
28                     SumX := SumX + (GetVectorElement(CardinalVector, i) - barX) *
29                         (GetVectorElement(CardinalVector, i) - barX);
30             s := Sqrt(SumX / TotalI);
31             GoodI := 0;
32             BadI := 0;
33             SumXGood := 0;
34             SumXBad := 0;
35             FOR i := 1 TO n DO
36                 IF (IsNaN(GetVectorElement(CardinalVector, i)) OR
37                     IsNaN(GetVectorElement(BinaryVector, i)))
38                 THEN // ignore
39                 ELSE IF (GetVectorElement(BinaryVector, i) = 1)
40                 THEN
41                     BEGIN
42                         INC(GoodI);
43                         SumXGood := SumXGood + GetVectorElement(CardinalVector, i);
44                     END

```

## 11. Correlation coefficients

```

45          ELSE IF (GetVectorElement(BinaryVector, i) = 0)
46          THEN
47          BEGIN
48              INC(BadI);
49              SumXBad := SumXBad +
50                  GetVectorElement(CardinalVector, i);
51          END
52          ELSE
53          BEGIN
54              c := WriteErrorMessage('Point biserial correlation:
55                  illegal value in exam table');
56              CorrelationsError := TRUE;
57              EXIT;
58          END;
59      IF ((GoodI = 0) OR (BadI = 0))
60      THEN
61          r := 0
62      ELSE
63          BEGIN
64              SumXGood := SumXGood / GoodI;
65              SumXBad := SumXBad / BadI;
66              x := Sqrt(GoodI * BadI / TotalI / TotalI);
67              y := (SumXGood - SumXBad) / s;
68              r := x * y;
69          END;
70      CorrelationSignificance(BinaryVector, CardinalVector, r, Significance);
71      Result := r;
72  END;

```

## 11.4. Ordinal data

Ordinal and nominal data are conveniently summarised in contingency tables like tab. 11.2, with  $t_{ij}$  the number of data where  $x = i$  and  $y = j$ .  $X$  is the independent variable and defines the rows of the table,  $Y$  is the dependent variable and defines the columns.  $n$  is the length of the data vectors,  $r, c$  the number of rows and columns of the contingency table, which is also the number of unique values in  $X, Y$ , respectively. Conventionally, loop counters are  $i$  for rows and  $j$  for columns.

Listing 11.16: Handling of contingency tables

```

1  PROCEDURE Contingency(Data1, Data2: VectorTyp; VAR Cont: ContTable);
2
3  VAR
4      i, j, l, m, Length : WORD;
5      c                  : char;
6

```

Table 11.2.: Contingency table  $T$  for nominal and ordinal data vectors  $\mathbf{x}, \mathbf{y}$  of length  $n$  with definitions of the symbols used in the text.  $\mathbf{R}, \mathbf{C}$  are the row- and column-sums, respectively.

$\mathbf{x} \setminus \mathbf{y}$	$y_1$	$\dots$	$y_j$	$\dots$	$y_c$	$ $	$\mathbf{R}$
$\mathbf{x}_1$	$t_{11}$						$r_{1\cdot}$
$\vdots$							
$\mathbf{x}_i$			$t_{ij}$				$r_{i\cdot}$
$\vdots$							
$\mathbf{x}_r$					$t_{rc}$		$r_{r\cdot}$
$\mathbf{C}$	$c_{\cdot 1}$	$\dots$	$c_{\cdot j}$	$\dots$	$c_{\cdot c}$	$ $	$n$

```

7   BEGIN
8     IF NOT (TestDataVectorLength(Data1, Data2)) THEN EXIT;
9     Length := VectorLength(Data1);
10    TRY
11      GetMem(Cont, SizeOf(ContStruc));
12    except
13      c := WriteErrorMessage(' Contingency table: Not enough memory to create
14        table');
15      EXIT;
16    END;
17    FOR i := 0 TO MaxSteps DO
18      // we don't know number of different values yet, so use maximum
19      BEGIN
20        FOR j := 0 to MaxSteps DO
21          Cont^.Table[i, j] := 0;
22          Cont^.RowSums[i] := 0;
23          Cont^.ColumnSums[i] := 0;
24        END;
25        Cont^.n := 0;
26        Cont^.r := 0;
27        Cont^.c := 0;
28        FOR i := 1 TO Length DO
29          BEGIN
30            IF (IsNaN(GetVectorElement(Data1, i)) OR
31                IsNaN(GetVectorElement(Data2, i)))
32              THEN // ignore
33            ELSE
34              BEGIN
35                l := trunc(GetVectorElement(Data1, i));
36                m := trunc(GetVectorElement(Data2, i));
37                Inc(Cont^.Table[l, m]);

```

## 11. Correlation coefficients

```
36          IF (l > Cont^.r)
37              THEN Cont^.r := l; // set r to highest value of Data1
38          IF (m > Cont^.c)
39              THEN Cont^.c := m; // set c to highest value of Data2
40          Inc(Cont^.n);
41      END;
42  END;
43  FOR i := 0 TO Cont^.r DO
44      FOR j := 0 TO Cont^.c DO
45          BEGIN
46              Cont^.RowSums[i] := Cont^.RowSums[i] + Cont^.Table[i, j];
47              Cont^.ColumnSums[j] := Cont^.ColumnSums[j] + Cont^.Table[i, j];
48          END;
49      END;
50
51
52 PROCEDURE DestroyContingency(var Cont : ContTable);
53
54 BEGIN
55     FreeMem(Cont, SizeOf(ContStruc));
56 END;
57
58
59 PROCEDURE WriteContingency(CONST Cont : ContTable; MedStr : String;
60                           ValidFigures : byte);
61
62 VAR i, j      : WORD;
63     Medium    : TEXT;
64     c          : CHAR;
65
66 BEGIN
67     IF (MedStr = 'CON')
68     THEN
69         BEGIN
70             FOR i := 0 TO Cont^.r DO
71                 BEGIN
72                     FOR j := 0 TO Cont^.c Do
73                         write(Cont^.Table[i,j]:ValidFigures, ' ');
74                         writeln(' ', Cont^.RowSums[i]:ValidFigures);
75                 END;
76                 FOR j := 0 TO Cont^.c DO
77                     FOR i := 1 TO succ(ValidFigures) DO write('_');
78                     write(' ');
79                     FOR i := 1 TO succ(ValidFigures) DO write('_');
80                     writeln;
```

```

80      FOR j := 0 TO Cont^.c DO write(Cont^.ColumnSums[j]:ValidFigures, ' ');
81      writeln(' | ', Cont^.n:ValidFigures);
82  END
83 ELSE
84 BEGIN
85 TRY
86 assign(Medium, MedStr);
87 rewrite(Medium);
88 EXCEPT
89   c := WriteErrorMessage(' Writing contingency table: could not
90     open file ', MedStr:length(MedStr));
91   CorrelationsError := true;
92   EXIT;
93 END;
94 FOR i := 0 TO Cont^.r DO
95 BEGIN
96   FOR j := 0 TO Cont^.c DO
97     write(Medium, Cont^.Table[i,j]:ValidFigures, ' ');
98     writeln(Medium, ' | ', Cont^.RowSums[i]:ValidFigures);
99   END;
100  FOR j := 0 TO Cont^.c DO
101    FOR i := 1 TO succ(ValidFigures) DO write(Medium, '_');
102    write(Medium, '|');
103    FOR i := 1 TO succ(ValidFigures) DO write(Medium, '_');
104    writeln(Medium);
105    FOR j := 0 TO Cont^.c DO write(Medium,
106      Cont^.ColumnSums[j]:ValidFigures, ' ');
107      writeln(Medium, ' | ', Cont^.n:ValidFigures);
108    END;
109  END;
110 END;

```

### 11.4.1. Ordinal/ordinal associations

According to [8], data on an at least ordinal (but possibly higher) scale can be arranged in an ordered contingency table so that  $x_1 < x_2 < \dots < x_n$  and  $y_1 < y_2 < \dots < y_n$ . If there are  $n$  observations, then there are  $P = \binom{n}{2} = (n^2 - n)/2$  pairs of observation, of which there are

- C concordant pairs
- D discordant pairs
- $X_0$  pairs tied in only  $x$
- $Y_0$  pairs tied in only  $y$
- Z pairs tied in both  $x$  and  $y$

with  $P = C + D + X_0 + Y_0 + Z$ . If  $P = C$ , then the data have perfect positive association. Likewise, if  $P = D$ , association is perfect but negative. The difference  $C - D$  indicates how

## 11. Correlation coefficients

much the data follow either of these limiting cases. If  $C = D$  then  $x$  and  $y$  are independent and the difference is 0. If instead of absolute numbers the probabilities  $P(C)$  and  $P(D)$  are used, the measure becomes independent of sample size.

Monotonic functions in mathematics can be divided as follows:

increasing	for all $x_i > x_j \Rightarrow y_i > y_j, x_i = x_j \Rightarrow y_i = y_j$
decreasing	for all $x_i > x_j \Rightarrow y_i < y_j, x_i = x_j \Rightarrow y_i = y_j$
non-decreasing	for all $x_i > x_j \Rightarrow y_i \geq y_j$
non-increasing	for all $x_i > x_j \Rightarrow y_i \leq y_j$

In statistics, however, several repetitions of measurements for a  $x_i$  may (and usually will) result in different  $y$ -values, due to measurement error. We are therefore not dealing with *functions*, but with *relations*.  $Z$ -ties occur if all measurements of  $x_i$  result in the same  $y_i$ , they are *replications* that occur in the same cell of the contingency table and have no effect on the model. When mapping from  $\mathfrak{X}$  to  $\mathfrak{Y}$ , monotonic one-to-one relations disallow both  $X_0$ -ties and  $Y_0$ -ties, monotonic functions (many to one) exclude  $X_0$ -ties, but  $Y_0$ -ties are allowed. In monotonic relations (many to many), both are allowed. These three models have different association measures, with different PRE interpretation:

### KENDALL's $\tau$ for monotonic relations with $X_0 = Y_0 = Z = 0$

If we guess the order of  $\mathfrak{Y}$  without reference to  $\mathfrak{X}$ , then  $y_i > y_j$  has a probability of 1/2 and  $E_1 = (C+D)/2$ . If we take  $\mathfrak{X}$  into account, we guess concordance for all pairs if  $C > D$  and discordance if  $D > C$ . Then  $E_2$  is the smaller of  $C$  and  $D$  and

$$\tau = \frac{(C+D)/2 - D}{(C+D)/2} = \frac{C-D}{C+D} \quad (11.39)$$

For an example, see table 11.3.

### GOODMAN & KRUSKAL's $\gamma$ if ties are ignored

The same reasoning applies as for  $\tau$ , except that ties are not prohibited, but simply ignored.

### KIM's $d_{y,x}$ for monotonic functions

We choose only data pairs where  $y_i \neq y_j$ , and therefore eliminate  $Y_0$  and  $Z$ . Then the initial guessing error is 1/2 for all remaining pairs, that is,  $E_1 = 1/2(C+D+X_0)$ . If we guess taking  $\mathfrak{X}$  into account as discussed above, the cases where  $x_i = x_j$  give us no information and their error is 1/2. This has to be added to the errors for the untied cases and  $E_2 = D+X_0/2$

$$d_{y,x} = \frac{C-D}{C+D+X_0} \quad (11.41)$$

$$d_{x,y} = \frac{C-D}{C+D+Y_0} \quad (11.42)$$

Table 11.3.: Example data for the calculation of  $\tau$  and  $\gamma$ , taken from [16], p. 84

Family interest	Expert Rank	Empirical Rank	Inversion (D)	Agreement (C)
Demonstrating affection	10	9	0	0
Planning for future	9	6	0	1
Planning saving	8	10	2	0
Training children	7	1	0	3
Planning budget	6	8	2	2
Plans for children	5	7	2	3
Home decoration	4	4	1	5
Preparing meals	3	5	2	5
Shopping	2	3	1	7
Houscleaning	1	2	1	8
Totals			11	34

$$\gamma = \frac{C - D}{C + D} = \frac{34 - 11}{34 + 11} = 0.511 \quad (11.40)$$

$d_{y,x}$  (dependent  $\Psi$  from independent  $\Xi$ ) and  $d_{x,y}$  (dependent  $\Xi$  from independent  $\Psi$ ) are asymmetric measures of association, which have PRE interpretation. The average of  $d_{x,y}$  and  $d_{y,x}$ , however, cannot be used as a symmetric PRE measure of association.

### WILSON's $e$ for monotonic one-to-one correspondence

The guessing to determine  $E_1$  and  $E_2$  are performed twice, for  $\Xi$  and  $\Psi$ . To guess  $\Psi$ , we make the condition that  $x_i \neq x_j$ , and to guess  $\Xi$ , we make the condition that  $y_i \neq y_j$ .

$$E_1(Y) = (C + D)/2 + Y_0 \quad (11.43)$$

$$E_1(X) = (C + D)/2 + X_0 \quad (11.44)$$

$$E_1 = E_1(X) + E_1(Y) = C + D + X_0 - Y_0 \quad (11.45)$$

$$E_2(Y) = D + Y_0 \quad (11.46)$$

$$E_2(X) = D + X_0 \quad (11.47)$$

$$E_2 = E_2(X) + E_2(Y) = 2D + X_0 + Y_0 \quad (11.48)$$

$$e = \frac{C - D}{C + D + X_0 + Y_0} \quad (11.49)$$

is a symmetrical measure of association with PRE interpretation. In the special case of  $X_0 = Y_0 = Z = 0$  then  $\tau = e$ .

## 11. Correlation coefficients

### Significance testing of these coefficients

For  $n > 10$ , the limiting sampling distribution of  $S = |C - D|$  is the normal distribution, even when ties are present. If there is no correlation between  $\chi$  and  $\psi$  then  $E(\gamma) = 0$ .  $S$  should be corrected for the step-wise (rather than smooth) distribution according to

$$\hat{S} = |C - D| - \frac{n}{2(R - 1)(C - 1)} \quad (11.50)$$

and the standard error of  $\hat{S}$  is

$$s = \sqrt{\frac{U_2 V_2}{n-1} - \frac{U_2 V_3 + V_2 U_3}{n(n-1)} + \frac{U_3 V_3}{n(n-1)(n-2)}} \quad (11.51)$$

with  $U_x, V_x$  the sum of products of row and column totals taken  $x$  at a time. Example (taken from [16]):

y \ x	4	3	2	1	r
2	4	5	0	1	10
1	0	2	6	4	12
C	4	7	6	5	22

Then

$$U_2 = 10 \times 12 = 120$$

$$V_2 = 4 \times 7 + 4 \times 6 + 4 \times 5 + 7 \times 6 + 7 \times 5 + 6 \times 5 = 179$$

$$U_3 = 0 \quad \text{since there are only two rows}$$

$$V_3 = 4 \times 7 \times 6 + 4 \times 7 \times 5 + 4 \times 6 \times 5 = 638$$

$$s = \sqrt{\frac{120 \times 179}{21} - \frac{120 \times 638 + 179 \times 0}{22 \times 21} + \frac{0}{22 \times 21 \times 20}} \\ = \sqrt{1022.86 - 165.07 + 0} = \sqrt{857.79} = 29.29$$

$$\hat{S} = |98 - 8| - \frac{22}{2 \times 1 \times 3} = 90 - 3.67 = 86.33$$

$$z = \frac{\hat{S}}{s} = \frac{86.33}{29.29} = 2.95$$

$z$  is so large that the probability of the Null-hypothesis is less than 0.01.

Listing 11.17: Correlation coefficients for ordinal/ordinal associations

```

1  FUNCTION OrdinalCorrelations(CONST Data1, Data2: VectorTyp; Formula: CHAR;
2    VAR Significance: SignificanceType): float;
3
4  VAR
5    i, j, k, C, D, X0, Y0, Z, Length: WORD;
6    x1, x2, y1, y2, U2, U3, V2, V3: float;
7    Cont: ContTable;
```

```

8
9 BEGIN
10 IF NOT (TestDataVectorLength(Data1, Data2)) THEN EXIT;
11 Length := VectorLength(Data1);
12 Contingency(Data1, Data2, Cont);
13 C := 0;
14 D := 0;
15 X0 := 0;
16 Y0 := 0;
17 Z := 0;
18 FOR i := 1 TO Length DO
19   FOR j := Succ(i) TO Length DO
20     BEGIN
21       x1 := GetVectorElement(Data1, i);
22       x2 := GetVectorElement(Data1, j);
23       y1 := GetVectorElement(Data2, i);
24       y2 := GetVectorElement(Data2, j);
25       IF (IsNaN(x1) OR IsNaN(x2) OR IsNaN(y1) OR IsNaN(y2))
26         THEN // ignore NaNs
27       ELSE IF (Trunc(x1) = Trunc(x2))
28         THEN IF (Trunc(y1) = Trunc(y2))
29           THEN INC(Z)                      // tie IN X AND Y
30           ELSE INC(X0)                   // tie IN X only
31       ELSE IF (Trunc(y1) = Trunc(y2))
32           THEN INC(Y0)                  // tie IN Y only
33       ELSE IF (Trunc(x1) > Trunc(x2))
34         THEN IF (Trunc(y1) > Trunc(y2))
35           THEN INC(C)
36           ELSE INC(D)
37       ELSE IF (Trunc(y1) > Trunc(y2))
38           THEN INC(D)
39           ELSE INC(C);
40     END; // FOR j
41 CASE UpCase(Formula) OF
42   'T', 'G': BEGIN // calculate Kendall's tau or Goodman & Kruskal's gamma
43     Result := (1.0 * C + D);
44     IF (Result = 0)
45       THEN      // ???
46       ELSE Result := (1.0 * C - D) / Result;
47   END;
48   'D': BEGIN // calculate Kim's d_{xy} (note that d_{xy} <> d_{yx})
49     Result := C + D + X0;
50     if (Result = 0)
51       then      // ???
52       else Result := (1.0 * C - D) / Result;
53   end;

```

## 11. Correlation coefficients

```

54     'E':      begin // calculate Wilson's e
55         Result := C + D + X0 + Y0;
56         IF (Result = 0)
57             THEN      // ???
58             ELSE Result := (1.0 * C - D) / Result;
59         END;
60     END; // case
61     U2 := 0; // now start WITH significance
62     FOR i := 0 TO Cont^.r DO
63         FOR j := Succ(i) TO Cont^.r DO
64             U2 := U2 + Cont^.RowSums[i] * Cont^.RowSums[j];
65             // sum OF product OF row sums 2 at a time
66             V2 := 0;
67             FOR i := 0 TO Cont^.c DO
68                 FOR j := Succ(i) TO Cont^.c DO
69                     U2 := U2 + Cont^.ColumnSums[i] * Cont^.ColumnSums[j];
70                     // sum OF product OF column sums 2 at a time
71             U3 := 0;
72             FOR i := 0 TO Cont^.c DO
73                 FOR j := Succ(i) TO Cont^.c DO
74                     FOR k := Succ(j) TO Cont^.c DO
75                         U3 := U3 + Cont^.RowSums[i] * Cont^.RowSums[j] * Cont^.RowSums[k];
76                         // sum OF product OF row sums 3 at a time
77             V3 := 0;
78             FOR i := 0 TO Cont^.c DO
79                 FOR j := Succ(i) TO Cont^.c DO
80                     FOR k := Succ(j) TO Cont^.c DO
81                         U3 := U3 + Cont^.ColumnSums[i] * Cont^.ColumnSums[j] *
82                             Cont^.ColumnSums[k];
83                         // sum of product of column sums 3 at a time
84             Significance.Freedom := Sqrt(U2 * V2 / Pred(Cont^.n) - (U2 * V3 + V2 *
85                 U3) /
86                 (Cont^.n * Pred(Cont^.n)) + U3 * V3 / (Cont^.n * Pred(Cont^.n) *
87                 (Cont^.n - 2)));
88             // standard deviation
89             Significance.P0 := 0;
90             // if it can't be calculated
91             Significance.TestValue := (2 * pred(Cont^.r) * pred(Cont^.c));
92             IF (Significance.TestValue <> 0) THEN
93                 Significance.TestValue := abs(C - D) - Cont^.n /
94                 Significance.TestValue; // average
95             IF (Significance.Freedom <> 0) THEN
96                 Significance.P0 := IntegralGauss(Significance.TestValue /
97                 Significance.Freedom);
98                 // error corresponding to z
99             END;

```

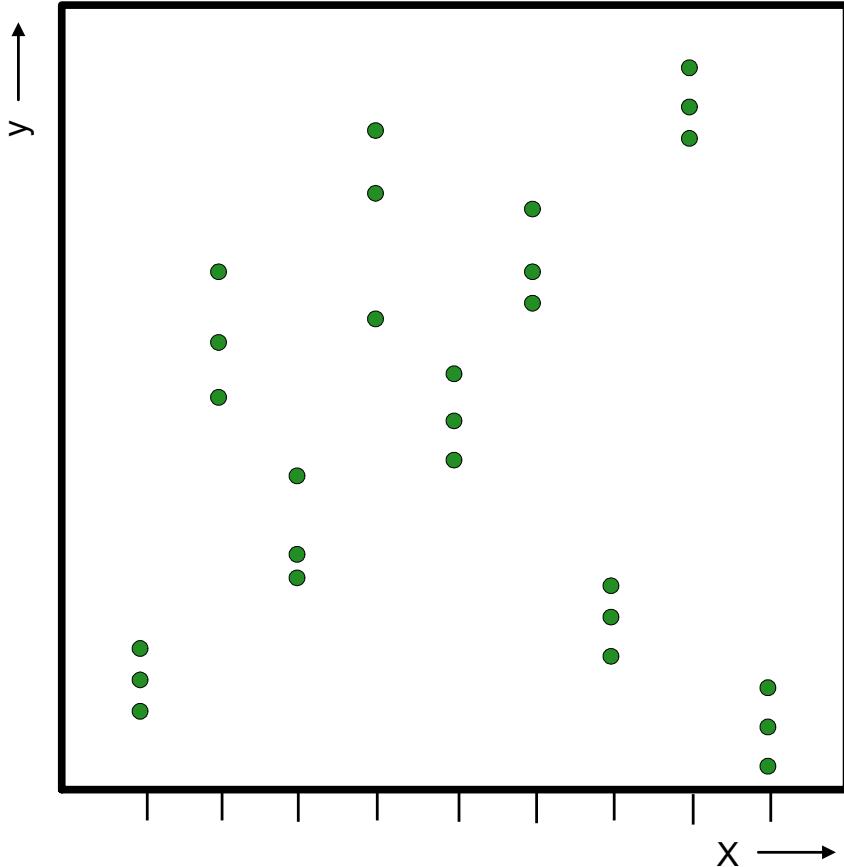


Figure 11.2.: High intraclass correlation: each class ( $\mathfrak{x}$ ) is associated with a particular range of  $y$ -values. Note that since  $\mathfrak{x}$  is on a nominal scale the order of  $\mathfrak{x}$ -values is random. The rank correlation coefficient  $r_s = 0.04$ , but  $r_i = 0.94$ .

## 11.5. Nominal data

### 11.5.1. Intraclass correlation for nominal/cardinal association

For nominally scaled data, neither  $r_p$  nor  $r_s$  is appropriate. However, we can use  $r_i$  [17] to check whether certain values of  $y$  concentrate in certain classes of  $\mathfrak{x}$  (see fig. 11.2). If we have  $A$  groups with  $B$  data points per group, then  $r_i$  becomes:

$$r_i = \frac{B}{B-1} \frac{A^{-1} \sum_{a=1}^A (\bar{y}_a - \bar{y})^2}{s^2} - \frac{1}{B-1} \quad (11.52)$$

$$\approx \frac{A^{-1} \sum_{a=1}^A (\bar{y}_a - \bar{y})}{s^2} \quad \text{for large } B \quad (11.53)$$

$$s^2 = \frac{1}{AB} \sum_{b=1}^B \sum_{a=1}^A (y_{a,b} - \bar{y})^2 \quad (11.54)$$

## 11. Correlation coefficients

with  $\bar{y}_a$  the arithmetic mean of the  $y$ -values in the  $a$ -th group. Biggest limitation of  $r_i$  is that all groups need to have the same number of data points,  $B$ .

### 11.5.2. CRAMÉR's V for nominal/nominal association

To compare two nominal scaled data vectors, each of length  $n$ , one uses a contingency table  $\mathcal{T}$  with  $r$  rows and  $c$  columns. Then  $\chi^2$  is calculated as follows:

$$\mathfrak{h}_i = \sum_{j=1}^c \frac{\mathcal{T}_{i,j}^2}{\mathfrak{C}_j} \quad (11.55)$$

$$A = \sum_{i=1}^r \frac{\mathfrak{h}_i}{r_i} \quad (11.56)$$

$$\chi^2 = (A - 1) \times n \quad (11.57)$$

$$f = (r - 1)(c - 1) \quad (11.58)$$

$$(11.59)$$

with  $\mathfrak{C}_j$  the column sums,  $r_i$  the row sums of the contingency table and  $f$  the degrees of freedom [18]. With  $\varphi^2 = \chi^2/n$

$$V = \sqrt{\frac{\varphi^2}{\min(r-1, c-1)}} \quad (11.60)$$

$V$  reaches from 0...1 (*not* -1...1, with nominal values increase and decrease have no meaning!), but tends to overestimate the strength of association between the data vectors. To correct for this bias,  $\tilde{V}$  is used:

$$\tilde{V} = \sqrt{\frac{\tilde{\varphi}^2}{\min(\tilde{r}-1, \tilde{c}-1)}} \quad (11.61)$$

$$\tilde{\varphi}^2 = \max(0, \varphi^2 - \frac{(c-1) \times (r-1)}{(n-1)}) \quad (11.62)$$

$$\tilde{c} = c - \frac{(c-1)^2}{(n-1)} \quad (11.63)$$

$$\tilde{r} = r - \frac{(r-1)^2}{(n-1)} \quad (11.64)$$

To test for significance ( $H_0$ : no association between variables) one uses the  $\chi^2$ -test with  $f$  degrees of freedom.  $V$  does not have a PRE interpretation.

Listing 11.18: Common correlation measures without PRE-interpretation for enumeration types

```
1 FUNCTION Chi2(const Contingency: ContTable;
```

```

2  var Significance: SignificanceType): float;
3
4  VAR
5    H: array[0..MaxSteps] of float;
6    i, j: word;
7    A: float;
8
9  BEGIN
10   FOR i := 1 TO MaxSteps DO
11     H[i] := 0.0;
12   FOR i := 0 TO Contingency^.r DO
13     FOR j := 0 TO Contingency^.c DO
14       BEGIN
15         A := Contingency^.Table[i, j];
16         IF Contingency^.ColumnSums[j] = 0
17           THEN A := 0
18         ELSE A := A * A / Contingency^.ColumnSums[j];
19         H[i] := H[i] + A;
20       END;
21   A := 0;
22   FOR i := 0 TO Contingency^.r DO // calculate first phi^2 = chi^2 / n
23     BEGIN
24       IF (Contingency^.RowSums[i] = 0)
25         THEN
26           ELSE A := A + H[i] / Contingency^.RowSums[i];
27     END;
28   A := (A - 1) * Contingency^.n;
29   Significance.TestValue := A;
30   Significance.Freedom := pred(Contingency^.r) * pred(Contingency^.c);
31   Significance.P0 := IntegralChi(Significance.TestValue,
32                                 round(Significance.Freedom));
33   Result := A;
34
35
36  FUNCTION Phi2(CONST Contingency: ContTable;
37    VAR Significance: SignificanceType): float;
38
39  VAR
40    x: float;
41
42  BEGIN
43    x := chi2(Contingency, Significance);
44    Result := x / Contingency^.n;
45  END;
46

```

## 11. Correlation coefficients

```

47 FUNCTION CramersVTilde(chi2: float; r, c, n: word): float;
48
49 VAR
50   phi2, hilfs, ct, rt: float;
51
52 BEGIN
53   phi2 := chi2 / n;
54   hilfs := phi2 - pred(c) * pred(r) / pred(n);      // now calculate
      tilde-version
55   IF hilfs > 0
56     THEN phi2 := hilfs
57   ELSE phi2 := 0;
58   ct := c - PRED(c) * PRED(c) / PRED(n);
59   rt := r - PRED(r) * PRED(r) / PRED(n);
60   IF rt > ct
61     THEN rt := ct;                                // determine min(r~,c~)
62   Result := sqrt(phi2 / (rt - 1));
63 end;

```

### 11.5.3. GUTTMAN's coefficient of relative predictability $\lambda$ for nominal/nominal association

A different approach to measuring association is to attempt to predict the values of one variable in two different ways [15, 19, 20]. First, the values for one variable are predicted without knowing the values of a second variable. Then the values of the first variable are predicted again, after taking into account the values of the second variable. The extent to which the error of prediction for values of one variable can be reduced by knowing the value of a second variable forms the basis for the reduction in error approach to measuring association.

If the variables are cross-tabulated as described for CRAMÉR's V above, then without knowledge of the columns one would pick that row  $i$  with the largest row-sum  $r_i$ , as this prediction would minimise prediction error, which is  $E_n = n - r_i$ .

If, however, it is known that the same individual is in column  $j$ , then one would pick the  $i$  such that  $T_{i,j}$  is maximal, and the prediction error would become  $C_j - T_{i,j}$ . The total prediction error  $E_j$  is obtained by summing up the errors thus obtained for all rows.

$\lambda$  is defined as the proportional reduction in error (PRE), that is

$$\lambda = \frac{E_n - E_j}{E_n} \quad (11.65)$$

and is the fraction by which the prediction error for the dependent variable can be reduced by knowledge of independent. For example,  $\lambda = 0.63$  means that the error can be reduced by 63 % or about 2/3. Therefore,  $0 \leq \lambda \leq 1$ , with  $\lambda = 0$  meaning no

Table 11.4.: Example data from [15]: Hair colour *vs.* eye color. The results for various measures of association are given at the bottom.

Eye \ Hair	Fair	Brown	Black	Red	$r$
Blue	1768	807	189	47	2811
Grey/Green	946	1387	746	53	3132
Brown	115	438	288	16	857
$\mathfrak{C}$	2829	2632	1223	116	n = 6800

$\lambda_r$	$\lambda_c$	$\lambda$	$\phi^2$	V
0.2241	0.1924	0.2076	0.1581	0.2812

association,  $\lambda = 1$  meaning perfect association. Thus:

$$\text{EC}_j = \mathfrak{C}_j - \max(\mathcal{T}_{\cdot,j}) \quad (11.66)$$

$$\text{EC} = \sum_{j=1}^c \text{EC}_j \quad (11.67)$$

$$\text{ET}_r = n - \max_i(r) \quad (11.68)$$

$$\lambda_r = \frac{\text{ET}_r - \text{E}_c}{\text{ET}_r} = \frac{n - \max_i(r) - \sum_{j=1}^r (\mathfrak{C}_j - \max(\mathcal{T}_{\cdot,j}))}{n - \max_i(r)} \quad (11.69)$$

and analogous for  $\lambda_c$ :

$$\text{ER}_i = r_i - \max(\mathcal{T}_{i,\cdot}) \quad (11.70)$$

$$\text{ER} = \sum_{i=1}^r \text{ER}_i \quad (11.71)$$

$$\text{ET}_c = n - \max_j(\mathfrak{C}) \quad (11.72)$$

$$\lambda_c = \frac{\text{ET}_c - \text{ER}}{\text{ET}_c} = \frac{n - \max_j(\mathfrak{C}) - \sum_{i=1}^c (r_i - \max(\mathcal{T}_{i,\cdot}))}{n - \max_j(\mathfrak{C})} \quad (11.73)$$

Note that  $\lambda_r$  and  $\lambda_c$  are usually not identical. [15] gives a formula to calculate a symmetrical version of  $\lambda$ :

$$\lambda = \frac{\sum_{i=1}^r \max(\mathcal{T}_{i,\cdot}) + \sum_{j=1}^c \max(\mathcal{T}_{\cdot,j}) - \max_i(r) - \max_j(\mathfrak{C})}{2n - [\max_i(r) + \max_j(\mathfrak{C})]} \quad (11.74)$$

An almost identical result is obtained by calculating the average of  $\lambda_r$  and  $\lambda_c$  by eqn. 11.12.

Listing 11.19: GUTTMAN's asymmetrical and symmetrical  $\lambda$

## 11. Correlation coefficients

```
1  FUNCTION lambda(CONST Contingency: ContTable): float;
2
3  VAR
4      i, j: word;
5      MaxR, MaxC, SumMaxTi, SumMaxTj: float;
6      MaxTi, MaxTj: ARRAY[0..MaxSteps] OF WORD;
7      // can't be vectorType AS 0 must be legal index
8
9  BEGIN
10     FOR i := 0 TO MaxSteps DO
11         BEGIN
12             MaxTi[i] := 0;
13             MaxTj[i] := 0;
14         END;
15         SumMaxTi := 0;
16         FOR i := 0 TO Contingency^.r DO
17             BEGIN
18                 FOR j := 0 TO Contingency^.c DO
19                     IF MaxTi[i] < Contingency^.Table[i, j] THEN
20                         MaxTi[i] := Contingency^.Table[i, j];
21                     SumMaxTi := SumMaxTi + MaxTi[i];
22                 END;
23                 SumMaxTj := 0;
24                 FOR j := 0 TO Contingency^.c DO
25                     BEGIN
26                         FOR i := 0 TO Contingency^.r DO
27                             IF MaxTj[j] < Contingency^.Table[i, j]
28                                 THEN MaxTj[j] := Contingency^.Table[i, j];
29                         SumMaxTj := SumMaxTj + MaxTj[j];
30                     END;
31                     MaxR := Contingency^.RowSums[0];
32                     FOR i := 1 TO Contingency^.r DO
33                         IF (Contingency^.RowSums[i] > MaxR)
34                             THEN MaxR := Contingency^.RowSums[i];
35                     MaxC := Contingency^.ColumnSums[0];
36                     FOR j := 1 TO Contingency^.c DO
37                         IF (Contingency^.ColumnSums[j] > MaxC)
38                             THEN MaxC := Contingency^.ColumnSums[j];
39                     Result := (2 * Contingency^.n - (MaxR + MaxC));
40                     IF Result = 0
41                         // ???
42                     ELSE lambda := (SumMaxTi + SumMaxTj - MaxR - MaxC) / lambda;
43                 END;
44
45
46 FUNCTION lambda_r(CONST Contingency: ContTable): float;
```

```

47
48 VAR
49   i, j: WORD;
50   MaxR, MaxT, SumCT: float;
51
52 BEGIN
53   MaxR := 0;
54   FOR i := 0 TO Contingency^.r DO
55     IF (Contingency^.RowSums[i] > MaxR)
56       THEN MaxR := Contingency^.RowSums[i];
57   SumCT := 0;
58   FOR j := 0 TO Contingency^.c DO
59     BEGIN
60       MaxT := Contingency^.Table[0, j];
61       FOR i := 1 TO Contingency^.r DO
62         IF (MaxT < Contingency^.Table[i, j])
63           THEN MaxT := Contingency^.Table[i, j];
64       SumCT := SumCT + Contingency^.ColumnSums[j] - MaxT;
65     END;
66   Result := Contingency^.n - MaxR;
67   IF (Result = 0)
68     THEN // ???
69   ELSE lambda_r := (Contingency^.n - MaxR - SumCT) / lambda_r;
70 END;
71
72
73 FUNCTION lambda_c(CONST Contingency: ContTable): float;
74
75 VAR
76   i, j: WORD;
77   MaxC, MaxT, SumRT: float;
78
79 BEGIN
80   MaxC := 0;
81   FOR j := 0 TO Contingency^.c DO
82     IF (Contingency^.ColumnSums[j] > MaxC)
83       THEN MaxC := Contingency^.ColumnSums[j];
84   SumRT := 0;
85   FOR i := 0 TO Contingency^.r DO
86     BEGIN
87       MaxT := Contingency^.Table[i, 0];
88       FOR j := 1 TO Contingency^.c DO
89         IF (MaxT < Contingency^.Table[i, j])
90           THEN MaxT := Contingency^.Table[i, j];
91       SumRT := SumRT + Contingency^.RowSums[i] - MaxT;
92     END;

```

## 11. Correlation coefficients

```

93  lambda_c := (Contingency^.n - MaxC);
94  IF lambda_c = 0
95    THEN          // ???
96    ELSE lambda_c := (Contingency^.n - MaxC - SumRT) / lambda_c;
97  END;

```

### 11.5.4. FREEMAN's measure of association $\theta$ for nominal/ordinal association

$\theta$  measures the reduction of error if an ordinal dependent variable is predicted from an independent variable on the nominal scale, compared to random guessing [16, 21]. It is therefore a PRE measure of association. Because variables on the nominal scale are unordered  $\theta$  can have only positive values, ranging from 0...1:

$$\theta = \frac{\sum_{i=1}^r D_i}{T} \quad (11.75)$$

with  $D_i = |f_b - f_a|$  for each comparison the difference of frequencies below and above, and  $T$  the number of comparisons made. Ties are considered to arise from our inability to precisely rank people and should be split between orders, however,  $|(f_b - 1/2f_{ties}) - (f_a - 1/2f_{ties})| = |f_b - f_a|$ , so ties are ignored. [16] gives as example social adjustment (ranked, i.e., ordinal) *vs.* marital status (nominal):

Status \ Rank	5	4	3	2	1	$\sum D_i$
Single	1	2	5	2	0	10
Married	10	5	5	0	0	20
Widowed	0	0	2	2	1	5
Divorced	0	0	0	2	3	5

For the comparison of single and married persons, we get:

Rank	$n_r$	$n_b$	$\Pi_b$	$n_a$	$\Pi_a$
5	1	10	10	0	0
4	2	5	10	10	20
3	5	0	0	15	75
2	2	0	0	20	40
1	0	0	0	18	0
$f$		20		135	

and  $D_i = |f_b - f_a| = |20 - 135| = 115$ . For single and widowed persons, we get:

Rank	$n_r$	$n_b$	$\Pi_b$	$n_a$	$\Pi_a$
5	1	5	5	0	0
4	2	5	10	0	0
3	5	3	15	0	0
2	2	1	2	2	4
1	0	0	0	4	0
$f$		32		4	

and  $D_i = |f_b - f_a| = |32 - 4| = 28$ . Analogously for the other comparisons:

Status 1	Status 2	$f_b$	$f_a$	$D_i$
single	divorced	46	0	46
married	widowed	90	0	90
married	divorced	100	0	100
widowed	divorced	16	2	14

Thus,  $\sum D_i = 115 + 28 + 46 + 90 + 100 + 14 = 393$ . T is calculated from the row sums of the data table:

$$T = 10 * 20 + 10 * 5 + 10 * 5 + 20 * 5 + 20 * 5 + 5 * 5 = 525 \quad (11.76)$$

Then  $\Theta = 393/525 = 0.75$

### Significance of $\Theta$

The distribution of  $\Theta$  is unknown. If the nominal vector has only two different values, the WILCOXON-MANN-WHITNEY U-test (first published in [22]) can be used.

Listing 11.20: FREEMAN's measure of association  $\Theta$

```

1  FUNCTION theta(CONST Contingency: ContTable): float;
2
3  VAR
4      i, j, k, l: word;
5      SumT, fa, fb, SumDi, nr: float;
6
7  BEGIN
8      SumT := 0;
9      FOR i := 0 TO Contingency^.r DO
10         FOR j := succ(i) TO Contingency^.r DO
11             SumT := SumT + Contingency^.RowSums[i] * Contingency^.RowSums[j];
12         SumDi := 0;
13         FOR i := 0 TO Contingency^.r DO
14             BEGIN
15                 FOR k := succ(i) TO Contingency^.r DO
16                     BEGIN
17                         fa := 0;
18                         fb := 0;
```

## 11. Correlation coefficients

```

19      FOR j := 0 TO Contingency^.c DO
20      BEGIN
21          nr := Contingency^.Table[i, j];
22          IF (j < Contingency^.c)
23              THEN
24                  FOR l := succ(j) TO Contingency^.c DO
25                      fa := fa + Contingency^.Table[k, l] * nr;
26          IF (j > 0)
27              THEN // this check actually is necessary
28                  FOR l := pred(j) DOWNTO 0 DO
29                      fb := fb + Contingency^.Table[k, l] * nr;
30          END;
31          SumDi := SumDi + abs(fb - fa);
32      END;
33  END;
34  IF (SumT = 0)
35      THEN Result := 0      // ???
36  ELSE Result := SumDi / SumT;
37 END;
```

### 11.5.5. $\eta^2$ for nominal/cardinal association

$\eta^2$  is also a PRE measure [23]. Assume we have a vector  $\mathbf{y}$  of cardinal (interval or rational) data. If we had to predict a  $y_i$ , we would use the arithmetic mean  $\bar{y}$  and the prediction error becomes

$$E_1 = \sum_{i=1}^n (y_i - \bar{y})^2 \quad (11.77)$$

If, however, we know that the test person  $i$  belongs into group  $k$  of some nominal scaled variable  $\mathbf{x}$ , then the predicted value would be the group average  $\bar{y}_k$  and the prediction error becomes:

$$E_2 = \sum_k \sum_{i=1}^n (y_i - \bar{y}_k)^2 \delta_{ik} \quad \text{for } \delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{else} \end{cases} \quad (11.78)$$

Then  $\eta^2$  becomes:

$$\eta^2 = \frac{E_1 - E_2}{E_1} = 1 - \frac{E_2}{E_1} = 1 - \frac{\sum_k \sum_{i=1}^n (y_i - \bar{y}_k)^2 \delta_{ik}}{\sum_{i=1}^n (y_i - \bar{y})^2} \quad (11.79)$$

Thus  $\eta^2$  is the reduction in error of  $y_i$  by knowing  $x_i$ . It goes from 0...1 and is unidirectional because nominal data are unordered.

#### Significance testing of $\eta^2$

The correlation is the greater the more different the means of the cardinal variable are in the different classes of the nominal. Thus the Null-hypothesis would be  $H_0$  : All classes

have indistinguishable means. FISHER's F-test can be used. It is based on the idea that the average of the sample variances  $s_j^2$  ( $j = 1..r$ ) is a first estimate  $\hat{s}_1^2$  of the variance in the population  $\sigma^2$ . If all samples were taken from the same population, the variance  $s_{\bar{Y}}^2$  of the  $r$  means should give a second estimate  $\hat{s}_2^2$  of the population variance. F is the ratio between these estimates:

$$\hat{s}_1^2 = \frac{\sum_{j=1}^r s_j^2}{r} \quad (11.80)$$

$$s_{\bar{Y}}^2 = \frac{\sum_{j=1}^r (\bar{x}_j - \bar{x})^2}{r-1} \quad (11.81)$$

$$\hat{s}_2^2 = n_j s_{\bar{Y}}^2 \quad (11.82)$$

$$F = \frac{\hat{s}_2^2}{\hat{s}_1^2} = \frac{\eta^2}{1-\eta^2} \times \frac{n-r}{r-1} \quad (11.83)$$

$$(11.84)$$

with  $n_j$  the size of the groups. The degree of freedom for the numerator is  $r-1$ , for the denominator  $n-r$ , with  $n$  the total sample size. The second part of the equation for F can be used even if the group sizes are not equal.

If this ratio is near 1.0, differences of group means probably represent sampling variation between samples taken from a single population. If F is large, this Null-hypothesis can be rejected, the samples probably come from different populations.

Listing 11.21: Nominal/Nominal association

```

1 FUNCTION eta_sqr(CONST NominalVector, CardinalVector: VectorTyp;
2   VAR Significance: SignificanceType): float;
3
4 VAR
5   i, k, n, Classes, Length, Value1: WORD;
6   E1, E2, total, r: float;
7   SumOfK: array [0..MaxSteps] of float;
8   NofK: ARRAY [0..MaxSteps] OF WORD;
9
10 BEGIN
11   IF NOT (TestDataVectorLength(NominalVector, CardinalVector)) THEN EXIT;
12   Length := VectorLength(NominalVector);
13   Classes := 0;
14   FOR i := 1 TO Length DO
15     BEGIN
16       IF IsNaN(GetVectorElement(NominalVector, i))
17         THEN
18           ELSE IF (round(GetVectorElement(NominalVector, i)) > Classes)
19             THEN Classes := ROUND(GetVectorElement(NominalVector, i));
20           // find largest value for nominal vector
21     END;

```

## 11. Correlation coefficients

```

22  FOR i := 0 TO Classes DO
23    BEGIN
24      NofK[i] := 0;
25      SumOfK[i] := 0;
26    END;
27    Total := 0;
28    n := 0;
29    FOR i := 1 TO Length DO
30      IF ((IsNaN(GetVectorElement(NominalVector, i))) OR
31          (IsNaN(GetVectorElement(CardinalVector, i))))
32      THEN // ignore NaNs
33      ELSE
34        BEGIN
35          Value1 := trunc(GetVectorElement(NominalVector, i));
36          Inc(NofK[Value1]);
37          // how many x are there for any of the 0..Classes values of x?
38          SumOfK[Value1] := SumOfK[Value1] + GetVectorElement(CardinalVector,
39                           i);
40        END;
41    FOR k := 0 TO Classes DO
42      BEGIN
43        n := n + NofK[k];                                // number of data pairs that
44        are not NaN
45        Total := Total + SumOfK[k];
46        IF (NofK[k] <> 0)
47        THEN SumOfK[k] := SumOfK[k] / NofK[k]; // group-averages
48      END;
49      Total := Total / n;                                // average over all data
50      E1 := 0;
51      E2 := 0;
52      FOR i := 1 TO Length DO
53        BEGIN
54          IF ((IsNaN(GetVectorElement(NominalVector, i))) OR
55              (IsNaN(GetVectorElement(CardinalVector, i))))
56          THEN
57          ELSE E1 := E1 + sqr(GetVectorElement(CardinalVector, i) - Total);
58        END;
59      FOR i := 1 TO Length DO
60        BEGIN
61          IF ((IsNaN(GetVectorElement(NominalVector, i))) OR
62              (IsNaN(GetVectorElement(CardinalVector, i))))
63          THEN
64          ELSE E2 := E2 + sqr(GetVectorElement(CardinalVector, i) -
65                               SumOfK[trunc(GetVectorElement(NominalVector, i))]);
66        END;
67      IF (E1 = 0)

```

```

66  THEN r := 0.0 // ???
67  ELSE r := 1 - E2 / E1;
68 IF (r = 1)
69  THEN
70  BEGIN
71      Significance.P0 := NaN;
72      Significance.Freedom := n - r;
73      Significance.TestValue := NaN;
74  END
75 ELSE
76  BEGIN
77      Significance.TestValue := r / (1 - r) * (n - r) / (r - 1);
78      // calculate F
79      Significance.Freedom := n - r;
80      // other is r-1 and can be calculated in calling program
81      Significance.P0 := Integral_F(Significance.TestValue, round(n - r),
82                                    round(r - 1));
83      // calculate probability for 0-hypotheses
84      Result := r;
85 END;
85END;

```

### 11.5.6. Latent correlation

Sometimes variables on an ordinal scale can be thought of as quantised representation of an interval-scaled, unobserved (latent) variable. In these cases the **tetrachoric** correlation (for binary variables) or the **polychoric** correlation coefficients can be used. These names refer to series expansions that were used before the availability of computers to calculate the correlation coefficient, this calculation method is obsolete so the term "latent" correlation is now preferred [24, 25]. Note that according to [9] the tetrachoric correlation coefficient (his  $A_{30}$ ) has undesirable statistical properties and should be replaced by McCONNAUGHEY's correlation.

For example, latent correlation is used to measure rater agreement (correlation of results if ratings were made on a interval rather than on a LIKERT [26] scale). The method can be used even to compare studies with different rating levels or raters who use different rating levels. Example: A disease may be diagnosed if a trait (a cardinal variable) exceeds a certain threshold, resulting in a dichotomous scale (disease present/absent). From the contingency table of the diagnostic results of two raters one can estimate the threshold of both raters ( $t_1, t_2$ ) and the radius of the ellipse that would form if both raters used a cardinal scale and their measurements were plotted against each other ( $\rho$  or tetrachoric correlation coefficient  $r_*$ ). The polychoric case is an extension of this concept to multiple rating levels, that is, multiple thresholds and more cells in the contingency table.

If a patient comes to the raters with a true latent trait level  $t$ , the raters will get the impression of latent trait level  $y_1, y_2$ , leading to the manifest rating  $x_1, x_2$ . Then the

model is:

$$y_1 = b_1 t + \varepsilon_1 \quad (11.85)$$

$$y_2 = b_2 t + \varepsilon_2 \quad (11.86)$$

with  $b_1, b_2$  regression coefficients and  $\varepsilon_1, \varepsilon_2$  error terms. If both  $t$  and the errors are normally distributed and independent between raters and cases, then  $y_1, y_2$  must also be normally distributed and  $b_1 = b_2 = b$ . We define  $r_* \equiv b^2$ . It is possible to extend this model to skewed latent traits or for more than two raters. The R-packages `polycor` and `psych` can calculate latent correlations.

## References

- [1] H. SIES: A new parameter for sex education, *Nature* **332** (1988), 495 DOI: [10.1038/332495a0](https://doi.org/10.1038/332495a0).
- [2] K. PEARSON: Notes on regression and inheritance in the case of two parents, *Proc. Royal Soc. Lond.* **58** (1895), 240–242 URL: <http://www.jstor.org/stable/115794>.
- [3] Z. GNIAZDOWSKI: Geometric interpretation of a correlation, *Zesz. Naukowed. Warsz. Wyższej Szk. Inform.* **9**:7 (2013), 27–35 URL: [http://zeszyty-naukowe.wwsi.edu.pl/zeszyty/zeszyt9/Geometric\\_interpretation\\_of\\_a\\_correlation.pdf](http://zeszyty-naukowe.wwsi.edu.pl/zeszyty/zeszyt9/Geometric_interpretation_of_a_correlation.pdf).
- [4] H.L. COSTNER: Criteria for Measures of Association, *Am. Sociol. Rev.* **30**:3 (1965), 341–353 DOI: [10.2307/2090715](https://doi.org/10.2307/2090715).
- [5] C. PERNET, R. WILCOX, G. ROUSSELET: Robust Correlation Analyses: False Positive and Power Validation Using a New Open Source Matlab Toolbox, *Front. Psychol.* **3** (2013), 606 DOI: [10.3389/fpsyg.2012.00606](https://doi.org/10.3389/fpsyg.2012.00606).
- [6] I. OLKIN, J.W. PRATT: Unbiased Estimation of Certain Correlation Coefficients, *Ann. Math. Stat.* **29**:1 (1958), 201–211 DOI: [10.2307/2237306](https://doi.org/10.2307/2237306) URL: <http://www.jstor.org/stable/2237306>.
- [7] C. SPEARMAN: The Proof and Measurement of Association between Two Things, *Am. J. Psychol.* **15**:1 (1904), 72–101 DOI: [10.2307/1412159](https://doi.org/10.2307/1412159).
- [8] L.C. FREEMAN: Order-based statistics and monotonicity: A family of ordinal measures of association, *J. Math. Sociol.* **12**:1 (1986), 49–69 URL: <http://moreno.ss.uci.edu/41.pdf>.
- [9] Z. HUBALEK: Coefficients of association and similarity, based on binary (presence-absence) data: an evaluation, *Biol. Rev.* **57**:4 (1982), 669–689 DOI: [10.1111/j.1469-185X.1982.tb00376.x](https://doi.org/10.1111/j.1469-185X.1982.tb00376.x).
- [10] P. JACCARD: Nouvelles recherches sur la distribution florale, *Bull. Soc. Vaud. Sci. Nat.* **44** (1908), 223–270.
- [11] P.F. RUSSELL, T.R. RAO: On habitat and association of species of anopheline larvae in south-eastern Madras. *J. Malar. Inst. India* **3**:1 (1940), 153–178.

- [12] B.H. MC CONNAUGHEY: The determination and analysis of plankton communities, *Penelit. Laut di Indones. [Marine Res. Indones.]* **7**:30 (1964), 1–40.
- [13] B.W. MATTHEWS: Comparison of the predicted and observed secondary structure of T4 phage lysozyme, *Biochim. Biophys. Acta* **405**:2 (1975), 442–451 DOI: [10.1016/0005-2795\(75\)90109-9](https://doi.org/10.1016/0005-2795(75)90109-9).
- [14] J. GORODKIN: Comparing two K-category assignments by a K-category correlation coefficient, *Comput. Biol. Chem.* **28**:5 (2004), 367–374 DOI: [10.1016/j.combiolchem.2004.09.006](https://doi.org/10.1016/j.combiolchem.2004.09.006).
- [15] L.A. GOODMAN, W.H. KRUSKAL: Measures of association for cross classifications, *J. Am. Stat. Assoc.* **49**:268 (1954), 732–764 URL: [http://www.nssl.noaa.gov/users/brooks/public\\_html/feda/papers/goodmankruskal1.pdf](http://www.nssl.noaa.gov/users/brooks/public_html/feda/papers/goodmankruskal1.pdf).
- [16] L.C. FREEMAN: *Elementary applied statistics* New York, London, Sidney: John Wiley & Sons, 1965.
- [17] J.A. HARRIS: On the Calculation of Intra-Class and Inter-Class Coefficients of Correlation from Class Moments when the Number of Possible Combinations is Large, *Biometrika* **9**:3/4 (1913), 446–472 DOI: [10.1093/biomet/9.3-4.446](https://doi.org/10.1093/biomet/9.3-4.446).
- [18] K. PEARSON: On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling, *Philos. Mag.* **50**:302 (1900), 157–175 DOI: [10.1080/14786440009463897](https://doi.org/10.1080/14786440009463897).
- [19] L. GUTTMAN: The quantification of a class of attributes: A theory and method of scale construction, In: *The prediction of personal adjustment* P. HORST, P. WALLIN, L. GUTTMAN et al. (editor) New York: Social Science Research Council, 1941, 319–348.
- [20] P. GINGRICH: *Association between variables* lecture script, accessed 2017-07-23 2006 URL: <http://uregina.ca/~gingrich/ch11a.pdf>.
- [21] L.C. FREEMAN: A further note on Freeman's measure of association, *Psychometrika* **41**:2 (1976), 273–275 URL: <http://moreno.ss.uci.edu/22.pdf>.
- [22] G. DEUCHLER: Über die Methoden der Korrelationsrechnung in der Pädagogik und Psychologie, *Z. pädagog. Psychol. Jugendkd.* **15**:3 (1914), 114–131 URL: [http://www.digizeitschriften.de/dms/resolveppn/?PID=ZDB026398621\\_0015%7CLOG\\_0024](http://www.digizeitschriften.de/dms/resolveppn/?PID=ZDB026398621_0015%7CLOG_0024).
- [23] ANONYMOUS: *Fehlerreduktionsmaße* Web-site, accessed 2017-07-29 2016 URL: <https://de.wikipedia.org/wiki/Fehlerreduktionsma%C3%9Fe#.CE.B72>.
- [24] J.S. UEBERSAX: *Introduction to the Tetrachoric and Polychoric Correlation Coefficients* Web-site, accessed 2017-07-14 2015 URL: <http://www.john-uebersax.com/stat/tetra.htm>.

- [25] U. OLSSON: Maximum likelihood estimation of the polychoric correlation coefficient, *Psychometrika* 44:4 (1979), 443–460 URL: [https://www.researchgate.net/profile/Ulf\\_Olsson/publication/24062390\\_Maximum\\_likelihood\\_estimation\\_of\\_the\\_polychoric\\_correlation\\_coefficient/links/53f30b5e0cf2dd48950c8372/Maximum-likelihood-estimation-of-the-polychoric-correlation-coefficient.pdf](https://www.researchgate.net/profile/Ulf_Olsson/publication/24062390_Maximum_likelihood_estimation_of_the_polychoric_correlation_coefficient/links/53f30b5e0cf2dd48950c8372/Maximum-likelihood-estimation-of-the-polychoric-correlation-coefficient.pdf).
- [26] R. LIKERT: *A Technique for the Measurement of Attitudes*, Published as Arch. Psychol. vol. 140 PhD thesis New York: Columbia University, 1932.

# 12. Linear and linearising Regression

## Abstract

In many cases, a dependent variable  $y$  can be described as linear function of an independent (predictor) variable  $x$ , using the criterion of minimal sum of squared residuals. Some important non-linear functions can be linearised by a suitable transformation of  $x, y$ ; even if this results in biased estimates for the parameters. Non-parametric methods should be used for regression if the conditions of normality, uncorrelatedness and homoscedasticity of the error are violated.

The interface for the unit is

Listing 12.1: Interface

```
1 UNIT LinearRegression;
2
3 INTERFACE
4
5 USES math, MathFunc, Vector, Stat, Correlations;
6
7 CONST RegressionError : BOOLEAN = FALSE;
8
9 TYPE CurveTyp = (Origin, Linear, Exponential, Power, Hyperbola, Inverse,
10   Maximum,
11     Sigmoidal, ExpSigmoidal, ModPower, Hill);
12   ResultTyp = RECORD
13     a, sa,                                // intercept
14     b, sb,                                // slope
15     m, sm,                                // exponent
16     r, t, P0 : double;                    // TEST statistics
17   END;
18
19 PROCEDURE Approximation(CONST x, y, weight : VectorTyp; ct : CurveTyp;
20   VAR yCalc : VectorTyp; VAR Res : ResultTyp);
21 // linear regression for least sum of squares
22
23 PROCEDURE Transformation(CONST xOrig, yOrig : VectorTyp; VAR xTrans, yTrans
24   : VectorTyp;
25     ct : CurveTyp; Schaetzwert : double);
26 // transformation of x, y TO linearise
```

## 12. Linear and linearising Regression

## 12.1. Linear regression

For further information, consult [1–4]. Given a pair of vectors  $\mathbf{x}, \mathbf{y}$ , with  $\mathbf{x}$  the independent (controlled, predictor) and  $\mathbf{y}$  the dependent (predicted) variable, it is possible to calculate a line  $\hat{\mathbf{y}} = f(\mathbf{x}) = a + b\mathbf{x} + \epsilon$  through these data pairs so that the sum of squared residuals

$\sum_{i=1}^n (y_i - \hat{y}_i)^2$  becomes minimal. Note that this is possible even when the data are not well represented by the model, in that case the fit will be poor, the squared residuals will be large and a plot of the residuals as function of  $x$  (or  $\hat{y}$ ) will show a non-random pattern.

Hence it is of utmost importance to *always* plot both  $y_i$  and  $\hat{y}_i - y_i$  as function of  $x_i$  and to visually check these plots for unexpected patterns.

The most common case of linear regression is data that are on a straight line, however, we speak of linear regression whenever the second and higher derivatives of the fitting function with respect to the parameters are all zero [5]. Then the Gauss-Newton algorithm will require only a single iteration for any initial “guesses” of the fitting parameter values, which therefore can all be set to zero.

### 12.1.1. Requirements for linear regression

The data  $x, y$  should meet a couple of requirements:

- All error is in the  $y$ -vector, the  $x$ -vector is error-free. If both dependent and independent variables have errors, one can try a DEMING regression [6].
- The error terms  $\epsilon$  should be **uncorrelated**, that is, knowledge of  $\epsilon_i$  should not allow the prediction of  $\epsilon_{i+1}$ . For example, time series data are prone to correlations of error terms, because drift affects data the more similarly, the closer they are in time. A plot of  $\hat{y}_i - y_i$  as function of time can reveal this, perhaps followed by a runs test. Correlation of error may also occur if some data points have factors in common that the study doesn't test for. For example, in a study of height vs weight of persons, such correlations can arise if some test subjects were raised under the same environmental influences (say, in siblings), or if repeat measurements were made on the same subject. In effect, such correlations reduce the confidence intervals of parameters, making the data appear more precise than they actually are.
- The error should be **normally distributed**.
- **Homoscedasticity** of the data means that  $y_i - \hat{y}_i$  is independent of  $x$ . Depending on experimental design, especially for  $y$ -values spanning several orders of magnitude,  $\hat{y}_i - y_i$  may increase with  $x$ , so that the relative error stays constant. In a plot of residuals, one would see a funnel. Linear regression of such data by the least-squares criterion leads to biased parameters with underestimated standard deviations. A better approach would be a fit by minimal  $\chi^2$ , for example by the simplex algorithm (see chapter 13 on page 425). Alternatively, one can perform a weighted linear regression.
- The data should be free of **outliers**, that is, data points with extremely large residuals. Such data are obvious in a plot of the residuals. Even better is to plot

## 12. Linear and linearising Regression

the *studentised residuals*, that is the residuals divided by their estimated standard error. Any data point with a studentised residual larger than 3 is suspect. If the outlier is the result of a measurement error, it should be removed. However, outliers may be the result of an incomplete model (variables that affect the result, but were not accounted for). Robust (non-parametric) fitting methods, for example the THEIL-SEN-KENDALL-estimator [7], can be used if there are many outliers.

- The distribution of data with respect to  $\mathbf{x}$  should be reasonably even. Single data points with very extreme  $\mathbf{x}$ -values may have a strong influence on the regression line, they have **high leverage**. Any problems with such data points will unduly affect the data fit. The leverage statistics for the  $i$ -th data point  $h_i = \frac{(\mathbf{x}_i - \bar{\mathbf{x}})^2}{\sum(\mathbf{x} - \bar{\mathbf{x}})^2}$  is a number  $0 \leq h_i \leq 1$ , and  $\bar{h} = \frac{p}{n}$ . Thus any observation where  $h_i \gg \bar{h}$  is suspect. In multiple regression,  $\mathbf{h} = \text{diag}(\mathcal{H}) = \text{diag}(\mathcal{X}(\mathcal{X}^T \mathcal{X})^{-1} \mathcal{X}^T)$  the diagonal of the hat-matrix.

### 12.1.2. Algorithm

#### Parameters

In matrix terms, the position of the minimum of the **RSS** is given by  $\hat{\beta} = (\mathcal{X}^T \mathcal{X})^{-1} \mathcal{X}^T \mathbf{y}$ , its value by  $\text{RSS}(\hat{\beta}) = \mathbf{y}^T \mathbf{y} - \hat{\beta}^T \mathcal{X}^T \mathcal{X} \hat{\beta}$  and the **RSS** in the vicinity of the minimum by  $\text{RSS}(\beta) = \text{RSS}(\hat{\beta}) + (\beta - \hat{\beta})^T \mathcal{X}^T \mathcal{X} (\beta - \hat{\beta})$  [8].

For a two-variable regression  $\hat{y}_i = a + b \mathbf{x}_i$ , the calculation is performed as follows: With  $S_{\mathbf{x}} = \sum_{i=1}^n \mathbf{x}_i$ ,  $S_{\mathbf{y}} = \sum_{i=1}^n \mathbf{y}_i$ ,  $S_{\mathbf{xx}} = \sum_{i=1}^n (\mathbf{x}_i^2)$ ,  $S_{\mathbf{yy}} = \sum_{i=1}^n (\mathbf{y}_i^2)$ ,  $S_{\mathbf{xy}} = \sum_{i=1}^n (\mathbf{x}_i \mathbf{y}_i)$ , the estimated slope of the regression line  $b = \hat{\beta}_1$  is

$$b = \frac{nS_{\mathbf{xy}} - S_{\mathbf{x}} S_{\mathbf{y}}}{nS_{\mathbf{xx}} - S_{\mathbf{x}}^2} = \frac{\text{Cov}_{\mathbf{x}, \mathbf{y}}}{\text{Var}_{\mathbf{x}}} \quad (12.1)$$

and the intercept  $a = \hat{\beta}_0$

$$a = \bar{\mathbf{y}} - b \bar{\mathbf{x}} \quad (12.2)$$

In other words, the regression line goes through the centroid of the data  $(\bar{\mathbf{x}}, \bar{\mathbf{y}})$  and is rotated until the sum of squared residuals is minimal.

A line can be forced through the origin, that is,  $a = 0$ . Then

$$b = \frac{S_{\mathbf{xy}}}{S_{\mathbf{xx}}} \quad (12.3)$$

Instead of  $(0,0)$ , the line can go through any point  $(h, k)$ :

$$b = \frac{\overline{(\mathbf{x} - h)(\mathbf{y} - k)}}{(\bar{\mathbf{x}} - h)^2} = \frac{\text{Cov}_{\mathbf{x}, \mathbf{y}} + (\bar{\mathbf{x}} - h)(\bar{\mathbf{y}} - k)}{\text{Var}_{\mathbf{x}} + (\bar{\mathbf{x}} - h)^2} \quad (12.4)$$

## Correlation coefficient

PEARSON's product moment correlation between  $\mathbf{x}$  and  $\mathbf{y}$  is

$$r = \frac{nS_{xy} - S_x S_y}{\sqrt{(nS_{xx} - S_x^2)(nS_{yy} - S_y^2)}} \quad (12.5)$$

The square of  $r$  is the **coefficient of determination**  $r^2$ , the fraction of variance in the  $\mathbf{y}$ -data that is explained by  $\mathbf{x}$ . If the  $\mathbf{y}$ -data were predicted without knowledge of  $\mathbf{x}$ , then one would use  $\bar{\mathbf{y}}$  as best predictor, and the total error would be  $E_1 = \sum_{i=1}^n (\mathbf{y}_i - \bar{\mathbf{y}})^2$ , the **total sum of squares (TSS)**. If  $\mathbf{x}$  is used for prediction, the new total error becomes  $E_2 = \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2$ , the **residual sum of squares (RSS)**. Then  $r^2 = \frac{E_1 - E_2}{E_1} = 1 - \frac{E_2}{E_1}$ , the **proportionate reduction of error (PRE)** from knowledge of  $\mathbf{x}$  [9]. The **explained sum of squares (ESS)** =  $\sum_{i=1}^n (\hat{\mathbf{y}}_i - \bar{\mathbf{y}})^2 = TSS - RSS$ .

If one calculates the regression line between  $\mathbf{x}$  and  $\mathbf{y}$  with the parameters  $a_x, b_x$  and the regression line between  $\mathbf{y}$  and  $\mathbf{x}$  with the parameters  $a_y, b_y$  (see fig. 12.1), then these lines form an angle  $\alpha$  – recall that both lines intersect at  $(\bar{\mathbf{x}}, \bar{\mathbf{y}})$ . This angle is proportional to  $r$ , it is 0 for  $r = 1$  and 90 for  $r = 0$ . The correlation coefficient is  $r = \sqrt{b_x b_y}$ .

Yet another alternative interpretation of  $r$  is the slope of the regression line through the  $z$ -standardised data points. This line passes through the origin, because the arithmetic mean of  $z$ -standardised data is zero.

It is possible to test the regression for significance, that is, to test the 0-hypothesis  $H_0 : r = 0$  against the alternative hypothesis  $H_1 : r \neq 0$ . For this, a  $t$ -value is calculated according to eqn. 11.7, then the degrees of freedom  $v = n - q$  is the number of measured values  $n$  minus the number of parameters used  $q$ .

In simple regression,  $R^2 = r^2$ , but in multiple regression,  $R^2 = \text{Cor}(\mathbf{y}, \hat{\mathbf{y}}) \neq r^2$ . The algorithm then maximises  $R^2$ .

It is sometimes necessary to decide between different models, with different number of parameters  $p$ . Because

$$R^2 = 1 - \frac{\sum (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2}{\sum (\mathbf{y}_i - \bar{\mathbf{y}})^2} \quad (12.6)$$

,  $R^2$  will always increase with  $p$ . This results from a reduction in training error, which may not correspond to a reduction in test error (overfitting). Thus, to compare models with different  $p$ , we use the **adjusted R<sup>2</sup>**

$$R_a^2 = 1 - \frac{\sum (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 / (n - p - 1)}{\sum (\mathbf{y}_i - \bar{\mathbf{y}})^2 / (n - 1)} \quad (12.7)$$

which may increase or decrease as additional parameters are added. Irrelevant parameters will decrease the sum of squared residuals only marginally, this is over-compensated by the increase of  $p$  in the denominator. Other values used to identify the best number of

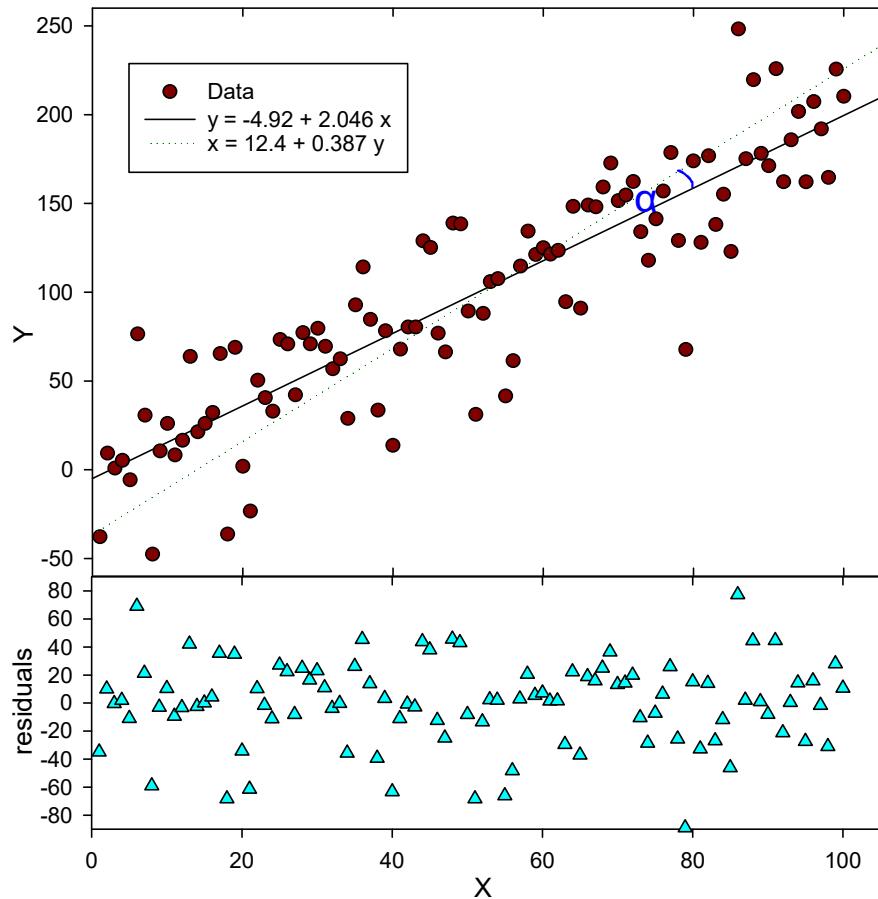


Figure 12.1.: *Top:* Regression analysis for synthetic data (100 data points,  $y = 1 + 2x$  with random noise added). The regression of  $y$  from  $x$  yields a different line from the regression of  $x$  from  $y$ . Both lines go through  $(\bar{x}, \bar{y})$ , their angle  $\alpha$  is proportional to the correlation coefficient  $r$ . Note that the noise affects the estimate for the intercept much more than that for the slope. *Bottom:* The residuals are randomly distributed around their average and show no discernible pattern.

parameters are

$$C_p = \frac{1}{n} \left( \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 + 2p\hat{\sigma}^2 \right) \quad (12.8)$$

$$\text{AIC} = \frac{1}{n\hat{\sigma}^2} \left( \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 + 2p\hat{\sigma}^2 \right) \quad (12.9)$$

$$\text{BIC} = \frac{1}{n} \left( \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 + \log(n)p\hat{\sigma}^2 \right) \quad (12.10)$$

, which are closely related to each other. Whilst we are looking for a maximum of  $R^2$ , we look for the minimum of  $C_p$ , [BIC](#) and [AIC](#).

### Error estimates for the parameters $a, b$

The variance of the residuals is

$$s_\epsilon^2 = \frac{1}{n(n-2)} [nS_{yy} - S_y^2 - b^2(nS_{xx} - S_x^2)] \quad (12.11)$$

, the variance of  $b$

$$s_b^2 = \frac{ns_\epsilon^2}{nS_{xx} - S_x^2} \quad (12.12)$$

and the variance of  $a$

$$s_a^2 = \frac{1}{n} s_b^2 S_{xx} \quad (12.13)$$

The degrees of freedom is  $v = n - q$  with  $q$  the number of parameters estimated, here 2. Given the 0.975-quantile of the  $t$ -distribution with  $v$  degrees of freedom  $t_v^*$ , we can calculate the 97.5 % confidence intervals for the parameters to

$$a \pm t_v^* \sqrt{s_a^2} \quad b \pm t_v^* \sqrt{s_b^2} \quad (12.14)$$

### Tests for $H_0 : r = 0$ , $H_0 : b = b_0$ and $H_0 : a = a_0$

It is sometimes necessary to decide, whether there actually is a significant relationship between the dependent and independent variable, that is, we test  $H_0 : r = 0$  against  $H_1 : r \neq 0$ . For this we can either do an F-test:

$$\begin{aligned} F &= \frac{r^2}{1-r^2} \\ v_1 &= 1 \\ v_2 &= n-q \end{aligned} \quad (12.15)$$

## 12. Linear and linearising Regression

or a  $t$ -test:

$$\begin{aligned} t &= \sqrt{n-q} \frac{r^2}{\sqrt{1-r^2}} \\ v &= n-q \end{aligned} \quad (12.16)$$

Similarly, it may be interesting to test whether the  $y$ -intercept  $a$  is significantly different from any  $a_0$ . The most common application is  $a_0 = 0$ , *i.e.*, are we looking at a line through the origin?

$$\begin{aligned} t &= \frac{|a - a_0|}{s_a} \\ v &= n-q \end{aligned} \quad (12.17)$$

Lastly, one might wish to compare the slope  $b$  with some value  $b_0$ :

$$\begin{aligned} t &= \frac{|b - b_0|}{s_b} \\ v &= n-q \end{aligned} \quad (12.18)$$

Again, the most common case is the test for  $b = 0$ .

Listing 12.2: Linear regression by RSS

```

1 PROCEDURE Approximation(CONST x, y, weight : VectorTyp; ct : CurveTyp;
2   VAR yCalc : VectorTyp; VAR Res : ResultTyp);
3
4 VAR Sx, Sy, Sw, Sxx, Syy, Sxy, SDxDy, SDx2, SDy2, xMean, yMean : double;
5   C : CHAR;
6   n : WORD;
7
8 PROCEDURE DataSums;
9
10 VAR
11   i: INTEGER;
12   DX, dy, xi, yi, wi : double;
13
14 BEGIN
15   Sx := 0;
16   Sy := 0;
17   Sw := 0;
18   Sxy := 0;
19   Sxx := 0;
20   Syy := 0;
21   SDxDy := 0;
22   SDx2 := 0;
23   SDy2 := 0;
```

```

24   FOR i := 1 TO VectorLength(x) DO
25     IF IsNaN(GetVectorElement(x, i)) OR IsNaN(GetVectorElement(y, i))
26       THEN
27       ELSE
28         BEGIN
29           xi := GetVectorElement(x, i);
30           yi := GetVectorElement(y, i);
31           wi := GetVectorElement(weight, i);
32           Sx := Sx + xi * wi;
33           Sy := Sy + yi * wi;
34           Sw := Sw + wi;
35           Sxy := Sxy + xi * yi * wi;
36           Sxx := Sxx + xi * xi * wi;
37           Syy := Syy + yi * yi * wi;
38         END;
39         xMean := Sx / Sw;
40         yMean := Sy / Sw;
41   FOR i := 1 TO VectorLength(x) DO
42     IF IsNaN(GetVectorElement(x, i)) OR IsNaN(GetVectorElement(y, i))
43       THEN
44       ELSE
45         BEGIN
46           DX := GetVectorElement(x, i) - xMean;
47           dy := GetVectorElement(y, i) - yMean;
48           wi := GetVectorElement(Weight, i);
49           SDxDy := SDxDy + DX * dy * wi;
50           SDx2 := SDx2 + DX * DX * wi;
51           SDy2 := SDy2 + dy * dy * wi;
52         END;
53         SDxDy := SDxDy / Sw;
54         SDx2 := SDx2 / Sw;
55         SDy2 := SDy2 / Sw;
56   END;

57
58
59 PROCEDURE Parameters;
60
61 VAR i: INTEGER;
62   a1, a2, a3, a4 : double;
63
64 BEGIN
65   IF (ct = Origin)
66     THEN
67       BEGIN // Line through origin
68         Res.b := Sxy / Sxx;
69         Res.a := 0;

```

## 12. Linear and linearising Regression

```

70      END;
71  ELSE
72    BEGIN // all others
73      a1 := Sw * Sxy;
74      a2 := Sx * Sy;
75      a3 := Sw * Sxx;
76      a4 := Sx * Sx;
77      a4 := (a1-a2) / (a3-a4);
78      Res.b := (Sw * Sxy - Sx * Sy) / (Sw * Sxx - Sx * Sx);
79      Res.a := yMean - Res.b * xMean;
80    END;
81  FOR i := 1 TO VectorLength(x) DO
82    IF IsNaN(GetVectorElement(x, i))
83      THEN SetVectorElement(yCalc, i, NaN)
84      ELSE SetVectorElement(yCalc, i, Res.a + Res.b * GetVectorElement(x,
85        i));
86
87
88  PROCEDURE ErrorEstimates;
89
90  VAR i, f : INTEGER;
91      Se : double;
92
93  BEGIN
94    CASE ct OF
95      Origin           : f := Round(n - 1); {degrees of freedom = data
96      points - parameters}
97      Linear..Maximum : f := Round(n - 2);
98      Sigmoidal..Hill  : f := Round(n - 3);
99    END;
100   WITH Res DO
101     BEGIN
102       se := 1/(Sw * f) * (Sw*Syy - Sy*Sy - Res.b*Res.b*(Sw*Sxx - Sx*Sx));
103       sb := Sw*se / (Sw*Sxx - Sx*Sx);
104       IF (ct = Origin)
105         THEN sa := 0
106         ELSE sa := (1/Sw) * sb * Sxx;
107       se := Sqrt(se);
108       sa := Sqrt(Res.sa);
109       sb := Sqrt(Res.sb);
110       r := SDxDy / Sqrt(SDx2 * SDy2);
111       IF (r < 1) // prevent division by 0
112         THEN
113           BEGIN
114             t := r * Sqrt(f/(1-r*r));

```

```

114      P0 := Integral_t(t, f);
115      END;
116      ELSE
117          BEGIN
118              t := NaN;
119              P0 := 0;
120          END;
121      END;
122  END;

124  BEGIN
125      IF VectorLength(x) <> VectorLength(y)
126      THEN
127          BEGIN
128              c := WriteErrorMessage('Linear regression: unequal length of
129                  dependent and independent data vector');
130              RegressionError := TRUE;
131              EXIT;
132          END;
133      n := VectorLength(x);
134      CreateVector(yCalc, VectorLength(x), 0.0);
135      DataSums;
136      Parameters;
137      ErrorEstimates;
138  END;

```

## 12.2. Weighted regression

Sometimes, the data points have uneven standard deviation, that is, they are not homoscedastic. In that case, we can weight the data points with the reciprocal of their

## 12. Linear and linearising Regression

standard deviation. Then

$$\begin{aligned}
 \bar{x} &= \frac{\sum(w_i x_i)}{\sum(w_i)} \\
 \bar{y} &= \frac{\sum(w_i y_i)}{\sum(w_i)} \\
 b &= \frac{\sum(w_i(x_i - \bar{x})(y_i - \bar{y}))}{\sum(w_i(x_i - \bar{x})^2)} = \frac{\sum(w_i x_i y_i) - \sum(w_i x_i) \sum(w_i y_i) / \sum(w_i)}{\sum(w_i x_i^2) - (\sum(w_i x_i))^2 / \sum w_i} \\
 a &= \bar{y} - b \bar{x} \\
 s_y^2 &= \frac{\sum(w_i(y_i - \hat{y}_i))}{n - q} = \frac{\sum(w_i y_i^2) - \bar{y} \sum(w_i y_i) - b \sum(w_i(x_i - \bar{x})(y_i - \bar{y}))}{n - q} \\
 s_b^2 &= \frac{s_y^2}{\sum(w_i(x_i - \bar{x})^2)} \\
 s_a^2 &= s_y^2 \left[ \frac{1}{\sum(w_i)} + \frac{\bar{x}^2}{\sum(w_i(x_i - \bar{x})^2)} \right]
 \end{aligned} \tag{12.19}$$

In case the line is supposed to go through the origin,  $a = 0$ , replace all  $x_i$  with  $(x_i - \bar{x})$  and  $s_y^2 = \frac{\sum(w_i y_i^2) - b \sum(w_i(x_i - \bar{x})(y_i - \bar{y}))}{n - 1}$ .

### 12.3. Robust linear regression

Sometimes we want to calculate a regression line when the data are not homoscedastic. The following procedure [10, pp. 590-8] allows us to do this. The data have three columns:  $x, y$  and the standard deviation of  $y$ . If the latter is not available, the variable `mwt` must be set to `FALSE`.

Listing 12.3: Robust linear regression

```

1 PROCEDURE LinFit (Data : MatrixTyp; mwt: BOOLEAN; VAR a, b, siga, sigb,
      chi2, q: real);
2
3 VAR i, nData           : INTEGER;
4   wt, t, sy, sxoss,
5   sx, st2, ss, sigdat : float;
6
7 BEGIN
8   sx := 0.0;
9   sy := 0.0;
10  st2 := 0.0;
11  b := 0.0;
12  nData := MatrixRows(Data);
13  IF mwt
14    THEN
15      BEGIN

```

```

16    ss := 0.0;
17    FOR i := 1 TO ndata DO
18        BEGIN
19            wt := 1.0 / Sqr(GetMatrixElement(Data, i, 3));
20            ss := ss + wt;
21            sx := sx + GetMatrixElement(Data, i, 1) * wt;
22            sy := sy + GetMatrixElement(Data, i, 2) * wt
23        END
24    END
25    ELSE
26        BEGIN
27            FOR i := 1 TO ndata DO
28                BEGIN
29                    sx := sx + GetMatrixElement(Data, i, 1);
30                    sy := sy + GetMatrixElement(Data, i, 2);
31                END;
32            ss := ndata
33        END;
34    sxoss := sx/ss;
35    IF mwt
36    THEN
37        BEGIN
38            FOR i := 1 TO ndata DO
39                BEGIN
40                    t := (GetMatrixElement(Data, i, 1) -sxoss) /
41                        GetMatrixElement(Data, i, 3);
42                    st2 := st2+t*t;
43                    b := b + t * GetMatrixElement(Data, i, 2) /
44                        GetMatrixElement(Data, i, 3);
45                END
46        END
47    ELSE
48        BEGIN
49            FOR i := 1 TO ndata DO
50                BEGIN
51                    t := GetMatrixElement(Data, i, 1) - sxoss;
52                    st2 := st2 + t * t;
53                    b := b + t * GetMatrixElement(Data, i, 2);
54                END
55            END;
56        b := b / st2;
57        a := (sy - sx * b) / ss;
58        siga := Sqrt((1.0 + sx * sx / (ss * st2)) / ss);
59        sigb := Sqrt(1.0 / st2);
60        chi2 := 0.0;
61    IF NOT mwt

```

## 12. Linear and linearising Regression

```

60   THEN
61   BEGIN
62     FOR i := 1 TO ndata DO
63       chi2 := chi2 + Sqr(GetMatrixElement(Data, i, 2) - a - b *
64                     GetMatrixElement(Data, i, 1));
65     q := 1.0;
66     sigdat := Sqrt(chi2 / (nData - 2));
67     sigma := sigma * sigdat;
68     sigb := sigb * sigdat;
69   END
70 ELSE
71 BEGIN
72   FOR i := 1 TO ndata DO
73     chi2 := chi2 + Sqr((GetMatrixElement(Data, i, 2) - a - b *
74                         GetMatrixElement(Data, i, 1)) / GetMatrixElement(Data, i,
75                         3));
76 q := IntegralChi(chi2, nData-2);
77 END;
78END;

```

### 12.4. Linearising regression

Before computers became generally available, which allowed curve fitting by iterative methods, regression to suitably transformed data was generally used for curve fitting. The biggest disadvantage of these methods was that any method that transforms  $y$ , implicitly also transforms its error distribution. As a consequence, the requirements of homoscedasticity (see fig. 12.2) and of normality (see fig. 12.3) are violated, leading to biased estimates of parameters and, more importantly, their standard deviation. Note that this applies only to transformations of the dependent variable, as the independent variable is assumed to be error free its error distribution cannot be affected by transformation. Plots of transformed data still have a place in science for presentation purposes, as the human eye is very sensitive to deviations from a straight line. However, regression analysis of such data is largely obsolete.

Iterative regression methods require starting estimates of the parameters that should be reasonably close to the final value, firstly to reduce the number of iterations required and hence computer time. Secondly, iterative methods can become trapped in local minima of the error function, and then deliver suboptimal results. In my experience, the LEVENBERG-MARQUARDT-algorithm [11, 12] is more sensitive to this than NELDER-MEAD's simplex algorithm [13–15]. Although the parameter estimates from linearising regression are biased, they are certainly good enough as starting values for iterative methods.

Listing 12.4: Linearisation of curved data

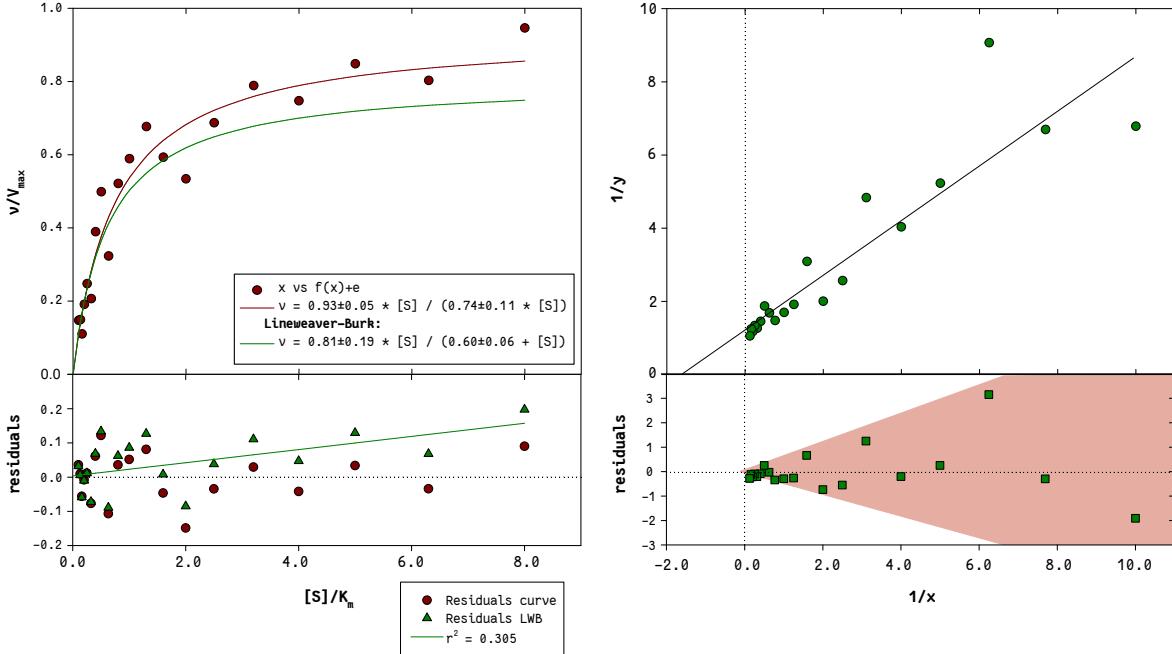


Figure 12.2.: Linearising regression. *Left, red:* Artificial data following a hyperbolic curve  $\hat{y} = \frac{1.0 \times x}{1.0 + x}$  with added noise. The residuals are random, without discernible pattern. *Right:* The same data, linearised by taking reciprocals of both  $x$  and  $y$ . The residuals increase with  $1/x$  (*pink shade*), they are no longer homoscedastic. As a result, the residuals of the re-transformed data increase with  $x$  (*lower left, green triangles and line*). The resulting hyperbolic curve (*upper left, green line*) poorly represents the data at high  $x$ .

## 12. Linear and linearising Regression

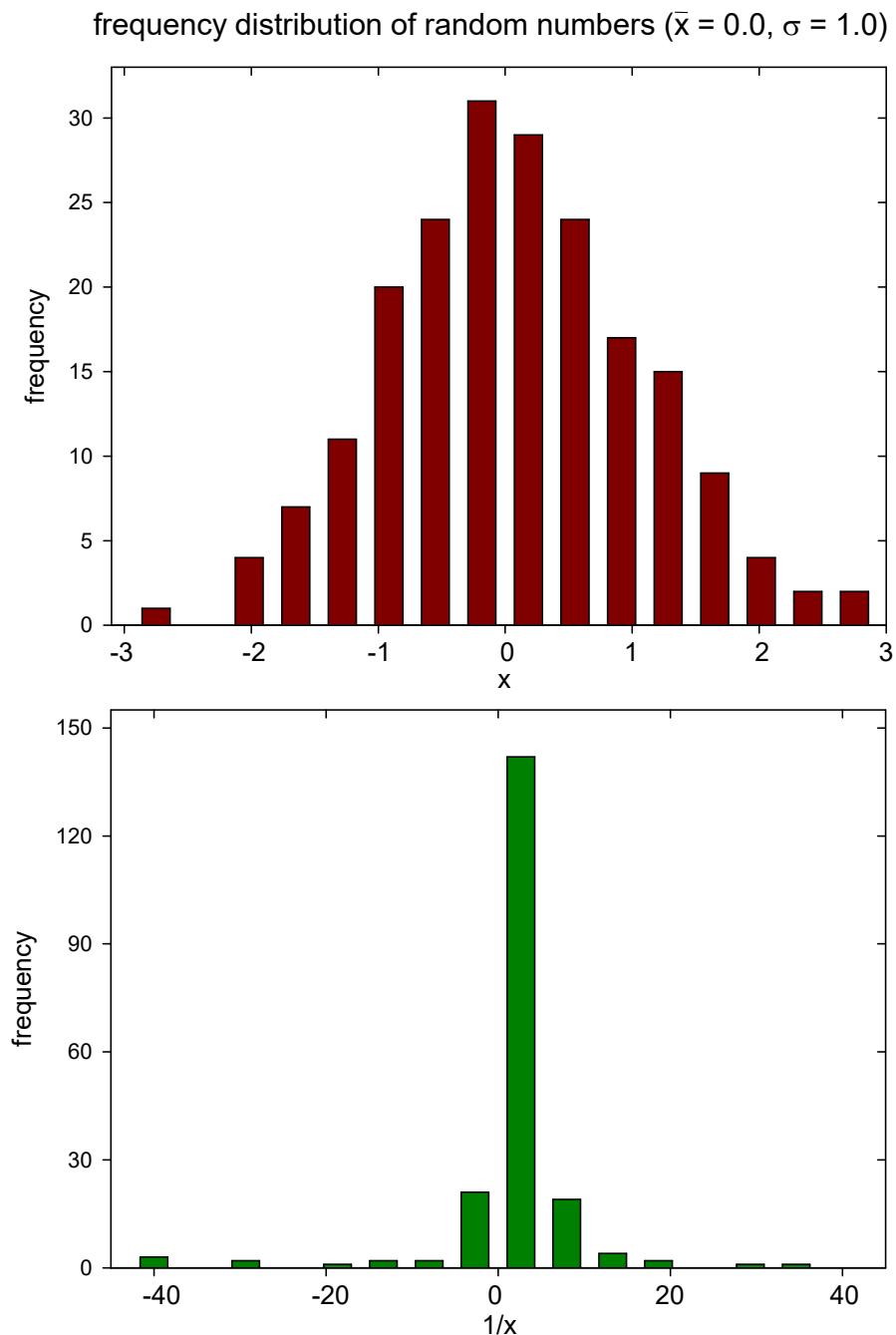


Figure 12.3.: *Top:* Distribution of 200 normally distributed random figures. *Bottom:* The distribution of the same data after taking reciprocals. The reciprocal of a GAUSSIAN is not GAUSSIAN!

Table 12.1.: Equations used in linearising regression. For the top equations, regression can happen directly, the bottom equations require an estimated parameter, either the  $y_{\max}$  or  $y_{x=0}$ , respectively. The weights can be used to counteract the effect of the transformation on the error of the y-value.

No	Name	Equation	x-transform	y-transform	weight	$a =$	$b =$
0	Line through origin	$y = b * x$	$x^* = x$	$y^* = y$	1	$a^*$	$b^*$
1	linear	$y = a + b * x$	$x^* = x$	$y^* = y$	1	$a^*$	$b^*$
2	exponential	$y = a * e^{m*x}$	$x^* = x$	$y^* = \ln(y)$	$y^2$	$\exp(a^*)$	
3	power	$y = a * x^m$	$x^* = \ln(x)$	$y^* = \ln(y)$	$y^2$	$\exp(a^*)$	
4	hyperbolic	$y = (a * x)/(b + x)$	$x^* = 1/x$	$y^* = 1/y$	$y^4$	$1/a^*$	$b^* a$
5	modif.invers	$y = a/(b + x)$	$x^* = x$	$y^* = 1/y$	$y^4$	$1/b^*$	$a^* a$
6	maximum	$y = a * x * e^{m*x}$	$x^* = x$	$y^* = \ln(x/y)$	$(x * y)^2 / (y - x)^2$	$\exp(-a^*)$	
7	sigmoidal	$y = a/(1 + b * x^m)$	$x^* = \ln(x)$	$y^* = \ln(y_{\max}/y - 1)$	$(y_{\max}/y - 1)^2 / (y_{\max}/y^2)^2$	$y_{\max}$	$\exp(a)$
8	exp.sigmoidal	$y = a/(1 + b * \exp(m * x))$	$x^* = x$	$y^* = \ln(y_{\max}/y - 1)$	$(y_{\max}/y - 1)^2 / (y_{\max}/y^2)^2$	$y_{\max}$	$\exp(a)$
9	mod. power	$y = a * x^m + b$	$x^* = \ln(x)$	$y^* = \ln(y - y_{x=0})$	$(y - y_{x=0})^2$	$y_{x=0}$	$\exp(a)$
10	Hill	$y = a * x^m / (b + x^m)$	$x^* = \log_{10}(x)$	$y^* = \log_{10}(y / (y_{\max} - y))$	$y^2 / \log_{10}(e * (y_{\max}^2 - y))^2$	$y_{\max}$	$10^{-a}$

12.4. Linearising regression

## 12. Linear and linearising Regression

```

1  PROCEDURE Transformation(CONST xOrig, yOrig : VectorTyp; VAR xTrans, yTrans
2   : VectorTyp;
3   ct : CurveTyp; Schatzwert : double);
4   // transformation OF x, y TO linearise
5
6   VAR i, n : INTEGER;
7
8   BEGIN
9     n := VectorLength(xOrig);
10    CreateVector(xTrans, n, 0.0);
11    CreateVector(yTrans, n, 0.0);
12    FOR i := 1 TO n DO
13      IF IsNaN(GetVectorelement(xOrig, i))
14      THEN
15        SetVectorElement(xTrans, i, NaN)
16      ELSE
17        CASE ct OF
18          Origin, Linear, Exponential, Inverse, Maximum, ExpSigmoidal :
19            SetVectorElement(xTrans, i, GetVectorElement(xOrig, i));
20          Power, Sigmoidal, ModPower :
21            SetVectorElement(xTrans, i, Ln(GetVectorElement(xOrig, i)));
22          Hyperbola :
23            SetVectorElement(xTrans, i, 1 / GetVectorElement(xOrig, i));
24          Hill :
25            SetVectorElement(xTrans, i, log(GetVectorElement(xOrig, i),
26            10));
27        END;
28    FOR i := 1 TO n DO
29      IF IsNaN(GetVectorelement(yOrig, i))
30      THEN
31        SetVectorElement(yTrans, i, NaN)
32      ELSE
33        CASE ct OF
34          Origin, Linear : SetVectorElement(yTrans, i,
35            GetVectorElement(yOrig, i));
36          Exponential, Power : SetVectorElement(yTrans, i,
37            Ln(GetVectorElement(yOrig, i)));
38          Hyperbola, Inverse : SetVectorElement(yTrans, i, 1 /
39            GetVectorElement(yOrig, i));
40          Maximum : IF IsNaN(GetVectorelement(xOrig, i))
41            THEN SetVectorElement(yTrans, i,
42              NaN)
43            ELSE SetVectorElement(yTrans, i,
44              Ln(GetVectorElement(xOrig, i) /
45              GetVectorElement(yOrig, i)));
46          Sigmoidal, ExpSigmoidal : SetVectorElement(yTrans, i,
47

```

```

Ln(Schaetzwert / GetVectorElement(yOrig, i) - 1));
39 ModPower           : SetVectorElement(yTrans, i,
40             Ln(GetVectorElement(yOrig, i) - Schaetzwert));
41 Hill               : SetVectorElement(yTrans, i,
42             log(GetVectorElement(yOrig, i) / (Schaetzwert -
43             GetVectorElement(yOrig, i)), 10));
44
45 END;
46
47
48
49 PROCEDURE Retransform (CONST x, yTrans : Vectortyp; VAR TRes, Res :
50                         ResultTyp;
51                         ct : CurveTyp; Schaetzwert : double;
52                         VAR yCalc : VectorTyp);
53
54 VAR i, n : WORD;
55
56 BEGIN
57   n := VectorLength(x);
58   CreateVector(yCalc, n, 0.0);
59   CASE ct OF
60     Origin, Linear : BEGIN
61       CopyVector(yTrans, yCalc);
62       Res := TRes;
63       Res.m := 0;
64       Res.sm := 0;
65     END;
66     Exponential : BEGIN
67       Res.a := Exp(TRes.a);
68       Res.sa := TRes.sa/Tres.a * Res.a;
69       Res.b := 0;
70       Res.sb := 0;
71       Res.m := TRes.b;
72       Res.sm := TRes.sb;
73       CreateVector(yCalc, VectorLength(yTrans), 0.0);
74       FOR i := 1 TO n DO
75         BEGIN
76           IF IsNaN(GetVectorElement(yTrans, i))
77           THEN SetVectorElement(yCalc, i, Nan)
78           ELSE SetVectorElement(yCalc, i, Res.a *
79                           Exp(Res.m * GetVectorElement(x, i)));
80         END;
81     END;
82     Power : BEGIN
83       Res.a := Exp(TRes.a);
84       Res.sa := TRes.sa/Tres.a * Res.a;

```

## 12. Linear and linearising Regression

```

80                     Res.b := 0;
81                     Res.sb := 0;
82                     Res.m := TRes.b;
83                     Res.sm := TRes.sb;
84                     CreateVector(yCalc, VectorLength(yTrans), 0.0);
85                     FOR i := 1 TO n DO
86                         BEGIN
87                             IF IsNaN(GetVectorElement(yTrans, i))
88                                 THEN SetVectorElement(yCalc, i, NaN)
89                             ELSE SetVectorElement(yCalc, i, Res.a *
90                                     Pot(GetVectorElement(x, i), Res.m));
91                         END;
92                     END;
93             Hyperbola : BEGIN
94                 Res.a := 1/TRes.a;
95                 Res.sa := TRes.sa/Tres.a * Res.a;
96                 Res.b := Tres.b * Res.a;
97                 Res.sb := TRes.sb/Tres.b * Res.b;
98                 Res.m := 0;
99                 Res.sm := 0;
100                CreateVector(yCalc, VectorLength(yTrans), 0.0);
101               FOR i := 1 TO n DO
102                   BEGIN
103                       IF IsNaN(GetVectorElement(yTrans, i))
104                           THEN SetVectorElement(yCalc, i, NaN)
105                           ELSE SetVectorElement(yCalc, i, Res.a *
106                               GetVectorElement(x, i)
107                               / (Res.b + GetVectorElement(x, i)));
108                   END;
109             Inverse : BEGIN
110                 Res.a := 1/TRes.b;
111                 Res.sa := TRes.sb/Tres.b * Res.a;
112                 Res.b := Tres.a * Res.a;
113                 Res.sb := TRes.sa/Tres.a * Res.b;
114                 Res.m := 0;
115                 Res.sm := 0;
116                 CreateVector(yCalc, VectorLength(yTrans), 0.0);
117                 FOR i := 1 TO n DO
118                     BEGIN
119                         IF IsNaN(GetVectorElement(yTrans, i))
120                             THEN SetVectorElement(yCalc, i, NaN)
121                             ELSE SetVectorElement(yCalc, i, Res.a /
122                                 (Res.b + GetVectorElement(x, i)));
123                     END;
124             END;

```

```

125 Maximum      : BEGIN
126   Res.a := Exp(-TRes.a);
127   Res.sa := Abs(TRes.sa/Tres.a * Res.a);
128   Res.b := 0;
129   Res.sb := 0;
130   Res.m := -TRes.b;
131   Res.sm := Abs(TRes.sb/Tres.b * Res.m);
132   CreateVector(yCalc, VectorLength(yTrans), 0.0);
133   FOR i := 1 TO n DO
134     BEGIN
135       IF IsNaN(GetVectorElement(yTrans, i))
136         THEN SetVectorElement(yCalc, i, NaN)
137         ELSE SetVectorElement(yCalc, i, Res.a *
138           GetVectorElement(x, i)
139           * Exp(Res.m * GetVectorElement(x, i)));
140     END;
141   END;
142 Sigmoidal    : BEGIN
143   Res.a := Schaetzwert;
144   Res.sa := NaN;
145   Res.b := Exp(TRes.a);
146   Res.sb := TRes.sa/Tres.a * Res.a;
147   Res.m := TRes.b;
148   Res.sm := TRes.sb/Tres.b * Res.m;
149   CreateVector(yCalc, VectorLength(yTrans), 0.0);
150   FOR i := 1 TO n DO
151     BEGIN
152       IF IsNaN(GetVectorElement(yTrans, i))
153         THEN SetVectorElement(yCalc, i, NaN)
154         ELSE SetVectorElement(yCalc, i, Res.a /
155           (1 + Res.b * Pow(GetVectorElement(x,
156           i), Res.m)));
157     END;
158   END;
159 ExpSigmoidal : BEGIN
160   Res.a := Schaetzwert;
161   Res.sa := NaN;
162   Res.b := Exp(TRes.a);
163   Res.sb := TRes.sa/Tres.a * Res.a;
164   Res.m := TRes.b;
165   Res.sm := TRes.sb/Tres.b * Res.m;
166   CreateVector(yCalc, VectorLength(yTrans), 0.0);
167   FOR i := 1 TO n DO
168     BEGIN
169       IF IsNaN(GetVectorElement(yTrans, i))
170         THEN SetVectorElement(yCalc, i, NaN)

```

## 12. Linear and linearising Regression

```

169                               ELSE SetVectorElement(yCalc, i, Res.a /
170                                         (1 + Res.b * Exp(GetVectorElement(x, i)
171                                         * Res.m)));
171
172 END;
173 ModPower : BEGIN
174     Res.a := Schaetzwert;
175     Res.sa := NaN;
176     Res.b := Exp(TRes.a);
177     Res.sb := TRes.sa/Tres.a * Res.a;
178     Res.m := TRes.b;
179     Res.sm := TRes.sb/Tres.b * Res.m;
180     CreateVector(yCalc, VectorLength(yTrans), 0.0);
181     FOR i := 1 TO n DO
182         BEGIN
183             IF IsNaN(GetVectorElement(yTrans, i))
184                 THEN SetVectorElement(yCalc, i, NaN)
185                 ELSE SetVectorElement(yCalc, i, Res.a + Res.b *
186                                         Pot(GetVectorElement(x, i), Res.m));
187         END;
188     END;
189 Hill : BEGIN
190     Res.a := Schaetzwert;
191     Res.sa := NaN;
192     Res.b := pot(10, -TRes.a);
193     Res.sb := Abs(TRes.sa/Tres.a * Res.a);
194     Res.m := TRes.b;
195     Res.sm := TRes.sb/Tres.b * Res.m;
196     CreateVector(yCalc, VectorLength(yTrans), 0.0);
197     FOR i := 1 TO n DO
198         BEGIN
199             IF IsNaN(GetVectorElement(yTrans, i))
200                 THEN SetVectorElement(yCalc, i, NaN)
201                 ELSE SetVectorElement(yCalc, i, Res.a *
202                                         pot(GetVectorElement(x, i), Res.m)
203                                         / (Res.b + Pot(GetVectorElement(x, i),
204                                             Res.m)));
204         END;
205     END; // CASE
206     Res.r := TRes.r;
207     Res.t := TRes.t;
208     Res.P0 := TRes.P0;
209 END;

```

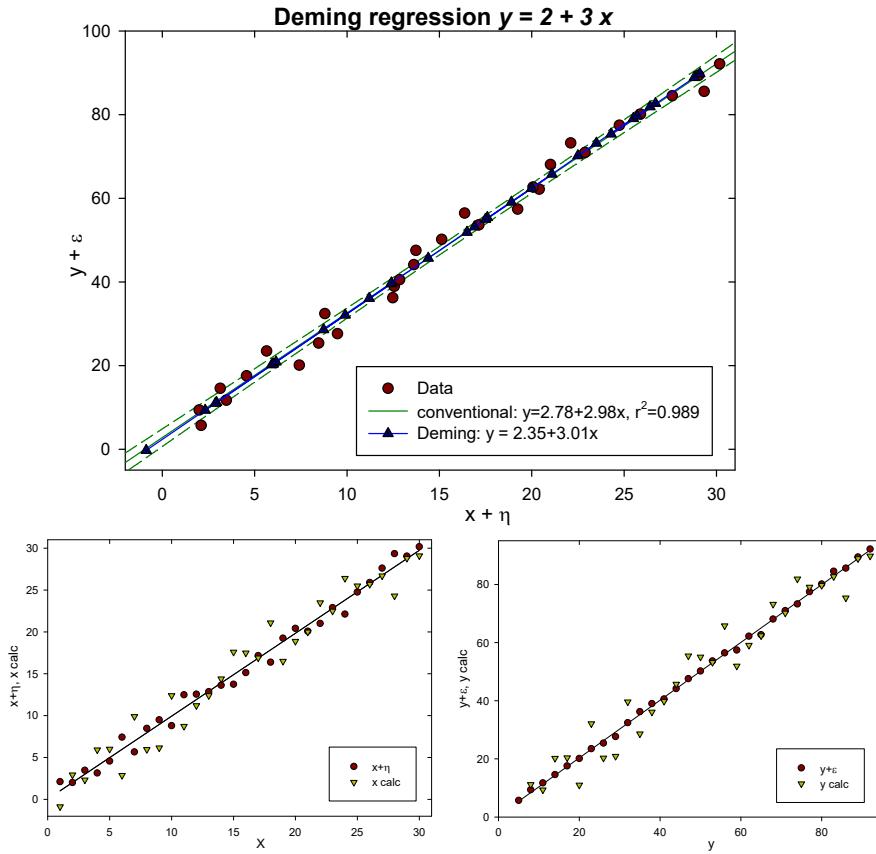


Figure 12.4.: DEMING-regression of an artificial data set with random noise. *Top*: The data and the results of standard and DEMING-regression. *Bottom*: Effect of DEMING-regression on the noise of the  $x$ - (left) and  $y$ -data (right). The regression results are closer to the line through the origin than the noisy data were.

## 12.5. DEMING-regression

A DEMING-regression [6, 16, 17] is used when both dependent and independent variable have errors (see fig. 12.4). The errors of dependent and independent variable are assumed independent, normally distributed, and with known ratio of their variance:

$$y_i = y_i^* + \epsilon_i \quad (12.20)$$

$$x_i = x_i^* + \eta_i \quad (12.21)$$

$$\delta = \frac{\sigma_\epsilon^2}{\sigma_\eta^2} \quad (12.22)$$

The most common problem is that  $\delta$  is unknown. In the special case  $\delta = 1$ , DEMING-regression becomes orthogonal regression, which minimises the sum of squared distances between data points and regression line perpendicular to the line.

## 12. Linear and linearising Regression

As estimate for  $\epsilon$  and  $\eta$  we can use the reciprocals of the reading error (obtained by error propagation, where necessary), divided by  $\sqrt{k}$  if they are the average of  $k$  measurements.

Then the weighted sum of squared distances is minimised

$$\text{RSS} = \sum_{i=1}^n \left( \frac{\epsilon_i^2}{\sigma_\epsilon^2} + \frac{\eta_i^2}{\sigma_\eta^2} \right) = \frac{1}{\sigma_\epsilon^2} \sum_{i=1}^n \left( (\mathbf{y}_i - \beta_0 - \beta_1 \mathbf{x}_i^*)^2 + \delta (\mathbf{x}_i - \mathbf{y}_i^*)^2 \right) \quad (12.23)$$

With  $s_{\mathbf{xx}} = 1/n \sum_{i=1}^n (\mathbf{x}_i - \bar{\mathbf{x}})^2$  the sample variance of  $\mathbf{x}$ ,  $s_{\mathbf{yy}} = 1/n \sum_{i=1}^n (\mathbf{y}_i - \bar{\mathbf{y}})^2$  the sample variance of  $\mathbf{y}$ , and  $s_{\mathbf{xy}} = 1/n \sum_{i=1}^n ((\mathbf{x}_i - \bar{\mathbf{x}})(\mathbf{y}_i - \bar{\mathbf{y}}))$  the sample covariance of  $\mathbf{x}, \mathbf{y}$ . Note the small s, which distinguishes these second order moments from the first order ones used above.

We then get

$$b = \frac{s_{\mathbf{yy}} - \delta s_{\mathbf{xx}} + \sqrt{(s_{\mathbf{yy}} - \delta s_{\mathbf{xx}})^2 + 4\delta s_{\mathbf{xy}}^2}}{2s_{\mathbf{xy}}} \quad (12.24)$$

$$a = \bar{\mathbf{y}} - b\bar{\mathbf{x}} \quad (12.25)$$

$$\hat{\mathbf{x}}_i = \mathbf{x}_i + \frac{b}{b^2 + \delta} (\mathbf{y}_i - a - b\mathbf{x}_i) \quad (12.26)$$

$$(12.27)$$

The errors of the parameters are not available analytically and must be determined by bootstrapping.

Listing 12.5: Deming-regression

```

1  PROCEDURE Deming (CONST x, y : VectorTyp; delta : double;
2                      VAR xCalc, yCalc : VectorTyp; VAR Res : ResultTyp);
3
4  VAR i, n, s                         : WORD;
5      Sx, Sy, sxx, syy, sxy, xMean, yMean, b_ : double;
6      Significance : SignificanceType;
7      c           : CHAR;
8
9  BEGIN
10    IF VectorLength(x) <> VectorLength(y)
11    THEN
12      BEGIN
13        RegressionError := TRUE;
14        c := WriteErrorMessage('Deming regression: unequal length of data
15                               vectors');
16        EXIT;
17      END;
18    n := VectorLength(x);                  // calculate
19    s := 0;
20    FOR i := 1 TO n DO
21      BEGIN
22        xCalc[i] := x[i];
23        yCalc[i] := y[i];
24      END;
25    FOR i := 1 TO n DO
26      BEGIN
27        xCalc[i] := xCalc[i] + (s - b_* * yCalc[i]) / (b_* * b_ + delta);
28        yCalc[i] := yCalc[i] + (s - b_* * xCalc[i]) / (b_* * b_ + delta);
29      END;
30    Res := Deming(xCalc, yCalc, delta);
31  END;
32
33  FUNCTION Deming (CONST x, y : VectorTyp; delta : double) : ResultTyp;
34  VAR
35    i, n, s                         : WORD;
36    Sx, Sy, sxx, syy, sxy, xMean, yMean, b_ : double;
37    Significance : SignificanceType;
38    c           : CHAR;
39    Res          : ResultTyp;
40
41  BEGIN
42    n := VectorLength(x);
43    s := 0;
44    FOR i := 1 TO n DO
45      BEGIN
46        xMean := x[i];
47        yMean := y[i];
48      END;
49    FOR i := 1 TO n DO
50      BEGIN
51        xMean := xMean + x[i];
52        yMean := yMean + y[i];
53      END;
54    xMean := xMean / n;
55    yMean := yMean / n;
56    FOR i := 1 TO n DO
57      BEGIN
58        sxx := sxx + (x[i] - xMean) * (x[i] - xMean);
59        syy := syy + (y[i] - yMean) * (y[i] - yMean);
60        sxy := sxy + (x[i] - xMean) * (y[i] - yMean);
61      END;
62    b_ := (syy - delta * sxx) / (sxy + delta * sxx);
63    Res := Deming(x, y, b_, delta);
64  END;
65
66  FUNCTION Deming (CONST x, y : VectorTyp; b_ : double; delta : double) : ResultTyp;
67  VAR
68    i, n, s                         : WORD;
69    Sx, Sy, sxx, syy, sxy, xMean, yMean, b_ : double;
70    Significance : SignificanceType;
71    c           : CHAR;
72    Res          : ResultTyp;
73
74  BEGIN
75    n := VectorLength(x);
76    s := 0;
77    FOR i := 1 TO n DO
78      BEGIN
79        xMean := x[i];
80        yMean := y[i];
81      END;
82    FOR i := 1 TO n DO
83      BEGIN
84        xMean := xMean + x[i];
85        yMean := yMean + y[i];
86      END;
87    xMean := xMean / n;
88    yMean := yMean / n;
89    FOR i := 1 TO n DO
90      BEGIN
91        sxx := sxx + (x[i] - xMean) * (x[i] - xMean);
92        syy := syy + (y[i] - yMean) * (y[i] - yMean);
93        sxy := sxy + (x[i] - xMean) * (y[i] - yMean);
94      END;
95    b_ := (syy - delta * sxx) / (sxy + delta * sxx);
96    Res := Deming(x, y, b_, delta);
97  END;
98
99  FUNCTION Deming (CONST x, y : VectorTyp; b_ : double; delta : double) : ResultTyp;
100 END Deming;
```

```

19  Sx := 0;
20  Sy := 0;
21  FOR i := 1 TO n DO
22    BEGIN
23      IF IsNaN(GetVectorElement(x, i)) OR IsNaN(GetVectorElement(y, i))
24        THEN
25        ELSE
26          BEGIN
27            INC(s);                                // valid data
28            pairs
29            Sx := Sx + GetVectorElement(x, i);
30            Sy := Sy + GetVectorElement(y, i);
31          END;
32    END;
33    xMean := Sx / s;                         // calculate 2nd
34    moments
35    yMean := Sy / s;
36    sxx := 0;
37    syy := 0;
38    sxy := 0;
39  FOR i := 1 TO n DO
40    BEGIN
41      IF IsNaN(GetVectorElement(x, i)) OR IsNaN(GetVectorElement(y, i))
42        THEN
43        ELSE
44          BEGIN
45            sxx := sxx + Sqr(GetVectorElement(x, i) - xMean);
46            syy := syy + Sqr(GetVectorElement(y, i) - yMean);
47            sxy := sxy + (GetVectorElement(x, i) - xMean) *
48              (GetVectorElement(y, i) - yMean)
49          END;
50    END;
51    sxx := sxx/s; // sample variance x
52    syy := syy/s; // sample variance y
53    sxy := sxy/s; // sample covariance x,y           // calculate
54    params
55    Res.b := (syy - delta*sxx + Sqrt((syy-delta*sxx)*(syy-delta*sxx) +
56      4*delta*sxy*sxy))
57      / (2*sxy);
58    Res.sb := NaN; // Deming regression doesn't provide error estimates
59    Res.a := yMean - Res.b*xMean;
60    Res.sa := NaN;
61    Res.r := QuadrantCorrelation(x, y, Significance);
62    Res.P0 := Significance.P0;
63    Res.t := Significance.Testvalue; // actually chi^2, not t
64    CreateVector(xCalc, n, 0.0);           // calculate xCalc and

```

## 12. Linear and linearising Regression

```

yCalc
60 CreateVector(yCalc, n, 0.0);
61 b_ := Res.b/(Res.b*Res.b + delta);
62 FOR i := 1 TO n DO
63 BEGIN
64 IF IsNaN(GetVectorElement(x, i)) OR IsNaN(GetVectorElement(y, i))
65 THEN
66 ELSE
67 BEGIN
68 SetVectorElement(xCalc, i, GetVectorElement(x, i) +
69 (GetVectorElement(y, i) -
70 Res.a - Res.b * GetVectorElement(x, i)));
71 SetVectorElement(yCalc, i, Res.a + Res.b *
72 GetVectorElement(xCalc, i));
73 END;
74 END;
75 END;

```

## 12.6. THEIL-SEN-KENDALL-estimator for noisy data

In this method [7], the median of the slopes of all  $n(n-1)/2$  lines connecting two data points ( $(y_j - y_i)/(x_j - x_i)$   $\forall i, j \in [1 \dots n], i \neq j, x_j \neq x_i$ ) is taken as slope of the regression line  $b$ . The last condition is relevant when replicate data are available for the same  $x$ . Since the determination of slope is the more precise, the larger the difference  $\text{abs}(x_j - x_i)$  is, one can test for a minimal difference rather than for a difference of zero. The average distance is  $\frac{\max(x) - \min(x)}{n}$ , so the significant distance is chosen lower by some arbitrary factor.

The intercept is calculated from  $a = \tilde{y} - b\tilde{x}$ . The alternative estimator for  $a$ , the median of the intercept of all the lines, is less robust.

It is possible to weigh the slopes by  $x_1 - x_2$  on the grounds that larger distances allow a more accurate determination of slopes.

An estimate for the imprecision of slope and intercept can be calculated from their interquartile distance:  $(Q_3 - Q_1)/2$ . This is similar to the standard deviation in least squares regression, except that it includes the centre 50 % rather than 68 % of values. Similar again to linear regression, estimates of error require sufficient data,  $n > 600$ , to be meaningful.

The THEIL-SEN-KENDALL-estimator has the following properties:

- The KENDALL  $\tau$  rank correlation coefficient between  $x_i$  and the corresponding residual  $(y_i - \hat{y}_i)$  is approximately zero, that is, the probability of a point being above or below the regression line does not depend on  $x_i$ .
- The median of the residuals is approximately zero; that is, the fit line passes above and below equal numbers of points.

- The estimates are more robust against noise than the least squares estimator,  $1 - 1/\sqrt{2} \approx 30\%$  of the data may be arbitrarily corrupted before accuracy drops.
- The non-parametric THEIL-SEN-KENDALL-estimator is almost as efficient as the parametric least squares method in the absence of outliers and normality violations, and much better if these conditions are not met.

An even more robust method for slope calculation is to calculate first the median of all lines going through an  $\mathbf{x}_i$ , and then calculate the median of those for all  $\mathbf{x}$  [18]. The breakdown point of this method is 50 % corrupted data. However, this method is computationally more expensive, it also requires repeat measurements for each  $\mathbf{x}_i$ .

With the THEIL-SEN-KENDALL-estimator being a non-parametric method of linear regression, it makes sense to also use a non-parametric correlation coefficient, here the quadrant correlation (see subsection 11.2.3 on page 356). The significance of `Res.P0` is calculated from a  $\chi^2$  test with  $v = 1$ , thus `Res.t` actually contains  $\chi^2$ .

Listing 12.6: Theil-Sen-Kendall-estimator

```

1 PROCEDURE TheilSenKendall (CONST x, y : VectorTyp; VAR yCalc : VectorTyp;
2                               VAR Res : ResultTyp);
3
4 VAR i, j, n, s : WORD;
5   Slopes, Slopes_big, Intercepts, Intercepts_big, xSorted, ySorted :
6     VectorTyp;
7   x1, x2, y1, y2, xmin, xmax, xdiff, Sx, Sy, xMed, yMed :
8     double;
9   Significance : SignificanceType;
10  C : CHAR;
11  unknown : BOOLEAN;
12
13 BEGIN
14   IF VectorLength(x) <> VectorLength(y)
15     THEN
16       BEGIN
17         RegressionError := TRUE;
18         c := WriteErrorMessage('Linear regression: unequal Length OF
19           dependent AND independent data vector');
20         EXIT;
21       END;
22   n := VectorLength(x);
23   s := 0;
24   CreateVector(Slopes_big, Round(n*(n-1)/2 + 0.5), 0.0); // maximal
25   possible number OF slopes
26   xmax := FindLargest(x);
27   xmin := FindSmallest(x);

```

## 12. Linear and linearising Regression

```

24     xdiff := (xmax - xmin) / (5 * n); // factor 5 IS arbitrary
25     FOR i := 1 TO n DO
26         FOR j := Succ(i) TO n DO
27             BEGIN
28                 x1 := GetVectorElement(x, i);
29                 x2 := GetVectorElement(x, j);
30                 y1 := GetVectorElement(y, i);
31                 y2 := GetVectorElement(y, j);
32                 unknown := IsNaN(x1) OR IsNaN(x2) OR IsNaN(y1) OR IsNaN(y2);
33                 IF (Abs(x1 - x2) < xdiff) OR unknown
34                     THEN
35                 ELSE
36                     BEGIN
37                         INC(s);
38                         SetVectorElement(Slopes_big, s, (y1-y2)/(x1-x2));
39                     END;
40             END;
41             CreateVector(slopes, s, 0.0);
42             FOR i := 1 TO s DO // remove any empty values from slope vector
43                 SetVectorElement(slopes, i, GetVectorElement(slopes_big, i));
44             DestroyVector(Slopes_big);
45             Res.b := Median(slopes);
46             Res.sb := (Quantile(slopes, 0.75) - Quantile(slopes, 0.25)) / 2;
47             CopyVector(x, xSorted);
48             xMed := Median(xSorted);
49             CopyVector(y, ySorted);
50             yMed := Median(ySorted);
51             Res.a := yMed - Res.b * xMed; // most stable estimator FOR intercept
52             CreateVector(Intercepts_big, n, 0.0);
53             s := 0;
54             FOR i := 1 TO n DO // calculate intercepts FOR all x/y pairs
55                 IF IsNaN(GetVectorElement(x, i)) OR IsNaN(GetVectorElement(y, i))
56                     THEN
57                 ELSE
58                     BEGIN
59                         SetVectorElement(Intercepts_big, i, GetVectorElement(y, i) -
60                             Res.b * GetVectorElement(x, i));
61                         INC(s);
62                     END;
63             CreateVector(Intercepts, s, 0.0);
64             FOR i := 1 TO s DO // remove any empty values from intercept vector
65                 SetVectorElement(Intercepts, i, GetVectorElement(Intercepts_big, i));
66             DestroyVector(Intercepts_big);
67             Res.sa := (Quantile(intercepts, 0.75) - Quantile(intercepts, 0.25)) / 2;
68             Res.r := QuadrantCorrelation(x, y, Significance);
69             Res.P0 := Significance.P0;

```

```

69  Res.t := Significance.Testvalue; // actually chi^2, NOT t
70  CreateVector(yCalc, n, 0.0);
71  FOR i := 1 TO n DO
72    SetVectorElement(yCalc, i, Res.a + Res.b * GetVectorElement(x, i));
73  DestroyVector(xSorted);
74  DestroyVector(ySorted);
75  DestroyVector(Slopes);
76  DestroyVector(Intercepts);
77 END;

```

## 12.7. The direct plot of EISENTHAL & CORNISH-BOWDEN

This method is relevant for enzyme kinetics data, that is the measurement of the reaction velocity  $v$  as function of substrate concentration  $[S]$ , which is described by the HENRI-MICHAELIS-MENTEN- (HMM-) equation:

$$v = \frac{V_{\max}[S]}{K_M + [S]} \quad (12.28)$$

As mentioned earlier, for such hyperbolic relationships linearising regression may be used, but with significant bias. In particular, estimates of the MICHAELIS-constant  $K_M$  is error-prone.

As EISENTHAL & CORNISH-BOWDEN [19, 20] realised, if one places lines through the substrate concentrations on the  $x$ -axis and the corresponding velocity on the  $y$ -axis, all these lines should intersect at  $(K_M, V_{\max})$ . In practice, one gets a cloud of  $\frac{n(n-1)}{2}$  intersection points due to experimental noise (see fig. 12.5). Thus, the median of all intersections between the lines can be used as an unbiased, non-parametric estimator for these two parameters.

The parameters can be calculated from any pair  $i, j$  of measured  $[S], v$  by

$$V_{i,j} = \frac{[S]_i - [S]_j}{\frac{[S]_i}{v_i} - \frac{[S]_j}{v_j}} \quad (12.29)$$

$$K_{i,j} = \frac{v_j - v_i}{\frac{v_i}{[S]_i} - \frac{v_j}{[S]_j}} \quad (12.30)$$

Listing 12.7: Direct plot

```

1 PROCEDURE Eisenthal (CONST Substrate, Velocity : VectorTyp; VAR yCalc :
2   VectorTyp;
3   VAR Res : ResultTyp);

```

## 12. Linear and linearising Regression

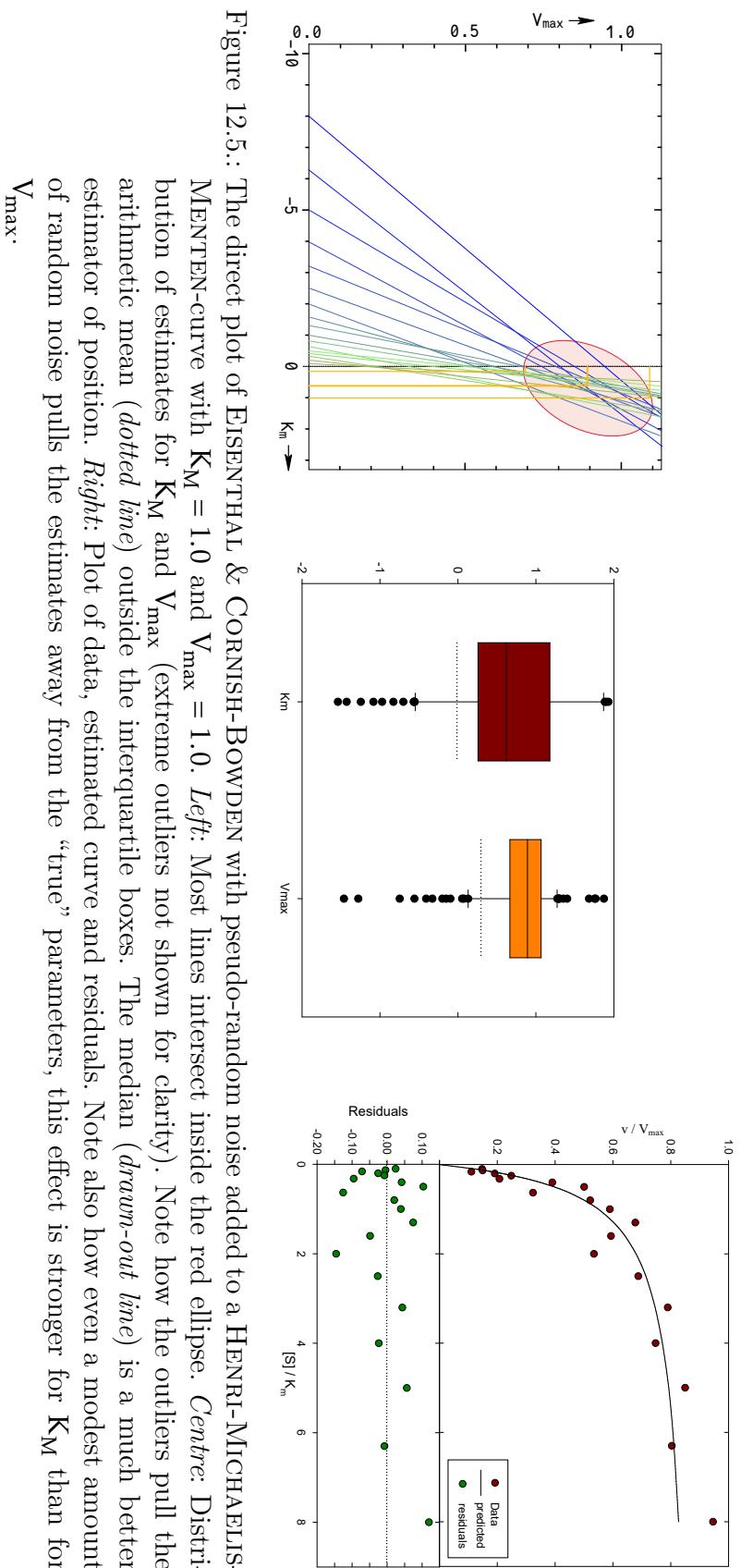


Figure 12.5.: The direct plot of EISENTHAL & CORNISH-BOWDEN with pseudo-random noise added to a HENRI-MICHAELIS-MENTEN-curve with  $K_M = 1.0$  and  $V_{max} = 1.0$ . *Left:* Most lines intersect inside the red ellipse. *Centre:* Distribution of estimates for  $K_M$  and  $V_{max}$  (extreme outliers not shown for clarity). Note how the outliers pull the arithmetic mean (*dotted line*) outside the interquartile boxes. The median (*drawn-out line*) is a much better estimator of position. *Right:* Plot of data, estimated curve and residuals. Note also how even a modest amount of random noise pulls the estimates away from the “true” parameters, this effect is stronger for  $K_M$  than for  $V_{max}$ .

```

4 VAR Km, Vmax, Km_big, Vmax_big : VectorTyp;
5   i, j, n, s                  : WORD;
6   c                           : CHAR;
7   SI, vi, Sj, vj              : double;
8   Significance                : SignificanceType;
9
10 BEGIN
11   IF VectorLength(Substrate) <> VectorLength(Velocity)
12   THEN
13     BEGIN
14       RegressionError := TRUE;
15       c := WriteErrorMessage('Direct plot: unequal Length OF dependent AND
16                               independent data vector');
17       EXIT;
18     END;
19   n := VectorLength(Substrate);
20   s := 0;
21   CreateVector(Km_big, Round(n*(n-1)/2 + 0.5), 0.0); // maximal possible
22   CreateVector(Vmax_big, Round(n*(n-1)/2 + 0.5), 0.0);
23   FOR i := 1 TO n DO
24     FOR j := Succ(i) TO n DO
25       BEGIN
26         SI := GetVectorElement(Substrate, i);
27         Vi := GetVectorElement(Velocity, i);
28         Sj := GetVectorElement(Substrate, j);
29         Vj := GetVectorElement(Velocity, j);
30         IF (IsNaN(SI) OR IsNaN(vi) OR IsNaN(Sj) OR IsNaN(vj))
31         THEN
32           // ignore
33         ELSE
34           BEGIN
35             INC(s);
36             SetVectorElement(Km_big, s, (vj - vi) / (vi/SI - vj/Sj));
37             SetVectorElement(Vmax_big, s, (SI - Sj) / (SI/vi - Sj/vj));
38           END;
39       END;
40   CreateVector(Km, s, 0.0);
41   CreateVector(Vmax, s, 0.0);
42   FOR i := 1 TO s DO // remove any empty values from result vectors
43     BEGIN
44       SetVectorElement(Km, i, GetVectorElement(Km_big, i));
45       SetVectorElement(Vmax, i, GetVectorElement(Vmax_big, i));
46     END;
47   DestroyVector(Km_big);
48   DestroyVector(Vmax_big);

```

## 12. Linear and linearising Regression

```

48  Res.a := Median(Km);
49  Res.sa := (Quantile(Km, 0.75) - Quantile(Km, 0.25)) / 2;
50  Res.b := Median(Vmax);
51  Res.sb := (Quantile(Vmax, 0.75) - Quantile(Vmax, 0.25)) / 2;
52  FOR i := 1 TO s DO
53    Writeln(i:3, ' ', GetVectorElement(Km, i):3:3, ' ',
54      GetVectorElement(Vmax, i):3:3);
55  Res.r := QuadrantCorrelation(Substrate, Velocity, Significance);
56  Res.P0 := Significance.P0;
57  Res.t := Significance.Testvalue; // actually chi^2, NOT t
58  CreateVector(yCalc, n, 0.0);
59  FOR i := 1 TO n DO
60    BEGIN
61      SI := GetVectorElement(Substrate, i);
62      IF IsNaN(SI)
63        THEN
64          SetVectorElement(yCalc, i, NaN)
65        ELSE
66          SetVectorElement(yCalc, i, Res.b * SI / (Res.a + SI));
67    END;
68  DestroyVector(Km);
69  DestroyVector(Vmax);
70
71END.
```

## 12.8. Aberrant data points

If the data set  $\mathbf{x}, \mathbf{y}$  contains outliers, that is, data points with an error much larger than the other points, the calculated slope will hardly be affected. However, the standard deviations for the parameters will be increased and  $r$  decreased. Such outliers should be identified and removed from the data set. On the other hand, data points with high leverage, that is,  $\mathbf{x}_i$  is far away from  $\bar{\mathbf{x}}$ , will strongly affect the parameters.

It is, however, important to use consistent rules for identification of aberrant points. Arbitrary removal of data points which may not fit with ones hypothesis will seriously bias the results and render them useless.

### 12.8.1. Leverage points

With

$$\mathcal{H} = \mathcal{X}(\mathcal{X}^T \mathcal{X})^{-1} \mathcal{X}^T \quad (12.31)$$

the hat-matrix, its  $i$ th diagonal element  $h_{ii}$  is the **leverage** of  $\mathbf{x}_i$ . It describes, how much this point will influence the calculated parameters of the regression and is largest for

those  $\mathbf{x}_i$  that are furthest away from  $\bar{\mathbf{x}}$ . For a simple linear regression,

$$\mathbf{h}_{ii} = \frac{1}{n} - \frac{(\mathbf{x}_i - \bar{\mathbf{x}})^2}{\sum_{j=1}^n (\mathbf{x}_j - \bar{\mathbf{x}})^2} \quad (12.32)$$

$0 \leq \mathbf{h}_{ii} \leq 1$  and any  $\mathbf{h}_{ii} > \frac{3q}{n}$  is considered high leverage.  $q$  is the number of fitted parameters (including intercept).

With those definitions,

$$\mathbf{t}_i = \frac{\hat{\epsilon}_i}{\hat{\sigma} \sqrt{1 - \mathbf{h}_{ii}}} \quad (12.33)$$

is called the **internally STUDENTised residual**. If the absolute value of this residual for any data point exceeds 3, this point can be considered an outlier. If a data point is suspected of being an outlier, it should be excluded from the summation in eqn. 12.34, and  $n$  in its denominator must be decreased accordingly. However, before it is removed from the data set, make sure that it does not result from an incomplete model!

## 12.8.2. Outliers

Unlike the errors  $\epsilon_i$  of the measured  $\mathbf{y}$ , the residuals  $\hat{\epsilon}_i = \mathbf{y}_i - \hat{\mathbf{y}}_i$  obtained from regression cannot be independent from each other, as their sum must equal zero. Their variance can be estimated as

$$\hat{\sigma}^2 = \frac{1}{(n-q)} \sum_{i=1}^n \hat{\epsilon}_i^2 = \frac{1}{(n-q)} \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \quad (12.34)$$

with  $q$  the number of parameters.

COOK's distance is defined as

$$D_i = \frac{(\hat{\beta} - \hat{\beta}^\ominus)^T (\mathcal{X}^T \mathbb{S})(\hat{\beta} - \hat{\beta}^\ominus)^2}{(1+q)s^2} = \frac{\sum_{j=i}^n (\bar{\mathbf{y}}_j - \bar{\mathbf{y}}_j^\ominus)^2}{q \text{MSE}} \quad (12.35)$$

where  $\hat{\beta}_\ominus$  is the calculated least squares slope and  $\bar{\mathbf{y}}_j^\ominus$  the calculated  $\mathbf{y}$  from a refitted regression model in which observation  $i$  has been omitted. MSE the mean squared error. Values with a COOK's distance of more than four times the mean are suspicious, this measure is somewhat more sensitive than other measures of outliers.

## 12.9. Shrinkage

Sometimes, the relative importance of the variables in  $\mathcal{X}$  is unknown. In such cases, it is possible to constrain the sum of parameters  $\beta$ , and thereby improve their variance [21, chapter 6].

### 12.9.1. Ridge regression

In ridge regression, we look for the minimum of

$$\min \left[ \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}})^2 + \lambda \sum_{j=1}^p \beta_j^2 \right] \quad (12.36)$$

, that is, we minimise the sum of both the residual sum of squares and the  $\ell_2$ -norm of the parameters (except intercept  $\beta_0$ ), *i.e.*, the sum of their squares. The relative weight of the later, **the shrinkage penalty**, is determined by  $\lambda \in [0 \dots \infty]$ . If  $\lambda = 0$  we have a conventional least squares regression. As  $\lambda$  increases, the estimates for  $\beta_j$  are shrunk toward zero. For each  $\lambda$ , there is a budget variable  $s$  so that  $\ell_2(\beta) \leq s$ . For small  $\lambda$ ,  $s$  is large and not restrictive, but as  $\lambda$  increases,  $s$  becomes smaller and eventually limiting.

A least square fit has little bias, but may have a lot of variance, that is, a small change in the training data may result in a significant change of the estimated parameters. This variance increases with  $p$ , if  $p > n$  regression isn't even possible. As  $\lambda$  increases, the flexibility of the regression decreases, increasing bias and decreasing variance. As the mean squared error  $MSE = \frac{\sum(\mathbf{y}_i - \hat{\mathbf{y}})^2}{n}$  is a function of the sum of squared bias and variance, it has an optimal value at a particular  $\lambda$ . Hence, the selection of  $\lambda$  is critical and performed by validation (see section 15.1.1 on page 508).

Before ridge regression, the data should be  $z$ -standardised so all variables are on the same scale (namely, standard deviations from mean). Otherwise, the final fit would depend on the scale at which the variables have been measured.

### 12.9.2. Lasso regression

In ridge regression, all parameters are shrunk in the same way, in other words, ridge regression cannot distinguish between important and irrelevant variables. A small change in the equation rectifies this:

$$\min \left[ \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}})^2 + \lambda \sum_{j=1}^p |\beta_j| \right] \quad (12.37)$$

, in other words, we use the  $\ell_1$ -, rather than the  $\ell_2$ -norm of the parameters. This yields **sparse models**, that is, some parameters may be set to exactly zero. Their number depends on  $\lambda$ .

**Method selection:** Lasso regression performs slightly poorer than ridge regression when all predictor variables make a seizable contribution to the predicted variable (signal variables); it is superior when there are irrelevant variables (noise variables). A plot of **MSE vs  $\lambda$**  will show whether shrinkage should be used at all, or whether simple least square regression is sufficient for the problem. Ridge regression can also uncover correlations between variables (**grouping**). It is possible to combine lasso- and ridge-regression

to elastic net regression:

$$\min \left[ \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}})^2 + \lambda_1 \ell_1(\beta) + \lambda_2 \ell_2(\beta) \right] \quad (12.38)$$

## References

- [1] S. NOACK: *Statistische Auswertung von Mess- und Versuchsdaten mit Taschenrechner und Tischcomputer: Anleitungen und Beispiele aus dem Laborbereich* Berlin / New York: De Gruyter, 1980 ISBN: 9783110072631.
- [2] J.D. SPAIN: *Basic Microcomputer Models in Biology* Reading (USA): Adison, 1982 ISBN: 9780201106787.
- [3] G.N. WILKINSON: Statistical Estimation in Enzyme Kinetics, *Biochem. J.* **80**:2 (1961), 324–332 DOI: [10.1042/bj0800324](https://doi.org/10.1042/bj0800324).
- [4] M. ABRAMOWITZ, I.A. STEGUN, eds.: *Handbook of Mathematical Functions* 9th printing New York: Dover, 1964.
- [5] M.L. JOHNSON, ed.: *Essential numerical computer methods* Reliable lab solutions Amsterdam et al.: Elsevier, 2010 ISBN: 9780123849977.
- [6] W.E. DEMING: *Statistical adjustment of data* New York: Wiley, 1943 ISBN: 9780486646855 URL: <https://archive.org/stream/inernet.dli.2015.18293/2015.18293.Statistical-Adjustment-Of-Data#page/n0/mode/2up>.
- [7] H. THEIL: A rank-invariant method of linear and polynomial regression analysis. I, II, III, *Proc. Nederl. Akad. Wetensch.* **53** (1950), 386–392, 521–525, 1397–1412 DOI: [10.1007/978-94-011-2546-8\\_20](https://doi.org/10.1007/978-94-011-2546-8_20).
- [8] M.L. JOHNSON: Use of Least-Squares Techniques in Biochemistry, In: *Essential numerical computer methods* M.L. JOHNSON (editor) Reliable lab solutions Amsterdam: Elsevier, 2010 chap. 1, 1–22 ISBN: 9780123849977 DOI: [10.1016/B978-0-12-384997-7.00001-7](https://doi.org/10.1016/B978-0-12-384997-7.00001-7).
- [9] L.C. FREEMAN: *Elementary applied statistics* New York, London, Sidney: John Wiley & Sons, 1965.
- [10] W.H. PRESS et al.: *Numerical recipes in Pascal: The art of scientific computing* Cambridge: Cambridge University Press, 1989 ISBN: 9780521375160.
- [11] K. LEVENBERG: A method for the solution of certain non-linear problems in least squares, *Quart. Appl. Math.* **2**:2 (1944), 164–168 URL: <https://www.ams.org/qam/1944-02-02/S0033-569X-1944-10666-0/S0033-569X-1944-10666-0.pdf>.
- [12] D. W. MARQUARDT: An algorithm for least-squares estimation of nonlinear parameters, *J. Soc. Industrial Appl. Math.* **11**:2 (1963), 431–441 URL: [http://137.204.42.130/~bittelli/materiale%5C lettura%5C fisica%5C terreno/marquardt%5C\\_63.pdf](http://137.204.42.130/~bittelli/materiale%5C lettura%5C fisica%5C terreno/marquardt%5C_63.pdf).
- [13] J. A. NELDER, R. MEAD: A simplex method for function minimization, *Computer J.* **7**:4 (1965), 308–313 URL: <http://www.ii.uib.no/~lennart/drgrad/Nelder1965.pdf>.

- [14] M. S. CACECI, W. P. CACHERIS: Fitting Curves to Data: The Simplex Algorithm is the Answer, *Byte* **9**:5 (1984), 340–362 URL: [https://www.researchgate.net/publication/246199710\\_Fitting\\_curves\\_to\\_data\\_The\\_simplex\\_algorithm\\_is\\_the\\_answer](https://www.researchgate.net/publication/246199710_Fitting_curves_to_data_The_simplex_algorithm_is_the_answer).
- [15] Y.-S. KIM: Refined Simplex Method for Data Fitting, In: *Astronomical Data Analysis Software and Systems VI* G. HUNT, H.E. PAYNE (editor) vol. 125 ASP Conference Series 1997, 206–209 URL: <https://www.cv.nrao.edu/adass/adassVI/kimys.html>.
- [16] R. J. ADCOCK: A Problem in Least Squares, *Analyst* **5**:2 (1878), 53–54 DOI: [10.2307/2635758](https://doi.org/10.2307/2635758).
- [17] C.H. KUMMELL: Reduction of Observation Equations Which Contain More Than One Observed Quantity, *Analyst* **6**:4 (1879), 97–105 DOI: [10.2307/2635646](https://doi.org/10.2307/2635646).
- [18] A.F. SIEGEL: Robust regression using repeated medians, *Biometrika* **69**:1 (1982), 242–244 DOI: [10.1093/biomet/69.1.242](https://doi.org/10.1093/biomet/69.1.242).
- [19] R. EISENTHAL, A. CORNISH-BOWDEN: The direct linear plot. A new graphical procedure for estimating enzyme kinetic parameters, *Biochem. J.* **139**:3 (1974), 715–720 DOI: [10.1042/bj1390715](https://doi.org/10.1042/bj1390715) URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1166335/pdf/biochemj00583-0239.pdf>.
- [20] A. CORNISH-BOWDEN, R. EISENTHAL: Statistical considerations in the estimation of enzyme kinetic parameters by the direct linear plot and other methods, *Biochem. J.* **139**:3 (1974), 721–730 DOI: [10.1042/bj1390721](https://doi.org/10.1042/bj1390721) URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1166336/pdf/biochemj00583-0245.pdf>.
- [21] G. JAMES et al.: *An introduction to statistical learning with applications in R* 1st ed. Springer texts in statistics New York, Heidelberg, Dordrecht, London: Springer, 2013 ISBN: 9781461471387 URL: [https://github.com/tpn/pdfs/blob/master/An%20Introduction%20To%20Statistical%20Learning%20with%20Applications%20in%20R%20\(ISLR%20Sixth%20Printing\).pdf](https://github.com/tpn/pdfs/blob/master/An%20Introduction%20To%20Statistical%20Learning%20with%20Applications%20in%20R%20(ISLR%20Sixth%20Printing).pdf).

# 13. Regression of curves: The simplex algorithm

## Abstract

The simplex algorithm is an iterative technique that can be used for any regression problem, from straight lines to curves of arbitrary complexity. The differential of the equation need not be known nor even exist. It is not limited to minimising the sum of squared residuals, but can use any optimisation criterium that is continuous. Minimum median residual is used for noisy data, minimum  $\chi^2$  for heteroscedastic data. The biggest disadvantage of the simplex algorithm is that it does not directly produce error estimates for the parameters, these need to be obtained by bootstrapping.

## 13.1. How does the simplex algorithm work?

Let us return to the example in fig. 12.2 on page 403, a HENRI-MICHAELIS-MENTEN curve with considerable scatter. We can now choose arbitrary pairs of the parameters,  $V_{\max}$  and  $K_m$ , each such pair will give a residual sum of squares (RSS) (see fig. 13.1). What we would like to find is the pair of parameters for which RSS becomes minimal. All RSS-values of all parameter sets (although we deal with two parameters here for ease of drawing, the method can handle an arbitrary number of parameters  $p$ , the error surface then has  $p + 1$  dimensions) together form the **error surface** of the problem. In effect, we look for way to take an arbitrary starting set of parameters and from there move “downhill” until we reached the minimum with sufficient precision. One way of doing this is to calculate the gradient ( $\nabla$ , multi-dimensional slope) of the function at your current estimate for the parameters and move downward in the direction to the **steepest descent** for a small distance, arriving at your new, improved estimate. This is repeated until further moves no longer reduce the RSS, that is, until the gradient becomes close to zero. The LEVENBERG-MARQUARDT [1, 2] algorithm uses (in part) this method. However, this requires the error function not only to be continuously differentiable (without singularities as for example in  $f(x) = 1/x, x = 0$ ), but the first derivative must also be known (how good were you in calculus?). The main advantage of steepest descent methods is that since the gradient is known, it is easy to calculate error estimates for the found parameters directly.

The simplex algorithm [3–5] takes a geometric approach, and is therefore more generally applicable. Neither the regression function nor its error surface need to be differentiable, all that is required is that we can calculate the dependent variable  $\hat{y}$  for any

### 13. Regression of curves: The simplex algorithm

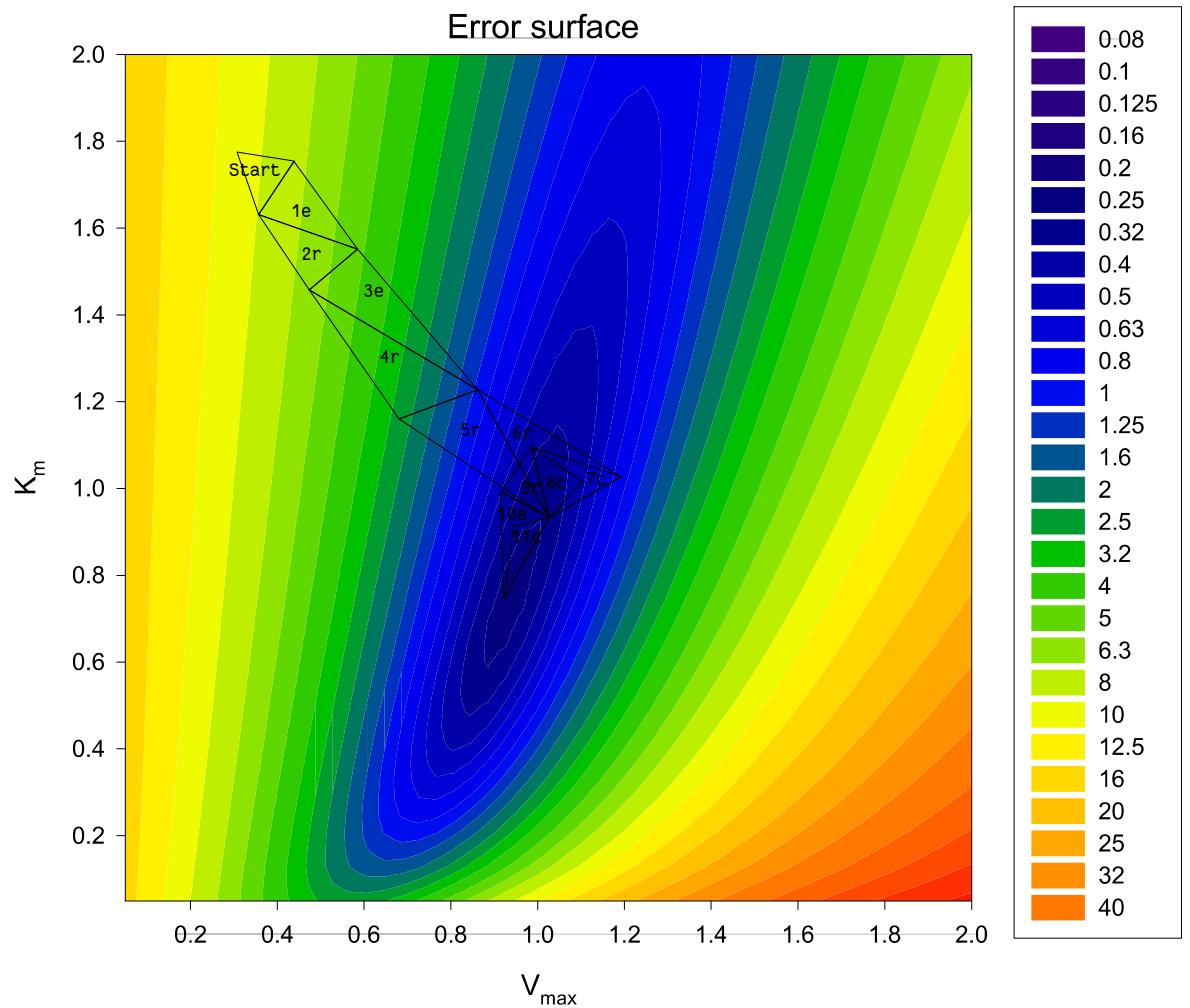


Figure 13.1.: RSS of the data in fig. 12.2 on page 403 as function of the values of the parameters. Each pair of parameters gives a RSS-value (colour-coded like a topographic map). The black triangles indicate the moves of the simplex from an arbitrary starting point, note how the simplex expands as it moves downhill and then contracts again near the minimum. For details see text.

### 13.1. How does the simplex algorithm work?

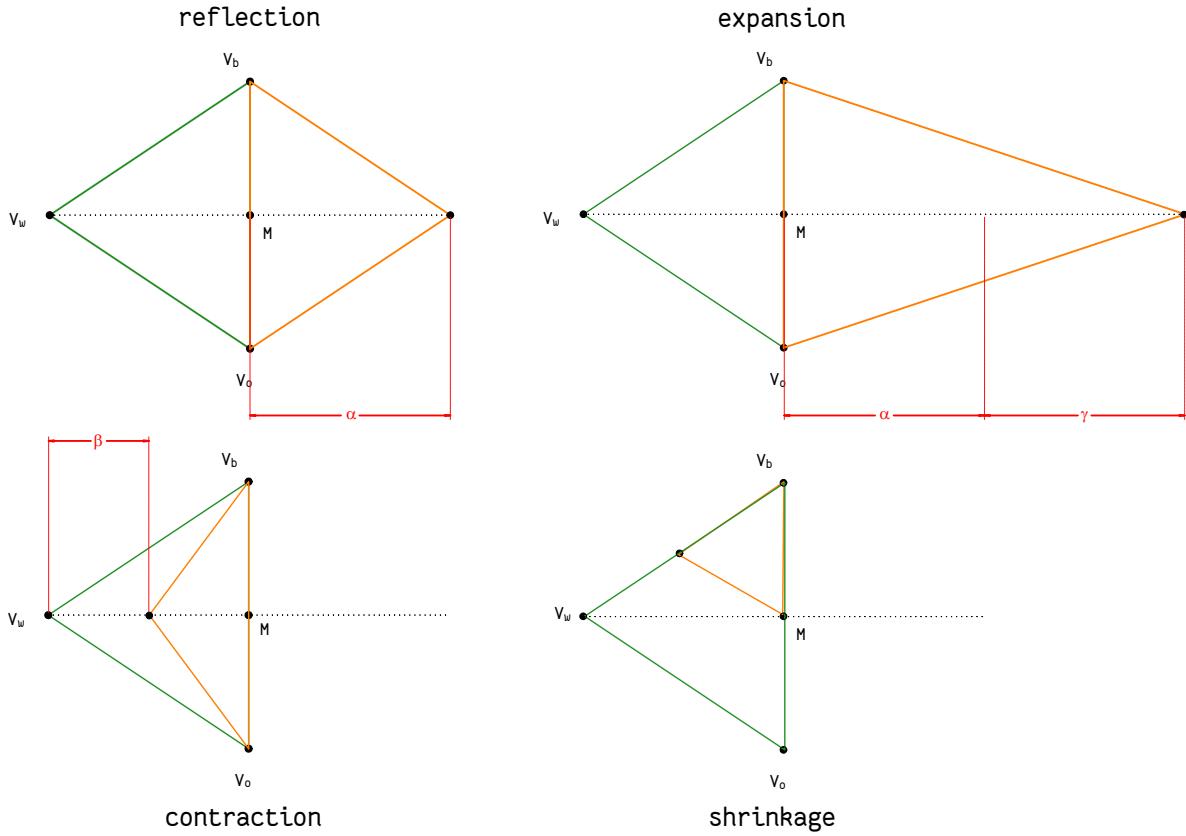


Figure 13.2.: Possible movements of a simplex. The old simplex is in *green*, the new in *orange*. For details see text.

data vector  $\mathbf{x}$  and any parameter vector  $\beta$ . The simplex method can minimise not only the [RSS](#), but also other error functions like  $\chi^2$  or the median of residuals. This can be useful for non-homoscedastic or noisy data. However, simplex does not directly give error estimates for the parameters, these need be calculated from bootstrapping [6, 7].

A simplex is a geometric figure that has one more vertex than the space in which it lives has dimensions. The  $p$  Parameters of a fitting problem define a  $p$ -dimensional space, in our example we have two parameters, the space hence is a plane and the simplex is a triangle. With tree parameters, the simplex would be a tetrahedron, *etc*. Each vertex is characterised by its parameters and, in addition, by the [RSS](#) associated with it. We then discard the vertex with the highest [RSS](#), replacing it with another one with (hopefully) lower [RSS](#), so that the vertex “moves” downhill. At each step, a vertex can do one of four things (see also fig. 13.2):

**reflection** move the worst vertex  $V_w$  along the line that connects it to the midpoint  $M$  of the other vertexes for the distance  $\overline{V_wM} + \alpha \overline{V_wM}$ . In other words, with  $\alpha = 1$  the worst vertex is mirrored on the connection between all other vertexes. The new vertex is accepted, if its [RSS](#) is neither higher than that of  $V_w$ , nor lower than that of the best vertex  $V_b$ .

### 13. Regression of curves: The simplex algorithm

**contraction** If the [RSS](#) of the reflected vertex is worse than  $V_w$ , the program tries to move this vertex for only  $\beta\overline{V_wM}$ . The new vertex is accepted if its [RSS](#) is lower than that of  $V_w$ .

**expansion** If the new vertex is better than the previously best vertex  $V_b$ , the program tries to move it even further along the line used for reflection, that is for a total of  $\overline{V_wM} + (\alpha + \gamma)\overline{V_wM}$ . This expanded vertex is accepted if its [RSS](#) is lower than that of the discarded  $V_w$ .

**shrinkage** If in contraction the [RSS](#) of the new vertex is lower than that of  $V_w$ , the program keeps the best vertex and moves all others toward it by half their distance.

It was shown in [5], however, that shrinkage can never occur, and hence is redundant. The simplex algorithm is guaranteed never to diverge, it is quite efficient as no matrix operations are required. Round-off errors are minimised, but all calculations should be done at least in double precision, with the old Turbo Pascal `real`-type the algorithm sometimes did not converge, the simplex continued to rotate around the optimum until the maximal number of iterations was reached. To avoid the simplex to become trapped in a local minimum, the starting estimates should be as close to the final parameters as possible. Alternatively, use several – wildly different – starting values and verify that they result in the same final parameters.

Instead of [RSS](#), other parameters may also be used to control the fit. For noisy data, the median of residuals is more stable against outlying data points. For data with constant relative (rather than absolute) error,  $\chi^2$  (the sum of squared relative errors) is the appropriate fitting criterium.

## 13.2. Examples

### 13.2.1. Enzyme kinetics with [S] spanning several orders of magnitude: heteroscedasticity

Normally, enzyme kinetics data cover substrate concentrations of two orders of magnitude, between  $0.1K_m$  and  $10K_m$ . A minimum of 12 data points, equally spaced, in this interval are required for fitting of the HENRI-MICHAELIS-MENTEN (HMM) curve [8]. However, some ATPases have several ATP-binding sites with very different  $K_m$ . For example, in P-type ATPases like Na/K-ATPase these are  $0.1 \mu M$  and  $300 \mu M$ , respectively. The question arose, whether ABC-type ATPases like Mdr1, who have two ATP-binding sites in each molecule, would operate with HMM-kinetics, or show cooperativity like the P-type ATPases (where several enzyme molecules with one binding site each have to work together). Therefore, the  $v/[S]$  curve of Mdr1 was measured in the concentration range from  $100 \text{ nM}$  to  $10 \text{ mM}$ , that is, over 5 orders of magnitude [9]. All experiments contained the same amount ( $3500 \text{ Bq}$ ) of  $\gamma^{33}\text{P}$ -ATP as label, but different concentrations of unlabelled ATP, resulting in different specific radioactivity. The formation of  $\text{H}^{33}\text{PO}_4^{2-}$  was measured, resulting in velocities that had the same *relative* error. Use of

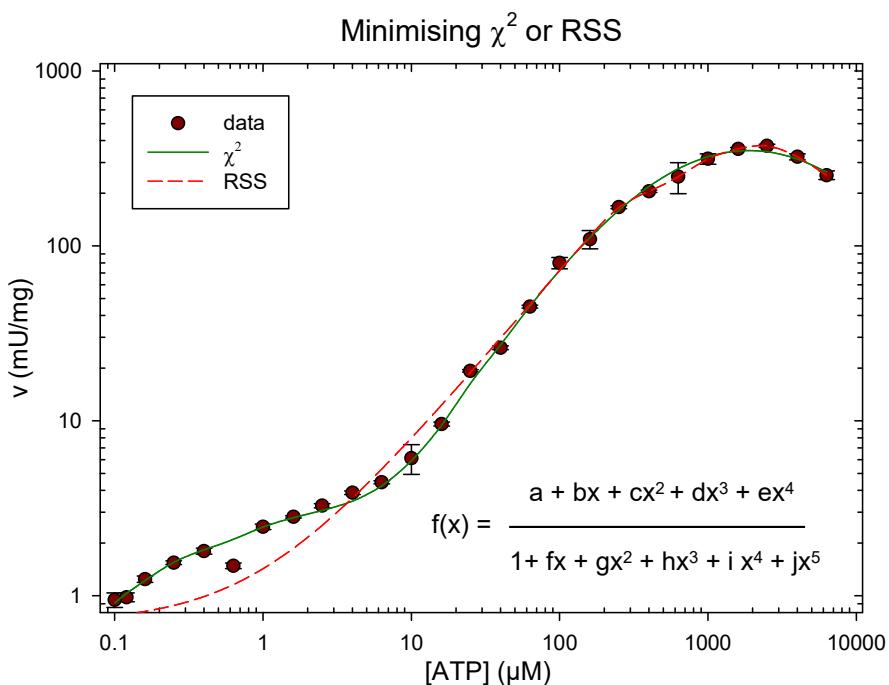


Figure 13.3.: ATPase-activity of Mdr1 as function of [ATP]. The data can be well described by a model with four catalytic and one inhibitory binding site, this results in a ratio of polynomials of 4/5 grade. However, while points at high [ATP] can be fitted by minimising  $\chi^2$  or RSS, at low concentrations the data can be fitted only with  $\chi^2$ . The data have equal relative error, at low [ATP] the velocities, and hence absolute errors, are so small that they make negligible contribution to RSS.

### 13. Regression of curves: The simplex algorithm

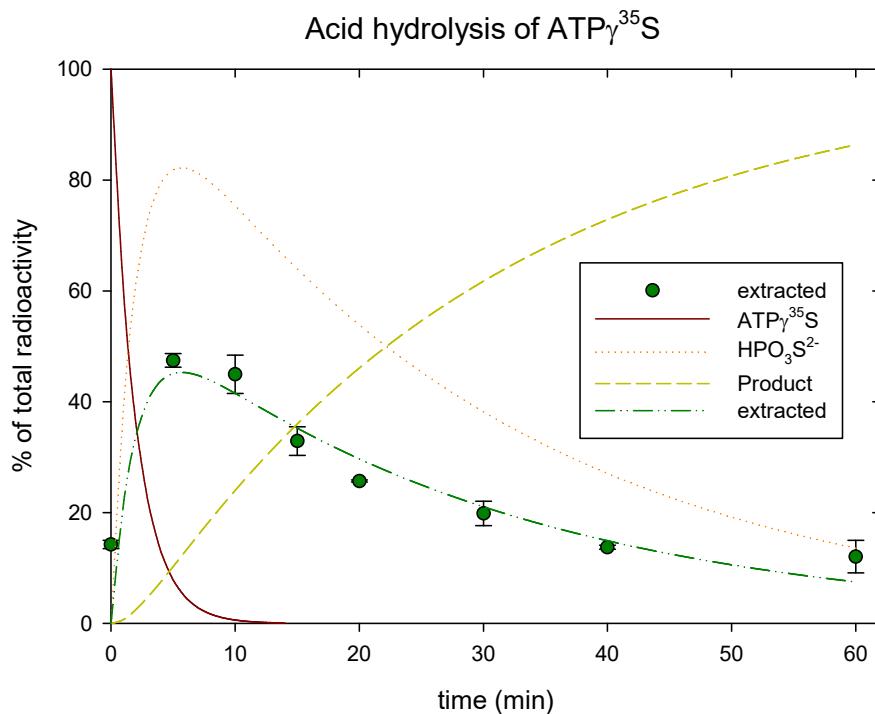


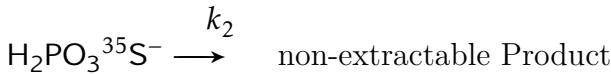
Figure 13.4.: System of linear differential equations for the equation  $A \rightarrow B \rightarrow C$ , the measured values are the intermediate product multiplied by an extraction efficiency (mean and standard deviation of 4 repetitions). For details see text.

the minimal RSS criterion, however, requires data with the same *absolute* error (homoscedasticity). Only minimising  $\chi^2 = \sum_{i=1}^n \left[ \frac{(\hat{y}_i - y_i)}{y_i} \right]^2$ , the sum of squared *relative* errors, allows these data to be fitted, RSS fails at low substrate concentrations (see fig. 13.3).

#### 13.2.2. Fitting of a system of differential equations to a data set

The question was whether 70 kDa heat shock protein could hydrolyse the ATP-analog ATP $\gamma^{35}\text{S}$ . The common method for the determination of  $^{33}\text{PO}_4^{2-}$  is to convert the phosphate to  $(\text{NH}_4)_3[\text{P}(\text{Mo}_3\text{O}_{10})_4]$  in the presence of carrier phosphate and to extract into organic solvent, the radioactivity in that extract is then counted. To determine whether this method could also be used for thiophosphate, a small amount of ATP $\gamma^{35}\text{S}$  was hydrolysed in 2 N  $\text{H}_2\text{SO}_4$  at 95°C, and samples taken at several time points. Under these conditions, ATP and similar high-energy compounds are hydrolysed quantitatively within 7 min. However, it turned out that the thiophosphate is destroyed by the boiling sulphuric acid. Thus, we have a system





giving a first-order system of coupled linear differential equations

$$\begin{cases} \frac{dA}{dt} = -k_1 A, & A_0 = 100\% \\ \frac{dB}{dt} = k_1 A - k_2 B, & B_0 = 0\% \\ \frac{dC}{dt} = k_2 B, & C_0 = 0\% \end{cases} \quad (13.1)$$

The reaction was simulated as a system of coupled differential expression with a DP5(4)T4 RUNGE-KUTTA-algorithm [10] with the parameters  $k_1, k_2$  and extraction efficiency, the independent variable time and the dependent variable radioactivity extracted. Fitting was performed by simplex with RSS as criterium. The best fit was obtained with  $k_1 = 0.509 \text{ min}^{-1}$ ,  $k_2 = 0.034 \text{ min}^{-1}$  and an extraction efficiency of 55.1 % (see fig. 13.4). The extraction method is therefore suitable for the determination of thiophosphate, if the relatively low extraction efficiency is taken into account.

This example shows the incredible flexibility of the simplex algorithm even for very unusual fitting problems.

## 13.3. Error estimation

### 13.3.1. Confidence intervals for parameters

When the noise in the data is normally distributed with constant variance, all information about the parameter vector  $\beta$  is in the  $\text{RSS}(\beta) = \sum_{i=1}^n [\mathbf{y}_i - f(\mathcal{X}_{i,.}, \beta)]$  [11]. The parameter inference region for the interference interval  $(1-\alpha)$  is an ellipsoid given by  $\text{RSS}(\beta) = \text{RSS}_F$  with

$$\text{RSS}_F = \text{RSS}(\hat{\beta}) \left[ 1 + \frac{p}{n-p} F(p, n-p, \alpha) \right] \quad (13.2)$$

Alternatively,  $\text{RSS}(\beta) = \text{RSS}_t$  can be used with

$$\text{RSS}_t = \text{RSS}(\hat{\beta}) \left[ 1 + \frac{t^2(n-p, \alpha/2)}{n-p} \right] \quad (13.3)$$

Thus, we can profile the RSS surface as follows:

1. Out of the  $p$  parameters, select  $\beta_j$  and the increment  $\Delta = 0.1 \times \text{se}(\hat{\beta}_j)$ .
2. Initialise  $\beta_j = \hat{\beta}_j$  and  $\tilde{\beta}(\beta_j) = \hat{\beta}$ .
3. Increment  $\beta_j \leftarrow \beta_j + \Delta$  and use previous  $\tilde{\beta}(\beta_j)$  as starting value. Converge to  $\tilde{\beta}(\beta_j)$  the profile RSS. Store  $\beta_j, \tilde{\beta}(\beta_j), \widetilde{\text{RSS}}(\beta_j)$ .
4. Repeat (3) as necessary.
5. Set  $\Delta = -\Delta$  and go to (2) to profile  $\beta_j < \hat{\beta}_j$ .

### 13. Regression of curves: The simplex algorithm

6. When enough information has been obtained on  $\beta_j$  to estimate its confidence interval, increment  $j$  and go to (1).

We want for all parameters to find those values of  $\beta_p$  where  $\text{RSS}(\beta_j) = \text{RSS}_t$ . We could do that by plotting  $\text{RSS}(\beta_j)$  against  $\beta_j$ , or, alternatively, by

1. Calculate STUDENTised parameters  $\delta(\beta_j) = \frac{(\beta_j - \hat{\beta}_j)}{\text{se}(\beta_j)}$ .
2. Convert the profile  $\text{RSS}$  to corresponding profile  $t$ -values by  $t(\beta_j) = \text{sgn}(\beta_j - \hat{\beta}_j) \sqrt{\frac{\text{RSS}(\beta_j) - \text{RSS}(\hat{\beta})}{s^2}}$ .
3. Plot  $t(\beta_j)$  against  $\delta(\beta_j)$ .
4. The points defining a  $1-\alpha$  marginal interval correspond to the points where  $t(\beta_j) = \pm t(n-p, \alpha/2)$ . This are the intersections of the  $t(\beta_j)$  against  $\delta(\beta_j)$  curve with the lines  $\pm t(n-p, \alpha/2)$ .
5. Convert the  $\delta$ - into  $\beta$ -values.

The conversion to STUDENTised parameters makes comparison between the parameters within the model and across different models (with a different number of parameters) easier. If the model is linear in the parameters, the profile  $t$  plot would be a line through the origin with slope 1. One can also plot the trace vector  $\tilde{\beta}(\beta_j)$  against the profile parameter  $\beta_j$ , in linear models this results in a straight line with slope the correlation  $r$  between the parameters, in non-linear models we get curves.

### Monte Carlo methods

This method [6, 7] is computationally expensive. It consists of the following steps:

1. Determine the most probable parameters
2. Calculate  $\hat{\mathbf{y}}$  from this model and treat this vector as “perfect”
3. Generate synthetic data by adding pseudo-random noise to  $\hat{\mathbf{y}}$ , repeat  $1 \times 10^2 - 1 \times 10^3$  times.
4. Determine and tabulate the most likely parameters of these simulated data sets
5. Generate a histogram of the distribution of the simulated parameters

The distribution of the noise should be identical to the standard deviation of the experimental data. Alternatively, one can use the residuals from step 1, reshuffling them between the data points. This does not make any assumptions about the error distribution of the data points.

### 13.3.2. Goodness of fit

If the function used for fitting is appropriate for the data, the residuals  $r_i = y_i - \hat{y}_i$  should have a random, Gaussian distribution. This can be tested in various ways.

#### The runs test

The runs test quantifies trends in residuals. If the residuals are distributed randomly, the probability of a residual  $r_i$  to be positive or negative is independent of the previous  $r_{i-1}$  and following  $r_{i+1}$  residual. If, however, there is a systematic deviation between the data and the model, clusters of positive and negative residuals will occur. A **run** is a group of consecutive residuals with the same sign. Trends will reduce the number of runs, serial correlations will increase it. The expected number of runs  $R_e$  can be calculated from the total number of positive ( $n_+$ ) and negative residuals ( $n_-$ ) as  $R_e = \frac{2n_+n_-}{n_++n_-} + 1$ , with a variance of  $\sigma_R^2 = \frac{2n_+n_-(2n_+n_- - n_+ - n_-)}{(n_++n_-)^2(n_++n_- - 1)}$ . We then compare the observed number of runs ( $R_o$ ) with the expected by calculating a test statistics  $Z = \left| \frac{R_o - R_e \pm 0.5}{\sigma_R} \right|$ , which with  $n_+$  and  $n_-$  both  $> 10$  will be distributed approximately as a standard normal variable, that is,  $Z$  is the number of standard deviations that  $R_e$  and  $R_o$  are apart. The value of  $\pm 0.5$  is a continuity correction to account for biases introduced by approximating a discrete distribution with a continuous one, it is positive when testing for too few runs, and negative when testing for too many runs. Usually, a value of  $Z \geq 1.65 \rightarrow P_0 < 5\%$  is reason for concern.

Serial correlation can be formally detected by the DURBIN-WATSON test, or operationally by plotting all residuals  $r_i$  against the residual  $r_{i+j}$   $j$  data points away from it (lag $_j$ -plot). Usually, the correlation is strongest with small  $j$  and vanishes as  $j$  is increased.

#### F-test

The runs-test described above is a non-parametric test, it is easy to apply but the sensitivity (ability to detect significant deviations) is lower than that of appropriate parametric tests. [12, pp. 286–290] describes such a test with  $H_0$ : the residuals can be explained by the standard deviation of the data points, *versus*  $H_1$ : the residuals are larger than expected given the standard deviation of the data points. The sensitivity of this test is paid for by its laborious nature.

The  $n$  data points occur in  $r$  different values for  $x$ , so that  $n_1 + n_2 + \dots + n_r = n$ . For each distinct  $x_i$ , we can define the  $y$ -average  $\bar{y}_i$ , and we can also define the total average  $\bar{y}$ . Then  $q = q_1 + q_2 = \sum_{i=1}^r \sum_{j=1}^{n_i} (y_{ij} - \hat{y}_i)^2$  can be resolved into two components,  $q_1 = \sum_{i=1}^r n_i(\bar{y}_i - \hat{y}_i)^2$  describes the scatter of the group means around the regression curve with  $v_1 = r - p$  degrees of freedom and  $q_2 = \sum_{i=1}^r \sum_{j=1}^{n_i} (y_{ij} - \bar{y}_i)^2$  the scatter within the groups with  $v_2 = n - r$  degrees of freedom,  $v_q = n - p$ . Then  $v_0 = \frac{q_1/v_1}{q_2/v_2}$  is randomly distributed with  $F(v_1, v_2)$ , and we look for  $(P(v_0 \leq c) = 1 - \alpha)$ , where  $\alpha$  is the desired level

### 13. Regression of curves: The simplex algorithm

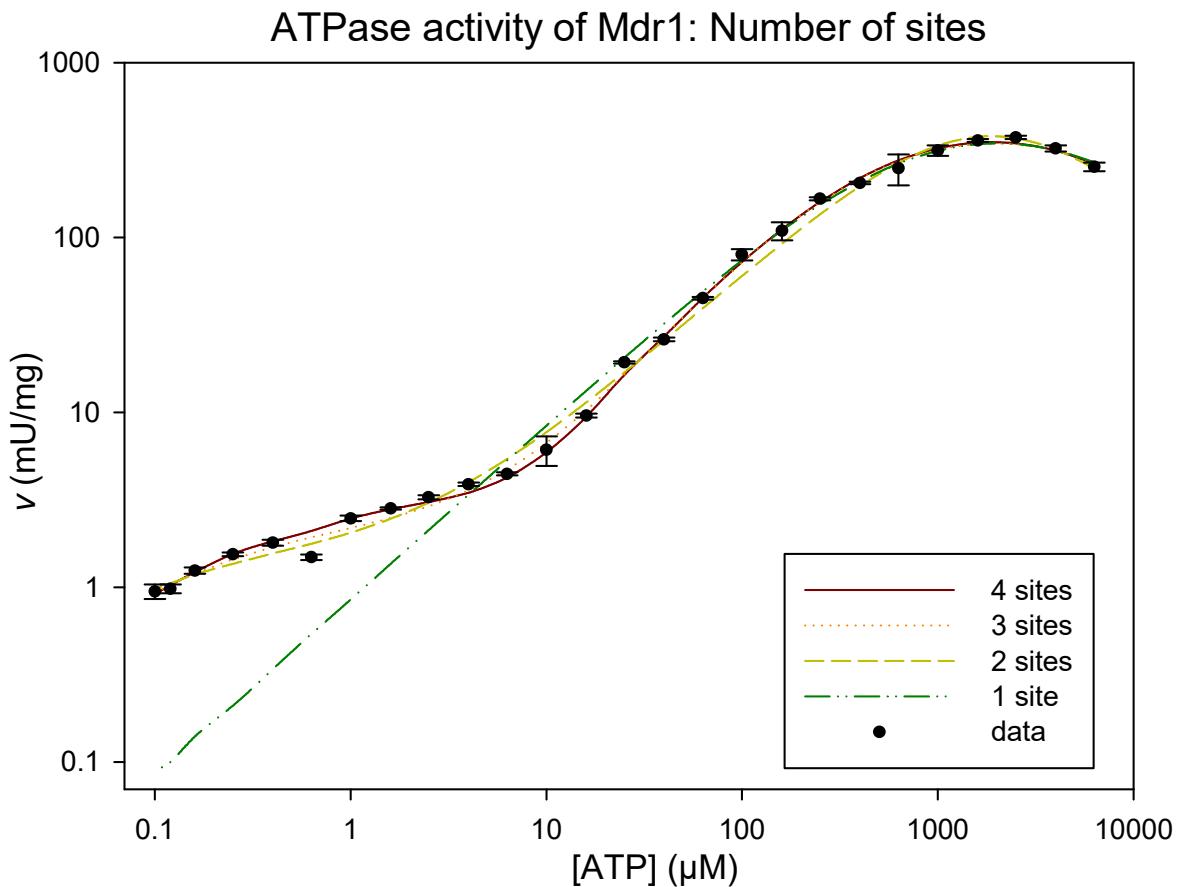


Figure 13.5.: ATPase activity of Mdr1: Number of kinetically different sites. For details see text.

of significance (say, 5 %). If  $v_0 \leq c$  we accept the regression curve as a valid representation of the data, otherwise, we need to look for a better regression equation.

**Example** To determine the number of ATP-binding sites in Mdr1 the substrat/velocity curve was measured (see fig. 13.5). As shown already in fig. 13.3, these data cannot be fitted by the least squares criterium, but the  $\chi^2$ -criterium works very well. Relative errors were used throughout the analysis. Increasing the number of sites (and hence the order of the ratio of polynoms) results in more flexible curves, and hence a better fit to the data. However, this may be caused by a fit to the noise (overfitting). Results for models with 1–4 catalytic binding sites, plus a site for substrate inhibition are summarised in this table,  $v_1 = 24$ ,  $v_2 = 212$ ,  $q_2 = 2079.4$ :

	1 site	2 sites	3 sites	4 sites
$q_1$	54762.6	3894.0	2290.1	603.2
F	232.6	16.5	9.73	1.56
P(0) %	< 0.1	< 0.1	< 0.1	5.2

The F-test shows that curves for 1–3 catalytic binding sites do not explain the data well, we can reject  $H_0$  (the residuals can be explained by the standard deviation of the data) with an error probability of less than 0.1 %. For a model with 4 catalytic sites, on the other hand, the null-hypothesis is accepted at the 5 % level. Thus, Mdr1 must have at least four kinetically different binding sites for ATP. We cannot be sure whether it is only four sites or even more; but we can exclude models with fewer sites.

For the runs test the one-site model gives us  $+ - - - + + - + + + + + - - - - -$ , the two-site model  $- - - + + + - - - - + + + + - - - + -$ , the three-site model  $+ - - - + + + + - - + + + + - - - + - - + -$  and four-site model  $+ - - - + + + + - - + + - - - + + + + + -$ . Thus, we get

	1	2	3	4
$n_+$	10	9	11	12
$n_-$	16	17	15	14
$n$	26	26	26	26
$R_o$	9	9	14	10
$R_e$	13.3	12.8	13.7	11.9
$\sigma_R$	1.42	5.07	5.94	6.16
Z	2.68	0.64	0.16	0.23
$P_0 \%$	0.37	25.1	43.6	40.9

In other words, we can reject the one-site model (even if not at the same high confidence as with the F-test), but we cannot see a significant improvement beyond the 2-site model. This lower sensitivity of the runs test is typical for non-parametric compared to parametric tests. On the other hand, non-parametric tests are a lot less work!

## 13.4. Code for Simplex-Regression

The following program was modified from [3, 4]. The formula compiler was published in [13], error limit determination by Monte-Carlo simulation was described in [6].

Listing 13.1: Simplex

```

1 UNIT SimplexFit;
2
3 {Regressionsanalyse nach M.Caceci, W.P. Cacheris: Fitting Curves to Data,
4 Byte Magazine May 1984, S. 340-350
5 Unter Benutzung des Formel-Compilers aus Pascal International 8/1987, S.
6 52-60
7 Die Bestimmung der Fehlergrenzen durch Monte-Carlo-Simulation ist
8 beschrieben
9 von Straume & Johnson, Meth. Enzymol. 210 (1992) 117-129
10 Die Idee der Minimierung von Chi2 für Daten mit bekannten Fehlergrenzen
11 stammt
12 aus Press et al.: Numerical Recepies in Pascal, Cambridge 1989
13 Gesamtprogramm copyright 1988-1994 by Dr. Engelbert Buxbaum }
```

### 13. Regression of curves: The simplex algorithm

```

11
12 INTERFACE
13
14 USES MathFunc,           // basic maths
15     crt,                 // low level system calls
16     Calc,                // formula compiler
17     Vector,               // vector arithmetic
18     Matrix,               // matrix arithmetic
19     Deskript,              // descriptive statistics
20     Zufall,                // random numbers
21 ;
22
23 CONST
24     SimplexError: BOOLEAN = FALSE;
25
26 PROCEDURE Approximation(Data: MatrixTyp; VAR yBerech: VectorTyp;
27     VAR ProblemName, Formel, xName, yName: STRING);
28
29
30 IMPLEMENTATION
31
32 CONST
33     alfa                  = 1.0;          // Reflektion coefficient
34     beta                  = 0.5;          // Kontraktion coefficient
35     gamma                 = 2.0;          // Expansion coefficient
36     MaxParameter          = 10;
37     MaxVariablen          = 10;
38     MaxN                  = MaxParameter + 1; // dimension of simplex
39     Lw                     = 5;            // linewidth of data field +
40         1
41     Page                  = 12;
42     Ja                     = 'Y';           // key for yes
43     Nein                  = 'N';           // key for no
44
45     var ch : char;          // for error handling
46
47 PROCEDURE Approximation(Data: MatrixTyp; VAR yBerech: VectorTyp;
48     VAR ProblemName, Formel, xName, yName: STRING);
49
50 TYPE
51     AVektor = ARRAY[1..MaxN] OF float;
52     DataRow = ARRAY [1..MaxVariablen] OF float;
53     Simpl   = ARRAY [1..MaxN] OF AVektor;        // Simplex
54     ParFeld = ARRAY [1..MaxParameter] OF STRING[10];
55     VarFeld = ARRAY [1..MaxVariablen] OF STRING[10];

```

```

56  VAR
57      FehlerGrenzen,           // Letzte Spalte der Daten-Matrix
58          Fehlengrenzen?
59
60      done: BOOLEAN;         // Konvergenz
61
62      i, j, n: BYTE;
63
63      h, l: ARRAY [1..MaxN] OF BYTE; // Zahl Hoch/Niedrig Parameter
64
64      Daten,                 // Zahl der Datenpunkte
65      MaxIter,               // max.Zahl der Iterationen
66      NIter: WORD;           // Zahl der Iterationen
67
67      Next,                  // neuer Vertex zu testen
68      center,                // Minimum aller Vertexe}
69      mean, error, MaxErr,   // Maximal zulaessiger Fehler}
70      p, q,                  // um ersten Simplex zu berechnen
71      step: AVektor;         // Eingabe Startschritte
72
72      simp: simpl;          // Simplex
73
73      NameA,                 // Name des Eingabefiles
74      NameB: STRING[64];     // Name des Ausgabefiles
75
75      Eindat,                // Eingabefile
76      Ausdat: TEXT;          // Ausgabefile}
77
77      x: Calc_VarTab;        // FOR formula compiler
78      dummy, Sigma: float;   // y-Standardabweichung
79      Formula: Calc_String;
80      FormProg: Calc_Prog;
81
81      Variablen, Parameter: BYTE;
82
82      ErrorStat,             // TRUE wenn Fehlerstatistik gewuenscht
83      Erster: BOOLEAN;       // Startsimplex nur einmal ausgeben
84
84      Antwort: CHAR;
85
85      Methode: (Summe, Median, ChiSqr); // was soll minimiert werden?
86
86
87      {*****}
88
88      PROCEDURE LiesFunction(VAR x: Calc_VarTab; VAR Formula: Calc_String;
89      VAR FormProg: Calc_Prog);

```

### 13. Regression of curves: The simplex algorithm

```

101
102     VAR
103         i: BYTE;
104         dummy: float;
105         Name: Calc_IdStr;
106         Input : TEXT;
107
108
109     PROCEDURE Hilfe;
110
111     BEGIN
112         Writeln('The function used for fitting to the data should be entered
113             in the ');
114         Writeln('following manner: ');
115         Writeln('1) Number and names of the variables (i. e. measured data)');
116         Writeln('2) Number and name(s) of the parameters');
117         Writeln('3) The formula itself. All names must be entered exactly as
118             defined.');
119         Writeln('    No undefined names are allowed. The formula must end with
120             a ');
121         Writeln('    semicolon.');
122         Writeln;
123         Writeln('The compiler ''knows'' the following constants and and
124             functions, which ');
125         Writeln('can not be redefined:');
126         Writeln('Constants: e, pi                                Basic operators: +,
127             -, *, /, ^');
128         Writeln('Integer: div, mod, ggt, kgv                      Logarithms: ln, lg,
129             ld, exp');
130         Writeln('sin, cos, tan, cot and the equivalent hyperbolic and arcus
131             functions');
132         Writeln('Various Functions: abs, deg, rad, fak, sgn');
133         Writeln;
134     END;
135
136
137     BEGIN
138         Hilfe;
139         Assign(Input, 'CON');
140         Reset(Input);
141         IF SimplexError THEN EXIT;
142         CalcDecMod := TRUE;                               {nur definierte Vars undParms}
143         x := NewVarTab;
144         Writeln('Data file has ', MatrixColumns(Data), 'columns');
145         IF MatrixColumns(Data) > 2
146             THEN

```

```

140   BEGIN
141     Write('Last column independend variable or error margin (v/e):');
142     REPEAT
143       ReadLn(Antwort);
144       Antwort := UpCase(Antwort);
145     UNTIL (Antwort IN ['V', 'F', 'E', #27]);
146     IF Antwort = #27
147     THEN
148       BEGIN
149         SimplexError := TRUE;
150         EXIT;
151       END;
152     FehlerGrenzen := (Antwort = 'F') OR (Antwort = 'E');
153   END
154   ELSE
155     FehlerGrenzen := FALSE;
156   IF FehlerGrenzen
157     THEN Variablen := Pred(MatrixColumns(Data))
158   ELSE Variablen := MatrixColumns(Data);
159   Daten := MatrixRows(Data);
160   FOR i := 1 TO Pred(Variablen) DO
161     BEGIN
162       Write('Name of the ', i, '. (independent) variable: ');
163       ReadLn(Name);
164       dummy := AddToVarTab(x, Name);
165     END;
166   Write('Name of the ', Variablen, '. (dependent) variable: ');
167   ReadLn(Name);
168   dummy := AddToVarTab(x, Name);
169   Write('How many parameters do you want to use [1..', MaxParameter, ']: ');
170   ReadLn(Parameter);
171   N := Parameter + 1;           {Dimensionen des Simplex}
172   FOR i := 1 TO Parameter DO
173     BEGIN
174       Write('Name of the ', i, '. parameter: ');
175       ReadLn(Name);
176       dummy := AddToVarTab(x, Name);
177     END;
178   REPEAT
179   Writeln('Please enter the equation: ');
180   Writeln;
181   Write(x^[Variablen].VarId, ' = ');
182   ReadLn(Formula);
183   CompileExpression(Formula, x, FormProg);
184   IF NOT CalcResult

```

### 13. Regression of curves: The simplex algorithm

```
185      THEN Writeln('Unable to compile the equation, please try again: ');
186      UNTIL CalcResult;
187      Formel := x^[Variablen].VarID + ' = ' + Formula;
188      xName := x^[1].VarID;
189      yName := x^[Variablen].VarID;
190  END;
191
192
193  FUNCTION f(p: AVektor; d: MatrixTyp; Zeile: WORD): float;
194
195  VAR
196    i: BYTE;
197
198  BEGIN
199    FOR i := 1 TO Variablen DO
200      AssignVar(x, x^[i].VarId, GetMatrixElement(d, Zeile, i));
201    FOR i := 1 TO Parameter DO
202      AssignVar(x, x^[Variablen + i].VarId, p[i]);
203    Result := CalcExpression(FormProg, x);
204  END;
205
206
207  PROCEDURE Inparam(VAR MaxIter: WORD; VAR simp: Simpl; VAR Step, MaxErr:
208    AVektor);
209  {Einlesen aller benutzerdefinierten Parameter}
210
211  VAR
212    FalscheEingabe: BOOLEAN;
213    Quelle: CHAR;
214    i: WORD;
215    c: CHAR;
216
217  BEGIN
218    Writeln('This routine calculates curve fits by the simplex algorithm');
219    Writeln;
220    LiesFunction(x, Formula, FormProg);
221    IF SimplexError THEN EXIT;
222    REPEAT
223      FalscheEingabe := FALSE;
224      Writeln;
225      Write('Please enter the name of the output file: ');
226      ReadLn(NameB);
227      Assign(Ausdat, NameB);
228      Rewrite(Ausdat);
229      IF IOResult <> 2
230        THEN  { d. h., Datei existiert schon }
```

```

230      BEGIN
231      REPEAT
232          Write(NameB, ' already exists. Overwrite (y/n): ');
233          ReadLn(c);
234          c := UpCase(c);
235          UNTIL (c = JA) OR (c = Nein) OR (c = #27);
236          IF (c = #27)
237          THEN
238              BEGIN
239                  SimplexError := TRUE;
240                  EXIT;
241              END;
242          FalscheEingabe := (c = Nein);
243      END;
244      UNTIL NOT FalscheEingabe;
245      Rewrite(Ausdat);
246      Write(Ausdat, 'best fit for equation: ');
247      Writeln(Ausdat, x^[Variablen].VarId, ' = ', Formula);
248      Writeln(Ausdat);
249      REPEAT
250          FalscheEingabe := FALSE;
251      REPEAT
252          IF FehlerGrenzen
253          THEN Write('Minimise sum of squares, median of squares or Chi2
254          (S/M/X): ')
255          ELSE Write('Minimise sum or median of squares (S/M): ');
256          ReadLn(c);
257          c := UpCase(c);
258          IF NOT ((c = 'S') OR (c = 'M') OR (c = 'X') OR (c = #27))
259          THEN
260              BEGIN
261                  Sound(400);
262                  Delay(50);
263                  NoSound;
264              END;
265          UNTIL (c = 'S') OR (c = 'M') OR (c = 'X') OR (c = #27);
266          CASE c OF
267              'S': BEGIN
268                  Methode := Summe;
269                  Writeln(Ausdat, 'Minimising sum of squared residuals ');
270              END;
271              'M': BEGIN
272                  Methode := Median;
273                  Writeln(Ausdat, 'Minimising median of squared residuals ');
274              END;
275              'X': BEGIN

```

### 13. Regression of curves: The simplex algorithm

```

275          IF FehlerGrenzen
276          THEN
277          BEGIN
278              Methode := ChiSqr;
279              Writeln(Ausdat, 'Minimizing Chi2');
280          END
281          ELSE
282          BEGIN
283              ch := WriteErrorMessage('Chi2 erfordert Fehlergrenzen
284                                in der letzten Daten-Spalte');
285              SimplexError := TRUE;
286              EXIT;
287          END;
288      #27: BEGIN
289          SimplexError := TRUE;
290          EXIT;
291      END;
292  { case }
293  Writeln(Ausdat);
294  Write(Ausdat, ' ');
295  FOR i := 1 TO Parameter DO
296      Write(Ausdat, x^[i + Variablen].VarId,
297            ': (ValidFigures + 2 - Length(x^[i + Variablen].VarId)));
298 CASE Methode OF
299     Summe: Writeln(Ausdat, 'sum of squares ');
300     Median: Writeln(Ausdat, 'median of squares');
301     ChiSqr: Writeln(Ausdat, 'Chi2');
302 END;
303 UNTIL NOT FalscheEingabe;
304 REPEAT
305     FalscheEingabe := FALSE;
306     Write('Please enter maximal number of iterations: ');
307     ReadLn(MaxIter);
308 UNTIL NOT FalscheEingabe;
309 REPEAT
310     FalscheEingabe := FALSE;
311     Writeln('Please enter initial guesses for all parameters: ');
312     Write(Ausdat, 'Start coordinates: ');
313     FOR i := 1 TO Parameter DO
314         BEGIN
315             Write(x^[i + Variablen].VarId, ' = ');
316             ReadLn(simp[1, i]);
317             IF (i MOD lw) = 0 THEN Writeln(Ausdat);
318             Write(Ausdat, FloatStr(simp[1, i], ValidFigures), ' ');
319         END;

```

```

320     Writeln(Ausdat);
321     Writeln(Ausdat);
322 UNTIL NOT FalscheEingabe;
323 REPEAT
324     FalscheEingabe := FALSE;
325     Writeln('Please enter starting step width for all parameters ');
326     Writeln('(ca. 1/10 to 1/2 of initial value)');
327     Write(Ausdat, 'Starting step width:    ');
328     FOR i := 1 TO Parameter DO
329         BEGIN
330             Write('Step width:    ', x^[i + Variablen].VarId, ' = ');
331             ReadLn(step[i]);
332             IF (i MOD lw) = 0 THEN Writeln(Ausdat);
333             Write(Ausdat, FloatStr(step[i], ValidFigures), ' ');
334         END;
335     Writeln(Ausdat);
336     Writeln(Ausdat);
337     Write(Ausdat, 'max. allowable error:    ');
338     FOR i := 1 TO n DO
339         BEGIN
340             MaxErr[i] := MaxError;
341             IF (i MOD lw) = 0 THEN Writeln(Ausdat);
342             Write(Ausdat, FloatStr(MaxErr[i], ValidFigures), ' ');
343         END;
344     Writeln(Ausdat);
345     Writeln(Ausdat);
346 UNTIL NOT FalscheEingabe;
347 REPEAT
348     Write('Do you want error margins for the parameters (y/n): ');
349     ReadLn(c);
350     c := UpCase(c);
351 UNTIL (c = JA) OR (c = Nein) OR (c = #27);
352 IF (c = #27)
353 THEN
354     BEGIN
355         SimplexError := TRUE;
356         EXIT;
357     END;
358     Writeln(UpCase(c));
359     ErrorStat := (c = JA);
360 END;
361
362
363 PROCEDURE sum_of_residuals(VAR z: AVektor; Data: MatrixTyp);
364 {Berechnet die Summe der Fehlerquadrate}
365

```

### 13. Regression of curves: The simplex algorithm

```
366  VAR
367      i: WORD;
368
369  BEGIN
370      z[n] := 0.0;
371      FOR i := 1 TO Daten DO
372          z[n] := z[n] + Sqr(f(z, Data, i) - GetMatrixElement(Data, i,
373                          Variablen));
374  END;
375
376  PROCEDURE Median_Of_Squares(VAR z: AVektor; Data: MatrixTyp);
377  { berechnet den Median der Fehlerquadrate }
378
379  VAR
380      i: WORD;
381      Residuals: VectorTyp;
382
383  BEGIN
384      CreateVector(Residuals, Daten, 0.0);
385      FOR i := 1 TO Daten DO
386          SetVectorElement(Residuals, i, Sqr(f(z, Data, i) -
387              GetMatrixElement(Data, i, Variablen)));
388      ShellSort(Residuals);
389      z[n] := Quantile(Residuals, 0.5);
390      DestroyVector(Residuals);
391  END;
392
393  PROCEDURE Chi(VAR z: AVektor; Data: MatrixTyp);
394  {Berechnet chi2}
395
396  VAR
397      i: WORD;
398
399  BEGIN
400      z[n] := 0.0;
401      FOR i := 1 TO Daten DO
402          z[n] := z[n] + Sqr(f(z, Data, i) - GetMatrixElement(Data, i,
403                          Variablen)) /
404              GetMatrixElement(Data, i, Succ(Variablen)));
405  END;
406
407  PROCEDURE report;
408  {berichtet Programstatistik}
409
```

```

410  VAR
411    dy, h, Zaehler, Nenner, Fehler, Mittel, rSqr: float;
412    d1, d2: TEXT;
413    HilfsStr: STRING[14];
414    i, j: WORD;
415
416  BEGIN
417    Writeln(Ausdat);
418    Writeln(Ausdat, 'Routine was left after ', NIter: 5, ' iterations');
419    Writeln(Ausdat);
420    Writeln(Ausdat);
421    Writeln(Ausdat, 'The final simplex is: ');
422    Write(Ausdat, ' ');
423    FOR j := 1 TO n DO
424      BEGIN
425        FOR i := 1 TO n DO
426          BEGIN
427            IF (i MOD lw) = 0 THEN
428              Writeln(Ausdat);
429              Write(Ausdat, FloatStr(simp[j, i], ValidFigures), ' ');
430            END;
431            Writeln(Ausdat);
432            Write(Ausdat, ' ');
433          END;
434          Writeln(Ausdat);
435          Write(Ausdat, 'The mean is: ');
436          FOR i := 1 TO n DO
437            BEGIN
438              IF (i MOD lw) = 0 THEN
439                Writeln(Ausdat);
440                Write(Ausdat, FloatStr(mean[i], ValidFigures), ' ');
441              END;
442              Writeln(Ausdat);
443              Writeln(Ausdat);
444              Write(Ausdat, 'error: ');
445              FOR i := 1 TO n DO
446                BEGIN
447                  IF (i MOD lw) = 0 THEN
448                    Writeln(Ausdat);
449                    Write(Ausdat, FloatStr(error[i], ValidFigures), ' ');
450                  END;
451                  Writeln(Ausdat);
452                  Writeln(Ausdat);
453                  Write(Ausdat, ' # ');
454                  FOR i := 1 TO Variablen DO

```

### 13. Regression of curves: The simplex algorithm

```

455     Write(Ausdat, x^[i].VarId, ' ': (ValidFigures + 2 -
456                                         Length(x^[i].VarId)));
457 IF Fehlergrenzen THEN
458     Write(Ausdat, 'Delta ', x^[Variablen].VarId, ' ': (ValidFigures -
459                                         2 - Length(x^[i].VarId)));
460     Write(Ausdat, x^[Variablen].VarId, ' ber.',
461           ' ': (ValidFigures - 2 - Length(x^[Variablen].VarId)));
462     Writeln(Ausdat, 'error:           ');
463     Zaehler := 0;
464     sigma := 0.0;
465 FOR i := 1 TO Daten DO
466 BEGIN
467     h := f(mean, Data, i);
468     SetVectorElement(yBerech, i, h);
469     dy := GetMatrixElement(Data, i, Variablen) - h;
470     sigma := sigma + Sqr(dy);
471     Write(Ausdat, i: 4, ' ');
472     FOR j := 1 TO MatrixColumns(Data) DO
473         Write(Ausdat, FloatStr(GetMatrixElement(Data, i, j),
474                               ValidFigures), ' ');
475     Writeln(Ausdat, FloatStr(h, ValidFigures), ' ', FloatStr(dy,
476                               ValidFigures));
477     Zaehler := Zaehler + GetMatrixElement(Data, i, Variablen);
478 END;
479 Writeln(Ausdat);
480 sigma := Sqrt(sigma / Daten);
481 Writeln(Ausdat, 'The standard deviation is:   ', FloatStr(sigma,
482               ValidFigures));
483 Fehler := sigma / Sqrt(Daten - Parameter);
484 Writeln(Ausdat, 'The error of the function is: ', FloatStr(Fehler,
485               ValidFigures));
486 Mittel := Zaehler / Daten;
487 Zaehler := 0;
488 Nenner := 0;
489 FOR i := 1 TO Daten DO
490 BEGIN
491     Zaehler := Zaehler + Sqr(GetVectorElement(yBerech, i) - Mittel);
492     Nenner := Nenner + Sqr(GetMatrixElement(Data, i, Variablen) -
493                             Mittel);
494 END;
495 rSqr := Zaehler / Nenner;
496 Writeln(Ausdat, 'r2:           ', FloatStr(rSqr,
497               ValidFigures));
498 END;

```

```

494 PROCEDURE First;
495
496 VAR
497   i, j: WORD;
498
499 BEGIN
500   Write(Ausdat, 'Start simplex ');
501   FOR j := 1 TO n DO
502     BEGIN
503       Write(Ausdat, ' simp[' , j: 3, ']');
504       FOR i := 1 TO n DO
505         BEGIN
506           IF (i MOD lw) = 0 THEN Writeln(Ausdat);
507           Write(Ausdat, FloatStr(simp[j, i], ValidFigures), ' ');
508         END;
509         Writeln(Ausdat);
510         Write(Ausdat, '          ');
511       END;
512     Writeln(Ausdat);
513   END;
514
515
516 PROCEDURE new_vertex;
517 {ersetzt worst durch next}
518
519 VAR
520   i, j: WORD;
521
522 BEGIN
523   IF erster THEN Write(' --- ', NIter: 4);
524   FOR i := 1 TO n DO
525     BEGIN
526       simp[h[n], i] := Next[i];
527       IF erster THEN Write(FloatStr(Next[i], ValidFigures));
528     END;
529   IF erster THEN Writeln;
530 END;
531
532
533 PROCEDURE order;
534 {Highs und Lows fuer jeden Parameter}
535
536 VAR
537   i, j: BYTE;
538
539 BEGIN

```

### 13. Regression of curves: The simplex algorithm

```

540     FOR j := 1 TO n DO
541         BEGIN
542             FOR i := 1 TO n DO
543                 BEGIN
544                     IF simp[i, j] < simp[l[j], j] THEN l[j] := i;
545                     IF simp[i, j] > simp[h[j], j] THEN h[j] := i;
546                 END;
547             END;
548         END;
549
550
551     PROCEDURE Iteration(Data: MatrixTyp);
552
553     VAR
554         i, j: WORD;
555
556     BEGIN
557         NIter := 0;
558         REPEAT
559             done := TRUE;
560             NIter := Succ(NIter);
561             FOR i := 1 TO n DO center[i] := 0.0;
562             FOR i := 1 TO n DO
563                 IF i <> h[n]
564                 THEN
565                     FOR j := 1 TO Parameter DO
566                         center[j] := center[j] + simp[i, j];
567             FOR i := 1 TO n DO
568                 BEGIN
569                     center[i] := center[i] / Parameter;
570                     Next[i] := (1.0 + alfa) * center[i] - alfa * simp[h[n], i];
571                 END;
572                 CASE Methode OF
573                     Summe: sum_of_residuals(Next, Data);
574                     Median: Median_Of_Squares(Next, Data);
575                     ChiSqr: Chi(Next, Data);
576                 END;
577                 IF Next[n] <= simp[l[n], n]
578                 THEN
579                     BEGIN
580                         new_vertex;
581                         FOR i := 1 TO n DO
582                             Next[i] := gamma * simp[h[n], i] + (1.0 - gamma) * center[i];
583                         CASE Methode OF
584                             Summe: sum_of_residuals(Next, Data);
585                             Median: Median_Of_Squares(Next, Data);

```

```

586     ChiSqr: Chi(Next, Data);
587   END;
588   IF Next[n] <= simp[l[n], n] THEN new_vertex;
589 END
590 ELSE
591 BEGIN
592   IF Next[n] <= simp[h[n], n]
593   THEN
594     new_vertex
595   ELSE
596     BEGIN
597       FOR i := 1 TO Parameter DO
598         Next[i] := beta * simp[h[n], i] + (1.0 - beta) *
599                     center[i];
600       CASE Methode OF
601         Summe: sum_of_residuals(Next, Data);
602         Median: Median_Of_Squares(Next, Data);
603         ChiSqr: Chi(Next, Data);
604       END;
605       IF Next[n] <= simp[h[n], n]
606       THEN
607         new_vertex
608       ELSE
609         BEGIN
610           FOR j := 1 TO Parameter DO
611             BEGIN
612               simp[i, j] := (simp[i, j] + simp[l[n], j]) *
613                             beta;
614               CASE Methode OF
615                 Summe: sum_of_residuals(simp[i], Data);
616                 Median: Median_Of_Squares(simp[i], Data);
617                 ChiSqr: Chi(simp[i], Data);
618               END; // case
619             END; // for j
620           END; // else
621         END; // else
622       END; //else
623     order;
624     FOR j := 1 TO n DO
625       BEGIN
626         error[j] := (simp[h[j], j] - simp[l[j], j]) / simp[h[j], j];
627         IF done
628           THEN IF error[j] > MaxErr[j]
629             THEN done := FALSE;
630       END;
631     UNTIL (done OR (NIter = MaxIter));

```

### 13. Regression of curves: The simplex algorithm

```

630   END;
631
632
633 PROCEDURE DoIteration(VAR Simp: Simpl; Data: MatrixTyp; VAR Mean:
634   AVektor);
635
636   VAR
637     i, j: WORD;
638
639   BEGIN
640     CASE Methode OF
641       Summe: sum_of_residuals(simp[1], Data);
642       Median: Median_Of_Squares(simp[1], Data);
643       ChiSqr: Chi(simp[1], Data);
644     END;
645     FOR i := 1 TO Parameter DO
646       BEGIN
647         p[i] := step[i] * (Sqrt(n) + Parameter - 1) / (Parameter * Sqrt(2));
648         q[i] := step[i] * (Sqrt(n) - 1) / (Parameter * Sqrt(2));
649       END;
650     FOR i := 2 TO n DO
651       BEGIN
652         FOR j := 1 TO Parameter DO simp[i, j] := simp[1, j] + q[j];
653         simp[i, i - 1] := simp[1, i - 1] + p[i - 1];
654         CASE Methode OF
655           Summe: sum_of_residuals(simp[i], Data);
656           Median: Median_Of_Squares(simp[i], Data);
657           ChiSqr: Chi(simp[i], Data);
658         END;
659       END;
660     FOR i := 1 TO n DO
661       BEGIN
662         l[i] := 1;
663         h[i] := 1;
664       END;
665     order;
666     IF Erster THEN First;
667     Iteration(Data);
668     FOR i := 1 TO n DO
669       BEGIN
670         mean[i] := 0.0;
671         FOR j := 1 TO n DO
672           mean[i] := mean[i] + simp[j, i];
673         mean[i] := mean[i] / n;
674       END;
675     END;

```

```

675
676
677 PROCEDURE CalculateErrorMargins;
678
679 CONST
680     Anzahl = 100;
681
682 VAR
683     Counter, Differenz, Ergebniss: AVektor;
684     SimulatedData, Ergebnisse: MatrixTyp;
685     i, j, k: WORD;
686
687
688 PROCEDURE Simulate(VAR SimulatedData: MatrixTyp; yBerech: VectorTyp;
689                     Sigma: float);
690
691 VAR
692     i: WORD;
693     Wert: float;
694
695 BEGIN
696     FOR i := 1 TO MatrixColumns(SimulatedData) DO
697         BEGIN
698             Wert := RandomLaplace(GetVectorElement(yBerech, i), Sigma);
699             SetMatrixElement(SimulatedData, i, Variablen, Wert);
700         END;
701     END;
702
703 BEGIN
704     Sigma := Sqr(Sigma);
705     CopyMatrix(Data, SimulatedData);
706     CreateMatrix(Ergebnisse, Anzahl, N, 0.0);
707     FOR j := 1 TO N DO
708         Counter[j] := 0.0;
709     FOR i := 1 TO 100 DO
710         BEGIN
711             Write(i: 3, ' ');
712             Simulate(SimulatedData, yBerech, Sigma);
713             DoIteration(Simp, SimulatedData, Ergebniss);
714             FOR j := 1 TO N DO
715                 BEGIN
716                     SetMatrixElement(Ergebnisse, i, j, Ergebniss[j]);
717                     Counter[j] := Counter[j] + Ergebniss[j];
718                     Write(FloatStr(Ergebniss[j], ValidFigures), ' ');
719                 END;
720             Writeln;

```

### 13. Regression of curves: The simplex algorithm

```

721      END;
722      DestroyMatrix(SimulatedData);
723      FOR j := 1 TO N DO
724          BEGIN
725              Counter[j] := Counter[j] / Anzahl;           { Mittelwerte der
726                  Parameter }
727              Differenz[j] := 0.0;
728          END;
729          FOR i := 1 TO Anzahl DO
730              FOR j := 1 TO N DO
731                  Differenz[j] := Sqr(Counter[j] - GetMatrixElement(Ergebnisse, i,
732                      j));
733                  Writeln(Ausdat);
734                  Writeln(Ausdat, 'Standard deviation of the parameters: ');
735                  Write(Ausdat, ' ');
736                  FOR j := 1 TO N DO
737                      BEGIN
738                          Differenz[j] := Sqrt(Differenz[j] / Pred(Anzahl));
739                          IF (j MOD lw) = 0 THEN Writeln(Ausdat);
740                          Write(Ausdat, FloatStr(Sqrt(Differenz[j]), ValidFigures), ' ');
741                      END;
742                      Writeln(Ausdat);
743                  END;
744
745      BEGIN {Approximation}
746          Erster := TRUE;
747          Inparam(MaxIter, simp, step, MaxErr);
748          IF SimplexError THEN EXIT;
749          DoIteration(Simp, Data, Mean);
750          CreateVector(yBerech, Daten, 0.0);
751          report;
752          Erster := FALSE;
753          IF ErrorStat THEN CalculateErrorMargins;
754          Close(Ausdat);
755      END;
756  END.

```

Listing 13.2: Test program

```

1  program Simplex;
2
3
4  USES
5      MathFunc,           // basic math functions
6      Vector,            // vector arithmetic
7      Matrix,             // matrix arithmetic

```

```

8   Zufall,           // random numbers
9   SimplexFit        // curve fit by simplex
10  ;
11
12 const MaxData = 100;
13
14 VAR Data           : MatrixTyp;
15   Calculated       : VectorTyp;
16   ProblemName, Formel, xName, yName : STRING;
17
18 { **** ***** **** ***** **** ***** **** ***** }
19
20 PROCEDURE CreateDataSet (VAR Data : MatrixTyp);
21
22 VAR i : WORD;
23   x, y : float;
24
25 BEGIN
26   FOR i := 1 TO MatrixRows(Data) DO
27     BEGIN
28       x := i/10 ;
29       SetMatrixElement(Data, i, 1, x);
30       y := x / (1+x);           // normalised Henri-Michaelis-Menten
31           law
32       y := y + RandomNormal(0, 0.1); // add normal-distributed random
33           noise
34       SetMatrixElement(Data, i, 2, y);
35     END;
36
37 BEGIN {Hauptprogram}
38   CreateMatrix(Data, MaxData, 2, 0.0);
39   CreateVector(Calculated, MaxData, 0.0);
40   CreateDataSet(Data);
41   Approximation(Data, Calculated, ProblemName, Formel, xName, yName);
42   DestroyMatrix(Data);
43   DestroyVector(Calculated);
44 END.

```

## References

- [1] K. LEVENBERG: A method for the solution of certain non-linear problems in least squares, *Quart. Appl. Math.* **2**:2 (1944), 164–168 URL: <https://www.ams.org/qam/1944-02-02/S0033-569X-1944-10666-0/S0033-569X-1944-10666-0.pdf>.

- [2] D. W. MARQUARDT: An algorithm for least-squares estimation of nonlinear parameters, *J. Soc. Industrial Appl. Math.* **11**:2 (1963), 431–441 URL: [http://137.204.42.130/~bittelli/materiale%5C\\_lettura%5C\\_fisica%5C\\_terreno/marquardt%5C\\_63.pdf](http://137.204.42.130/~bittelli/materiale%5C_lettura%5C_fisica%5C_terreno/marquardt%5C_63.pdf).
- [3] J. A. NELDER, R. MEAD: A simplex method for function minimization, *Computer J.* **7**:4 (1965), 308–313 URL: <http://www.ii.uib.no/~lennart/drgrad/Nelder1965.pdf>.
- [4] M. S. CACECI, W. P. CACHERIS: Fitting Curves to Data: The Simplex Algorithm is the Answer, *Byte* **9**:5 (1984), 340–362 URL: [https://www.researchgate.net/publication/246199710\\_Fitting\\_curves\\_to\\_data\\_The\\_simplex\\_algorithm\\_is\\_the\\_answer](https://www.researchgate.net/publication/246199710_Fitting_curves_to_data_The_simplex_algorithm_is_the_answer).
- [5] Y.-S. KIM: Refined Simplex Method for Data Fitting, In: *Astronomical Data Analysis Software and Systems VI* G. HUNT, H.E. PAYNE (editor) vol. 125 ASP Conference Series 1997, 206–209 URL: <https://www.cv.nrao.edu/adass/adassVI/kimys.html>.
- [6] M. STRAUME, M. L. JOHNSON: Monte Carlo Method for Determining Complete Confidence Probability Distributions of Estimated Modell Parameters, *Meth. Enzymol.* **210** (1992), 117–129 DOI: [10.1016/0076-6879\(92\)10009-3](https://doi.org/10.1016/0076-6879(92)10009-3).
- [7] M. STRAUME, M. L. JOHNSON: Monte Carlo Method for Determining Complete Confidence Probability Distributions of Estimated Model Parameters, In: *Essential numerical computer methods* M.L. JOHNSON (editor) Reliable lab solutions Amsterdam et al.: Elsevier, 2010 chap. 4, 55–66 ISBN: 9780123849977 DOI: [10.1016/B978-0-12-384997-7.00004-2](https://doi.org/10.1016/B978-0-12-384997-7.00004-2).
- [8] R. J. RITCHIE, T. PRVAN: A simulation studie on designing experiments to measure the  $K_m$  of Michaelis-Menten kinetics curves, *J. theor. Biol.* **178** (1996), 239–254 DOI: [10.1006/jtbi.1996.0023](https://doi.org/10.1006/jtbi.1996.0023).
- [9] E. BUxBAUM: Co-operating ATP sites in the multiple drug resistance transporter Mdr1, *Eur. J. Biochem.* **265**:1 (1999), 54–63 DOI: [10.1046/j.1432-1327.1999.00643.x](https://doi.org/10.1046/j.1432-1327.1999.00643.x).
- [10] G. HEINZEL: Beliebig genau: Moderne Runge-Kutta-Verfahren zur Lösung von Differentialgleichungen, *c't*:8 (1992), 192–185.
- [11] G.W. WATTS: Parameter Estimates from Nonlinear Models, In: *Essential numerical computer methods* M.L. JOHNSON (editor) Reliable lab solutions Amsterdam: Elsevier, 2010 chap. 2, 23–36 ISBN: 9780123849977 DOI: [10.1016/B978-0-12-384997-7.00002-9](https://doi.org/10.1016/B978-0-12-384997-7.00002-9).
- [12] E. KREYSZIG: *Statistische Methoden und ihre Anwendungen* 7th ed. Göttingen: Vandenhoeck & Ruprecht, 1979 ISBN: 9783525407172.
- [13] K. GIESELMANN, M. CEOL: CALC – ein mathematischer Compiler, *PASCAL Int.8* (1987), 52–60.
- [14] M.L. JOHNSON, ed.: *Essential numerical computer methods* Reliable lab solutions Amsterdam: Elsevier, 2010 ISBN: 9780123849977.

# 14. Circular data

## Abstract

There are data whose scale is not on a number ray, but on a circle. Examples include angles on the windrose, hour of day and season of year. Special statistical tools are required to describe such data because of cross-over problems.

Consider, for example, measurements of flight direction of birds enroute to their summer quarters. These will scatter around 0 (northwards, at least on the northern hemisphere). For example, we have two measurements, 5 and 355. If we try to calculate the arithmetic mean, we get  $\frac{5+355}{2} = 180$ , that is due south! Obviously, this mean is not a good measure of position of the data set. What is the problem? We conventionally assign 0 to north, and go clockwise from there. 360 is the same as 0 again. Similarly, we start the day at 00:00:00 (midnight), and after 23:59:59 we end at 00:00:00 again. A maternity ward may be interested in the time of day when most deliveries occur, so they can provide the required resources (full circle = 24 h = 1440 min). On the other hand, a conservation biologist may have the same question about the time of year when birthing activities occur (full circle = 1 a = 365 d), so that appropriate protective measures can be instigated (*i.e.*, road closures).

Thus, the data that are not on a number ray, but on a circle, require special methods for statistical description to avoid cross-over problems [1–3]. Such data may be described by the von Mises **distribution** [4], the circular equivalent of the normal distribution. It is characterised by two parameters, the position  $\mu$  and the dispersion  $\kappa$ . The density function is

$$f(\bar{x}|\mu, \kappa) = \frac{1}{2\pi I_0(\kappa)} e^{\kappa \cos(\theta - \bar{\theta})}, \bar{\theta} \in [0 \dots 2\pi], \kappa > 0 \quad (14.1)$$

where  $I_0(\kappa) = \frac{1}{2\pi} \int_0^{2\pi} \exp(\kappa \cos(\theta - \bar{\theta})) d\theta$  the modified BESSEL-function of 0th order. Here, as in the rest of this chapter, angles are in the unit rad,  $[0 \dots 2\pi]$ .

## 14.1. The interface

The interface is:

Listing 14.1: Interface of unit Circular

```
1 UNIT Circular;  
2
```

## 14. Circular data

```
3  INTERFACE
4
5  USES math,           // Free Pascal math UNIT
6    crt,              // Free pascal UNIT
7    MathFunc,          // basic math functions
8    Dynam,             // dynamic data structures
9    Complex,            // Complex numbers
10   Vector,             // vector arithmetic
11   Matrix,             // matrix algebra
12   Correlations,        // correlation coefficients
13   Deskript,            // descriptive statistics
14   Stat,               // statistical distributions
15   Zufall;             // pseudo-Random numbers
16
17 CONST CircleError : BOOLEAN = FALSE;
18     Ninety = Const_pi / 2;           // °90 in rad
19
20 { ***** description of a single data vector ***** }
21
22 PROCEDURE Transform (VAR Data : VectorTyp; FullCircle : float);
23
24 FUNCTION MedianDirection (TransformedData : VectorTyp) : float;
25
26 FUNCTION MeanVector (TransformedData : VectorTyp; p : word) : ComplexTyp;
27
28 FUNCTION CircularVariance (R : float) : float;
29
30 FUNCTION CircularStandardDeviation (R : float) : float;
31
32 FUNCTION CircularDispersion (TransformedData : VectorTyp; Mean :
33   ComplexTyp) : float;
34
35 FUNCTION Kappa (R : float; n : WORD) : float;
36
37 FUNCTION TrigonometricMoment (Data : VectorTyp; MeanAngle : float; p :
38   WORD) : ComplexTyp;
39
40 function CenteredMean(Moment : ComplexTyp) : ComplexTyp;
41
42 FUNCTION CircularSkew (TransformedData : VectorTyp; Mean : ComplexTyp) :
43   float;
44 FUNCTION CenteredCircularSkew (Mean1, Mean2 : ComplexTyp) : float;
45
46 FUNCTION CircularKurtosis (TransformedData : VectorTyp; Mean : ComplexTyp)
47   : float;
```

```

45
46 FUNCTION CenteredCircularKurtosis (Mean1, Mean2 : ComplexTyp) : float;
47
48 FUNCTION ConfidenceInterval (R, delta : float; n : WORD) : float;
49
50 { ***** random numbers ***** }
51
52 FUNCTION RandomVonMieses(mu, kappa : float) : float;
53
54 FUNCTION RandomUniformCircular : float;
55
56 { ***** Tests for preferred direction ***** }
57
58 FUNCTION Rayleigh (MeanVectorLength : float; n : WORD) : float;
59
60 FUNCTION HodgesAjne (TransformedData : VectorTyp; MeanAngle : float) :
61   float;
62
63 PROCEDURE ChiSqrTest (Data : VectorTyp; Direction : float;
64   Sectors : WORD; VAR ChiSqr : float; VAR dgf : WORD);
65
66 FUNCTION HomewardComponent (n : WORD; Mean : ComplexTyp;
67   Expected : float) : float;
68
69 FUNCTION Rao (TransformedData : VectorTyp) : float;
70
71 PROCEDURE Kuipers (Data : MatrixTyp; VAR K, U : float);
72
73 PROCEDURE CalculateCumulativeFrequencies (Data : VectorTyp;
74   FullCircle : float; VAR Result : MatrixTyp);
75
76 FUNCTION OneSample (MeanAngle, R, delta, TestAngle : float; n : WORD) :
77   BOOLEAN;
78
79 { ***** grouped data ***** }
80
81
82 { ***** Compare two circular distributions ***** }
83
84
85 PROCEDURE Difference (Data1, Data2 : VectorTyp;
86   VAR KuipersV, WatsonsUSqr : float);
87
88 FUNCTION FTest (Data1, Data2 : VectorTyp) : float;

```

## 14. Circular data

```
89
90 FUNCTION Wilcoxon (Data1, Data2 : VectorTyp;
91                 Direction1, Direction2 : float) : float;
92
93 PROCEDURE WatsonWilliams (TransformedData1, TransformedData2 : VectorTyp);
94
95 FUNCTION KruskalWallis (TransformedData1, TransformedData2 : VectorTyp) :
96     float;
97 { ***** linear-bivariate Data ***** }
98
99 PROCEDURE DescribeBivariat (Data : MatrixTyp);
100
101 { ***** linear dependent, circular independent Correlation ***** }
102
103 FUNCTION CAssociation (CONST theta, Y : VectorTyp; VAR Sig :
104     SignificanceType) : float;
105
106 PROCEDURE CircularLinearCorrelation (const TransformedData, yData:
107     VectorTyp;
108
109             VAR Correlation : float; VAR Sig :
110                 SignificanceType);
111
112 PROCEDURE LinearPeriodicRankCorrelation (const TransformedData, yData:
113     VectorTyp;
114
115             VAR Correlation, U : float);
116
117 PROCEDURE TrigonometricPolynomial (const TransformedData, yData: VectorTyp;
118     Periode : float);
119
120 { ***** Correlation circular dependent and independent variable ***** }
121
122 PROCEDURE PeriodicRankCorrelation (CONST alpha, beta : VectorTyp;
123
124             VAR rPlus, rMinus : float);
125
126 PROCEDURE CircularCircularCorrelation (Alpha, Beta : VectorTyp;
127
128             VAR Correlation : float; var Sig : SignificanceType);
129
130 { ***** }
131
132 IMPLEMENTATION
133
134
135 VAR xMax : WORD;
136     ch : CHAR;
```

## 14.2. Transformation of circular data

The following routine ensures that the value is in the range of a full circle,  $[0 \dots 2\pi]$ .

Listing 14.2: limit datum to  $0..2\pi$

```

1 FUNCTION ModuloTwoPi (Datum : float) : float;
2
3 VAR x : float;
4
5 BEGIN
6   x := Datum;
7   WHILE (x > Const_2pi) DO x := x - Const_2pi;
8   WHILE (x < 0.0) DO x := x + Const_2pi;
9   Result := x;
10 END;

```

The following routine transforms the data in a data vector to the range of a full circle,  $[0 \dots 2\pi]$ . The variable **FullCircle** contains the length to which the data are standardised (say, 24 h for a day).

Listing 14.3: Transformation to  $0..2\pi$

```

1 PROCEDURE Transform (VAR Data : VectorTyp; FullCircle : float);
2
3 VAR n, i : WORD;
4   Datum : float;
5
6 BEGIN
7   n := VectorLength(Data);
8   FOR i := 1 TO n DO
9     BEGIN
10    Datum := GetVectorElement(Data, i) * Const_2pi / FullCircle;
11    SetVectorElement(Data, i, ModuloTwoPi(Datum));
12   END;
13 END;

```

## 14.3. Description of a single vector of circular data

### 14.3.1. Position

The data are plotted as unit vectors from the centre of a unit circle, with the angle identifying the direction of the vector (see fig. 14.1). The mean vector is then determined

14. Circular data

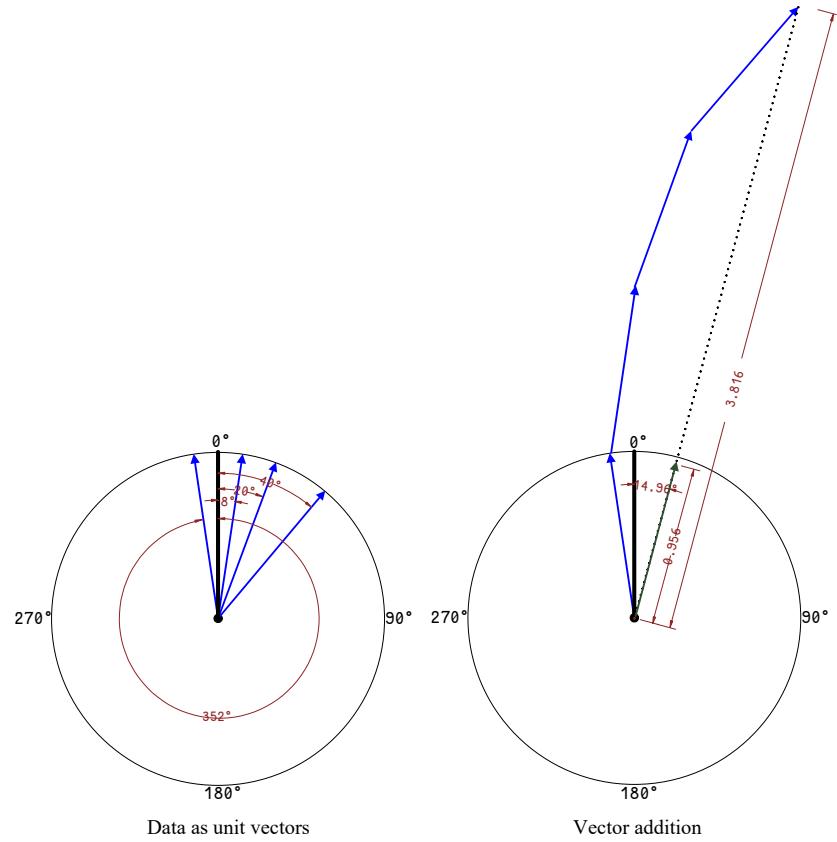


Figure 14.1.: Circular mean by vector addition. For details see text.

by vector addition of the data vectors.

$$\begin{aligned}
 C_p &= \sum_{i=1}^n \cos(p\theta_i) \\
 S_p &= \sum_{i=1}^n \sin(p\theta_i) \\
 R_p &= \sqrt{C_p^2 + S_p^2}, \quad [0 \dots n] \\
 \bar{R}_p &= R_p/n, \quad [0 \dots 1] \\
 \bar{\theta}_p &= \sin^{-1}(S_p/R_p) = \cos^{-1}(C_p/R_p) = \begin{cases} \cos^{-1}(C_p/R_p) = \cos^{-1}(0) = \pi/2 & C_p = 0 \\ \tan^{-1}(S_p/C_p) + \pi & C_p < 0 \\ \tan^{-1}(S_p/C_p) & S_p > 0, C_p > 0 \\ \tan^{-1}(S_p/C_p) + 2\pi & S_p < 0, C_p > 0 \end{cases}
 \end{aligned}$$

where  $\bar{\theta}_p$  is the mean vector direction and  $\bar{R}_p$  is the resultant length with respect to a positive whole number  $p$ . For the simple mean,  $p = 1$ , but higher values are required for some of the following. The sample statistics  $\bar{\theta}_1$  and  $\bar{R}_1$  (usually written  $\bar{\theta}, \bar{R}$ ) estimate the population parameters  $\mu$  and  $\rho$ .

Listing 14.4: Arithmetic mean of circular data

```

1  FUNCTION MeanVector (TransformedData : VectorTyp; p : word) : ComplexTyp;
2
3  VAR C, S, R, theta, Datum : float;
4      n, i : WORD;
5
6  BEGIN
7      n := VectorLength(TransformedData);
8      C := 0;
9      S := 0;
10     FOR i := 1 TO n DO
11         BEGIN
12             Datum := GetVectorElement(TransformedData, i);
13             C := C + Cos(p * Datum);
14             S := S + Sin(p * Datum);
15         END;
16     R := Sqrt(C*C + S*S) / n;
17     CASE signum(C) OF
18         0 : theta := Const_pi / 2; // arccos(C/R) = arccos(0)
19         -1 : theta := ArcTan(S/C) + Const_pi;
20         1 : CASE signum(S) OF
21             1 : theta := ArcTan(S/C);
22             -1 : theta := ArcTan(S/C) + Const_2pi;
23             0 : theta := 0;
24         END;

```

## 14. Circular data

```

25   END;
26   Result := ComplexInit(R, theta); // convert to complex number
27   END;

```

### 14.3.2. Spread

The sample circular variance  $V$  and the sample circular standard deviation  $v$  are calculated as follows:

$$V = 1 - \bar{R}_1, \quad [0 \dots 1] \quad (14.2)$$

$$v = \sqrt{-2 \ln(1 - V)} = \sqrt{-2 \ln(\bar{R}_1)}, \quad [0 \dots \infty] \quad (14.3)$$

Listing 14.5: Circular variance and standard deviation

```

1 FUNCTION circularVariance (R : float) : float;
2
3 BEGIN
4   Result := 1 - R;
5 END;
6
7 FUNCTION circularStandardDeviation (R : float) : float;
8
9 BEGIN
10  Result := sqrt(-2 * ln(R));
11 END;

```

The maximum likelihood estimator for the parameter  $\kappa$  of the VON MIESES distribution is given by the approximation

$$\hat{\kappa} = \begin{cases} 2\bar{R} + \bar{R}^3 + \frac{5\bar{R}^5}{6} & \forall \bar{R} < 0.53 \\ -0.4 + 1.39\bar{R} + \frac{0.43}{1-\bar{R}} & \forall \bar{R} \in [0.53 \dots 0.85[ \\ \frac{1}{\bar{R}^3 - 4\bar{R}^2 + 3\bar{R}} & \forall \bar{R} \geq 0.85 \end{cases} \quad (14.4)$$

which for  $n \leq 15$  needs a small sample correction

$$\hat{\kappa}_c = \begin{cases} \max((\hat{\kappa} - 2/(n\hat{\kappa})), 0) & \forall \hat{\kappa} < 2 \\ (n-1)^3 \frac{\hat{\kappa}}{n^3+n} & \forall \hat{\kappa} \geq 2 \end{cases} \quad (14.5)$$

Listing 14.6: Maximum likelihood estimator of  $\kappa$

```

1 FUNCTION Kappa (R : float; n : WORD) : float;
2
3 VAR k : float;
4
5 BEGIN

```

```

6   IF R < 0.53
7     THEN
8       k := 2*R + R*R*R + 5*pot(R, 5)/6
9     ELSE
10      IF R < 0.85
11        THEN k := -0.4 + 1.39*R + 0.43/(1-R)
12        ELSE k := 1 / (R*R*R - 4*R*R + 3*R);
13    IF n <= 15 // small sample correction
14    THEN
15      IF k < 2
16        THEN k := max((k - 2/(n*k)), 0)
17        ELSE k := pot(Pred(n), 3) * k / (n*n*n + n);
18    Result := k;
19 END;

```

The median direction of a circular sample is the diameter of the circle that divides the data into two groups of equal size. However, we can use the median function for linear data only if all data are on the same side of the 0-point.

Listing 14.7: Median of circular data

```

1 FUNCTION MedianDirection (TransformedData : VectorTyp) : float;
2
3 BEGIN
4   Result := Median(TransformedData);
5 END;

```

### 14.3.3. Higher moments

The  $p$ th trigonometric moment (centred, that is, relative to the mean direction) is calculated by

$$\mu_p = \frac{1}{n} \sum_{i=1}^n (\cos(p(\theta_i - \bar{\theta}))) + i \frac{1}{n} \sum_{i=1}^n (\sin(p(\theta_i - \bar{\theta}))) \quad (14.6)$$

Listing 14.8:  $p$ -th trigonometric moment

```

1 FUNCTION TrigonometricMoment (Data : VectorTyp; MeanAngle : float; p :
2   WORD) : ComplexTyp;
3
4 VAR n, i : WORD;
5   SumCos, SumSin, x : float;
6
7 BEGIN
8   n := VectorLength(Data);
9   SumCos := 0;
10  SumSin := 0;
11  FOR i := 1 TO n DO

```

## 14. Circular data

```

11   BEGIN
12     x := GetVectorElement(Data, i);
13     SumSin := SumSin + Sin(p * (x - MeanAngle));
14     SumCos := SumCos + Cos(p * (x - MeanAngle));
15   END;
16   Result := ComplexInit(SumCos/n, SumSin/n);
17 END;

```

From these moments, we can calculate the  $p$ -th centred mean vector just like we did for the mean vector above:

Listing 14.9: p-th centered mean

```

1 function CenteredMean(Moment : ComplexTyp) : ComplexTyp;
2
3 var S, C, R, theta : float;
4
5 BEGIN
6   C := Re(Moment);
7   S := Im(Moment);
8   R := sqrt(C*C + S*S);
9   CASE signum(C) OF
10     0 : theta := Const_pi / 2;      // arccos(C/R) = arccos(0)
11     -1 : theta := ArcTan(S/C) + Const_pi;
12     1 : CASE signum(S) OF
13       1 : theta := ArcTan(S/C);
14       -1 : theta := ArcTan(S/C) + Const_2pi;
15       0 : theta := 0;
16     END;
17   END;
18   Result := ComplexInit(R, theta); // convert to complex number
19 END;

```

From the first and second trigonometric moments  $m_1, m_2$  we calculate the circular dispersion as  $\delta = \frac{1-\mu_2}{2\mu_1^2}$

Listing 14.10: Circular dispersion

```

1 FUNCTION CircularDispersion (TransformedData : VectorTyp; Mean :
2   ComplexTyp) : float;
3
4 var n, i : word;
5   R, theta, m2 : float;
6
7 BEGIN
8   n := VectorLength(TransformedData);
9   theta := Im(Mean);
10  m1 := Re(TrigonometricMoment(TransformedData, theta, 1));
11  m2 := Re(TrigonometricMoment(TransformedData, theta, 2));

```

```

11  Result := (1 - m2) / (2 * m1 * m1);
12  END;

```

## Skew

Skew can be calculated either from the original ( $s$ ) or from the centred data ( $s_0$ )

$$s = \frac{1}{n} \sum_{i=1}^n \sin(2(\theta_i - \bar{\theta})) \quad (14.7)$$

$$s_0 = \frac{R_2 \sin(\theta_2 - 2\theta)}{1 - R} \quad (14.8)$$

A value close to 0 indicates a symmetrical distribution around the mean direction.

Listing 14.11: Circular skew

```

1  FUNCTION CircularSkew (TransformedData : VectorTyp; Mean : ComplexTyp) :
2      float;
3
4  VAR n, i : WORD;
5      theta, SumSin : float;
6
7  BEGIN
8      n := VectorLength(TransformedData);
9      theta := Im(Mean); // mean angle
10     SumSin := 0.0;
11     FOR i := 1 TO n DO
12         SumSin := SumSin + sin(2 * (GetVectorElement(TransformedData, i) -
13             theta));
14     Result := SumSin / n;
15 END;
16
17
18 FUNCTION CenteredCircularSkew (Mean1, Mean2 : ComplexTyp) : float;
19
20 VAR theta1, theta2, R1, R2 : float;
21
22 BEGIN
23     theta1 := Im(Mean1);
24     R1 := Re(Mean1);
25     theta2 := Im(Mean2);
26     R2 := Re(Mean2);
27     Result := pot(1-R1, 2/3);
28     IF Result = 0
29     THEN
30         BEGIN
31             CircleError := true;

```

## 14. Circular data

```

30      WriteLn('Error: Circular skew when resultant length is 1');
31      EXIT;
32  END;
33  Result := R2 * sin(theta2 - 2*theta1) / Result;
34 END;
```

### Kurtosis

The circular kurtosis we also can calculate centred or non-centred

$$k = \frac{1}{n} \sum_{i=1}^n \cos(2(\theta_i - \bar{\theta})) \quad (14.9)$$

$$k_0 = \frac{R_2 \cos(\theta_2 - 2\bar{\theta}) - R^4}{(1 - R)^2} \quad (14.10)$$

, a value close to 1 indicates a strongly peaked distribution. If the data come from a VON MIESES distribution, then  $k_0 = 0$ .

Listing 14.12: Circular kurtosis

```

1  FUNCTION CircularKurtosis (TransformedData : VectorTyp; Mean : ComplexTyp)
2    : float;
3
4  VAR n, i           : WORD;
5    theta, SumCos : float;
6
7  BEGIN
8    n := VectorLength(TransformedData);
9    theta := Im(Mean); // mean angle
10   SumCos := 0.0;
11   FOR i := 1 TO n DO
12     SumCos := SumCos + cos(2 * (GetVectorElement(TransformedData, i) -
13                               theta));
14   Result := SumCos / n;
15 END;
16
17
18 FUNCTION CenteredCircularKurtosis (Mean1, Mean2 : ComplexTyp) : float;
19
20 VAR theta1, theta2, R1, R2 : float;
21
22 BEGIN
23   theta1 := Im(Mean1);
24   R1 := Re(Mean1);
25   theta2 := Im(Mean2);
26   R2 := Re(Mean2);
27   Result := sqr(1-R1);
```

```

26 IF Result = 0
27 THEN
28 BEGIN
29   CircleError := true;
30   WriteLn('Error: Circular kurtosis when resultant length is 1');
31   EXIT;
32 END;
33 Result := (R2 * cos(theta2 - 2*theta1) - pot(R1, 4)) / Result;
34 END;

```

#### 14.3.4. Confidence interval for mean direction

If the allowable error is  $\delta$ , then the confidence interval  $d(1 - \delta)$  is computed by

$$d = \begin{cases} \arccos \left[ \frac{\sqrt{\frac{2n(2(\bar{R}n)^2 - n\chi_{\delta,1}^2)}{4n - \chi_{\delta,1}^2}}}{\bar{R}n} \right] & \forall \bar{R} \leq 0.9 \wedge \bar{R} > \sqrt{\frac{\chi_{\delta,1}^2}{2n}} \\ \arccos \left[ \frac{\sqrt{n - n^2 - (\bar{R}n)^2 \exp(\frac{\chi_{\delta,1}^2}{n})}}{\bar{R}*n} \right] & \forall \bar{R} > 0.9 \end{cases} \quad (14.11)$$

The confidence interval is then  $\bar{\theta} \pm d$ .

Listing 14.13: Confidence interval of circular mean

```

1 FUNCTION ConfidenceInterval (R, delta : float; n : WORD) : float;
2
3 VAR Rn, chi : float;
4   c          : CHAR;
5
6 BEGIN
7   Rn := R * n;
8   chi := SignificanceLimit_Chi2(delta, 1);
9   IF R > 0.9
10  THEN
11    Result := arccos(Sqrt(n - (n*n - Rn*Rn) * Exp(chi/n))/Rn)
12  ELSE IF R > Sqrt(chi/(2*n))
13  THEN
14    Result := arccos(Sqrt((2 * n * (2*Rn*Rn - n*chi)) / (4*n -
15      chi))/Rn)
16  ELSE
17    BEGIN
      c := WriteErrorMessage('Confidence interval of mean angle:
                                vector length too small');
    END;

```

## 14. Circular data

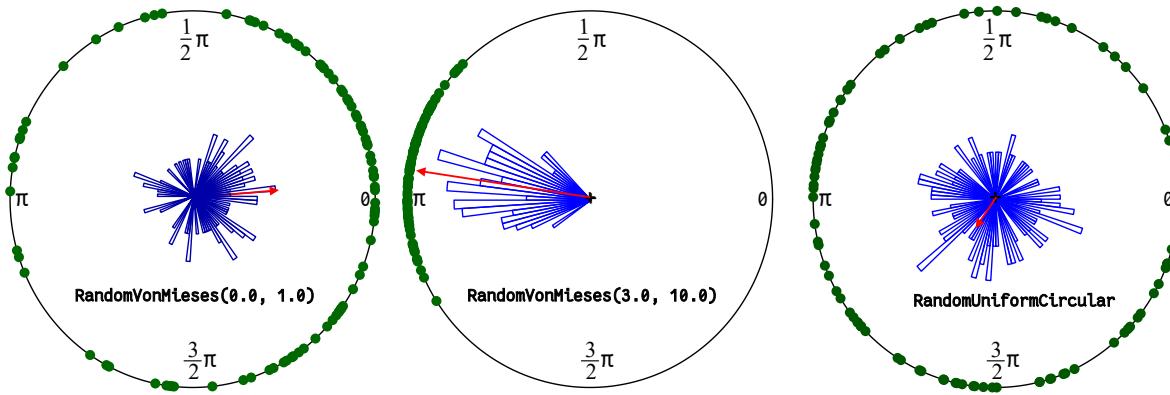


Figure 14.2.: Random variables of different distributions. *Right:* Uniformly distributed. *Left + middle:* VON MIESES distributed with different mean direction  $\bar{\theta}$  and spread  $\kappa$ .

```

18      CircleError := TRUE;
19      EXIT;
20  END;
21 END;

```

## 14.4. Pseudo-random variables

The following routine returns a VON MIESES-distributed pseudo-random number, the algorithm is described in [5] (see fig. 14.2).

Listing 14.14: von Mieses distributed random number

```

1 FUNCTION RandomVonMieses(mu, kappa : float) : float;
2
3 VAR s, u1, u2, theta : float;
4
5 BEGIN
6   IF kappa > 1.3
7     THEN s := 1/Sqrt(kappa)
8   ELSE s := Const_pi * Exp(-kappa);
9   REPEAT
10    u1 := Random;
11    u2 := Random;
12    theta := s*(2*u2 - 1) / u1; // generate prospective value
13    IF Abs(theta) < Const_pi // rejection pre-TEST
14      THEN
15        IF ((kappa*theta*theta) < (4 - 4*u1)) // acceptance pre-TEST
16        THEN
17          BEGIN
18            Result := ModuloTwoPi(theta + mu);

```

```

19      EXIT;
20  END
21 ELSE
22   IF ((kappa*Cos(theta)) > (2*Ln(u1) + kappa))
23 THEN
24 BEGIN
25   Result := ModuloTwoPi(theta + mu);
26   EXIT;
27 END
28 ELSE // IF < THEN TRY again
29 ELSE // IF > THEN TRY again
30 UNTIL FALSE;
31 END;

```

Random numbers uniformly distributed around the circle are a useful  $\theta$ -model:

Listing 14.15: Random data uniformly distributed over a circle

```

1 FUNCTION RandomUniformCircular : float;
2
3 BEGIN
4   Result := Random * Const_2pi;
5 END;

```

## 14.5. Statistical tests for a single circular data set

### 14.5.1. Do samples have a preferred direction

These procedures test  $H_0$ : The data are uniformly distributed against  $H_1$ : There is a preferred direction.

#### The RAYLEIGH-test

From the discussion above it is clear that the closer the data cluster toward a main direction, the larger the length of the mean vector becomes. Thus,  $H_0$  could also be phrased as  $\bar{R} = 0$  and  $H_1$  as  $\bar{R} > 0$ . For  $n > 10$  and VON MIESES-distributed data the probability for  $H_0$  becomes:

$$P_0 \approx \exp \left[ \sqrt{1 + 4n + 4(n^2 - R^2)} - (1 + 2n) \right] \quad (14.12)$$

This test is suitable only for unimodal data. Note that in the formula  $R$  is used, not  $\bar{R}$ . For consistency, the following function accepts  $\bar{R}$ , however.

Listing 14.16: Rayleigh-test

```

1 FUNCTION Rayleigh (MeanVectorLength : float; n : WORD) : float;
2

```

## 14. Circular data

```

3  VAR c : CHAR;
4      x : float;
5
6  BEGIN
7      IF (n < 10)
8          THEN
9              BEGIN
10                 c := WriteErrorMessage('Rayleigh test with less than 10 data
11                     points');
12                 CircleError := TRUE;
13                 EXIT;
14             END;
15             x := MeanVectorLength * n; // convert TO unscaled
16             Result := Exp(Sqrt(1 + 4*n + 4*(n*n - x*x)) - (1 + 2*n));
17         END;

```

### The *omnibus* test of HODGES-AJNE

If the data are plotted on a circle, and a diameter is drawn through that circle, some data will be on one, the others on the other side of that diameter. The diameter line is then rotated until the number of data points on the “wrong” side ( $m$ ) becomes minimal. Actually (at least in unimodal distributions), the diameter that produces minimal  $m$  must be the one at right angle to the mean vector.  $m$  will be the smaller the more directional the data are. This test is non-parametric and does not require the data to be von MIESES distributed (hence “*omnibus*” = Lat. “for all”, here: irrespective of a data model). The probability, to observe a  $m$  this small for randomly distributed data becomes

$$P_0 = \frac{1}{2^{n-1}} (n-2m) \binom{n}{m} \quad (14.13)$$

As a non-parametric test, this test is less powerful than the RAYLEIGH-test.

Listing 14.17: Omnibus-test

```

1  FUNCTION HedgesAjne (TransformedData : VectorTyp; MeanAngle : float) :
2      float;
3
4  VAR StartAngle, StopAngle, x : float;
5      i, m1, m2, n : WORD;
6
7  BEGIN
8      StartAngle := ModuloTwoPi(MeanAngle - Ninety);
9      StopAngle := ModuloTwoPi(MeanAngle + Ninety);
10     IF StartAngle > StopAngle
11         THEN
12             BEGIN // exchange start- and stop angle
13                 x := StartAngle;

```

```

13      StartAngle := StopAngle;
14      StopAngle := x;
15  END;
16 n := VectorLength(TransformedData);
17 m1 := 0;
18 m2 := 0;
19 FOR i := 1 TO n DO
20 BEGIN
21     x := GetVectorElement(TransformedData, i);
22     IF (x >= StartAngle) AND (x <= StopAngle)
23         THEN INC(m1) // count # OF data points on either side of the
24             diameter
25     ELSE INC(m2);
26 END;
27 IF (m1 > m2) THEN m1 := m2; // peak on the other side of diameter
28 Result := 1/pot(2, Pred(n)) * (n - 2*m1) * BinomialCoef(n, m1);
29 END;

```

### The $\chi^2$ -test

Input parameters are the data vector (transformed to  $[0\dots 2\pi]$ ), the direction of the mean vector, and the number of sectors. Output data are the  $\chi^2$  and the degrees of freedom.

Listing 14.18:  $\chi^2$ -test for homogeneous distribution

```

1 PROCEDURE ChiSqrTest (Data : VectorTyp; Direction : float; Sectors : WORD;
2                         VAR ChiSqr : float; VAR dgf : WORD);
3
4 VAR i, j, Counter, n, max : WORD;
5     Expected, Angle,
6     StartAngle,
7     Start, Finish, x      : float;
8     Dummy                 : CHAR;
9     ErrorStr, s          : STRING;
10
11 BEGIN
12     n := VectorLength(Data);
13     IF n < 8
14     THEN
15         BEGIN
16             CircleError := TRUE;
17             Dummy := WriteErrorMessage('Not enough data for ChiSqr-test (> 8
18             required)');
19             EXIT;
20         END;
21     max := n DIV 4;

```

## 14. Circular data

```

21  IF max > 250 THEN max := 250;
22  Expected := n / Sectors;
23  IF Expected < 4
24    THEN
25    BEGIN
26      CircleError := TRUE;
27      Str(n:4, ErrorStr);
28      Str(max:4, s);
29      ErrorStr := ErrorStr + 'Data points allow at most ' + s + '
30          sectors';
31      Dummy := WriteErrorMessage(ErrorStr);
32      EXIT;
33    END;
34  ChiSqr := 0.0;
35  Angle := Const_2pi / Sectors;
36  StartAngle := Direction - 0.5 * Angle;
37  dgf := Pred(Sectors);
38  FOR i := 1 TO Sectors DO
39    BEGIN
40      Start := StartAngle + Pred(i) * Angle;
41      Finish := Start + Angle;
42      Counter := 0;
43      FOR j := 1 TO n DO
44        BEGIN
45          x := GetVectorElement(Data, j);
46          IF (Start <= Const_2pi) AND (Finish <= Const_2pi)
47            THEN
48              IF (x > Start) AND (x < Finish)
49                THEN INC(Counter)
50                ELSE // outside SECTOR
51            ELSE
52              IF (Start < Const_2pi) AND (Finish > Const_2pi)
53                THEN
54                  IF ((x > Start) AND (x < Const_2pi)) OR (x <
55                      ModuloTwoPi(Finish))
56                    THEN INC(Counter)
57                    ELSE
58                  ELSE
59                    IF (x > ModuloTwoPi(Start)) AND (x < ModuloTwoPi(Finish))
60                      THEN INC(Counter);
61        END; // FOR j
62        ChiSqr := ChiSqr + Sqr(Counter - Expected) / Expected;
63    END; // FOR i
64 END;

```

### V-test for uniformity against a suspected direction

Sometimes one can specify an expected value for the mean vector direction *before* an experiment is undertaken. Then a test for randomness needs not exclude all other possible models, hence the “V-test” is more efficient [6]. Thus,  $H_0$ : **The data are homogeneously distributed** against  $H_1$ : **The data cluster around a hypothetical direction  $\theta_A$** . Note that the test applies only to randomness, it is not useful to decide whether the mean direction coincides with the expected direction. The test statistics is

$$V = \sqrt{\frac{2}{n}} R \cos(\bar{\theta} - \theta_A) \quad (14.14)$$

However, if the test fails to reach significance, it is unclear whether the data are uniformly distributed or the expected value was wrong.

Listing 14.19: V-test for random distribution against suspected direction

```

1 FUNCTION HomewardComponent (n : WORD; Mean : ComplexTyp; Expected : float)
2   : float;
3
4
5 VAR Length, Phi, v : float;
6
7 BEGIN
8   Polar(Mean, Length, Phi);
9   Phi := ModuloTwoPi(Phi);
10  v := Length * Cos(Phi - Expected);
11  Result := Sqrt(2*n) * v;
12 END;
```

### RAO's spacing test

If circular data are randomly distributed, then their distance from each other should be roughly  $\lambda = \frac{360}{n}$  [7]. If the data are clustered, some distances should be significantly larger than others. The test statistics becomes:

$$U = \frac{1}{2} \sum_{i=1}^{n-1} |(\theta_{i+1} - \theta_i) - \lambda| \quad (14.15)$$

RAO's test can be used for uni- and multimodal data, the critical values of **U** are tabulated in [8].

Listing 14.20: RAO's test

```

1 FUNCTION Rao (TransformedData : VectorTyp) : float;
2
3 VAR T, SumT, Expected : float;
4   i, n                 : WORD;
5
```

## 14. Circular data

```

6  BEGIN
7      n := VectorLength(TransformedData);
8      Expected := Const_2pi / n;
9      SumT := 0;
10     FOR i := 2 TO n DO
11         BEGIN
12             T := GetVectorElement(TransformedData, i) -
13                 GetVectorElement(TransformedData, Pred(i));
14             SumT := SumT + Abs(T - Expected);
15         END;
16         T := Const_2pi + GetVectorElement(TransformedData, 1) -
17             GetVectorElement(TransformedData, n);
18         SumT := SumT + Abs(T - Expected);
19         Rao := 0.5 * SumT;
20     END;

```

### KUIPERS's test against a suspected distribution

KUIPERS's test compares a given distribution of data with a theoretical model. The data matrix contains in the first column the angle, in the second the measured cumulative frequency and in the third the theoretical cumulative frequency (both in the range [0...1]). Returned are  $V_n$  and  $K$ .

Listing 14.21: KUIPERS's test

```

1  PROCEDURE Kuipers (Data : MatrixTyp; VAR K, U : float);
2
3  VAR DPlus, DMinus, c, vSqr, cvn, vSum,
4      Diff, Diff1, Diff2, Vn, vMean, x      : float;
5      n, i                               : WORD;
6      xVector, yVector,
7      bVector                           : VectorTyp;
8
9  BEGIN
10    n := MatrixRows(Data);
11    GetColumn(Data, 1, xVector);
12    GetColumn(Data, 2, yVector);
13    GetColumn(Data, 3, bVector);
14    vSum := NeumaierSum(bVector);
15    vMean := vSum / n;
16    DPlus := 0.0;
17    DMinus := 0.0;
18    vSqr := 0.0;
19    cvn := 0.0;
20    FOR i := 1 TO n DO
21        BEGIN
22            x := GetVectorElement(bVector, i);

```

```

23      Diff  := GetVectorElement(yVector, i) - x;
24      Diff1 := GetVectorElement(yVector, Pred(i)) - x;
25      Diff2 := GetVectorElement(yVector, Succ(i)) - x;
26      IF Diff > DPlus THEN DPlus := Diff;
27      IF Diff1 > DPlus THEN DPlus := Diff1;
28      IF Diff < DMinus THEN DMinus := Diff;
29      IF Diff1 < DMinus THEN DMinus := Diff1;
30      c := Pred(2 * i);
31      cvn := cvn + c * GetVectorElement(bVector, i) / n;
32      vSqr := vSqr + Sqr(GetVectorElement(bVector, i));
33  END; // FOR
34  Vn := (DPlus + Abs(DMinus)) * Sqrt(n);
35  K := Vn;
36  U := vSqr - cvn + n * (1/3 - Sqr(vMean - 0.5));
37  DestroyVector(xVector);
38  DestroyVector(yVector);
39  DestroyVector(bVector);
40 END;

```

The following routine creates the table with cumulative frequencies needed by KUIPERS's test.

Listing 14.22: Create frequency table

```

1  procedure CalculateCumulativeFrequencies (Data : VectorTyp; FullCircle :
   float;
                                              var Result : MatrixTyp);

2
3
4  var i, n : word;
   Wert : float;
   D    : VectorTyp;

5
6
7 begin
8   n := VectorLength(Data);
9   CopyVector(Data, D);
10  ShellSort(D);
11  for i := 1 to n do
12    begin
13      Wert := GetVectorElement(D, i);
14      SetMatrixElement(Result, i, 1, Wert);
15      SetMatrixElement(Result, i, 2, i/n);
16      SetMatrixElement(Result, i, 3, Wert/FullCircle);
17    end;
18  DestroyVector(D);
19 end;
20

```

## 14. Circular data

### One sample $t$ -test

This test is used to test  $H_0$ : **The population mean angle  $\bar{\theta}$  is equal to  $\theta_0$**  against  $H_1$ : **The population mean angle  $\bar{\theta}$  is different from  $\theta_0$** . This test is performed by checking if  $\theta_0 \in \bar{\theta} \pm d(1 - \delta)$ , that is,  $\theta_0$  is between the upper and lower confidence limits.

Listing 14.23: One-sample  $t$ -test

```
1 FUNCTION OneSample (MeanAngle, R, delta, TestAngle : float; n : WORD) :  
    BOOLEAN;  
2  
3 VAR x : float;  
4  
5 BEGIN  
6     x := ConfidenceInterval(R, delta, n);  
7     Result := (TestAngle > (MeanAngle - delta)) AND (TestAngle < (MeanAngle +  
        delta));  
8 END;
```

### Binomial test for median angle

This is a non-parametric test with a similar purpose as the one-sample test, however, it is based on the median rather than mean angle.  $H_0$ : **The population median angle  $\check{\theta}$  is equal to  $\theta_0$** . If  $H_0$  holds, then  $\theta_0$  should divide the data set into two nearly identical halves, the number of data on either side should fall under the binomial distribution  $k = B(n, 0.5)$ .

#### 14.5.2. Symmetry around $\check{\theta}$

If the data set is symmetrical around  $\check{\theta}$ , then the median circular distance of the data points from the median  $\check{\theta}$  should be zero. This can be tested by a WILCOXON signed rank test.

#### 14.5.3. Grouped data

The data matrix contains circular data in grouped form: first column the middle of the range of angles, the second column the counts. The angles must be distributed across the circle evenly.

Listing 14.24: Mean direction of grouped data

```
1 FUNCTION MeanVectorGrouped (Transformed : MatrixTyp; Difference : float) :  
    ComplexTyp;  
2  
3 VAR Total, n, i, j : WORD;  
4     x, y, z,  
5     Length, Angle : float;
```

```

6      Coord          : ComplexTyp;
7
8  BEGIN
9    n := MatrixRows(Transformed);
10   Total := 0;
11   x := 0.0;
12   y := 0.0;
13   FOR i := 1 TO n DO
14     BEGIN
15       j := Round(GetMatrixElement(Transformed, i, 2));
16       z := GetMatrixElement(Transformed, i, 1);
17       Total := Total + j;
18       x := x + j * Cos(z);
19       y := y + j * Sin(z);
20     END;
21   x := x / Total;
22   y := y / Total;
23   Coord := ComplexInit(x, y);      // Re, Im
24   Polar(Coord, Length, Angle);    // polar coordinates
25   z := Difference / 2;
26   z := z / Sin(z);
27   Length := Length * z;           // correct quantisation error
28   Result := Rect(Length, Angle);  // back TO cartesian coordinates }
29 END;

```

## 14.6. Multi-sample tests

### 14.6.1. WATSON-WILLIAMS-test: two vectors with circular data

This test is the circular equivalent of the two-sample  $t$ -test for linear data, it assesses the question whether  $s$  data sets come from distributions with the same mean angle [9, 10].

$$F = K \frac{(n-s)(\sum_{j=1}^s (R_j - R))}{(s-1)(n - \sum_{j=1}^s (R_j))}, \quad K = 1 + \frac{3}{8\kappa} \quad (14.16)$$

where  $R_j$  is the separate mean vector length for the  $j$ th group.  $\kappa$  is an estimator for the dispersion of the VON MIESES distribution. The resulting test statistics is compared with the critical value of  $F_{\delta,1,n-2}$ .

The test is relatively robust against violations of the VON MIESES distribution, as long as all groups have at least 5 members.

Listing 14.25: Watson-Williams test

```

1  PROCEDURE WatsonWilliams (TransformedData1, TransformedData2 : VectorTyp);
2
3  VAR Transformed3

```

## 14. Circular data

```

4      Mean1, Mean2, Mean3                      : ComplexTyp;
5      Length1, Length2, Length3, Phi1, Phi2, Phi3,
6      x, r, g, F, P0                          : float;
7      i, n1, n2, n3, nMean                   : WORD;
8      hst                                     : STRING;
9
10 BEGIN
11   n1 := VectorLength(TransformedData1);
12   n2 := VectorLength(TransformedData2);
13   n3 := n1 + n2;
14   CreateVector(Transformed3, n3, 0.0);
15   FOR i := 1 TO n1 DO
16     SetVectorElement(Transformed3, i, GetVectorElement(TransformedData1,
17                       i));
18   FOR i := 1 TO n2 DO
19     SetVectorElement(Transformed3, n1+i, GetVectorElement(TransformedData2,
20                       i));
21   Mean1 := MeanVector(TransformedData1);
22   Polar(Mean1, Length1, Phi1);
23   Phi1 := ModuloTwoPi(Phi1);
24   Mean2 := MeanVector(TransformedData2);
25   Polar(Mean2, Length2, Phi2);
26   Phi2 := ModuloTwoPi(Phi2);
27   Mean3 := MeanVector(Transformed3);
28   Polar(Mean3, Length3, Phi3);
29   Phi3 := ModuloTwoPi(Phi3);
30   r := (Length1 + Length2) / 2;
31   nMean := (n1 + n2) DIV 2;
32   g := 1 + 3 / (8 * Kappa(r, nMean));
33   F := g * (Length1 + Length2 - Length3) / (n3 - (Length1 + Length2));
34   P0 := Integral_F(F, 1, n3-2);
35   Writeln('F = ', FloatStr(F, ValidFigures), ', f1 = 1, f2 = ', n3-2:4,
36           ' => P0 = ' + FloatStr(P0, ValidFigures));
37   DestroyVector(Transformed3);
38 END;

```

### 14.6.2. Circular KRUSKAL-WALLIS-test for equal median

Non-parametric test for  $H_0: \theta_1 = \theta_2 = \dots = \theta_s$  against  $H_1: \text{At least one of the data set has a different median}$ . The test works by first calculating the overall median of the combined data set. Then for each group  $i$  we calculate  $m_i$ , the number of samples  $\theta_j^i$  ( $j$ th sample in the  $i$ th group) where the distance  $d(\theta_j^i, \bar{\theta})$  is negative. Then the test statistics

$$x = \frac{N^2}{M(N-M)} \sum_{i=1}^s \frac{m_i^2}{n_i} - \frac{NM}{N-M} \quad (14.17)$$

( $M, N$  total over all groups,  $m_i, n_i$  within the  $i$ th group) is compared with the upper  $1 - \delta$ th percentile of the  $\chi^2(\delta, s-1)$  distribution. The test is valid only if all  $n_i > 10$ .

Listing 14.26: Kruskal-Wallis test

```

1  FUNCTION KruskalWallis (TransformedData1, TransformedData2 : VectorTyp) :
2      float;
3
4  VAR Data3                      : VectorTyp;
5      Median3, x, xmin, xmax, P   : float;
6      i, n1, n2, n3, m1, m2, m : WORD;
7      c                         : CHAR;
8
9  BEGIN
10     n1 := VectorLength(TransformedData1);
11     n2 := VectorLength(TransformedData2);
12     n3 := n1 + n2;
13     IF (n1 <= 10) OR (n2 <= 10)
14     THEN
15         BEGIN
16             c := WriteErrorMessage('Circular Kruskal-Wallis test: n > 10
17                           required for all groups');
18             CircleError := TRUE;
19             EXIT;
20         END;
21     CreateVector(Data3, n3, 0.0);
22     FOR i := 1 TO n1 DO
23         SetVectorElement(Data3, i, GetVectorElement(TransformedData1, i));
24     FOR i := 1 TO n2 DO
25         SetVectorElement(Data3, n1+i, GetVectorElement(TransformedData2, i));
26     ShellSort(Data3);
27     Median3 := MedianDirection(Data3);
28     xmin := ModuloTwoPi(Median3 + Ninety);
29     xmax := ModuloTwoPi(Median3 - Ninety);
30     m1 := 0;
31     FOR i := 1 TO n1 DO
32         BEGIN
33             x := GetVectorElement(TransformedData1, i);
34             IF (x > xmin) AND (x < xmax) THEN INC(m1);
35         END;
36     m2 := 0;
37     FOR i := 1 TO n2 DO
38         BEGIN
39             x := GetVectorElement(TransformedData2, i);
40             IF (x > xmin) AND (x < xmax) THEN INC(m2);
41         END;
42     m := m1 + m2;

```

## 14. Circular data

```

41   P := n3*n3 / (m * (n3 - m)) * (m1*m1/n1 + m2*m2/n2) - n3 * m / (n3 - m);
42   Result := IntegralChi(P, 1);
43   Writeln('Chi^2 = ', FloatStr(P, ValidFigures), ' with 1 degrees of
44   freedom, P0 = ',
45   DestroyVector(Data3);
46 END;

```

### 14.6.3. Difference between two vectors

Calculates the cumulative frequencies of both data vectors and calculates KUIPER's V und WHATSON's U<sup>2</sup>. The data vectors must be sorted.

Listing 14.27: Difference between two vectors

```

1 PROCEDURE Difference (Data1, Data2 : VectorTyp; VAR KuipersV, WatsonsUSqr :
2   float);
3
4 TYPE ListenRecTyp = RECORD
5   Angle, k1, k2 : float;
6   END;
7
8 VAR n1, n2, Cumulative1, Cumulative2 : WORD;
9   Diff, Dif, DifSqr, DifPos, DifNeg, Angle1, Angle2 : float;
10  ListenRec : ListenRecTyp;
11  Liste : FiFo;
12
13 BEGIN
14   n1 := VectorLength(Data1);
15   n2 := VectorLength(Data2);
16   ShellSort(Data1);
17   ShellSort(Data2);
18   Cumulative1 := 0;
19   Cumulative2 := 0;
20   InitFiFo(Liste);
21   WHILE (Cumulative1 < n1) AND (Cumulative2 < n2) DO { Data
22     durchlaufen, }
23     BEGIN { cumulative
24       Frequenzen }
25       Angle1 := GetVectorElement(Data1, Succ(Cumulative1)); { erzeugen
26         und in FIFO }
27       Angle2 := GetVectorElement(Data2, Succ(Cumulative2)); { Liste
         speichern }
28       IF Angle1 < Angle2
29         THEN
30           BEGIN
31             INC(Cumulative1);

```

```

28      ListenRec.Angle := Angle1;
29  END
30 ELSE
31 BEGIN
32   INC(Cumulative2);
33   ListenRec.Angle := Angle2;
34 END;
35 ListenRec.k1 := 1.0 * Cumulative1 / n1;
36 ListenRec.k2 := 1.0 * Cumulative2 / n2;
37 Put(Liste, ListenRec, SizeOf(ListenRec));
38 END; { while }
39 WHILE (Cumulative1 < n1) DO          { eventuell
40   übrig gebliebene }
41 BEGIN                                     { Elemente aus
42   1. Datasatz }
43 INC(Cumulative1);                      { bearbeiten }
44 ListenRec.k1 := 1.0 * Cumulative1 / n1;
45 ListenRec.k2 := 1.0 * Cumulative2 / n2;
46 ListenRec.Angle := GetVectorElement(Data1, Cumulative1);
47 Put(Liste, ListenRec, SizeOf(ListenRec));
48 END;
49 WHILE (Cumulative2 < n2) DO          { dito üfr 2.
50   Datasatz }
51 BEGIN
52   INC(Cumulative2);
53   ListenRec.k1 := 1.0 * Cumulative1 / n1;
54   ListenRec.k2 := 1.0 * Cumulative2 / n2;
55   ListenRec.Angle := GetVectorElement(Data2, Cumulative2);
56   Put(Liste, ListenRec, SizeOf(ListenRec));
57 END;
58 DifPos := 0.0;
59 DifNeg := 0.0;
60 Dif    := 0.0;
61 DifSqr := 0.0;
62 WHILE NOT EmptyFiFo(Liste) DO          { Liste
63   durchlaufen und }
64 BEGIN                                     { öBgrte
65   positive und }
66 Get(Liste, ListenRec, SizeOf(ListenRec)); { negative
67   Difference suchen }
68 Diff := ListenRec.k1 - ListenRec.K2;
69 IF Diff > DifPos THEN DifPos := Diff;
70 IF Diff < DifNeg THEN DifNeg := Diff;
71 Dif := Dif + Diff;                      { aufsummieren
72   üfr Watson's Test }
73 DifSqr := DifSqr + Sqr(Diff);

```

## 14. Circular data

```

67      END;
68      KuipersV := n1 * n2 * (DifPos - DifNeg);
69      WatsonsUSqr := n1 * n2 / Sqr(n1+n2) * (DifSqr - Sqr(Dif) / (n1+n2));
70  END;

```

Under the condition that the data are VON-MISES-distributed, the following routine calculates the F-value for  $H_0: \kappa_1 = \kappa_2$  against  $H_1: \kappa_1 \neq \kappa_2$ . The probability of  $H_0$  is then obtained by integrating the F-distribution with  $n_1, n_2$  degrees of freedom. If  $F < 1$ , the integration has to be performed with  $1/F$  and exchanged degrees of freedom.

Listing 14.28: F-test

```

1  FUNCTION FTest (Data1, Data2 : VectorTyp) : float;
2
3  VAR Mean1, Mean2           : ComplexTyp;
4      Angle1, Angle2, Length1, Length2, rMean   : float;
5      n1, n2                   : WORD;
6
7  BEGIN
8      n1 := VectorLength(Data1);
9      Mean1 := MeanVector(Data1, 1);
10     Polar(Mean1, Length1, Angle1);
11     n2 := VectorLength(Data2);
12     Mean2 := MeanVector(Data2, 1);
13     Polar(Mean2, Length2, Angle2);
14     Length1 := Length1 * n1;
15     Length2 := Length2 * n2;
16     rMean := (Length1 + Length2) / (n1 + n2);
17     IF rMean < 0.7
18     THEN
19         BEGIN
20             CH := WriteErrorMessage('Mean vectors too short, F-Test not
21                           applicable');
22             CircleError := TRUE;
23             EXIT;
24         END;
25     Result := Pred(n2) * (n1 - Length1) / (Pred(n1) * (n2 - Length2));
26 END;

```

WILCOXON's non-parametric test for  $H_0$ : **The data sets have the same standard deviation** vs.  $H_1$ : **The data sets have different standard deviations**. Direction1 and -2 contain either the mean direction of the data vectors, or a common homeward component. In the latter case, the test is for equal/unequal home-finding ability. The function returns the WILCOXON-MANN-WHITNEY U-value. All data must be in  $[0..2\pi]$ .

Listing 14.29: Wilcoxon's non-parametric test for equal standard deviation

```

1  FUNCTION Wilcoxon (Data1, Data2 : VectorTyp; Direction1, Direction2 :
2      float) : float;

```

```

2
3 VAR U1, U2, X : float;
4   n1, n2, Rank, i, j, Sum1, Sum2 : WORD;
5   Diff1, Diff2 : VectorTyp;
6
7 BEGIN
8   n1 := VectorLength(Data1);
9   CreateVector(Diff1, n1, 0.0);
10  FOR i := 1 TO n1 DO
11    BEGIN
12      x := Abs(GetVectorElement(Data1, i) - Direction1);
13      IF x > Pi
14        THEN SetVectorElement(Diff1, i, Const_2pi-x)
15        ELSE SetVectorElement(Diff1, i, x);
16    END;
17  ShellSort(Diff1);
18  n2 := VectorLength(Data2);
19  CreateVector(Diff2, n2, 0.0);
20  FOR i := 1 TO n2 DO
21    BEGIN
22      x := Abs(GetVectorElement(Data2, i) - Direction2);
23      IF x > Pi
24        THEN SetVectorElement(Diff2, i, Const_2pi-x)
25        ELSE SetVectorElement(Diff2, i, x);
26    END;
27  ShellSort(Diff2);
28  Rank := 1;
29  j := 1;
30  Sum1 := 0;
31  Sum2 := 0;
32  IF n1 < n2
33    THEN
34      BEGIN
35        FOR i := 1 TO n1 DO
36          BEGIN
37            WHILE (GetVectorElement(Diff2, j) < GetVectorElement(Diff1, i))
38              DO
39                BEGIN
40                  INC(Sum2, Rank);
41                  INC(Rank);
42                  INC(j);
43                END;
44                INC(Sum1, Rank);
45                INC(Rank);
46            END;
47        FOR i := j TO n2 DO

```

## 14. Circular data

```

47      BEGIN
48          INC(Sum2, Rank);
49          INC(Rank);
50      END;
51  END
52 ELSE
53 BEGIN
54     FOR i := 1 TO n2 DO
55         BEGIN
56             WHILE (GetVectorElement(Diff1, j) < GetVectorElement(Diff2, i))
57                 DO
58                     BEGIN
59                         INC(Sum1, Rank);
60                         INC(Rank);
61                         INC(j);
62                     END;
63                     INC(Sum2, Rank);
64                     INC(Rank);
65                 END;
66             FOR i := j TO n1 DO
67                 BEGIN
68                     INC(Sum1, Rank);
69                     INC(Rank);
70                 END;
71             DestroyVector(Diff1);
72             DestroyVector(Diff2);
73             U1 := Sum1 - n1 * Succ(n1) / 2;
74             U2 := Sum2 - n2 * Succ(n2) / 2;
75             IF U1 < U2
76                 THEN Result := U1
77             ELSE Result := U2;
78         END;

```

```

1 PROCEDURE DescribeBivariat (Data : MatrixTyp);
2
3 VAR i, n1                               : WORD;
4     SumX, SumY, a, b, c, d, r, SumXSqr, SumYSqr, SumXDevYDev,
5     xSta, ySta, xMean, yMean, CoVariance, Correlation, Phi,
6     Major, Minor, xMin, xMax, yMin, yMax       : float;
7
8 BEGIN
9     SumX := 0.0;
10    SumY := 0.0;
11    SumXSqr := 0.0;
12    SumYSqr := 0.0;
13    SumXDevYDev := 0.0;

```

```

14  n1 := MatrixRows(Data);
15  xMin := MaxRealNumber;
16  xMax := MinRealNumber;
17  yMin := MaxRealNumber;
18  yMax := MinRealNumber;
19  FOR i := 1 TO n1 DO
20    BEGIN
21      a := GetMatrixElement(Data, i, 1);
22      IF a > xMax THEN xMax := a;
23      IF a < xMin THEN xMin := a;
24      SumX := SumX + a;
25      SumXSqr := SumXSqr + Sqr(a);
26      a := GetMatrixElement(Data, i, 2);
27      IF a > yMax THEN yMax := a;
28      IF a < yMin THEN yMin := a;
29      SumY := SumY + a;
30      SumYSqr := SumYSqr + Sqr(a);
31    END;
32  xMean := SumX / n1;
33  yMean := SumY / n1;
34  xSta := Sqrt((SumXSqr - Sqr(SumX) / n1) / Pred(n1));
35  ySta := Sqrt((SumYSqr - Sqr(SumY) / n1) / Pred(n1));
36  Write('Mean x = ', FloatStr(xMean, ValidFigures), ' ± ', FloatStr(xSta,
37    ValidFigures));
37  Writeln('Mean y = ', FloatStr(yMean, ValidFigures), ' ± ', FloatStr(ySta,
38    ValidFigures));
39  FOR i := 1 TO n1 DO
40    BEGIN
41      a := GetMatrixElement(Data, i, 1) - xMean;
42      b := GetMatrixElement(Data, i, 2) - yMean;
43      SumXDevYDev := SumXDevYDev + a * b;
44    END;
45  CoVariance := SumXDevYDev / Pred(n1);
46  Correlation := CoVariance / (xSta * ySta);
47  A := Sqr(ySta);
48  B := - CoVariance;
49  C := Sqr(xSta);
50  D := (1 - Sqr(Correlation)) * Sqr(xSta) * Sqr(ySta);
51  R := Sqrt(Sqr(A - C) + 4 * Sqr(B));
52  Phi := ArcTan(2 * B / (A - C - R));
53  Major := Sqrt(2 * D / (A + C - R));
54  Minor := Sqrt(2 * D / (A + C + R));
55  Writeln('r = ', FloatStr(Correlation, ValidFigures), ' φ = ',
    FloatStr(Phi, ValidFigures));
  Writeln('Major = ', FloatStr(Major, ValidFigures), ' Minor = ',
    FloatStr(Minor, ValidFigures));

```

```
56  Write('Press any key: ');
57  END;
```

## 14.7. Regression and correlation

### 14.7.1. Linear dependent, circular independent variable

In this case, the data are fitted to

$$\hat{y} = a + b \cos(\theta - \theta_0) = a + b \cos(\theta) + c \sin(\theta) \quad (14.18)$$

$\theta_0$  is the peak phase (acrophase) of the data. In effect, the dependent variable is plotted onto a cylinder, rather than on a sheet of paper. This is called an **C-linear association** and is a special case of the **C-association**, which means any function that

- has exactly one maximum and one minimum over the range of  $[0 \dots 2\pi]$
- has matching  $\hat{y}$  at 0 and  $2\pi$  (*i.e.*, is periodic)

#### Degree of C-association

The U-test gives the probability of **H<sub>0</sub>: The variables  $\theta$  and  $x$  are independent** ( $D_n = 0$ ) against **H<sub>1</sub>: The variables  $\theta$  and  $x$  are C-associated** ( $D_n > 0$ ):

$$D_n = a_n(T_c^2 + T_s^2) \quad (14.19)$$

$$T_c = \sum_{i=1}^n x_i \cos(\theta_i), \quad T_s = \sum_{i=1}^n x_i \sin(\theta_i) \quad (14.20)$$

$$a_n = \begin{cases} [1 + 5 \cot^2(\pi/n) + 4 \cot^4(\pi/n)]^{-1} & n \text{ even} \\ 2 \sin^4(\pi/n)/[1 + \cos(\pi/n)]^3 & n \text{ odd} \end{cases} \quad (14.21)$$

$$U_n = 24 \frac{T_c^2 + T_s^2}{n^3 + n^2} \approx \chi^2_{2,\alpha} \quad (14.22)$$

$$P(H_0) = \exp\left(-\frac{U_n^2}{2}\right) \quad \forall n > 100 \quad (14.23)$$

$D_n$  is a quantity in  $[0 \dots 1]$ , with values near 0 suggesting that there is no C-association. It can be considered a correlation coefficient, for which  $U_n$  is the test statistics. For  $n \leq 100$ , critical values are given in [11]. After correction of two obvious typos these follow an exponential equation

$$y = a * (1 - b^n) \quad (14.24)$$

$\alpha$	$a$	$\pm$	$b$	$\pm$	$r^2$
with the parameters	0.10	4.608	0.008	0.668	0.004
	0.05	5.909	0.026	0.773	0.004
	0.01	8.912	0.064	0.865	0.003

Listing 14.30: C-association between linear and circular data

```

1  FUNCTION CAssociation (CONST theta, Y : VectorTyp; VAR Sig :
2    SignificanceType) : float;
3
4  VAR Tc, Ts, an, Un, P, xi, ti : float;
5    i, n : WORD;
6    c : char;
7
8  FUNCTION InterpolateCritical (Un : float; n : WORD) : float;
9    // interpolates critical values in table A10 from Fisher (1993)
10
11  const a10 = 4.608; a05 = 5.909; a01 = 8.912;
12    b10 = 0.668; b05 = 0.773; b01 = 0.865;
13
14  BEGIN
15    if Un < (a10 * (1 - pot(b10, n)))           // critical value for
16      10%
17    THEN Result := 1.0                         // to give it some
18      value
19    ELSE if Un < (a05 * (1 - pot(b05, n)))       // critical value for
20      5%
21    THEN Result := 0.1
22    ELSE if Un < (a01 * (1 - pot(b01, n))) // critical value for
23      1%
24    THEN Result := 0.05
25    ELSE Result := 0.01;
26  END;
27
28  BEGIN
29    n := VectorLength(theta);
30    IF VectorLength(Y) <> n
31    THEN
32      BEGIN
33        c := WriteErrorMessage('Linear-circular association: length of theta
34          and x not equal');
35        CircleError := true;
36        EXIT;
37      END;
38    Tc := 0;
39    Ts := 0;
40    FOR i := 1 TO n DO
41      BEGIN
42        xi := GetVectorElement(Y, i);
43        ti := GetVectorElement(theta, i);
44        IF IsNaN(xi) OR IsNaN(ti)
45        THEN // ignore data pair if either is NaN

```

## 14. Circular data

```

40      ELSE
41      BEGIN
42          Tc := Tc + xi*cos(ti);
43          Ts := Ts + xi*sin(ti);
44      END;
45  END;
46  IF ODD(n)
47  THEN an := 2*pot(sin(Const_pi/n), 4) / pot(1+cos(Const_pi/n), 3)
48  ELSE an := 1 / (1 + 5*pot(cot(Const_pi/n), 2) + 4*pot(cot(Const_pi/n),
49  4));
50  Result := an * (Tc*Tc * Ts*Ts); // correlation coefficient
51  WITH Sig DO
52  BEGIN
53      TestValue := 24 * (Tc*Tc * Ts*Ts) / (pot(n, 3) + pot(n, 2)); // test
54      statistics Un
55      Freedom := n;
56      IF n > 100
57      THEN P0 := exp(-pot(Un, 2)/2)
58      ELSE P0 := InterpolateCritical(Un, n);
59  END;
60 END;

```

### Fitting a trigonometric polynomial

The following procedure fits a dependent linear (`yData`) to an independent circular variable (`TransformedData`) using

$$\hat{y} = m + a \cos(\theta - \theta_0) \quad (14.25)$$

where  $m$  is the mean,  $a$  the amplitude and  $\theta_0$  the acrophase angle.

Listing 14.31: Fit a trigonometric polynomial

```

1 PROCEDURE TrigonometricPolynomial (const TransformedData, yData: VectorTyp;
2                                     Periode : float);
3
4 VAR i, n : WORD;
5     Omega, t, s, c, y, tMax, tMin, yMax, yMin,
6     SumY, SumC, SumS, SumCSqr, SumSSqr, SumYC,
7     SumYS, SumCS, M, X, Z, Hilfs1, Hilfs2, Hilfs3, Hilfs4,
8     Correlation, U, A, Phi, r, TEST, P0 : float;
9     Sig : SignificanceType;
10    hstr : STRING;
11
12 BEGIN
13     n := VectorLength(TransformedData);
14     IF n <> VectorLength(yData)
15     THEN

```

```

16   BEGIN
17     ch := WriteErrorMessage('Linear-periodic correlation: Vectors of
18       different lengths');
19     CircleError := TRUE;
20     EXIT;
21   END;
22   SumY := 0.0;
23   SumC := 0.0;
24   SumS := 0.0;
25   SumCSqr := 0.0;
26   SumSSqr := 0.0;
27   SumYC := 0.0;
28   SumYS := 0.0;
29   SumCS := 0.0;
30   tMax := MinRealNumber;
31   tMin := MaxRealNumber;
32   yMax := MinRealNumber;
33   yMin := MaxRealNumber;
34   FOR i := 1 TO n DO
35     BEGIN
36       t := GetVectorElement(TransformedData, i); // circular datum
37       IF t > tMax THEN tMax := t;
38       IF t < tMin THEN tMin := t;
39       c := Cos(Periode*t);
40       s := Sin(Periode*t);
41       SumC := SumC + c;
42       SumS := SumS + s;
43       y := GetVectorElement(yData, i); // linear datum
44       IF y > yMax THEN yMax := y;
45       IF y < yMin THEN yMin := y;
46       SumY := SumY + y;
47       SumCSqr := SumCSqr + Sqr(c);
48       SumSSqr := SumSSqr + Sqr(s);
49       SumYC := SumYC + y * c;
50       SumYS := SumYS + y * s;
51       SumCS := SumCS + c * s;
52     END;
53   Hilfs1 := Sqr(SumC) + n * SumSSqr;
54   Hilfs2 := (SumS * SumC * SumCS - Sqr(SumS) - n * Sqr(SumCS))
55     / Hilfs1 - Sqr(SumS) / n + SumSSqr;
56   Hilfs3 := (Sqr(SumS) * SumYC - n * SumYC * SumCS +
57     SumCS * SumC * SumY - SumCS * SumC * SumS) / Hilfs1;
58   Hilfs4 := (Sqr(SumS) * Sqr(SumC) - SumS * Sqr(SumC) * SumY)
59     / (n * Hilfs1);
60   Z := (SumYS + Hilfs3 + Hilfs4 - SumS * SumY / n) / Hilfs2;
61   X := (n * SumYC - SumC * SumY + SumC * SumS - n * SumCS * Z)

```

## 14. Circular data

```

61      / Hilfs1;
62      M := (SumY - SumC * X - SumS * Z) / n;
63      A := Sqrt(Sqr(X) + Sqr(Z));
64      CASE signum(X) OF
65          1 : Phi := ArcTan(Z/X);
66          -1 : IF x < 0 THEN Phi := Const_pi + ArcTan(Z/X);
67          0 : CASE signum(y) OF
68              1 : Phi := Ninety;                                { ° 90 }
69              -1 : IF Y < 0 THEN Phi := Const_pi*3/2;     { °270 }
70              0 : BEGIN
71                  CH := WriteErrorMessage('unknown akrophase angle ');
72                  CircleError := TRUE;
73                  EXIT;
74              END;
75          END;
76      END;
77      IF tMin > 0 THEN tmin := 0;
78      IF yMin > 0 THEN yMin := 0;
79      IF A < 0
80          THEN hstr := '_'
81          ELSE hstr := '+';
82      Writeln('y = ', FloatStr(M, ValidFigures), ' ', HStr, ' ', FloatStr(A,
83          ValidFigures),
84          ' * cos(', FloatStr(Periode, ValidFigures), ' * t - ',
85          FloatStr(Phi, ValidFigures), ')');
86      CircularLinearCorrelation(TransformedData, yData, r, sig);
87      WITH Sig DO
88          Writeln('Parametric: r = ', FloatStr(r, ValidFigures), ' chi2 = ',
89          FloatStr(TestValue, ValidFigures), ' with 2 dgf, P(H0: r = 0) = ',
90          FloatStr(P0, ValidFigures));
91      LinearPeriodicRankCorrelation(TransformedData, yData, Correlation, u);
92      Writeln('Non-Parametric r = ', FloatStr(Correlation, ValidFigures), ' U =
93          ', FloatStr(U, ValidFigures));
94  END;

```

### Correlation $r_{cl}$ between linear and circular data

We calculate the correlation between a directional random variable  $\Theta$  and a linear random variable  $\mathbf{x}$  by first calculating  $r_{sx} = r_p(\sin(\theta), \mathbf{x})$ ,  $r_{cx} = r_p(\cos(\theta), \mathbf{x})$   $r_{sc} = r_p(\sin(\theta), \cos(\theta))$ , where  $r_p$  is PEARSON's product-moment correlation (see section 11.2.1). Then

$$r_{cl} = \sqrt{\frac{r_{cx}^2 + r_{sx}^2 - 2r_{cx}^2 r_{sx}^2 r_{cs}^2}{1 - r_{cs}^2}} \quad (14.26)$$

and the test statistics  $\chi^2(n r_{cl}, 2)$ .

Listing 14.32: Correlation between linear and circular data

```

1 PROCEDURE CircularLinearCorrelation (const TransformedData, yData:
2   VectorTyp;
3
4   VAR Correlation : float; VAR Sig :
5     SignificanceType);
6
7
8
9 BEGIN
10  n := VectorLength(TransformedData);
11  IF n <> VectorLength(yData)
12    THEN
13      BEGIN
14        Hilfs := WriteErrorMessage('Linear-periodic correlation: Vectors of
15          different lengths');
16        CircleError := TRUE;
17        EXIT;
18      END;
19  CreateVector(SinPhi, n, 0.0);
20  CreateVector(CosPhi, n, 0.0);
21  CreateVector(y, n, 0.0);
22  FOR i := 1 TO n DO
23    BEGIN
24      x := GetVectorElement(TransformedData, i);
25      SetVectorElement(SinPhi, i, Sin(x));
26      SetVectorElement(CosPhi, i, Cos(x));
27    END;
28  ryc := PearsonProductMomentCorrelation(yData, CosPhi, sig);
29  rys := PearsonProductMomentCorrelation(yData, SinPhi, sig);
30  rcs := PearsonProductMomentCorrelation(CosPhi, SinPhi, sig);
31  DestroyVector(SinPhi);
32  DestroyVector(CosPhi);
33  Correlation := (Sqr(ryc) + Sqr(rys) - 2 * ryc * rys * rcs) / (1 -
34    Sqr(rcs)); // r^2
35  WITH Sig DO
36    BEGIN
37      TestValue := n * Correlation;
38      Freedom := 2;
39      P0 := IntegralChi(TestValue, 2);
40    END;
41  Correlation := Sqrt(Correlation);
42 END;

```

## Rank correlation

Rank correlation between a linear and a periodic variable. For SPEARMAN's rank correlation  $r_s$  between linear variables see chapter 11. Returns both  $r$  and the test statistics U. The test works even if the conditions for a parametric test are not met.

Listing 14.33: Rank correlation between linear and circular data

```

1 PROCEDURE RankXY (const TransformedData, yData : VectorTyp; var Ranks :
2   VectorTyp);
3
4 VAR n, i, Rank, MaxPos : WORD;
5   Kopie           : MatrixTyp;
6   Maximum, x      : float;
7
8 BEGIN
9   n := VectorLength(TransformedData);
10  CreateMatrix(Kopie, n, 2, 0.0);
11  SetColumn(Kopie, TransformedData, 1);
12  SetColumn(Kopie, yData, 2);
13  Rank := n;                      { Feld nach aufsteigenden x sortieren }
14 REPEAT
15   Maximum := MinRealNumber;
16   FOR i := 1 TO Rank DO
17     BEGIN
18       x := GetMatrixElement(Kopie, i, 1);
19       IF x > Maximum
20         THEN
21           BEGIN
22             Maximum := x;
23             MaxPos := i;
24           END;
25       x := GetMatrixElement(Kopie, MaxPos, 2);
26       SetMatrixElement(Kopie, MaxPos, 1, GetMatrixElement(Kopie, Rank, 1));
27       SetMatrixElement(Kopie, MaxPos, 2, GetMatrixElement(Kopie, Rank, 2));
28       SetMatrixElement(Kopie, Rank, 1, Rank);
29       SetMatrixElement(Kopie, Rank, 2, x);
30     DEC(Rank);
31   UNTIL (Rank = 0);
32   Rank := n;                      { jetzt die ÄRnge der y-Elemente
33   bestimmen }
34 REPEAT
35   Maximum := MinRealNumber;
36   FOR i := 1 TO Rank DO
37     BEGIN
38       x := GetMatrixElement(Kopie, i, 2);
39       IF x > Maximum

```

```

39      THEN
40      BEGIN
41          Maximum := x;
42          MaxPos := i;
43      END;
44  END;
45  x := GetMatrixElement(Kopie, MaxPos, 1);
46  SetMatrixElement(Kopie, MaxPos, 1, GetMatrixElement(Kopie, Rank, 1));
47  SetMatrixElement(Kopie, MaxPos, 2, GetMatrixElement(Kopie, Rank, 2));
48  SetMatrixElement(Kopie, Rank, 2, Maximum);
49  SetMatrixElement(Kopie, Rank, 1, x);
50  SetVectorElement(Ranks, Round(x), Rank);
51  DEC(Rank);
52 UNTIL (Rank = 0);
53 DestroyMatrix(Kopie);
54 END;
55
56
57 PROCEDURE LinearPeriodicRankCorrelation (const TransformedData, yData:
58     VectorTyp;
59
60     VAR Correlation, U : float);
61
62 VAR Ranks           : VectorTyp;
63     n, i             : WORD;
64     epsilon, x, y, c, s : float;
65     hilfs            : char;
66
67 BEGIN
68     n := VectorLength(TransformedData);
69     IF n <> VectorLength(yData)
70     THEN
71         BEGIN
72             Hilfs := WriteErrorMessage('Linear-periodic correlation: Vectors of
73                 different lengths');
74             CircleError := TRUE;
75             EXIT;
76         END;
77     epsilon := Const_2pi / n;
78     CreateVector(Ranks, n, 0.0);
79     RankXY(TransformedData, yData, Ranks);
80     c := 0.0;
81     s := 0.0;
82     FOR i := 1 TO n DO
83         BEGIN
84             x := GetVectorElement(Ranks, i);
85             y := epsilon * i;

```

## 14. Circular data

```

83      c := c + x * Cos(y);
84      s := s + x * Sin(y);
85  END;
86  IF Odd(n)
87    THEN x := 2 * pot(Sin(Pi/n),4) / pot(1 + Cos(Pi/n), 3)
88    ELSE x := 1 / (1 + 5 * pot(cot(Pi/n),2) + 4 * pot(cot(Pi/n),4));
89  y := 24 / (x * Sqr(n) * (n+1));
90  Correlation := x * (Sqr(c) + Sqr(s));
91  U := y * Correlation;
92  DestroyVector(Ranks);
93 END;

```

### 14.7.2. Circular dependent, linear independent variable

The circular variable  $\theta$  may depend on either a single explanatory variable  $x$ , or on a vector of  $k$  explanatory variables  $\mathbf{x} = x_1 \dots x_k$ . Either the mean  $\mu_i$  or the dispersion  $\kappa_i$  or both can depend on  $x_i$ . As dispersion is not a well-defined concept for general circular variables, this discussion is usually limited to circular variables that are drawn from a VON MIESES-distribution.

### 14.7.3. Circular dependent, circular independent variable

#### Circular-circular correlation $r_{cc}$

Assume two circular data sets  $\alpha, \beta$  with mean direction  $\bar{\alpha}, \bar{\beta}$ , respectively. Then

$$r_{cc} = \frac{\sum_{i=1}^n (\sin(\alpha_i - \bar{\alpha}) \sin(\beta_i - \bar{\beta}))}{\sqrt{\sum_{i=1}^n \sin^2(\alpha_i - \bar{\alpha}) \sin^2(\beta_i - \bar{\beta})}} \quad (14.27)$$

and we can test  $H_0 : r_{cc} = 0$  against  $H_1 : r_{cc} \neq 0$  with the test statistics

$$f = N \frac{\sum_{i=1}^n \sin^2(\alpha_i - \bar{\alpha}) \sum_{i=1}^n \sin^2(\beta_i - \bar{\beta})}{\sum_{i=1}^n \sin^2(\alpha_i - \bar{\alpha}) (\sin^2(\beta_i - \bar{\beta}))}, \quad t = \sqrt{f} r_{cc} \quad (14.28)$$

Listing 14.34: Correlation between linear and circular data

```

1 PROCEDURE CircularCircularCorrelation (Alpha, Beta : VectorTyp;
2   VAR Correlation : float; var Sig : SignificanceType);
3
4 VAR c : CHAR;
5   SumSinSin, SumSinSinSqr, SumSinA, SumSinB,
6   MeanA, MeanB, f, x, y : float;
7   n, na, i : WORD;
8
9 BEGIN

```

```

10  n := VectorLength(alpha);
11  IF (n <> VectorLength(beta))
12    THEN
13      BEGIN
14          c := WriteErrorMessage('Circular-circular correlation: Vectors of
15              unequal length');
16          CircleError := TRUE;
17          EXIT;
18      END;
19  MeanA := MedianDirection(Alpha);
20  MeanB := MedianDirection(Beta);
21  SumSinSin := 0;
22  SumSinSinSqr := 0;
23  SumSinA := 0;
24  SumSinB := 0;
25  na := 0;
26  FOR i := 1 TO n DO
27    BEGIN
28        x := GetVectorElement(Alpha, i);
29        y := GetVectorElement(Beta, i);
30        IF IsNaN(x) OR IsNaN(y)
31            THEN // Do nothing, allows data containing NaNs
32        ELSE
33            BEGIN
34                INC(na); // actual n, excluding NaNs
35                x := Sin(x - MeanA);
36                y := Sin(y - MeanB);
37                SumSinSin := SumSinSin + x*y;
38                SumSinSinSqr := SumSinSinSqr + x*x * y*y;
39                SumSinA := SumSinA + x*x;
40                SumSinB := SumSinB + y*y;
41            END;
42    Correlation := SumSinSin / Sqrt(SumSinSinSqr);
43    f := na * SumSinA * SumSinB / SumSinSinSqr;
44    WITH Sig DO
45      BEGIN
46          TestValue := Sqrt(f) * Correlation;
47          P0 := IntegralGauss(TestValue);
48      END;
49 END;

```

### T-monotone association

Assume, again, a data set of two circular variables  $\alpha, \beta$ . If you take samples of three pairs and plot  $\alpha_1, \alpha_2, \alpha_3$  on a circle, they may go clockwise or anti-clockwise. Do the

## 14. Circular data

same with  $\beta_1, \beta_2, \beta_3$ , and there are three possible outcomes:

**+1** both triples rotate in the same direction (concordant)

**-1** both triples rotate in different directions (discordant)

**0** there are ties either in the  $\alpha$ - and/or  $\beta$ -triple

Computationally, this is done by calculating

$$\delta = \operatorname{sgn}(\alpha_1 - \alpha_2) \operatorname{sgn}(\alpha_2 - \alpha_3) \operatorname{sgn}(\alpha_3 - \alpha_1) \times \operatorname{sgn}(\beta_1 - \beta_2) \operatorname{sgn}(\beta_2 - \beta_3) \operatorname{sgn}(\beta_3 - \beta_1) \quad (14.29)$$

Repeat this for all possible  $\binom{n}{3}$  possible combinations, and calculate

$$\hat{\Delta}_n = \frac{\sum_{1 \leq i < j < k \leq n} \delta_{i,j,k}}{\binom{n}{3} - N_0} \quad (14.30)$$

where  $N_0$  is the number of ties. If  $n$  is too large to calculate this for all possible triples, take a random sample and replace  $\binom{n}{3}$  in the denominator by the sample size  $m$ . Then we can calculate the probability of **H<sub>0</sub>:  $\alpha$  and  $\beta$  are not T-monotone** ( $\Delta = 0$ ) against **H<sub>1</sub>:  $\alpha$  and  $\beta$  are T-monotone** ( $\Delta = \pm 1$ ).

There is a computationally simpler method, the circular rank correlation  $\hat{\Pi}_n$ . If  $\gamma_i, \epsilon_i$  are the uniform scores ( $\gamma_i = \frac{2\pi r_i}{n}$ ,  $r_i$  is the  $i$ th order statistics) of  $\alpha_i, \beta_i$ , then

$$\Pi_n = \frac{4}{n^2} \sum_{1 \leq i < j \leq n} [\sin(\gamma_i - \gamma_j) \sin(\epsilon_i - \epsilon_j)] = \frac{4}{n^2} (AB - CD) \quad (14.31)$$

$$A = \sum_{i=1}^n [\cos(\gamma_i) \cos(\epsilon_i)] \quad (14.32)$$

$$B = \sum_{i=1}^n [\sin(\gamma_i) \sin(\epsilon_i)] \quad (14.33)$$

$$C = \sum_{i=1}^n [\cos(\gamma_i) \sin(\epsilon_i)] \quad (14.34)$$

$$D = \sum_{i=1}^n [\sin(\gamma_i) \cos(\epsilon_i)] \quad (14.35)$$

Obviously, it will depend on the  $\theta$ -direction. The critical values for  $\hat{\Pi}_n$  are in [2, table A13].

### T-linear association

In T-linear association, the relationship between  $\alpha$  and  $\beta$  is

$$\beta = (\pm \alpha + \alpha_0) \bmod 2\pi \quad (14.36)$$

Note that there is no scaling parameter as in  $\beta = \pm a(\alpha + \alpha_0)$ , as this would be difficult to estimate and to interpret. The correlation is

$$\rho_T = \frac{\sum_{1 \leq i < j \leq n} [\sin(\alpha_i - \alpha_j) \sin(\beta_i - \beta_j)]}{[\sum_{1 \leq i < j \leq n} \sin(\alpha_i - \alpha_j) \sum_{1 \leq i < j \leq n} \sin(\beta_i - \beta_j)]^{1/2}} \quad (14.37)$$

$$= \frac{4(AB - CD)}{[(n^2 - E^2 - F^2)(n^2 - G^2 - H^2)]^{1/2}} \quad (14.38)$$

$$A = \sum_{i=1}^n [\cos(\alpha_i) \cos(\beta_i)] \quad (14.39)$$

$$B = \sum_{i=1}^n [\sin(\alpha_i) \sin(\beta_i)] \quad (14.40)$$

$$C = \sum_{i=1}^n [\cos(\alpha_i) \sin(\beta_i)] \quad (14.41)$$

$$D = \sum_{i=1}^n [\sin(\alpha_i) \cos(\beta_i)] \quad (14.42)$$

$$E = \sum_{i=1}^n \cos(2\alpha_i) \quad (14.43)$$

$$F = \sum_{i=1}^n \sin(2\alpha_i) \quad (14.44)$$

$$G = \sum_{i=1}^n \cos(2\beta_i) \quad (14.45)$$

$$H = \sum_{i=1}^n \sin(2\beta_i) \quad (14.46)$$

JUPP-MARDIA

Listing 14.35: Jupp-Mardia

```

1 FUNCTION JuppMardia (CONST alpha, beta : VectorTyp) : float;
2
3 VAR rCC, rCs, rSC, rSS, r1, r2, x, y : float;
4   cosX, sinX, cosY, sinY           : VectorTyp;
5   n, i                           : WORD;
6   Sig                            : SignificanceType;
7
8 BEGIN
9   n := VectorLength(alpha);
10  IF (n <> VectorLength(beta))

```

## 14. Circular data

```
11   THEN
12   BEGIN
13     ch := WriteErrorMessage('Circular-circular correlation: Vectors of
14       unequal length');
15     CircleError := TRUE;
16     EXIT;
17   END;
18   CreateVector(cosX, n, 0.0);
19   CreateVector(sinX, n, 0.0);
20   CreateVector(cosY, n, 0.0);
21   CreateVector(sinY, n, 0.0);
22   FOR i := 1 TO n DO
23     BEGIN
24       x := GetVectorElement(alpha, i);
25       y := GetVectorElement(beta, i);
26       SetVectorElement(cosX, i, Cos(x));
27       SetVectorElement(sinX, i, Sin(x));
28       SetVectorElement(cosY, i, Cos(Y));
29       SetVectorElement(sinY, i, Sin(y));
30     END;
31   rCC := PearsonProductMomentCorrelation(cosX, cosY, Sig);
32   rCS := PearsonProductMomentCorrelation(cosX, sinY, Sig);
33   rSC := PearsonProductMomentCorrelation(sinX, cosY, Sig);
34   RSS := PearsonProductMomentCorrelation(sinX, sinY, Sig);
35   r1 := PearsonProductMomentCorrelation(cosX, sinX, Sig);
36   r2 := PearsonProductMomentCorrelation(cosY, sinY, Sig);
37   DestroyVector(cosX);
38   DestroyVector(sinX);
39   DestroyVector(cosY);
40   DestroyVector(sinY);
41   Result := (Sqr(rCC) + Sqr(rCS) + Sqr(rSC) + Sqr(RSS) + 2 * (rCC * RSS +
42     rCS * rSC) * r1 * r2
43     - 2 * (rCC * rCS + rSC * RSS) * r2 - 2 * (rCC * rSC + rCS * RSS)
44     * r1)
45     / ((1 - Sqr(r1)) * (1 - Sqr(r2)));
46 END;
```

## 14.8. Test program

Listing 14.36: Test program

```
1 PROGRAM TestCircular;
2
3 USES math,           // standard math LIBRARY
4     MathFunc,        // mathematical functions
```

```

5      Complex,           // complex numbers
6      Vector,            // vector arithmetic
7      Matrix,             // matrix arithmetic
8      Zufall,             // Random numbers
9      Stat,               // statistical significance
10     Deskript,            // descriptive statistics
11     Nonparam,            // non-parametric tests
12     Circular;           // circular statistics
13
14 CONST Length = 100;
15     Steps = 30;
16
17 TYPE Counts = ARRAY [0..Steps] OF WORD;
18
19 VAR x, y, z, xR, yR, zR, xTheta,
20     yTheta, zTheta, a           : float;
21     xMean, yMean, zMean       : ComplexTyp;
22     xVec, yVec, zVec, aVec   : VectorTyp;
23     i                         : WORD;
24     xCounts, yCounts, zCounts : Counts;
25
26 PROCEDURE CreateRandomVectors (VAR xVec, yVec, zVec : VectorTyp;
27                                 VAR xCounts, yCounts, zCounts : Counts);
28 VAR i          : WORD;
29     x, y, z : float;
30
31 BEGIN
32     CreateVector(xVec, Length, 0.0);
33     CreateVector(yVec, Length, 0.0);
34     CreateVector(zVec, Length, 0.0);
35 FOR i := 1 TO Length DO
36     BEGIN
37         x := RandomVonMieses(0.0, 1.0);
38         SetVectorElement(xVec, i, x);
39         INC(xCounts[Round(x/a)]);
40         y := RandomVonMieses(3.0, 10.0);
41         SetVectorElement(yVec, i, y);
42         INC(yCounts[Round(y/a)]);
43         z := RandomUniformCircular;
44         SetVectorElement(zVec, i, z);
45         INC(zCounts[Round(z/a)]);
46     END;
47 END;
48
49 PROCEDURE ReadRandomVectors (VAR xVec, yVec, zVec : VectorTyp;
50

```

## 14. Circular data

```

51                                     VAR  xCounts, yCounts, zCounts : Counts);
52
53     VAR i      : WORD;
54         x, y, z : float;
55         InFile : TEXT;
56
57 BEGIN
58     CreateVector(xVec, Length, 0.0);
59     CreateVector(yVec, Length, 0.0);
60     CreateVector(zVec, Length, 0.0);
61     ASSIGN(InFile, 'Kreis.csv');
62     TRY
63         RESET(InFile);
64     EXCEPT
65         Write('could not open file "Kreis.csv"');
66         ReadLn;
67         HALT;
68     END;
69     ReadLn(InFile); // headline
70     FOR i := 1 TO Length DO
71         BEGIN
72             x := ReadFloat(Infile);
73             SetVectorElement(xVec, i, x);
74             INC(xCounts[Round(x/a)]);
75             y := ReadFloat(Infile);
76             SetVectorElement(yVec, i, y);
77             INC(yCounts[Round(y/a)]);
78             z := ReadFloat(Infile);
79             SetVectorElement(zVec, i, z);
80             INC(zCounts[Round(z/a)]);
81             ReadLn(InFile);
82         END;
83     CLOSE(InFile);
84 END;
85
86 BEGIN
87     inc(ValidFigures);
88     a := Const_2pi/Steps;
89     CreateVector(aVec, Length, 0.0);
90     FOR i := 0 TO Steps DO
91         BEGIN
92             xCounts[i] := 0;
93             yCounts[i] := 0;
94             zCounts[i] := 0;
95         END;
96 // CreateRandomVectors(xVec, yVec, zVec, xCounts, yCounts, zCounts);

```

```

97  ReadRandomVectors(xVec, yVec, zVec, xCounts, yCounts, zCounts);
98  Writeln('      x          y          z');
99  xCounts[Steps] := xCounts[Steps] + xCounts[0]; // deal WITH cross-over
100 yCounts[Steps] := yCounts[Steps] + yCounts[0];
101 zCounts[Steps] := zCounts[Steps] + zCounts[0];
102 FOR i := 1 TO Steps DO
103   Writeln(a*(Pred(i)+i)/2:1:3, ' ', xCounts[i]:4, ' ', ' ',
104           yCounts[i]:4, ' ', ' ', zCounts[i]:4);
105 Writeln;
106 Writeln('Median: ', FloatStr(MedianDirection(xVec), ValidFigures), ' ',
107           FloatStr(MedianDirection(yVec), ValidFigures), ' ',
108           FloatStr(MedianDirection(zVec), ValidFigures));
109 xMean := MeanVector(xVec,1);
110 yMean := MeanVector(yVec,1);
111 zMean := MeanVector(zVec,1);
112 xR := Re(xMean);
113 xTheta := Im(xMean);
114 yR := Re(yMean);
115 yTheta := Im(yMean);
116 zR := Re(zMean);
117 zTheta := Im(zMean);
118 Writeln('R: ', FloatStr(xR, ValidFigures), ' ',
119           FloatStr(yR, ValidFigures), ' ',
120           FloatStr(zR, ValidFigures));
121 Writeln('Theta: ', FloatStr(xTheta, ValidFigures), ' ',
122           FloatStr(yTheta, ValidFigures), ' ',
123           FloatStr(zTheta, ValidFigures));
124 writeln('Variance ', FloatStr(CircularVariance(xR), ValidFigures), ' ',
125           FloatStr(CircularVariance(yR), ValidFigures), ' ',
126           FloatStr(CircularVariance(zR), ValidFigures));
127 writeln('std.dev. ', FloatStr(CircularStandardDeviation(xR),
128           ValidFigures), ' ',
129           FloatStr(CircularStandardDeviation(yR),
130           ValidFigures), ' ',
131           FloatStr(CircularStandardDeviation(zR),
132           ValidFigures));
133 Writeln('kappa: ', FloatStr(Kappa(xR, Length), ValidFigures), ' ',
134           FloatStr(Kappa(yR, Length), ValidFigures), ' ',
135           FloatStr(Kappa(zR, Length), ValidFigures));
136 xMean2 := CenteredMean(TrigonometricMoment(xVec, xTheta, 2));
137 yMean2 := CenteredMean(TrigonometricMoment(yVec, yTheta, 2));
138 zMean2 := CenteredMean(TrigonometricMoment(zVec, zTheta, 2));
139 Writeln('skew: ', FloatStr(CenteredCircularSkew(xMean, xMean2),
140           ValidFigures), ' ',
141           FloatStr(CenteredCircularSkew(yMean, yMean2),
142           ValidFigures), ' ');

```

```

137             FloatStr(CenteredCircularSkew(zMean, zMean2),
138                         ValidFigures));
139             WriteLn('kurtosis:', FloatStr(CenteredCircularKurtosis(xMean, xMean2),
140                         ValidFigures), ' ',
141                         FloatStr(CenteredCircularKurtosis(yMean, yMean2),
142                         ValidFigures), ' ',
143                         FloatStr(CenteredCircularKurtosis(zMean, zMean2),
144                         ValidFigures));
145             WriteLn('delta   :', FloatStr(CircularDispersion(xVec, xMean),
146                         ValidFigures), ' ',
147                         FloatStr(CircularDispersion(yVec, yMean),
148                         ValidFigures), ' ',
149                         FloatStr(CircularDispersion(zVec, zMean),
150                         ValidFigures));
151
152             WriteLn('Raileigh  ', 100*Rayleigh(Re(xMean), Length):3:3, '%      ',
153                 100*Rayleigh(Re(yMean), Length):3:3,
154                 '%      ', 100*Rayleigh(Re(zMean), Length):3:3, '%');
155             WriteLn('Hodges:   ', 100*HodgesAjne(xVec, 0.0):3:3, '%      ',
156                 100*HodgesAjne(yVec, 3.0):1:3,
157                 '%      ', 100*HodgesAjne(zVec, 0.0):1:3, '%');
158             Write('Press return:');
159             ReadLn;
160             DestroyVector(xVec);
161             DestroyVector(yVec);
162             DestroyVector(zVec);

```

## References

- [1] E. BATSCHELET: *Circular Statistics in Biology* Mathematics in biology London: Academic Press, 1981 ISBN: 0-12-081050-6.
- [2] N.I. FISHER: *Statistical Analysis of circular data* Cambridge: Cambridge University Press, 1993 ISBN: 9780521568906.
- [3] P. BERENS: CircStat: A MATLAB toolbox for circular statistics, *J. Stat. Software* **31**:10 (2009), 1–21 DOI: [10.18637/jss.v031.i10](https://doi.org/10.18637/jss.v031.i10).
- [4] R. von MIESES: Über die „Ganzzahligkeit“ der Atomgewichte und verwandte Fragen, *Phys. Z.* **19**:21 (1918), 490–500 URL: [https://digitalisate.sub.uni-hamburg.de/de/nc/detail.html?id=1901&tx\\_dlf%5Bid%5D=36847&tx\\_dlf%5Bpage%5D=522](https://digitalisate.sub.uni-hamburg.de/de/nc/detail.html?id=1901&tx_dlf%5Bid%5D=36847&tx_dlf%5Bpage%5D=522).
- [5] L. BARABESI: Generating von Mises variates by the ratio-of-uniforms method, *Stat. Appl.* **7**:4 (1995), 417–426 URL: <http://sa-ijas.stat.unipd.it/sites/sa-ijas.stat.unipd.it/files/417-426.pdf>.

- [6] J.A. GREENWOOD, D. DURAND: The distribution of length and components of the sum of  $n$  random unit vectors, *Ann. math. stat.* **26**:2 (1955), 233–246 URL: [https://projecteuclid.org/download/pdf\\_1/euclid.aoms/1177728540](https://projecteuclid.org/download/pdf_1/euclid.aoms/1177728540).
- [7] J.S. RAO: Some tests based on arc-lengths for the circle, *Sankhyā: Indian J. Stat., Ser. B* **36**:4 (1976), 329–338 URL: [http://jammalam.faculty.pstat.ucsb.edu/html/Some%20Publications/1976\\_ArcLengthsCircle-Sankhya.pdf](http://jammalam.faculty.pstat.ucsb.edu/html/Some%20Publications/1976_ArcLengthsCircle-Sankhya.pdf).
- [8] G.S. RUSSELL, D.J. LEVITIN: An expanded table of probability values for Rao's spacing test, *Commun. Stat. - Simul. Comp.* **24**:4 (1995), 879–888 DOI: [10.1080/03610919508813281](https://doi.org/10.1080/03610919508813281) URL: <http://cogprints.org/645/1/AnExpand.htm>.
- [9] G.S. WATSON, E.J. WILLIAMS: On the construction of significance tests on the circle and the sphere, *Biometrika* **43**:3/4 (1956), 344–352 DOI: [10.2307/2332913](https://doi.org/10.2307/2332913).
- [10] M.A. STEPHENS: Tests for Randomness of Directions against Two Circular Alternatives, *J. Am. Stat. Assoc.* **64**:325 (1969), 280–289 DOI: [10.1080/01621459.1969.10500971](https://doi.org/10.1080/01621459.1969.10500971).
- [11] K.V. MARDIA: Linear-Circular Correlation Coefficients and Rhythmometry, *Biometrika* **63**:2 (1976), 403–405 DOI: [10.2307/2335637](https://doi.org/10.2307/2335637).



# **Part III.**

# **Multivariate Statistics**



# 15. Definitions and concepts for multivariate statistics

## Abstract

Multivariate statistics is used for dimensionality reduction, anomaly detection, unsupervised (feature extraction) and supervised learning (regression and classification) and for data mining. Incomplete data are a common, perhaps even universal, problem in studies using multivariate statistics.

## 15.1. Tasks for multivariate statistics

In descriptive statistics, a single variable is described for location (mean, median...) and spread (standard deviation, interquartile distance...). In simple regression, we use one independent variable  $\mathbf{x}$  to predict a second, dependent variable  $\mathbf{y}$ . In multivariate regression, each observation  $\mathbf{x}_i$  consists of several variables. The corresponding dependent datum (if any) may be a scalar ( $y_i$ ) or a vector ( $\mathbf{y}_{i, \cdot}$ ).

Multivariate statistics is used for

**dimensionality reduction** many correlated variables are reduced to fewer, uncorrelated ones. [PCA](#) is used for this.

**anomaly detection** out of a large number of data, a few are isolated that come from a different distribution than the rest (outliers detection).

**unsupervised learning** there are only independent data available, aim is to discover the structure of these. Methods used include cluster and factor analysis.

**supervised learning** tries to predict the outcome (dependent) variables from the input (independent) variables  $\hat{\mathbf{y}} = f(\mathcal{X})$ . A data set where both independent and dependent variables are known is used to train the algorithm. We distinguish

**classification** the outcome variable(s) are discrete (binary, ordinal, nominal).

**regression** the outcome variable(s) are cardinal (continuous).

The input data may be of any level, even mixed.

### 15.1.1. Errors

Any prediction value  $\hat{y}$  from multivariate statistics will not be exactly equal to the measured value  $y$ . We distinguish

**bias** is the result of a model that does not exactly describe the data. For example, fitting a straight line to parabolic data results in bias. Sometimes also the data collection was suboptimal. For example, in a study which managers are successful, the result may be biased against female or coloured candidates, simply because most real existing managers are white males. As a result of bias, a model **underfits** the data.

**variance** is the change of estimated parameters we would get if we used a different training data set from the same distribution. Reduction of variance means fitting a model also to the noise in the data, that is, **overfitting**.

**variance of the error term**  $\epsilon$  This term relates to measurement error of  $y$  and, in some cases, also of  $X$ . It may also relate to the influence of additional, unmeasured variables.

The mean squared error (**MSE**) is the sum of the variance, the squared bias and the variance of the error term  $\epsilon$ . Because all three terms (being squares) are necessarily positive, the total **MSE** cannot fall below  $\text{var}(\epsilon)$ , the **irreducible error**. Squared bias and variance together form the **reducible error**, which we try to get as low as possible. As we increase the flexibility of the model (number of parameters), the bias will initially decrease faster than the variance increases, resulting in a decrease of **MSE**. Beyond a certain optimal number of parameters, however, the bias will no longer decrease substantially. Thus, the increasing variance will lead to an increase in **MSE**. If we plot **MSE** against the number of parameters, we get a U-shaped curve. For classification, we use the classification error rate  $\frac{1}{n} \sum_{i=1}^n I(\hat{y}_i \neq y_i)$ . The error calculated from the training data is nearly always lower than that calculated from a separate test data set not used in setting up the model.

### Validation of models

Ideally, we have one data set to calculate the model from (learning set), and a second, independent set for validating the models accuracy (“out of bag observations”). In particular, we want to know whether we have over-fitted the data, that is, whether we may have inadvertently build a model for the noise component.

The problem with this approach is that models become the more precise, the more data are used for their calculation. This becomes the more important, the more parameter the model has. Holding back a sizable fraction of the data for a test set will necessarily increase the bias of the model.

One possible solution is to make the test set as small as possible, reducing it to a single case (**Leave-One-Out Cross-Validation (LOOCV)**). Thus, the model is built from  $n - 1$  cases, then a prediction is made for the remaining case. This process of selecting a test case and building the model for the remaining cases is repeated  $n$  times. The  $n$  models should be quite similar, as removing a single test case should not affect the parameters

too much (if it does, we have identified a **high leverage** point). We also get  $n$  test results, from those we can calculate average deviation  $\text{CV}_n = n^{-1} \sum_{i=1}^n (\mathbf{y}_i - \hat{\mathbf{y}})_i$ .

With least squares linear or polynomial fits, it is actually unnecessary to run the  $n$  fits, because

$$\text{CV}_n = n^{-1} \sum_{i=1}^n \left( \frac{\mathbf{y}_i - \hat{\mathbf{y}}}{1 - \mathbf{h}_i} \right)^2 \quad (15.1)$$

where  $\mathbf{h}_i$  is the leverage of the  $i$ th point from equation 12.31 on page 420, which reflects the amount that an observation influences its own fit.

Alternatively, it is possible to split the data set into  $k$  sets of equal size,  $n - 1$  sets are used for model fitting, one set for calculating the squared errors. This process is repeated  $k$  times. Again, we get  $n$  error estimates, from which we can calculate the average. However, in this  **$k$ -fold cross validation** we had to build the model only  $k$ , not  $n$ , times, which is computationally more effective. In addition, rather than building  $n$  highly correlated models, we build only  $k$ , and their data sets overlap less than in **LOOCV**. That tends to reduce variance, at the expense of a slight increase in bias.  $k$  is empirically set to between 5 and 10, to minimise both bias and variance. It is possible to repeat this process  $l$  times, so that  $l$  estimates are obtained for all  $n \hat{\mathbf{y}}_i$ .

## 15.2. Missing data

### 15.2.1. Classes of missing data

We classify missing data according to seriousness [1, 2]:

**missing completely at random (MCAR)** the probability of a datum  $\mathcal{X}_1$  missing is unrelated to any of the observable variables, the unobserved variables (factors) or the response variable. The subjects with missing data are a random sample of the entire test population. In blind studies, randomness of treatment is assumed to be preserved. Example: sample vials accidentally destroyed during analysis. Both mean imputation and removal of incomplete cases can be performed without biasing the results of a study.

**missing at random (MAR)** the probability of  $\mathcal{X}_1$  missing correlates with either another observed covariate  $\mathcal{X}_2$  or with the predictor or unobserved variable  $\mathbf{y}$ , but not with  $\mathcal{X}_1$  itself. In principle, the missing value can be estimated from the non-missing data (multivariate imputation). Example: The willingness to answer in surveys questions about income may depend on other factors like sex, education, race or age. Thus, missing data on income can be ignored only if income statistics were generated with taking these factors into account (**ignorability assumption**). Otherwise, a non-response bias would result.

**missing not at random (MNAR)** the probability that a variable is missing depends on

**unobserved predictors** For example in medical studies, patients may drop out more frequently if the treatment causes discomfort. Unless discomfort is recorded as additional variable, this will cause bias (underreporting of side effects).

**to the value of the variable  $\mathcal{X}_1$  itself** Example: Students relegated from university in early semesters for poor grades would, on average, perform poorer than their peers in later semesters, if they had been allowed to continue (**non-ignorable non-response**). In the worst case, all cases where  $\mathcal{X}_1$  exceeded a certain threshold are missing, this is called **censoring**.

In **MNAR**, both mean imputation of the missing data and removal of incomplete cases would seriously bias the result.

This can be written as the probability of a datum in  $\mathbf{x}_1$  missing, given the feature  $\mathbf{x}_1$  itself, another correlating feature  $\mathbf{x}_2$  and the predictor variable (or factor)  $\mathbf{y}$ :

$$\text{logit}(\mathbf{x}_1 \text{missing} | \mathbf{x}_1, \mathbf{x}_2, \mathbf{y}) = \begin{cases} \alpha & \text{MCAR} \\ \alpha + \beta \mathbf{x}_2 & \text{MAR } \mathbf{x} \\ \alpha + \beta \mathbf{x}_2 \mathbf{y} & \text{MAR } \mathbf{x}\mathbf{y} \\ \alpha + \beta \mathbf{x}_1 & \text{MNAR } \mathbf{x} \\ \alpha + \beta \mathbf{x}_1 \mathbf{y} & \text{MNAR } \mathbf{x}\mathbf{y} \end{cases} \quad (15.2)$$

Written like this,  $\beta$  would be a measure of the severity of **MNAR** and **MAR**.

It is, however, impossible to mathematically proof which class of missingness is realised in a particular study or in a particular variable. In particular missingness on unobserved variables is difficult to exclude, as by definition we cannot check the influence of “lurking variables” that we have not observed. In the end, we have to rely on our good judgement and do the best we can.

### 15.2.2. Handling missing data

Several methods exist to deal with missing values:

**listwise deletion** uses only complete cases. This method throws away a lot of information and will bias the data in case of **MNAR**. In studies with a large number of variables  $p$  most if not all cases may be removed.

**pairwise deletion** (available case analysis) uses all available data pairs. Example: If a datum  $\mathbf{x}_{ij}$  is missing, the correlation coefficient between  $\mathbf{x}_{.j}$  and any other column of  $\mathbf{X}$  is calculated, ignoring only the  $i$ th row. This results in covariance and correlation matrices that are not positive semi-definite. Standard error estimates are biased. However, all available information is used.

**impute the mean (median, modal) value** of  $\mathbf{x}_{.j}$  for each missing  $\mathbf{x}_{ij}$ . Works with **MCAR** data, but causes bias for **MNAR**. It may also reduce error estimates and increase correlation between variables.

**last value carried forward** For example, in the study of student performance mentioned above, one may take the last available grade of a student and impute that for all missing ones that follow. This ignores any development that may have occurred over time.

**hot-deck imputation** determines the  $k$  nearest neighbours (or another scoring function) of  $\mathcal{X}_i$ , with a datum  $\mathcal{X}_{ij}$  missing, ignoring the  $\mathcal{X}_{.j}$  vector. Then, calculate the average of the available  $\mathcal{X}_{.j}$  in the neighbourhood and use for imputation.

**impute random values** from a distribution that resembles the data.

**multiple imputation** Repeatedly draw imputation values for the missing data from a suitable distribution, perform the analysis and then compare the results. This gives a feeling for the sensitivity of the method on the missing data.

**impute estimated value** For various machine learning algorithms versions are available that can handle missing values, this will be discussed in the respective chapters. In such cases, the method itself may produce imputation values that are iteratively changed to optimise the learning result. This works best for MAR.

## References

- [1] D.B RUBIN: Inference and missing data, *Biometrika* **63**:3 (1976), 581–592 DOI: [10.1093/biomet/63.3.581](https://doi.org/10.1093/biomet/63.3.581).
- [2] T.G. STEWART, D. ZENG, M.C. WU: Constructing support vector machines with missing data, *Wiley Interdisciplinary Reviews: Computational Statistics* **10**:4 (2018), e1430 DOI: [10.1002/wics.1430](https://doi.org/10.1002/wics.1430) URL: <https://onlinelibrary.wiley.com/doi/10.1002/wics.1430> URL: <https://research.fhcrc.org/content/dam/stripe/wu/files/Publications/2018wires-svm.pdf>.



# 16. Principle component analysis

## Abstract

PCA is a method to reduce the dimensionality of data, *i.e.*, the number of variables. In many data sets, several variables are correlated, that is, measure (at least partially) the same thing. This **multi-collinearity** severely interferes with multivariate analysis. Thus, several correlating variables are combined into fewer new variables – called **components** – that contain most of the information, but less noise, than the original variables. These artificial variables can then be used for further analysis.

Although this is theoretically incorrect, PCA is in practice successfully used also as a numerically simple method of **factor analysis (FA)**, that is, the resulting components are interpreted as **factors**, the underlying causes for the correlation of variables. To ease interpretation, components may be rotated in space to achieve a **simple structure**, we then speak of **rotated components**.

## 16.1. Principal component analysis (PCA) *vs.* factor analysis (FA)

The following sources may offer additional help: [1–5]. If two observed variables  $\mathbf{x}, \mathbf{y}$  are correlated, then a change in a third, underlying (confounding), latent (unobserved) variable (**factor**) may be the cause (see fig. 16.1). Factor analysis is used to uncover such factors from an observed data set:

$$z_{i,j} = \psi_{i,j} + \sum_{k=1}^p \mathcal{L}_{j,k} \mathcal{C}_{i,k} \quad (16.1)$$

with:

$z_{i,j}$  result of person  $i$  on item  $j$ , in standard deviations from mean

$\psi_{i,j}$  measurement error of person  $i$  on item  $j$

$\mathcal{L}_{j,k}$  loading of factor  $k$  on variable  $j$ , that is, the correlation of  $\mathbf{f}_{j,k}$  with variable  $j$

$\mathcal{C}_{i,k}$  factor score of student  $i$  on factor  $k$  ( $\bar{\mathcal{C}} = 0, \sigma_{\mathcal{C}} = 1$ )

## 16. Principle component analysis

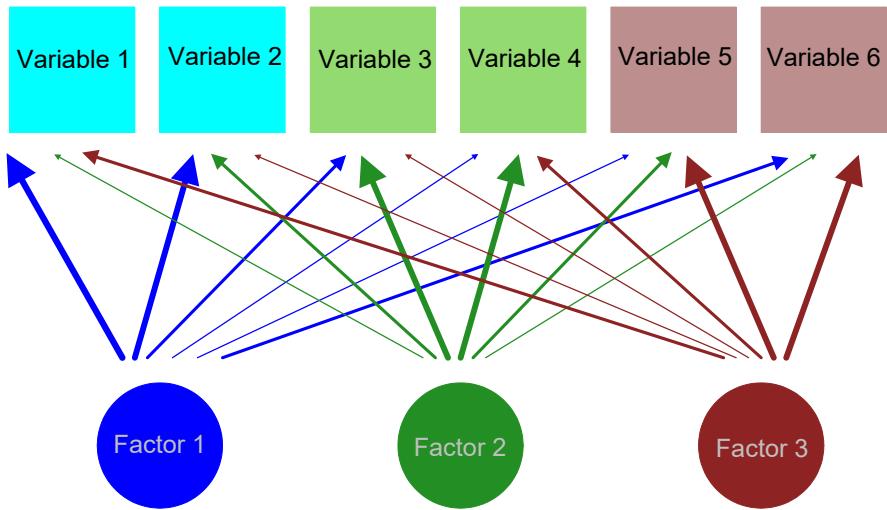


Figure 16.1.: The fundamental idea of factor analysis. The observed variables are determined by underlying (unobserved) factors. Each factor has effects on several variables, each variable is determined by several factors. How strongly a factor affects a variable – the factor loadings on that variable – is represented by the weight of the arrows. In principal component analysis, the direction of the arrows is reversed.

The number of latent factors  $p$  is initially equal to the number of observed variables. However, often only a small number  $q$  of the factors is significant. The **eigenvalue** for each factor is the sum of the squared loadings of a factor on all variables. It explains how much of the variance  $\sigma$  in the data is explained by that factor (column sum to the loading matrix). The data are usually  $z$ -standardised and hence have a variance of 1.0. An eigenvalue of 1.0 therefore means that the factor explains as much variance as an observed variable. The ratio of eigenvalues is ratio of explanatory power of two factors. One can therefore use only those factors with significant eigenvalues, the remainder are thought to originate from experimental noise. The **communalities**  $\mathfrak{h}$  are the sum of the squared loadings of a variable over all factors (row sum of the squared loading matrix). It is the proportion of variance of the variable that can be explained by the factors. In the literature one may find the **uniqueness** instead, which is the  $1 - \mathfrak{h}$ .

The differences between Factor analysis and principle component analysis are:

- **PCA** produces components, **FA** factors. Components are a linear combination of observed variables, variables are a linear combination of the underlying factors.
- **PCA** assumes that the data matrix is error free (all diagonal elements of the correlation matrix are unity), the components therefore contain error and are therefore not equal to latent variables (they are not necessarily interpretable). In **FA** the data are assumed to contain error (diagonal elements of the correlation matrix contain communalities), and the factors extracted contain only the common variance (variance shared with other variables), but not the unique variance (variance

## 16.2. Is a data set suitable for factor analysis and PCA?

specific to a particular variable, which is taken as error term). As a result, the correlation matrix is reconstructed by the components, but only approximated by the factors.

- PCA is used to describe empirical data in causal modelling and to reduce the number of dimensions to ease further analysis (*e.g.*, by cluster analysis or multi-dimensional regression). FA is used to theoretically explore the underlying factor structure.
- In fig. 16.1 the arrows have opposite direction for PCA.
- PCA tries to explain as much as possible of the variance of the variables ( $\sum_{i=1}^p s_{ii}$ ) by reducing the elements of  $\mathcal{R}$  near the diagonal. FA tries to account for the covariance between variables ( $\sum_{i,j=1}^p s_{ij}, i \neq j$ ) by reducing the elements far from the diagonal. However, since the one cannot be achieved without the other, both methods do in practice nearly the same thing.

The communality is the variance a variable shares with all other variables. With a high enough number of variables and subjects, the method used seems to have little impact on the results obtained (at least if the communalities are  $> 0.4$ ), so PCA is the method of choice when there is no specific reason to use another ([2], but see also [6] for a different opinion).

## 16.2. Is a data set suitable for factor analysis and PCA?

### 16.2.1. Number of data

There are several criteria available. As a rule of thumb [2], there should be at least 6, better more (up to 20 has been suggested) subjects per variable, so if 100 variables are studied, 600 subjects are required, 2000 would be ideal. Each factor should have a chance to be represented by several variables, for most studies that means that there should be  $> 30$  variables.

The sampling density (average distance between data points) is proportional to  $n^{\frac{1}{p}}$ , thus the number of samples  $n$  required for a reasonable representation of the data space increases rapidly with  $p$ .

### 16.2.2. Correlation coefficients

Most (off-diagonal) correlation coefficients should be  $0.3 < r < 0.8$ , with around 0.5 being ideal.

### 16.2.3. Multivariate normality of data

Both FA and PCA require that the data are distributed multivariate normal, that is, every linear combination of its  $p$  components has a univariate normal distribution ( $ax_i +$

## 16. Principle component analysis

Table 16.1.: Interpretation of MSA-values according to [12]

KMO	Suitability
> 0.90	marvellous
0.80..0.90	meritorious
0.70..0.89	middling
0.60..0.69	mediocre
0.50..0.59	miserable
< 0.50	unacceptable

$bx_{\cdot j}$  is normally distributed  $\forall i, j \in [1 \dots p]$  and  $a, b \in \mathbb{R}$ ). [7] recommends the procedures in [8] (FORTRAN-code available in [9]) and [10] to test this, both are available in the R-package MVN [11]. It is also possible to test for normality of each variable, this must be true if multivariate normality is true. Note, however, that normality of the individual data is necessary but not sufficient for multivariate normality of the entire set (see for example fig. 10.4 on page 335).

### 16.2.4. The KAISER-MEYER-OLKIN (KMO) criterium

For a correlation matrix  $r$  to be useful for factor analysis,  $r^{-1}$  should be near diagonal. The KMO-criterium measures how close  $r^{-1}$  is to being diagonal:

$$KMO = \frac{\sum_{i=1}^n \sum_{j=1}^n r_{ij}^2}{\sum_{i=1}^n \sum_{j=1}^n r_{ij}^2 + \sum_{i=1}^n \sum_{j=1}^n q_{ij}^2}, \quad i \neq j \quad (16.2)$$

with  $r_{ij}$  the correlation of variables  $i$  and  $j$  and  $p_{ij}$  the partial correlation [13]:

$$\mathcal{Q} = \mathcal{D}r^{-1}\mathcal{D} \quad (16.3)$$

$$\mathcal{D} = [(\text{diag } r^{-1})^{1/2}]^{-1} \quad (16.4)$$

A large partial correlation means that the correlation matrix has little common variance and results in a small KMO-value (see table 16.1). It is also possible to use the **measure of sampling adequacy (MSA)** to determine whether a particular variable  $j$  should be included in the analysis:

$$MSA_j = \frac{\sum_{i \neq j} r_{ij}^2}{\sum_{i \neq j} r_{ij}^2 + \sum_{i \neq j} q_{ij}^2} \quad (16.5)$$

If variables with a poor MSA are removed from the analysis, the overall KMO increases.

However, as shown by [14, p. 147], MSA is affected not only by the quality of  $r$ , but also decreases with the number of significant factors. It therefore is of limited value.

### 16.2.5. BARTLETT's sphere test

Tests  $H_0 : r = I$  (the identity matrix) vs  $H_1 : r \neq I$  [15]. If  $H_0$  were true then there is no covariance between data, and the data points form a perfect sphere in  $p$ -dimensional space. Then of course there would be no principal component, all eigenvalues are identical except for noise.

$$\chi^2 \approx -\left[(n-1) - \frac{2p+5}{6}\right] \times \log(\|\mathcal{R}\|) \quad (16.6)$$

$$f = \frac{p(p-1)}{2} \quad (16.7)$$

with  $\|\mathcal{R}\|$  the determinant of the correlation matrix (which would be unity if  $r = I$ ). The correlation matrix is suitable if  $P_0 < 5\%$ . The test is sensitive to deviation of the data from multivariate normal distribution, which may lead to false acceptance of  $H_0$ .

BARTLETT's test compares the volume of the data ellipsoid with the volume of the spheroid that would result if all axes were identical. Another way to calculate  $\chi^2$  is via the ratio between the geometric and arithmetic mean of all eigenvalues of the correlation matrix (which would be unity if the data were spherical):

$$r = \frac{\tilde{\lambda}}{\bar{\lambda}} = \frac{\sqrt[p-k]{\prod_{i=k+1}^p \lambda_i}}{\frac{1}{p-k} \sum_{i=k+1}^p \lambda_i} \quad (16.8)$$

$$\chi^2 = -(n - (p - k) - 0.5) \times \ln(r^{p-k})$$

$$f = \frac{(p - k - 2)(p - k - 1)}{2}$$

with  $k$  the number of eigenvalues excluded from the analysis (in this context 0, but see eqn. 16.21, where this becomes important).

Listing 16.1: BARTLETT's sphere test

```

1  function Bartlett (const Cor : MatrixTyp; Cases : word) : float;
2
3  var f, Variables : word;
4      chi2, Det, P0 : extended;
5
6  begin
7      Variables := MatrixRows(Cor);
8      Det := Determinante(Cor);
9      writeln('Determinante = ', det:10, ' ');
10     chi2 := -(pred(Cases) - (2 * Variables + 5) / 6) * log(Det, 10);
11     f := round(Variables * pred(Variables) / 2);
12     writeln('chi2 := ', chi2:5:3, ' ', 'f = ', f:5, ' ');
13     P0 := IntegralChi(chi2, f); // should be larger than 0.95
14     writeln('P0 = ', (100 * P0):3:1, '%');
15     Bartlett := P0;
16 end;

```

## 16. Principle component analysis

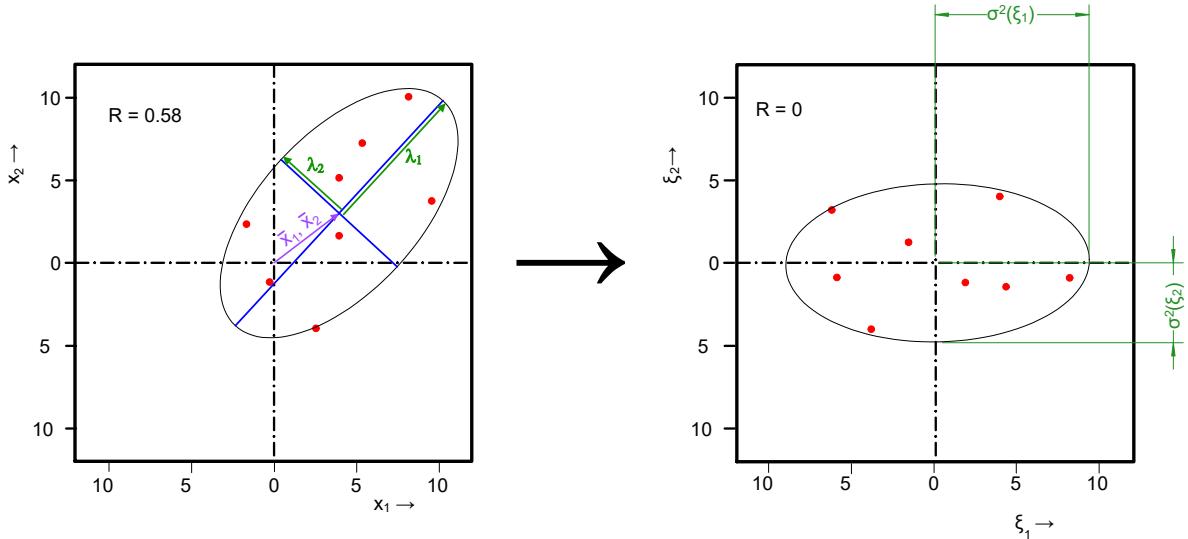


Figure 16.2.: Geometrical interpretation of PCA, for a two-dimensional variable set. Each attribute carrier is represented by a (red) dot in the  $x_1, x_2$ -plane. All data form an ellipsoid cloud, with the ellipsoid representing a surface of identical density. Transformation occurs by centering the ellipsoid (subtracting the vector of averages  $\bar{x}_1, \bar{x}_2, \dots, \bar{x}_p$ , purple) and rotating the coordinate system so that the first principal component (green) is the largest radius of that ellipsoid, the eigenvalue is the length of this vector. The second principal component is orthogonal to the first, and represents the second-largest radius of the ellipsoid, and so on for higher dimensions.

### 16.3. Mathematical basis of principle component analysis

PCA was invented by HOTELLING [16], using the work of several authors. The data in a data matrix  $\mathbf{X}$  can be interpreted as points in a  $p$ -dimensional cloud (see fig. 16.2). The data points are enveloped by an ellipsoid (ellipse in the two-dimensional case), which represents a surface of equal density.

The first principle component is, in effect, a kind of regression line through the data cloud (actually, linear regression minimises the distances from the regression line as measured orthogonal to the  $x$ -axis, in PCA they are measured orthogonal to the component, see fig. 16.3). If we calculate the distance of all data points from the regression line, the second principle component would be a regression through those, and so on. The first component coincides with the major axis of the ellipsoid. This vector has  $p$  elements and is called an *eigenvector*. In EUKLIDIAN space, the length of a vector is the root of the sum of the squares of the elements (see eqn. 16.2), this length is called *eigenvalue*.

For the data matrix a variance-covariance matrix  $\mathcal{S} = \mathbf{X}^T \mathbf{X} / (p - 1)$  (if the data are z-standardised, otherwise the correlation matrix  $\mathcal{R}$  is used) is calculated.

### 16.3. Mathematical basis of principle component analysis

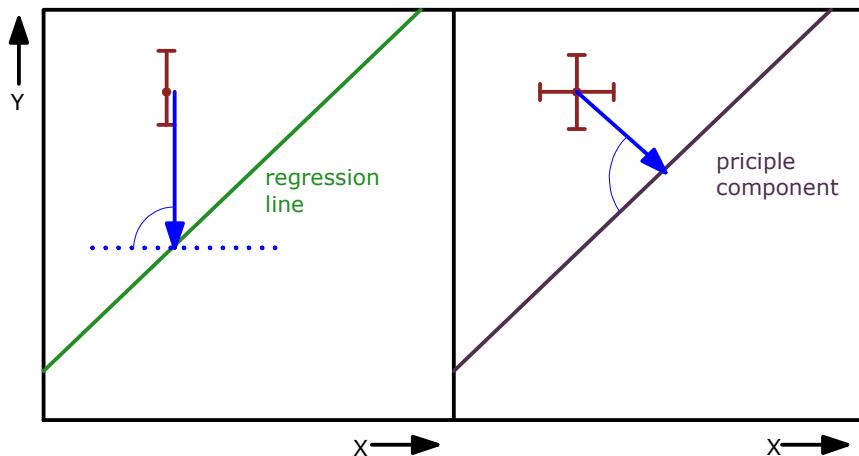


Figure 16.3.: Distance of a point to the regression line (*green, left*) and the principle component (*purple, right*) for the 2D case. In linear regression, the independent variable  $x$  is assumed to be error free, and the distance of a data point to the regression line is measured in the direction of the dependent variable  $y$  (orthogonal to the  $x$ -axis). In PCA all variables are equivalent, measurements for each individual have errors in both  $x$ - and  $y$ -direction. The distance of the data point to the principle component is orthogonal to that component (as in DEMING-regression with  $\sigma = 1$ , see section 12.5 on page 411).

Figure 16.4.: There is only one direction of the principle axis that maximises the spread of the projection of the data (variance), in this position the distance of the data points from the axis is minimal. Data points in blue, projections in red. One can think of the distances as springs, which apply a force onto the principle component according to HOOKE's law. Then the component will orient itself so that the overall force is zero. Figure from <https://stats.stackexchange.com/questions/2691/making-sense-of-principal-component-analysis-eigenvectors-eigenvalues?rq=1>.

## 16. Principle component analysis

Aside: Interconversion of  $\mathcal{S}$  and  $\mathcal{R}$ :  $r_{ij} = \frac{s_{ij}}{\sqrt{s_i s_j}}$ . Hence if  $\mathcal{A} = \text{diag}(s_1, s_2, \dots, s_p)$  a diagonal matrix of variances, then  $\mathcal{S} = \mathcal{A}\mathcal{R}\mathcal{A}$  is the variance-covariance matrix and  $\mathcal{R} = \mathcal{A}^{-1}\mathcal{S}\mathcal{A}^{-1}$ .

The sum of squares for both data vectors in the example is 500. For this purpose the data matrix  $\mathbf{x}_{n \times p}$  is multiplied with a weight matrix  $\mathbf{f}$ , resulting in a factor score matrix  $\mathbf{c}$ :  $\mathbf{c} = \mathbf{x}\mathbf{f}$ . Then the sums of squares (= eigenvalues) are now differently distributed (400 and 100 instead of 208 and 292, respectively), but their sum is the same. Also, the correlation between  $c_1$  and  $c_2$  is now 0, the components are independent. The eigenvalues are arranged in a diagonal matrix  $\Lambda$ :

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} = \begin{pmatrix} 400 & 0 \\ 0 & 100 \end{pmatrix} \quad (16.9)$$

$\mathbf{f}$  represents tests,  $\mathcal{C}$  persons,  $\mathbf{x} = \mathcal{C}\mathbf{f}'$ .

How do we get  $\Lambda$ ? From the *characteristic equation*:

$$\|\mathcal{S} - \lambda_i \mathcal{I}\| = 0 \quad (16.10)$$

with  $\|\cdot\|$  the determinant and  $\mathcal{I}$  the identity matrix (all diagonal elements 1, all other elements 0). Thus, in our example the determinant of:

$$\begin{pmatrix} 208 & 144 \\ 144 & 292 \end{pmatrix} - \lambda_i \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 208 - \lambda_i & 144 - 0 \\ 144 - 0 & 292 - \lambda_i \end{pmatrix} \quad (16.11)$$

This leads to

$$\lambda_i^2 - 500\lambda_i + 40000 = 0 \quad (16.12)$$

which, when solved for  $\lambda_i$  gives (400, 100). For each  $\lambda_i$  we can calculate the corresponding latent vector  $\mathbf{f}_i$  from

$$(\mathcal{S} - \lambda_i \mathcal{I})\mathbf{f}_i = 0 \quad (16.13)$$

which is for  $\lambda_1 = 400$ :

$$\left[ \begin{pmatrix} 208 & 144 \\ 144 & 292 \end{pmatrix} - 400 \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right] \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} = \begin{pmatrix} -192 & 144 \\ 144 & -108 \end{pmatrix} \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad (16.14)$$

Where do the equations

$$\|\mathcal{S} - \lambda_i \mathcal{I}\| = 0 \quad (\mathcal{S} - \lambda_i \mathcal{I})\mathbf{f}_i = 0 \quad (16.15)$$

come from? If a vector  $\mathbf{x}$  is left-multiplied with an array  $\mathcal{A}$ , another vector  $\mathbf{y}$  is obtained, which is called a transformation of  $\mathbf{x}$ :  $\mathcal{A}\mathbf{x} = \mathbf{y}$ . In the case of PCA,  $\mathbf{x}$  maintains its direction, in our example:  $\mathcal{S}\mathbf{f} = \lambda_i \mathbf{f}$ . This is equivalent to

$$\mathcal{S}\mathbf{f} - \lambda_i \mathbf{f} = 0 = (\mathcal{S} - \lambda_i \mathcal{I})\mathbf{f} \quad (16.16)$$

Unless the determinant  $\|\mathcal{S} - \lambda_i \mathcal{I}\| = 0$  the vector  $\mathbf{f}$  will be 0.

Considering fig. 16.2, there are a few points on the ellipse where the normal of the tangent has the same direction as the line connecting the point to the focus of the ellipse. This identity determines a latent vector.

To do a **PCA** perform the following steps:

### 16.3. Mathematical basis of principle component analysis

1. Arrange data in a matrix with persons in rows and variables in columns ( $\mathbf{X}_{n \times p}$ ).
2. Calculate the arithmetic means and standard deviation of all variables of  $\mathbf{X}$  (that is, of all columns).
3. Center the data by subtracting the column means from all columns.
4. From the centered data matrix  $\mathcal{X}$  calculate the variance-covariance matrix  $\mathcal{S}$ ,  $s_{i,j} = \frac{\sum_{k=1}^n (x_{k,i} - \bar{x}_i) \times \sum_{k=1}^n (x_{k,j} - \bar{x}_j)}{n-1}$  or the correlation matrix  $\mathcal{R}$ . The variance-covariance matrix may also be calculated by  $\mathcal{S} = \mathcal{X}^T \mathcal{X} / (p-1)$ , the correlation matrix from  $\mathcal{R} = \mathcal{Z}^T \mathcal{Z}$ .
5. Perform an eigenanalysis of either matrix, resulting in the diagonal matrix of eigenvalues  $\Lambda$  and the corresponding eigenvectors (components)  $\mathcal{E}_{p \times p}$ . Note that using  $\mathcal{R}$  or  $\mathcal{S}$  will yield different results unless the data matrix is not only centered, but z-standardised.
6. Calculate the proportion of explained variance by dividing each eigenvalue  $\lambda_i$  by the sum of all eigenvalues  $\sum_{j=1}^p \lambda_j = p$ , and its cumulative sum.
7. Calculate the acceleration  $f''(j) \approx (f(j+1) - 2f(j) + f(j-1))$ , the second derivative of the scree-plot
8. Calculate the product of the eigenvalues  $\prod_{i=1}^p \lambda_i = \|\mathcal{R}\|$  required for the sphericity test (see above).
9. Generate the diagonal matrix  $\Lambda$  with eigenvalues on the diagonal. This is the variance-covariance of the components.
10. Determine the number of significant eigenvalues  $q$ , and keep the corresponding eigenvectors in a projection matrix  $\mathcal{F}_{p \times q}$  column-wise from left to right.
11. Calculate the factor score matrix (principle components) by  $\mathcal{C}_{n \times q}^T = \mathcal{X}_{n \times p} \mathcal{F}_{p \times q}$ .
12. Calculate the correlation between variables and factors, the **loadings**  $\mathcal{L}_{p \times q}$ . This is the proportion of variance in a variable that is accounted for by a factor. For each variable, the **communality**, is calculated as
$$h_j = \sum_{k=1}^q l_{j,k}^2$$
, the row sum of squared loadings. Sometimes, in the literature the **uniqueness** =  $1 - h_j$  is used instead.
13. Likewise, the column sum of squared loadings is the **sum of squared loadings**. The SSLoadings divided by its total is **variance accounted for**.

## 16. Principle component analysis

14. HOFFMAN's index of complexity [17] for each item is  $\frac{(\sum_{k=1}^q l_{jk}^2)^2}{\sum_{j=1}^q l_{jk}^4}$ .
15. If PCA is used as a method of factor analysis, identify for each factor the  $\approx 5$  variables that have the highest and lowest correlation and use these to interpret the factors.
16. A rotation of the coordinate system can be tried to obtain more interpretable components. In that case, the scores have to be recalculated to match the rotated loadings. In this case, one should no longer speak of "principal components", but of "rotated components".

Asside: PCA can also be performed using singular value decomposition on the centered data matrix ( $\mathcal{X}_{n \times p} = \mathcal{P}_{n \times p} \Delta_{p \times p} \mathcal{Q}_{p \times p}^T$ ). Columns of  $\mathcal{P}$  contain the left singular vectors, and  $\mathcal{F} = \mathcal{P}\Delta$ . These singular vectors are the normalised eigenvectors of  $\mathcal{X}\mathcal{X}^T$ . Columns of the loading matrix  $\mathcal{Q}$  contain the right singular vectors, and  $\mathcal{X}\mathcal{Q}$  gives the projections of data onto the principal components. The vectors in  $\mathcal{Q}$  are the normalised eigenvalues of  $\mathcal{X}^T\mathcal{X}$ . The diagonal matrix  $\Delta$  contains the singular values and is related to eigenvalues by  $\lambda_i = \delta_i^2$ , where  $\Lambda$  contains the eigenvalues of both  $\mathcal{X}\mathcal{X}^T$  and  $\mathcal{X}^T\mathcal{X}$ , which are identical.

The **non-linear iterative partial least squares (NIPALS)** algorithm calculates the first eigenvector and the first score column of a data matrix, their outer product is subtracted from the data matrix, the resulting residual matrix is used to calculate the second eigenvector/loading vector and so on. With very large data sets (e.g., in -omics), which have only a few significant components, this can lead to significant savings in computer time, however, the algorithm is sensitive to roundoff and cancellation errors.

### 16.3.1. Calculation of component and factor scores

Once the loading matrix  $\mathcal{L}$  has been calculated and – if desired – rotated, the next step is the calculation of the scores each individual has on the  $q$  relevant factors. As these scores are free of co-linearity, they can be used as variables for further analysis, for example regression or cluster analysis. The dimension of the score matrix  $\mathcal{C}$  is  $n \times q$ . For this purpose, a matrix of regression coefficients  $\mathcal{A}_{p \times q}$  is calculated, then  $\mathcal{C} = \mathcal{X}\mathcal{A}$  or, better,  $\mathcal{Z}\mathcal{A}$  if the correlation matrix or the centered values of  $\mathcal{X}$  if the covariance matrix was used. The regression matrix may be saved and used to calculate scores for new observations. The scores calculated have a variance of unity (or nearly so). There are several methods to calculate  $\mathcal{A}$  [18]:

#### The coarse (CATTELL's) method

$\mathcal{A} = \mathcal{L}$ . The data matrix  $\mathcal{X}$  should be centered (if the covariance-matrix  $\mathcal{S}$  was used to calculate  $\mathcal{L}$ ) or z-standardised (if  $\mathcal{R}$  was used). It is possible to set low values of  $\mathcal{A}$  to zero, or even to dichotomise all elements of  $\mathcal{L}$  to either zero or unity. The main value of this method used to be that it does not require matrix inversion and hence is computationally cheap, but with the increased power of personal computers that is

no longer an issue. Score values also tend to be more stable from sample to sample if sampling is not ideal, the coarse method is hence used mainly in exploratory analysis, where the reliability and validity of the factors has not yet been tested.

### The refined methods

After unrotated PCA one can simply use  $\mathcal{A} = \mathcal{L}\Lambda^{-1} = \mathcal{R}^{-1}\mathcal{L}$ , that is, each column of  $\mathcal{L}$  is divided (scaled) by the corresponding eigenvalue. The method results in scores that have a variance of unity. If rotation was performed,  $\mathcal{A} = (\mathcal{L}^+)^T$ , the transposed pseudoinverse.

After factor analysis several methods are available, including:

**Regression method** [19, 20]  $\mathcal{A} = \mathcal{R}^{-1}\mathcal{L}\mathcal{I}$ . This formula may also be used in PCA (and is the only method available in SPSS after PCA extraction).

**HORST's method** [21]  $\mathcal{A} = \mathcal{L}\mathcal{L}^T\mathcal{L}\mathcal{I} = (\mathcal{L}^+)^T$ .

**ANDERSON-RUBIN's method**  $\mathcal{A} = \mathcal{L}^T\mathcal{U}^{-1}\mathcal{R}\mathcal{U}^{-1}\mathcal{L}$ . The scores have a variance of exactly one. This method can be used only after orthogonal, not oblique, rotations.

**MCDONALD-ANDERSON-RUBIN's method**  $\mathcal{A} = \mathcal{R}^{1/2}\mathcal{G}\mathcal{H}^T\mathcal{I}^{1/2}$ , where  $\text{svd}(\mathcal{R}^{1/2}\mathcal{U}^{-1}\mathcal{L}\mathcal{I}^{1/2}) = \mathcal{G}\Delta\mathcal{H}^T$ . This method is an extension of Anderson-Rubin's method that works after both orthogonal and oblique rotations.

**GREEN's method** is a modification of MCDONALD-ANDERSON-RUBIN's method, where  $\text{svd}(\mathcal{R}^{1/2}\mathcal{L}\mathcal{I}^{3/2}) = \mathcal{G}\Delta\mathcal{H}^T$ . This method may also be used after PCA, giving the same result as HORST's.

## 16.4. Pascal program for PCA

The main program then is as follows:

Listing 16.2: Main program for PCA

```

1 PROGRAM Principal;
2
3 USES
4   Math,           // Lazarus math UNIT
5   mathfunc,       // basic mathematical routines FOR real AND INTEGER
6   Complex,        // routines FOR complex numbers
7   Vector,         // vector algebra
8   Matrix,         // basic matrix algebra
9   stat,           // statistical TEST distributions
10  Correlations,  // various types OF correlation coefficients
11  PCA,            // principal component analysis
12  PerformShrinkage, // Ledoit-Wolf shrinkage OF correlation matrix
13  Rotation,       // rotation OF loading matrix
14  SystemSolve,    // linear equations

```

## 16. Principle component analysis

```
15 EigenValues,           // calculation OF eigenvalues
16 Zufall;                // generate Random numbers
17
18 VAR
19   Types                      : TypeVector;
20   EV, Variance, CumVariance, InputVector, Acc,
21   Communalities, Uniqueness, VarianceAccounted      : VectorTyp;
22   Data, Cor, EigenVectoren, Scores, Identity, Loadings : MatrixTyp;
23   Freqs                     : FreqsType;
24   x                          : float;
25   Cases, i, iter             : WORD;
26
27
28 BEGIN
29   ProblemName := 'Anonym';
30   SigVecs := 40;
31   ReadCSV(Types, Data);
32   CalculateCorrelationMatrix(Data, Types, Cor);
33   WriteCorrelations(Cor, FALSE);
34   Cases := MatrixRows(Data);
35   Average(Cor, 0.5);          // perform shrinkage by average
36   correlation
37   WriteCorrelations(Cor, TRUE);
38   AnalyseFrequencies(Cor, Types, Freqs);
39   WriteCorrelations(Cor, TRUE);
40   x := Bartlett(Cor, Cases);
41   i := Jacobi(Cor, EV, EigenVectoren, Iter); // eigenanalysis
42   Writeln('Jacobi: ', Iter: 3, ' iterations, Result = ', i:2, ': ',
43         EigenError[i]);
44   DestroyMatrix(Cor);
45   SortEigenValues(EV, EigenVectoren);
46   ExplainedVariance(EV, Acc, Variance, CumVariance);
47   WriteEigenVector(EigenVectoren);
48   Writeln('Eigenvectors and -values written');
49   MaximumLikelihood(Data, Types, InputVector);      // Scores
50   RobustProduct(Data, EigenVectoren, InputVector, Scores);
51   WriteComponentScores(Scores);
52   Writeln('scores written');
53   CalculateLoadings(Data, Scores, Types, Loadings); // unrotated Loadings
54   CalculateCommunalities(Loadings, Communalities, Uniqueness,
55                           VarianceAccounted);
56   WriteLoadings(Loadings, Communalities, Uniqueness, VarianceAccounted,
57                 FALSE);
58   Writeln('Loadings and communalities calculated and written');
59   DestroyVector(Variance);
60   DestroyVector(CumVariance);
```

```

57 DestroyVector(Communalities);
58 CreateIdentityMatrix(Identity, SigVecs);
59 GradProjAlgOrth(Loadings, Identity, Varimax);      // rotation
60 CalculateCommunalities(Loadings, Communalities, Uniqueness,
   VarianceAccounted);
61 WriteLoadings(Loadings, Communalities, Uniqueness, VarianceAccounted,
   TRUE);
62 DestroyMatrix(Identity);
63 DestroyMatrix(Data);
64 DestroyMatrix(EigenVectoren);
65 DestroyMatrix(Scores);
66 DestroyMatrix(Loadings);
67 DestroyVector(EV);
68 DestroyVector(ImputVector);
69 DestroyVector(Communalities);
70 Writeln('All done, press <CR> to finish:');
71 ReadLn;
72 END.
```

The actual beef is in units PCA, PerformShrinkage and Rotation.

## 16.5. Example: Phylogenetic trees from nucleic acid sequences

All life on earth is thought to originate from a single primordial Archaea-like cell that arose about  $3.5 \times 10^9$  years ago. This common origin is reflected by sequence similarities in the sequences of common genes. More closely related species have more similar sequences than species whose last common ancestor lived long ago. Thus, it is possible to derive the *tree of life* from sequence comparison, this is a task of bioinformatics.

Usually, sequences are compared by aligning them and then counting differences (insertions, deletions, mutations). However, multiple sequence alignment is computationally expensive, the problem is NP-complete. There are heuristic solutions to the problem implemented in programs like **clustal** [22] or **MUSCLE** [23].

It has been shown that a lot of information about a nucleic acid sequence is contained in the word count [24, 25]. For example, the sequence “acg” contains two words of length two: “ac” and “cg”. Thus, the result of this operation is a 2D data matrix with  $n$  rows (**OTUs**) and the possible words (16, 64, 256, 1024, for word lengths of 2, 3, 4, 5, respectively) in columns. To avoid too many empty cells, word length should be  $< \log_4(m)$ , where  $m$  is the sequence length. The resulting frequency table can be either used directly to calculate a distance matrix, or subjected to **PCA** first. In the latter case, the first two principal components are sufficient for clustering (see fig. reffig:PincComp).

## 16. Principle component analysis

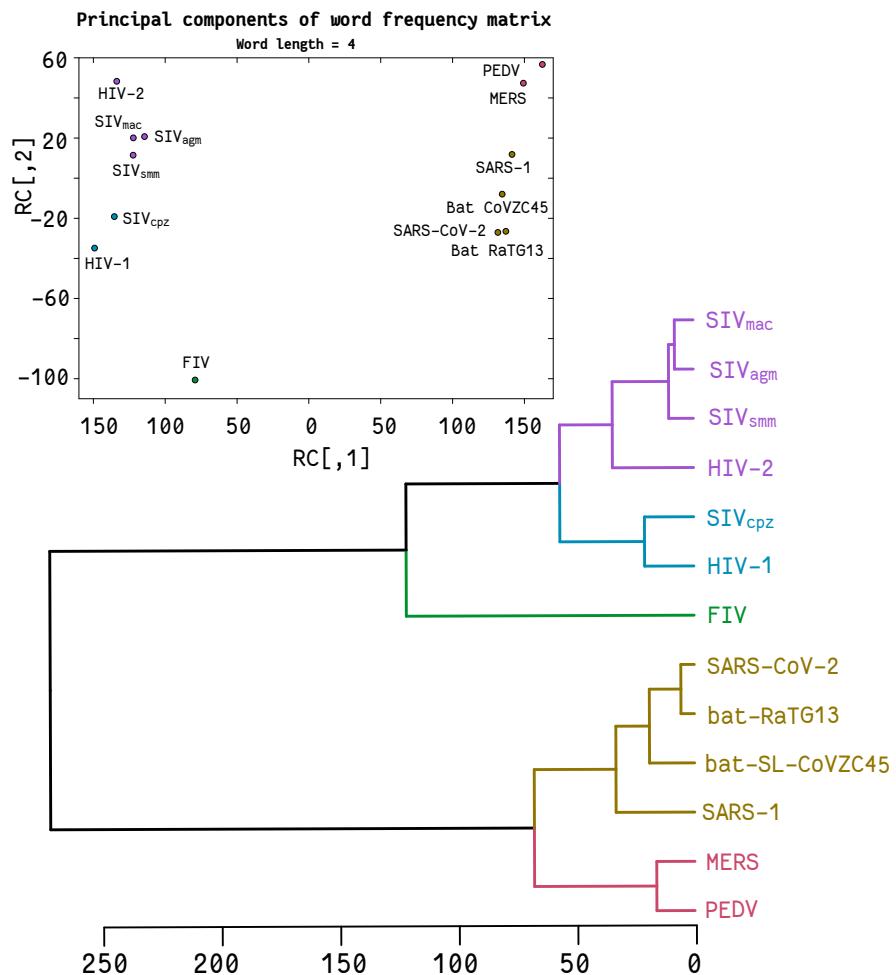


Figure 16.5.: Phylogenetic tree of Corona- and Lentivirus, using nucleic acid word frequencies (word length = 4). The first principal component of that matrix corresponds to the virus family, the second to similarity within a family. These principal components are sufficient to calculate a distance matrix and a hierarchical tree that agrees well with our knowledge of virus evolution. Compared to Clustal, which required almost 4 h to calculate a similar tree by multiple sequence alignment, this calculation took only seconds. For details see text.

## 16.6. The unit PCA

The unit **PCA** contains routines used by the program **Principal**. The interface is

Listing 16.3: Interface of unit PCA

```

1  UNIT PCA;
2
3  { Performs routines in the context of principle component analysis }
4
5  INTERFACE
6
7  USES Math, Mathfunc, Stat, Correlations, Vector, Matrix;
8
9  CONST
10 MaxVariables = 180;
11 SepChar = ';';
12 // IN Middle Europe variables IN CSV separated by ";" AS "," IS decimal
   separator
13 SigVecs: WORD = 5;
14 // number OF significant vectors, can be changed by calling PROGRAM
15 ProblemName: STRING = ''; // first name OF all files produced;
16 Border = 20; // number OF ranges FOR statistical analysis OF correlations
17
18 TYPE
19   DataTypes = (binary, nominal, ordinal, interval, rational);
20   TypeVector = ARRAY[1..MaxVariables] OF DataTypes;
21   FreqsType = ARRAY[DataTypes, DataTypes, -Border..Border] OF WORD;
22
23
24 PROCEDURE ReadCSV(VAR Types: TypeVector; VAR Data: MatrixTyp);
25 { Read data from CSV file. }
26
27 PROCEDURE CalculateCorrelationMatrix(CONST Data: MatrixTyp;
28   CONST Types: TypeVector; VAR Cor: MatrixTyp);
29 { calculates correlation matrix for a mixed type data matrix. NaN-values
30 are handled. }
31
32 PROCEDURE ReadCorrelations(VAR Cor: MatrixTyp);
33 { If the correlation matrix has been calculated previously,
34   read it from CSV file }
35
36 PROCEDURE WriteCorrelations(CONST Cor: MatrixTyp; Shrunk: BOOLEAN);
37 { Writes the correlation matrix into a csv-file. File name will depend on
38   whether the matrix has been shrunk.}
39
40 PROCEDURE AnalyseFrequencies(CONST Cor: MatrixTyp; CONST Types: TypeVector;
41   VAR Freqs: FreqsType);
```

## 16. Principle component analysis

```
42 { Determine distribution of correlation coefficients by variable type }
43
44 FUNCTION Bartlett(CONST Cor: MatrixTyp; Cases: WORD): double;
45 { Bartlett's test for sphericity }
46
47 PROCEDURE ExplainedVariance(CONST EigenValues: VectorTyp;
48     VAR Acc, Variance, CumVariance: VectorTyp);
49 { calculate acceleration, explained variance and cumulative explained
    variance
50 from eigenvalues and writes them into a csv-file }
51
52 PROCEDURE WriteEigenVector(CONST EigenVectors: MatrixTyp);
53 { writes eigenvectors into a csv-file }
54
55 PROCEDURE MaximumLikelihood(CONST Data: MatrixTyp; CONST Types: TypeVector;
56     VAR ImputVector: VectorTyp);
57 { Determin average (cardinal) or most common (other) value of a column for
    imputation }
58
59 PROCEDURE RobustProduct(CONST A, B: MatrixTyp; CONST ImputVector: VectorTyp;
60     VAR C: MatrixTyp);
61 { robust matrix product, elements of A (original data) may be NaN and are
    replaced
62 by the j-th element in ImputVector. For B (Eigenvectors) this precaution
    is not
63 necessary. Only the first Max eigenvectors are used. }
64
65 PROCEDURE WriteComponentScores(CONST Scores: MatrixTyp);
66 { writes component scores into a csv-file }
67
68 PROCEDURE CalculateLoadings(CONST Data, Scores: MatrixTyp;
69     CONST Types: TypeVector; VAR Loadings: MatrixTyp);
70
71 PROCEDURE CalculateCommunalities(CONST Loadings: MatrixTyp;
72     VAR Communalities, Uniqueness,
73     VarianceAccounted: VectorTyp);
74 { Row and column sums of squared loadings }
75
76 PROCEDURE WriteLoadings(CONST Loadings: MatrixTyp;
77     CONST Communalities, Uniqueness,
78     VarianceAccounted: VectorTyp;
79     Rotated: BOOLEAN);
80 { Writes loadings into a csv-file. File name will depend on whether the
    loadings have been rotated. }
81
82
83 IMPLEMENTATION
84
```

```

85
86 VAR
87   Significance: SignificanceType;

```

The following procedure reads data from a .csv-file, the first row contains the variable names, the second the variable level (nominal, ordinal, interval and rational). The routine can be used for files following both US- and European conventions for decimal indicator (./) and separation character (;/):

```

1 PROCEDURE ReadCorrelations(VAR Cor: MatrixTyp);
2 // Read correlation matrix from CSV FILE
3
4 VAR
5   InputFile: TEXT;
6   c: CHAR;
7   s: STRING;
8   Variables, Error, i, j: WORD;
9   x: double;
10
11 BEGIN
12   Assign(InputFile, Problemname + '-NativeCorr.csv');
13   Reset(InputFile);
14   Variables := 0;
15   WHILE NOT EoLn(InputFile) DO // Read first Line WITH variable numbers,
16     count Variables
17   BEGIN // nothing needs TO be done with these values
18     INC(Variables);
19     i := ReadInt(InputFile);
20   END; { while }
21   ReadLn(InputFile); // go TO next Line
22   DEC(Variables); // Line number
23   CreateMatrix(Cor, Variables, Variables, 0.0);
24   FOR i := 1 TO Variables DO // now Read data from following lines
25     BEGIN
26       j := ReadInt(InputFile); // Line number
27       FOR j := 1 TO Variables DO
28         SetMatrixElement(Cor, i, j, ReadFloat(InputFile)); // values
29         ReadLn(InputFile);
30     END; { for i }
31   Close(InputFile);
32   Writeln(Variables: 3, ' variables read, ');
33 END;

```

Calculating the correlation matrix can be time consuming. If this has been done already in the past, the following routine reads such a matrix from a file.

Listing 16.4: Read correlation matrix from a csv-file

```

1 PROCEDURE ReadCorrelations(VAR Cor: MatrixTyp);

```

## 16. Principle component analysis

```

2 // Read correlation matrix from CSV FILE
3
4 VAR
5   InputFile: TEXT;
6   C: CHAR;
7   S: STRING;
8   Variables, Error, i, j: WORD;
9   x: double;
10
11 BEGIN
12   Assign(InputFile, Problemname + '-NativeCorr.csv');
13   Reset(InputFile);
14   Variables := 0;
15   WHILE NOT EoLn(InputFile) DO // Read first Line WITH variable numbers,
16     count Variables
17     BEGIN // nothing needs TO be done with these values
18       INC(Variables);
19       i := ReadInt(InputFile);
20     END; { while }
21   ReadLn(InputFile); // go TO next Line
22   DEC(Variables); // Line number
23   CreateMatrix(Cor, Variables, Variables, 0.0);
24   FOR i := 1 TO Variables DO // now Read data from following lines
25     BEGIN
26       j := ReadInt(InputFile); // Line number
27       FOR j := 1 TO Variables DO
28         SetMatrixElement(Cor, i, j, ReadFloat(InputFile)); // values
29       ReadLn(InputFile);
30     END; { for i }
31   Close(InputFile);
32   Writeln(Variables: 3, ' variables read, ');
33 END;

```

The following routine analyses the distribution frequency for the values in  $\mathcal{R}$  and writes them to a file.

Listing 16.5: Determine frequency distribution of correlation coefficients

```

1 PROCEDURE AnalyseFrequencies(CONST Cor: MatrixTyp; CONST Types: TypeVector;
2                               VAR Freqs: FreqsType);
3
4 CONST
5   TypeText: ARRAY [DataTypes] OF STRING[8] =
6     ('binary', 'nominal', 'ordinal', 'interval', 'rational');
7
8 VAR
9   i, j: DataTypes;
10  l, m, Variables: WORD;

```

```

11  k: INTEGER;
12  OutFile: TEXT;
13  x: double;
14
15 BEGIN
16   Assign(OutFile, ProblemName + '-Frequencies.csv');
17   Rewrite(OutFile);
18   Variables := MatrixRows(Cor);
19   FOR i := LOW(DataTypes) TO High(DataTypes) DO
20     FOR j := LOW(DataTypes) TO High(DataTypes) DO
21       FOR k := -Border TO Border DO
22         Freqs[i, j, k] := 0;
23   FOR l := 1 TO Variables DO
24     FOR m := 1 TO Variables DO
25       BEGIN
26         x := GetMatrixElement(Cor, l, m) * Border;
27         INC(Freqs[Types[l], Types[m], Round(x)]);
28       END;
29   FOR i := LOW(DataTypes) TO High(DataTypes) DO
30     BEGIN
31       Writeln(OutFile, TypeText[i]: 8, SepChar);
32       FOR j := LOW(DataTypes) TO High(DataTypes) DO
33         BEGIN
34           Write(OutFile, SepChar, TypeText[j]: 8, SepChar);
35           FOR k := -Border TO Border DO
36             Write(OutFile, Freqs[i, j, k]: 4, SepChar);
37             Writeln(OutFile);
38           END;
39         END;
40       Close(OutFile);
41     END;

```

The procedure `CalculateCorrelationMatrix` can calculate correlation coefficients appropriate for the levels of the data. In practice, it is often better to calculate PEARSON's product moment correlation  $r_p$  for binary, ordinal, interval and rational data, and to avoid nominal data at all. This minimises rank deficiency of the correlation matrix. If that is all that is desired, the `case`-statement can be simplified accordingly.

Listing 16.6: Calculate the correlation matrix

```

1 PROCEDURE CalculateCorrelationMatrix(CONST Data: MatrixTyp; CONST Types:
2   TypeVector; VAR Cor: MatrixTyp);
3
4 VAR
5   i, j, Variables: WORD;
6   IVector, JVector: VectorTyp;
7   Rxy: double;
8   Cont: ContTable;

```

## 16. Principle component analysis

```
9
10 BEGIN
11   Variables := MatrixColumns(Data);
12   CreateIdentityMatrix(Cor, Variables);
13   GetColumn(Data, i, IVector);
14   for i := 1 to Variables do
15     begin
16       SetMatrixElement(Cor, i, i, 1.0);      // diagonal
17       GetColumn(Data, i, IVector);
18       FOR j := Succ(i) TO Variables DO
19         BEGIN
20           GetColumn(Data, j, JVector);
21           CASE Types[i] OF
22             nominal: CASE Types[j] OF
23               nominal: BEGIN
24                 Contingency(IVector, JVector, Cont);
25                 Rxy := lambda(Cont);
26                 Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
27                 DestroyContingency(Cont);
28               END;
29               ordinal: BEGIN
30                 Contingency(IVector, JVector, Cont);
31                 Rxy := theta(Cont);
32                 Rxy := Sqrt(Rxy);
33                 // asymmetric: only positive values!
34                 DestroyContingency(Cont);
35               END;
36               binary: BEGIN
37                 Contingency(IVector, JVector, Cont);
38                 Rxy := lambda(Cont);
39                 Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
40                 DestroyContingency(Cont);
41               END;
42               interval,
43               rational: BEGIN
44                 Rxy := eta_sqr(IVector, JVector,
45                               Significance);
46                 Rxy := Sqrt(Rxy);
47                 // asymmetric: only positive values!
48               END;
49             ordinal: CASE Types[j] OF
50               nominal: BEGIN
51                 Contingency(JVector, IVector, Cont);
52                 Rxy := theta(Cont);
53                 Rxy := Sqrt(Rxy);
```

```

54                                         DestroyContingency(Cont);
55
56         ordinal: BEGIN
57             Rxy := OrdinalCorrelations(IVector, JVector,
58                                         'E', Significance);
59             Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
60         END;
61         binary: BEGIN
62             Rxy := OrdinalCorrelations(IVector, JVector,
63                                         'E', Significance);
64             Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
65         END;
66         interval,
67         rational: Rxy := SpearmanRankCorrelation(IVector,
68                                         JVector, Significance);
69     END;
70     binary: CASE Types[j] OF
71         nominal: BEGIN
72             Contingency(IVector, JVector, Cont);
73             Rxy := lambda(Cont);
74             Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
75             DestroyContingency(Cont);
76         END;
77         ordinal: BEGIN
78             Rxy := OrdinalCorrelations(IVector,
79                                         JVector, 'E', Significance);
80             Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
81         END;
82         binary: BEGIN
83             Contingency(IVector, JVector, Cont);
84             Rxy := lambda(Cont);
85             Rxy := sign(Rxy) * Sqrt(Abs(Rxy));
86             DestroyContingency(Cont);
87         END;
88         interval,
89         rational: Rxy := PointBiserialCorrelation(IVector,
90                                         JVector, Significance);
91     END;
92     interval,
93     rational: CASE Types[j] OF
94         nominal: BEGIN
95             Rxy := eta_sqr(JVector, IVector,
96                             Significance);
97             Rxy := Sqrt(Rxy);
98         END;
99     END;
100 
```

## 16. Principle component analysis

```

94           END;
95           ordinal: Rxy := SpearmanRankCorrelation(IVector,
96                                         JVector, Significance);
96           binary: Rxy := PointBiserialCorrelation(JVector,
97                                         IVector, Significance);
97           interval,
98           rational: Rxy :=
99               PearsonProductMomentCorrelation(IVector, JVector,
100                                         Significance);
100          END;
100         END; { case Types[i] }
101         SetMatrixElement(Cor, i, j, Rxy); // upper half
102         SetMatrixElement(Cor, j, i, Rxy); // lower half
103         Writeln(i:4, ' ', j:4, ' ', Rxy:3:3);
104         DestroyVector(JVector);
105     END; { for j }
106     DestroyVector(IVector);
107   END; { for i }
108   Write('Correlation matrix calculated, ');
109 END;

```

Listing 16.7: Write the correlation matrix to a csv-file

```

1 PROCEDURE WriteCorrelations(CONST Cor: MatrixTyp; Shrunk: BOOLEAN);
2
3 VAR
4   i, j: WORD;
5   OutFile: TEXT;
6
7 BEGIN
8   IF Shrunk
9     THEN Assign(OutFile, ProblemName + '-ShrunkCorr.csv')
10    ELSE Assign(OutFile, Problemname + '-NativeCorr.csv');
11   Rewrite(OutFile);
12   Write(OutFile, SepChar);
13   FOR j := 1 TO Cor^.Columns DO
14     Write(OutFile, j: 4, SepChar); // LABEL columns
15   Writeln(OutFile);
16   FOR i := 1 TO Cor^.Rows DO // the matrix itself
17     BEGIN
18       Write(OutFile, i: 4, SepChar); // LABEL row
19       FOR j := 1 TO Cor^.Columns DO
20         Write(OutFile, GetMatrixElement(Cor, i, j):7:4, ';');
21       Writeln(OutFile);
22     END; { for i }
23   Close(OutFile);
24   Writeln('Correlations written to file, ');

```

```
25 END; { WriteCorrelations}
```

BARTLETT's test for sphericity calculates the probability for the null-hypothesis  $H_0 : \mathcal{R} = \mathcal{I}$ . In that case, the data would be uncorrelated (except for experimental error), and a PCA or factor analysis would be pointless.

Listing 16.8: Perform Bartlett's sphericity-test

```
1 FUNCTION Bartlett(CONST Cor: MatrixTyp; Cases: WORD): double;
2
3 VAR
4   f, Variables: WORD;
5   chi2, Det, P0: extended;
6
7 BEGIN
8   Variables := MatrixRows(Cor);
9   Det := Determinante(Cor);
10  Write('Determinante = ', det: 10, ' ');
11  IF (Det < Zero)                      // singular matrix
12    THEN
13      BEGIN
14        Result := NaN;
15        Writeln('Bartlet = NaN');
16        EXIT;
17      END;
18  chi2 := -(Pred(Cases) - (2 * Variables + 5) / 6) * log(Det, 10);
19  f := Round(Variables * Pred(Variables) / 2);
20  Write('chi2 := ', chi2: 5: 3, ' ', 'f = ', f: 5, ' ');
21  P0 := IntegralChi(chi2, f);
22  Writeln('P0 = ', P0: 1: 4);
23  Result := P0;
24 END;
```

Listing 16.9: Write eigenvalues and statistics to file

```
1 PROCEDURE ExplainedVariance(CONST EigenValues: VectorTyp;
2   VAR Acc, Variance, CumVariance: VectorTyp);
3
4 VAR
5   i, Variables: WORD;
6   x, Sum, Cummulative: double;
7   OutFile: TEXT;
8
9 BEGIN
10  Variables := VectorLength(EigenValues);
11  CreateVector(Variance, Variables, 0.0);
12  CreateVector(CumVariance, Variables, 0.0);
13  CreateVector(Acc, Variables, 0.0);
14  Sum := 0;
```

## 16. Principle component analysis

```

15   i := 1;
16   FOR i := 1 TO Variables DO
17     Sum := Sum + GetVectorElement(EigenValues, i);
18   Cummulative := 0;
19   FOR i := 2 TO Pred(Variables) DO // second derivative OF the scree-curve
20     by finite differences
21     SetVectorElement(Acc, i, GetVectorElement(EigenValues, Succ(i)) -
22       2*GetVectorElement(EigenValues, i) + GetVectorElement(EigenValues,
23       Pred(i)));
24     SetVectorElement(Acc, 1, NaN);
25     SetVectorElement(Acc, Variables, NaN);
26     Assign(OutFile, ProblemName + '-EigenValues.csv');
27     Rewrite(OutFile);
28     FOR i := 1 TO Variables DO
29       BEGIN
30         x := GetVectorElement(EigenValues, i) / Sum;
31         Cummulative := Cummulative + x;
32         SetVectorElement(Variance, i, x);
33         SetVectorElement(CumVariance, i, Cummulative);
34         Write(OutFile, i: 3, SepChar, GetVectorElement(EigenValues, i):3:3,
35               SepChar,
36               x:3:3, SepChar, Cummulative:3:3);
37         Writeln(OutFile, SepChar, GetVectorElement(Acc, i):3:3);
38       END;
39     Close(OutFile);
40   END;

```

Listing 16.10: Write eigenvectors to file

```

1 PROCEDURE WriteEigenVector(CONST EigenVectors: MatrixTyp);
2
3 VAR
4   i, j: WORD;
5   OutFile: TEXT;
6
7 BEGIN
8   Assign(OutFile, ProblemName + '-EigenVectors.csv');
9   Rewrite(OutFile);
10  FOR i := 1 TO MatrixRows(EigenVectors) DO
11    BEGIN
12      Write(OutFile, i: 3, SepChar);
13      FOR j := 1 TO SigVecs DO
14        Write(OutFile, GetMatrixElement(EigenVectors, i, j): 3: 3, SepChar);
15      Writeln(OutFile);
16    END;
17  Close(OutFile);
18 END;

```

For the calculation of the correlation matrix we ignored missing values. If this is done for the calculation of the scores, the product of the data with the projection matrix  $\mathcal{F}$ , then each missing value would set both the corresponding row and column of  $\mathcal{C}$  to `NaN`. That way, even relatively few missing data may result in an all-`NaN` score matrix  $\mathcal{C}$ . The calculation of the score matrix therefore requires imputation (see chapter 15 for a discussion of missing values). In this unit, for interval and rational data we use the arithmetic mean of each data vector as maximum likelihood estimator for the missing datum, for binary, nominal and ordinal data, we use the most common value of each vector (see section 15.2.2 on page 510 on methods to handle missing data).

Listing 16.11: Robust calculation of scores

```

1 PROCEDURE MaximumLikelihood(CONST Data: MatrixTyp; CONST Types: TypeVector;
2   VAR ImputVector: VectorTyp);
3
4 VAR
5   i, j, n, p, s, iMax: WORD;
6   JVector: VectorTyp;
7   x: double;
8   Numbers: ARRAY [1..MaxSteps] OF WORD;
9
10 BEGIN
11   n := MatrixRows(Data);
12   p := MatrixColumns(Data);
13   CreateVector(ImputVector, p, 0.0);
14   FOR j := 1 TO p DO
15     BEGIN
16       GetColumn(Data, p, JVector);
17       CASE Types[j] OF
18         binary,
19         nominal,
20         ordinal: BEGIN
21           FOR i := 1 TO MaxSteps DO Numbers[i] := 0;
22           FOR i := 1 TO n DO
23             BEGIN
24               x := GetVectorElement(JVector, i);
25               IF IsNaN(x)
26                 THEN
27                   ELSE INC(Numbers[Round(x)]);
28             END;
29             x := 0;
30             iMax := 0;
31             FOR i := 1 TO MaxSteps DO
32               IF (Numbers[i] > x)
33                 THEN
34                   BEGIN
35                     x := Numbers[i];

```

## 16. Principle component analysis

```
36                     iMax := i;
37                     END;
38                     SetVectorElement(ImputVector, j, iMax); // most common
39                     element
40                     END;
41                     interval,
42                     rational: BEGIN
43                         x := NeumaierSum(JVector)/ActualElements(JVector);
44                         //arithmetic mean
45                         SetVectorElement(ImputVector, j, x);
46                     END;
47                     END; // case
48                     DestroyVector(JVector);
49                     END;
50 PROCEDURE RobustProduct(CONST A, B: MatrixTyp; CONST ImputVector: VectorTyp;
51                         VAR C: MatrixTyp);
52
53 VAR
54     i, j, k: WORD;
55     Sum: double;
56
57 BEGIN
58     IF MatrixColumns(A) <> MatrixRows(B)
59     THEN
60         BEGIN
61             Write(' Matrix multiplication: A.Columns <> B.Rows');
62             ReadLn;
63             EXIT;
64         END;
65     IF MatrixColumns(B) < SigVecs
66     THEN
67         BEGIN
68             Write(' Matrix multiplication: Number of Eigenvectors < SigVecs ');
69             ReadLn;
70             EXIT;
71         END;
72     CreateMatrix(C, MatrixRows(A), SigVecs, 0);
73     FOR i := 1 TO MatrixRows(A) DO
74         FOR j := 1 TO SigVecs DO
75             BEGIN
76                 Sum := 0;
77                 FOR k := 1 TO MatrixColumns(A) DO
78                     IF IsNaN(GetMatrixElement(A, i, k))
79                     THEN Sum := Sum + GetVectorElement(ImputVector, k) *
```

```

80                     GetMatrixElement(B, k, j)
81             // replace NaN WITH max likelihood estimator
82             ELSE Sum := Sum + GetMatrixElement(A, i, k) *
83                 GetMatrixElement(B, k, j);
84             SetMatrixElement(C, i, j, Sum);
85         END;
86     END;
87
88 PROCEDURE WriteComponentScores(CONST Scores: MatrixTyp);
89
90 VAR
91     i, j: WORD;
92     OutFile: TEXT;
93
94 BEGIN
95     Assign(OutFile, ProblemName + '-Scores.csv');
96     Rewrite(OutFile);
97     FOR i := 1 TO MatrixRows(Scores) DO
98         BEGIN
99             FOR j := 1 TO SigVecs DO
100                 Write(OutFile, GetMatrixElement(Scores, i, j): 3: 3, SepChar);
101                 Writeln(OutFile);
102             END;
103             Close(OutFile);
104         END;

```

The loadings  $\mathcal{L}$  are the correlations between each variable and each column of the score matrix. Just as with the calculation of the correlation matrix  $\mathcal{R}$ , it has been found that assuming all binary, ordinal, interval and rational variables to be rational, and not to use nominal data, gives better results than the mixed correlation coefficients used in the routine below. If this is all that is desired, the `case`-statement below can be simplified.

Listing 16.12: Calculate and write loadings to csv-file

```

1 PROCEDURE CalculateLoadings(CONST Data, Scores: MatrixTyp;
2     CONST Types: TypeVector; VAR Loadings: MatrixTyp);
3
4 VAR
5     i, j, p : WORD;
6     Rxy: double;
7     DataVector, ScoreVector: VectorTyp;
8     Significance: SignificanceType;
9
10 BEGIN
11     p := MatrixColumns(Data);
12     CreateMatrix(Loadings, p, SigVecs, 0.0);
13     FOR i := 1 TO p DO                      // over all variables
14         BEGIN

```

## 16. Principle component analysis

```

15      GetColumn(Data, i, DataVector);
16      FOR j := 1 TO SigVecs DO           // over all Scores
17          BEGIN
18              GetColumn(Scores, j, ScoreVector);    // JVector IS always
19                  rational
20                  CASE Types[i] OF
21                      binary:   Rxy := PointBiserialCorrelation(DataVector,
22                                         ScoreVector, Significance);
23                      nominal: Rxy := Sqr(eta_sqr(DataVector, ScoreVector,
24                                         Significance));
25                          // by definition always positive
26                      ordinal: Rxy := SpearmanRankCorrelation(DataVector,
27                                         ScoreVector, Significance);
28                          interval,
29                          rational: Rxy := PearsonProductMomentCorrelation(DataVector,
30                                         ScoreVector, Significance);
31          END;
32          SetMatrixElement(Loadings, i, j, Rxy);
33          DestroyVector(ScoreVector);
34      END;
35  END;

```

The communality is the column vector of the rows sum of squared loadings:  $h_j = \sum_{k=1}^q l_{jk}^2$ , this gives the proportion of variance in a variable that is explained by the factors. Sometimes the uniqueness  $u_j = 1 - h_j$  is used instead, this is the specific variance of a variable, the variance it does not have in common with other variables. HOFFMAN's index of complexity [17] for each item is  $\frac{(\sum_{k=1}^q l_{jk}^2)^2}{\sum_{j=1}^q l_{jk}^4}$ .

If the sum of squared column elements is calculated, we get the sum of squared loadings. Their sum is equal to the sum of the first  $q$  eigenvalues. Dividing each element of this vector by the total, we get the variance accounted for, the part of the variance in the data that is explained by each factor.

Listing 16.13: Calculate statistics of loading-matrix

```

1  PROCEDURE CalculateCommunalities(CONST Loadings: MatrixTyp;
2      VAR Communalities, Uniqueness, VarianceAccounted: VectorTyp);
3
4  VAR
5      i, j, Rows, Columns: WORD;
6      Sum1, Sum2, x, y: double;
7
8  BEGIN
9      Rows := MatrixRows(Loadings);
10     Columns := MatrixColumns(Loadings);

```

```

11 CreateVector(Communalities, Rows, 0);
12 FOR i := 1 TO Rows DO
13   BEGIN
14     Sum1 := 0;
15     FOR j := 1 TO SigVecs DO
16       Sum1 := Sum1 + Sqr(GetMatrixElement(Loadings, i, j));
17       // row sum OF squared loadings
18     SetVectorElement(Communalities, i, Sum1);
19   END;
20 CreateVector(VarianceAccounted, Columns + 2, 0.0);
21 FOR j := 1 TO Columns DO
22   BEGIN
23     Sum1 := 0;
24     FOR i := 1 TO Rows DO
25       Sum1 := Sum1 + Sqr(GetMatrixElement(Loadings, i, j));
26       // column sum OF squared loadings
27     SetVectorElement(VarianceAccounted, j, Sum1);
28   END;
29 CreateVector(Uniqueness, Rows, 0);
30 Sum1 := 0;
31 Sum2 := 0;
32 FOR i := 1 TO Rows DO
33   BEGIN
34     x := GetVectorElement(Communalities, i);
35     y := 1 - x;
36     SetVectorElement(Uniqueness, i, y);
37     Sum1 := Sum1 + x;                      // explained
38     Sum2 := Sum2 + y;                      // unexplained
39   END;
40 SetVectorElement(VarianceAccounted, Columns + 1, Sum1);
41 SetVectorElement(VarianceAccounted, Columns + 2, Sum2);
42 END;
43
44
45 PROCEDURE WriteLoadings(CONST Loadings: MatrixTyp; CONST Communalities,
46                           Uniqueness,
47                           VarianceAccounted: VectorTyp; Rotated: BOOLEAN);
48
49 VAR
50   n, i, j: WORD;
51   OutFile: TEXT;
52   Variance: double;
53
54 BEGIN
55   IF Rotated
      THEN Assign(OutFile, ProblemName + '-RotLoad.csv')

```

## 16. Principle component analysis

```

56     ELSE Assign(OutFile, ProblemName + '-Loadings.csv');
57     Rewrite(OutFile);
58     n := MatrixRows(Loadings);
59     FOR i := 1 TO n DO
60         BEGIN
61             FOR j := 1 TO SigVecs DO
62                 Write(OutFile, GetMatrixElement(Loadings, i, j):3:3, SepChar);
63                 Writeln(OutFile, SepChar, GetVectorElement(Communalities, i):3:3,
64                         SepChar,
65                         GetVectorElement(Uniqueness, i):3:3, SepChar);
66             END;
67             Writeln(OutFile);
68             FOR j := 1 TO SigVecs DO
69                 Write(OutFile, GetVectorElement(VarianceAccounted, j):3:3, SepChar);
70                 Writeln(OutFile, GetVectorElement(VarianceAccounted, SigVecs + 1):3:3,
71                         SepChar,
72                         GetVectorElement(VarianceAccounted, SigVecs + 2):3:3, SepChar);
73                 Variance := GetVectorElement(VarianceAccounted, SigVecs + 1) +
74                     GetVectorElement(VarianceAccounted, SigVecs + 2);
75                 FOR j := 1 TO SigVecs DO
76                     Write(OutFile, GetVectorElement(VarianceAccounted, j) / Variance:3:3,
77                           SepChar);
78                     Writeln(OutFile, GetVectorElement(VarianceAccounted, SigVecs + 1) /
79                             Variance:3:3, SepChar,
80                             GetVectorElement(VarianceAccounted, SigVecs + 2) / Variance:3:3,
81                             SepChar);
82             Close(OutFile);
83         END;
84
85
86     END. // PCA

```

## 16.7. Real world data in PCA

### 16.7.1. If the correlation matrix is not positive definite: Shrinkage

It is possible to calculate the correlation matrix  $\mathcal{R}$  robustly towards missing values (see fig. 16.6). However, the resulting matrix will no longer be positive definite and large eigenvalues will be overestimated, while small eigenvalues are underestimated. To be suitable for factor analysis or PCA,  $\mathcal{R}$  must be

**positive definite** which means that  $\mathbf{x}^T \mathcal{R} \mathbf{x} > 0 \forall \mathbf{x}$  with at least one element  $\neq 0$ . A positive definite matrix has a determinant  $\|\mathcal{R}\| > 0$ . In our case,  $\|\mathcal{R}\| = 1 \times 10^{-56} \approx 0$ . For a positive definite matrix all leading principal minors (the  $i$ -th **leading principal minor** of an  $n \times n$  matrix is the determinant of the submatrix containing

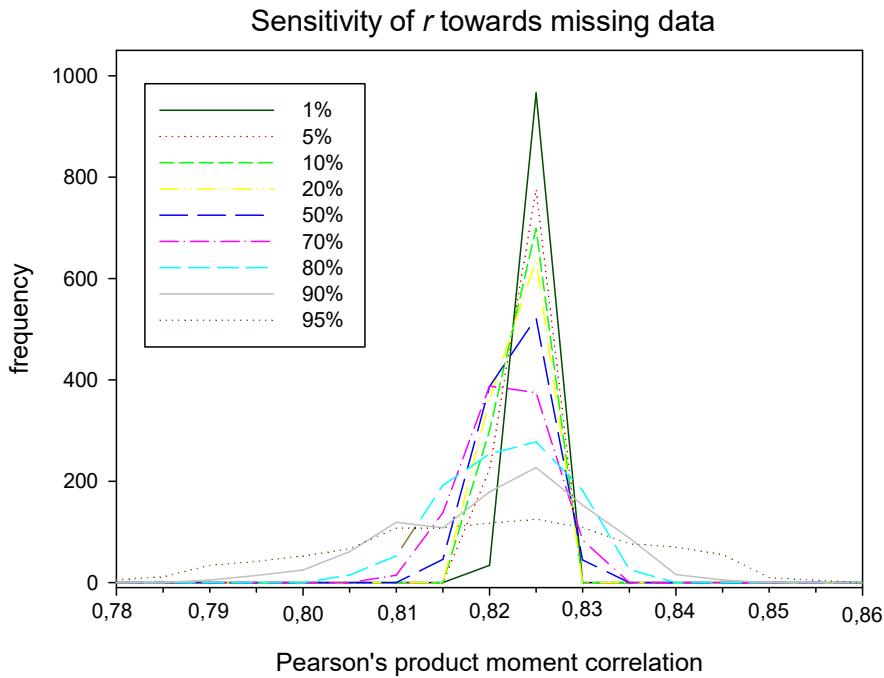


Figure 16.6.: Simulation of the effect on missing data on PEARSON's product moment correlation. Data vectors  $\mathbf{x}_i = i, \mathbf{y} = 1 + 2\mathbf{x} + \text{NormalRandom}(0, 4000)$  with 10 000 elements were prepared, the correlation coefficient was 0.825. From these data, a variable proportion of elements was randomly set to NaN, and the correlation coefficients calculated. For up to 20 % missing data the resulting sample  $r$  are within  $\pm 0.01$  of the population value.

## 16. Principle component analysis

its first  $i$  rows and its first  $i$  columns) are also positive (SYLVESTER's criterion). Computational errors may result in small positive or negative values instead of zero for semi-definite matrices.

**invertible** No data column  $i$  may be a constant, as this would result in  $\mathcal{R}_{\cdot i} = \mathcal{R}_{i \cdot} = 0$ . No data column  $i$  can be replicated by a linear combination of any of the remaining  $p - 1$  columns, otherwise the  $i$ -th row and column of  $\mathcal{R}$  will be linear combinations of other columns or rows, respectively and therefore  $\|\mathcal{R}\| = 0$ .

### Shrinkage of $\mathcal{S}$

The LEDOIT-WOLF-procedure [26–28] tries to turn a semi-definite matrix into a definite by “shrinkage”. In effect, all eigenvalues are moved towards their grand mean, while maintaining the eigenvectors.

**Weighted average of  $\mathcal{S}$  with another matrix** For example, a  $\mathcal{B}_{p \times p}$  diagonal matrix with  $\mathcal{B}_{ii} = \mathcal{S}_{ii}$  the variances in the diagonal (that is,  $\mathcal{B} = \text{diag}(\mathcal{S})$ ) is always positive definite, because  $\mathbf{x}^T \mathcal{B} \mathbf{x} = \sum_{j=1}^p \mathbf{x}_j^2 \mathcal{B}_{jj} > 0$  if there is at least one element  $\mathbf{x}_j \neq 0$ . Then the weighted average of  $\mathcal{B}$  and  $\mathcal{S}$  is defined as  $\widehat{\mathcal{S}} = (1 - \omega)\mathcal{S} + \omega\mathcal{B}$  with a weight  $0 < \omega < 1$ , and is always positive definite because  $\mathbf{x}^T \widehat{\mathcal{S}} \mathbf{x} = (1 - \omega)\mathbf{x}^T \mathcal{S} \mathbf{x} + \omega \mathbf{x}^T \mathcal{B} \mathbf{x} > 0$ . If  $\omega = 0$ , then  $\widehat{\mathcal{S}} = \mathcal{S}$ , and if  $\omega = 1$ , then  $\widehat{\mathcal{S}} = \mathcal{B}$ .

**Optimal  $\omega$**  The covariances  $s_{ij}, i \neq j$  in  $\mathcal{S}$  are sample estimators of the population covariances  $\sigma_{ij}$ . Then  $[(1 - \omega)s_{ij} - \sigma_{ij}]^2$  can be viewed as loss, and we would like to find the  $\omega$  that minimises  $\sum_{i=1}^p \sum_{j=i+1}^p [(1 - \omega)s_{ij} - \sigma_{ij}]^2$  for all elements of the upper triangle of  $\mathcal{S}$  (of course, the lower triangle would give the same result because of symmetry) [27]. Then

$$\omega = \frac{\sum_{i=1}^p \sum_{j=i+1}^p \text{var}(s_{ij})}{\sum_{i=1}^p \sum_{j=i+1}^p [\text{var}(s_{ij}) + \sigma_{ij}^2]} \quad (16.17)$$

We note that because the nominator is larger than the denominator and both are positive, the condition  $0 < \omega < 1$  is met by necessity. To get  $\text{var}(s_{ij})$  we introduce for each pair of variables  $i, j = 1..p$  a random variable  $\mathbf{g}_{ij}$ , which is the product of the centered data columns  $\mathbf{x}_{\cdot i}$  and  $\mathbf{x}_{\cdot j}$ . Then  $\bar{\mathbf{g}}_{i,j} = 1/n \sum_{k=1}^n [(\mathbf{x}_{k,i} - \bar{\mathbf{x}}_{\cdot i})(\mathbf{x}_{k,j} - \bar{\mathbf{x}}_{\cdot j})]$  and  $\text{var}(s_{ij}) = \frac{n}{(n-1)^3} \sum_{k=1}^n (\mathbf{g}_{kij} - \bar{\mathbf{g}}_{i,j})^2$  and  $E(s_{ij}) = \sigma_{ij}$  and hence  $\sigma_{ij}^2 = s_{ij}^2$   $\omega$  can be calculated.

**Non-linear shrinkage** In the simplest case, shrinkage is done linearly by a certain factor  $\omega$  as discussed. Nonlinear shrinkage has been shown to be at least as effective, and most of the time more, than linear [29]. R package `nlshrink` provides this method. The method is described for  $\mathcal{S}$ , it is unclear how this would extend to  $\mathcal{R}$ .

### Shrinking a correlation matrix $\mathcal{R}$

Under some simplifying assumptions, the basic idea is the same as with the variance-covariance matrix, but shrinkage is achieved by calculating a weighted average between  $\mathcal{R}$  and the identity matrix  $\mathcal{I}$  [26, 27]. Then equation 16.17 becomes

$$\omega = \frac{\sum_{i=1}^p \sum_{j=i+1}^p \text{var}(r_{ij})}{\sum_{i=1}^p \sum_{j=i+1}^p [\text{var}(r_{ij}) + \rho_{ij}^2]} \quad (16.18)$$

where  $\rho_{ij}$  is the population correlation coefficient of variables  $ij$ .  $\text{var}(r_{ij}) = \frac{\text{var}(s_{ij})}{s_{ii}s_{jj}}$  and  $\rho_{ij}^2 = \frac{s_{ij}^2}{s_{ii}s_{jj}}$ . The R-package **corpcor** provides shrinkage for  $\mathcal{S}$  and  $\mathcal{R}$ .

Alternative is to first calculate the average correlation of each variable with all others:

$$\bar{r}_{i \cdot} = \frac{1}{p-1} \sum_{j=1}^p r_{ij} \quad \forall j \neq i \quad (16.19)$$

Then the correlation coefficient  $\hat{r}_{ij} = (\bar{r}_{i \cdot} + \bar{r}_{j \cdot})/2$  [30]. This matrix is positive definite and performs very similar to that described by [27] with optimal  $\omega$  (see fig. 16.7 and 16.8).

**Recalculation of  $\mathcal{R}$  and  $\mathcal{S}$  from results of an initial eigenanalysis** If the correlation matrix  $\mathcal{R}$  is indefinite and returns negative eigenvalues, these can be seen as experimental error (they are usually small). Hence, they are set to zero and a definite correlation matrix  $\widehat{\mathcal{R}}$  is produced from the corrected eigenvalue matrix  $\text{diag}(\hat{\lambda}) = \hat{\Lambda}$  and the eigenvectors  $\mathcal{E}$  by

$$\widehat{\mathcal{R}} = \mathcal{E} \hat{\Lambda} \mathcal{E}^{-1} \quad (16.20)$$

Then the final eigenanalysis is performed on  $\widehat{\mathcal{R}}$  [31].

It was found, however, that the result of this procedure are 0.5 % correlation coefficients that are outside the range  $[-1..+1]$ , and many diagonal elements  $\neq +1$ . If this is corrected, negative eigenvalues return, but fewer than with the original correlation matrix. Hence this procedure can be repeated iteratively, until the number of illegal entries in the correlation matrix becomes zero.

Listing 16.14: Correction of non-definite correlation matrices

```

1 UNIT PerformShrinkage;
2
3 INTERFACE
4
5 USES Math, MathFunc, Vector, Matrix, EigenValues;
6
7 PROCEDURE ErzeugeLambda(CONST Eigenvalues: VectorTyp; VAR Lambda:
8   MatrixTyp);
  { Turn vector of eigenvalues into a diagonal matrix }
```

## 16. Principle component analysis

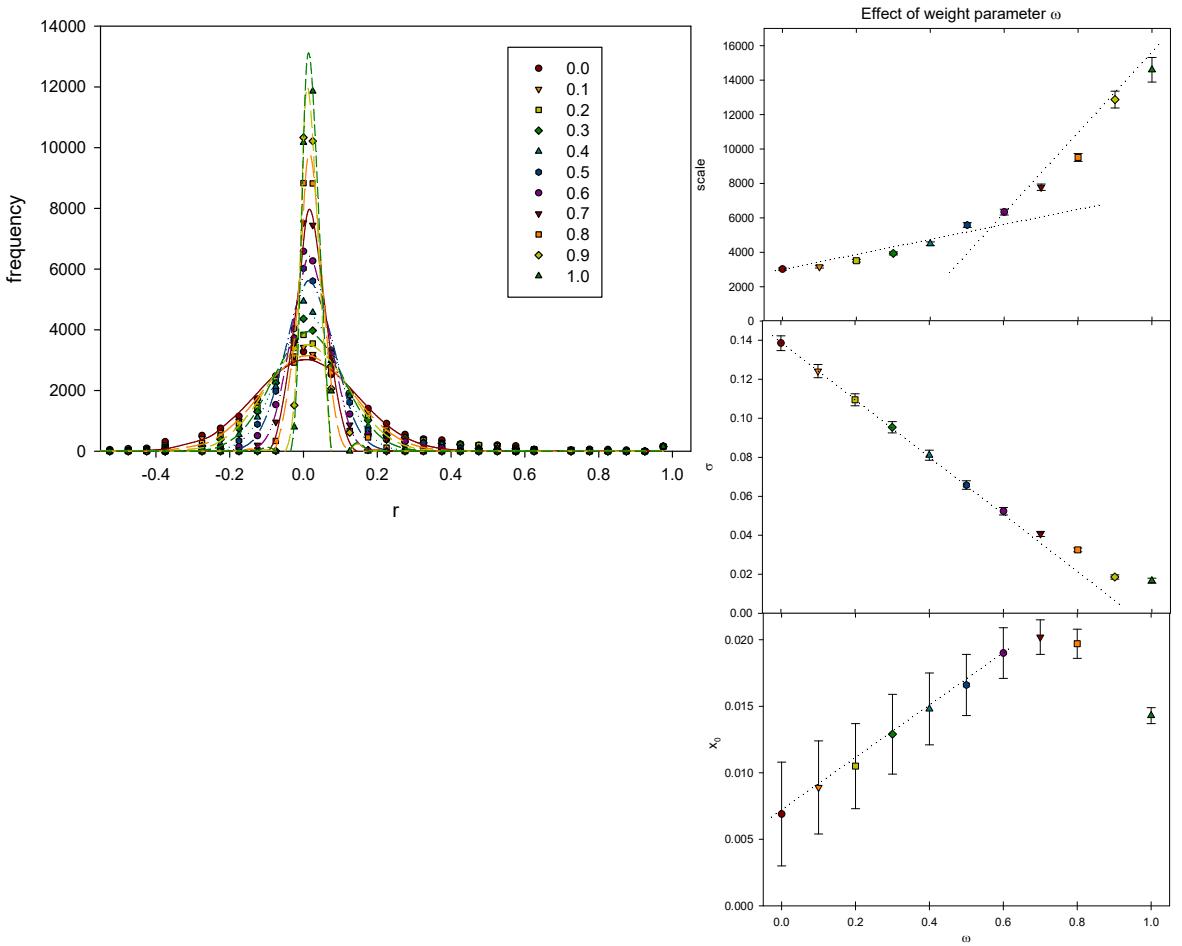


Figure 16.7.: Effect of shrinkage (using  $\bar{\mathcal{R}}$ ) on a correlation matrix that is not positive semi-definite. With increasing  $\omega$  the correlation coefficients move towards the common mean, the distribution becomes narrow and steep. An  $\omega = 0.5\dots 0.6$  is optimal with this matrix, further increases of  $\omega$  have no beneficial effect.

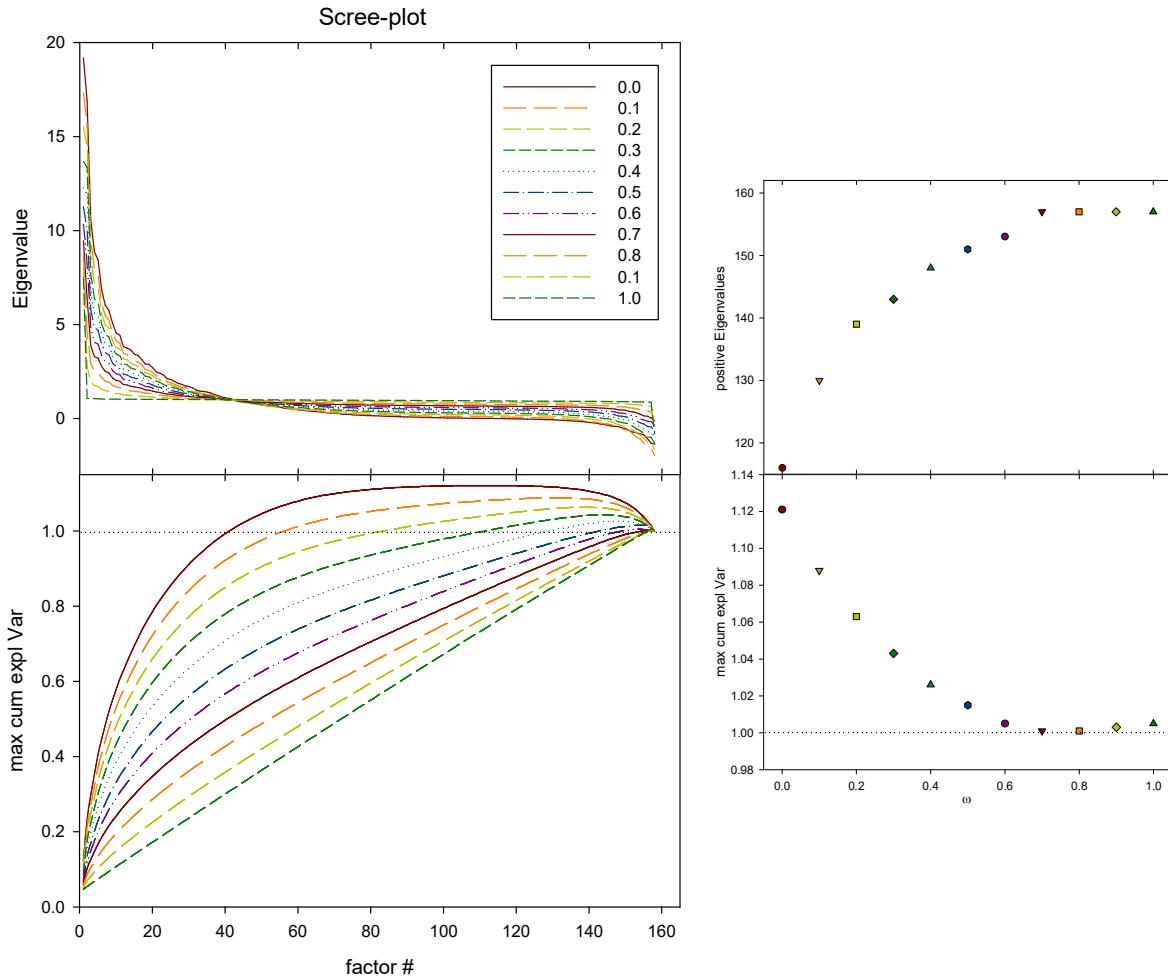


Figure 16.8.: Effect of shrinkage (using  $\bar{\mathcal{R}}$ ) of a correlation matrix that is not positive semi-definite. With increasing  $\omega$  the number of positive eigenvalues increases, the maximum cumulative explained variance decreases. An  $\omega = 0.5 \dots 0.6$  is optimal with this matrix, further increases of  $\omega$  have no beneficial effect. In the scree-plot the lines of different  $\omega$  intersect in a common point, this point is to the right of the number of valid components.

## 16. Principle component analysis

```
9
10 PROCEDURE Average(VAR R: MatrixTyp; omega: double);
11 { Shrink the correlation matrix R by calculating the weighted average
12   between R
13   and the row means of R for each i and j. omega (0..1) is the relative
14   weight
15   of of the average, the relative weight of R is (1-omega) }
16
17 PROCEDURE Shrinkage(VAR R: MatrixTyp; omega: double);
18 { Shrink the correlation matrix R by calculating the weighted average
19   between R
20   and the identity matrix. omega (0..1) is the relative weight of I, the
21   relative
22   weight of R is (1-omega) }
23
24 PROCEDURE NormaliseCorrelations(VAR R: MatrixTyp);
25 { make an indefinite correlation matrix positive definite for eigenanalysis,
26   by setting all eigenvalues < 0 to zero and re-calculating R = E Lambda
27   E^-{1}. }
28
29 IMPLEMENTATION
30
31 PROCEDURE ErzeugeLambda(CONST Eigenvalues: VectorTyp; VAR Lambda:
32   MatrixTyp);
33
34 VAR
35   p, i, j: WORD;
36
37 BEGIN
38   p := VectorLength(Eigenvalues);
39   CreateIdentityMatrix(Lambda, p);
40   FOR j := 1 TO p DO
41     SetMatrixElement(Lambda, j, j, GetVectorElement(Eigenvalues, j));
42   END;
43
44
45 PROCEDURE Average(VAR R: MatrixTyp; omega: double);
46
47 VAR
48   p, i, j: WORD;
49   Averages: VectorTyp;
50   Sum, w: double;
```

```

49   BEGIN
50     Writeln('Shrinkage: omega not in [0..1]');
51     HALT;
52   END;
53   p := MatrixRows(R);
54   CreateVector(Averages, p, 0);           // calculate row-averages OF
55   R
56   FOR i := 1 TO p DO
57     BEGIN
58       Sum := 0;
59       FOR j := 1 TO p DO
60         IF (i = j)
61           THEN // ignore correlation WITH self
62           ELSE Sum := Sum + GetMatrixElement(R, i, j);
63       SetVectorElement(Averages, i, Sum / Pred(p));
64     END;
65   w := 1 - omega;
66   omega := omega / 2;                   // weight FOR each r_i
67   AND r_j
68   FOR i := 1 TO p DO                  // calculate NEW elements
69     OF R
70     FOR j := Succ(i) TO p DO
71       BEGIN
72         Sum := (w * GetMatrixElement(R, i, j) + omega *
73             GetVectorElement(Averages, i) + omega *
74             GetVectorElement(Averages, j));
75         SetMatrixElement(R, i, j, Sum);
76         SetMatrixElement(R, j, i, Sum);
77       END;
78     DestroyVector(Averages);
79   END;
80
81 PROCEDURE Shrinkage(VAR R: MatrixTyp; omega: double);
82
83 VAR
84   p, i, j: WORD;
85   Sum, w: double;
86
87 BEGIN
88   IF (omega < 0) OR (omega > 1)
89   THEN
90     BEGIN
91       Writeln('Shrinkage: omega not in [0..1]');
92       HALT;
93     END;

```

## 16. Principle component analysis

```

91   w := 1 - omega;
92   p := MatrixRows(R);
93   FOR i := 1 TO p DO
94     OF R
95     FOR j := Succ(i) TO p DO
96       BEGIN
97         Sum := (W * GetMatrixElement(R, i, j)); // non-diagonal elements OF
98         I are 0
99         SetMatrixElement(R, i, j, Sum);           // AND can be ignored
100        SetMatrixElement(R, j, i, Sum);
101      END;
102
103 PROCEDURE NormaliseCorrelations(VAR R: MatrixTyp);
104
105 VAR
106   Eigenvalues: VectorTyp;
107   Eigenvectors, Hilfs, Lambda, RecipEV: MatrixTyp;
108   i, iter, n, NoNegs, j: WORD;
109   Rneu: double;
110
111 BEGIN
112   n := MatrixRows(R);
113   Writeln('Initial calculation of eigenvalues');
114   iter := 1;
115   REPEAT
116     INC(iter);
117     i := Jacobi(R, Eigenvalues, Eigenvectors, j); // eigenanalysis
118     Write(iter: 2, ': ', j: 3, ' iterations, Result = ', i: 1);
119     CASE i OF
120       0 : Write(' ok          ');
121       5 : Write(' no convergence ');
122     ELSE Write(' error          ');
123     END;
124     NoNegs := 0;
125     FOR i := 1 TO n DO
126       BEGIN
127         IF (GetVectorElement(Eigenvalues, i) < 0)
128           THEN
129             BEGIN
130               SetVectorElement(Eigenvalues, i, 0);
131               INC(NoNegs);
132             END;
133         END;
134       Writeln(NoNegs: 3);

```

```

135   DestroyMatrix(R);
136   ErzeugeLambda(Eigenvalues, Lambda);
137   CopyMatrix(EigenVectors, RecipEV);
138   InverseMatrix(RecipEV);
139   MatrixInnerProduct(EigenVectors, Lambda, Hilfs);      // calculate R = E
140   Lambda E^{-1}
141   MatrixInnerProduct(Hilfs, RecipEV, R);
142   DestroyMatrix(EigenVectors);
143   DestroyMatrix(RecipEV);
144   DestroyMatrix(Hilfs);
145   DestroyMatrix(Lambda);
146   DestroyVector(EigenValues);
147   FOR i := 1 TO n DO
148     BEGIN
149       FOR j := 1 TO n DO
150         BEGIN
151           Rneu := GetMatrixElement(R, i, j);
152           IF (Abs(Rneu) > 1.0)
153             THEN SetMatrixElement(R, i, j, sign(Rneu)); // max + OR - 1
154           END;
155           SetMatrixElement(R, i, i, 1.0);                // diagonal
156           elements 1
157         END;
158       UNTIL (iter >= 100) OR (NoNegs = 0);
159     END;
159 END.

```

The various methods of shrinkage perform similarly in practice, so numerical simplicity can be the overriding selection criterium [30]. The one exception is the recalculation of  $\mathcal{R}$  and  $\mathcal{S}$  from initial eigenanalysis, which raises suspicion against that method.

### 16.7.2. Effect of discrete variables on PCA

Component analysis was originally developed for interval- and rational scaled variables with multivariate normal distribution. In [32] the effects of the use of ordinal and binary scaled, and/or non-normally distributed variables (skew and kurtosis) is investigated. If one assumes that categorial data are generated by quantising underlying cardinal data (with  $< 5$  categories) the correlation coefficients will be biased towards 0 (their fig. 2). This will affect the principal component weights. Categorization can be viewed as a measurement error with nonlinear properties. Errors can be minimised if all categories have similar numbers of observation. PCA is quite robust toward different distances between the categories of a variable. On average, a discrete variable contains 2/3 of the information of the underlying cardinal variable.

Sometimes dummy variables corresponding to individual categories are used. *E.g.*, an ordinal value for affluence may be mode of transport (walk, bicycle, motorcycle, car). One

## 16. Principle component analysis

could replace this with four independent, binary variables (has car, has motorcycle...). However, the natural order of the variable, and hence information, would be lost. Also, the effect of quantisation is aggravated. Most weight is attached to the category with the highest number of observations. Also, **PCA** becomes numerically less stable.

It is incorrect to use PEARSON's correlation coefficient for nominal or ordinal data. The correlation coefficient between two variables is influenced by both their substantive similarity and by their statistical distributions. If the distributions are dissimilar, then **FA** or **PCA** may give a spurious multidimensional results. Polychoric correlations have been successfully tried with LIKERT-scale data [33].

### 16.7.3. Number of components

**PCA** is used to reduce the dimensionality of the data set, this makes interpretation easier and removes random noise. It may also make further analysis (multiple regression, clustering) more stable by removing co-linearity. However, it also leads to a loss of information. The question therefore is how many components are significant enough to be included in the following analysis. In general, too many components cause fewer problems than too few [33].

#### Scree-plot

In a plot of eigenvalue vs factor or component number (see fig. 16.9) [34] one sees initially a steep exponential decline ("cliff") followed by a shallow area with linear decline ("rubble" or "scree"). The cliff (data left of the "knee") represents the significant components. However, distinguishing cliff and scree can be difficult and subjective. The **acceleration factor** is calculated as the second derivative of the scree-curve by finite differences,  $f''(j) \approx (f(j+1) - 2f(j) + f(j-1))$ , it should have a maximum at the knee. This is usually not discernible, but for non-significant components this value oscillates around zero.

#### KAISER-criterion

All components with an eigenvalue  $> \bar{\lambda}$  are considered significant. This average is 1.0 when a  $z$ -standardised data matrix is used (or when the eigenanalysis is performed on  $\mathcal{R}$  rather than  $\mathcal{S}$ ), so the variance of each variable is 1.0. The eigenvalue is the sum of the squared loadings of a component with all variables and corresponds to the variance explained by this component. However, in an analysis with many variables the KAISER-criterion overestimates the number of significant components, as some eigenvalues will be  $> 1$  by chance (variables uncorrelated in the population have small correlation in the sample, leading to significant eigenvalues).

#### Variance explained

The retained components are supposed to retain the significant information of the data, but not the noise. One can simply set a minimal value for explained variance, say, 5 % or

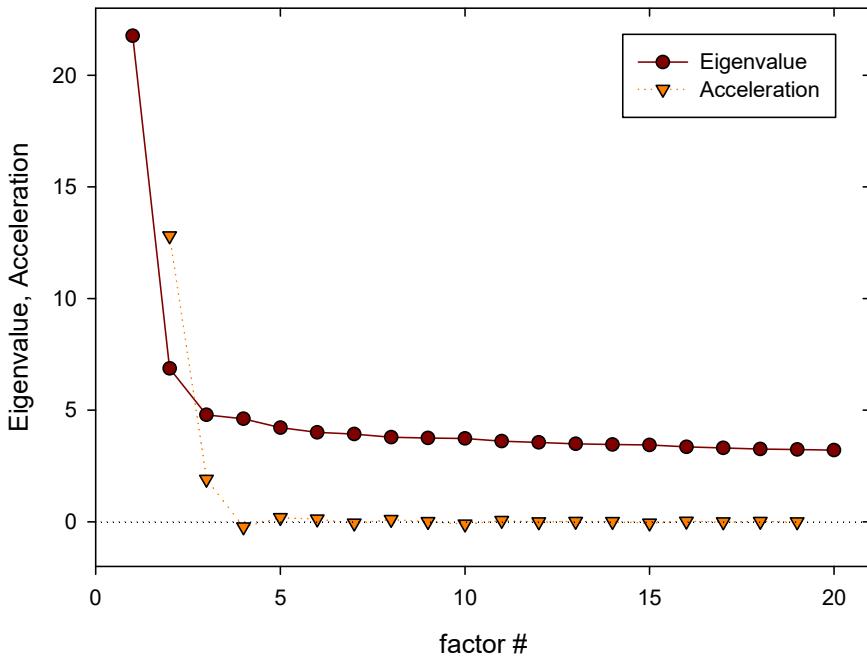


Figure 16.9.: Example for a scree-plot, from an investigation on validity of exam questions. The two leftmost components form the mountain and are used for analysis, the remaining components are discarded as rubble. The acceleration is the second derivative of the scree-plot and can sometimes help to identify the “knee”. Here, it would suggest that third eigenvalue may also be significant.

## 16. Principle component analysis

10 %. Components that account for less are discarded. Alternatively, enough components are retained so that the cumulative explained variance is 70–80 %.

### Parallel analysis

PCA is performed both with the empirical data set and with (at least 100) data sets of random numbers (if data are normally distributed) or a permuted data set (if data have a non-normal distribution). The number of cases and variables in the random set should be identical to the empirical. Only those eigenvalues in the empirical data are considered significant that are larger than those from the random data. Standard statistical packages do not include this computationally expensive method, so it is under-used.

### BARTLETT's test for sphericity

It is possible to use BARTLETT's sphericity test to calculate the significance of the difference between eigenvalues. Say, we have a test with  $n = 100$  persons and  $p = 3$  variables. The eigenvalues are 8, 4, 2 [1]. Then according to eqn. 16.8  $r = 0.8571, r^3 = 0.6295, \chi^2 = 44.66$  and  $f = 5$ , therefore  $P_0 < 0.1\%$ . Next, we eliminate the first eigenvalue and calculate the test values for the remaining eigenvalues:

$$\begin{aligned} r &= \frac{(4 \times 2)^{1/2}}{1/2 \times (4 + 2)} = \frac{\sqrt{8}}{3} \\ r^2 &= \frac{8}{9} \\ \chi^2 &= -(100 - 3 - 1/2)(-0.11779) = 11.37 \\ f &= \frac{(3 - 1 + 2)(3 - 1 - 1)}{2} = 2 \end{aligned} \tag{16.21}$$

We then take the difference between both  $\chi^2$ -values ( $44.66 - 11.37 = 33.29$ ) and degrees of freedom ( $5 - 2 = 3$ ) and use the result to test for the significance of the difference between the first and second eigenvalue ( $P_0 < 0.1\%$ ). This process is continued until that difference becomes non-significant or until only one eigenvalue is left.

From this procedure we learn how many data are required to *describe* the data, not necessarily how many can be used to *describe* them. For singular correlation matrices  $\chi^2$  will remain significant for all  $p$  components.

### Very simple structure (VSS)

This method compares the original correlation matrix with that reproduced from the retained eigenvalues and -vectors [35]. As the number of retained components increases, VSS index increases sharply to a maximum, then decreases slowly. Procedure:

1. Extract components by the method of your choice, and make an initial estimate of the number of relevant components  $q$

2. Rotate the components by the method of your choice, resulting in a rotated component matrix  $\mathcal{B}_{p \times q}$
3. Calculate VSS:
  - a) For a VSS of complexity  $v$ , replace the smallest  $q - v$  elements of  $\mathcal{B}$  by zero, resulting in a degraded component matrix  $\mathcal{B}'_{p \times q}$
  - b) Calculate the degraded correlation matrix  $\mathcal{R}' = \mathcal{B}'\Lambda\mathcal{B}'^{-1}$
  - c) calculate the mean square residual  $M$  of both  $\mathcal{R}$  and  $\mathcal{R}'$  and  $VSS_{v,p} = 1 - \frac{M_{\mathcal{R}'}}{M_{\mathcal{R}}}$
4. Repeat this for all  $q$  from 1 to the rank of the matrix and plot the results against  $q$

### VELICER's minimum average partial (MAP) criterium

[36]. The test has been revised with the partial correlations raised to the 4th power (rather than squared). Source code is available under <https://people.ok.ubc.ca/briocnn/nfactors/nfactors.html> and in the R-package `paramap`.

### Extrapolation

One can draw a line between the  $p$ -th eigenvalue and the  $i+1$ -th. Then one can estimate the  $i$ -th eigenvalue by extrapolation. Measured eigenvalues are considered relevant if they are significant larger than the estimated one. This method is used by the R-module `nFactors`.

### Newer methods

GAVISH & DONOHO [37] have shown that for a data matrix which is large compared to its rank, the optimal number of singular values to retain is  $\omega\tilde{\sigma}$ , where  $\tilde{\sigma}$  is the median of the singular values. For  $n \times p$  matrices,  $\omega$  depends on  $\beta = p/n$  as  $\omega(\beta) \approx 0.56\beta^3 - 0.95\beta^2 + 1.82\beta + 1.43$ .

MINKA [38] uses BAYESian statistics to derive a probability function that reaches its highest value at the optimal number of components

$$\begin{aligned}\hat{v} &= \frac{\sum_{j=q+1}^p \lambda_j}{p-q} \\ m &= pq - \frac{q(q+1)}{2} \\ P(\mathcal{X}|q) &= \left( \prod_{j=1}^q \lambda_i \right)^{-n/2} \hat{v}^{-n(q-q)/2} n^{-(m+q)/2}\end{aligned}\tag{16.22}$$

where  $\hat{v}$  is the average of the left-out eigenvalues, that is, the maximum-likelihood noise variance.

### Importance of minor components

Components that explain only a small percentage of the variance of the population as a whole may explain a large part of the variance of a small number of persons. For example, in a psychiatric test the answer to the question “Did somebody ever try to murder you?” will be “no” for the vast majority of subjects. A component loading on this question therefore might be dropped as noise. However, for the few people who answer “yes”, the discriminatory value of that question might be quite high. It can therefore be useful to look for individuals that score high on such minor components.

## 16.8. Rotation of factors or components

To make the result of a factor analysis (or [PCA](#) if it is used as proxy for [FA](#)) more interpretable the found factors are rotated to achieve a simple structure [39]. A simple structure should fulfill the following criteria (for more than four factors extracted):

- each row (variable) contains at least one near zero
- each column (factor) should have several near zeros
- for any pair of factors, there are
  - some variables with near zero loadings on one factor and large loadings on the other
  - a sizable number of variables that have near zero loadings on both
  - a small number of variables that have large loadings on both

Geometrically, the loadings in each row of the loading matrix constitute the coordinates of a point in loading space. If there are clusters of points, we try to put our axes near these clusters, so that each group of variables becomes associated with a factor. If all points are close to an axis, then they would load highly on the corresponding factor, and only little on all others. The complexity of a variable is the number of factors that this variable has moderate or high loadings on. The simple structure then has a complexity of 1.

In other words, the variance of factor loadings per factor is maximised. Note that since rotation is performed only on the factors retained, the result of rotation will change if more or fewer factors are retained. There are two classes of rotation, each with several methods [40–42]:

**orthogonal** only right angles between axes are allowed, factors are uncorrelated. The *orthomax function* maximised is  $Q_{\max} = \sum_{j,k=1}^{p,q} l_{jk}^4 - \gamma/p \sum_{k=1}^q (\sum_{j=1}^p l_{jk}^2)^2$ .

**Varimax** was introduced by KAISER [43, 44] and expressed in matrix terms by others [45, 46]. The axes are rotated until all loadings become either large or small, thereby number of variables with high loadings on several factors is minimised. The orthomax function is used with  $\gamma = 1$ .

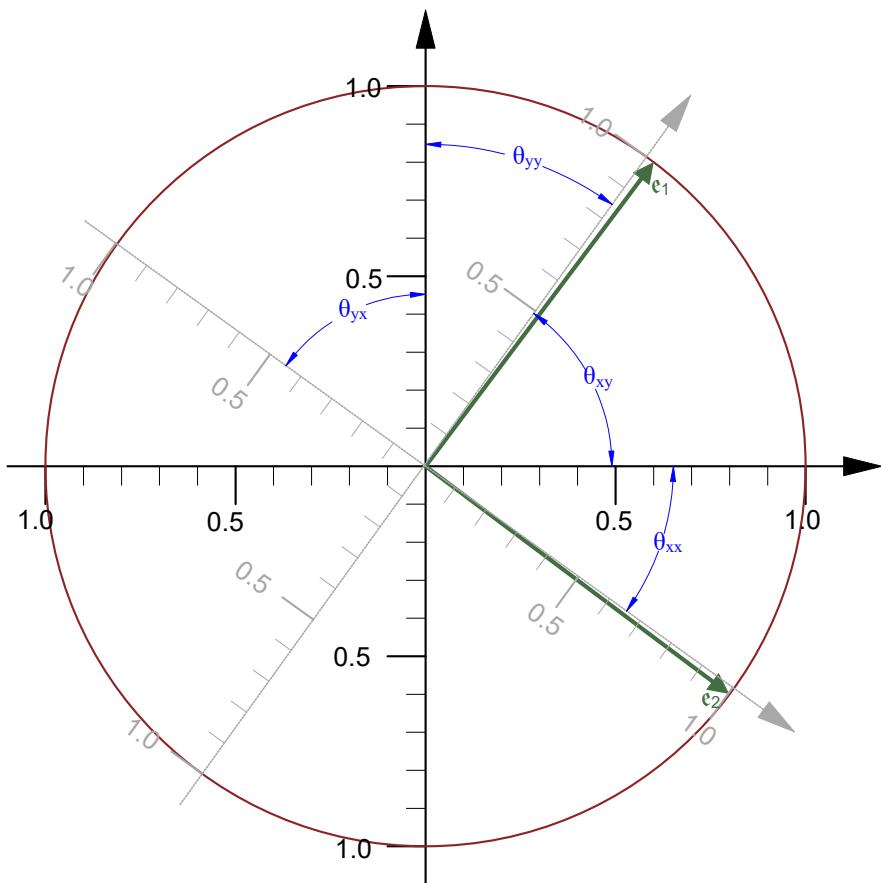


Figure 16.10.: Eigenvectors plotted into the coordinate system of the components obtained. All eigenvectors of PCA have unit length and point from the centre to the circumference of a unit circle (sphere or spheroid in higher dimensional cases). Rotation of the coordinate system (*blue arrow*) achieves a structure that is simpler to interpret. The angles between the original and the new coordinate axis are called  $\theta_{xy}, \theta_{xx}, \theta_{yx}, \theta_{yy}$ , respectively. They are measured in mathematically positive direction (counterclockwise).

## 16. Principle component analysis

**Quartimax** The number of factors required to explain a variable is minimised, this makes interpretation of the variables easier. The orthomax function is used with  $\gamma = 0$ .

**Equamax** is a mixture of Varimax and Quartimax.

**oblique** angle between axes may be different from 90, there is correlation between rotated factors. However, this correlation is small, as highly correlated factors would be better interpreted as one factor. Oblique rotations produce two matrices as solution, called pattern and structure matrix, respectively. The most important oblique methods are:

**Promax** is fast and therefore the most frequently used oblique rotation, especially with large data sets. A target matrix is calculated from a varimax rotation whose loadings are raised to a power  $2 < \kappa < 4$ , maintaining the sign. This reduces the loadings, but increases the ratio of loadings. Often, an average between the varimax and promax solutions is calculated (procrustean rotation).

**Oblimin** produces results that look very much like varimax, except that they are oblique. The degree of obliqueness allowed is controlled by a parameter  $\delta$ , negative values decrease factor correlation, positive increase it.  $\delta = -4$  produces uncorrelated factors, default in SPSS is 0, maximum should be 0.8 or so. The method minimises the cross-product between loadings, the *oblimin criterium* minimised is  $Q_{\min} = \sum_{r \neq s} \left( \sum_{j=1}^p l_{jr}^2 l_{js}^2 - \gamma/p \sum_{j=1}^p l_{jr}^2 \sum_{j=1}^p l_{js}^2 \right)$ . The special cases of  $\gamma = 0$  and  $\gamma = 1$  are called **quartimin** and **covarimin**, respectively.

An orthogonal rotation matrix  $\Theta$  is defined, where the rows represent the original and the columns the new axes [41]. Each element of the matrix is the cosine of the angle between the old and new axis, measured in mathematically positive direction (counterclockwise, see fig. 16.10)). For the 2D-case:

$$\Theta = \begin{pmatrix} \cos(\theta_{xx}) & \cos(\theta_{xy}) \\ \cos(\theta_{yx}) & \cos(\theta_{yy}) \end{pmatrix} = \begin{pmatrix} \cos(\theta_{xx}) & -\sin(\theta_{xx}) \\ \sin(\theta_{xx}) & \cos(\theta_{xx}) \end{pmatrix} \quad (16.23)$$

This matrix is orthonormal ( $\Theta^T \Theta = I$ ). The purpose of rotation is to maximise  $Q_{\max}$  (or minimise  $Q_{\min}$ )

Some authors claim that it is better to perform an oblique rotation first, and a orthogonal only if the resulting angles between factors are not very different from 90 [6]. This way, one doesn't enforce a structure which the data don't have. On the other hand, if two factors are correlated (angle significantly different from 90), then what causes the correlation? In addition, oblique rotation induces additional variability that will make replication of the results in future studies less likely (less factor invariance). The arbitrary setting of  $\kappa$  and  $\delta$  by the researcher adds to this problem. The better fit of oblique rotations may be a result of over-fitting. When **PCA** rather than **FA** is used to extract factors, the resulting factors (actually: components) are by definition uncorrelated and oblique rotation will produce similar results to orthogonal.

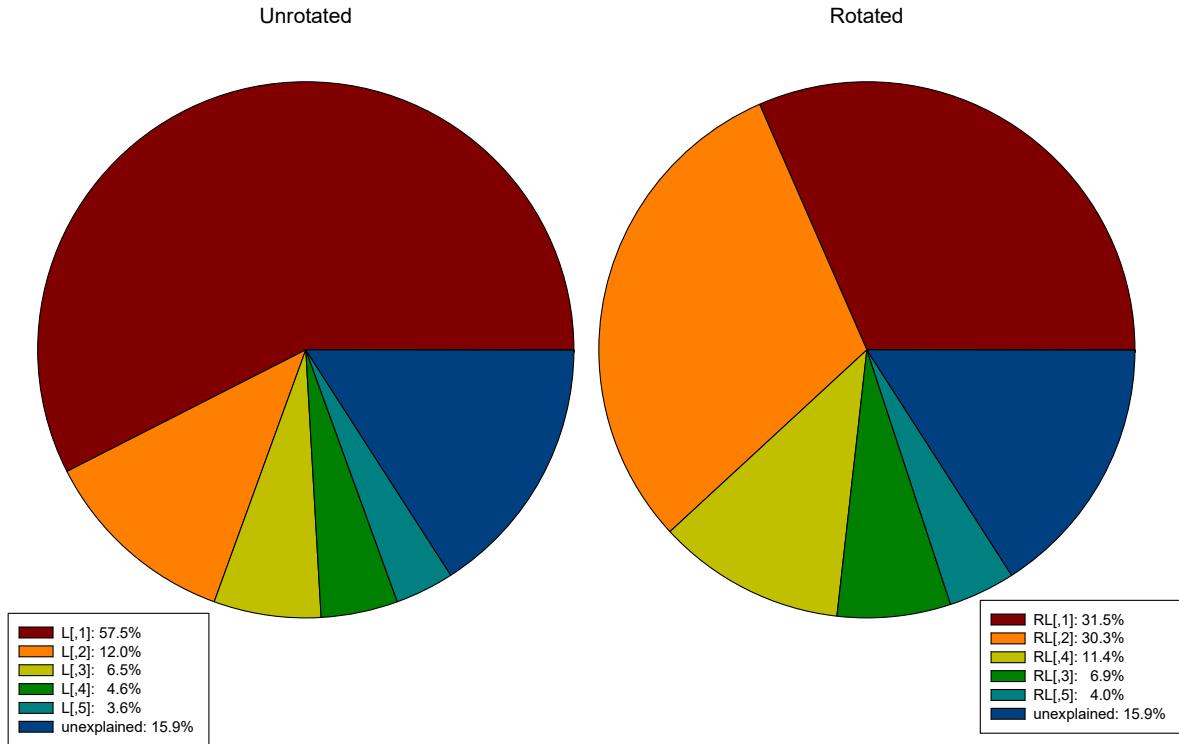


Figure 16.11.: Redistribution of explained variance between factors due to rotation. The total explained variance doesn't change, but the relative importance of the components does.

If there is a strong factor structure in the data, the results from various rotation methods should look similar, if they don't you have a problem anyway. Therefore, the simplest method (varimax) may be used as default.

After rotation, the new loading matrix  $\mathcal{L}'$  reproduce the covariance matrix just like  $\mathcal{L}$ :

$$\mathcal{S} = \mathcal{L}'\mathcal{L}'^T + \Psi = \mathcal{L}\theta\theta^T\mathcal{L}^T + \Psi = \mathcal{L}\mathcal{L}^T + \Psi \quad (16.24)$$

with  $\Psi_p$  the specific variance. However, the relative importance of factors changes, and is no longer given by the original eigenvalues (see fig. 16.11). A new variance-accounted-for statistics, the **trace**, is used. After orthogonal rotation, the trace are the column sums of squared factor loadings, and the communalities the row sums (just as in the unrotated case). For oblique rotations, the pattern coefficients are multiplied with the corresponding structure coefficient, the results are summed column-wise for the trace and row-wise for the communalities.

### 16.8.1. Mathematical procedure for orthogonal rotation

#### Gradient projection algorithm (GPA)

If  $\mathcal{L}$  is the loading matrix, and  $Q_{\min}(\mathcal{L})$  a rotation criterium, then this criterium is to be minimised over all possible rotations, defined in the rotation matrix  $\Theta$ . The rotation then becomes [47]  $\mathcal{L}' = \mathcal{L}\Theta$  and the function to be minimised over all  $\Theta$  is  $f(\Theta) = Q_{\max}(\mathcal{L}\Theta)$ . For algorithms like varimax, which originally required maximising a quality function, we simply take the negative. The basic algorithm then becomes:

1. Choose an  $\alpha \geq 0$  and a rotation matrix  $\Theta$  (can be the identity matrix or a random orthogonal matrix)
2. Compute  $\mathcal{G} = df/d\Theta$ , with  $df/d\Theta$  the matrix of partial derivatives
3. Move  $\alpha$  units down the gradient by computing a singular value decomposition  $\mathcal{U}\Delta\mathcal{V}^T$  of  $\mathcal{G} + \alpha\Theta$
4. Replace  $\Theta$  by  $\mathcal{U}\mathcal{V}^T$
5. either go to 2) or stop

This algorithm will converge stationary if  $\alpha$  is sufficiently large. The differential  $\mathcal{G} = df/d\Theta = \mathcal{L}^T \frac{dQ}{d\mathcal{L}}$  for the quartimax function  $Q(\mathcal{L}') = 1/4 \sum \sum l_{jk}^4$  becomes  $\frac{dQ}{d\mathcal{L}} = \mathcal{L}'^3$ , which is the element-wise cube of  $\mathcal{L}'$ . For quartimax and varimax rotation,  $\alpha = 0$  is sufficient to ensure convergence in a few steps. A fixed point of  $\Theta$ , that is a  $\Theta$  that doesn't change when the algorithm is applied to it, is also a stationary point of  $f(\Theta)$ . It can be shown that with sufficiently large  $\alpha$  the algorithm converges monotonically toward such a fixed point, independently of the starting value of  $\Theta$ . As a stationary point is approached, the measure  $v$  approaches zero:  $v = \|\text{skm}\Theta^T\mathcal{G}\| + \|(\mathcal{I} - \Theta\Theta^T)\mathcal{G}\|$ , where the skew-symmetric part of a square matrix is  $\text{skm}(\mathcal{M}) = 1/2\mathcal{M} - (\mathcal{M})^T$ . In practice, one can stop the iteration when  $v \leq 1 \times 10^{-6}$ .

$\alpha$  needs to be large enough to guarantee monotonicity, but too large an  $\alpha$  will make the algorithm run slower. Operationally, one can run the algorithm with a chosen  $\alpha$  (say, 0) and see if  $v$  monotonically declines. If not,  $\alpha$  needs to be increased by some  $\Delta\alpha$  until it does. [48, 49] provides more general optimisation criteria, component loss functions that use the second, rather than the fourth, power functions, for example an entropy function  $O(\mathcal{L}') = -\sum \sum l_{jk}^2 \log l_{jk}^2$  that results in simpler  $\mathcal{L}'$  than either quartimax or varimax, small changes in the loading matrix at start produce less big changes in final outcome. Code examples may be found at [50]. An interesting case of rotation are the tandem criteria [51]: A loading matrix is first rotated by tandem 1 to determine the number of relevant factors, the reduced loading matrix is then rotated by the tandem 2 algorithm.

Table 16.2.: Quality criteria and their differentials, all given for minimisation [49].  $\langle \mathcal{X}, \mathcal{Y} \rangle = \text{tr}(\mathcal{X}^T \mathcal{Y})$  represents the FROBENIUS-scalar product of matrices  $\mathcal{X}_{n \times p}, \mathcal{Y}_{n \times p}$ ,  $\langle \mathcal{X} \rangle = \langle \mathcal{X}^T, \mathcal{X} \rangle = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}$  the FROBENIUS-norm of the matrix  $\mathcal{X}$ ,  $\mathcal{X} \odot \mathcal{Y}$  the element-wise (HADAMARD-SCHUR) product and  $\mathcal{X}^2 = \mathcal{X} \odot \mathcal{X}$  the element-wise square of a matrix.  $\mathcal{I}$  is the identity matrix,  $\mathcal{N}$  a (compatible) square matrix with 0 in the diagonal, 1 everywhere else, and  $\mathcal{M}_{p \times p}$  a matrix with  $m_{ij} = 1/p$ .  $\mathbf{U}_p, \mathbf{U}_q$  are column vectors of 1.

Criterium	$Q_{\min}(\mathcal{L}')$	$\mathcal{G} = df/d\Theta$
quartimax	$-1/4 \sum_i \sum_j l_{ij} = -\langle \mathcal{L}'^2, \mathcal{L}'^2 \rangle / 4$	$-\mathcal{L}'^3$
oblimin	$\langle \mathcal{L}'^2, (\mathcal{I} - \gamma \mathcal{M}) \mathcal{L}'^2 \mathcal{N} \rangle / 4$	$\mathcal{L}' \odot [(\mathcal{I} - \gamma \mathcal{M}) \mathcal{L}'^2 \mathcal{N}]$
CRAWFORD-FERGUSON	$(1 - \kappa) \langle \mathcal{L}'^2, \mathcal{L}'^2 \mathcal{N} \rangle / 4 + \kappa \langle \mathcal{L}'^2, \mathcal{N} \mathcal{L}'^2 \rangle / 4$	$(1 - \kappa) \mathcal{L}' \odot \mathcal{L}'^2 \mathcal{N} + \kappa \mathcal{L}' \odot \mathcal{N} \mathcal{L}'^2$
quartimax	$\langle \mathcal{L}', \mathcal{K} = 0 \rangle$	$\langle \mathcal{L}', \mathcal{K} = 0 \rangle$
varimax	$\langle \mathcal{L}', \mathcal{K} = 1/p \rangle$	$\langle \mathcal{L}', \mathcal{K} = 1/p \rangle$
equamax	$\langle \mathcal{L}', \mathcal{K} = q/2p \rangle$	$\langle \mathcal{L}', \mathcal{K} = q/2p \rangle$
parsimax	$\langle \mathcal{L}', \mathcal{K} = \frac{q-1}{p+q-2} \rangle$	$\langle \mathcal{L}', \mathcal{K} = \frac{q-1}{p+q-2} \rangle$
factor parsimony	$\langle \mathcal{L}', \mathcal{K} = 1 \rangle$	$\langle \mathcal{L}', \mathcal{K} = 1 \rangle$
JENNICH minimum entropy	$-\langle \mathcal{L}'^2, \log \mathcal{L}'^2 \rangle / 2$	$-\mathcal{L}' \odot \log(\mathcal{L}'^2) - \mathcal{L}'$
Oblimax	$-\log \langle \mathcal{L}'^2 \rangle^2 + 2 \log \langle \mathcal{L}' \rangle^2$	$-\frac{4\mathcal{L}'^3}{\langle \mathcal{L}'^2 \rangle^2} + \frac{4\mathcal{L}'}{\langle \mathcal{L}' \rangle^2}$
tandem criteria I	$-\langle \mathcal{L}'^2, (\mathcal{L}' \mathcal{L}'^T)^2 \mathcal{L}'^2 \rangle$	$-4\mathcal{L}' \odot [(\mathcal{L}' \mathcal{L}'^T)^2 \mathcal{L}'^2]$
tandem criteria II	$\langle \mathcal{L}'^2, [\mathcal{U} \mathcal{U}^T - (\mathcal{L}' \mathcal{L}'^T)^2] \mathcal{L}'^2 \rangle$	$4\mathcal{L}' \odot [\mathcal{U} \mathcal{U}^T - ((\mathcal{L}' \mathcal{L}'^T)^2)] \mathcal{L}'^2 - 4[\mathcal{L}' \mathcal{L}'^T] [\mathcal{L}'^2 (\mathcal{L}'^2)^T] \mathcal{L}'$

## 16. Principle component analysis

### Penalised varimax

Interpretation of rotated factors may be complicated by the different communalities of each factor. It is possible to constrain rotation so that communalities are the same for all factors [52]. The varimax rotation does a fairly good job in this respect, which may be one of the reasons for its popularity. It maximises  $Q_{\max} = \sum_{j,k=1}^{p,q} l_{jk}^4 - 1/p \sum_{k=1}^q (\sum_{j=1}^p l_{jk}^2)^2$ . The rotation can be defined in matrix terms

$$\begin{aligned}\mathcal{M}(\Theta) &= \mathcal{N}^T \mathcal{O} \mathcal{N} \\ \mathcal{N} &= \mathcal{L}' \odot \mathcal{L}' \\ \mathcal{O} &= \mathcal{I} - \frac{\mathcal{I} \mathcal{I}^T}{p}\end{aligned}\tag{16.25}$$

where  $\odot$  represents the HADAMARD-SCHUR (elementwise) product of two matrices.  $\mathcal{M}/(p-1)$  is the covariance matrix of the squared  $\mathcal{L}'$ . Hence we have to maximise  $\sigma(\Theta) = \text{trace}(\mathcal{M}(\Theta))$ . In an orthogonal rotation, the sum of squared loadings is not changed,  $\text{trace}(\mathcal{L}^T \mathcal{L}) = \text{trace}(\mathcal{L}'^T \mathcal{L}') = \text{const}$ . The requirement that all factors should have equal sums of squares means that  $\mathcal{L}'_{.1}^T \mathcal{L}'_{.1} = \dots = \mathcal{L}'_{.q}^T \mathcal{L}'_{.q}$ . In order to achieve this, the varimax function must be modified with an additional constraint  $PV(\Theta) = \text{trace}(\mathcal{N}^T \mathcal{O} \mathcal{N} - \mu \mathcal{I}^T \mathcal{N} \mathcal{N}^T \mathcal{I})$  with  $\mu$  a positive number that controls the importance of the penalty term, adjusting the result from pure varimax for  $\mu = 0$  to equal sum of squares for large  $\mu$ . The gradient of this function is  $4\mathcal{L}^T \mathcal{L}' \odot [(\mathcal{O} - \mu \mathcal{I} \mathcal{I}^T) \mathcal{N}]$ .

### 16.8.2. Pascal code for rotation

Rotation of loading matrices is performed by gradient projection according to [49], using R and SAS example code deposited on [50]. For the time being, only orthogonal rotation is implemented. The rotation methods available are listed in an enumeration type. These can be used by the calling program to specify the desired method:

Listing 16.15: Interface of unit Rotation

```

1  UNIT Rotation;
2
3  { Rotation of loading matrices by gradient projection according to
4    C.A. Bernaards & R.I. Jennrich: Gradient Projection Algorithms and
5      Software
6      for Arbitrary Rotation Criteria in Factor Analysis,
7      Educ. Psychol. Meas. 65:5 (2005) 676-696, doi:10.1177/0013164404272507.
8      Some code in R and SAS is deposited on
9      http://www.stat.ucla.edu/research/gpa/ }
10
11 INTERFACE
12
13 USES MathFunc, Vector, Matrix, SingularValue;
```

```

13 CONST
14   Convergence = -5; // log OF convergence criterium considered success
15
16 TYPE
17   RotType = (Varimax, Quartimax, Equamax, Parsimax, Parsimony, Quartimin,
18     Biquartimin, Covarimin, Entropy, Tandem1, Tandem2);
19
20 PROCEDURE GradProjAlgOrth(VAR Loading, Rotation: MatrixTyp; RotAlg:
21   RotType);
22 { Gradient Projection Algorithm for orthogonal rotation. Rotation algorithm
23   used
24   is selected by RotAlg.  }
25
26
27
28 IMPLEMENTATION

```

## Rotation methods

The calculation of the gradient matrix and of the quality measure in each rotation are performed in procedures that depend on the rotation method used.

**Varimax** The varimax method [43] is one of the oldest, most used and generally successful rotation methods. As alternative to below procedure the CRAWFORD-FERGUSON-routine with  $\kappa = 1/p$  may also be used. Varimax is used for orthogonal rotation.

Listing 16.16: Varimax

```

1 PROCEDURE FVarimax(CONST ActLoading: MatrixTyp; VAR Gradient: MatrixTyp;
2   VAR Quality: double);
3
4 VAR
5   cm, L2, QL, Zwischen: MatrixTyp;
6   n: WORD;
7
8 BEGIN
9   HadamardSchurProduct(ActLoading, ActLoading, L2); // L^2
10  n := MatrixRows(L2);
11  CreateMatrix(cm, n, n, 1.0 / n);
12  MatrixInnerProduct(cm, L2, Zwischen);
13  DestroyMatrix(cm);
14  NegativeMatrix(Zwischen);
15  MatrixAdd(L2, Zwischen, QL);
16  DestroyMatrix(L2);
17  DestroyMatrix(Zwischen);

```

## 16. Principle component analysis

```

18   Quality := -FrobeniusNorm(QL) / 4;
19   CopyMatrix(ActLoading, Zwischen);
20   NegativeMatrix(Zwischen);
21   HadamardSchurProduct(Zwischen, QL, Gradient);
22   DestroyMatrix(Zwischen);
23   DestroyMatrix(QL);
24 END;

```

**Quartimax** Quartimax is also used for orthogonal rotation. It is less often used than varimax, as it may produce a general factor that loads on most variables. As alternative to below procedure the CRAWFORD-FERGUSON-routine with  $\kappa = 0$  may also be used.

Listing 16.17: Quartimax

```

1 PROCEDURE FQuartimax (CONST ActLoading : MatrixTyp;
2                         VAR Gradient : MatrixTyp;
3                         VAR Quality : float);
4
5 VAR L2 : MatrixTyp;
6
7 BEGIN
8   HadamardSchurProduct(ActLoading, ActLoading, L2); // L^2
9   Quality := -0.25 * sqr(FrobeniusNorm(L2));          // -1/4 <L^2>^2
10  HadamardSchurProduct(L2, ActLoading, Gradient);     // L^3
11  NegativeMatrix(Gradient);                           // -L^3
12  DestroyMatrix(L2);
13 END;

```

**The oblimin family** Oblimin rotations are used for oblique rotations. Obliqueness is controlled by the parameter  $\gamma$ :

**0.0** quartimin (most oblique)

**0.5** bi-quartimin

**1.0** covarimin

Listing 16.18: Oblimin

```

1 PROCEDURE FOblimin(CONST ActLoading: MatrixTyp; gamma: double;
2                     VAR Gradient: MatrixTyp; VAR Quality: double);
3 { gamma determines type:
4   0.0 quartimin (most oblique)
5   0.5 Bi-quartimin
6   1.0 covarimin }
7
8 VAR

```

```

9  Rows, Columns, i, j: WORD;
10 Ident, N, M, Zwischen, Zwischen2, L2: MatrixTyp;
11
12 BEGIN
13   Rows := MatrixRows(ActLoading);
14   Columns := MatrixColumns(ActLoading);
15   CreateIdentityMatrix(Ident, Columns);
16   NegativeMatrix(Ident);
17   CreateMatrix(Zwischen, Columns, Columns, 1.0);
18   MatrixAdd(Zwischen, Ident, N);
19   DestroyMatrix(Zwischen);
20   DestroyMatrix(Ident);
21   CreateIdentityMatrix(Ident, Rows);
22   FOR i := 1 TO Rows DO
23     FOR j := 1 TO Rows DO
24       SetMatrixElement(Ident, i, j, GetMatrixElement(Ident, i, j) - gamma);
25   CreateMatrix(M, Rows, Rows, 1 / Rows);
26   HadamardSchurProduct(ActLoading, ActLoading, L2); // L^2
27   MatrixInnerProduct(L2, N, Zwischen);
28   HadamardSchurProduct(Ident, M, Zwischen2);
29   DestroyMatrix(Ident);
30   DestroyMatrix(M);
31   DestroyMatrix(N);
32   MatrixInnerProduct(Zwischen2, Zwischen, N);
33   // [(I(p)-gamma \cdot M) * L2 * N]
34   DestroyMatrix(Zwischen);
35   DestroyMatrix(Zwischen2);
36   Quality := FrobeniusSkalarProduct(L2, N) / 4;
37   // <L2, [(I(p)-gamma \cdot M) * L2 * N]>
38   HadamardSchurProduct(ActLoading, N, Gradient);
39   // L \cdot [(I(p)-gamma \cdot M) * L2 * N]
40   DestroyMatrix(N);
41   DestroyMatrix(L2);
42 END;

```

**Entropy** This method was introduced as alternative to varimax in [49]; it gives good results for orthogonal, but poor for oblique rotations.

Listing 16.19: Entropy

```

1  PROCEDURE FEntropy(CONST ActLoading: MatrixTyp; VAR Gradient: MatrixTyp;
2    VAR Quality: double);
3
4  VAR
5    L2, lnL2, Zwischen: MatrixTyp;
6    i, j: WORD;
7

```

## 16. Principle component analysis

```

8  BEGIN
9      HadamardSchurProduct(ActLoading, ActLoading, L2);           // L^2
10     CopyMatrix(L2, lnL2);
11     FOR i := 1 TO MatrixRows(lnL2) DO
12         FOR j := 1 TO MatrixColumns(lnL2) DO
13             SetMatrixElement(lnL2, i, j, Ln(GetMatrixElement(lnL2, i, j)));
14         Quality := -FrobeniusSkalarProduct(L2, lnL2) / 2;           // -<L2,
15             log(L2)>/2
16         DestroyMatrix(L2);
17         HadamardSchurProduct(ActLoading, lnL2, Zwischen);
18         DestroyMatrix(lnL2);
19         MatrixAdd(Zwischen, ActLoading, Gradient);                  // L \cdot Ln(L) -
20             L
21         NegativeMatrix(Gradient);
22         DestroyMatrix(Zwischen);
23     END;

```

**Tandem criteria** Introduced in [51]. Criterion I is based upon the principle that variables which appear on the same factor should be correlated and is used to determine the number of relevant factors. Criterion II is based upon the principle that variables which are uncorrelated should not appear on the same factor, and is used to explore factor structure.

Listing 16.20: Tandem criteria

```

1  PROCEDURE FTandem1(CONST ActLoading: MatrixTyp; VAR Gradient: MatrixTyp;
2      VAR Quality: double);
3
4  VAR
5      TL, TL2, LL, LL2, L2, gq1, gq2, Zwischen, Zwischen1: MatrixTyp;
6
7  BEGIN
8      MatrixTranspose(ActLoading, TL);
9      MatrixInnerProduct(ActLoading, TL, LL);
10     DestroyMatrix(TL);
11     HadamardSchurProduct(LL, LL, LL2);
12     HadamardSchurProduct(ActLoading, ActLoading, L2);
13     MatrixTranspose(L2, TL2);
14     MatrixInnerProduct(LL2, L2, Zwischen);
15     DestroyMatrix(LL2);
16     MatrixInnerProduct(TL2, Zwischen, Zwischen1);
17     Quality := -MatrixTrace(Zwischen1);
18     DestroyMatrix(Zwischen1);
19     HadamardSchurProduct(ActLoading, Zwischen, gq1);
20     SkalarMultiplikation(gq1, 4);
21     DestroyMatrix(Zwischen);
22     MatrixInnerProduct(L2, TL2, Zwischen);

```

```

23 DestroyMatrix(TL2);
24 DestroyMatrix(L2);
25 HadamardSchurProduct(LL, Zwischen, Zwischen1);
26 DestroyMatrix(LL);
27 DestroyMatrix(Zwischen);
28 MatrixInnerProduct(Zwischen1, ActLoading, gq2);
29 SkalarMultiplikation(gq2, 4);
30 DestroyMatrix(Zwischen1);
31 NegativeMatrix(gq1);
32 NegativeMatrix(gq2);
33 MatrixAdd(gq1, gq2, Gradient);
34 DestroyMatrix(gq1);
35 DestroyMatrix(gq2);
36 END;
37
38
39 PROCEDURE FTandem2(CONST ActLoading: MatrixTyp; VAR Gradient: MatrixTyp;
40   VAR Quality: double);
41
42 VAR
43   TL, TL2, LL, LL2, L2, U, TU, UU, gq1, gq2,
44   Zwischen, Zwischen1, Copy : MatrixTyp;
45   p: WORD;
46
47 BEGIN
48   p := MatrixRows(ActLoading);
49   MatrixTranspose(ActLoading, TL);
50   MatrixInnerProduct(ActLoading, TL, LL);
51   DestroyMatrix(TL);
52   HadamardSchurProduct(LL, LL, LL2);
53   NegativeMatrix(LL2);
54   DestroyMatrix(LL);
55   CopyMatrix(ActLoading, Copy);
56   HadamardSchurProduct(Copy, Copy, L2);
57   CreateMatrix(U, p, 1, 1.0);
58   CreateMatrix(TU, 1, p, 1.0);
59   MatrixInnerProduct(U, TU, UU);
60   DestroyMatrix(U);
61   DestroyMatrix(TU);
62   MatrixAdd(UU, LL2, Zwischen);
63   DestroyMatrix(UU);
64   MatrixInnerProduct(Zwischen, L2, Zwischen1);
65   Quality := FrobeniusSkalarProduct(L2, Zwischen1);
66   SkalarMultiplikation(Copy, 4);
67   HadamardSchurProduct(Copy, Zwischen1, gq1); {*****}
68   DestroyMatrix(Copy);

```

## 16. Principle component analysis

```

69  DestroyMatrix(Zwischen);
70  DestroyMatrix(Zwischen1);
71  MatrixTranspose(L2, TL2);
72  MatrixInnerProduct(L2, TL2, Zwischen);
73  DestroyMatrix(L2);
74  DestroyMatrix(TL2);
75  HadamardSchurProduct(LL2, Zwischen, Zwischen1);
76  DestroyMatrix(Zwischen);
77  DestroyMatrix(LL2);
78  MatrixInnerProduct(Zwischen1, Copy, gq2);
79  DestroyMatrix(Zwischen1);
80  SkalarMultiplikation(gq2, -4);
81  MatrixAdd(gq1, gq2, Gradient);
82  DestroyMatrix(gq1);
83  DestroyMatrix(gq2);
84 END;

```

**CRAWFORD-FERGUSON** This is a general method for orthogonal rotation [53], that depending on the parameter  $\kappa$  performs like

**0** Quartimax

$1/p$  Varimax

$q/(2 * p)$  Equamax

$(q - 1)/(p + q - 2)$  Parsimax

**1** Factor parsimony

Listing 16.21: CRAWFORD-FERGUSON

```

1 PROCEDURE FCrawfordFerguson(CONST ActLoading: MatrixTyp; kappa: double;
2   VAR Gradient: MatrixTyp; VAR Quality: double);
3
4 VAR
5   Rows, Columns: WORD;
6   Ident, N, M, Zwischen, L2, g1, g2: MatrixTyp;
7   f1, f2: double;
8
9 BEGIN
10  Rows := MatrixRows(ActLoading);
11  Columns := MatrixColumns(ActLoading);
12  CreateIdentityMatrix(Ident, Columns);
13  NegativeMatrix(Ident);
14  CreateMatrix(Zwischen, Columns, Columns, 1.0);
15  MatrixAdd(Zwischen, Ident, N);

```

```

16 DestroyMatrix(Ident);
17 DestroyMatrix(Zwischen);
18 CreateIdentityMatrix(Ident, Rows);
19 NegativeMatrix(Ident);
20 CreateMatrix(Zwischen, Rows, Rows, 1.0);
21 MatrixAdd(Zwischen, Ident, M);
22 DestroyMatrix(Ident);
23 DestroyMatrix(Zwischen);
24 HadamardSchurProduct(ActLoading, ActLoading, L2); // L^2
25 MatrixInnerProduct(L2, N, Zwischen);
26 f1 := (1 - kappa) * FrobeniusSkalarProduct(L2, Zwischen) / 4;
27 // (1-kappa) <L2, (L2*N)> / 4
28 HadamardSchurProduct(ActLoading, Zwischen, g1); // L \cdot
29   (L2 * N)
30 DestroyMatrix(Zwischen);
31 DestroyMatrix(N);
32 MatrixInnerProduct(M, L2, Zwischen);
33 f2 := kappa * FrobeniusSkalarProduct(L2, Zwischen) / 4; // kappa *
34   <L2, (M*L2)> / 4
35 HadamardSchurProduct(ActLoading, Zwischen, g2); // L \cdot (M
36   * L2)
37 DestroyMatrix(Zwischen);
38 DestroyMatrix(M);
39 Quality := f1 + f2;
40 SkalarMultiplikation(g1, 1 - kappa);
41 SkalarMultiplikation(g2, kappa);
42 MatrixAdd(g1, g2, Gradient);
43 DestroyMatrix(L2);
44 DestroyMatrix(g1);
45 DestroyMatrix(g2);
46 END;

```

BENDLER's method [54]:

Listing 16.22: BENDLER

```

1 PROCEDURE FBentler(CONST ActLoading: MatrixTyp; VAR Gradient: MatrixTyp;
2   VAR Quality: double); // funktioniert nicht
3
4 VAR
5   L2, TL2, M, D, Zwischen, Zwischen1: MatrixTyp;
6   Det1, Det2: double;
7
8 BEGIN
9   HadamardSchurProduct(ActLoading, ActLoading, L2);
10  MatrixTranspose(L2, TL2);
11  MatrixInnerProduct(TL2, L2, M);

```

## 16. Principle component analysis

```

12  DestroyMatrix(TL2);
13  CopyMatrix(M, D);
14  Det1 := Determinante(D);
15  Diag(D);
16  Det2 := Determinante(D);
17  Quality := -Ln(Det1 / Det2);
18  InverseMatrix(M);
19  InverseMatrix(D);
20  MatrixInnerProduct(M, D, Zwischen);
21  DestroyMatrix(M);
22  DestroyMatrix(D);
23  MatrixInnerProduct(L2, Zwischen, Zwischen1);
24  DestroyMatrix(L2);
25  DestroyMatrix(Zwischen);
26  CopyMatrix(Zwischen1, Zwischen);
27  SkalarMultiplikation(Zwischen, -4);
28  HadamardSchurProduct(Zwischen, Zwischen1, Gradient);
29  DestroyMatrix(Zwischen);
30  DestroyMatrix(Zwischen1);
31 END;

```

### Gradient projection algorithm for orthogonal rotation

Listing 16.23: Implementation of the gradient projection algorithm for orthogonal rotation

```

1 PROCEDURE GradProjAlgOrth(VAR Loading, Rotation: MatrixTyp; RotAlg: RotType);
2
3 VAR
4   alpha, s, s1, Q, Qold: double;
5   Columns, Rows, j, Iter: WORD;
6   NewLoading, LoadingTrans, RotationTrans, GradQ, Grad, GradP,
7   Manifold, ManifoldTrans, Zwischen, Skm2, X, V, VT: MatrixTyp;
8   Delta: VectorTyp;
9
10 BEGIN
11   alpha := 1.0;
12   Columns := MatrixColumns(Loading);
13   Rows := MatrixRows(Loading);
14   Writeln;
15   Writeln('Rotation:');
16   Writeln('Iter Q      s          Log10(s)  alpha ');
17   MatrixInnerProduct(Loading, Rotation, NewLoading);
18   CreateMatrix(GradQ, Columns, Columns, 0.0);
19   CASE RotAlg OF
20     Varimax: FCrawfordFerguson(NewLoading, 1 / Rows, GradQ, Q);

```

```

21 Quartimax: FCrawfordFerguson(NewLoading, 0.0, GradQ, Q);
22 Equamax: FCrawfordFerguson(NewLoading, Columns / (2 * Rows), GradQ, Q);
23 Parsimax: FCrawfordFerguson(NewLoading, Pred(Columns) /
24     (Rows + Columns - 2), GradQ, Q);
25 Parsimony: FCrawfordFerguson(NewLoading, 1.0, GradQ, Q);
26 Quartimin: FOblimin(NewLoading, 0.0, GradQ, Q);
27 Biquartimin: FOblimin(NewLoading, 0.5, GradQ, Q);
28 Covarimin: FOblimin(NewLoading, 1.0, GradQ, Q);
29 Entropy: FEntropy(NewLoading, GradQ, Q);
30 Tandem1: FTandem1(NewLoading, GradQ, Q);
31 Tandem2: FTandem2(NewLoading, GradQ, Q);
32 // Bentler : FBentler(NewLoading, GradQ, Q);
33 END;
34 Qold := Q;
35 MatrixTranspose>Loading, LoadingTrans);
36 MatrixInnerProduct(LoadingTrans, GradQ, Grad); // calculate gradient
37 Iter := 0;
38 REPEAT
39 MatrixTranspose(Rotation, RotationTrans);
40 MatrixInnerProduct(RotationTrans, Grad, Manifold); // M = T' * G
41 MatrixTranspose(Manifold, ManifoldTrans);
42 MatrixAdd(Manifold, ManifoldTrans, Skm2);
43 SkalarMultiplikation(Skm2, 0.5); // S = (M+M')/2
44 MatrixInnerProduct(Rotation, Skm2, Zwischen);
45 DestroyMatrix(Skm2);
46 NegativeMatrix(Zwischen);
47 MatrixAdd(Grad, Zwischen, GradP); // Gp = G - T*S
48 DestroyMatrix(Zwischen);
49 s := FrobeniusNorm(GradP);
50 s1 := log(s, 10);
51 Writeln(Iter: 3, ' ', Q: 2: 3, ' ', s: 2: 8, ' ', s1: 2: 3, ' ',
52 alpha: 2: 2);
53 IF (s1 > Convergence)
54 THEN
55 BEGIN
56     alpha := 2 * alpha;
57     j := 0;
58     REPEAT // ensure that Q
59         IS decreased
60         DestroyMatrix(NewLoading);
61         SkalarMultiplikation(GradP, -alpha);
62         MatrixAdd(Rotation, GradP, X); // X = T - a1*Gp
63         SVDCmp(X, Delta, V); // X now
64             contains U
65             MatrixTranspose(V, Vt);
66             DestroyMatrix(V);

```

## 16. Principle component analysis

```

64      MatrixInnerProduct(X, Vt, Zwischen);           // Tt = U T'
65      MatrixInnerProduct>Loading, Zwischen, NewLoading); // L = A * Tt
66      CASE RotAlg OF
67          Quartimax: FCrawfordFerguson(NewLoading, 0.0, GradQ, Q);
68          Varimax: FCrawfordFerguson(NewLoading, 1 / Rows, GradQ, Q);
69          Equamax: FCrawfordFerguson(NewLoading, Columns / (2 * Rows),
70                                         GradQ, Q);
71          Parsimax: FCrawfordFerguson(NewLoading, pred(Columns) /
72                                         (Rows + Columns - 2), GradQ, Q);
73          Parsimony: FCrawfordFerguson(NewLoading, 1.0, GradQ, Q);
74          Quartimin: FOblimin(NewLoading, 0.0, GradQ, Q);
75          Biquartimin: FOblimin(NewLoading, 0.5, GradQ, Q);
76          Covarimin: FOblimin(NewLoading, 1.0, GradQ, Q);
77          Entropy: FEntropy(NewLoading, GradQ, Q);
78          Tandem1: FTandem1(NewLoading, GradQ, Q);
79          Tandem2: FTandem2(NewLoading, GradQ, Q);
80          // Bentler : FBentler(NewLoading, GradQ, Q);
81      END;
82      IF (Q > (Qold - 0.5 * s * s * alpha))
83          THEN alpha := alpha / 2;
84      Inc(j);
85      DestroyMatrix(X);
86      DestroyMatrix(Vt);
87      DestroyVector(Delta);
88      UNTIL ((j = 10) OR (Q < (Qold - 0.5 * s * s * alpha)));
89      CopyMatrix(Zwischen, Rotation);
90      DestroyMatrix(Zwischen);
91      END;
92      Qold := Q;
93      MatrixInnerProduct>LoadingTrans, GradQ, Grad);
94      Inc(Iter);
95      DestroyMatrix(RotationTrans);
96      DestroyMatrix(Manifold);
97      DestroyMatrix(ManifoldTrans);
98      UNTIL ((s1 < Convergence) OR (Iter > MaxIter));
99      DestroyMatrix>NewLoading);
100     MatrixInnerProduct>Loading, Rotation, NewLoading);
101     DestroyMatrix>Loading);
102     CopyMatrix>NewLoading, Loading);
103     DestroyMatrix>NewLoading);
104     DestroyMatrix>LoadingTrans);
105     DestroyMatrix>GradQ);
106     DestroyMatrix>GradP);
107     DestroyMatrix>Grad);
108 END;
```

## Sorted absolute loadings

The sorted absolute loadings (SAL) of a loading matrix can be used for a SAL-plot [49], which is a simple method to see how well the rotation worked. SAL plotted versus number should result in a sigmoidal function.

Listing 16.24: Calculation of SAL

```

1 PROCEDURE CalculateSAL(CONST Loading: MatrixTyp; VAR SAL: VectorTyp);
2
3 VAR
4   i, j, Rows, Columns, Length: word;
5
6 BEGIN
7   Rows := MatrixRows(Loading);
8   Columns := MatrixColumns(Loading);
9   Length := Rows * Columns;
10  CreateVector(SAL, Length, 0.0);
11  FOR i := 1 TO Rows DO
12    FOR j := 1 TO Columns DO
13      SetVectorElement(SAL, i * j, abs(GetMatrixElement(Loading, i, j)));
14  Shellsort(SAL);
15 END;
16
17 END.

```

### 16.8.3. Examples

#### THURSTONE's box problem

The initial loading matrix  $\mathcal{L}$  of THURSTONE's box problem is:

## 16. Principle component analysis

#	1	2	3
1	0.659	-0.736	0.138
2	0.725	0.180	-0.656
3	0.665	0.537	0.500
4	0.869	-0.209	-0.443
5	0.834	0.182	0.508
6	0.836	0.519	0.152
7	0.856	-0.452	-0.269
8	0.848	-0.426	0.320
9	0.861	0.416	-0.299
10	0.880	-0.341	-0.354
11	0.889	-0.417	0.436
12	0.875	0.485	-0.093
13	0.667	-0.725	0.109
14	0.717	0.246	-0.619
15	0.634	0.501	0.522
16	0.936	0.257	0.165
17	0.966	-0.239	-0.083
18	0.625	-0.720	0.166
19	0.702	0.112	-0.650
20	0.664	0.536	0.488

Using quartimax rotation and the  $\mathcal{I}_3$  as initial rotation matrix the iteration should converge as follows

iter	f	log10	alpha
0	-10.7152	-0.1406	1.0000
1.0000	-13.2425	0.3893	2.0000
2.0000	-14.1458	0.0407	0.2500
3.0000	-14.1964	-0.4122	0.0625
4.0000	-14.2029	-0.7978	0.0625
5.0000	-14.2041	-1.0890	0.0625
6.0000	-14.2046	-1.6858	0.1250
7.0000	-14.2046	-2.2221	0.1250
8.0000	-14.2046	-2.5689	0.0625
9.0000	-14.2046	-3.1207	0.1250
10.0000	-14.2046	-3.4477	0.0625
11.0000	-14.2046	-4.0147	0.1250
12.0000	-14.2046	-4.3231	0.0625
13.0000	-14.2046	-4.9043	0.1250
14.0000	-14.2046	-5.1958	0.0625

The rotation matrix  $\theta$  becomes

	1	2	3
0.5723	-0.6075	-0.5508	
0.6025	0.7672	-0.2201	
0.5563	-0.2059	0.8051	

The final loading matrix  $\mathcal{L}'$  is ( $|l| > 0.7$  in red,  $< 0.3$  in blue):

#	1	2	3
1	0.0105	-0.9934	-0.0899
2	0.1585	-0.1673	-0.9671
3	0.9823	-0.0950	-0.0819
4	0.1250	-0.5971	-0.7893
5	0.8696	-0.4716	-0.0904
6	0.8757	-0.1410	-0.4523
7	0.0679	-0.8114	-0.5886
8	0.4067	-0.9079	-0.1157
9	0.5771	-0.1424	-0.8065
10	0.1013	-0.7233	-0.6946
11	0.5001	-0.9497	-0.0468
12	0.7413	-0.1403	-0.6636
13	0.0056	-0.9838	-0.1200
14	0.2142	-0.1194	-0.9474
15	0.9551	-0.1083	-0.0392
16	0.7823	-0.4054	-0.4393
17	0.3627	-0.7531	-0.5463
18	0.0162	-0.9662	-0.0521
19	0.1077	-0.2067	-0.9346
20	0.9744	-0.0926	-0.0908

## Football players

Table 16.8.3 lists data collected by G.R.BRYCE and R.M.BARKER (Brigham Young University) as part of a preliminary study of a possible link between football helmet design and neck injuries:

Table 16.3.: WDIM = head width at widest dimension, CIRCUM = head circumference, FBEYE = front-to-back measurement at eye level, EYEHD = eye-to-top-of-head measurement, EARHD = ear-to-top-of-head measurement, JAW = jaw width. Group 2: college players Group 3: college non-players.

16. Principle component analysis

Group	WDIM	CIRCUM	FBEYE	EYEHLD	EARHD	JAW
2	15.5	60.0	21.1	10.3	13.4	12.4
2	15.4	59.7	20.0	12.8	14.5	11.3
2	15.1	59.7	20.2	11.4	14.1	12.1
2	14.3	56.9	18.9	11.0	13.4	11.0
2	14.8	58.0	20.1	9.6	11.1	11.7
2	15.2	57.5	18.5	9.9	12.8	11.4
2	15.4	58.0	20.8	10.2	12.8	11.9
2	16.3	58.0	20.1	8.8	13.0	12.9
2	15.5	57.0	19.6	10.5	13.9	11.8
2	15.0	56.5	19.6	10.4	14.5	12.0
2	15.5	57.2	20.0	11.2	13.4	12.4
2	15.5	56.5	19.8	9.2	12.8	12.2
2	15.7	57.5	19.8	11.8	12.6	12.5
2	14.4	57.0	20.4	10.2	12.7	12.3
2	14.9	54.8	18.5	11.2	13.8	11.3
2	16.5	59.8	20.2	9.4	14.3	12.2
2	15.5	56.1	18.8	9.8	13.8	12.6
2	15.3	55.0	19.0	10.1	14.2	11.6
2	14.5	55.6	19.3	12.0	12.6	11.6
2	15.5	56.5	20.0	9.9	13.4	11.5
2	15.2	55.0	19.3	9.9	14.4	11.9
2	15.3	56.5	19.3	9.1	12.8	11.7
2	15.3	56.8	20.2	8.6	14.2	11.5
2	15.8	55.5	19.2	8.2	13.0	12.6
2	14.8	57.0	20.2	9.8	13.8	10.5
2	15.2	56.9	19.1	9.6	13.0	11.2
2	15.9	58.8	21.0	8.6	13.5	11.8
2	15.5	57.3	20.1	9.6	14.1	12.3
2	16.5	58.0	19.5	9.0	13.9	13.3
2	17.3	62.6	21.5	10.3	13.8	12.8
3	14.9	56.5	20.4	7.4	13.0	12.0
3	15.4	57.5	19.5	10.5	13.8	11.5
3	15.3	55.4	19.2	9.7	13.3	11.5
3	14.6	56.0	19.8	8.5	12.0	11.5
3	16.2	56.5	19.5	11.5	14.5	11.8
3	14.6	58.0	19.9	13.0	13.4	11.5
3	15.9	56.7	18.7	10.8	12.8	12.6

c

continued on next page

c	continued from previous page						
3	14.7	55.8	18.7	11.1	13.9	11.2	
3	15.5	58.5	19.4	11.5	13.4	11.9	
3	16.1	60.0	20.3	10.6	13.7	12.2	
3	15.2	57.8	19.9	10.4	13.5	11.4	
3	15.1	56.0	19.4	10.0	13.1	10.9	
3	15.9	59.8	20.5	12.0	13.6	11.5	
3	16.1	57.7	19.7	10.2	13.6	11.5	
3	15.7	58.7	20.7	11.3	13.6	11.3	
3	15.3	56.9	19.6	10.5	13.5	12.1	
3	15.3	56.9	19.5	9.9	14.0	12.1	
3	15.2	58.0	20.6	11.0	15.1	11.7	
3	16.6	59.3	19.9	12.1	14.6	12.1	
3	15.5	58.2	19.7	11.7	13.8	12.1	
3	15.8	57.5	18.9	11.8	14.7	11.8	
3	16.0	57.2	19.8	10.8	13.9	12.0	
3	15.4	57.0	19.8	11.3	14.0	11.4	
3	16.0	59.2	20.8	10.4	13.8	12.2	
3	15.4	57.6	19.6	10.2	13.9	11.7	
3	15.8	60.3	20.8	12.4	13.4	12.1	
3	15.4	55.0	18.8	10.7	14.2	10.8	
3	15.5	58.4	19.8	13.1	14.5	11.7	
3	15.7	59.0	20.4	12.1	13.0	12.7	
3	17.3	61.7	20.7	11.9	13.3	13.3	
$\bar{\chi}_i$		15.5	57.6	19.8	10.5	13.6	11.9
$\sigma(\bar{\chi}_i)$		0.6	1.6	0.7	1.2	0.7	0.6

The var-covariance matrix and correlation matrix are:

$$\mathcal{S} = \begin{pmatrix} 0.3702 & 0.6020 & 0.1488 & 0.0444 & 0.1071 & 0.2093 \\ 0.6020 & 2.6530 & 0.8083 & 0.6645 & 0.1019 & 0.3796 \\ 0.1488 & 0.8083 & 0.4583 & 0.0113 & -0.0132 & 0.1198 \\ 0.0444 & 0.6645 & 0.0113 & 1.4740 & 0.2522 & -0.0544 \\ 0.1071 & 0.1019 & 0.0132 & 0.2522 & 0.4880 & -0.0356 \\ 0.2093 & 0.3796 & 0.1198 & 0.0544 & -0.0356 & 0.3237 \end{pmatrix} \quad (16.26)$$

$$\mathcal{R} = \begin{pmatrix} 1.0000 & 0.6075 & 0.3613 & 0.0601 & 0.2520 & 0.6047 \\ 0.6075 & 1.0000 & 0.7331 & 0.3361 & 0.0895 & 0.4097 \\ 0.3613 & 0.7331 & 1.0000 & 0.0137 & -0.0280 & 0.3112 \\ -0.0601 & 0.3361 & 0.0137 & 1.0000 & 0.2974 & -0.0787 \\ 0.2520 & 0.0895 & -0.0280 & 0.2974 & 1.0000 & -0.0896 \\ 0.6047 & 0.4097 & 0.3112 & -0.0787 & -0.0896 & 1.0000 \end{pmatrix} \quad (16.27)$$

BARTLETT's sphere test shows that both matrices  $\mathcal{S}$  and  $\mathcal{R}$  are significantly different from the identity matrix  $\mathcal{I}$ :

## 16. Principle component analysis

Matrix	Det	$\chi^2$	f	$P_0$
$\mathcal{S}$	0.010	112.8	15	< 0.1 %
$\mathcal{R}$	0.094	57.8	15	< 0.1 %

Eigenvalues from  $\mathcal{S}$  and  $\mathcal{R}$  (see also fig. 16.12):

EV	Variance	Cum Var	EV	Variance	Cum Var
3.3450	0.5800	0.5800	2.5680	0.4280	0.4280
1.3770	0.2388	0.8188	1.3690	0.2281	0.6561
0.4764	0.0826	0.9014	0.9338	0.1556	0.8117
0.3252	0.0564	0.9578	0.6780	0.1130	0.9247
0.1550	0.0269	0.9846	0.3209	0.0535	0.9782
0.0886	0.0154	1.0000	0.1310	0.0218	1.0000

Total variance of  $\mathcal{S} = \sum_{j=1}^6 s_{jj} = \sum_{j=1}^6 \lambda_j = 5.77$ . The first two eigenvalues account for 81.8 % of variance. Total variance of  $\mathcal{R} = \sum_{j=1}^6 r_{jj} = \sum_{j=1}^6 \lambda_j = 6.00$ . However, the first two eigenvalues account for only 65.6 % of the variance. Relevant eigenvectors and loadings from  $\mathcal{S}$ :

	$e_1$	$e_2$	$l_1$	$l_2$	Commun	Uniq
WDIM	0.0783	-0.2200	0.6199	-0.2680	0.4561	0.5439
CIRCUM	0.7415	-0.5191	0.9819	-0.1560	0.9884	0.0116
FBEYE	0.2153	-0.2626	0.7072	-0.3979	0.6585	0.3415
EYEHLD	0.5601	0.7260	0.4852	0.8630	0.9801	0.0199
EARHD	0.2568	0.2667	0.1690	0.3739	0.1684	0.8316
JAW	0.1344	-0.1224	0.4115	-0.3825	0.3156	0.6844
var acc			2.2818	1.2854	3.5672	2.4329
prop var			0.3803	0.2142	0.5945	0.4055

Relevant eigenvectors and loadings from  $\mathcal{R}$ :

	$e_1$	$e_2$	$l_1$	$l_2$	Commun	Uniq
WDIM	0.0783	-0.2200	0.7429	0.0817	0.5586	0.4414
CIRCUM	0.7415	-0.5191	0.9634	0.2677	0.9998	0.0002
FBEYE	0.2153	-0.2626	0.7538	-0.0536	0.5711	0.4289
EYEHLD	0.5601	0.7260	0.3271	0.9018	0.9202	0.0798
EARHD	0.2568	0.2667	0.1659	0.6505	0.4507	0.5493
JAW	0.1344	-0.1224	0.5661	-0.2453	0.3807	0.6193
var acc			2.5032	1.3778	3.8811	2.1189
prop var			0.4172	0.2296	0.6469	0.3532

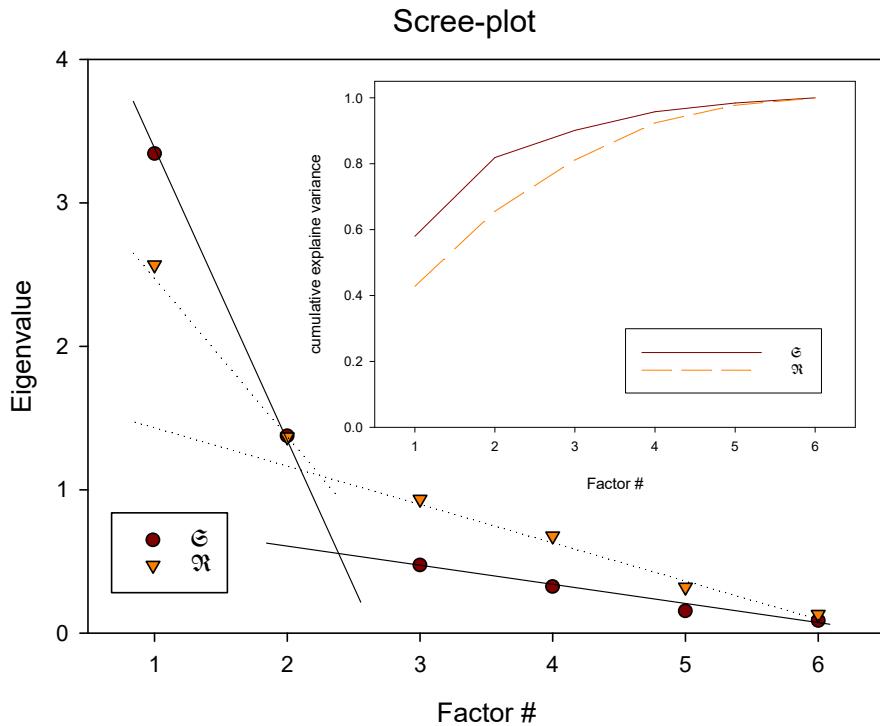


Figure 16.12.: Scree-plot of the football data.

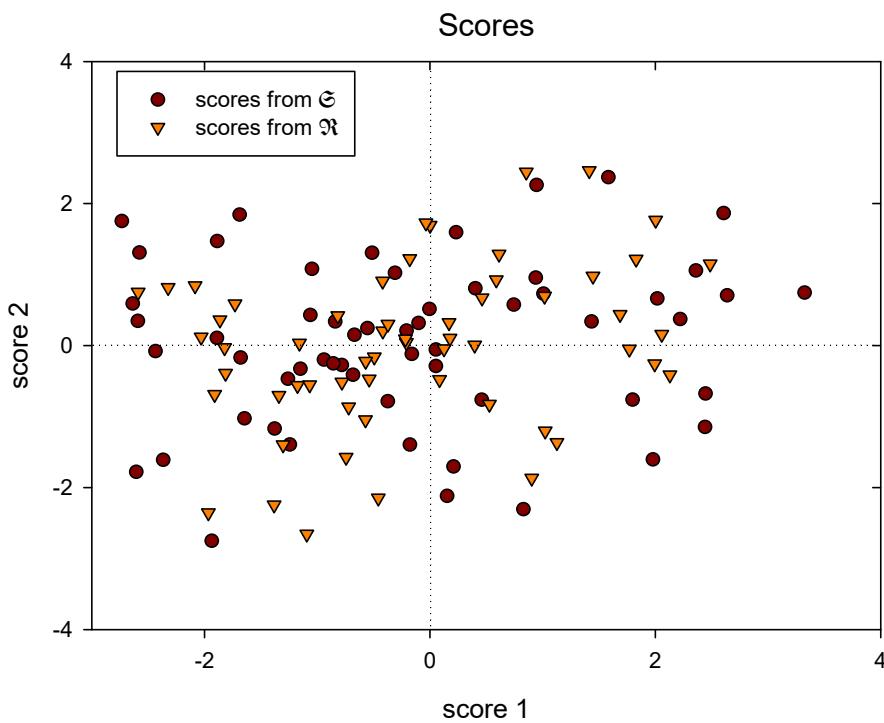


Figure 16.13.: The scores resulting from the football data

## 16. Principle component analysis

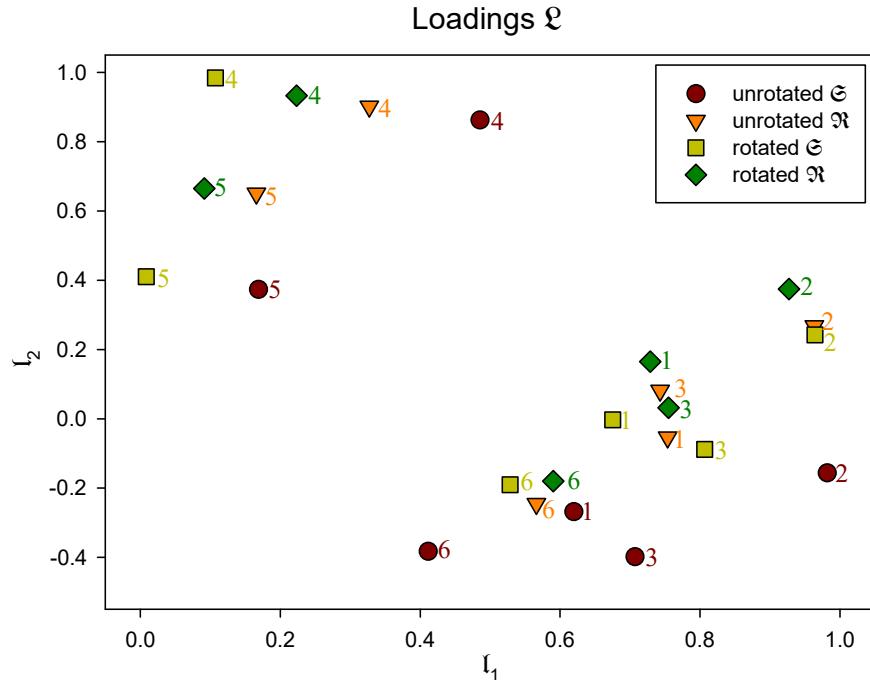


Figure 16.14.: Unrotated and rotated loadings from the football-data

Varimax-rotation of these loadings proceeds in 5 steps for  $\mathcal{S}$  and 12 steps for  $\mathcal{R}$ .

	S				R			
	$l_1$	$l_2$	Commun	Uniq	$l_1$	$l_2$	Commun	Uniq
WDIM	0.6753	-0.0030	0.4561	0.5439	0.7289	0.1650	0.5586	0.4414
CIRCUM	0.9643	0.2421	0.9884	0.0116	0.9271	0.3746	0.9998	0.0002
FBEYE	0.8067	-0.0882	0.6585	0.3415	0.7551	0.0318	0.5711	0.4289
EYEHD	0.1073	0.9842	0.9801	0.0199	0.2233	0.9329	0.9202	0.0798
EARHD	0.0086	0.4102	0.1684	0.8316	0.0915	0.6651	0.4507	0.5493
JAW	0.5286	-0.1901	0.3156	0.6844	0.5902	-0.1799	0.3807	0.6193
var acc	2.3277	1.2395	3.5672	2.4329	2.3676	1.5136	3.8811	2.1189
prop var	0.3880	0.2066	0.5945	0.4055	0.3946	0.2523	0.6469	0.3532

### 16.8.4. Interpretation

How many variables should be included when interpreting a factor? According to [55] the following rules are supported by Monte Carlo simulation:

- when at least 4 variables load on a factor with at least 0.60, sample size is of little consequence
- when at least 10 variables load on a factor with at least 0.40, sample size is of little consequence
- If these conditions are not met, sample size should be at least 300

- If sample size is less than 300, random loading patterns may become relevant. The study should be reproduced.

## References

- [1] K. HOPE: *Methoden multivariater Analyse* Weinheim, Basel: Beltz, 1975 ISBN: 3407510918.
- [2] B. WILLIAMS, A. ONSMAN, T. BROWN: Exploratory factor analysis: A five-step guide for novices, *Australas. J. Paramed.* **8**:3 (2010), 1–13 URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.414.4818&rep=rep1&type=pdf>.
- [3] E. KLOPP: *Explorative Faktorenanalyse* Web site, accessed 2017-07-02 2010 URL: <http://hdl.handle.net/20.500.11780/3369>.
- [4] G. FAES: *Faktorenanalyse* Web-site, accessed 2017-07-03 2016 URL: <http://www/faes.de/Basis/Basis-Lexikon/Basis-Lexikon-Multivariate/Basis-Lexikon-Faktorenanalyse/basis-lexikon-faktorenanalyse.html>.
- [5] C.M. FRIEL: *Notes on Factor Analysis* Web-site, accessed 2017-10-03 URL: <https://www.researchgate.net/file.PostFileLoader.html?id=566435736225ff665e8b4577&assetKey=AS%3A303652726607873%401449407859829>.
- [6] A.B. COSTELLO, J.W. OSBORNE: Best Practices in Exploratory Factor Analysis: Four Recommendations for Getting the Most From Your Analysis, *Practical Assessment, Research & Evaluation* **10**:7 (2005), 1–9 URL: <http://pareonline.net/pdf/v10n7.pdf>.
- [7] C. ZYGMONT, M.R. SMITH: Robust factor analysis in the presence of normality violations, missing data, and outliers: Empirical questions and possible solutions, *Quant. Meth. Psychol.* **10**:1 (2014), 40–55 DOI: [10.20982/tqmp.10.1.p40](https://doi.org/10.20982/tqmp.10.1.p40).
- [8] P. ROYSTON: Remark AS R94: A Remark on Algorithm AS 181: The W-test for Normality, *J. Royal Stat. Soc. C (Appl. Stat.)* **44**:4 (1995), 547–551.
- [9] A. MILLER: *Fortran Software* Web-site, accessed 2018-11-11 2004 URL: <https://jblevins.org/mirror/amiller/>.
- [10] N. HENZE, B. ZIRKLER: A class of invariant consistent tests for multivariate normality, *Commun. Stat. Theory Meth.* **19**:10 (1990), 3595–3617 DOI: [10.1080/03610929008830400](https://doi.org/10.1080/03610929008830400).
- [11] S. KORKMAZ, D. GOKSULUK, G. ZARARSIZ: MVN: An R package for assessing multivariate normality, *The R Journal* **6**:2 (2014), 151–162.
- [12] H.F. KAISER, J. RICE: Little Jiffy, Mark IV. *Educ. Psychol. Meas.* **34**:1 (1974), 11–117 DOI: [10.1177/001316447403400115](https://doi.org/10.1177/001316447403400115).
- [13] B.A. CERNY, H.F. KAISER: A Study Of A Measure Of Sampling Adequacy For Factor-Analytic Correlation Matrices, *Multivar. Behav. Res.* **12**:1 (1977), 43–47 DOI: [10.1207/s15327906mbr1201\\\_3](https://doi.org/10.1207/s15327906mbr1201\_3).

- [14] A.C. RENCHER: *Methods of Multivariate Analysis* 2nd ed. New York: John Wiley & Sons, 2002.
- [15] M.S. BARTLETT: The Effect of Standardization on a  $\chi^2$  Approximation in Factor Analysis, *Biometrika* **38**:3/4 (1951), 337–344 URL: <http://www.jstor.org/stable/2332580>.
- [16] H. HOTELLING: *Analysis of a complex of statistical variables into principal components* Baltimore: Warwick & York, 1933 URL: <https://babel.hathitrust.org/cgi/pt?id=wu.89097139406;view=1up;seq=5>.
- [17] R.J. HOFMANN: Complexity And Simplicity As Objective Indices Descriptive Of Factor Solutions, *Multivariate Behavioral Research* **13**:2 (1978), 247–250 DOI: [10.1207/s15327906mbr1302\\_9](https://doi.org/10.1207/s15327906mbr1302_9).
- [18] TTNPHNS: *Methods of computation of factor/component scores* Web-site, accessed 220-01-27 2014 URL: <https://stats.stackexchange.com/questions/126885/methods-to-compute-factor-scores-and-what-is-the-score-coefficient-matrix-in/126985#126985>.
- [19] G.H. THOMSON: The definition and measurement of g (general intelligence), *J. Educ. Psychol.* **26**:4 (1935), 241–262 DOI: [10.1037/h0059873](https://doi.org/10.1037/h0059873).
- [20] L. L. THURSTONE: *The vectors of mind: Multiple-factor analysis for the isolation of primary traits* Chicago, IL.: University of Chicago Press, 1935 URL: <https://archive.org/details/vectorsofmindmul010122mbp/page/n29/mode/2up>.
- [21] P. HORST: *Factor analysis of data matrices* New York, Chicago, San Francisco, Toronto, London: Holt, Rinehart and Winston, 1965 ISBN: 9780030502507.
- [22] M.A. LARKIN et al.: Clustal W and Clustal X version 2.0. *Bioinformatics* **23**:21 (2007), 2947–2948 DOI: [10.1093/bioinformatics/btm404](https://doi.org/10.1093/bioinformatics/btm404).
- [23] R.C. EDGAR: MUSCLE: multiple sequence alignment with high accuracy and high throughput, *Nucleic Acids Res.* **32**:5 (2004), 1792–1797 DOI: [10.1093/nar/gkh340](https://doi.org/10.1093/nar/gkh340).
- [24] S. BASU, D.P. BURMA, P. CHAUDHURI: Words in DNA sequences: some case studies based on their frequency statistics, *J. Math. Biol.* **46**:6 (2003), 479–503 DOI: [10.1007/s00285-002-0185-3](https://doi.org/10.1007/s00285-002-0185-3).
- [25] M. HACKENBERG et al.: WordCluster: detecting clusters of DNA words and genomic elements, *Algorithms Mol Biol.* **6**:2 (2011) DOI: [10.1186/1748-7188-6-2](https://doi.org/10.1186/1748-7188-6-2) URL: <https://almob.biomedcentral.com/track/pdf/10.1186/1748-7188-6-2.pdf>.
- [26] J. SCHÄFER, K. STRIMMER: A shrinkage approach to large-scale covariance matrix estimation and implications for functional genomics, *Stat. Appl. Genet. Mol. Biol.* **4**:1 (2005), Art. 32 DOI: [10.2202/1544-6115.1175](https://doi.org/10.2202/1544-6115.1175) URL: [https://s3.amazonaws.com/academia.edu.documents/35132267/sagmb.2005.4.1.1175.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1512737720&Signature=fntrNEmD5Ja2QVAnVAw1Kqi0h4%3D&response-content-disposition=inline%3B%20filename%3DA\\_Shrinkage\\_Approach\\_to\\_Large-Scale\\_Cova.pdf](https://s3.amazonaws.com/academia.edu.documents/35132267/sagmb.2005.4.1.1175.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1512737720&Signature=fntrNEmD5Ja2QVAnVAw1Kqi0h4%3D&response-content-disposition=inline%3B%20filename%3DA_Shrinkage_Approach_to_Large-Scale_Cova.pdf).

- [27] C.C.Y. KWAN: An Introduction to Shrinkage Estimation of the Covariance Matrix: A Pedagogic Illustration, *Spreadsheets Educ.* **4**:3 (2011), Art. 6 URL: <http://epublications.bond.edu.au/cgi/viewcontent.cgi?article=1099&context=ejsie>.
- [28] O. LEDOIT, M. WOLF: Honey, I Shrunk the Sample Covariance Matrix, *J. Portf. Manag.* **30**:4 (2003), 110–119 URL: <http://www.ledoit.net/honey.pdf>.
- [29] O. LEDOIT, M. WOLF: *Direct nonlinear shrinkage estimation of large-dimensional covariance matrices* University of Zurich Department of Economics Working Paper No. 264 2017 URL: [https://papers.ssrn.com/sol3/papers.cfm?abstract\\_id=3047302](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3047302).
- [30] D.J. DISATNIK, S. BENNINGA: Shrinking the Covariance Matrix – Simpler is Better, *J. Portf. Manag.* **33**:4 (2007), 56–63 URL: <https://cssanalytics.files.wordpress.com/2013/10/shrinkage-simpler-is-better.pdf>.
- [31] Team LATTE: *Cleaning a Correlation Matrix of Asset Returns - Spreadsheet Example* Web-site, accessed 2018-01-25 2007 URL: <https://risklatte.com/Articles/QuantitativeFinance/QF152.php>.
- [32] S. KOLENIKOV, G. ANGELES: *The Use of Discrete Data in PCA: Theory, Simulations, and Applications to Socioeconomic Indices* Web-site, accessed 2017-07-14 2004 URL: [https://www.measureevaluation.org/resources/publications/wp-04-85/at\\_download/document](https://www.measureevaluation.org/resources/publications/wp-04-85/at_download/document).
- [33] M. BASTO, J.M. PEREIRA: An SPSS R-Menu for Ordinal Factor Analysis, *J. Stat. Software* **46**:i04 (2012), 1–29 DOI: <10.18637/jss.v046.i04>.
- [34] R.B. CATTELL: The scree test for the number of factors, *Multivar. Behav. Res.* **1**:2 (1966), 245–276 DOI: [10.1207/s15327906mbr0102\\_10](10.1207/s15327906mbr0102_10).
- [35] W. REVELLE, T. ROCKLIN: Very Simple Structure: An Alternative Procedure For Estimating The Optimal Number Of Interpretable Factors, *Multivar. Behav. Res.* **14**:4 (1979), 403–414 DOI: [10.1207/s15327906mbr1404\\\_2](10.1207/s15327906mbr1404\_2).
- [36] W.F. VELICER: Determining the number of components from the matrix of partial correlations, *Psychometrika* **41**:3 (1976), 321–327 DOI: <10.1007/BF02293557> URL: [https://www.researchgate.net/profile/Wayne\\_Velicer/publication/24062214\\_Determining\\_the\\_Number\\_of\\_Components\\_from\\_the\\_Matrix\\_of\\_Partial\\_Correlations/\\_links/0a85e52f39b1ccbd3f000000.pdf](https://www.researchgate.net/profile/Wayne_Velicer/publication/24062214_Determining_the_Number_of_Components_from_the_Matrix_of_Partial_Correlations/_links/0a85e52f39b1ccbd3f000000.pdf).
- [37] M. GAVISH, D.L. DONOHO: The Optimal Hard Threshold for Singular Values is  $4/\sqrt{3}$ , *IEEE Trans. Inf. Theory* **60**:8 (2014), 5040–5053 DOI: <10.1109/TIT.2014.2323359>.
- [38] T.P. MINKA: *Automatic choice of dimensionality for PCA* MIT Media Laboratory Perceptual Computing Section Technical Report 514 Cambridge, MA, 2000 URL: <https://hd.media.mit.edu/tech-reports/TR-514.pdf>.
- [39] L.L. THURSTONE: *Multiple-factor analysis: A development and expansion of the vectors of mind* Chicago: University of Chicago Press, 1947 ISBN: 9780226801094.

- [40] K.M. KIEFFER: Orthogonal versus Oblique Factor Rotation: A Review of the Literature regarding the Pros and Cons, In: *Proceedings of the twenty-seventh annual meeting of the Mid-south Educational Research Association* New Orleans: ERIC ED427031, 1998 URL: <https://eric.ed.gov/?id=ED427031>.
- [41] H. ABDI: Factor Rotations in Factor Analyses, In: *Encyclopedia of Social Sciences Research Methods* M. LEWIS-BECK, A. BRYMAN, T. FUTING (editor) Thousand Oaks (CA): Sage, 2003, 792–795 ISBN: 9780761923633 URL: <http://www.utd.edu/~herve/Abdi-rotations-pretty.pdf>.
- [42] D.B. CLARKSON, R.I. JENNICH: Quartic rotation criteria and algorithms, *Psychometrika* **53**:2 (1988), 251–259 DOI: [10.1007/BF02294136](https://doi.org/10.1007/BF02294136).
- [43] H.F. KAISER: The varimax criterion for analytic rotation in factor analysis, *Psychometrika* **23**:3 (1958), 187–200 DOI: [10.1007/BF02289233](https://doi.org/10.1007/BF02289233) URL: <https://pdfs.semanticscholar.org/fb0b/812d9ba1a4b33f399fd51b56693e1935c745.pdf>.
- [44] H.F. KAISER: Computer Program for Varimax Rotation in Factor Analysis, *Educ. Psychol. Measurement* **19**:3 (1959), 413–420 DOI: [10.1177/001316445901900314](https://doi.org/10.1177/001316445901900314).
- [45] R.J. SHERIN: A matrix formulation of Kaiser's varimax criterion, *Psychometrika* **31**:4 (1966), 535–538 DOI: [10.1007/BF02289522](https://doi.org/10.1007/BF02289522).
- [46] K. NEVELS: A direct solution for pairwise rotations in Kaiser's varimax method, *Psychometrika* **51**:2 (1986), 327–329 DOI: [10.1007/BF02293988](https://doi.org/10.1007/BF02293988).
- [47] R.I. JENNICH: A simple general procedure for orthogonal rotation, *Psychometrika* **66**:2 (2001), 289–306 URL: <http://www.atmo.arizona.edu/students/courselinks/fall09/atmo529/Lectures/Jennrich2001.pdf>.
- [48] R.I. JENNICH: Rotation to simple loadings using component loss functions: The orthogonal case, *Psychometrika* **69**:2 (2004), 257–273 DOI: [10.1007/BF02295943](https://doi.org/10.1007/BF02295943) URL: <https://cloudfront.escholarship.org/dist/prd/content/qt0f84000d/qt0f84000d.pdf>.
- [49] C.A. BERNAARDS, R.I. JENNICH: Gradient Projection Algorithms and Software for Arbitrary Rotation Criteria in Factor Analysis, *Educ. Psychol. Meas.* **65**:5 (2005), 676–696 DOI: [10.1177/0013164404272507](https://doi.org/10.1177/0013164404272507).
- [50] C.A. BERNAARDS, R.I. JENNICH: *Gradient Projection Algorithms* Web-site, accessed 2018-02-03 2005 URL: <http://www.stat.ucla.edu/research/gpa/>.
- [51] A.L COMREY: Tandem criteria for analytic rotation in factor analysis, *Psychometrika* **32**:2 (1967), 143–154 DOI: [10.1007/BF02289422](https://doi.org/10.1007/BF02289422).
- [52] N.T. TRENDAFILOV, D. GRAGN: *Penalized varimax* Web-site, accessed 2018-01-31 URL: [http://statistics.open.ac.uk/802576CB00593013/\(httpInfoFiles\)/5FB28E2799052DE38025\\$file/Penalized\\_Varimax.pdf](http://statistics.open.ac.uk/802576CB00593013/(httpInfoFiles)/5FB28E2799052DE38025$file/Penalized_Varimax.pdf).
- [53] Crawford C.B., Ferguson G.A.: A General Rotation Criterion and its Use in Orthogonal Rotation, *Psychometrika* **35**:3 (1970), 321–332 DOI: [10.1007/BF02310792](https://doi.org/10.1007/BF02310792).

- [54] P.M. BENTLER: Factor simplicity index and transformations, *Psychometrika* **42**:2 (1977), 277–295 DOI: [10.1007/BF02294054](https://doi.org/10.1007/BF02294054).
- [55] E. GUADAGNOLI, W.F. VELICER: Relation to sample size to the stability of component patterns. *Psychol. Bull.* **103**:2 (1988), 265–275 URL: [https://www.researchgate.net/profile/Wayne\\_Velicer/publication/19793365\\_Relation\\_of\\_Sample\\_Size\\_to\\_the\\_Stability\\_of\\_Component\\_Patterns/links/09e4150b684d485e50000000.pdf](https://www.researchgate.net/profile/Wayne_Velicer/publication/19793365_Relation_of_Sample_Size_to_the_Stability_of_Component_Patterns/links/09e4150b684d485e50000000.pdf).



# 17. Cluster-analysis

Cluster analysis is described in [1], a very thorough yet readable introduction. We use the following definitions:

Listing 17.1: Program head

```
1 PROGRAM Similarity;
2
3 USES math,           // Lazarus math unit
4     MathFunc,        // general arithmetic
5     Vector,          // vector arithmetic
6     Matrix,          // matrix arithmetic
7     BigSet,          // sets of arbitrary cardinality
8     Correlations;   // correlation coefficients
9
10 CONST MaxVariables = 180;
11     MaxCases      = 1400;
12     SepChar       = ';' ;    // separates data IN csv-fles
13
14 TYPE DataTypes  = (binary, nominal, ordinal, interval, rational);
15 TypeVector = ARRAY[1..MaxVariables] OF DataTypes;
16
17 VAR Types       : TypeVector;
18     Data, Dist1, Dist2 : MatrixTyp;
19     Variables,
20     Cases,i,j : WORD;
21     r          : double;
22
23 TYPE ClusterFeld = ARRAY [1..MaxCases] OF SetType;      // declarations
24     should be local TO ClusterAnalysis
25
26 VAR HilfsMatrix : MatrixTyp;    { akt. Distanzen der Cluster oben, coph.
27     Matrix unten und Clusterzustand diagonal }
28     Cluster      : ClusterFeld;
29     v            : VectorTyp;
```

## 17.1. Read the data matrix

The data are stored in a comma separated value (csv) file according to [RFC 4180](#), which can be exported from Excel. However, in Middle Europe the decimal separator is comma

## 17. Cluster-analysis

rather than point (as in the US), so that the semicolon is used as variable separator. The routine needs to handle this:

Listing 17.2: Read data from comma-separated file

```
1 PROCEDURE ReadCSV(VAR Cases, Variables : WORD; VAR Types: TypeVector;
2                     VAR Data: MatrixTyp);
3
4 CONST MaxCases = 200;
5
6 VAR
7   InputFile: TEXT;
8   c: CHAR;
9   s: STRING;
10  Error, i, j: WORD;
11  Interim: MatrixTyp;
12  x: double;
13
14 BEGIN
15   Variables := 0;
16   Cases := 0;
17   Assign(InputFile, 'All.csv');
18   Reset(InputFile);
19   ReadLn(InputFile);           // ignore Line WITH variable names
20   WHILE NOT EoLn(InputFile) DO // Read data types from second Line, count
21     Variables
22   BEGIN
23     INC(Variables);
24     s := '';
25     REPEAT
26       Read(InputFile, c);
27       IF c <> SepChar THEN
28         S := S + c;
29     UNTIL ((c = SepChar) OR EoLn(InputFile));
30     CASE s OF
31       'binary' : Types[Variables] := binary;
32       'ordinal' : Types[Variables] := ordinal;
33       'nominal' : Types[Variables] := nominal;
34       'interval' : Types[Variables] := interval;
35       'rational' : Types[Variables] := rational;
36     ELSE
37       BEGIN          // Format error: abort PROGRAM WITH error message
38         Write('unknown data type ', s, ' at n=', Variables: 3, '. Press
39             <CR>:');
40         ReadLn;
41         HALT;
42     END; { else }
```

```

41    END; { case }
42    END; { while }
43    ReadLn(InputFile);           // go TO next Line
44    CreateMatrix(Interim, MaxCases, Variables, 0.0); // Note: number OF
        persons NOT yet known
45    WHILE NOT EoF(InputFile) DO // now Read data from following lines AND
        count cases
46    BEGIN
47        INC(Cases);
48        FOR i := 1 TO Variables DO
49            BEGIN
50                x := ReadFloat(InputFile);
51                IF MathError
52                    THEN
53                        BEGIN
54                            MathError := FALSE;
55                            HALT;
56                        END;
57                SetMatrixElement(Interim, Cases, i, x);
58            END;
59            ReadLn(InputFile);
60        END; { while }
61        Close(InputFile);
62        CreateMatrix(Data, Cases, Variables, 0.0); // now generate correctly
            sized data matrix
63        FOR i := 1 TO Cases DO
64            FOR j := 1 TO Variables DO
65                SetMatrixElement(Data, i, j, GetMatrixElement(Interim, i, j));
66            DestroyMatrix(Interim);                      // destroy the interim
                matrix
67            Writeln(Cases: 4, ' cases with ', Variables: 3, ' variables read, ');
68    END;

```

## 17.2. Calculate the similarity/distance matrix

A distance between two data points  $\mathbf{x}_{i\cdot}, \mathbf{x}_{j\cdot}$  is metric, if it meets the following conditions:

**non-negativity**  $D(\mathbf{x}_{i\cdot}, \mathbf{x}_{j\cdot}) \geq 0$

**isolation**  $D(\mathbf{x}_{i\cdot}, \mathbf{x}_{i\cdot}) = 0$

**symmetry**  $D(\mathbf{x}_{i\cdot}, \mathbf{x}_{j\cdot}) = D(\mathbf{x}_{j\cdot}, \mathbf{x}_{i\cdot})$

**triangular inequality**  $D(\mathbf{x}_{i\cdot}, \mathbf{x}_{j\cdot}) \geq D(\mathbf{x}_{i\cdot}, \mathbf{x}_{h\cdot}) + D(\mathbf{x}_{h\cdot}, \mathbf{x}_{j\cdot})$

, and analogously for similarities.

## 17. Cluster-analysis

Because we have data of mixed type we use the universal similarity coefficient of GOWER [2], see also [1, chapter 4.4], which for two **individuals**  $i, j$  is defined as

$$S_G = \frac{\sum_{k=1}^p w_{i,j,k} s_{i,j,k}}{\sum_{k=1}^p w_{i,j,k}} \quad (17.1)$$

over all variables  $k$ . The **weight**  $w_{i,j,k}$  is set to 1 if a comparison on character  $k$  can be made for the individuals  $i, j$ , and 0 if such a comparison cannot be made (*i.e.*, if at least one of the two data is missing). In that case, both denominator and numerator will increase by 0, the data pair will be ignored. Thus,  $S_G$  is robust with respect to missing data.  $s_{i,j,k}$  is calculated depending on data type:

**binary**  $s_{i,j,k} = 1$  if both data are 1, or 0 otherwise. If the data table consists of only binary variables, this would correspond to using JACCARD's coefficient (see page 358). It is possible to have  $s_{i,j,k} = 1$  also for negative matches, but this is rarely warranted and has negative statistical implications [3].

**nominal**  $s_{i,j,k} = 1$  if the two data match, 0 otherwise.

**ordinal** In the original publication  $s_{i,j,k} = 1$  for match, 0 for mismatch. However, [4] suggested to rank the data and then use the ranks like cardinal values. This yields more information for each comparison. Because we already use numbers to encode ordinal data in the data table, this is easily accomplished.

**cardinal**  $s_{i,j,k} = 1 - \frac{|X_{i,k} - X_{j,k}|}{R_k}$ , with  $R_k$  the range of variable  $k$  (either range in the data set or, if known, range in the wild). This is the complement to the mean character difference of [5].

For clustering, we need distances rather than similarities, which is simply the complement to 1.0.

Listing 17.3: Distance matrix

```

1  PROCEDURE Gowlers(CONST Types : TypeVector; CONST Data : MatrixTyp;
2                  VAR Dist : MatrixTyp);
3
4  var SumW, i, j, k, Cases, Variables : word;
5      S, SumWS, x, y           : double;
6      Maximum, Minimum, Range : array [2..MaxVariables] of double; // as
7          first column is case number
8
9  begin
10    Variables := MatrixColumns(Data);
11    Cases     := MatrixRows(Data);
12    for k := 2 to Variables do // calculate range of intervall and rational
13        data, ignore StudyNumber
14    begin

```

## 17.2. Calculate the similarity/distance matrix

```

13     Maximum[k] := -MaxRealNumber;
14     Minimum[k] := MaxRealNumber;
15     case Types[k] of
16         ordinal, interval, rational : begin
17             for i := 1 to Cases do
18                 begin
19                     x := GetMatrixElement(Data, i, k);
20                     if not isNaN(x)
21                         then
22                             begin
23                                 if (x > Maximum[k]) then Maximum[k] := x;
24                                 if (x < Minimum[k]) then Minimum[k] := x;
25                             end;
26                     end; { for i }
27                     Range[k] := Maximum[k] - Minimum[k];
28                 end
29             else Range[k] := 0.0; // binary, nominal: just give it a defined
30             value, won't be used
31         END; { case }
32     END; { for k }
33     Writeln('ranges calculated');
34     Write('Distances: ');
35     CreateIdentityMatrix(Dist, Cases);
36     FOR i := 1 TO Cases DO
37         BEGIN
38             FOR j := Succ(i) TO Cases DO // calculate upper half OF similarity
39                 matrix
40                 BEGIN
41                     SumW := 0;
42                     SumWS := 0;
43                     FOR k := 2 TO Variables DO // ignore CASE numbers
44                         BEGIN
45                             x := GetMatrixElement(Data, i, k);
46                             y := GetMatrixElement(Data, j, k);
47                             Write(OutFile, i:4, ';', j:4, ';', k:4, ';',
48 //                               FloatStr(x, 8), ',',
49 //                               FloatStr(y, 8), ',');
50                             IF (isNaN(x) OR isNaN(y))
51                                 THEN // SumW AND SumWS both increase by 0
52                             ELSE
53                                 CASE Types[k] OF
54                                     binary, nominal : BEGIN
55                                         IF x = y
56                                             THEN S := 0
57                                             ELSE S := 1;
58                                         INC(SumW);
59                                         SumWS := SumWS + S; // AS W = 1 ->
60                                     END;
61                                 END;
62                             END;
63                         END;
64                     END;
65                 END;
66             END;
67         END;
68     END;
69 
```

## 17. Cluster-analysis

```

      WS = S
      END;
      ordinal, interval, rational :
      BEGIN
      S := Abs(x - y) / Range[k];
      INC(SumW);
      SumWS := SumWS + S;
      END;
      END; { case }
      END; { for k }
      IF SumW = 0
      THEN x := NaN // so that such cases can be
      identified IN the distance matrix
      ELSE x := SumWS / SumW;
      SetMatrixElement(Dist, i, j, x);
      SetMatrixElement(Dist, j, i, SumW); // put number OF valid
      compares into lower half OF similarity matrix
      END; { for j }
      IF (i MOD 20 = 0) THEN Write('.');
      END; { for i }
      Writeln(' calculated, ');
      END; { Gowers }

      PROCEDURE WriteDistances (CONST Data, Dist : MatrixTyp);

      VAR i, j, cases : WORD;
          OutFile : TEXT;
          MaxS, MinS, MaxC, MinC, x : double;

      BEGIN
          Cases := MatrixRows(Dist);
          Assign(OutFile, 'Distances.csv');
          Rewrite(OutFile);
          MaxS := MinRealNumber;
          MaxC := MinRealNumber;
          MinS := MaxRealNumber;
          MinC := MaxRealNumber;
          Write(OutFile, 'StudyNum'); // LABEL columns
          FOR i := 1 TO Cases DO Write(OutFile, Round(GetMatrixElement(Data, i,
              1)):4, ',');
          Writeln(OutFile);
          FOR i := 1 TO Cases DO // the matrix itself
          BEGIN
              Write(OutFile, Round(GetMatrixElement(Data, i, 1)):4, ',');
              FOR j := 1 TO Cases DO
                  IF (j>i)

```

```

98      THEN
99      BEGIN
100     x := GetMatrixElement(Dist, i, j);
101     Write(OutFile, x:6:4, ',''); // upper half WITH similarity
102     IF IsNaN(x)
103         THEN
104         ELSE
105             BEGIN
106                 IF (x > MaxS) THEN MaxS := x;
107                 IF (x < MinS) THEN MinS := x;
108             END;
109         END
110     ELSE
111         BEGIN
112             x := GetMatrixElement(Dist, i, j);
113             Write(OutFile, Round(x):4, ',''); // lower half WITH # OF
114             comparisons
115             IF (j < i) // ignore i=j
116                 THEN
117                     BEGIN
118                         IF (x > MaxC) THEN MaxC := x;
119                         IF (x < MinC) THEN MinC := x;
120                     END;
121             Writeln(OutFile)
122         END; { for i }
123     Close(OutFile);
124     Writeln;
125     Writeln('Distances written to file, ');
126     Writeln('Range ', MinS:5:3, '-', MaxS:5:3, ' with ', MinC:3:0, '-',
127             MaxC:3:0, ' comparisons');
128 END; { WriteDistances }

129 PROCEDURE AnalyseFrequencies(CONST Dist: MatrixTyp);
130
131 CONST Border = 50; // number OF ranges FOR statistical analysis OF
132     correlations
133
134 TYPE FreqsType = ARRAY[-Border..Border] OF WORD;
135
136 VAR
137     i, j, Cases : WORD;
138     k : INTEGER;
139     OutFile : TEXT;
140     x : double;
141     Freqs : FreqsType;

```

## 17. Cluster-analysis

```
141
142 BEGIN
143   Assign(OutFile, 'DistFreq.csv');
144   Rewrite(OutFile);
145   Cases := MatrixRows(Dist);
146   FOR k := -Border TO Border DO
147     Freqs[k] := 0;
148   FOR i := 1 TO Cases DO
149     FOR j := Succ(i) TO Cases DO
150       BEGIN
151         x := GetMatrixElement(Dist, i, j) * Border;
152         INC(Freqs[Round(x)]);
153       END;
154     FOR k := -Border TO Border DO
155       Writeln(OutFile, k/Border:1:4, SepChar, Freqs[k]:6, SepChar);
156     Writeln(OutFile);
157     Close(OutFile);
158   END;
```

### 17.3. Hierarchic-agglomerative clustering

A cluster is a set of **operational taxonomic unit (OTU)**s that were combined at a certain similarity. Initially, each **OTU** forms a cluster by itself, then in each step the most similar clusters are combined and the process is continued until all **OTUs** have been combined in a single cluster. The result is a binary tree. The similarity of two clusters is calculated as the unweighted average of the similarities of all members of one cluster with all members of the other (average linkage, **unweighted pair group method with arithmetic mean (UPGMA)**). Alternatively, the distance between the most distant (complete linkage) or the closest **OTUs** () of the clusters may be used, however, clustering tends to be better with average linkage.

The routine returns the **cophenetic correlation coefficient**, which describes how well the tree preserves the pairwise distance of the **OTUs**.

Listing 17.4: clustering

```
1  FUNCTION ClusterAnalysis (CONST Dist : MatrixTyp) : double;
2
3  VAR i, j,
4    AnzCluster : WORD;
5    MinAbst    : double;
6    ClusterDatei : TEXT;
7
8  PROCEDURE Minimum (CONST HilfsMatrix : MatrixTyp; VAR MinAbst :
9                      double);
10
11  VAR i, j      : WORD;
```

```

11
12   BEGIN
13     FOR i := 1 TO Pred(Cases) DO
14       FOR j := Succ(i) TO Cases DO
15         IF GetMatrixElement(HilfsMatrix, i, j) < MinAbst
16           THEN MinAbst := GetMatrixElement(HilfsMatrix, i, j);
17   END;
18
19
20   PROCEDURE UniteCluster (VAR Cluster : ClusterFeld; i, j : WORD;
21                           MinAbst : double);
22
23   VAR k : WORD;
24
25   BEGIN
26     SetUnion(Cluster[i], Cluster[i], Cluster[j]);
27     ClearAllBits(Cluster[j]);
28     Writeln(ClusterDatei);
29     Writeln;
30     Write(ClusterDatei, MinAbst:6:4, ' ');
31     Write(MinAbst:6:4, ' ');
32     FOR k := 1 TO Cases DO
33       IF InSet(Cluster[i], k)
34         THEN
35           BEGIN
36             Write(ClusterDatei, Round(GetMatrixElement(Data, k, 1)):4,
37                   ' );
38             Write(Round(GetMatrixElement(Data, k, 1)):4, ' ');
39           END;
40           Writeln(ClusterDatei);
41           Writeln;
42   END;
43
44   PROCEDURE Cophen (VAR HilfsMatrix : MatrixTyp; CONST Cluster :
45                     ClusterFeld;
46                     MinAbst : double);
47
48   VAR i, j, l : WORD;
49
50   BEGIN
51     FOR i := 1 TO Cases DO
52       IF NOT EmptySet(Cluster[i])
53         THEN
54           FOR j := 1 TO Pred(Cases) DO
55             IF InSet(Cluster[i], j)

```

## 17. Cluster-analysis

```

54          THEN
55              FOR l := Succ(i) TO Cases DO
56                  IF InSet(Cluster[i], l)
57                      THEN
58                          IF (GetMatrixElement(HilfsMatrix, l, j) = 0)
59                              THEN SetMatrixElement(HilfsMatrix, l, j,
60                                  MinAbst);
61
62
63      PROCEDURE NewDistances (CONST Cluster : ClusterFeld; CONST Dist :
64                                MatrixTyp;
65
66          VAR i, j, k, l, b : WORD;
67          a                  : double;
68
69      BEGIN
70          FOR i := 1 TO Pred(Cases) DO
71              FOR j := Succ(i) TO Cases DO
72                  BEGIN
73                      IF ((NOT EmptySet(Cluster[i])) AND (NOT EmptySet(Cluster[j])))
74                          THEN
75                          BEGIN
76                              a := 0.0;
77                              b := 0;
78                              FOR k := 1 TO Cases DO
79                                  IF InSet(Cluster[i], k)
80                                      THEN
81                                          FOR l := 1 TO Cases DO
82                                              IF InSet(Cluster[j], l)
83                                              THEN
84                                              BEGIN
85                                                  a := a + GetMatrixElement(Dist, k, l);
86                                                  INC(b);
87                                              END;
88                                              SetMatrixElement(HilfsMatrix, i, j, a/b);
89                                          END
90                                      ELSE
91                                          SetMatrixElement(HilfsMatrix, i, j, MaxInt);
92                                  END; { for }
93      END; {NeueSimularitaeten}
94
95
96      FUNCTION NewDiagonal (CONST Cluster: ClusterFeld; VAR HilfsMatrix :
97                                MatrixTyp) : WORD;

```

```

97
98  VAR i, j : WORD;
99
100 BEGIN
101   j := 0;
102   FOR i := 1 TO Cases DO
103     IF NOT EmptySet(Cluster[i])
104       THEN
105         BEGIN
106           SetMatrixElement(HilfsMatrix, i, i, 1);
107           INC(j);
108         END
109       ELSE
110         SetMatrixElement(HilfsMatrix, i, i, 0);
111       NewDiagonal := j;
112     END;
113
114
115 FUNCTION Correlation (CONST Dist, HilfsMatrix : MatrixTyp) : double;
116
117 VAR DistMittel, CoMittel,
118   DistKor, CoKor,
119   SumDistKorSqr, SumCoKorSqr,
120   SumDistKorCoKor          : double;
121
122
123 PROCEDURE Mittelwerte (CONST Dist, HilfsMatrix : MatrixTyp);
124
125 VAR SumDist, SumCo : double;
126   i, j, z          : WORD;
127
128 BEGIN
129   SumDist := 0;
130   SumCo := 0;
131   z := 0;
132   FOR i := 1 TO Pred(Cases) DO
133     FOR j := Succ(i) TO Cases DO
134       BEGIN
135         SumDist := SumDist + GetMatrixElement(Dist, i, j);
136         SumCo := SumCo + GetMatrixElement(HilfsMatrix, j, i);
137         INC(z);
138       END;
139   DistMittel := SumDist / z;
140   CoMittel := SumCo / z;
141 END;
142

```

## 17. Cluster-analysis

```

143
144     FUNCTION Summen (CONST Dist, HilfsMatrix : MatrixTyp) : double;
145
146     VAR i, j : WORD;
147
148     BEGIN
149         SumDistKorSqr := 0;
150         SumCoKorSqr := 0;
151         SumDistKorCoKor := 0;
152         FOR i := 1 TO Pred(Cases) DO
153             FOR j := Succ(i) TO Cases DO
154                 BEGIN
155                     DistKor := GetMatrixElement(Dist, i, j) - DistMittel;
156                     CoKor := GetMatrixElement(HilfsMatrix, j, i) - CoMittel;
157                     SumDistKorSqr := SumDistKorSqr + Sqr(DistKor);
158                     SumCoKorSqr := SumCoKorSqr + Sqr(CoKor);
159                     SumDistKorCoKor := SumDistKorCoKor + DistKor * CoKor;
160                 END;
161                 Writeln(SumDistKorCoKor:4:1, ' ', SumDistKorSqr:4:1, ' ',
162                         SumCoKorSqr:4:1);
163                 Summen := SumDistKorCoKor / (Sqrt(SumDistKorSqr) *
164                                         Sqrt(SumCoKorSqr));
165             END;
166
167             BEGIN
168                 Mittelwerte(Dist, HilfsMatrix);
169                 Correlation := Summen(Dist, HilfsMatrix);
170             END;
171
172             BEGIN
173                 ClusterAnalysis := 0.0;
174                 Assign(ClusterDatei, 'Similarities.CLU');
175                 Rewrite(ClusterDatei);
176                 CopyMatrix(Dist, HilfsMatrix);
177                 FOR i := 1 TO Cases DO          // jedem OTU sein eigenes Cluster
178                     BEGIN
179                         ClearAllBits(Cluster[i]);
180                         SetBit(Cluster[i], i);
181                     END;
182                     REPEAT                                { Beginn der
183                         Analysenschleife }
184                         MinAbst := MaxRealNumber;
185                         Minimum(HilfsMatrix, MinAbst);
186                         FOR i := 1 TO Pred(Cases) DO          { neue Cluster bilden }
187                             FOR j := Succ(i) TO Cases DO
188                                 IF (GetMatrixElement(HilfsMatrix, i, j) = MinAbst)

```

```

186         THEN UniteCluster(Cluster, i, j, MinAbst);
187     Cophen(HilfsMatrix, Cluster, MinAbst);
188     NewDistances(Cluster, Dist, HilfsMatrix);
189     AnzCluster := NewDiagonal(Cluster, HilfsMatrix);
190 UNTIL AnzCluster = 1;
191 Close(ClusterDatei);
192 Result := Correlation(Dist, Hilfsmatrix);
193 END;

```

Listing 17.5: compare distance matrices

```

1 FUNCTION MatrixCorrelation (CONST Dist1, Dist2 : MatrixTyp) : float;
2
3 VAR i, j, k : WORD;
4     SumXY, SumX, SumY, SumX2, SumY2, varX, varY, covXY, x, y, r : float;
5     OutFile : TEXT;
6
7 BEGIN
8     Assign(OutFile, 'DistCorr.csv');
9     Rewrite(OutFile);
10    Cases := MatrixRows(Dist1);
11    SumXY := 0;
12    SumX := 0;
13    SumY := 0;
14    SumX2 := 0;
15    SumY2 := 0;
16    k := 0;
17    FOR i := 1 TO Cases DO
18        FOR j := Succ(i) TO Cases DO
19            BEGIN
20                x := GetMatrixElement(Dist1, i, j);
21                y := GetMatrixElement(Dist2, i, j);
22                Writeln(OutFile, i:3, ';', j:3, ';', FloatStr(x, 10), ';',
23                         FloatStr(y, 10), ';');
24                SumXY := SumXY + x * y;
25                SumX := SumX + x;
26                SumY := SumY + y;
27                SumX2 := SumX2 + x * x;
28                SumY2 := SumY2 + y * y;
29                INC(k); // actual number OF valid comparisons
30            END; { for }
31    varX := k * SumX2 - SumX * SumX;
32    varY := k * SumY2 - SumY * SumY;
33    covXY := k * SumXY - SumX * SumY;
34    IF Abs(varX * varY) < Zero
35        THEN Result := 0      // ???
            ELSE Result := covXY / Sqrt(varX * varY);

```

## 17. Cluster-analysis

```

36   Writeln(OutFile, FloatStr(Result, 10));
37   Close(OutFile);
38 END; { MatrixCorrelation }
```

The main part of the program is then quite short:

Listing 17.6: Main program

```

1 BEGIN
2   ReadCSV (Cases, Variables, Types, Data);
3   Gowlers(Types, Data, Dist1);
4   CalcCaseCorrelations(Data, Dist2);
5   WriteDistances(Data, Dist2); // Data IS source OF study-number
6   AnalyseFrequencies(Dist2);
7   FOR i := 1 TO Cases DO
8     FOR j := Succ(i) TO Cases DO
9       BEGIN
10      SetMatrixElement(Dist1, j, i, GetMatrixElement(Dist1, i, j));
11      // symmetrieren
12      SetMatrixElement(Dist2, j, i, GetMatrixElement(Dist2, i, j));
13    END;
14    LeadingPrincipleMinors(Dist2, V);
15    FOR i := 1 TO Cases DO
16      Writeln(FloatStr(GetVectorElement(V, i), 10));
17      DestroyVector(V);
18      r := MatrixCorrelation (Dist1, Dist2);
19      r := ClusterAnalysis(Dist2);
20      Write('Cophenetic correlation', r:1:3, ' Press <CR> to finish:');
21      ReadLn;
22  END.
```

## 17.4. *k*-means clustering

The data (which need to be cardinal) are grouped into  $k$  clusters  $\mathcal{C}_i$ , where  $i \in [1..k], k \ll n$  is a number that must be chosen at the begin of the study. The algorithm is as follows [6, 7]:

1. Randomly select  $k$  OTUs as centres of the clusters, and assign each OTU to the closest centre (by squared EUKLIDIAN distance).
2. Repeat
  - a) Calculate the centroids of all clusters, that is, the vector of the  $p$  feature means for the observations in the  $i$ th cluster.
  - b) Assign each OTU to the cluster whose centroid is closest until OTUs are no longer assigned to different centres.

This partitions the data space into VORONOI cells. Should a cluster become empty, a new centre needs to be chosen. In effect, the between-cluster distances are maximised, the within-cluster distance minimised.

Finding optimal partitioning (minimal sum of squared distances from the centres  $\sum_{i=1}^k \sum_{\mathbf{x} \in C_i} \text{dist}(\mathbf{c}_i, \mathbf{x})^2$ ) is NP-hard, the result of above heuristic depends somewhat on the starting assignments, the program needs to be run several times. Finally, the best solution is chosen. The algorithm is  $\mathbf{O}(npkm)$ , where  $m$  is the number of iterations required. Local optima can be left by swapping OTUs between clusters. Missing data can be handled by imputation of a probable range [8].

### 17.4.1. Modifications

***k*-means++ algorithm** selects one OTU randomly as starting point. Then the distances of all OTUs to this starting point are calculated, and the next centre is determined so that the probability of an OTU to become the centre depends on the square of the distance to the original starting point. This is repeated until all  $k$  centres have been selected. Then perform the *k*-means algorithm. This method converges twice as fast as the simple *k*-means algorithm, with similar end results.

***k*-median algorithm** uses medians instead of means and Manhatten-distances  $\ell_1$  (absolute value of difference) instead of EUKLIDIAN  $\ell_2$  (squared distance, see section 5.3.5 on page 132).

**Partitioning around medoids (PAM) algorithm** tries to minimise the distance of the data points from the medoid instead of the centroid. While the centroid as mean of the data is synthetic, the medoid is a member of the data set that represents its group [9]. Since the calculation of means is sensitive to outliers, PAM is more robust than *k*-means clustering.  $k$  data vectors are randomly assigned as medoids, other data points are assigned to their closest medoid. Then data and medoids are exchanged to minimise an objective (= distance) function.

***k*-modes clustering** can handle categorical data. The distance of an OTU to the centre of a cluster is given by the sum over all  $p$  variables of 1 for mismatches and  $1 - \frac{n_j^r}{n_j}$  for matches, that is, matches for rare categories are weighted heavier than matches of frequent ones. Mixed categorical/cardinal data can be handled with universal distance functions like GOWER's.

**kernel *k*-means** is used for data that are not linearly separable. Kernel methods try to classify cases by transforming the data into higher dimensions.

**expectation–maximization algorithm** is a generalisation of *k*-means clustering that works even if the clusters have very different sizes, densities or non-spherical shape. However, it is much more complex.

### 17.4.2. Selection of optimal $k$

#### Elbow-method

For all  $k$ , calculate the within-cluster sum of square (wss) and plot against  $k$ . The optimal  $k$  is the position of the elbow in the plot (similar to scree-plot in PCA, see section 16.7.3 on page 552).

#### Average silhouette method

A silhouette  $S(\mathbf{x})$  is a measure how close an observation  $\mathbf{x}_i$  is to the nearest two clusters, and thus a measure of the quality of clustering. The average distance between an observation and a cluster  $\mathcal{C}_i$  is defined as the arithmetic average of the distances of  $\mathbf{x}_i$  to all (other) members of  $\mathcal{C}_i$ . Then

$$S(\mathbf{x}) = \begin{cases} 0 & \text{if } \mathbf{x}_i \text{ is the only element of } \mathcal{C}_i \\ \frac{\text{dist}(\mathbf{x}_i, \mathcal{C}_i) - \text{dist}(\mathbf{x}_i, \mathcal{C}_j)}{\max(\text{dist}(\mathbf{x}_i, \mathcal{C}_i), \text{dist}(\mathbf{x}_i, \mathcal{C}_j))} & \text{else} \end{cases} \quad (17.2)$$

$s(\mathcal{C}_i)$  is the arithmetic mean of all silhouettes of the cluster  $\mathcal{C}_i$ . If it is close to +1, the data point is in the appropriate cluster, if it is -1, it should be a member of the neighbouring cluster. A value near zero indicates that  $\mathbf{x}_i$  is close to the border between clusters. The mean of all  $s(\mathcal{C}_i)$  of  $\mathcal{C}_i$  indicates how tightly grouped all the points in the cluster are. The mean  $s(\mathcal{C}_i)$  over all data is a measure of how well the data have been clustered. The silhouette coefficient is the largest  $s(\mathcal{C}_i)$  over all data in the entire data set. This can be used as criterium for optimising  $k$ .

The cluster (of which  $\mathbf{x}_i$  is not a member) with minimal  $S(\mathbf{x})$  is said to be the neighbouring cluster.

The silhouette plot plots a line of length  $S(\mathbf{x})$  for all elements of the cluster, in descending order.

#### Gap statistics

Here we compare the within-cluster sum of square with that obtained on a synthetic data set with random data, which have the same range ( $\min(\mathbf{x}_j), \max(\mathbf{x}_j)$ ) as the original data set (bootstrapping). Then for  $B$  artificial data sets, calculate average and standard deviation of the logarithms of wss and plot  $\text{Gap}(k) = \overline{\log(\text{wss})_B} - \log(\text{wss}_k)$  as function of  $k$ . Then  $s_k = \text{sd}_k \times \sqrt{1 + 1/B}$  and we look for the smallest  $k$  where  $\text{Gap}(k) \geq \text{Gap}(k+1) - s_{k+1}$ .

### 17.4.3. Sourcecode for $k$ -means clustering with $k$ -fold crossvalidation

#### $k$ -means clustering

Listing 17.7: unit kNN: *k*-means clustering

```

1  UNIT kNN;
2
3  INTERFACE
4
5  USES Math,
6      MathFunc,
7      Vector,
8      Matrix,
9      Deskript,
10     Zufall,
11     CrossValidation;
12
13 CONST
14   kNNError: BOOLEAN = FALSE;
15   KNNmax = 15;
16   pMax = 100;
17
18 TYPE
19   GroupTyp = ARRAY[1..MaxVector] OF WORD;
20
21 FUNCTION kMeansCluster(CONST Data: MatrixTyp;           // data matrix
22   k: WORD;                                              // number OF clusters
23   VAR Centroids: MatrixTyp;                            // centroids FOR all k
24   VAR clusters AND p variables
25   VAR Group: GroupTyp): float;                         // group no FOR each
26   item
27 { splits a data matrix into k groups by the k-nearest neighbour method and
28   returns the sum of Euklidian distances of the data from their centroids. }
29
30 PROCEDURE AssignTestData(CONST TestData, Centroids: MatrixTyp;
31   kValidate : WORD;                                     // number OF
32   cross-validation groups
33   VAR Count : WORD;                                    // no OF
34   results calculated already
35   VAR ValidationResult: MatrixTyp);                  // Result
36 { assigns all rows of Data to the closest Centroid and
37   returns the study number, assigned group and squared distance to the
38   centroid
39   in ValidationResult }

40 FUNCTION WithinSumOfSquares(CONST ValidationResult: MatrixTyp): float;
41 { calculates the sum of the squared distance of all data from their
42   centroid,
43   which is stored in the third column of ValidationResult }
44

```

## 17. Cluster-analysis

```
40  IMPLEMENTATION
41
42  PROCEDURE StartingCentroids(CONST Data: MatrixTyp; k: WORD; VAR Centroids:
43      MatrixTyp);
44  // select the starting centroids using knn++ METHOD (distance-controlled
45  // Random)
46
47  VAR
48      v, vi, row, Distance : VectorTyp;
49      i, j, l, n, p         : WORD;
50      r                   : LONGINT;
51      Used                : ARRAY[1..KNNmax] OF WORD;
52      u                   : BOOLEAN;
53      c                   : CHAR;
54      d, Sum               : float;
55
56  BEGIN
57      n := MatrixRows(Data);
58      p := MatrixColumns(Data);
59      CreateMatrix(Centroids, k, p, 0.0);
60      i := RandomLaplace(1, n);           // randomly select first centroid
61      Writeln('first centroid = ', i: 3);
62      GetRow(Data, i, v);
63      SetRow(Centroids, v, 1);
64      Used[1] := i;
65      FOR j := 2 TO k DO                  // select the remaining
66          centroids
67          BEGIN
68              CreateVector(Distance, n, 0.0);
69              IF VectorError
70                  THEN
71                      BEGIN
72                          c := WriteErrorMessage('program terminated');
73                          HALT;
74                      END;
75              Sum := 0;
76              FOR i := 1 TO n DO // calculate distance between all items AND last
77                  centroid
78                  BEGIN
79                      GetRow(Data, i, vi);
80                      d := SquaredEuklidianDistance(v, vi, TRUE);
81                      DestroyVector(vi);
82                      SetVectorElement(Distance, i, d);
83                      FOR l := 1 TO Pred(j) DO          // ignore IF datum i IS already
84                          a centroid
85                      IF Used[l] = i THEN u := TRUE;
```

```

81         IF NOT (u) THEN Sum := Sum + d;
82     END;
83     l := ceil(Sum);
84     r := RandomLaplace(1, l); // distance-controlled Random selection OF
85     // NEW centroid
86     Sum := 0;
87     i := 0;
88     REPEAT
89         INC(i);
90         u := FALSE;
91         FOR l := 1 TO Pred(j) DO
92             IF Used[l] = i THEN u := TRUE;
93             IF NOT (u) THEN Sum := sum + Round(GetVectorElement(Distance, i));
94         UNTIL (Sum >= r);
95         GetRow(Data, i, vi);
96         SetRow(Centroids, vi, j);
97         FOR i := 1 TO p DO
98             SetVectorElement(v, i, GetVectorElement(vi, i));
99             DestroyVector(vi);
100            DestroyVector(Distance);
101        END; // FOR j
102        DestroyVector(v);
103    END; // StartingCentroids

104 FUNCTION AssignItems(CONST Data, Centroids: MatrixTyp;
105   k: WORD;
106   VAR Group: GroupTyp): float;
107   // Assign each datum TO the cluster from the centre OF which it has
108   // minimal
109   // distance. Returns sum OF the minimal distances over all data

110 VAR
111   i, j, l, n, p : WORD;
112   d, min, Sum   : float;
113   vj, vi         : VectorTyp;

114 BEGIN
115   n := MatrixRows(Data);
116   p := MatrixColumns(Data);
117   Sum := 0;
118   FOR i := 1 TO n DO           // FOR all data rows
119     BEGIN
120       min := MaxRealNumber;
121       GetRow(Data, i, vi);
122       FOR j := 1 TO k DO        // find centroid WITH minimal distance TO
123         the datum

```

## 17. Cluster-analysis

```
124      BEGIN
125          GetRow(Centroids, j, vj);
126          d := SquaredEuklidianDistance(vj, vi, TRUE);
127          IF (d < min)
128          THEN
129              BEGIN
130                  min := d;
131                  l := j;
132              END; // THEN
133              DestroyVector(vj);
134          END; // FOR j
135          Group[i] := l;
136          Sum := Sum + min;
137          DestroyVector(vi);
138      END; // FOR i
139      Result := Sum;
140  END; // AssignItems

141
142
143 PROCEDURE CalculateNewCentroids(CONST Data: MatrixTyp;
144                                     k: WORD;
145                                     VAR Group: GroupTyp;
146                                     VAR Centroids: MatrixTyp
147                                     );
148
149 VAR
150     i, j, l, n, p, nk : WORD;
151     Sums               : ARRAY [1..pMax] OF float;
152     x                  : float;
153
154 BEGIN
155     n := MatrixRows(Data);
156     p := MatrixColumns(Data);
157     FOR l := 1 TO k DO
158         BEGIN
159             FOR j := 1 TO p DO
160                 Sums[j] := 0.0;
161             nk := 0;
162             FOR i := 1 TO n DO
163                 BEGIN
164                     IF Group[i] = l
165                     THEN
166                         BEGIN
167                             FOR j := 1 TO p DO
168                             BEGIN
169                                 x := GetMatrixElement(Data, i, j);
```

```

170           Sums[j] := Sums[j] + x;
171       END;
172   END;
173   INC(nk);
174 END;
175 FOR j := 1 TO p DO
176   IF (nk = 0)
177     THEN
178       SetMatrixElement(Centroids, k, j, 0) // no data IN group,
179         shouldn't happen
180     ELSE
181       SetMatrixElement(Centroids, k, j, Sums[j] / nk);
182   END; // for l
183 END; // CalculateNewCentroids
184
185 FUNCTION kMeansCluster(CONST Data: MatrixTyp;
186   k: WORD;
187   VAR Centroids: MatrixTyp;
188   VAR Group: GroupTyp): float;
189
190 VAR
191   p, n, Iter : WORD;
192   TotalDist : float;
193
194 BEGIN // kMeansCluster
195   n := MatrixRows(Data);
196   p := MatrixColumns(Data);
197   StartingCentroids(Data, k, Centroids);
198   TotalDist := MaxRealNumber;
199   Iter := 0;
200   REPEAT
201     OldDist := TotalDist;
202     Inc(Iter);
203     TotalDist := AssignItems(Data, Centroids, k, Group);
204     CalculateNewCentroids(Data, k, Group, Centroids);
205   UNTIL (abs(OldDist - TotalDist) < MaxError) OR (Iter > MaxIter);
206   Result := TotalDist;
207 END; // kMeansCluster
208
209 PROCEDURE AssignTestData(CONST TestData, Centroids: MatrixTyp;
210   kValidate : WORD;
211   VAR Count : WORD;
212   VAR ValidationResult: MatrixTyp);
213
214 VAR

```

## 17. Cluster-analysis

```
215     i, j, k, n, p, Opt : WORD;
216     vi, vj             : VectorTyp;
217     d, MinD            : float;
218
219 BEGIN
220     n := MatrixRows(TestData);
221     p := MatrixColumns(TestData);
222     k := MatrixRows(Centroids);
223     CreateVector(vi, p, 0.0);
224     FOR i := 1 TO n DO          // for all data in test data matrix
225         BEGIN
226             GetRow(TestData, i, vi);
227             MinD := MaxRealNumber;
228             Opt := 0;
229             FOR j := 1 TO k DO  // find centroid of minimal distance
230                 BEGIN
231                     GetRow(Centroids, j, vj);
232                     d := SquaredEuklidianDistance(vi, vj, True);
233                     IF d < MinD
234                         THEN
235                             BEGIN
236                                 MinD := d;
237                                 Opt := j;
238                             END; // then
239                             DestroyVector(vj);
240                         END; // for j
241                         SetMatrixElement(ValidationResult, Count, 1,
242                                         GetMatrixElement(TestData, i, 1));
243                         // study number
244                         SetMatrixElement(ValidationResult, Count, 2, Opt); // optimal
245                                         centroid
246                         SetMatrixElement(ValidationResult, Count, 3, MinD); // distance from
247                                         centroid
248                         DestroyVector(vi);
249                         INC(Count);
250                     END; // for i
251     END; // AssignTestData
252
253 FUNCTION WithinSumOfSquares(CONST ValidationResult: MatrixTyp): float;
254
255 VAR
256     i, n: WORD;
257     WSS: float;
258
259 BEGIN
```

```

258 n := MatrixRows(ValidationResult);
259 WSS := 0;
260 FOR i := 1 TO n DO
261   WSS := WSS + GetMatrixElement(ValidationResult, i, 3);
262   Result := WSS;
263 END; // WithinSumOfSquares
264
265 end. //kNN

```

## *k*-fold crossvalidation

Listing 17.8: *k*-fold crossvalidation

```

1 UNIT CrossValidation;
2
3 INTERFACE
4
5 USES MathFunc, Vector, Matrix, Zufall;
6
7 CONST
8   MaxK = 15;
9   CrossValidationError: BOOLEAN = FALSE;
10
11 TYPE
12   SplitDataTyp = ARRAY [1..MaxK] OF MatrixTyp;
13
14
15 PROCEDURE SplitDataMatrix(CONST Data: MatrixTyp;           // data matrix
16   kValidate: WORD;                                // no OF validation
17   groups
18   VAR SplitData: SplitDataTyp);                  // randomly distributed
19   data
20 { randomly splits the data matrix into kValidate sub-matrices }
21
22 PROCEDURE CreateTestData(CONST SplitData: SplitDataTyp;
23   k, kValidate: WORD;
24   VAR TestData: MatrixTyp);
25 { combine all Groups except the out of box group k into test matrix }
26
27
28 IMPLEMENTATION
29
30 PROCEDURE SplitDataMatrix(CONST Data: MatrixTyp; kValidate: WORD;
31   VAR SplitData: SplitDataTyp);
32
33 VAR

```

## 17. Cluster-analysis

```
32  h, i, j, n, p : WORD;
33  c : CHAR;
34  Available : ARRAY [1..MaxK] OF WORD;
35  CurrentRow : VectorTyp;
36
37 BEGIN
38   IF (kValidate > MaxK)
39   THEN
40    BEGIN
41      CrossValidationError := TRUE;
42      c := WriteErrorMessage('k-fold cross-validation: kValidate >
43          maximum');
44      EXIT;
45    END;
46   n := MatrixRows(Data);
47   p := MatrixColumns(Data);
48   j := n DIV kValidate; // number OF elements OF all submatrices except
49          last
50   FOR i := 1 TO Pred(kValidate) DO
51   BEGIN
52     CreateMatrix(SplitData[i], j, p, 0.0);
53     IF MatrixError
54     THEN
55      BEGIN
56        CrossValidationError := TRUE;
57        MatrixError := FALSE;
58        c := WriteErrorMessage('k-fold cross-validation: not enough
59          memory');
60        EXIT;
61      END;
62     Available[i] := j;
63   END;
64   CreateMatrix(SplitData[kValidate], j + (n MOD kValidate), p, 0.0);
65   // put left-overs into last group
66   IF MatrixError
67   THEN
68    BEGIN
69      CrossValidationError := TRUE;
70      MatrixError := FALSE;
71      c := WriteErrorMessage('k-fold cross-validation: not enough
72          memory');
73    EXIT;
74  END;
75  Available[kValidate] := j + (n MOD kValidate);
76  FOR i := 1 TO n DO // randomly put each data row into one OF the
77          kValidate submatrices
```

```

73   BEGIN
74     REPEAT
75       j := Round(RandomLaplace(1, kValidate)); // select group
76     UNTIL (Available[j] > 0);
77     GetRow(Data, i, CurrentRow);
78     h := Succ(MatrixRows(SplitData[j]) - Available[j]);
79     SetRow(SplitData[j], CurrentRow, h);
80     DEC(Available[j]);
81     DestroyVector(CurrentRow);
82   END;
83 END; { SplitDataMatrix }

84
85 PROCEDURE CreateTestData(CONST SplitData: SplitDataTyp; k, kValidate: WORD;
86                           VAR TestData: MatrixTyp);

87
88 VAR
89   j, n, p, Sum : WORD;
90   i             : 1..MaxK;
91   v             : VectorTyp;
92   c             : CHAR;

93
94 BEGIN
95   Sum := 0;
96   FOR i := 1 TO kValidate DO // calculate number OF rows OF TEST data
97     IF i <> k
98     THEN
99       BEGIN
100      n := MatrixRows(SplitData[i]);
101      Sum := Sum + n;
102    END;
103   p := MatrixColumns(SplitData[1]);
104   CreateMatrix(TestData, Sum, p, 0.0);
105   IF MatrixError
106   THEN
107     BEGIN
108       CrossValidationError := TRUE;
109       MatrixError := FALSE;
110       c := WriteErrorMessage('k-fold cross-validation: not enough
111                               memory');
112       EXIT;
113     END;
114   Sum := 0;
115   FOR i := 1 TO kValidate DO
116     IF i <> k
117     THEN
           BEGIN

```

## 17. Cluster-analysis

```
118      FOR j := 1 TO MatrixRows(SplitData[i]) DO
119      BEGIN
120          INC(Sum);
121          GetRow(SplitData[i], j, v);
122          SetRow(TestData, v, Sum);
123          DestroyVector(v);
124      END;
125  END;
126 { CreateTestData }
127
128 END.
```

### Main program

Listing 17.9: Main program

```
1 PROGRAM kNNTest;
2
3 USES
4     Math,           // free pascal standard math UNIT
5     MathFunc,       // basic math routines
6     Vector,         // vector algebra
7     Matrix,         // matrix algebra
8     Zufall,         // Random numbers
9     Deskript,       // descriptive statistics
10    kNN,            // k means clustering
11    CrossValidation // k-fold cross-validation
12    ;
13
14 CONST
15    n = 112;        // data sets
16    p = 23;         // variables
17
18 VAR
19    Data, ValidationResult, TestData, Centroids : MatrixTyp;
20    SplitData                                : SplitDataTyp;
21    i, j, k, l, kValidate, kmeans, done, Count : WORD;
22    c                                         : CHAR;
23    Distance                                  : float;
24    Group                                     : GroupTyp;
25    WSS                                       : ARRAY[1..kNNmax] OF float;
26
27 PROCEDURE ReadCSV(n, p: WORD; FileName: STRING; VAR Data: MatrixTyp);
28 // Read correlation matrix from CSV FILE
29
30 VAR
```

```

31     InputFile : TEXT;
32     i, j      : WORD;
33     x          : float;
34     c          : CHAR;

35
36 BEGIN
37   Assign(InputFile, Filename);
38   Reset(InputFile);
39   IF IOResult <> 0
40     THEN
41       BEGIN
42         c := WriteErrorMessage('Unable to open file, Press <CR>');
43         HALT;
44       END;
45   ReadLn(InputFile);           // ignore first Line WITH headers
46   ReadLn(InputFile);           // ignore second Line WITH types
47   CreateMatrix(Data, n, p, 0.0);
48   IF MatrixError
49     THEN
50       BEGIN
51         c := WriteErrorMessage('program terminated');
52         HALT;
53       END;
54   FOR i := 1 TO n DO          // now Read data from following lines
55     BEGIN
56       FOR j := 1 TO p DO
57         BEGIN
58           x := ReadFloat(InputFile);
59           IF MathError
60             THEN
61               BEGIN
62                 c := WriteErrorMessage('Unable to read datum, press
63                   <CR>');
64                 HALT;
65               END;
66               SetMatrixElement(Data, i, j, x);
67             END; { for j }
68           ReadLn(InputFile);
69         END; { for i }
70       Close(InputFile);
71     END; { ReadCSV }

72
73 BEGIN
74   k := 2;
75   ReadCSV(n, p, 'All.csv', Data);

```

```

76     ShellSortMatrix(Data, 1);
77     // ensure that data are sorted by study no
78     kValidate := floor(Sqrt(n));                                // groups FOR k-fold
    cross-validation
79     IF kValidate > MaxK THEN kValidate := MaxK;
80     SplitDataMatrix(Data, kValidate, SplitData);
81     IF CrossValidationError
82     THEN
83         BEGIN
84             c := WriteErrorMessage('program terminated');
85             HALT;
86         END;
87         CreateMatrix(ValidationResult, n, 3, 0.0);
88     IF MatrixError
89     THEN
90         BEGIN
91             c := WriteErrorMessage('program terminated');
92             HALT;
93         END;
94     Count := 1;
95     FOR i := 1 TO kvalidate DO
96         BEGIN
97             CreateTestData(SplitData, i, kValidate, TestData); // create a data
               vector
98             Writeln('Test data created ', i: 3);
99             IF CrossValidationError
100            THEN
101                BEGIN
102                    c := WriteErrorMessage('program terminated');
103                    HALT;
104                END;
105                Distance := kMeansCluster(TestData, k, Centroids, Group);
106                AssignTestData(SplitData[i], Centroids, Count, kValidate,
               ValidationResult);
107                WSS[i] := WithinSumOfSquares(ValidationResult);
108            END;
109        END.

```

## References

- [1] P.H.A. SNEATH, R.R. SOKAL: *Numerical Taxonomy: The Principles and Practice of Numerical Classification* San Francisco: Freeman, 1973 ISBN: 9780716706977.
- [2] J.C. GOWER: A general coefficient of similarity and some of its properties, *Biometrics* **27**:4 (1971), 857–871 DOI: [10.2307/2528823](https://doi.org/10.2307/2528823).

- [3] Z. HUBALEK: Coefficients of association and similarity, based on binary (presence-absence) data: an evaluation, *Biol. Rev.* **57**:4 (1982), 669–689 DOI: [10.1111/j.1469-185X.1982.tb00376.x](https://doi.org/10.1111/j.1469-185X.1982.tb00376.x).
- [4] J. PODANI: Extending Gower's general coefficient of similarity to ordinal characters, *Taxon* **48**:2 (1999), 331–340 DOI: [10.2307/1224438](https://doi.org/10.2307/1224438).
- [5] A.J. CAIN, G.A. HARRISON: An analysis of the taxonomist's judgment of affinity, *Proc. Zool. Soc. London* **131**:1 (1958), 85–98 DOI: [10.1111/j.1096-3642.1958.tb00634.x](https://doi.org/10.1111/j.1096-3642.1958.tb00634.x).
- [6] J. MACQUEEN: Some methods for classification and analysis of multivariate observations, In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* L.M. LE CAM, J. NEYMAN (editor) vol. 1: Theory of Statistics 14 Berkeley, Los Angeles: University of California Press, 1967, 281–297 URL: <https://www.cs.cmu.edu/~bhiksha/courses/mlsp.fall2010/class14/macqueen.pdf>.
- [7] J.A. HARTIGAN, M.A. WONG: A k-means clustering algorithm, *J. Royal Stat. Soc. C (Appl. Stat.)* **28**:1 (1979) Algorithm AS136: k-means clustering, 100–108 DOI: [10.2307/2346830](https://doi.org/10.2307/2346830).
- [8] J. LI et al.: Robust K-Median and K-Means Clustering Algorithms for Incomplete Data, *Math. Prob. Eng.* (2016), 4321928 DOI: [10.1155/2016/4321928](https://doi.org/10.1155/2016/4321928).
- [9] L. KAUFMAN, P.J. ROUSSEEUW: *Partitioning around medoids (program pam)*, In: *Finding groups in data: an introduction to cluster analysis* New York: Wiley, 1990 chap. 2, 68–125 ISBN: 0471735787.



# **Part IV.**

# **Appendix**



# Symbols and abbreviations

## 1. Symbols used

---

$n$		number of cases
$p$		number of variables
$q$		number of extracted (significant) eigenvalues/-vectors
$r$		rows of a contingency table
$c$		columns of a contingency table
$\psi$		error term
$\omega$		weight of form matrix during shrinkage
$v$		degrees of freedom
$\mathcal{A}$	$a$ $p \times q$	transformation matrix to calculate $\mathcal{C}$ from $\mathcal{L}$
$\mathcal{B}$	$b$ $p \times q$	rotated factor matrix
$\mathcal{C}$	$c$ $n \times q$	factor scores ( $\mathcal{C} = \mathcal{X}_{n \times p} \mathcal{F}_{p \times q}$ )
$\mathcal{E}$	$e$ $p \times p$	eigenvectors in columns, all eigenvector have unit length
$\mathcal{F}$	$f$ $p \times q$	projection matrix, $q$ most significant eigenvectors
$\mathcal{G}$	$g$ $q \times q$	gradient matrix (during rotation)
$\mathcal{H}$	$h$ $n \times n$	hat-matrix
$\mathcal{I}$	$i$	identity matrix (diagonal 1, all other elements 0)
$\mathcal{L}$	$l$ $p \times q$	factor loadings, correlations between variables and factors
$\mathcal{R}$	$r$ $p \times p$	correlation matrix
$\mathcal{S}$	$s$ $p \times p$	variance-covariance matrix $\mathcal{S} = \mathcal{X}^T \mathcal{X} / (p - 1)$
$\mathcal{T}$	$t$ $r \times c$	contingency table
$\mathcal{U}$	$u$ $p \times p$	diagonal matrix of uniqueness
$\mathcal{X}$	$x$ $n \times p$	original data
$\mathcal{Z}$	$z$ $n \times p$	$z$ -transformed data
$\Theta$	$\theta$ $q \times q$	rotation matrix
$r$	$r$ $r$	row sums of a contingency table
$\mathfrak{C}$	$c$ $c$	column sums of a contingency table
$\Lambda$	$\lambda$ $p \times p$	diagonal matrix of eigenvalues
$\mathfrak{h}$	$h$ $p$	communalities, sum of squared factor loadings for each variable

---

## 2. Abbreviations

**AIC** AKAIKE information criterion

**BIC** BAYESian information criterion

## *Symbols and abbreviations*

**CP/M** control program for microcomputers, outdated 8 bit operating system

**ESS** explained sum of squares,  $\sum_{i=1}^n (\hat{y}_i - \bar{y})^2$

**FA** factor analysis

**knn**  $k$ -nearest neighbours

**LOOCV** Leave-One-Out Cross-Validation

**MAD** median absolute deviation from the median, robust measure of scale in symmetrical distributions

**MAR** missing at random

**MCAR** missing completely at random

**MCD** minimum covariance determinant, robust estimator of location and scatter

**MNAR** missing not at random

**MSE** mean squared error,  $\frac{1}{n}\text{RSS}$

**NaN** not a number, 80x87 representation of illegal or missing data

**OTU** operational taxonomic unit, cases in cluster analysis

**PAM** Partitioning around medoids, clustering method

**PCA** principle component analysis

**PRE** proportionate reduction of error, basis of several correlation coefficients

**RSS** residual sum of squares,  $\sum_{i=1}^n (y_i - \hat{y}_i)^2$

**RSE** residual standard error,  $\sqrt{\frac{\text{RSS}}{n-p-1}}$

**TSS** total sum of squares,  $\sum_{i=1}^n (y_i - \bar{y})^2$

**UPGMA** unweighted pair group method with arithmetic mean, clustering method

**US** United States [of America]

# GNU GENERAL PUBLIC LICENSE

Copyright © 2007 Free Software Foundation, Inc. <https://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document,  
but changing it is not allowed.

## Preamble

T

he GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

## GNU GENERAL PUBLIC LICENSE

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

## TERMS AND CONDITIONS

### 0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an

appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

## 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

## 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output,

## GNU GENERAL PUBLIC LICENSE

given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

### 3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

### 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

### 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

- a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

- b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.
- c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.
- d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

## 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

- a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.
- b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.
- c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.
- d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the

same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

- e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which

it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

## 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

- a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
- b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
- c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
- d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
- e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
- f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

## GNU GENERAL PUBLIC LICENSE

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

### 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

### 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you

do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

#### 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

#### 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement).

## GNU GENERAL PUBLIC LICENSE

To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

### 12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from

those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License “or any later version” applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

## GNU GENERAL PUBLIC LICENSE

### 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

### 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

## END OF TERMS AND CONDITIONS

### How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <text><year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <<https://www.gnu.org/licenses/>>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
<program> Copyright (C) <year> <name of author>
```

```
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an “about box”.

You should also get your employer (if you work as a programmer) or school, if any, to sign a “copyright disclaimer” for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <https://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <https://www.gnu.org/licenses/why-not-lgpl.html>.