***MATRIX – END MATRIX functions***

SPSS macros by Kirill Orlov

kior@akado.ru, ttnphns@gmail.com

<https://www.spsstools.net/en/KO-spssmacros>

All rights reserved

*Functions for MATRIX – END MATRIX.* Large collection of useful statistical, mathematical, restructuring and other functions for matrix session in SPSS. For advanced users in the aid of data analysis and writing of statistical algorithms.

What’s new May 2025

* !KO\_levenshtein new function
* !KO\_lcs new function
* !KO\_needlwun new function
* !KO\_needlwun2 new function

What’s new Jan 2025

* !KO\_apriori new function

What’s new Nov 2024

* !KO\_robustadp new function
* !KO\_robustlts new function
* !KO\_trimmean new function
* !KO\_mestim new function
* !KO\_cronalpha new function
* !KO\_mcdomega new function

**Some functions that you might need often:**

[mean](#_VECTOR_OF_ARITHMETIC);[variance](#_VECTOR_OF_VARIANCES); [descriptive statistics by groups](#_VARIOUS_UNIVARIATE_STATISTICS,); [median](#_VECTOR_OF_MEDIANS); [percentiles](#_PERCENTILES);

[frequencies in variable](#_DUMMY_VARIABLES_AND); [frequency crosstabulation](#_TWO-WAY_FREQUENCY_CROSSTABULATION); [multi-way frequency aggregation](#_INTERACTION_DUMMY_VARIABLES); [frequencies in categorical multiple response set](#_ELEMENTARY_VARIABLES_AND);

[frequencies of given values](#_FREQUENCIES_OF_SPECIFIED); [crosstab for given values](#_TWO-WAY_K_x); [count elements equal to any of the given](#_COUNT_VALUES_IN);

[covariances](#_COVARIANCE_MATRIX); [correlations](#_CORRELATION_MATRIX); [covariances/correlations by groups](#_COVARIANCE/CORRELATION_MATRIX,_BY);

[centering](#_CENTRATION_OF_DATA); [standardization](#_Z-STANDARDIZATION_OF_DATA); [centering by groups](#_CENTRATION_OF_DATA_1); [standardization by groups](#_Z-STANDARDIZATION_OF_DATA_1);

[random values from normal distribution](#_RANDOM_VALUES_FROM); [randomly select n elements](#_SELECT_N_RANDOM);

[recode data](#_RECODE_VALUES_(EXACT);

[write out positions of nonzero elements](#_POSITIONS_OF_NONZERO); [position of the first encounter of a value](#_POSITION_OF_FIRST);

[sort elements](#_SORTING_OF_VECTOR); [sort rows](#_SIMPLE_SORTING_OF); [split data by groups](#_SPLITTING_OF_MATRIX);

[GENERAL INFORMATION](#_Toc198324975)

[MATHEMATICAL FUNCTIONS](#_Toc198324976)

* [CUMULATIVE SUMS IN COLUMNS [!KO\_ccum]](#_Toc198324977)
* [PRODUCT OF ELEMENTS OF MATRIX [!KO\_prod]](#_Toc198324978)
* [SUM OF ABOVE-DIAGONAL OR BELOW-DIAGONAL ELEMENTS OF MATRIX [!KO\_trsum]](#_Toc198324979)
* [CHECK TRIANGULAR INEQUALITY IN DISSIMILARITY MATRIX [!KO\_trineq]](#_Toc198324980)
* [LOWER INCOMLETE GAMMA FUNCTION [!KO\_ligamma]](#_Toc198324981)
* [QR DECOMPOSITION BY HAUSHOLDER REFLECTIONS METHOD [!KO\_qrdc]](#_Toc198324982)
* [HESSENBERG MATRIX / TRIDIAGONAL MATRIX [!KO\_hess]](#_Toc198324983)
* [EIGENDECOMPOSITION OF 2x2 MATRIX [!KO\_eig2x2]](#_Toc198324984)
* [EIGENDECOMPOSITION OF 2x2 MATRIX (SCHUR VARIANT) [!KO\_schur2x2]](#_Toc198324985)
* [EIGENDECOMPOSITION (SCHUR VARIANT) [!KO\_qreig]](#_Toc198324986)
* [DOMINANT EIGENVALUE AND ITS EIGENVECTOR [!KO\_poweig]](#_Toc198324987)

[STATISTICAL DESCRIPTIVE FUNCTIONS](#_Toc198324988)

* [VECTOR OF ARITHMETIC MEANS [!KO\_mean]](#_Toc198324989)
* [VECTOR OF VARIANCES [!KO\_variance]](#_Toc198324990)
* [VECTOR OF VARIANCES (DF=N) [!KO\_variance2]](#_Toc198324991)
* [VECTOR OF SKEWNESSES [!KO\_skewness]](#_Toc198324992)
* [VECTOR OF KURTOSES [!KO\_kurtosis]](#_Toc198324993)
* [VECTOR OF MEDIANS [!KO\_median]](#_Toc198324994)
* [PERCENTILES [!KO\_ptile]](#_Toc198324995)
* [MINIMUM, MAXIMUM, MEAN, AND VARIANCE, BY GROUPS [!KO\_gdescr]](#_Toc198324996)
* [VARIOUS UNIVARIATE STATISTICS, BY GROUPS [!KO\_aggr]](#_Toc198324997)
* [VALUE CLOSEST TO A SPECFIED ONE, FROM BELOW OR ABOVE [!KO\_closest]](#_Toc198324998)
* [COVARIANCE MATRIX [!KO\_cov]](#_Toc198324999)
* [COVARIANCE MATRIX (DF=N) [!KO\_cov2]](#_Toc198325000)
* [CORRELATION MATRIX [!KO\_corr]](#_Toc198325001)
* [COSINE MATRIX [!KO\_cosine]](#_Toc198325002)
* [MATRIX OF COEFFICIENTS OF IDENTITY [!KO\_idc]](#_Toc198325003)
* [MATRIX OF SIMILARITY RATIOS [!KO\_simr]](#_Toc198325004)
* [COVARIANCE, CORRELATION, COSINE, COEFFICIENT OF IDENTITY, SIMILARITY RATIO [!KO\_biv]](#_Toc198325005)
* [RECTANGULAR MATRIX OF COVARIANCES/CORRELATIONS/COSINES [!KO\_rect]](#_Toc198325006)
* [COVARIANCE/CORRELATION MATRIX, BY GROUPS AND POOLED [!KO\_gcov]](#_Toc198325007)
* [BETWEEN-GROUP AND POOLED WITHIN-GROUP SCATTER MATRICES [!KO\_bwscat]](#_Toc198325008)
* [IMAGE OR ANTI-IMAGE MATRIX [!KO\_image]](#_Toc198325009)
* [INFLUENCE MATRIX (HAT MATRIX) [!KO\_hat]](#_Toc198325010)
* [CO-COUNTS MATRICES FOR BINARY DATA [!KO\_bincnt]](#_Toc198325011)
* [MATRIX OF SQUARED EUCLIDEAN DISTANCES [!KO\_seuclid]](#_Toc198325012)
* [RECTANGULAR MATRIX OF SQUARED EUCLIDEAN DISTANCES [!KO\_seuclidr]](#_Toc198325013)
* [MATRIX OF POWERED WEIGHTED MINKOWSKI DISTANCES [!KO\_pwmink]](#_Toc198325014)
* [RECTANGULAR MATRIX OF POWERED WEIGHTED MINKOWSKI DISTANCES [!KO\_pwmink]](#_Toc198325015)
* [MATRIX OF SQUARED MAHALANOBIS DISTANCES [!KO\_smahal]](#_Toc198325016)
* [VECTOR OF SQUARED MAHALANOBIS DISTANCES TO CENTROID [!KO\_smahalc]](#_Toc198325017)
* [MATRIX OF SQUARED BUTLER DISTANCES [!KO\_sbutler]](#_Toc198325018)
* [IS THERE IDENTICAL VALUES IN A MATRIX](#_Toc198325019)
* [LIST OF UNIQUE VALUES OF VECTOR AND POSITION OF THEIR FIRST OCCURRENCE [!KO\_unique]](#_Toc198325020)
* [FREQUENCIES OF SPECIFIED VALUES IN COLUMNS [!KO\_freqval]](#_Toc198325021)
* [TWO-WAY FREQUENCY CROSSTABULATION FOR SPECIFIED VALUES [!KO\_ctabval]](#_Toc198325022)
* [DUMMY VARIABLES AND FREQUENCIES OF VALUES OF A COLUMN [!KO\_freq]](#_Toc198325023)
* [TWO-WAY FREQUENCY CROSSTABULATION [!KO\_crosstab]](#_Toc198325024)
* [INTERACTION DUMMY VARIABLES AND MULTI-WAY FREQUENCY AGGREGATION [!KO\_aggrtab]](#_Toc198325025)
* [INTERACTION DUMMY VARIABLES AND MULTI-WAY FREQUENCY AGGREGATION (DICHOTOMOUS DATA) [!KO\_baggrtab]](#_Toc198325026)
* [ELEMENTARY VARIABLES AND FREQUENCIES IN A CATEGORICAL MULTIPLE RESPONSE SET [!KO\_mrfreq]](#_Toc198325027)
* [PERCENT OF CORRECT CLASSIFICATION [!KO\_classres]](#_Toc198325028)
* [COUNTS OF CONCORDANT, DISCORDANT, AND TIED PAIRS [!KO\_concdisc]](#_Toc198325029)
* [COUNTS OF CONCORDANT, DISCORDANT, AND TIED PAIRS (FROM TABLE) [!KO\_concdisct]](#_Toc198325030)
* [COMEMBERSHIP CONFUSION MATRIX FOR TWO PARTITIONS [!KO\_ccm]](#_Toc198325031)
* [COUNT VALUES IN ROWS (EXACT MATCH) [!KO\_count1]](#_Toc198325032)
* [COUNT VALUES IN ROWS (HITTING INTO RANGE) [!KO\_count2]](#_Toc198325033)
* [CHI-SQUARE (PEARSON AND LIKELIHOOD RATIO) OF TWO-WAY CONTINGENCY TABLE [!KO\_chitab]](#_Toc198325034)
* [PROPORTIONS AND RESIDUALS IN TWO-WAY CONTINGENCY TABLE [!KO\_cells]](#_Toc198325035)
* [MATRIX OF SQUARED CHI-SQUARE DISTANCES IN CONTINGENCY TABLE [!KO\_schitab]](#_Toc198325036)
* [MATRIX OF CHI-SQUARES BETWEEN ROWS OF COUNT DATA PAIRWISELY [!KO\_schi2c]](#_Toc198325037)
* [DISTANCES TO GROUP CENTROIDS [!KO\_dtoc]](#_Toc198325038)
* [DISTANCES BETWEEN GROUP CENTROIDS](#_Toc198325039)
* [DISTANCES TO GROUP CENTROIDS (COMPUTATION FROM DISTANCE MATRIX) [!KO\_dtocfrd]](#_Toc198325040)
* [DISTANCES BETWEEN GROUP CENTROIDS (COMPUTATION FROM DISTANCE MATRIX) [!KO\_dbwcfrd]](#_Toc198325041)
* [AVERAGE DISTANCES TO GROUPS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_datofrd]](#_Toc198325042)
* [AVERAGE DISTANCES BETWEEN GROUPS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dabwfrd]](#_Toc198325043)
* [DISTANCES TO GROUP FARTHEST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dtoffrd]](#_Toc198325044)
* [DISTANCES BETWEEN GROUP FARTHEST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dbwffrd]](#_Toc198325045)
* [DISTANCES TO GROUP NEAREST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dtonfrd]](#_Toc198325046)
* [DISTANCES BETWEEN GROUP NEAREST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dbwnfrd]](#_Toc198325047)
* [DISTANCES TO GROUP MEDOIDS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dtomfrd]](#_Toc198325048)
* [WITHIN-GROUP SUMS OF SQUARES OF DEVIATIONS (COMPUTATION FROM DISTANCE MATRIX) [!KO\_sswfrd]](#_Toc198325049)
* [K NEAREST NEIGBOURS (WRITING OUT), VERSION "RANDOM" [!KO\_knnr]](#_Toc198325050)
* [K NEAREST NEIGBOURS (WRITING OUT), VERSION "PLUS" [!KO\_knnp]](#_Toc198325051)
* [INTRACLASS CORRELATION COEFFICIENT (ONE-WAY) [!KO\_iccow]](#_Toc198325052)
* [INTRACLASS CORRELATION COEFFICIENT (ONE-WAY, BALANCED CLASSES) [!KO\_iccowb]](#_Toc198325053)
* [COEFFICIENT OF DIAGONALNESS OF MATRIX [!KO\_diagns]](#_Toc198325054)
* [OUTLIER-ROBUST MEAN AND VARIANCE (ADP) [!KO\_robustadp]](#_Toc198325055)
* [OUTLIER-ROBUST MEAN AND VARIANCE (LTS) [!KO\_robustlts]](#_Toc198325056)
* [TRIMMED MEAN [!KO\_trimmean]](#_Toc198325057)
* [M-ESTIMATORS OF LOCATION [!KO\_mestim]](#_Toc198325058)
* [CRONBACH'S ALPHA [!KO\_cronalpha]](#_Toc198325059)
* [MCDONALD'S OMEGA [!KO\_mcdomega]](#_Toc198325060)

[STATISTICAL TRANSFORMATION FUNCTIONS](#_Toc198325061)

* [CENTRATION OF DATA COLUMNS [!KO\_center]](#_Toc198325062)
* [Z-STANDARDIZATION OF DATA COLUMNS [!KO\_zscore]](#_Toc198325063)
* [Z-STANDARDIZATION OF DATA COLUMNS (DF=N) [!KO\_zscore2]](#_Toc198325064)
* [SCALING OF DATA COLUMNS TO SS=1 [!KO\_scale]](#_Toc198325065)
* [CENTRATION OF DATA COLUMNS, BY GROUPS [!KO\_gcenter]](#_Toc198325066)
* [Z-STANDARDIZATION OF DATA COLUMNS, BY GROUPS [!KO\_gzscore]](#_Toc198325067)
* [SCALING OF DATA COLUMNS, BY GROUPS [!KO\_gscale]](#_Toc198325068)
* [LINEAR RESCALING [!KO\_rescale]](#_Toc198325069)
* [TWO-WAY CENTRATION](#_Toc198325070)
* [CONVERT COVARIANCE MATRIX INTO CORRELATION [!KO\_covcorr]](#_Toc198325071)
* [CONVERT CORRELATION MATRIX INTO COVARIANCE [!KO\_corrcov]](#_Toc198325072)
* [DOUBLE CENTERING OF MATRIX OF SQUARED DISTANCES [!KO\_dcenter]](#_Toc198325073)
* [CONVERT MATRIX OF ANGULAR SIMILARITIES INTO SQUARED DISTANCES BY COSINE THEOREM [!KO\_sdcosth]](#_Toc198325074)
* [SWITCHING MATRIX OF SCALAR PRODUCTS ONTO NEW DIAGONAL (PRESERVING ANGLE) [!KO\_swdiag1]](#_Toc198325075)
* [SWITCHING MATRIX OF SCALAR PRODUCTS ONTO NEW DIAGONAL (PRESERVING DISTANCE) [!KO\_swdiag2]](#_Toc198325076)
* [SWITCHING MSCP MATRIX ONTO NEW DATA CENTROID [!KO\_swcentr]](#_Toc198325077)
* [DUMMY VARIABLES](#_Toc198325078)
* [MULTIPLE RESPONSE SET VARIABLES](#_Toc198325079)
* [RANK VALUES](#_Toc198325080)
* [RIDITS OF ORDINAL LEVELS [!KO\_ridit]](#_Toc198325081)
* [EXPONENTIAL NORMALIZATION (SOFTMAX FUNCTION) [!KO\_nef]](#_Toc198325082)
* [CENTERED LOGRATIO TRANSFORM [!KO\_clr]](#_Toc198325083)
* [BOX-COX TRANSFORM [!KO\_boxcox]](#_Toc198325084)
* [WHITENING (SPHERING) MATRIX [!KO\_white]](#_Toc198325085)
* [TRANSFORM VARIABLES EXACTLY TO HAVE SPECIFIED COVARIANCES [!KO\_tocov]](#_Toc198325086)
* [TRANSFORM A VARIABLE EXACTLY TO HAVE SPECIFIED COVARIANCES WITH OTHER VARIABLES [!KO\_ytocov]](#_Toc198325087)
* [NORMAL CLOUD INTO UNIFORM BALL [!KO\_unifball]](#_Toc198325088)

[STATISTICAL ANALYTIC FUNCTIONS](#_Toc198325089)

* [LINEAR REGRESSION [!KO\_regress]](#_Toc198325090)
* [PRINCIPAL COMPONENTS [!KO\_pcomp]](#_Toc198325091)
* [PRINCIPAL COMPONENTS (FOR N<P) [!KO\_pcomp2]](#_Toc198325092)
* [PRINCIPAL COMPONENTS (MATRIX INPUT) [!KO\_pca]](#_Toc198325093)
* [ORTHOGONAL PROCRUSTES ROTATION [!KO\_procr]](#_Toc198325094)
* [LINEAR DISCRIMINANTS [!KO\_discrim]](#_Toc198325095)
* [GAUSSIAN CLASSIFIER [!KO\_gaclass]](#_Toc198325096)
* [GAUSSIAN CLASSIFIER (WITH OPTION "LEAVE-ONE-OUT") [!KO\_gaclass2]](#_Toc198325097)
* [CANONICAL CORRELATIONS [!KO\_cancorr]](#_Toc198325098)
* [PRINCIPAL COORDINATES [!KO\_pcoord]](#_Toc198325099)
* [BIPLOT [!KO\_biplot]](#_Toc198325100)
* [SIMPLE CORRESPONDENCE ANALYSIS [!KO\_corresp]](#_Toc198325101)
* [FACTORS BY PRINCIPAL AXIS METHOD [!KO\_paf]](#_Toc198325102)
* [ORTHOGONAL ANALYTIC ROTATIONS [!KO\_ortrot]](#_Toc198325103)
* [OBLIQUE ROTATION PROMAX AND PROMAJ [!KO\_promax]](#_Toc198325104)
* [FACTOR SCORE COEFFICIENTS [!KO\_fsc]](#_Toc198325105)
* [K-MEANS CLUSTERING [!KO\_kmeans]](#_Toc198325106)
* [BLOCK-DIAGONALIZATION BY VAT/IVAT METHOD [!KO\_vat]](#_Toc198325107)
* [K-NEAREST NEIGHBOURS (WITH DEPENDENT VARIABLE) [!KO\_knnpred]](#_Toc198325108)
* [NAIVE BAYES CLASSIFIER [!KO\_nbclass]](#_Toc198325109)

[RANDOM DATA FUNCTIONS](#_Toc198325110)

* [RANDOM VALUES FROM STANDARD UNIFORM DISTRIBUTION](#_Toc198325111)
* [RANDOM VALUES FROM STANDARD NORMAL DISTRIBUTION [!KO\_normal]](#_Toc198325112)
* [RANDOM VALUES FROM NORMAL DISTRIBUTION WITH SPECIFIED PARAMETERS [!KO\_rvnorm]](#_Toc198325113)
* [RANDOM VALUES FROM UNIFORM DISTRIBUTION WITH SPECIFIED PARAMETERS [!KO\_rvunif]](#_Toc198325114)
* [RANDOM VALUES FROM CHI-SQUARE DISTRIBUTION [!KO\_chisq]](#_Toc198325115)
* [RANDOM VALUES FROM BINOMIAL DISTRIBUTION [!KO\_binom]](#_Toc198325116)
* [RANDOM VALUES FROM CATEGORICAL DISTRIBUTION [!KO\_categ]](#_Toc198325117)
* [SAMPLE FROM CATEGORICAL DISTRIBUTION WITHOUT REPLACEMENT [!KO\_catwor]](#_Toc198325118)
* [SELECT N RANDOM ELEMENTS FROM VECTOR [!KO\_sample]](#_Toc198325119)
* [RANDOM MATRIX FROM WISHART DISTRIBUTION [!KO\_wishart]](#_Toc198325120)
* [RANDOM DATA FROM NORMAL POPULATION WITH SPECIFIED COVARIANCES [!KO\_mvnorm]](#_Toc198325121)
* [CREATE RANDOM DATA EXACTLY HAVING SPECIFIED COVARIANCES](#_Toc198325122)

[RECODE/REPLACE FUNCTIONS](#_Toc198325123)

* [RECODE VALUES (EXACT MATCH) [!KO\_recode1]](#_Toc198325124)
* [RECODE VALUES (HITTING INTO RANGE) [!KO\_recode2]](#_Toc198325125)
* [VARIOUS UNIVARIATE STATISTICS, BY GROUPS: REPLACE ORIGINAL VALUES WITH THEM [!KO\_aggrv]](#_Toc198325126)

[SEARCH/HIGHLIGHT FUNCTIONS](#_Toc198325127)

* [POSITIONS OF NONZERO ELEMENTS IN VECTOR [!KO\_indices]](#_Toc198325128)
* [POSITIONS OF NONZERO AND ZERO ELEMENTS IN VECTOR [!KO\_indices2]](#_Toc198325129)
* [POSITIONS OF NONZERO ELEMENTS IN ROWS OR COLUMNS OF MATRIX](#_Toc198325130)
* [POSITIONS OF NONZERO ELEMENTS IN MATRIX (AND THE ELEMENTS THEMSELVES) [!KO\_indicesm]](#_Toc198325131)
* [POSITION OF FIRST ENCOUNTER OF VALUE IN ROW/ROWS [!KO\_indx]](#_Toc198325132)
* [POSITION OF LAST ENCOUNTER OF VALUE IN ROW/ROWS [!KO\_rindx]](#_Toc198325133)
* [POSITION OF RANDOM ENCOUNTER OF VALUE IN ROW/ROWS [!KO\_randindx]](#_Toc198325134)
* [POSITION (PAIR OF INDICES) OF ONE VALUE IN MATRIX [!KO\_ij]](#_Toc198325135)
* [HIGHLIGHT IN ROW AND COLUMN NO MORE THAN ONE ELEMENT EQUAL TO VALUE [!KO\_prime]](#_Toc198325136)
* [INDICATION OF CASES HAVING SPECIFIED PROFILE OF VALUES [!KO\_indic]](#_Toc198325137)
* [HIGHLIGHT DIRECT CHAINS [!KO\_runs]](#_Toc198325138)
* [HIGHLIGHT DIRECT CHAINS (ANOTHER ALGORITHM) [!KO\_runs2]](#_Toc198325139)
* [HIGHLIGHT SLANT CHAINS [!KO\_slant]](#_Toc198325140)
* [HIGHLIGHT SLANT CHAINS (ANOTHER ALGORITHM) [!KO\_slant2]](#_Toc198325141)
* [DEPTH-FIRST SEARCH IN UNIPARTITE GRAPH [!KO\_dfs]](#_Toc198325142)
* [BREADTH-FIRST SEARCH IN UNIPARTITE GRAPH [!KO\_bfs]](#_Toc198325143)
* [BRON-KERBOSCH MAXIMAL CLIQUES ALGORITHM (PIVOT VERSION) [!KO\_bronkerb]](#_Toc198325144)
* [BRON-KERBOSCH MAXIMAL CLIQUES ALGORITHM (BASE VERSION) [!KO\_bronkerb2]](#_Toc198325145)
* [CONVEX HULL: 2D JARVIS ALGORITHM [!KO\_chjarv]](#_Toc198325146)
* [CONVEX HULL: 2D GRAHAM ALGORITHM [!KO\_chgrah]](#_Toc198325147)
* [MAGNITUDES OF ELEMENTS OFFSET FROM MATRIX DIAGONAL [!KO\_diagoff]](#_Toc198325148)
* [MAGNITUDES OF ELEMENTS OFFSET FROM MATRIX DIAGONAL BAND [!KO\_diagboff]](#_Toc198325149)

[RESTRUCTURING FUNCTIONS](#_Toc198325150)

* [READ INLINE DATA [!KO\_read]](#_Toc198325151)
* [READ INLINE DATA (NO UNQUOTING) [!KO\_read2]](#_Toc198325152)
* [SPARSE MATRIX INTO LIST OF NONZERO VALUES [!KO\_nzlist]](#_Toc198325153)
* [LIST INTO SPARSE MATRIX [!KO\_nzunlist]](#_Toc198325154)
* [RAMMING HORIZONTALLY NONZERO ELEMENTS OR THEIR INDICES [!KO\_ram]](#_Toc198325155)
* [RAMMING VERTICALLY NONZERO ELEMENTS OR THEIR INDICES [!KO\_vram]](#_Toc198325156)
* [PROPAGATE ROWS [!KO\_propag]](#_Toc198325157)
* [SELECT ROWS (FILTER CASES)](#_Toc198325158)
* [ADD A CONSTANT TO ELEMENTS OF A MATRIX TRIANGLE [!KO\_tradd]](#_Toc198325159)
* [MULTIPLY ELEMENTS OF A MATRIX TRIANGLE BY A CONSTANT [!KO\_trmult]](#_Toc198325160)
* [EXPONENTIATE ELEMENTS OF A MATRIX TRIANGLE BY A CONSTANT [!KO\_trexp]](#_Toc198325161)
* [UNFOLD THE TRIANGLES OF SQUARE MATRIX INTO VECTORS [!KO\_unftri]](#_Toc198325162)
* [FOLD TWO VECTORS INTO THE TRIANGLES OF SQUARE MATRIX [!KO\_foltri]](#_Toc198325163)
* [SYMMETRIZATION OF SQUARE MATRIX: REPLACE ONE TRIANGLE BY THE OTHER [!KO\_symtri1]](#_Toc198325164)
* [SYMMETRIZATION OF SQUARE MATRIX: REPLACE GREATER ELEMENT BY LESSER OR LESSER BY GREATER [!KO\_symtri2]](#_Toc198325165)
* [SORTING OF VECTOR [!KO\_sort]](#_Toc198325166)
* [SIMPLE SORTING OF ROWS IN MATRIX [!KO\_rsort]](#_Toc198325167)
* [SORTING OF ROWS IN MATRIX BY VALUES OF EXTERNAL COLUMN [!KO\_rsortc]](#_Toc198325168)
* [HIERARCHICAL SORTING OF ROWS IN MATRIX [!KO\_hiesort]](#_Toc198325169)
* [SORT IN RANDOM ORDER](#_Toc198325170)
* [SPLITTING OF MATRIX INTO TWO [!KO\_split]](#_Toc198325171)
* [MERGING OF TWO MATRICES [!KO\_merge]](#_Toc198325172)
* [SPLITTING OF MATRIX HORIZONTALLY IN SEVERAL, BY GROUPS [!KO\_msplit]](#_Toc198325173)
* [RESTRUCTURING "VARIABLES INTO CASES" [!KO\_vartocas]](#_Toc198325174)
* [RESTRUCTURING "CASES INTO VARIABLES" [!KO\_castovar]](#_Toc198325175)
* [SHIFT ROWS HORIZONTALLY [!KO\_shift]](#_Toc198325176)
* [BLOCK-INCUT [!KO\_blockic]](#_Toc198325177)

[COMBINATORIAL FUNCTIONS AND SETS](#_Toc198325178)

* [DYADIC SET OPERATIONS [!KO\_setdo]](#_Toc198325179)
* [CARTESIAN PRODUCT BETWEEN TWO SETS [!KO\_cartp]](#_Toc198325180)
* [CARTESIAN PRODUCT BETWEEN N SETS [!KO\_ncartp]](#_Toc198325181)
* [COMBINATIONS BY K ELEMENTS [!KO\_combk]](#_Toc198325182)
* [ALL COMBINATIONS BY K=2,3... ELEMENTS [!KO\_allcomb]](#_Toc198325183)
* [COMBINATIONS BY K ELEMENTS FROM DIFFERENT SETS [!KO\_dscombk]](#_Toc198325184)
* [ALL COMBINATIONS BY K=2,3... ELEMENTS FROM DIFFERENT SETS [!KO\_dsallcomb]](#_Toc198325185)
* [HORIZONTAL ARITHMETIC OPERATIONS IN COMBINATIONS OF VARIABLES [!KO\_comboper]](#_Toc198325186)
* [QUANTITIES ON THE BASIS OF HORIZONTAL SUMS OR MEANS IN COMBINATIONS OF VARIABLES [!KO\_turflike]](#_Toc198325187)
* [APRIORI ALGORITHM [!KO\_apriori]](#_Toc198325188)

[COMBINATORIAL OPTIMIZATION FUNCTIONS](#_Toc198325189)

* [MATCHING BY HUNGARIAN ALGORITHM [!KO\_hungar]](#_Toc198325190)
* [MATCHING BY HOPCROFT-KARP ALGORITHM [!KO\_hopckarp]](#_Toc198325191)
* [SIMPLE GREEDY MATCHING [!KO\_greedm]](#_Toc198325192)
* [SIMPLE GREEDY MATCHING (DISCRETE DATA) [!KO\_greedm2]](#_Toc198325193)
* [FLOYD-WARSHALL ALGORITHM: SHORTEST PATHS / EASIEST PASSES [!KO\_flowar]](#_Toc198325194)
* [FLOYD-WARSHALL ALGORITHM (SYMMETRIC MATRIX) [!KO\_sflowar]](#_Toc198325195)
* [DIJKSTRA ALGORITHM: SHORTEST PATH / EASIEST PASS [!KO\_dijkstra]](#_Toc198325196)
* [PRIM'S MINIMUM SPANNING TREE ALGORITHM [!KO\_prim]](#_Toc198325197)

[SEQUENCES](#_Toc198325198)

* [LEVENSHTEIN DISTANCE (WAGNER-FISCHER ALGORITHM) [!KO\_levenshtein]](#_Toc198325199)
* [LONGEST COMMON SUBSEQUENCE [!KO\_lcs]](#_Toc198325200)
* [NEEDLEMAN-WUNSCH ALGORITHM [!KO\_needlwun]](#_Toc198325201)
* [NEEDLEMAN-WUNSCH ALGORITHM (WITH SIMILARITY MATRIX) [!KO\_needlwun2]](#_Toc198325202)

# GENERAL INFORMATION

Matrix session in SPSS Statistics is a series of special commands standing between commands MATRIX and END MATRIX (description of commands of the matrix language see in *SPSS Statistics Command Syntax Reference*). In a matrix session you operate with data as with whole matrices, what allows quickly and effectively to perform transformations that would be hard to program with regular (out-of-matrix) syntax. Matrix session is convenient for programming your own statistical or other algorithms. Unfortunately, options of matrix language in SPSS have long remained unupgraded; there is obvious shortage of useful functions. Therefore, I’ve written a number of functions which are so far absent in the matrix language of SPSS and in which I see a sharp need. These functions are written on the SPSS matrix language itself and are designed as SPSS macros. Thus, the functions are generally not as quick as when they would have been written on a low-level language. Nevertheless, until IBM SPSS radically enhance its matrix facility language I recommend these functions to users.

***Rules***

1) Arguments in functions are separated by character %, not comma.

2) Functions should be written and run like commands, starting a line – not as “right hand side”, and the name of the result to return should be written among the arguments as the last one of them. I.e., correct: **!function(argument % result)**; not correct: **result = !function(argument).** Some functions return more than one result.

3) Any argument except the result may be expression, not just name or value. For example, **!function(1.5\*name(2,4)+3 % result)** is allowed. The expression however cannot contain the functions (the macros) described in this document.

4) The result name may coincide with any of the arguments’ name. I.e., it is allowed, for example: **!function(name1 % name2 % name1).** In this case the result will be *name1*, and the object which formerly carried this name will be replaced.

5) Do not use in your syntax within MATRIX - END MATRIX names starting by character @ because this character is reserved for the functions here, and so using it as the 1st character is fraught with loss of data.

6) Functions *do not check and do not issue* special notifications whether your input data meet the requirements of the given function (so was made for speed performance). Therefore, to minimize chance of error or erroneous result do attentively read the requirements of a specific function.

7) Functions may be put inside LOOP - END LOOP and DO IF - END IF.

# 

8) To disable a function in syntax use not \* but /\*, for example:

/\*!func(arg%result).

or

/\*!func(arg%result).\*/.

9) These matrix functions were designed for numeric input data. Some of the functions, not tied with numeric analysis, *might* work with text (string) data (such as counting of values, recoding, restructuring), but it is not guaranteed since the code was written in expectation of numbers. So, try and check with string data yourself.

10) Many of these functions require to set in advance, prior the first MATRIX command in a session, the maximum number of cycles to a sufficiently big value, in order to cope with potentially large data. I recommend to do beforehand the setting on big number of cycles just always when you are about to work with these functions – to save from worrying about that in the sequel. For example:

**set mxloops 100000000.** /\*or 1E8

**matrix.**

**etc.**

11) Variance/covariance and correlation, when computed within the functions, not always use one and the same calculation formula. For example, /\*!KO\_variance \*/ or /\*!KO\_corr\*/ use one formula (giving more numerically stable result) while other functions may calculate by another formula (giving less numerically stable result).

# MATHEMATICAL FUNCTIONS

### CUMULATIVE SUMS IN COLUMNS [!KO\_ccum]

\*/\*!KO\_ccum(mat%dir%name)\*/\*.

\*Version 1.

\*Takes matrix or column MAT with number of rows at least 2 and returns matrix NAME of the same size;

\*elements in each column of NAME are cumulative sums in that column by the direction of summation

\*downward (if DIR positive) or upward (if DIR nonpositive).

\*DIR - scalar.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### PRODUCT OF ELEMENTS OF MATRIX [!KO\_prod]

\*/\*!KO\_prod(mat%name)\*/\*.

\*Version 2.

\*Computes product of elements of MAT.

\*Returns scalar NAME.

EXAMPLE. Factorial of 6.

matrix.

!KO\_prod({1:6}%fact).

print fact.

end matrix.

### SUM OF ABOVE-DIAGONAL OR BELOW-DIAGONAL ELEMENTS OF MATRIX [!KO\_trsum]

\*/\*!KO\_trsum(mat%tr%name)\*/\*.

\*Version 1.

\*Sums up elements of the upper or the lower triangle of MAT.

\*Returns scalar NAME.

\*Matrix MAT needs not be square, but must be at least size 2x2.

\*TR - scalar; if positive, elements above the main diagonal are summed;

\*if nonpositive, elements below the main diagonal are summed.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### CHECK TRIANGULAR INEQUALITY IN DISSIMILARITY MATRIX [!KO\_trineq]

\*/\*!KO\_trineq(dis%ineq%stop%name1%name2)\*/\*.

\*Version 1.

\*Takes square matrix of distances (dissimilarities) DIS and verifies if there are distances

\*violating "axiom of triangular inequality". If there are such, distances in matrix are not metric.

\*The function checks only above-diagonal elements of DIS; if you want to check below-diagonal

\*elements of DIS - transpose the matrix.

\*DIS - square, with nonnegative values, sized at least 3.

\*INEQ - digit 0 or 1 (not name or expression; optionally may cover the digit by quotes or apostrophes).

\*If 0, slack triangular inequality will be verified: Dab <= Dac+Dbc (where a b c are objects, i.e.

\*matrix rows/columns). If 1, strict triangular inequality will be verified: Dab < Dac+Dbc (Dab = Dac+Dbc

\*doesn't count as violation).

\*STOP - digit 0 or 1 (not name or expression; optionally may cover the digit by quotes or apostrophes).

\*If 0, checked will be all above-diagonal elements: the function works till the end. If 1, the function

\*breaks upon the first encounter of violation of triangular inequality, if there are any.

\*Results:

\*NAME1 - row vector of length 2: first value is the number of triplets (combination of elements by 3)

\*checked above the matrix diagonal; second value - number of triplets of them with triangle inequality

\*violated. (With STOP=1 this value is always =1.)

\*NAME2 - row vector of length 4; it contains maximal found magnitude of triangle inequality violation

\*(1st value in the vector) and the three objects' indices (matrix rows/columns) making that triplet.

\*If no violations were found, NAME2 contains all -1.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1000000.

matrix.

compute dis= {0, 12, 10, 17;

12, 0, 22, 8;

10, 22, 0, 13;

17, 8, 13, 0}.

print dis.

!KO\_trineq(dis%'1'%'0'%name1%name2).

print name1 /title 'Number of triplets of (above-diagonal) elements checked'+

' and number of defective triplets'.

print name2 /title 'The size of maximal defect encountered and the indices of'+

' corresponding three objects'.

end matrix.

### LOWER INCOMLETE GAMMA FUNCTION [!KO\_ligamma]

\*/\*!KO\_ligamma(arg%ubound%name)\*/\*.

\*Version 1.

\*Returns value of lower incomplete gamma function for argument ARG and upper bound

\*of integration UBOUND.

\*ARG – positive number or numbers: scalar, vector, or matrix.

\*UBOUND – non-negative number or numbers: matrix sized as ARG.

\*Returns matrix NAME sized as ARG.

\*This function requires prior setting of limit for number of cycles to 100 (or more)

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 100.

matrix.

!KO\_ligamma(0.5%12.6%result).

print result.

end matrix.

### QR DECOMPOSITION BY HAUSHOLDER REFLECTIONS METHOD [!KO\_qrdc]

\*/\*!KO\_qrdc(mat%name1%name2)\*/\*.

\*Version 1.

\*Does QR decomposition of matrix MAT into upper triangular matrix R (NAME2) and orthonormal

\*matrix Q (NAME1) by Housholder reflections method. Q\*R=MAT.

\*MAT - matrix with number of columns not less than the number of rows.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*In-built function GSCH (Gram-Schmidt orthonormalization) is also a QR decomposition, using another

\*method and giving different result.

### HESSENBERG MATRIX / TRIDIAGONAL MATRIX [!KO\_hess]

\*/\*!KO\_hess(mat%name1%name2)\*/\*.

\*Version 1.

\*Turns square matrix MAT (of size not less than 3) into upper Hessenberg matrix NAME2.

\*That matrix is upper triangular with the upper sub-diagonal filled also, and its eigenvalues

\*are the same as of MAT. There returned is also an orthonormal matrix NAME1, such

\*that NAME1\*MAT\*t(NAME1)=NAME2 and t(NAME1)\*NAME2\*NAME1=MAT.

\*If MAT is a symmetric matrix, then NAME2 comes out tridiagonal.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### EIGENDECOMPOSITION OF 2x2 MATRIX [!KO\_eig2x2]

\*/\*!KO\_eig2x2(mat%name1%name2)\*/\*.

\*Version 1.

\*Returns (real) eigenvalues (NAME2) and eigenvectors (columns of NAME1) of a square

\*matrix MAT sized 2x2. MAT may be symmetric or asymmetric; in the latter case

\*the eigenvectors may be nonorthogonal (use function /\*!KO\_schur2x2\*/ to obtain

\*orthogonal eigenvectors).

\*Restoration of the matrix: MAT = NAME1\*mdiag(NAME2)\*inv(NAME1).

\*If eigenvalues of MAT are not real, NAME1 and NAME2 are returned as zero scalars.

EXAMPLE.

matrix.

compute A= {4,-2;-1,3.6}.

print A.

!KO\_eig2x2(A%eivec%eival).

print eival. /\*eigenvalues L

print eivec. /\*eigenvectors V

print sscp(eivec). /\*V are not orthogonal if A is asymmetric

print ( eivec\*mdiag(eival)\*inv(eivec) ). /\*A restored

end matrix.

### EIGENDECOMPOSITION OF 2x2 MATRIX (SCHUR VARIANT) [!KO\_schur2x2]

\*/\*!KO\_schur2x2(mat%name1%name2)\*/\*.

\*Version 1.

\*Returns (real) eigenvalues and eigenvectors of a square matrix MAT sized 2x2.

\*Eigenvalues are placed on the diagonal of the upper triangular matrix NAME2, and eigenvectors

\*are the columns of NAME1. These are orthogonal eigenvectors and NAME1 is a unitary matrix.

\*Restoration of the matrix: MAT = NAME1\*NAME2\*t(NAME1).

\*If MAT is symmetric, then NAME2 is a diagonal matrix.

\*If eigenvalues of MAT are not real, NAME1 and NAME2 are returned as zero scalars.

\*For Schur decomposition of a matrix of greater size use /\*!KO\_qreig\*/.

### EIGENDECOMPOSITION (SCHUR VARIANT) [!KO\_qreig]

\*/\*!KO\_qreig(mat%needvec%conv%name1%name2%name3)\*/\*.

\*Version 1.

\*Does eigendecomposition of a square matrix MAT (symmetric or asymmetric).

\*Algorithm used is Implicit QR iterations on recursively splitted Hessenberg matrix with

\*double (Francis) shifts. The function returns upper triangular matrix NAME2 (with the eigenvalues

\*on its diagonal, Schur matrix) and unitary matrix NAME1, which columns are the mutually orthogonal

\*eigenvectors (or, rather, these orthogonal vectors should be called Schur vectors).

\*The order of eigenvectors corresponds to the order of eigenvalues but the eigenvalues are not

\*necessarily sorted by magnitude. If MAT is symmetric, NAME2 is diagonal.

\*Restoration of the matrix: MAT = NAME1\*NAME2\*t(NAME1).

\*Argument NEEDVEC - digit (not name or expression) 1 or 0 (the digit may optionally be in quotes or

\*apostrophes). If "1", the function will return eigenvalues NAME2 and eigenvectors NAME1. If "0",

\*the function will return only eigenvalues NAME2, not spending time on computation of the

\*eigenvectors (NAME1 will come out as scalar 1).

\*Argument CONV - scalar; this is the being set precision of convergence at computation of

\*eigenvalues; recommended value: from 1E-8 to 1E-12.

\*NAME3 - this output scalar is the maximal required number of iterations to compute a single

\*eigenvalue.

\*Unconvergence/fail. Asymmetric matrices MAT sometimes have complex (not real) eigenvalues tied

\*in pairs. If some subdiagonal element in NAME2 appeared nonzero while NAME3 is not above 100

\*(the limiting number of iterations in the function), a pair of complex eigenvalues is evident.

\*The real parts of these two values are found above and to the right from the unconverged-to-zero

\*subdiagonal element, however they are not necessarily calculated precisely. If a subdiagonal

\*element in NAME2 appeared nonzero but NAME3=101 (i.e. all iterations exhausted), a fail of the

\*algorithm is evident - this is relatively rare case, associated with a special defectness of

\*the given MAT.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If MAT is size 2x2, function /\*!KO\_schur2x2\*/ does the same decomposition as this function.

\*If MAT is symmetric, the in-built function CALL EIGEN does the same decomposition as this function.

EXAMPLE. Decomposition of a random matrix with preset eigenvalues.

set mxloops 10000.

matrix.

compute L= (uniform(10,1)-.3)\*2. /\*Some random values to use as eigenvalues

print L.

compute s= uniform(10,10)-.3.

compute A= s\*mdiag(L)\*inv(s). /\*Random matrix A with eigenvalues L

print A.

!KO\_qreig(A%1%1E-10%eivec%schur%mxit).

print mxit.

print schur. /\*Schur matrix T: normally upper triangular matrix with eigenvalues

print diag(schur). /\*occupying its diagonal

print eivec. /\*Columns are corresponding eigenvectors U

print sscp(eivec). /\*which are orthogonal

print ( eivec\*schur\*t(eivec) ). /\*Reconstruction of A: U\*T\*t(U)

end matrix.

EXAMPLE. Matrix with complex eigenvalues.

set mxloops 10000.

matrix.

compute A=

{7, 3, 4, -11, -9, -2;

-6, 4, -5, 7, 1, 12;

-1, -9, 2, 2, 9, 1;

-8, 0, -1, 5, 0, 8;

-4, 3, -5, 7, 2, 10;

6, 1, 4, -11, -7, -1}. /\*Matrix with two real eigenvalues (3, 4) and

/\*two complex conjugate paired eigenvalues (1±2i, 5±6i)

print A.

!KO\_qreig(A%0%1E-10%eivec%schur%mxit).

print mxit.

print schur. /\*Because of complex eigenvalues, lower triangle of the Schur matrix

/\*is not completely empty; still, out of 6 eigenvalues both real and

/\*one pair of complex ones converged to their true values

end matrix.

### DOMINANT EIGENVALUE AND ITS EIGENVECTOR [!KO\_poweig]

\*/\*!KO\_poweig(mat%inivec%conv%name1%name2%name3)\*/\*.

\*Version 1.

\*Takes square n x n matrix MAT and returns its dominant eigenvalue NAME2

\*(i.e., the eigenvalue with the greatest absolute magnitude) and associated with it

\*eigenvector NAME1. Used is the Power Iteration method.

\*The algorithm converges if the dominant eigenvalue exists in a single copy,

\*and converges the faster the stronger it dominates over the other eigenvalues (i.e., the

\*"spectral gap" is large). According to the Perron-Frobenius theorem, any positive square matrix

\*and some (so called primitive) non-negative square matrices have dominant eigenvalue

\*strictly in one copy (and all of its eigenvector elements are of the same sign).

\*Therefore in case of these matrices the convergence is guaranteed.

\*INIVEC - initial eigenvector for iterations, column of length n. Usually it is random numbers

\*from the uniform distribution. If you have more accurate initial estimate of the eigenvector,

\*specify it as INIVEC.

\*CONV - criteria of iterative convergence: vector of 2 numbers, for example {1E-9,100}. The first is

\*the threshold for eigenvector change, nonnegative number; if the greatest change in it is less

\*than the threshold, the vector will be considered stabilized and iterations will stop. The second

\*number is the maximal allowed number of iterations - specify positive integer.

\*NAME3 - report of convergence, vector of 2 values which is comparable with argument CONV:

\*the 1st value is the maximal change in eigenvector observed on the last iteration done and

\*which, as desired, should be less than the threshold in CONV. The 2nd number is the number of

\*iterations done; if this appears by 1 greater than the specified in CONV then it means that the

\*maximal number of iterations was done but the requested threshold of convergernce was not reached.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

# STATISTICAL DESCRIPTIVE FUNCTIONS

### VECTOR OF ARITHMETIC MEANS [!KO\_mean]

\*/\*!KO\_mean(data%name)\*/\*.

\*Version 1.

\*Computes means in columns of DATA.

\*Returns row vector NAME.

### VECTOR OF VARIANCES [!KO\_variance]

\*/\*!KO\_variance(data%name)\*/\*.

\*Version 1.

\*Computes variances in columns of DATA.

\*Returns row vector NAME.

### VECTOR OF VARIANCES (DF=N) [!KO\_variance2]

\*/\*!KO\_variance2(data%name)\*/\*.

\*Version 1.

\*Computes biased (computed on "df=n", not "df=n-1") variances in columns of DATA.

\*Returns row vector NAME.

### VECTOR OF SKEWNESSES [!KO\_skewness]

\*/\*!KO\_skewness(data%name)\*/\*.

\*Version 1.

\*Computes skewnesses in columns of DATA.

\*Returns row vector NAME.

### VECTOR OF KURTOSES [!KO\_kurtosis]

\*/\*!KO\_kurtosis(data%name)\*/\*.

\*Version 1.

\*Computes kurtoses (tailedness) in columns of DATA. DATA must have at least three rows.

\*Returns row vector NAME.

### VECTOR OF MEDIANS [!KO\_median]

\*/\*!KO\_median(data%name)\*/\*.

\*Version 1.

\*Computes medians in columns of DATA.

\*Returns row vector NAME.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 100000.

matrix.

get vars /variables= v1 to v3.

!KO\_median(vars%medians).

print medians.

end matrix.

### PERCENTILES [!KO\_ptile]

\*/\*!KO\_ptile(data%sort%p%method%name)\*/\*.

\*Version 1.

\*Computes requested percentiles in columns of DATA.

\*Returns matrix NAME with rows corresponding to the percentile levels.

\*SORT - digit (not name or expression) 1 or 0 (optionally in quotes or apostrophes). Use 0 only

\*in the special case when values in each column of DATA are already sorted ascendingly; this

\*option allows to spare time doing the sorting.

\*P - scalar or vector (row or column) containing percents (percent levels, percentile ranks) for

\*which you need to determine percentile values in the variables (each column of DATA).

\*Values in P must be greater than 0 and lesser than 100. They need not to go in ascending order.

\*METHOD - method to compute percentiles, capitalized keyword (optionally in quotes or apostrophes):

\*"WAVER" - Weighted Average Definition 2

\*"HAVER" - Weighted Average Definition 1

\*"HAVER2" - Weighted Average Definition 1

\*"EMPIR" - Empirical Distribution Function

\*"AEMPIR" - Empirical Distribution Function with Averaging

\*"ROUND" - Observation closest aka Rounded

\*These are the methods found in SPSS procedure EXAMINE (Explore). HAVER and HAVER2 are the same,

\*but HAVER2 does not compute percentiles that are "too high" for the input data (replacing them

\*in NAME with arbitrary number -999), like SPSS procedures Explore and Frequencies do for the

\*default method "HAVERAGE". HAVER puts, instead of "-999", the maximal value in the variable.

\*Most popular or recommended methods are HAVER/HAVER2 and AEMPIR. 50th percentile of these methods

\*coincides with the median after function /\*!KO\_median\*/.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 100000.

matrix.

get vars /variables= v1 v2 v3.

compute plevels= {5,25,50,90}. /\*Request percentiles for

!KO\_ptile(vars%1%plevels%AEMPIR%ptiles). /\*5-th, 25-th, 50-th, 90-th %

print {t(plevels),ptiles} /clabel= '%-level' 'v1' 'v2' 'v3'.

end matrix.

### MINIMUM, MAXIMUM, MEAN, AND VARIANCE, BY GROUPS [!KO\_gdescr]

\*/\*!KO\_gdescr(col%bin%name1%name2%name3%name4)\*/\*.

\*Version 2.

\*Takes column COL (one analyzed variable) and one or more binary (1 vs 0)

\*variables constituting columns of matrix BIN with number of rows as in COL.

\*Each column of BIN represents a group, and value 1 in it means that the case (row) belongs to it.

\*Groups may intersect or be disjoint by the composition of cases (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros, in which case the group corresponds to the

\*whole dataset, all its rows: statistic "in total" will be computed.

\*The function returns 4 columns of by-group univariate statistics, with rows corresponding to the groups, columns of BIN:

\*minimal value (NAME1), maximal value (NAME2), arithmetic mean (NAME3),

\*variance (on "df=n-1", NAME4). NAME4 won't be computed (issuing the error of division by zero)

\*if any column in BIN contains less than 2 ones.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE. Basic statistics for a quantitative variable, by groups.

matrix.

get var /variable= v1.

get group /variable= gr. /\*Grouping variable

!KO\_freq(group%1%dummy%freq%codes).

!KO\_gdescr(var%dummy%min%max%mean%variance).

print {codes,freq,min,max,mean,variance}

/clabels= 'group' 'n' 'min' 'max' 'mean' 'variance'.

end matrix.

### VARIOUS UNIVARIATE STATISTICS, BY GROUPS [!KO\_aggr]

\*/\*!KO\_aggr(data%bin%stat%name)\*/\*.

\*Version 1.

\*Computes the required descriptive statistic in the columns of DATA, in total and/or by groups

\*of cases (the rows).

\*This function resembles out-of-MATRIX command AGGREGATE.

\*Takes data DATA (matrix with any number of columns, variables) and one or more binary (1 vs 0)

\*variables constituting columns of matrix BIN with number of rows as in DATA.

\*Each column of BIN represents a group, and value 1 in it means that the case (row) belongs to it.

\*Groups may intersect or be disjoint by the composition of cases (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros, in which case the group corresponds to the

\*whole dataset, all its rows: statistic "in total" will be computed.

\*Result NAME - matrix with columns corresponding to columns of DATA, i.e. the variables, and with rows

\*corresponding to columns of BIN, i.e. the groups. NAME contains values of the computed statistic for

\*each variable in each group.

\*STAT - the requested statistic. It is a keyword in capital letters. Quotes or apostrophes around the

\*keyword are optional. The following statistics are available:

\*"SUM" - sum

\*"MEAN" - arithmetic mean

\*"SSDEV" - sum of squared deviations from the mean

\*"VARIANCE" - variance [sum in every column of BIN must be >1]

\*"VARIANCE2" - variance, computed on "df=n", not "df=n-1"

\*"SEMEAN" - st. error of the arithmetic mean [sum in every column of BIN must be >1]

\*"MIN" - minimal value

\*"MAX" - maximal value

\*"FIRST" - first value

\*"LAST" - last value

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

\*If you want to replace, by the computed statistic, the values of DATA, use function /\*!KO\_aggrv\*/.

EXAMPLE. Mean in total and by groups.

matrix.

get vars /variables= v1 v2 /names= names.

get grvar /variable= group. /\*Categorical grouping variable

!KO\_freq(grvar%1%dummy%freq%codes). /\*Create dummy variables

compute dummy= {make(nrow(dummy),1,1),dummy}. /\*Add constant 1 as the 1st column

!KO\_aggr(vars%dummy%'MEAN'%table). /\*Compute means

compute names= {'Group',names}.

print {{9999;codes},table} /format= f8.4 /cnames= names

/title= 'Means: Groups X Variables table (Group=9999 is Total sample):'.

end matrix.

EXAMPLE. Count number of values, satisfying some condition, in groups.

matrix.

get vars /variables= v1 v2 /names= names. /\*Get some variable(s)

get grvar /variable= group. /\*Categorical grouping variable

!KO\_freq(grvar%1%dummy%freq%codes). /\*Create dummy variables

print codes /title 'Group codes'.

compute flag= vars>.2 and vars<.9. /\*Compute binary matrix which flags (by 1s)

/\*values in VARS which are >.2 but <.9 (some condition)

!KO\_aggr(flag%dummy%'SUM'%table). /\*Compute sum of flagged elements

print table /cnames= names

/title 'Count, by groups, of values which are >.2 but <.9:'.

end matrix.

### VALUE CLOSEST TO A SPECFIED ONE, FROM BELOW OR ABOVE [!KO\_closest]

\*/\*!KO\_closest(data%ineq%val%name)\*/\*.

\*Version 1.

\*In each column of data DATA, finds out the observed value closest to value VAL by size;

\*and condition INEQ stipulates whether it must be lesser or greater than VAL.

\*INEQ is inequality operator, keyword (may put in quotes or apostrophes)

\*to choose: >, >=, <, <= (or letters: GT, GE, LT, LE). For instance, <= means to identify the greatest

\*observed value which is <=VAL, i.e., it adjoins it from "below" and is either less or equals it.

\*VAL - scalar or row vector lengthed as the number of columns of DATA. Vector means specifying

\*a separate value VAL for each column of DATA.

\*Result is row NAME containing the sought-for values. Value 1E+300 or 1E-300 in NAME says that

\*in the given column of DATA there was found no value satisfying condition INEQ VAL.

EXAMPLE.

matrix.

compute data= rnd((uniform(8,5)-.3)\*15). /\*Some random data, 5 variables

print data /format f8.

!KO\_mean(data%mean).

print mean /format f8.3 /title 'Column means'.

!KO\_closest(data%GT%mean%name).

print name /format f8 /title 'Smallest values that are Greater Than the means'.

!KO\_closest(data%LE%0%name).

print name /format f8 /title 'Largest values that are Lesser or Equal to 0'.

end matrix.

### COVARIANCE MATRIX [!KO\_cov]

\*/\*!KO\_cov(data%name)\*/\*.

\*Version 1.

\*Computes covariances between colums of DATA.

\*Returns square symmetric matrix NAME.

EXAMPLE.

matrix.

get vars /variables= v1 v2 v3 /names= names.

!KO\_cov(vars%cov).

print cov /rnames= names /cnames= names.

end matrix.

### COVARIANCE MATRIX (DF=N) [!KO\_cov2]

\*/\*!KO\_cov2(data%name)\*/\*.

\*Version 1.

\*Computes covariances between colums of DATA. The values are computed on "df=n" (not on "df=n-1").

\*Returns square symmetric matrix NAME.

### CORRELATION MATRIX [!KO\_corr]

\*/\*!KO\_corr(data%name)\*/\*.

\*Version 1.

\*Computes Pearson correlations between colums of DATA.

\*Returns square symmetric matrix NAME.

### COSINE MATRIX [!KO\_cosine]

\*/\*!KO\_cosine(data%name)\*/\*.

\*Version 1.

\*Computes cosine similarities (coefficient of proportionality or Tucker's coefficient

\*of congruence) between columns of DATA.

\*Returns square symmetric matrix NAME.

### MATRIX OF COEFFICIENTS OF IDENTITY [!KO\_idc]

\*/\*!KO\_idc(data%name)\*/\*.

\*Version 1.

\*Computes coefficients of identity [Zegers, ten Berge, 1985]

\*between columns of DATA.

\*Returns square symmetric matrix NAME.

### MATRIX OF SIMILARITY RATIOS [!KO\_simr]

\*/\*!KO\_simr(data%name)\*/\*.

\*Version 1.

\*Computes similarity ratio coefficients, also known as Kohonen similarity,

\*between columns of DATA.

\*Returns square symmetric matrix NAME.

### COVARIANCE, CORRELATION, COSINE, COEFFICIENT OF IDENTITY, SIMILARITY RATIO [!KO\_biv]

\*/\*!KO\_biv(col1%col2%name)\*/\*.

\*Version 2.

\*For two column vectors COL1 and COL2 (equal length) computes row of five statistics of similarity:

\*coefficient of covariation (1st element of NAME), Pearson correlation (2nd element), coefficient of

\*proportionality (cosine, 3rd element), coefficient of identity (4th element),

\*similarity ratio (5th element).

### RECTANGULAR MATRIX OF COVARIANCES/CORRELATIONS/COSINES [!KO\_rect]

\*/\*!KO\_rect(vars1%vars2%type%name)\*/\*.

\*Version 1.

\*Computes requested association coefficent between columns of VARS1 and columns of VARS2.

\*These two matrices must have the same number of rows (rows are cases, and columns are variables).

\*Returns rectangular matrix NAME sized num\_col\_VARS1 x num\_col\_VARS2.

\*TYPE - requested association measure, capitalized keyword (optionally may put in quotes or

\*apostrophes):

\*"COV" - coefficient of covariance (on "df=n-1")

\*"COV2" - coefficient of covariance (on "df=n")

\*"CORR" - Pearson correlation coefficient

\*"COSINE" - cosine similarity (Tucker's coefficient of congruence).

### COVARIANCE/CORRELATION MATRIX, BY GROUPS AND POOLED [!KO\_gcov]

\*/\*!KO\_gcov(data%bin%type%out%name)\*/\*.

\*Version 2.

\*Computes scatter, covariance, or correlation matrix of variables by groups

\*of cases, and also the pooled (across the groups) matrix.

\*Takes data DATA (matrix with p columns-variables, p>=1) and one or more binary (1 vs 0)

\*variables constituting columns of matrix BIN with number of rows as in DATA.

\*Each column of BIN represents a group, and value 1 in it means that the case (row) belongs to it.

\*Groups may intersect or be disjoint by the composition of cases (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros, in which case the group corresponds to the

\*whole dataset, all its rows: matrix for the total sample will be computed.

\*Arguments TYPE and OUT is a keyword in capital letters. Quotes or apostrophes around

\*the keyword are optional.

\*TYPE - the requested type of matrix. The following types are available:

\*"SCAT" - scatter matrix

\*"COV" - covariance matrix [warning: sum in every column of BIN must be >1]

\*"COV2" - covariance matrix, computed on "df=n", not "df=n-1"

\*"CORR" - correlation matrix [warning: sum in every column of BIN must be >1]

\*OUT - this argument determines the composition of the returned result NAME:

\*"BYGR" - matrices, own for each group, column of BIN; the matrices are stacked, one below another,

\*their order follows the order of BIN's columns

\*"POOL" - pooled matrix of the groups

\*"BOTH" - both this and that (the pooled matrix will be the upper one in the stack).

\*What the pooled matrix is in case of different matrix type:

\*- the pooled scatter matrix is the summed scatter matrix of the groups;

\*- the pooled covariance matrix is the weighted averaged group covariance matrix, where weights are the

\*number of the degrees of freedom in a group ("n-1" with COV or "n" with COV2, where n is the frequency

\*in a group);

\*- the pooled correlation matrix is obtained from the pooled covariance matrix

\*(or the pooled scatter matrix), not by averaging the group correlation matrices.

\*When computing a pooled matrix of any type the function takes for the total sample frequency

\*the sum of frequencies in groups, i.e. number of ones in BIN, - and not the number of rows in BIN

\*(for in BIN some rows might be empty or might count in more than one group).

\*Because, if DATA has only one column, COV (or COV2) matrix turns in a scalar, the variance,

\*you may use the current function also to compute the pooled group variance.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE.

set mxloops 1E9.

matrix.

get vars /variables= v1 to v5.

get group /var= gr. /\*Categorical grouping variable

!KO\_freq(group%1%dummy%freq%codes). /\*Create dummy variables

!KO\_gcov(vars%dummy%COV%BOTH%m).

print m(1:ncol(m),:) /title 'Pooled within-group covariance matrix'.

print m((ncol(m)+1):nrow(m),:)

/title 'Within-group covariance matrices, stacked one under another'.

print codes /title 'The group codes'.

end matrix.

### BETWEEN-GROUP AND POOLED WITHIN-GROUP SCATTER MATRICES [!KO\_bwscat]

\*/\*!KO\_bwscat(data%dummy%name1%name2)\*/\*.

\*Version 1.

\*Takes data DATA and groups DUMMY and returns between-group scatter matrix NAME1 and

\*pooled (summed) within-group scatter matrix NAME2.

\*(SSbetween = trace(NAME1) and SSwithin = trace(NAME1).)

\*DATA - data with any number of columns (variables).

\*NAME1 (attention!) depends on whether DATA has centered

\*columns or not, but NAME2 does not depend (except for numerical stability issue, which is

\*better under centered data).

\*DUMMY - binary dummy (indicator) variables marking groups (each variable corresponds to a group,

\*with value 1 = the case belongs to it, 0 = doesn't belong to it).

\*Each column in DUMMY must contain at least one 1.

\*The groups must be disjoint by membership: sum in each row in DUMMY must not exceed 1.

\*If DUMMY has rows complete of zeros (cases out of the groups) then NAME2 won't be correct.

\*The returned matrices sum into the sscp matrix of variables of the whole dataset: NAME1+NAME2=sscp(DATA).

\*You can create dummy variables DUMMY out of a categorical grouping variable

\*with the help of function /\*!KO\_freq\*/.

\*If you need a pooled within-group scatter matrix from groups which intersect by case membership,

\*use function /\*!KO\_gcov\*/.

EXAMPLE. Some principal one-way MANOVA statistics.

matrix.

get vars /variables= v1 to v5.

get group /var= gr. /\*Categorical grouping variable

!KO\_freq(group%0%dummy%freq%codes). /\*Create dummies

!KO\_center(vars%vars). /\*Center the data

!KO\_bwscat(vars%dummy%b%w).

print (det(w)/det(b+w)) /title "Wilks' lambda".

print trace(inv(w)\*b) /title "Hotelling's trace".

print trace(b\*(inv(b+w))) /title "Pillai's trace".

end matrix.

### IMAGE OR ANTI-IMAGE MATRIX [!KO\_image]

\*/\*!KO\_image(cov%type%name)\*/\*.

\*Version 1.

\*Takes covariance or correlation matrix COV and returns as NAME the image

\*(if TYPE is positive scalar) or anti-image (if TYPE is nonpositive scalar) covariance matrix.

\*COV must be square symmetric nonsingular, normally - positive definite.

\*TYPE is scalar.

\*"Images" stand on the diagomal of image matrix, equal to Rsq\*Ssq, where Rsq is the squared multiple

\*correlation for the variable as dependent on all the other, аnd Ssq is the diagonal element of COV.

\*Off-diagonal elements of anti-image matrix are the (negated) coefficients of partial covariance.

\*If you negate the sign of off-diagonal elements of the anti-image matrix and add it with the

\*image matrix, the initial matrix COV would be restored.

\*This function might be used for any sscp-type matrix: correlation, covariance, cosine, raw sscp.

### INFLUENCE MATRIX (HAT MATRIX) [!KO\_hat]

\*/\*!KO\_hat(data%name)\*/\*.

\*Version 1.

\*Takes data matrix DATA which must be with noncollinear columns.

\*If necessary do prior centration or standardization of DATA columns.

\*Returns influence matrix (hat matrix, projection matrix) NAME.

\*Values on its diagonal are known as leverages.

\*If DATA (having n rows) has centered columns then NAME\*(n-1) is the double centration

\*(see function /\*!KO\_dcenter\*/) of matrix of squared Mahalanobis distances between rows of DATA,

\*and leverages\*(n-1) are squared Mahalanobis distances to centroid of DATA.

### CO-COUNTS MATRICES FOR BINARY DATA [!KO\_bincnt]

\*/\*!KO\_bincnt(data%name1%name2)\*/\*.

\*Version 1.

\*Takes matrix of binary (values 0 and 1) data DATA and returns two square matrices with co-counts

\*between columns of DATA:

\*NAME1 - symmetric matrix which element ij shows the number of rows in DATA where column i

\*contains 1 and column j contains 1. These counts are known as counts A (characteristic is in both columns).

\*NAME2 - asymmetric matrix which element ij shows the number of rows in DATA where column i

\*contains 0 and column j contains 1. These counts are known as counts B (characteristic present in one

\*and is absent in the other column).

\*Counts C matrix is transposed B.

\*Counts D (characteristic is absent on both columns) = number\_of\_rows\_in\_DATA-A-B-C.

\*Counts A, B, C, D are useful for computation of various proximity measures invented for binary data.

EXAMPLE. Compute between columns 1) Jaccard similarity and 2) pattern difference.

matrix.

compute data= rnd(uniform(10,5)).

print data.

!KO\_bincnt(data%a%b).

compute c= t(b).

compute d= nrow(data)-a-b-c.

compute jaccard= a/(a+b+c).

print jaccard /title 'Jaccard similarity matrix '.

compute pattern= (b&\*c)/(a+b+c+d)&\*\*2.

print pattern /title 'Pattern difference matrix '.

end matrix.

### MATRIX OF SQUARED EUCLIDEAN DISTANCES [!KO\_seuclid]

\*/\*!KO\_seuclid(data%name)\*/\*.

\*Version 2.

\*Computes square euclidean distances between rows of DATA.

\*Returns square symmetric matrix NAME.

\*If the number of columns of DATA >1, the function computes the distances via scalar products;

\*this is more fast, but calculatory potentially less precise mode than the "direct" mode

\*via raising differences to square (if you need "direct mode", use more general

\*function /\*!KO\_pwmink\*/).

EXAMPLE.

matrix.

get data /variables= v1 v2 v3.

!KO\_seuclid(data%dist).

save dist /outfile= \*.

end matrix.

### RECTANGULAR MATRIX OF SQUARED EUCLIDEAN DISTANCES [!KO\_seuclidr]

\*/\*!KO\_seuclidr(cases1%cases2%name)\*/\*.

\*Version 2.

\*Computes square euclidean distances between rows of CASES1 and rows of CASES2.

\*These two matrices must have the same number of columns (columns are data dimensions).

\*Returns rectangular matrix NAME sized num\_rows\_CASES1 x num\_rows\_CASES2.

\*If the number of columns of data >1, the function computes the distances via scalar products;

\*this is more fast, but calculatory potentially less precise mode than the "direct" mode

\*via raising differences to square.

### MATRIX OF POWERED WEIGHTED MINKOWSKI DISTANCES [!KO\_pwmink]

\*/\*!KO\_pwmink(data%power%weights%name)\*/\*.

\*Version 1.

\*Computes Minkowski distances (raised to power POWER) between rows of DATA.

\*Returns square symmetric matrix NAME.

\*POWER - scalar (typically >=1); this is the power for the distance.

\*WEIGHTS - vector (row or column) lengthed as the number of columns of DATA and containing positive numbers;

\*these are weights for columns of DATA. Only ratio between the weights size matters because the function

\*normalizes them. If all weights are equal there will come out unweighted distance.

\*The distance between rows a and b = Sum(w\_i\*|a\_i-b\_i|^p), where a\_i, b\_i are values in column i,

\*p is power, w\_i is weight of column i, Sum is sum across all columns.

\*If POWER=2 the function will return (weighted) squared euclidean distances.

\*If POWER=1 the function will return (weighted) Manhattan distances.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Weighted Manhattan distance.

set mxloops 10000.

matrix.

get data /variables= v1 v2 v3.

!KO\_pwmink(data%1%{1,2,1}%dist).

save dist /outfile= \*.

end matrix.

### RECTANGULAR MATRIX OF POWERED WEIGHTED MINKOWSKI DISTANCES [!KO\_pwmink]

\*/\*!KO\_pwminkr(cases1%cases2%power%weights%name)\*/\*.

\*Version 1.

\*Computes Minkowski distances (raised to power POWER) between rows of CASES1 and rows of CASES2.

\*These two matrices must have the same number of columns (columns are data dimensions).

\*Returns rectangular matrix NAME sized num\_rows\_CASES1 x num\_rows\_CASES2.

\*Arguments POWER and WEIGHTS are the same as in function /\*!KO\_pwmink\*/.

\*If POWER=2 the function will return (weighted) squared euclidean distances.

\*If POWER=1 the function will return (weighted) Manhattan distances.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### MATRIX OF SQUARED MAHALANOBIS DISTANCES [!KO\_smahal]

\*/\*!KO\_smahal(data%cov%name)\*/\*.

\*Version 1.

\*Computes square Mahalanobis distances between rows of DATA (matrix n x p, p>=1).

\*Returns square symmetric matrix NAME.

\*Argument COV - either square p x p symmetric nonsingular matrix that is a user-specified

\*covariance matrix between the variables, or scalar 0 - then matrix COV will

\*be set equal to the observed covariance matrix between the p variables of DATA.

\*Then DATA must have more rows than columns and the columns must be noncollinear.

\*The distance formula: d^2 = (x-y)'inv(COV)(x-y).

\*If the covariance matrix is identity matrix, the result are squared euclidean distances.

\*If the input COV is a correlation matrix, the unscaled squared Mahalanobis distances

\*are obtained.

\*If you have a user-specified inv(COV) instead of COV, you can use function /\*!KO\_sbutler\*/

\*to get Mahalanobis distances.

### VECTOR OF SQUARED MAHALANOBIS DISTANCES TO CENTROID [!KO\_smahalc]

\*/\*!KO\_smahalc(data%mean%cov%name)\*/\*.

\*Version 2.

\*Computes squared Mahalanobis distances between rows of DATA and a centroid.

\*Returns column vector NAME with number of rows as DATA has.

\*DATA must have p (p>=1) columns and any number of rows. It is coordinates of points in

\*p-dimensional space.

\*If you want to get distances to the centroid of the observed DATA cloud itself, specify

\*argument COV as zero scalar. DATA will then need to have at least two rows and

\*noncollinear columns. Argument MEAN will be ignored.

\*If you want to get distances of DATA points to the centroid of some other cloud in the same space,

\*then you have to specify vector of means MEAN and covariance matrix COV of that cloud:

\*MEAN must have p columns and 1 row. These are coordinates of the centre of that cloud.

\*COV must be p X p, symmetric nonsingular. It is the covariances in that cloud.

\*If you want to get distance between two groups from the same population, enter row DATA = centroid

\*of one group, row MEAN = centroid of the other group, COV = pooled within-group covariance matrix

\*(you can obtain the latter by function /\*!KO\_gcov\*/).

\*If COV is identity matrix, the result are squared euclidean distances from the points to

\*centroid MEAN.

\*If COV is nongramian (has negative eigenvalues), some distances will come out negative.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Mahalanobis distances from cloud’s points to the centroid of that same cloud.

matrix.

get data /variables= v1 v2 v3.

!KO\_smahalc(data%0%0%dist).

print sqrt(dist).

end matrix.

### MATRIX OF SQUARED BUTLER DISTANCES [!KO\_sbutler]

\*/\*!KO\_sbutler(data%cos%name)\*/\*.

\*Version 1.

\*Computes square Butler's distances between rows of DATA (matrix n x p, p>=1).

\*Returns square symmetric matrix NAME.

\*"Butler's" distances are euclidean distances not in cartesian (orthogonal) but in

\*oblique system, where the angle cosines between the axes are equal to the correlations

\*between the variables or are specified by the user; and input data are taken for skew

\*(contravariant) coordinates in the system. The formula: d^2 = (x-y)'COS(x-y).

\*Argument COS - either square p x p symmetric matrix of angle cosines between the axes

\*(this is user-specification of the cosines), or a scalar 0 - then the matrix of angle

\*cosines COS will be set equal to the observed correlation matrix between the p

\*variables of DATA.

\*If the angle cosines matrix is identity matrix, the result are squared euclidean distances.

\*If as COS you input an inverted covariance matrix, the result are squared Mahalanobis distances,

\*and if an inverted correlation matrix - then unscaled squared Mahalanobis distances.

### IS THERE IDENTICAL VALUES IN A MATRIX

Use for this expression any(grade(mat)<>rnkorder(mat)) which returns 1 if there are duplicates in MAT, and returns 0 if all its values are different.

### LIST OF UNIQUE VALUES OF VECTOR AND POSITION OF THEIR FIRST OCCURRENCE [!KO\_unique]

\*/\*!KO\_unique(vec%ignore%name1%name2)\*/\*.

\*Version 3.

\*Takes vector (column or row) VEC and returns row vector NAME1, which is

\*the list of unique values in VEC; the values go in the order of their first occurrence in VEC.

\*The positions of the first occurrence are returned as the corresponding row NAME2.

\*Argument IGNORE - digit (not name or expression, optionally in quotes or apostrophes): if 1,

\*value 0 in VEC is ignored (considered nonvalid); if 0, zero is considered valid value.

\*Result NAME2 in the form of scalar 0 under IGNORE=1 means that VEC consisted of just zeros.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

compute var= {4;0;8;8;0;-1;8;6;3}.

print var.

!KO\_unique(var%0%vals%pos).

print {vals;pos}.

end matrix.

EXAMPLE. Positions of last (not first) occurrences of values.

matrix.

compute var= {4;0;8;8;0;-1;8;6;3}.

print var.

!KO\_unique(var(nrow(var):1)%0%vals%pos).

print {vals;nrow(var)-pos+1}.

end matrix.

### FREQUENCIES OF SPECIFIED VALUES IN COLUMNS [!KO\_freqval]

\*/\*!KO\_freqval(data%vals%name)\*/\*.

\*Version 1.

\*Takes matrix DATA consisting of one or more columns, and list of values VALS (scalar or vector),

\*and returns frequency of each of these values in each column of DATA. Result NAME is matrix of the

\*frequencies; its rows correspond to the values VALS and its columns correspond to the columns of DATA.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### TWO-WAY FREQUENCY CROSSTABULATION FOR SPECIFIED VALUES [!KO\_ctabval]

\*/\*!KO\_ctabval(rcol%ccol%rvals%cvals%name)\*/\*.

\*Version 1.

\*Takes column vectors RCOL and CCOL which must be of equal length, and lists of values: RVALS (for RCOL)

\*and CVALS (for CCOL). RVALS and CVALS are vectors (or scalars), their length may be different.

\*Returns frequency cross-table NAME of size r x c with rows defined by the r values RVALS and columns

\*defined by the c values CVALS.

\*Counts in the table are the number of cases with corresponding combination of values between RCOL and CCOL.

EXAMPLE.

matrix.

compute x= {1;2;2;0;0;0;1;1;2;1;0;2;2;2;2;0;0;1;0;0;0;1;1;0;1;1;2;1;2;2;3;2;2;3;9;2}.

compute y= {0;1;0;1;1;1;1;1;2;1;1;1;1;1;3;2;1;2;2;2;9;2;1;2;2;2;2;2;1;2;2;2;2;3;2;2}.

print {x,y}.

!KO\_ctabval(x%y%{0,1,3,9}%{1,2}%tab).

print tab /title 'X by Y frequency crosstabulation for the requested lists'

+' of values: (0,1,3,9) by (1,2).'.

end matrix.

### DUMMY VARIABLES AND FREQUENCIES OF VALUES OF A COLUMN [!KO\_freq]

\*/\*!KO\_freq(col%sort%name1%name2%name3)\*/\*.

\*Version 2.

\*Takes column vector COL which must not be a constant and returns:

\*matrix NAME1 of dummy variables, one column for each unique value of COL;

\*column vector NAME2 with frequencies of unique values of COL (it's sums within columns of NAME1);

\*column vector NAME3 - list of unique values of COL.

\*Argument SORT (scalar) - whether to sort the results ascendingly by the values: if positive argument,

\*columns in NAME1 and rows in NAME2 and NAME3 will be sorted in ascending order of the unique values

\*of COL; but if the argument is nonpositive then the order of rows and columns in the results is

\*determined by the sequence of encounter of the unique values of COL (screening COL top-down).

\*This function may require computer to have a lot of RAM memory in case of massive data.

EXAMPLE.

matrix.

compute var= {4;0;8;8;0;-1;8;6;3}.

print var.

!KO\_freq(var%1%dummy%freq%vals).

print {vals,freq} /clabels= 'Value' 'Freq'.

print dummy.

end matrix.

### TWO-WAY FREQUENCY CROSSTABULATION [!KO\_crosstab]

\*/\*!KO\_crosstab(rcol%ccol%sort%name1%name2%name3)\*/\*.

\*Version 2.

\*Takes column vectors RCOL and CCOL which must not be constants and must be of the same length.

\*Returns frequency cross-table NAME1 in which the rows are defined by unique values of RCOL

\*and the columns are defined by unique values of CCOL. The lists of the unique values themselves

\*are returned as column vector NAME2 (values of RCOL) and row vector NAME3 (values of CCOL).

\*Argument SORT (scalar) - whether to sort the results ascendingly by the values: if positive argument,

\*rows in NAME1 and NAME2 will be sorted in ascending order of the unique values of RCOL and columns in

\*NAME1 and NAME3 will be sorted in ascending order of the unique values of CCOL;

\*but if the argument if nonpositive then the order of rows and columns in the results is determined

\*by the sequence of encounter of the unique values in RCOL and CCOL, respectively

\*(screening these columns top-down).

\*This function may require computer to have a lot of RAM memory in case of massive data.

EXAMPLE.

matrix.

get var1 /variable= v1.

get var2 /variable= v2.

!KO\_crosstab(var1%var2%1%tab%vals1%vals2).

print tab.

print vals1.

print vals2.

end matrix.

### INTERACTION DUMMY VARIABLES AND MULTI-WAY FREQUENCY AGGREGATION [!KO\_aggrtab]

\*/\*!KO\_aggrtab(cols%sort%empty%name1%name2%name3)\*/\*.

\*Version 1.

\*Does frequency crosstabulation for any number of variables.

\*Returns results in the form of aggregated set (not in the form of cross-tables).

\*(Same way out-of-matrix command AGGREGATE returns frequencies.)

\*The function also returns binary dataset of dummy variables corresponding to the interaction of

\*the highest order.

\*COLS - categorical data where variables are the columns; there must be at least two columns and they

\*must not be constants.

\*The function returns frequency column vector NAME2 where each element corresponds to one combination

\*of values from all the variables, i.e. it is the frequency in a cell of multi-way table defined by

\*the variables.

\*Variable values' combinations themselves are contained in NAME3 where rows correspond to rows of NAME2

\*and columns correspond to variables, columns of COLS.

\*Vector NAME2 is the sums in columns of the returned dataset NAME1, which is the dummy variables: they

\*correspond to the above combinations of unique values. I.e. columns of NAME1 correspond to rows

\*of NAME2 and NAME3.

\*Argument SORT (scalar) - whether to sort results ascendingly by the values: if positive, then

\*columns in NAME1 and rows in NAME2 and NAME3 will be sorted in ascending order of the unique values

\*of the variables; but if nonpositive, then the mentioned order in the results is determined by the

\*sequence of encounter of the unique values in data (screening columns COLS top-down).

\*Argument EMPTY (scalar) - if positive, columns (in NAME1) and rows (in NAME2 and NAME3) with frequency 0

\*in NAME2 will be omitted in the results. If the argument is nonpositive, results will be returned in full.

\*This function may require computer to have a lot of RAM memory in case of massive data.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Three-variable frequency crosstabulation (aggregation) in MATRIX and by AGGREGATE.

set mxloops 10000.

matrix.

get data /variables= gender agegroup marriage.

!KO\_aggrtab(data%1%1%dummy%freq%codes).

print {codes,freq} /clabels= 'gender' 'agegr' 'marri' 'n'.

end matrix.

dataset declare aggr.

aggregate /outfile= 'aggr' /break= gender agegroup marriage /freq= n.

### INTERACTION DUMMY VARIABLES AND MULTI-WAY FREQUENCY AGGREGATION (DICHOTOMOUS DATA) [!KO\_baggrtab]

\*/\*!KO\_baggrtab(cols%empty%name1%name2%name3)\*/\*.

\*Version 1.

\*Does frequency crosstabulation for any number of dichotomous variables.

\*The variables must be coded binary (1 vs 0); if you have other values, make them binary.

\*This function is equivalent to /\*!KO\_aggrtab\*/ but it is for all variables COLS

\*binary (0 vs 1). It is faster than /\*!KO\_aggrtab\*/ and spares memory.

\*Argument EMPTY and results NAME1, NAME2, NAME3 - see in the description of /\*!KO\_aggrtab\*/.

\*The order of values in results is always sorted as 0 1, so there is no argument SORT.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### ELEMENTARY VARIABLES AND FREQUENCIES IN A CATEGORICAL MULTIPLE RESPONSE SET [!KO\_mrfreq]

\*/\*!KO\_mrfreq(mrc%dupl%sort%name1%name2%name3)\*/\*.

\*Version 1.

\*This function is like /\*!KO\_freq\*/ - returns binary variables and frequencies of values - but as

\*input it takes not one categorical variable but a set of those; those variables having common

\*pool of values. It is "categorical multiple response set". If each case is a respondent then he

\*might have thus selected several, not just single, responses.

\*MRC - input data which columns are the categorical variables constituting the mentioned set.

\*Zero in MRC is considered the "filler", i.e. empty cell. Any other value is a valid value.

\*It is allowed that some rows (cases) or some columns (variables) be empty.

\*Argument DUPL (scalar) - if positive, each unique value counts in a MRC row no more than once,

\*i.e. repeating of the same response by the same respondent, if encounters, is ignored;

\*but if the argument is nonpositive, then all repetition by a respondent get counted as valid.

\*The function returns:

\*NAME1 - matrix, one column for each unique valid value of MRC. Entry (i,j) in NAME1

\*is how much times respondent i gave response j (if DUPL nonpositive), or 0 or 1

\*(if DUPL positive); thus under positive DUPL NAME1 is the "dichotomous (binary) multuple

\*response set".

\*NAME2 - column vector with frequencies of unique values of MRC (it's sums within columns of NAME1);

\*NAME3 - column vector, list of unique values of MRC.

\*Argument SORT (scalar) - whether to sort the results ascendingly by the values: if positive argument,

\*columns in NAME1 and rows in NAME2 and NAME3 will be sorted in ascending order of the unique values

\*of MRC; but if the argument is nonpositive then the order of rows and columns in the results is

\*determined by the sequence of encounter of the unique values of MRC (screening MRC top-down, 1st col,

\*then 2nd col...).

\*This function may require computer to have a lot of RAM memory in case of massive data.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

get mrc /variables= v1 to v6 /miss= 0. /\*Categorical vars with common pool of values

!KO\_mrfreq(mrc%1%1%bin%freq%codes).

print {codes,freq}. /\*Values and their frequencies

save bin /out= \*. /\*Binary variables (each corresponding to a unique value)

end matrix.

### PERCENT OF CORRECT CLASSIFICATION [!KO\_classres]

\*/\*!KO\_classres(tab%name1%name2%name3).\*/\*.

\*Version 1.

\*Takes square frequency (count) k x k cross-table TAB displaying results of a classification

\*of cases into k classes. Table rows are the observed classes and table columns are these same

\*classes predicted, going in the same order. Thus, counts of the correct prediction

\*form the diagonal of the table. (You can obtain table TAB from the variables

\*"observed class" and "predicted class" by function /\*!KO\_crosstab\*/).)

\*Returns row NAME1 ("Overall Percent"), column NAME2 ("Percent Correct"), and scalar NAME3

\*(the corner value, "Overall Percent Correct").

EXAMPLE.

matrix.

get obs /variable= obs. /\*Grouping variable with (say) 2 classes

get pred /variable= pred. /\*Corresponding variable with these 2 classes predicted

/\*(both variables must have the same class codes

!KO\_crosstab(obs%pred%1%tab%name2%name3). /\*Produce square classification table:

/\*frequency crosstabulation with observed classes as rows and predicted –

/\*as columns;

/\*the counts of correctly classified cases is the diagonal of the table

/\*Note that argument SORT is set to a positive value (1)

print tab.

!KO\_classres(tab%overall%correct%ocorrect). /\*Assess the classification results

print {tab,correct;overall,ocorrect}

/rlabels= 'Class1' 'Class2' 'Overall%'

/clabels= 'Class1' 'Class2' '%Correct'

/title 'Classification table: Observed x Predicted classes'.

end matrix.

### COUNTS OF CONCORDANT, DISCORDANT, AND TIED PAIRS [!KO\_concdisc]

\*/\*!KO\_concdisc(col1%col2%name1%name2%name3%name4%name5)\*/\*.

\*Version 1.

\*Takes column vectors COL1 and COL2, which must not be constants and must be of the same length

\*(it is two scale or ordinal variables) and returns 5 scalar quantities necessary for computation

\*of such coefficients of association between ordinal variables as Goodman-Kruskal Gamma,

\*Somer's coefficient, Kendall correlation, etc.

\*These quantities:

\*NAME1 - number of concordant pairs (P);

\*NAME2 - number of discordant pairs (Q);

\*NAME3 - number of pairs tied by the 1st variable, COL1 (Tx);

\*NAME4 - number of pairs tied by the 2nd variable, COL2 (Ty);

\*NAME5 - number of pairs tied by both variables at once (Txy).

\*If COL1 and COL2 are categorical (not continuous) data, another function, /\*!KO\_concdisct\*/

\*calculating the same quantities may occur more fast.

EXAMPLE.

matrix.

get var1 /variable= v1.

get var2 /variable= v2.

!KO\_concdisc(var1%var2%p%q%t\_v1%t\_v2%t\_v12).

print {p,q,t\_v1,t\_v2,t\_v12}.

end matrix.

### COUNTS OF CONCORDANT, DISCORDANT, AND TIED PAIRS (FROM TABLE) [!KO\_concdisct]

\*/\*!KO\_concdisct(tab%name1%name2%name3%name4%name5)\*/\*.

\*Version 2.

\*Takes frequency cross-table TAB obtained for two scale or ordinal variables

\*(i.e. rows and columns in the table go in ascending order of the unique values of the variables)

\*and returns 5 scalar quantities necessary for computation of such coefficients of association

\*between ordinal variables as Goodman-Kruskal Gamma, Somer's coefficient, Kendall correlation, etc.

\*These quantities:

\*NAME1 - number of concordant pairs (P);

\*NAME2 - number of discordant pairs (Q);

\*NAME3 - number of pairs tied by the 1st variable, forming the table rows (Tx);

\*NAME4 - number of pairs tied by the 2nd variable, forming the table columns (Ty);

\*NAME5 - number of pairs tied by both variables at once (Txy).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Compute several association coefficients between two ordinal variables.

matrix.

get var1 /variable= v1.

get var2 /variable= v2.

!KO\_crosstab(var1%var2%1%tab%vals1%vals2).

print tab.

!KO\_concdisct(tab%p%q%t\_v1%t\_v2%t\_v12).

compute gamma= (p-q)/(p+q).

compute somer= (p-q)/((p+q+t\_v1+p+q+t\_v2)/2).

compute taub= (p-q)/sqrt((p+q+t\_v1)\*(p+q+t\_v2)).

print gamma /title 'Goodman-Kruskal Gamma'.

print somer /title 'Somer D (symmetric)'.

print taub /title 'Kendall tau-b correlation'.

end matrix.

### COMEMBERSHIP CONFUSION MATRIX FOR TWO PARTITIONS [!KO\_ccm]

\*/\*!KO\_ccm(part1%part2%name)\*/\*.

\*Version 1.

\*Comembership confusion matrix (or table) is such a 2x2 confusion matrix where the unit of count is a pair

\*of objects (not an object). This matrix is the basis of assessment of similarity of two classifications

\*(partitions) of the same objects.

\*Let there be n objects that were classified by two methods, I and II. Then in the comembership confusion

\*matrix cell (1,1) contains number of pairs in which both objects are met in a same class both in

\*classification I and in classification II; cell (1,2) contains number of pairs in which both objects

\*are met in a same class in classification I but not in classification II; cell (2,1) contains number of

\*pairs in which both objects are met in a same class in classification II but not in classification I;

\*cell (2,2) contains number of pairs in which both objects are not met in a same class neither in

\*classification I nor in classification II.

\*The sum in such table is equal to the number of object pairs, n(n-1)/2.

\*Input PART1 and PART2 are two arrays with the same number of rows (rows are the objects), representing

\*themselves as partitions/classifications I and II, respectively. The array may be either a column with

\*any values or several columns with binary (1 and 0) values. When classification is specified by a

\*single column the function takes it for a categorical variable: each unique value represents a class.

\*While when classification is specified by more than one column the function takes it for a set of binary

\*variables each representing a class, the binary values designating belonging to it

\*(1=belongs, 0=doesn't belong). Binary set can be any, for example may include rows or columns full of

\*zeros or of ones; i.e. a binary set can render not only classifications with classes disjoint by objects

\*membership, but any kind, including classifications with empty classes or objects not belonging to classes.

\*In case if your classification represents only one class, then specify it as two columns: one binary

\*(belongs - doesn't belong), and the other is empty, of zeros.

\*Result NAME - comembership confusion matrix (table).

EXAMPLE.

matrix.

get part1 /variables= groups1. /\*Categorical grouping: partition I

get part2 /variables= bin1 to bin4. /\*Set of binary variables: partition II

!KO\_ccm(part1%part2%ct).

print ct /title 'Confusion table for pairs of cases, I x II'

/rlabels 'InSameGr' 'NotSame' /clabels 'InSameGr' 'NotSame'.

end matrix.

### COUNT VALUES IN ROWS (EXACT MATCH) [!KO\_count1]

\*/\*!KO\_count1(data%vals%name)\*/\*.

\*Version 2.

\*Counts in the rows of DATA number of elements which match any of the values of list VALS.

\*Returns column NAME lengthed as the number of rows in DATA.

\*This function is similar to non-MATRIX command COUNT, where the values are specified in list.

\*VALS - vector (row or column) of any length. Values in the list may duplicate: it doesn't affect

\*the result, but it slows work.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### COUNT VALUES IN ROWS (HITTING INTO RANGE) [!KO\_count2]

\*/\*!KO\_count2(data%low%high%name)\*/\*.

\*Version 2.

\*Counts in the rows of DATA number of elements which lie in any of the ranges from LOW to HIGH (inclusive).

\*Returns column NAME lengthed as the number of rows in DATA.

\*This function is similar to non-MATRIX command COUNT, where the values are specified by range(s) from...to.

\*LOW and HIGH - two scalars or two vectors of equal length. Each pair of their correspondent elements

\*LOW(i) and HIGH(i) define a range; HIGH(i) should normally be >= LOW(i). Ranges may intersect.

\*In order to count all values <=HIGH, set LOW on a knowingly too small value.

\*In order to count all values >=LOW, set HIGH on a knowingly too large value.

\*If arguments LOW and HIGH are identical the function will return the same result as /\*!KO\_count1\*/.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Count values within cases and equivalent out-of-MATRIX command COUNT.

matrix.

get data /variables= v1 to v10.

!KO\_count2(data%{-1,6,7,12}%{3,8,9,12}%name).

print name.

end matrix.

count x= v1 to v10 (-1 thru 3, 6 thru 8, 7 thru 9, 12).

list x.

### CHI-SQUARE (PEARSON AND LIKELIHOOD RATIO) OF TWO-WAY CONTINGENCY TABLE [!KO\_chitab]

\*/\*!KO\_chitab(tab%type%name)\*/\*.

\*Version 1.

\*Computes chi-square statistic which tells about association of rows and columns in a two-way

\*contingency table. Returns scalar NAME.

\*Argument TYPE (scalar) - which version of the statistic to return: Pearson chi-square (TYPE

\*non-positive) or likelihood ratio chi-square (TYPE positive).

\*Table TAB must contain nonnegative numbers, and sums in its rows and columns

\*must be all nonzero.

### PROPORTIONS AND RESIDUALS IN TWO-WAY CONTINGENCY TABLE [!KO\_cells]

\*/\*!KO\_cells(tab%stat%name)\*/\*.

\*Version 1.

\*Takes two-way contingency table (for example frequency crosstable) TAB with

\*(normally) nonnegative elements and (necessarily) nonzero sums in rows and columns.

\*Returns table NAME of the same size, containing the required statistic in its cells.

\*The required statistic - argument STAT, keyword in capital letters:

\*"TABPR" - proportions of the table sum

\*"ROWPR" - proportions of the sum in rows

\*"COLPR" - proportions of the sum in columns

\*"EXPECT" - expected (under independence of rows and colunds) values

\*"RESID" - residuals

\*"SRESID" - standardized residuals

\*"ASRESID" - adjusted standardized residuals.

\*Quotes or apostrophes around the keyword are optional.

### MATRIX OF SQUARED CHI-SQUARE DISTANCES IN CONTINGENCY TABLE [!KO\_schitab]

\*/\*!KO\_schitab(tab%name)\*/\*.

\*Version 1.

\*Computes squared chi-square distances between rows of contingency table TAB.

\*Returns square symmetric matrix NAME.

\*TAB is a matrix with nonnegative elements, and sums in its rows and columns must

\*be all nonzero.

\*Chi-square distance between the rows is the weighted euclidean distance between the row proportions,

\*where the inverted sums in columns serve as weights. The (squared) distance between rows i и i' is:

\*N \* Sum\_c[1/Nj \* (Nij/Ni-Ni'j/Ni')^2], where Nij is the value in the cell, Ni is the sum in row i, Nj is

\*the sum in column j, N is the total sum, Sum\_c means sum across all columns.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### MATRIX OF CHI-SQUARES BETWEEN ROWS OF COUNT DATA PAIRWISELY [!KO\_schi2c]

\*/\*!KO\_schi2c(data%type%name)\*/\*.

\*Version 1.

\*Takes data DATA sized not less than 2x2 and with nonnegative values which are considered as frequencies.

\*Sums in rows and columns must be nonzero.

\*For each two rows of DATA pairwise, isolates table 2xNC (NC is number of cols in DATA) and computes

\*chi-square (or phi-square) Pearson statistic for the table. That statistic can be thought of a measure of

\*dissimilarity between the data rows: the chi-square distance (squared).

\*Returned is square symmetric matrix NAME of such distances between all rows of DATA.

\*These are same distances, only squared, as returned by command PROXIMITIES /MEASURE=CHISQ (or /MEASURE=PH2).

\*These "distances" are not to confuse with the chi-square distances given by function /\*!KO\_schitab\*/.

\*TYPE - scalar: if nonpositive, chi-square is computed; if positive, phi-square is computed

\*(chi-square is divided by the total frequency in 2xNC table), phi-square is insensitive to nonequality of

\*sums in rows of DATA.

\*If in a 2xNC table there encounters empty column the function ignores it and attempts to compute based on

\*the remaining columns.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

get data /vari= v1 to v4.

!KO\_schi2c(data%0%chi).

print sqrt(chi).

end matrix.

PROXIMITIES v1 to v4

/VIEW= CASE

/MEASURE= CHISQ

/STANDARDIZE= NONE.

### DISTANCES TO GROUP CENTROIDS [!KO\_dtoc]

\*/\*!KO\_dtoc(data%bin%name)\*/\*.

\*Version 1.

\*Takes data DATA with any number of rows (points) and columns (variables).

\*And takes one or more binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME - matrix points x groups containing squared euclidean distances between the points and

\*the group centroids.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE.

matrix.

get data /vari= v1 to v4.

get gr /var= gr. /\*Grouping variable

!KO\_freq(gr%1%bin%freq%codes). /\*Convert it to dummies

!KO\_dtoc(data%bin%sqdev).

print sqdev. /\*Sq. distances from all cases to all group centroids

!KO\_aggr(data%bin%MEAN%centr).

print centr. /\*Display group centroids (means): groups X variables

end matrix.

### DISTANCES BETWEEN GROUP CENTROIDS

Use function /\*!KO\_aggr\*/ to compute group means (centroids) and then compute euclidean distances between them by function /\*!KO\_seuclid\*/.

### DISTANCES TO GROUP CENTROIDS (COMPUTATION FROM DISTANCE MATRIX) [!KO\_dtocfrd]

\*/\*!KO\_dtocfrd(dis%bin%self%name)\*/\*.

\*Version 2.

\*Takes square symmetric matrix of distances between points DIS. The distances must be

\*squared euclidean; anyway the function will perceive them as squared euclidean.

\*And takes one or more binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME - matrix points x groups containing squared euclidean distances between the points and

\*the group centroids.

\*Argument SELF - digit 0 or 1 (not name or expression; you may surround the digit by quotes/apostrophes).

\*If 1, then at computation of the distance between point i and the centroid of the group to which it

\*belongs (and which contains other points too), point i participates in definition of the centroid.

\*If 0, then such point i does not participate in definition of the centroid: the point temporarily

\*is considered as an external point.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

### DISTANCES BETWEEN GROUP CENTROIDS (COMPUTATION FROM DISTANCE MATRIX) [!KO\_dbwcfrd]

\*/\*!KO\_dbwcfrd(dis%bin%name)\*/\*.

\*Version 1.

\*Takes square symmetric matrix of distances between points DIS. The distances must be

\*squared euclidean; anyway the function will perceive them as squared euclidean.

\*And takes no less than two binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME - square symmetric matrix of squared euclidean distances between centroids of

\*the groups of the points.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE.

matrix.

get data /vari= v1 to v10. /\*Variables

get clu /var= clu. /\*Categorical grouping variable

!KO\_freq(clu%1%bin%n%code). /\*Convert it to the set of dummy variables

!KO\_seuclid(data%d). /\*Squared euclidean distances between cases

!KO\_dbwcfrd (d%bin%d\_c).

print d\_c. /\*Squared euclidean distances between group centroids

end matrix.

### AVERAGE DISTANCES TO GROUPS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_datofrd]

\*/\*!KO\_datofrd(dis%bin%self%name)\*/\*.

\*Version 1.

\*Average distance from each point to points of each of the offered groups.

\*Takes square symmetric matrix of distances (dissimilarities) between points DIS.

\*It must be nonnegative matrix with zero diagonal.

\*And takes one or more binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME - matrix points x groups: element ij in it is the averaged distance between

\*point i and points of group j.

\*Argument SELF - digit 0 or 1 (not name or expression; you may surround the digit by quotes/apostrophes).

\*If 1, the distance of a point to itself (the zero) when the point belongs to the group - is counted, i.e.

\*it enters the averaging. If 0, the distance of a point to itself is not counted, i.e. doesn't enter the

\*averaging, except when the point is the only member of the group.

\*If your matrix DIS are similarities convert them first to dissimilarities by any method you like.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

### AVERAGE DISTANCES BETWEEN GROUPS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dabwfrd]

\*/\*!KO\_dabwfrd(dis%bin%self%name)\*/\*.

\*Version 1.

\*Distances of between-group average linkage, i.e. average distances between points from every two groups.

\*Takes square symmetric matrix of distances (dissimilarities) between points DIS.

\*It must be nonnegative matrix with zero diagonal.

\*And takes no less than two binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME - square symmetric matrix groups x groups containing averaged distances between

\*points of the different groups. Averaged distances within groups stand on the diagonal.

\*Argument SELF - digit 0 or 1 (not name or expression; you may surround the digit by quotes/apostrophes).

\*If 1, the distance of a point to itself (the zero) when the point belongs to the group (or to the both

\*groups) - is counted, i.e. it enters the averaging. If 0, the distance of a point to itself is not

\*counted, i.e. doesn't enter the averaging, except when the point is the only member of the group

\*(or of both groups, when the speach is about the different groups).

\*If your matrix DIS are similarities convert them first to dissimilarities by any method you like.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

### DISTANCES TO GROUP FARTHEST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dtoffrd]

\*/\*!KO\_dtoffrd(dis%bin%alsoind%name)\*/\*.

\*Version 1.

\*Distance from each point to its farthest neighbour of each of the offered groups.

\*Takes square symmetric matrix of distances (dissimilarities) between points DIS.

\*It must be nonnegative matrix with zero diagonal.

\*And takes one or more binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Argument ALSOIND - digit 0 or 1 (not name or expression; you may surround the digit by quotes/apostrophes).

\*If you need only distances to farthest neighbours, specify 0. Result NAME will be matrix

\*points x groups: element ij in it is the distance (a value from DIS) between point i and its farthest

\*neighbour of group j. If you want to know also the numbers of those farthest neighbours (their row indices in

\*DIS and BIN), specify ALSOIND as 1. Then there will be twice more columns in NAME: to the matrix with distances

\*to farthest neighbours there will be right-joined the matrix containing the numbers (indices) of those points

\*which are the farthest neighbours.

\*If there exist more than one farthest neighbour for point i in group j, the number of the last one of them

\*(i.e. with the greatest index) is shown.

\*Only if a group consists of single point the neighbour of it in that group is that point itself.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your matrix DIS are similarities convert them first to dissimilarities by any method you like.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE. Distance from every point to its farthest neighnour (no groups).

matrix.

get mx /vari= var1 to var100. /\*Dissimilarity matrix 100 cases

compute bin= make(nrow(mx),1,1). /\*There is no groups in this example,

/\*so simply create column of ones

!KO\_dtoffrd(mx%bin%1%name).

print name(:,1:ncol(bin))

/title 'Distance from every case to its farthest neighbour'.

print name(:,(ncol(bin)+1):ncol(name))

/title 'The case number of the farthest neighbour'.

end matrix.

### DISTANCES BETWEEN GROUP FARTHEST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dbwffrd]

\*/\*!KO\_dbwffrd(dis%bin%alsoind%name)\*/\*.

\*Version 1.

\*Complete-linkage distances between groups, i.e. maximal distance between points of every two groups.

\*Takes square symmetric matrix of distances (dissimilarities) between points DIS.

\*It must be nonnegative matrix with zero diagonal.

\*And takes no less than two binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Argument ALSOIND - digit 0 or 1 (not name or expression; you may surround the digit by quotes/apostrophes).

\*If you need only distances between farthest neighbours, specify 0. Result NAME will be square symmetric

\*matrix groups x groups: element ij in it is the distance (a value from DIS) between most apart from each

\*other points, one of group i, the other of group j. On the diagonal are the distances between two most

\*apart points inside each group. If you want to know also the numbers of those farthest neighbours (their

\*row indices in DIS and BIN), specify ALSOIND as 1. Then there will be twice plus one more columns in NAME:

\*to the matrix with distances between farthest neighbours there will be right-joined the matrix containing

\*the numbers (indices) of those points which are the farthest neighbours. In this right-joined matrix and

\*which will be asymmetric, element ij is the number of the point which is the farthest neighbour from the

\*group i side (i.e. the row group). For example, elements (2,4)=10 and (4,2)=6 mean that the maximal

\*distance between groups 2 and 4 is the distance between point 10 (it is from group 2) and point 6

\*(it is from group 4). As for the numbers of most far apart points inside each group, the index of one of

\*the two is on the diagonal of that right-joined matrix, while the index of the other is carried into the

\*separate column, which is joined to the right of all. Thus, if k is the number of columns of BIN

\*(number of groups) then with ALSOIND=1 NAME has 2\*k+1 columns.

\*A point may come to be its own neighbour only in case it is the only group member in both groups being

\*currently compared.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your matrix DIS are similarities convert them first to dissimilarities by any method you like.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE. Complete linkage distances (between farthest neighbours) between groups.

matrix.

get mx /vari= var1 to var100. /\*Dissimilarity matrix 100 cases

get clu /var= clu. /\*Categorical grouping variable (in this example, groups

/\*will be disjoint, defined by a categorical variable)

!KO\_freq(clu%1%dummy%n%code). /\*Convert it to the set of dummy variables

!KO\_dbwffrd(mx%dummy%1%name).

print name(:,1:nrow(name))

/title 'Distances between farthest neighbours between (and within) groups'.

print name(:,(nrow(name)+1):(2\*nrow(name)))

/title 'Case numbers of the farthest neighbours'.

print /title 'Case number of one farthest neighbour within groups'.

print name(:,ncol(name)) /space 0

/title '(the number of the other one is on the diagonal of the prev. table)'.

end matrix.

### DISTANCES TO GROUP NEAREST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dtonfrd]

\*/\*!KO\_dtonfrd(dis%bin%alsoind%name)\*/\*.

\*Version 1.

\*Distance of every point to its nearest neighbour of each of the offered groups.

\*This function is same as function /\*!KO\_dtoffrd\*/ but it outputs distances to nearest, not farthest,

\*neighbours.

### DISTANCES BETWEEN GROUP NEAREST NEIGHBOURS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dbwnfrd]

\*/\*!KO\_dbwnfrd(dis%bin%alsoind%name)\*/\*.

\*Version 1.

\*Single-linkage distances between groups, i.e. minimal distance between points of every two groups.

\*This function is same as function /\*!KO\_dbwffrd\*/ but it outputs distances between nearest, not farthest,

\*neighbours.

### DISTANCES TO GROUP MEDOIDS (COMPUTATION FROM DISSIMILARITY MATRIX) [!KO\_dtomfrd]

\*/\*!KO\_dtomfrd(dis%bin%name1%name2)\*/\*.

\*Version 1.

\*Distance of every point to the medoid of each of the offered groups.

\*(Medoid is the point in a group of points whose sum of distances to the group's points is minimal.)

\*Takes square symmetric matrix of distances (dissimilarities) between points DIS.

\*It must be nonnegative matrix with zero diagonal.

\*And takes no less than two binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME1 - matrix points x groups containing distances between the points and the group medoids.

\*NAME2 is a row showing indices of points that are the medoids in their groups.

\*If more than one point in a group appear to be medoids (such as, for example, in a group of 2 points),

\*the function appoints the point with the greater index to be the group's medoid.

\*The matrix of distances between group medoids is DIS(NAME2,NAME2).

\*If your matrix DIS are similarities convert them first to dissimilarities by any method you like.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

### WITHIN-GROUP SUMS OF SQUARES OF DEVIATIONS (COMPUTATION FROM DISTANCE MATRIX) [!KO\_sswfrd]

\*/\*!KO\_sswfrd(dis%bin%name)\*/\*.

\*Version 1.

\*Takes square symmetric matrix of distances between points DIS. The distances must be

\*squared euclidean; anyway the function will perceive them as squared euclidean.

\*And takes one or more binary (1 vs 0) variables forming matrix BIN with number of rows as in DIS.

\*Each column of BIN represents a group, and value 1 in it means that the point belongs to it.

\*In whole, groups may intersect or be disjoint by the composition of their points (in the latter case

\*BIN appears to be a set of dummy variables). Each column of BIN must contain at least one one.

\*A column may, as a specific case, contain no zeros.

\*Result NAME - column lenghted as the number of columns in BIN, showing sum of squares of deviations

\*from centroid (SSW) in each of these groups.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE. Computation of SSwithin from data and from distances, in the whole sample.

matrix.

get vars /variables= v1 v2 v3.

compute bin= make(nrow(vars),1,1). /\*One group, the whole sample

!KO\_aggr(vars%bin%SSDEV%ssdev).

print rsum(ssdev). /\*Multivariate SSwithin

!KO\_seuclid(vars%d). /\*Matrix of squared euclidean distances

!KO\_sswfrd(d%bin%ssw).

print ssw. /\*Multivariate SSwithin (same value)

end matrix.

### K NEAREST NEIGBOURS (WRITING OUT), VERSION "RANDOM" [!KO\_knnr]

\*/\*!KO\_knnr(dis%k%name1%name2)\*/\*.

\*Version 1.

\*Takes matrix of distances (dissimilarities) DIS (R x C) between row points and column points.

\*For each of the C column points, finds K nearest to it row points and writes out them

\*to NAME1 (their numbers, indices) and NAME2 (the distances to them). Elements in NAME1 and NAME2

\*are mutually corresponding, the neighbours go in a column top down in the order the distancing

\*increases from the column point.

\*DIS - rectangular or square matrix with nonnegative elements.

\*In case DIS represent not two but one set of points, with distances between each other,

\*DIS will be square symmetric and you might want to ignore the diagonal elements (distances

\*from points to own selves); then set on the diagonal some number knowingly greater than the rest

\*elements of the matrix.

\*K - number of nearest neighbours, positive scalar <=R (usually K<<R).

\*NAME1 and NAME2 have size K x C.

\*If DIS is discrete values, i.e., contain ties, the result may partly depend on random number

\*seed (see below). To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*This function is beneficial when K<<R. If K is great and approaches to R, beneficial (from the

\*speed p.o.w) may be using sorting /\*!KO\_sort\*/ of each column of DIS with subsequent selection of

\*the K first rows of the result.

\*Difference between /\*!KO\_knnr\*/ and /\*!KO\_knnp\*/.

\*These functions are equivalent if DIS is continuous values (but /\*!KO\_knnp\*/ is somewhat faster then).

\*If DIS is discrete values, i.e., with ties, these functions show their algorithmic discrepancy

\*as follows. If points pretending for the nearest neighbourhood appear to be more than K,

\*/\*!KO\_knnr\*/ will choose from the tied neighbours randomly, in order there to be left exactly K

\*nearest neighbours; but /\*!KO\_knnp\*/ will output all the nearest neighboours, counting more than K.

\*Thus, in case o discrete DIS the result of /\*!KO\_knnr\*/ may depend on the random number seed, while

\*/\*!KO\_knnp\*/ may return more neighbours for a point, than K. Example. Let K=2, and the nearest

\*neighbours of a point have distances 2 3 3 3 4 from it. /\*!KO\_knnr\*/ will return two nearest

\*neighbours - with distances 2 and 3, selecting one of the three "3" randomly. /\*!KO\_knnp\*/ will return

\*four nearest neighbours: 2 3 3 3.

EXAMPLE. See right below.

### K NEAREST NEIGBOURS (WRITING OUT), VERSION "PLUS" [!KO\_knnp]

\*/\*!KO\_knnp(dis%k%name1%name2)\*/\*.

\*Version 1.

\*Takes matrix of distances (dissimilarities) DIS (R x C) between row points and column points.

\*For each of the C column points, finds K nearest to it row points and writes out them

\*to NAME1 (their numbers, indices) and NAME2 (the distances to them). Elements in NAME1 and NAME2

\*are mutually corresponding, the neighbours go in a column top down in the order the distancing

\*increases from the column point.

\*DIS - rectangular or square matrix with nonnegative elements.

\*In case DIS represent not two but one set of points, with distances between each other,

\*DIS will be square symmetric and you might want to ignore the diagonal elements (distances

\*from points to own selves); then set on the diagonal some number knowingly greater than the rest

\*elements of the matrix.

\*K - number of nearest neighbours, positive scalar <=R (usually K<<R).

\*If DIS is continuous values, NAME1 and NAME2 will have size K x C.

\*If DIS is discrete values, the number of rows in NAME1 and NAME2 may appear greater than K (see

\*below about it). Value 0 in NAME1 means emplty cell and, correspondingly, empty cell in that position

\*in NAME2.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*This function is beneficial when K<<R. If K is great and approaches to R, beneficial (from the

\*speed p.o.w) may be using sorting /\*!KO\_sort\*/ of each column of DIS with subsequent selection of

\*the K first rows of the result.

\*Difference between /\*!KO\_knnr\*/ and /\*!KO\_knnp\*/.

\*These functions are equivalent if DIS is continuous values (but /\*!KO\_knnp\*/ is somewhat faster then).

\*If DIS is discrete values, i.e., with ties, these functions show their algorithmic discrepancy

\*as follows. If points pretending for the nearest neighbourhood appear to be more than K,

\*/\*!KO\_knnr\*/ will choose from the tied neighbours randomly, in order there to be left exactly K

\*nearest neighbours; but /\*!KO\_knnp\*/ will output all the nearest neighboours, counting more than K.

\*Thus, in case o discrete DIS the result of /\*!KO\_knnr\*/ may depend on the random number seed, while

\*/\*!KO\_knnp\*/ may return more neighbours for a point, than K. Example. Let K=2, and the nearest

\*neighbours of a point have distances 2 3 3 3 4 from it. /\*!KO\_knnr\*/ will return two nearest

\*neighbours - with distances 2 and 3, selecting one of the three "3" randomly. /\*!KO\_knnp\*/ will return

\*four nearest neighbours: 2 3 3 3.

EXAMPLE. Continuous input distances.

set mxloops 1E6.

set seed 56986875.

matrix.

compute d= uniform(8,6).

!KO\_knnr(d%4%ind%dist).

print ind.

print dist.

!KO\_knnp(d%4%ind%dist).

print ind.

print dist.

end matrix.

EXAMPLE. Discrete input distances.

set mxloops 1E6.

set seed 56986875.

matrix.

compute d= rnd(uniform(8,6)\*5).

!KO\_knnr(d%4%ind%dist).

print ind.

print dist.

!KO\_knnp(d%4%ind%dist).

print ind.

print dist.

end matrix.

### INTRACLASS CORRELATION COEFFICIENT (ONE-WAY) [!KO\_iccow]

\*/\*!KO\_iccow(col%dummy%type%name)\*/\*.

\*Version1.

\*Computes intraclass correlation (ICC) in situation of one-way random design: scalar NAME.

\*Classes may be unequal-size (unbalanced).

\*COL - column of quantitative data.

\*DUMMY - binary dummy (indicator) variables (at least two) marking classes (groups): each variable

\*corresponds to a group, with value 1 = the case belongs to it, 0 = doesn't belong to it.

\*Each row of DUMMY must contain exactly one 1.

\*Each column in DUMMY must contain at least two 1s (with TYPE=PAIR2 or PAIR3), or at least one 1

\*and at least one column more than one 1s (with other TYPE).

\*TYPE - ICC estimator version, keyword in capital letters (optionally in quotes or apostrophes)

\*[see Donner, 1986: A review of inference procedures...; Shieh, 2012: A comparison of two indices...]:

\*"ANOVA" - ANOVA estimator.

\*"CETASQ" - Corrected Eta-square estimator.

\*"PAIR1" - Combinatorial pairing r Pearson estimator, without class weighting, classes of greater

\*size will informationally over-prevail.

\*"PAIR2" - as above, but with weighted classes: classes are given equal weight of importance irrespective

\*of their unequal size.

\*"PAIR3" - as above with intermediate, compromise variant of weighting: large classes will moderately

\*prevail.

\*If all classes (groups) are of same size (i.e. balanced) then PAIR1=PAIR2=PAIR3=CETASQ.

\*NAME from data may sometimes be negative whereas ICC in population is nonnegative; you may follow

\*the advice to zero-off negative values.

\*You can create dummy variables DUMMY out of a categorical grouping variable

\*with the help of function /\*!KO\_freq\*/.

### INTRACLASS CORRELATION COEFFICIENT (ONE-WAY, BALANCED CLASSES) [!KO\_iccowb]

\*/\*!KO\_iccowb(data%type%name)\*/\*.

\*Version1.

\*This function computes same ICC as function /\*!KO\_iccow\*/, but it is for particular case of

\*equal-size (balanced) classes only. The input here must be shaped in the "wide" format:

\*DATA is a data matrix (not less than 2x2) classes x measures, i.e. each class is a row, all rows

\*containing the same number of values. One shouldn't apply to DATA columns the meaning of regular

\*raters because in the current case one-way random design is assumed.

\*TYPE - ICC estimator version, keyword in capital letters (optionally in quotes or apostrophes)

\*[see Donner, 1986: A review of inference procedures...]

\*"ANOVA" - ANOVA estimator.

\*"PAIR" - Combinatorial pairing r Pearson estimator.

\*Because groups are of equal size, pairing estimator is identical to the Corrected Eta-square and

\*to maximum likelihood estimators, and it is unbiased, in contrast to the ANOVA estimator.

\*NAME from data may sometimes be negative whereas ICC in population is nonnegative; you may follow

\*the advice to zero-off negative values.

### COEFFICIENT OF DIAGONALNESS OF MATRIX [!KO\_diagns]

\*/\*!KO\_diagns(mat%name)\*/\*.

\*Version 1.

\*One of possible coefficients measuring the degree of concentration of large (by absolute magnitude)

\*values near its main diagonal. Ranges between 0 and 1.

\*Let MAT be an arbitrary matrix, then the coefficient is 1-D/Dmax, where D is the weighted Manhattan

\*physical distance of elements from the main diagonal, equal Sum[abs(MAT)&\*OFF], where OFF is

\*matrix sized as MAT, wherein the main diagonal comprised 0s, the pair of diagonals next to it

\*(i.e. apart of it by order 1) are comprised of 1s, the pair of diagonals apart of it by order 2

\*are comprised of 2s, and so forth. (&\* is multiplication of correspondent elements, and Sum is the sum

\*of elements.)

\*Dmax is the maximally possible distance for the given MAT, i.e. observed when all the large elements

\*of MAT are concentrated at bottom-left and top-right corners of the matrix; Dmax is computed as D,

\*only after sorting all MAT values by absolute magnitude and sorting OFF values by their magnitude.

\*If MAT is not square then not one but several diagonals in it pretend to be called "main diagonal".

\*These are all diagonals (going top-left to bottom-right) of the same, maximal length. For example,

\*in a matrix sized 4x6 it is three diagonals starting respectively from positions (1,1), (1,2) and (1,3).

\*The function will return, respectively, three coefficients, one pertaining to each diagonal: result NAME

\*is the column vector containing as much coefficients as there "main diagonals" embedding in the matrix,

\*and the 1st coefficient pertains to the diagonal starting from position (1,1).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### OUTLIER-ROBUST MEAN AND VARIANCE (ADP) [!KO\_robustadp]

\*/\*!KO\_robustadp(col%threshold%name1%name2)\*/\*.

\*Version 1.

\*For scale variable COL (a column) computes robust, with respect to possible outliers in it,

\*values of the mean (NAME1) and the variance (NAME2).

\*The function uses the equiinterval algorithm of SPSS command ADP (see SPSS Statistics Algorithms -

\*Automated Data Preparation Algorithms - Outlier Identification and Handling).

\*THRESHOLD - number between 0 and 1. It is the maximal proportion of extreme observations which the

\*algorithm will hold from influencing the being computed robust statistics, supposing the former to

\*be potential outliers. For example, THRESHOLD=0.05 will exclude no more than 5% of extreme

\*observations from influencing.

\*The point of the algorithm is simple: the data are split into intervals of width 1 sd; tail intervals

\*containing proportion of observations close to THRESHOLD from beneath are excluded; the robust

\*statistics are computed as usual mean and variance on the observations left.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

get x /variable= x.

!KO\_robustadp(x%.05%robmean%robvrnc).

compute robsd= sqrt(robvrnc).

print robmean /format= f8.6.

print robsd /format= f8.6.

compute flag= x<robmean-robsd\*3 or x>robmean+robsd\*3.

print {x,flag}. /\*Flag as outliers values that are farther

/\*than the 3 robust sd from the robust mean

end matrix.

### OUTLIER-ROBUST MEAN AND VARIANCE (LTS) [!KO\_robustlts]

\*/\*!KO\_robustlts(col%threshold%name1%name2)\*/\*.

\*Version 1.

\*For scale variable COL (a column) computes robust, with respect to possible outliers in it,

\*values of the mean (NAME1) and the variance (NAME2).

\*The function uses algorithm LTS (Least Trimmed Squares, see Rousseeuw, Leroy (1987),

\*"Robust regression and outlier detection", p. 171-172).

\*THRESHOLD - number between 0 and 1 (exclusive). It is the proportion of extreme observations which

\*the algorithm will hold from influencing the being computed robust statistics, supposing the former

\*to be potential outliers. For example, THRESHOLD=0.05 will exclude 5% of extreme observations

\*from influencing.

\*The point of the algorithm: N observations are sorted ascendingly of values; N-trunc(THRESHOLD\*N) is

\*the width of the band of observations on which the mean and variance are computed, this band

\*starts on the left and is shifted each time by one observation to the right. The function returns

\*as the result the least thus computed variance and its associated mean.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

get x /variable= x.

!KO\_robustlts(x%.05%robmean%robvrnc).

compute robsd= sqrt(robvrnc).

print robmean /format= f8.6.

print robsd /format= f8.6.

compute flag= x<robmean-robsd\*3 or x>robmean+robsd\*3.

print {x,flag}. /\*Flag as outliers values that are farther

/\*than the 3 robust sd from the robust mean

end matrix.

### TRIMMED MEAN [!KO\_trimmean]

\*/\*!KO\_trimmean(col%left%right%%name%)\*/\*.

\*Version 1.

\*Takes data column COL and computes trimmed mean NAME.

\*LEFT - proportion of observations to cut off from left (from the side of low values).

\*RIGHT - proportion of observations to cut off from right (from the side of high values).

\*LEFT+RIGHT must be below 1.

\*If the number of being cut cases (N\*LEFT or N\*RIGHT) is a fractional number,

\*a boundary case is cut off partially - as much as the fractional part is big.

### M-ESTIMATORS OF LOCATION [!KO\_mestim]

\*/\*!KO\_mestim(col%method%const%conv%name)\*/\*.

\*Version 1.

\*For scale variable (data column COL) computes a robust M-estimator of location.

\*METHOD - capitalized keyword (optionally may take in quotes or apostrophes):

\*"HUBER", "ANDREW", or "TUKEY".

\*CONST - positive scalar (weighting constant). Typically, the values used are:

\*1.339 for HUBER, 1.34 for ANDREW, 4.685 for TUKEY.

\*CONV - coefficient managing convergence on iterations of calculation, a small positive scalar

\*(for example, 0.005 or 0.001).

\*Algorithm - see SPSS Statistics Algorithms - EXAMINE - M-Estimation (Robust Location Estimation).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

get x /variable= x.

!KO\_mestim(x%HUBER%1.339%.005%m).

print m /title "Huber's M-estimator of location".

end matrix.

### CRONBACH'S ALPHA [!KO\_cronalpha]

\*/\*!KO\_cronalpha(cov%name)\*/\*.

\*Version 1.

\*Computes a measure of internal consistency (one of reliability indices) - Cronbach's alpha.

\*COV - p x p (p>2) matrix of covariances (or correlations) between the variables (test's items).

\*Normally all the covariances ought to be positive. Correlations in place of covariances

\*correspond to variables standardized.

\*Alpha NAME - positive (normally) coefficient between 0 and 1.

\*Negative value of the coefficient is possible with some covariances negative.

### MCDONALD'S OMEGA [!KO\_mcdomega]

\*/\*!KO\_mcdomega(cov%name)\*/\*.

\*Version 1.

\*Computes a measure of internal consistency (one of reliability indices) - McDonald's omega.

\*Used is the method by Hancock and Ann, 2020, - see SPSS Statistics Algorithms - Reliability -

\*Estimation of McDonald's Omega.

\*COV - p x p (p>2) matrix of covariances (or correlations) between the variables (test's items).

\*Normally all the covariances ought to be positive. Correlations in place of covariances

\*correspond to variables standardized.

\*Omega NAME - positive coefficient between 0 and 1.

\*NAME = -1 means that omega could not be computed due to nonpositive covariances.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

# STATISTICAL TRANSFORMATION FUNCTIONS

### CENTRATION OF DATA COLUMNS [!KO\_center]

\*/\*!KO\_center(data%name)\*/\*.

\*Version 1.

\*Centers columns of DATA (brings mean in them to 0).

\*Returns data NAME.

EXAMPLE.

matrix.

get vars /variables= v1 v2 v3.

!KO\_center(vars%vars).

save vars /outfile= \* /variables= v1 v2 v3.

end matrix.

### Z-STANDARDIZATION OF DATA COLUMNS [!KO\_zscore]

\*/\*!KO\_zscore(data%name)\*/\*.

\*Version 1.

\*Standardizes columns of DATA (brings mean in them to 0 and variance to 1).

\*Returns data NAME.

### Z-STANDARDIZATION OF DATA COLUMNS (DF=N) [!KO\_zscore2]

\*/\*!KO\_zscore2(data%name)\*/\*.

\*Version 1.

\*Standardizes columns of DATA (brings mean in them to 0 and variance to 1).

\*Standard deviation standardizing the data is computed on "df=n" (not "df=n-1").

\*Returns data NAME.

### SCALING OF DATA COLUMNS TO SS=1 [!KO\_scale]

\*/\*!KO\_scale(data%name)\*/\*.

\*Version 1.

\*Scales columns of DATA (brings sum of squares of values, SS, to 1).

\*Returns data NAME.

\*If need to make SS=num\_rows DATA, multiply the result by sqrt(num\_rows).

### CENTRATION OF DATA COLUMNS, BY GROUPS [!KO\_gcenter]

\*/\*!KO\_gcenter(data%dummy%name)\*/\*.

\*Version 1.

\*Centers columns of DATA (brings mean in them to 0), by groups of cases.

\*Returns data NAME.

\*DUMMY - binary dummy (indicator) variables marking groups (each variable corresponds to a group,

\*with value 1 = the case belongs to it, 0 = doesn't belong to it).

\*Each column in DUMMY must contain at least one 1.

\*The groups must be disjoint by membership: sum in each row in DUMMY must not exceed 1.

\*If DUMMY has rows complete of zeros, i.e. cases out of the groups, the data for the cases remain intact.

\*You can create dummy variables DUMMY out of a categorical grouping variable

\*with the help of function /\*!KO\_freq\*/.

### Z-STANDARDIZATION OF DATA COLUMNS, BY GROUPS [!KO\_gzscore]

\*/\*!KO\_gzscore(data%dummy%df%name)\*/\*.

\*Version 2.

\*Standardizes columns of DATA (brings mean in them to 0 and variance to 1), by groups of cases.

\*Returns data NAME.

\*DUMMY - binary dummy (indicator) variables marking groups (each variable corresponds to a group,

\*with value 1 = the case belongs to it, 0 = doesn't belong to it).

\*The groups must be disjoint by membership, and each case must belong to a group,

\*i.e. sum in each row of DUMMY must be equal to 1.

\*DF - scalar. If positive, the standardizing st. deviation will be computed on "df=n-1"

\*(then in each column of DUMMY there must be at least two ones); if non-positive, the st. deviation

\*will be computed on "df=n" (in each column of DUMMY there must be at least one one).

\*You can create dummy variables DUMMY out of a categorical grouping variable

\*with the help of function /\*!KO\_freq\*/.

### SCALING OF DATA COLUMNS, BY GROUPS [!KO\_gscale]

\*/\*!KO\_gscale(data%dummy%scale%name)\*/\*.

\*Version 1.

\*Scales columns of DATA, by groups of cases.

\*Returns data NAME.

\*SCALE - scalar. If nonpositive, sum of squares of variable values in each group will be

\*brought to 1; but if positive, then to the frequency in a group (i.e. mean square will be 1).

\*DUMMY - binary dummy (indicator) variables marking groups (each variable corresponds to a group,

\*with value 1 = the case belongs to it, 0 = doesn't belong to it).

\*Each column in DUMMY must contain at least one 1.

\*The groups must be disjoint by membership, and each case must belong to a group,

\*i.e. sum in each row of DUMMY must be equal to 1.

\*You can create dummy variables DUMMY out of a categorical grouping variable

\*with the help of function /\*!KO\_freq\*/.

### LINEAR RESCALING [!KO\_rescale]

\*/\*!KO\_rescale(data%type%val1%val2%name)\*/\*.

\*Version 1.

\*Linearly rescales variables in the dataset (columns of DATA) to the wanted magnitudes of statistics

\*in these data such as minimum, maximum, mean, st. deviation.

\*Data DATA must be of no less than two rows and the variables (columns) must be not constants.

\*Each variable is being rescaled independently; result NAME will have in each column

\*the linearly transformed data exactly satisfying two specified values VAL1 and VAL2 of the two

\*corresponding statistics according to TYPE.

\*VAL1 and VAL2 - scalars or row vectors lengthed as the number of columns of DATA; vector(s) mean

\*that the variables need to be brought each to its own specified value of the statistic.

\*TYPE (keyword in capital letters, optionally quoted or in apostrophes) sets which two

\*statistics the data in the variables are to be brought to:

\*"MIN\_MAX" - to specified minimum (VAL1) and maximum (VAL2), (VAL1<VAL2)

\*"MIN\_MEAN" - to specified minimum (VAL1) and mean (VAL2), (VAL1<VAL2)

\*"MAX\_MEAN" - to specified maximum (VAL1) and mean (VAL2), (VAL1>VAL2)

\*"MIN\_SD" - to specified minimum (VAL1) and standard deviation (VAL2), (VAL2 positive)

\*"MAX\_SD" - to specified maximum (VAL1) and standard deviation (VAL2), (VAL2 positive)

\*"MEAN\_SD" - to specified mean (VAL1) and standard deviation (VAL2), (VAL2 positive).

EXAMPLE.

matrix.

get vars /variables= v1 v2 v3 /names= names.

compute varmin= cmin(vars).

print varmin. /\*Minimal values, row vector

!KO\_rescale(vars%MIN\_MEAN%varmin%10%newvars). /\*Rescale variables to have same

/\*mean 10 while keeping their original minimal values

/\*(10 must be greater than any of those)

print cmin(newvars).

print (csum(newvars)/nrow(newvars)).

save newvars /out= \* /names= names.

end matrix.

### TWO-WAY CENTRATION

It is the centration of elements of matrix MAT both by its rows and its columns: MAT-R\*C/N, where R – the column of row sums, C – the row of column sums, N – total sum. Two-way centration is proportionally insensitive to multiplication of the initial matrix by a constant, not to adding a constant. You can obtain such centration by function /\*!KO\_cells\*/, requesting RESID there – frequency residuals is mathematically the two-way centration of a table of numbers.

### CONVERT COVARIANCE MATRIX INTO CORRELATION [!KO\_covcorr]

\*/\*!KO\_covcorr(cov%name)\*/.

\*Version 1.

\*Converts covariance matrix into correlation one (or sscp matrix into cosine similarities).

\*Obtained matrix NAME has ones on the diagonal.

\*Input matrix COV must have positive diagonal elements.

### CONVERT CORRELATION MATRIX INTO COVARIANCE [!KO\_corrcov]

\*/\*!KO\_corrcov(corr%dg%name)\*/.

\*Version 1.

\*Converts correlation matrix into covariance (or cosine similarity matrix into sscp matrix).

\*Obtained matrix NAME has vector DG on the diagonal.

\*Input matrix CORR must have ones on the diagonal.

\*DG - column vector of positive values, lengthed as the number of rows/columns of CORR.

### DOUBLE CENTERING OF MATRIX OF SQUARED DISTANCES [!KO\_dcenter]

\*/\*!KO\_dcenter(dis%name)\*/\*.

\*Version 1.

\*Converts matrix of squared dissimilarities DIS into matrix of similarities NAME which are scalar products.

\*If DIS is squared dissimilarities between points of a cloud, then on the diagonal of NAME are squared

\*euclidean distances from the points to the centroid of the cloud, and off-diagonal elements are scalar

\*products between vectors shed from the centroid and ending on the points.

\*If the dissimilarities are squared euclidean distances then this operation is a "geometrically correct"

\*transformation them into similarities. Namely, then NAME=X\*t(X), where matrix X is any data (points by

\*dimensions, and the dimensions, the columns, are centered) having between the points the sq. euclidean

\*distances equal to DIS.

\*In double-centered matrix NAME row and column sums equal 0.

\*DIS must be square symmetric with non-negative elements and with 0 on the diagonal.

\*For back conversion use the opposite function /\*!KO\_sdcosth\*/.

### CONVERT MATRIX OF ANGULAR SIMILARITIES INTO SQUARED DISTANCES BY COSINE THEOREM [!KO\_sdcosth]

\*/\*!KO\_sdcosth(sim%name)\*/\*.

\*Version 1.

\*Converts matrix of similarities SIM, which it takes as scalar products, into matrix of squared

\*dissimilarities NAME. Diagonal values of SIM are used as the squares of the distances from the points to the space

\*origin. If SIM are scalar products and SIM is positive semidefinite then dissimilarities NAME are

\*squared euclidean distances. If SIM is not positive semidefinite then NAME may be sometimes

\*squared euclidean distances (converge in euclidean space).

\*This operation is a "geometrically correct" transformation of covariances, correlations or cosine similarities into

\*euclidean distances corresponding to them.

\*SIM must be square symmetric with positive values on the diagonal.

\*This function is opposite to functions /\*!KO\_dcenter\*/ or /\*!KO\_dscosth\*/.

### SWITCHING MATRIX OF SCALAR PRODUCTS ONTO NEW DIAGONAL (PRESERVING ANGLE) [!KO\_swdiag1]

\*/\*!KO\_swdiag1(sim%diag%name)\*/.

\*Version 1.

\*Recalculates elements of SSCP-type matrix (= matrix of scalar products; in particular,

\*covariance, correlation, cosine similarity matrices belong to) onto the given diagonal values.

\*At that, angles (cosines) between the vectors are kept unchanged.

\*With this function one can, for example, turn a covariance matrix into a correlation or vice versa.

\*SIM must be square symmetric, with positive elements on the diagonal. The matrix is treated as

\*a matrix of similarities of SSCP type (a matrix of scalar products).

\*DIAG - vector (row or column) lengthed as the number of rows/columns in SIM, containing positive

\*elements; it is the new diagonal for SIM.

\*Result NAME is SIM with diagonal DIAG and recalculated off-diagonal elements.

\*If the input SIM is gramian (all eigenvalues nonnegative), NAME is also gramian.

EXAMPLE.

matrix.

compute vars= uniform(20,8). /\*Some columns, they are the vectors

compute sp= sscp(vars). /\*Their sscp matrix, the scalar products

print sp.

compute newdiag= uniform(1,8)\*2+1. /\*Let this be the new diagonal for it

/\*All values must be positive

print newdiag.

!KO\_swdiag1(sp%newdiag%newsp). /\*Switch sp onto the new diagonal

print newsp.

print eval(newsp). /\*The new matrix will always be gramian (provided the initial

/\*one was)

\*Now check the cosines (angles) between the vectors.

!KO\_cosine(vars%csn). /\*The cosines in the initial matrix

/\*(can get them right from the data)

print csn.

!KO\_covcorr(newsp%newcsn). /\*What the cosines became after we replaced the diagonal

/\*(we can get it from newsp exactly like we convert a

/\*covariance matrix into correlation)

print newcsn.

print abs(csn-newcsn). /\*The cosines appear to be preserved

end matrix.

### SWITCHING MATRIX OF SCALAR PRODUCTS ONTO NEW DIAGONAL (PRESERVING DISTANCE) [!KO\_swdiag2]

\*/\*!KO\_swdiag2(sim%diag%name)\*/.

\*Version 1.

\*Recalculates elements of SSCP-type matrix (= matrix of scalar products; in particular,

\*covariance, correlation, cosine similarity matrices belong to) onto the given diagonal values.

\*At that, euclidean distances between the vectors' endpoints are kept unchanged.

\*SIM must be square symmetric, with positive elements on the diagonal. The matrix is treated as

\*a matrix of similarities of SSCP type (a matrix of scalar products).

\*DIAG - vector (row or column) lengthed as the number of rows/columns in SIM, normally containing

\*positive elements; it is the new diagonal for SIM.

\*Result NAME is SIM with diagonal DIAG and recalculated off-diagonal elements.

\*Depending on DIAG (and, of course, on the input SIM), NAME may appear gramian

\*(all eigenvalues nonnegative) or nongramian.

EXAMPLE.

matrix.

compute vars= uniform(20,8). /\*Some columns, they are the vectors

compute sp= sscp(vars). /\*Their sscp matrix, the scalar products

print sp.

compute newdiag= uniform(1,8)\*2+1. /\*Let this be the new diagonal for it

/\*Normally the values should be positive (though mathematically there is

/\*no such requirement)

print newdiag.

!KO\_swdiag2(sp%newdiag%newsp). /\*Switch sp onto the new diagonal

print newsp.

print eval(newsp). /\*The new matrix may be gramian or nongramian depending on newdiag

/\*(and whether the initial matrix was)

\*Now check the euclidean distances between the vectors’ endpoints.

!KO\_seuclid(t(vars)%d). /\*The (squared) distances in the initial matrix

/\*(can get them right from the data)

print d.

!KO\_sdcosth(newsp%newd). /\*What the distances became after we replaced the diagonal

/\*(we can get it from newsp by convertion through

/\*the cosine theorem

print newd.

print abs(d-newd). /\*The distances appear to be preserved

end matrix.

### SWITCHING MSCP MATRIX ONTO NEW DATA CENTROID [!KO\_swcentr]

\*/\*!KO\_swcentr(sim%oldmean%newmean%name)\*/.

\*Version 1.

\*Recalculates elements of MSCP matrix defined by columns of some data onto a new centroid, that is,

\*as would be if those data are shifted onto that new centroid (multivariate mean). The function does not

\*need the original data. Say, a MSCP matrix (it is SSCP-матрица - i.e. matrix t(data)\*data - devided by

\*the number of cases in the data) was computed for some variables, as well as the vector of their means.

\*Then the data were lost. What would be the new MSCP matrix after the data would have been changed their

\*means arbitrarily by adding some constant to each variable? The function answers this question in the

\*absence data themselves. For example this function can restore the MSCP matrix of the original data

\*from the covariance matrix (which, as we know, is a MSCP matrics too) if the original means of the

\*variables are known.

\*SIM - square symmetric MSCP matrix between coluns of some data; means in those columns are recorded

\*as vector (row or column) OLDMEAN. After shifting of this centroid onto new coordinates NEWMEAN

\*the new MSCP matrix corresponding to the shifted data is just the matrix NAME which is returned.

\*Be aware that the result matrix does not have to be gramian (positive [semi]-definite) -

\*the gramianness will depend on the input matrix SIM and feasability of OLDMEAN and NEWMEAN. The function

\*does not watch after realisticity of the inputs. When input statistics come from real variables there

\*should be no problem with gramianness.

EXAMPLE. Converting of MSCP matrix into covariance and back: demonstration.

matrix.

get vars /variables= v1 v2 v3 v4.

compute n= nrow(vars).

compute mscp= sscp(vars)/n.

!KO\_mean(vars%mean).

print mscp /title 'MSCP matrix of some variables'.

print mean /title 'Means (centroid) of those variables'.

!KO\_swcentr(mscp%mean%{0,0,0,0}%mscp0). /\*Translate the given MSCP onto the new

/\*centroid, in this example, zero

print mscp0 /title 'MSCP matrix shifted to the zero centroid...'.

!KO\_cov2(vars%cov2).

print cov2 /title '... is (by definition) the cov matrix (computed on "df=n")'.

!KO\_swcentr(mscp0%{0,0,0,0}%mean%mscp\_). /\*Now translate it back

print mscp\_ /title 'Shifting back the cov matrix into the original MSCP'.

end matrix.

EXAMPLE. The positive definitness issue.

matrix.

compute mx= {1.8, 0.5, -0.2, 0.2;

0.5, 3.1, -0.1, 0.4;

-0.2, -0.1, 2, 0.3;

0.2, 0.4, 0.3, 1.6}. /\*A MSCP matrix

print mx.

print all(eval(mx)>0). /\*It is gramian (positive definite)

!KO\_swcentr(mx%{1.2,0,-1.1,0}%{0,0,0,0}%mx\_). /\*Assuming the variables had means

/\*{1.2,0,-1.1,0}, what would be their covariance matrix

/\*(i.e. MSCP with centroid 0)?

print mx\_. /\*It would be this

print all(eval(mx\_)>0). /\*Unfortunately, it is nongramian, that is, impossible

/\*in real variables; so, either MX or the means were unrealistic; Sorry

end matrix.

matrix.

compute mx= {3.1, 0.8, -0.2, 0.4;

0.8, 2.1, -0.1, 0.2;

-0.2, -0.1, 2.3, 0.3;

0.4, 0.2, 0.3, 1.9}.

print mx.

print all(eval(mx)>0).

!KO\_swcentr(mx%{1.2,0,-1.1,0}%{0,0,0,0}%mx\_).

print mx\_.

print all(eval(mx\_)>0). /\*Under these MX and the means, there is no problem,

/\*the cov matrix MX\_ could really exist

end matrix.

### DUMMY VARIABLES

To recode a categorical variable into a set of binary indicator (dummy) variables - see function /\*!KO\_freq\*/. If sorting of the order of the dummy variables ascendingly by the codes of the categories is not needed then you may as well use in-built function design(). For the backward operation – turn a set of binary dummy variables into one categorical variable, use operation dummy\*t(1:ncol(dummy)), where dummy is the matrix which columns are dummy variables.

### MULTIPLE RESPONSE SET VARIABLES

To recode a categorical multiple response set (several categorical variables with common pool of codes) into dichotomous multiple response set (set of binary variables) – see function /\*!KO\_mrfreq\*/. For the backward operation – turn a set of binary variables into several categorical variables forming categorical multiple response set, use function /\*!KO\_ram\*/ (see example there).

# 

### RANK VALUES

Use in-built functions grade() and rnkorder().

### RIDITS OF ORDINAL LEVELS [!KO\_ridit]

\*/\*!KO\_ridit(vec%name)\*/\*.

\*Version 1.

\*Computes ridit scores of ordinal levels.

\*VEC - vector (row or column) containing frequencies (or proportions) of levels of an ordinal variable.

\*1-st element – frequency in the 1-st level, 2-nd element – frequency in the 2-nd level, and so on.

\*NAME – levels' ridits.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### EXPONENTIAL NORMALIZATION (SOFTMAX FUNCTION) [!KO\_nef]

\*/\*!KO\_nef(data%name)\*/\*.

\*Version 1.

\*Exponentially normalizes elements in columns of DATA.

\*Result NAME - in each column values are exponentially transformed so that

\*they lie in the range [0,1] and their sum is equal to 1.

### CENTERED LOGRATIO TRANSFORM [!KO\_clr]

\*/\*!KO\_clr(data%name)\*/\*.

\*Version 1.

\*Performs centered logratio transformation of elements in columns of DATA. Values in DATA

\*must be positive.

\*The transform consists first of taking column sum to 1, then division by

\*geometric mean in column, and taking logarithm.

\*Result NAME - values in each column have zero mean.

### BOX-COX TRANSFORM [!KO\_boxcox]

\*/\*!KO\_boxcox(col%grid%name1%name2)\*/\*.

\*Version 1.

\*Transforms a quantitative variable COL monotonically by exponentiation so that its distribution

\*become closer to normal. Selecting of the optimal value for the power parameter lambda is done by

\*grid search the same way SPSS command ADP does it when performing Box-Cox transform.

\*COL - input variable, column of positive values. If nonpositive values are present, please add

\*a constant beforehand to make all values positive (as a variant, you may always, irrespective

\*whether there are or there are no nonpositive data, do COMPUTE COL= COL-cmin(COL)+1, the way ADP

\*command does, for example).

\*GRID - vector (row or column) of three values - this is the grid to search a value of lambda:

\*{min,max,step}. Max must be > Min, Step positive, and (Max-Min)/Step must give integer.

\*For example, often used grid is {-3,3,0.5}.

\*Result:

\*NAME2 - the selected power lambda of the Box-Cox transform.

\*NAME1 - the transformed by it data COL. (Command ADP additionally z-standardizes it, but this

\*function omits doing it).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Box-Cox transform same as done by ADP command.

matrix.

get y /variable= y.

compute y= y-cmin(y)+1.

!KO\_boxcox(y%{-3,3,0.5}%y\_%lambda).

!KO\_zscore(y\_%y\_).

print lambda.

save y\_ /out= \*.

end matrix.

### WHITENING (SPHERING) MATRIX [!KO\_white]

\*/\*!KO\_white(cov%method%name)\*/\*.

\*Version 1.

\*Whitening or sphering is a name for transforming multivariate data into "spherical" cloud.

\*The function takes covariance or correlation matrix COV and computes whitening (sphering) matrix

\*NAME such, so if there are some variables X corresponding to COV, then variables Z=X\*NAME

\*will have identity covariance matrix. Thus, whitened data are decorrelated and standardized

\*variables, while the function returns the matrix to do such transformation.

\*COV - square p x p symmetric matrix, covariance or correlation.

\*METHOD - whitening method, keyword in capital letters (might take in quotes or apostrophes):

\*"ZCA" - zero-phase component method; returns NAME (symmetric, with this method) minimizing squared

\*differences between values of Z and X. Use this method if you need variables Z to depart minimally

\*from their corresponding variables X.

\*"PCA" - principal component method; returns NAME maximizing the account of variance of X in the first

\*columns of Z; Z are actually the principal components of X, and Z values are the standardized principal

\*component scores. Use this method if you may wish to reduce dimensionality (number of columns) in Z

\*comparatively with X, with minimal informational loss.

\*"CHOL" - Cholesky method; returns NAME (upper triangular, with this method) providing complete

\*correlation of Z1 with X1 and orthogonality of Zp with all X except Xp. Use if it needs that Z1

\*be associated with all Xs, especially X1; Z2 be not associated with X1; Z3 be not associated with X1

\*and X2, etc.

\*With methods "ZCA" and "PCA" whitened variables Z and their relation with the initial variables X will

\*depend on whether COV is covariance or correlation matrix (so it matters if you input covariance or

\*correlation COV into the function). Let matrix R be covariances (if COV is covariance) or correlations

\*(if COV is correlation) between X variables (rows of R) and Z variables (columns of R). Then:

\*-"ZCA" method maximizes diagonal elements of R (they're positive).

\*-with "PCA" method, R is the matrix of principal component loadings, so the method maximizes in the

\*first place the sum of squared elements of the 1st column in R (i.e. covariances/correlations of Z1 with

\*all Xs), in the second place the sum of squared elements of the 2nd column in R (i.e. Z2 with

\*all Xs), etc.

\*With method "CHOL" whitened variables Z will be the same be COV covariance or correlation matrix

\*(so it matters not if you input covariance or correlation COV into the function).

\*This method leads to correlation matrix R such that it is lower triangular with positive diagonal

\*and element (1,1)=1, while in its last column all elements except (p,p) are zero.

\*It is important to note that for all the methods there must be followed the rule: if COV is correlation

\*then at whitening Z=X\*NAME columns of X must be standardized variables (not raw or just centered).

\*Methods "ZCA" and "PCA" are connected together via data rotation so that ZCA whitening is equivalent

\*to PCA whitening with subsequent procrustes rotation of Z back to X, and the matrix of such rotation is

\*eigenvectors of COV.

EXAMPLE. ZCA whitening maximazing correlations.

matrix.

get x /variables= x1 x2 x3 x4.

!KO\_corr(x%corr). /\*Correlations

!KO\_white(corr%ZCA%w).

print w. /\*Whitening matrix

!KO\_zscore(x%x). /\*With correlation matrix as basis of whitening, X must be standard

compute z= x\*w. /\*Z is whitened X

!KO\_cov(z%covz).

print covz. /\*Covariances in Z are identity matrix

!KO\_corr({x,z}%corrxz).

print corrxz(1:4,5:8) /rlab= 'x1' 'x2' 'x3' 'x4' /clab= 'z1' 'z2' 'z3' 'z4'.

/\*Correlations between original and whitened variables:

/\*diagonal is maximized

save z /outfile= \* /variables= z1 z2 z3 z4.

end matrix.

### TRANSFORM VARIABLES EXACTLY TO HAVE SPECIFIED COVARIANCES [!KO\_tocov]

\*/\*!KO\_tocov(ingot%type%cov%method%name)\*/\*.

\*Version 2.

\*Transforming of multivariate quantitative data INGOT so that the observed covariances (or correlations

\*or other sscp measures of association) between these variables become exactly equal to the

\*user-specified ones.

\*Argument INGOT - n cases x p variables (n>p) data matrix with values serving an "ingot", the stuff to

\*obtain variates associations between which will satisfy matrix COV. INGOT could be randomly generated

\*data that you want to take to the needed correlatedness, or INGOT could be existing, analysis data

\*which you want to modify minimally and make them reproduce the specified correlatedness

\*(see METHOD below). Columns in INGOT must be noncollinear.

\*Argument TYPE - keyword in capital letters (optionally may put in quotes or apostrophes). Specify

\*the type of the target matrix you input as argument COV:

\*"COV" - covariance matrix (on "df=n-1");

\*"COV2" - covariance matrix (on "df=n") or mscp matrix;

\*"CORR" - correlation matrix or cosine similarity matrix.

\*(All the listed matrices are akin: they are matrices of a sscp(variables)/"df" kind.)

\*Argument COV - p x p square symmetric positive definite matrix of associations (its type you marked

\*in TYPE). It is this target matrix for which there must be obtained, out of INGOT, the variates

\*satisfying its coefficients.

\*Result NAME - n x p data where columns are such variates that they exactly reproduce the specified

\*coefficients COV.

\*Use the following rules to specify input:

\*-If the target matrix COV is covariance one (on "df=n-1"), then type="COV", INGOT columns must be centered

\* (one can center variables with function /\*!KO\_center\*/).

\*-If the target matrix COV is covariance one (on "df=n"), then type="COV2", INGOT columns must be centered.

\*-If the target matrix COV is mean squares-and-crossproducts (mscp), then type="COV2", INGOT is raw data

\* as is.

\*-If the target matrix COV is correlation one (unit must be on the diagonal), then type="CORR", INGOT

\* columns must be centered. At output, variables will retain their initial scale (variance).

\*-If the target matrix COV is cosine similarity (unit must be on the diagonal), then type="CORR", INGOT is

\* raw data as is. At output, variables will retain their initial scale (mean square).

\*Argument METHOD - scalar, defines the fitting algorithm. If it is positive then at data transformation

\*there will be applied component loadings of the target matrix procrustes-rotated towards component

\*loadings of INGOT. Use this variant if you want the values of NAME to differ minimally in terms of

\*overall sum of squares from the values of INGOT: mssq(INGOT-NAME)=min. In other words, if the task

\*is to "correct" INGOT, taking the data to the wanted amount correlation.

\*If argument METHOD is nonpositive then at data transformation there will be applied Cholesky root of

\*the target matrix. Use this more fast variant if you want simply to create data, i.e. when INGOT for

\*you is just a "pig" of random values that were generated before. This variant does not seek to modify

\*INGOT minimally.

\*Note: if INGOT are values from normal distribution then NAME are data with also multivariate

\*normal distribution; otherwise distribution in NAME does not have to relate to distribution

\*in INGOT.

EXAMPLE. Random data from normal distribution with specified observed correlations.

matrix.

compute mx= {1, 0.5, -0.2, 0.2;

0.5, 1, -0.1, 0.4;

-0.2, -0.1, 1, 0.3;

0.2, 0.4, 0.3, 1}. /\*Matrix of required correlations (4 variables)

print mx.

!KO\_normal(100%4%data). /\*Generate 100 cases x 4 variables of normal random data

/\*(it is the raw “ingot” variables)

!KO\_center(data%data). /\*Center exactly to zero means (because mx is correlation

/\*or covariance matrix)

!KO\_tocov(data%CORR%mx%0%vars). /\*Obtaining the random variables with correlations=mx;

/\*(they are centered), ready; but let us set them the following

compute mean= {2.0,3.6,2.1,-1.8}. /\*means

compute sd= {0.7,0.9,1.5,1.9}. /\*and st. deviations

!KO\_rescale(vars%"MEAN\_SD"%mean%sd%vars).

!KO\_corr(vars%r).

print r.

save vars /outfile= \*.

end matrix.

EXAMPLE. Correct data so that they have the needed covariances.

matrix.

compute mx= {1.8, 0.5, -0.2, 0.2;

0.5, 2.1, -0.1, 0.4;

-0.2, -0.1, 1, 0.3;

0.2, 0.4, 0.3, 0.6}. /\*Matrix of required covariances

print mx.

get data /variables= v1 v2 v3 v4. /\*Variables we want to “correct”

!KO\_cov(data%cov).

print cov. /\*Their observed covariances

!KO\_mean(data%mean). /\*Their means

!KO\_center(data%data). /\*Center exactly to zero means (because mx is correlation

/\*or covariance matrix)

!KO\_tocov(data%COV%mx%1%vars). /\*Transforming the variables towards covariances=mx;

/\*they are centered; and let us restore them

compute vars= vars+mean(make(1,nrow(vars),1),:). /\*their original means

!KO\_cov(vars%newcov).

print newcov.

save vars /outfile= \*.

end matrix.

### TRANSFORM A VARIABLE EXACTLY TO HAVE SPECIFIED COVARIANCES WITH OTHER VARIABLES [!KO\_ytocov]

\*/\*!KO\_ytocov(yingot%xvars%type%cov%name)\*/\*.

\*Version 1.

\*Transforms YINGOT variable of the dataset so that its observed covariances (or correlations or

\*other sscp association measures) with variables XVARS in the data will become exactly equal to

\*the user-specified ones. Result NAME - the modified YINGOT, altered minimally in the sense of sums of

\*squares: mssq(YINGOT-NAME)=min. XVARS will not be modified.

\*Argument INGOT - column of n cases, it is the "ingot" quantitative variable for obtaining variable NAME.

\*You may generate YINGOT as random values or it could be existing, analysis data.

\*Argument XVARS - one or more quantitative variables in the dataset, n x p, these p columns must be

\*noncollinear.

\*Argument TYPE - keyword in capital letters (optionally may put in quotes or apostrophes). Specify

\*the type of the target vector you input as COV argument:

\*"COV" - covariances (on "df=n-1");

\*"COV2" - covariances (on "df=n") or mean cross-products;

\*"CORR" - correlations or cosine similarities.

\*(All the listed types are akin: they are coefficients of the sscp(variables)/"df" kind.)

\*Argument COV - row vector of the target coefficients themselves (their type you've marked in TYPE).

\*It is the coefficients of association with XVARS which YINGOT will have to satisfy exactly when it

\*is transformed to NAME. The length of COV row is p+1. First p values correspond respectively to

\*the p columns (variables) XVARS, and the last additional value is the scale for variable NAME (see below).

\*Use the following rules to specify input:

\*-If your target coefficients COV are covariances (on "df=n-1"), then type="COV", the last element

\* of COV vector must be the desired variance for NAME (positive value), YINGOT and XVARS columns must be

\* centered (one can center variables with function /\*!KO\_center\*/).

\*-If your target coefficients COV are covariances (biased, on "df=n"), then type="COV2", the last element

\* of COV vector must be the desired biased variance for NAME (positive value), YINGOT and XVARS columns

\* must be centered.

\*-If your target coefficients COV are mean crossproducts, then type="COV2", the last element

\* of COV vector must be the desired mean square for NAME (positive value), YINGOT and XVARS are raw

\* data as is.

\*-If your target coefficients COV are correlations, then type="CORR", the last element of COV vector must

\* be 1, YINGOT and XVARS columns must be centered. NAME will retain YINGOT's initial scale (variance).

\*-If your target coefficients COV are cosine similarities, then type="CORR", the last element of COV vector

\* must be 1, YINGOT and XVARS are raw data as is. NAME will retain YINGOT's initial scale (mean square).

\*Note1: If the specified coefficients COV, their magnitudes, do not permit creation of a correct variable

\*NAME out of variable YINGOT (because of non-gramian, not positive-definite configuration occuring)

\*then the function will return NAME= scalar 0.

\*Note2: if YINGOT are values from normal distribution then NAME are values also with normal

\*distribution; otherwise distribution in NAME does not have to relate to distribution

\*in YINGOT.

EXAMPLE. Take Y to the needed correlations with one or more X.

matrix.

get y /variable= y. /\*Variable to modify

get x /variables= x1 x2 x3.

!KO\_center(y%y). /\*Centering required

!KO\_center(x%x). /\*because coefficients are correlations (or covariances)

!KO\_ytocov(y%x%CORR%{.32,-.22,0,1}%y#). /\*Run the function

/\*three target correlations, plus 1 as the last element (corr. of Y

/\*with itself)

do if nrow(y#)=1. /\*If the result is 0 scalar

-print /title 'Sorry, The input coefficients are unrealistic.'.

else.

-!KO\_corr({y#,x}%coef).

-print coef(1,2:ncol(coef)).

-!KO\_variance(y#%vrnc).

-print vrnc.

-save {y#,x} /outfile= \* /variables= y# x1 x2 x3.

end if.

end matrix.

### NORMAL CLOUD INTO UNIFORM BALL [!KO\_unifball]

\*/\*!KO\_unifball(data%alpha%name1%name2)\*/\*.

\*Version 1.

\*Takes uni- or- multivariate data DATA, which the function expects to be from

\*the standard normal distribution. The function reduces the data kurtosis by pulling

\*the periphery (tails) in stronger than the paracentral regions. Under parameter ALPHA=1

\*the normal cloud turns into the uniform, into the (hyper)ball.

\*DATA - data (cases x variables) from normal distribution with mean 0 and variance 1;

\*if there are multiple variables, the function expects that they are independent, uncorrelated.

\*(You can generate random standard orthogonal data by function /\*!KO\_normal\*/,

\*and to standardize variables, if necessary - by function /\*!KO\_zscore\*/.)

\*ALPHA - scalar, parameter of the degree of contraction of the periphery (tails) of data.

\*The higher it is, the stronger will the function reduce kurtosis in the data.

\*Under parameter 1 a normal cloud transforms to uniform one, and that will be uniform hyper(ball)

\*if the data are multivariate. Under parameter 0 the data will remain as they were at input.

\*With parameter between 0 and 1 you will obtain data with kurtosis in-between of normal and of

\*uniform. Parameter greater than 1 makes data yet bimodal (concave in the middle).

\*The formula of data transformation:

\*Y\_i = X\_i \* coef,

\*coef = 1 + alpha \* (sqrt(p+2)\*cdf.chisq(d^2,p)^(1/p)/d - 1),

\*where Y\_i and X\_i - values of data point i at output (NAME1) and at input (DATA);

\*coefficient of contraction coef is returned as NAME2;

\*p - dimensionality, number of columns of DATA;

\*d - euclidean distance from data point i in DATA to the origin, i.e. to 0 (according to the said

\*above, centroid of DATA should lie in 0 or approach 0);

\*cdf.chisq(val,df) - chi-square cumulative distribution function;

\*sqrt(p+2) factor preserves the same expected variability (i.e., 1 per dimension);

\*and when ALPHA=1, the hyperball NAME1 expects radius sqrt(p+2).

\*(See: https://stats.stackexchange.com/questions/79919).

\*Correlating variables. Although the function will reduce kurtosis in such a cloud and will make

\*the ellipsoid more ball-like, it will not attain uniformity of distribution under ALPHA=1. The

\*function accomplishes its role only when the input variables are uncorrelated. If you want to

\*convert correlated variables from normal distribution into ellipse with uniform distribution inside

\*it, do first whitening of the data (sphering and standardization), for instance, by principal

\*component analysis (function /\*!KO\_pcomp\*/). Then apply the current function, and in the end

\*do the transform backward of that principal component analysis - see EXAMPLE.

EXAMPLE.

matrix.

!KO\_normal(3000%2%data). /\*Some data from normal distribution

/\*3000 points, 2 uncorrelated variables

!KO\_zscore(data%data). /\*It is expected that the data are from standard

/\*normal variates; if not, standardize;

/\*You may prefer to standardize it anyway

!KO\_unifball(data%1%ball%coef). /\*Turn the spherical normal cloud into the uniform

/\*ball (alpha=1) cloud

\*print (ball/(coef\*make(1,ncol(ball),1)). /\*(This is the back-transform)

save {data,ball} /outfile= \* /variables= norm1 norm2 unif1 unif2.

end matrix.

GRAPH /SCATTERPLOT(BIVAR)= norm1 WITH norm2 /title 'Initial normal cloud'.

GRAPH /SCATTERPLOT(BIVAR)= unif1 WITH unif2

/title 'Resultant uniform spherical cloud'.

DESCRIPTIVES VARIABLES= norm1 norm2 unif1 unif2.

/\*Note that mean and st dev almost preserved

EXAMPLE.

matrix.

!KO\_normal(3000%1%data). /\*Some data from normal distribution

/\*3000 points, 1 variable

/\*!KO\_zscore(data%data).

!KO\_unifball(data%.5%trans%coef). /\*Turn the normal data into the one with kurtosis

/\*lesser than normal ("halfway" towards uniformity, alpha=0.5)

save {data,trans} /outfile= \* /variables= norm trans.

end matrix.

GRAPH /HISTOGRAM(NORMAL)= norm.

GRAPH /HISTOGRAM(NORMAL)= trans.

DESCRIPTIVES VARIABLES= norm trans.

EXAMPLE. Correlated variables.

matrix.

!KO\_mvnorm(3000%chol({1,.7,.5;.7,1.2,.2;.5,.2,.8})%0%data). /\*3 normal variates with

/\*the specified population covariances and mean 0, 3000 cases

!KO\_cov(data%cov). /\*Let's look at the

print cov. /\*observed covariances in the data

!KO\_pcomp(data%1%0%1%name1%load%name3%name4%pcscores). /\*Perform PCA in order to

/\*whiten the data (i.e., make the dimensions uncorrelated and with

/\*variances 1); Because parameter OUT=1, pcscores (pr. component scores) are

/\*already that standard

!KO\_unifball(pcscores%1%pcscores%coef). /\*So we can apply KO\_unifball to them;

/\*aplha=1, thus we've turned the normal cloud into the uniform ball

/\*(with name pcscores)

compute trans= pcscores\*t(load). /\*With the help of PC loadings, rotate that result

/\*back into the correlated state of original data

!KO\_cov(trans%cov\_trans). /\*Let's look at the covariances in

print cov\_trans. /\*the transformed data: reasonably close to what there was at input

save {data,trans} /outfile= \* /variables= norm1 norm2 norm3 unif1 unif2 unif3.

end matrix.

GRAPH /SCATTERPLOT(XYZ)= norm1 WITH norm2 WITH norm3 /title 'Initial normal cloud'.

GRAPH /SCATTERPLOT(XYZ)= unif1 WITH unif2 WITH unif3

/title 'Resultant uniform cloud'.

DESCRIPTIVES VARIABLES= norm1 norm2 norm3 unif1 unif2 unif3.

STATISTICAL ANALYTIC FUNCTIONS

### LINEAR REGRESSION [!KO\_regress]

\*/\*!KO\_regress(dv%ivs%const%name1%name2%name3)\*/\*.

\*Version 1.

\*Linear regression OLS (without significance tests).

\*Takes dependent variable DV and one or more independent variables IVS (which must be noncollinear).

\*Argument CONST (scalar) - request model with constant (CONST positive) or without constant (CONST nonpositive).

\*Results:

\*NAME1 - SSregression, SSresiduals, observed coefficient of determination R-square.

\*NAME2 - two-column matrix of regression coefficients: raw coefficients in the 1st col; standardized

\*coefficients (beta) in the 2nd col. If model was with constant, number of rows in NAME2 is 1 more

\*than the number of columns IVS, and the constant is in the last row.

\*NAME3 - predicted values (raw) of the dependent variable.

EXAMPLE.

matrix.

get y /variables= y.

get x /variables= x1 x2 x3 x4 /names= names.

!KO\_regress(y%x%1%summary%coef%pred).

print summary /clabels= 'SSregr' 'SSresid' 'Rsq'.

print coef /rnames= names /clabels= 'b' 'beta'.

print pred.

end matrix.

### PRINCIPAL COMPONENTS [!KO\_pcomp]

\*/\*!KO\_pcomp(data%df%m%out%name1%name2%name3%name4%name5)\*/\*.

\*Version 4.

\*Principal component analysis.

\*Takes data DATA (where rows are points, columns are space axes) and returns principal

\*components of these data.

\*Attention: The data are analyzed "as is", therefore if preliminary centration

\*or standardization is needed then use first the corresponding functions.

\*(Centration corresponds to the analysis of covariances and standardization - to the analysis of

\*correlations.)

\*Argument DF (scalar) - if positive, decomposed will be matrix sscp(DATA)/(n-1), if not positive

\*then matrix sscp(DATA)/n, where n is the number of rows in DATA.

\*Argument M (scalar) - how many 1st principal components to extract: specify positive integer

\*from 1 to number\_columns DATA. Or specify any nonpositive number: then all nonzero principal

\*components will be extracted.

\*Argument OUT (scalar) controls the form of results returned.

\*Results:

\*NAME1 contains in the 1st col eigenvalues of the decomposed matrix,

\*and in the 2nd col - proportions of explained variance (or scale), i.e. proportions of the trace

\*of the matrix.

\*NAME2 - eigenvectors (if OUT nonpositive) or loadings (if OUT positive); sum-of-squares

\*in columns of loadings = eigenvalues. Loadings are the coefficients of prediction of variables by

\*scaled components. Eigenvectors are the coefficients of prediction of variables by raw components,

\*or the cosines of angles of rotation.

\*NAME3 - rescaled loadings. Rescaled or standardized loading is a loading divided by the scale of

\*the variable - i.e., by the root of the corresponding diagonal element of the decomposed matrix.

\*By this division the effect of uniqual scales (variances) of variables on the loadings is taken off.

\*Rescaled loading is the cosine similarity or correlation between a component and a variable.

\*Squared elements of eigenvectors can be understood as the magnitude of contributions of variables

\*into components. Squared elements of rescaled loadings can be understood as the magnitude of

\*contributions of components into variables.

\*NAME4 - regressional coefficients of computation of component scores by variables. If OUT is

\*nonpositive, this matrix is identical to eigenvectors NAME2. If OUT is positive, this matrix is

\*computed from loadings NAME2. You can use matrix NAME4 to compute principal component scores of

\*the data points not participated in the analysis.

\*NAME5 - principal component scores. They are DATA\*NAME4. If OUT is nonpositive, these are

\*raw scores: sums-of-squares in the columns of NAME5 = eigenvalues of matrix sscp(DATA)).

\*If OUT is positive, these are scaled scores: sums-of-squares in the columns of NAME5 = n-1 or n,

\*depending on argument DF.

EXAMPLE. Principal Component analysis based on correlations.

matrix.

get vars /variables= v1 to v9.

!KO\_zscore(vars%vars). /\*Standardize the variables, so that their analysis will

/\*be the analysis of correlations

!KO\_pcomp(vars%1%3%1%eival%loads%rloads%b%scores). /\*Run the analysis

/\*with extraction of 3 first components;

/\*The requested mode of output:

print eival. /\*Eigenvalues (component variances) and % of variance explained

print loads. /\*Loadings

print rloads. /\*Rescaled loadings (they are = loadings, if you analyze correlations)

print scores. /\*Scaled (standardized) component scores

print b. /\*Matrix of coefficients to compute the scores

end matrix.

### PRINCIPAL COMPONENTS (FOR N<P) [!KO\_pcomp2]

\*/\*!KO\_pcomp2(data%df%m%out%name1%name2%name3%name4%name5)\*/\*.

\*Version 2.

\*Principal component analysis.

\*This function is identical with function /\*!KO\_pcomp\*/ but is faster when number of columns in data

\*is greater than number of rows. And is slower than it when number of columns in data is less than number of rows.

### PRINCIPAL COMPONENTS (MATRIX INPUT) [!KO\_pca]

\*/\*!KO\_pca(cov%w%m%out%name1%name2%name3%name4)\*/\*.

\*Version 2.

\*Principal component analysis (with option to weight variables).

\*This function, unlike /\*!KO\_pcomp\*/, takes as input not data but ready matrix

\*(covariance or like), and it does not compute itself principal component scores.

\*Present is an option to supply nonequal weights (importancies) to variables, i.e. perform

\*weighted PCA.

\*Input:

\*COV - square symmetric matrix of coefficients of sscp type - i.e. correlation, covariance,

\*cosine similarity or raw sscp matrix of association coefficients between variables.

\*W - vector of length as the number of rows/columns of COV, containing weights for the variables;

\*or specify a nonpositive scalar (to mean - no weights supplied, results will be identical

\*with /\*!KO\_pcomp\*/). Weights must be positive values in ratios you need (only ratios between

\*weights matter). Weight exactly 0 is not allowed, however you may give very close to zero positive

\*weight if you aim to make some variable "passive", supplementary.

\*Arguments M and OUT - same as in function /\*!KO\_pcomp\*/.

\*Results:

\*NAME1, NAME2, NAME3, NAME4 - analogous to those in function /\*!KO\_pcomp\*/.

\*Component score coefficient matrix NAME4 you can use to compute principal component scores,

\*which would be equal to DATA\*NAME4, where DATA is data cases x variables (if COV is covariance

\*matrix, the variables should be centered; if COV is correlation matrix, the variables should be

\*z-standardized, etc., i.e., there must be correspondence with the matrix type).

\*Nongramian COV input:

\*If among the M extracted components there occurs such with negative eigenvalue in NAME1,

\*i.e. imaginary one(s), then its corresponding results in NAME2, NAME3, NAME4 will be computed as

\*if that component is real (positive eigenvalue), but those its results should be

\*considered imaginary.

EXAMPLE. Weighted Principal Component analysis.

matrix.

get vars /variables= v1 to v9.

!KO\_cov(vars%cov).

compute w= {1,1.2,1,1.8,2,1,2.3,1,1}. /\*Weights (importances) of the variables

!KO\_pca(cov%w%3%1%eival%loads%rloads%b). /\*Run the analysis with

/\*extraction of 3 first components;

/\*The requested mode of output:

print eival. /\*Eigenvalues (component variances) and % of variance explained

print loads. /\*Loadings

print rloads. /\*Rescaled loadings (they are = loadings if you analyze correlations)

print b. /\*Matrix of coefficients to compute the scores

!KO\_center(vars%vars).

print (vars\*b). /\*Component scores

end matrix.

### ORTHOGONAL PROCRUSTES ROTATION [!KO\_procr]

\*/\*!KO\_procr(fit%targ%refl%iso%name1%name2)\*/\*.

\*Version 1.

\*Performs orthogonal procrustes rotation of fit configuration FIT into target

\*configuration TARG. The fitted (rotated) fit configuration is returned as NAME2,

\*and the rotation matrix as NAME1.

\*FIT and TARG - matrices of equal size. Their rows are data points and columns are dimensions (axes)

\*of space. The same row in the two matrices is a pair of correspondent points.

\*FIT and TARG each have their own space dimensions. Procrustes rotation is the rotation of FIT axes

\*into TARG axes which provides the maximal juxtaposition of the corresponding points.

\*NAME2 are the coordinates of FIT points in the space of TARG axes after the rotation.

\*If number of dimensions differ in FIT and TARG, add zero-filled columns to one of the matrices

\*because the function requires that the number of columns be equal in both.

\*In rotation matrix NAME1 rows correspond to axes of FIT and columns correspond to axes of TARG.

\*REFL is a scalar; if positive, the rotation is allowed to be with reflection (if it enhances the fitting);

\*if nonpositive, reflection is prohibited (fitting might be worse).

\*ISO is a scalar; if positive, additional operation of isoscaling (for better fitting) is applied

\*after rotation; if nonpositive, isoscaling is not used. This argument does not affect

\*the rotation matrix NAME1.

\*The function takes FIT and TARGET data "as is". If need translation (superposition of centroids)

\*and/or scaling (equalizing magnitudes) of both configurations prior operation of procrustes rotation

\*do these transformations in advance.

### LINEAR DISCRIMINANTS [!KO\_discrim]

\*/\*!KO\_discrim(data%dummy%m%out%name1%name2%name3%name4%name5)\*/\*.

\*Version 2.

\*Performs linear discriminant analysis (without classifying and tests of significance):

\*extracts discriminants.

\*Takes data DATA (rows - points, columns - axes of space) and grouping DUMMY.

\*Columns of DATA (variables) must be already centered (or standardized). Columns of DATA must be

\*noncollinear.

\*DUMMY - binary dummy (indicator) variables marking groups (each variable corresponds to a group,

\*with value 1 = the case belongs to it, 0 = doesn't belong to it).

\*Each column in DUMMY must contain at least one 1.

\*The groups must be disjoint by case membership, and each case must belong to a group,

\*i.e. sum in each row in DUMMY must be equal to 1.

\*Argument M (scalar) - how many discriminants to extract. Specify positive integer from 1 to

\*min(number\_groups-1,number\_variables). Or specify any nonpositive number: then all nonzero

\*discriminants will be extracted.

\*Argument OUT (scalar) controls the form of results returned.

\*Results:

\*NAME1 - two-column matrix: eigenvalues in the 1st col, canonical correlations in the 2nd.

\*NAME2 - eigenvectors with normalized columns (if OUT nonpositive) or raw (unstandardized) canonical

\*discriminant coefficients (if OUT positive).

\*NAME3 - standardized canonical discriminant coefficients (suitable for interpretation of the

\*discriminants).

\*NAME4 - pooled within-group correlations between variables and discriminants (structure matrix,

\*suitable for interpretation of the discriminants).

\*NAME5 - discriminant scores, they are DATA\*NAME2, i.e., computed with the help of normalized

\*eigenvectors (if OUT nonpositive) or with the help of discriminant coefficients (if OUT positive);

\*in the latter case the scores have this quality that their pooled within-group covariance matrix

\*is the identity matrix.

\*If you need to supply the vector of constants to the unstandardized coefficients (to obtain

\*un-centered discriminants) then it is equal: -csum(mdiag(MEAN)\*NAME2), where MEAN is the vector

\*of means of the original variables and NAME2 is the unstandardized canonical discriminant

\*coefficients.

\*You can create dummy variables DUMMY out of a categorical grouping variable with

\*the help of function /\*!KO\_freq\*/.

\*To classify, use for instance function /\*!KO\_gaclass\*/.

EXAMPLE.

matrix.

get vars /variables= v1 to v9.

get group /variable= group.

!KO\_center(vars%vars). /\*You normally SHOULD center variables before

!KO\_freq(group%1%dummy%freq%codes). /\*Grouping variable into dummies

!KO\_discrim(vars%dummy%0%1%cancorr%coef%scoef%corr%scores).

print cancorr. /\*Eigenvalues and canonical correlations

print scoef. /\*Standardized coefficients

print corr. /\*Correlations (structure matrix)

save scores /outfile= \*.

end matrix.

EXAMPLE. Discriminant analysis with subsequent varimax rotation of discriminants (you can do the same thing by SPSS command DISCRIMINANT through its syntax).

set mxloops 1E6.

matrix.

get vars /variables= v1 v2 v3 v4 v5.

get group /variable= cluster\_.

\*Extract discriminants.

!KO\_center(vars%vars).

!KO\_freq(group%1%dummy%freq%codes).

!KO\_discrim(vars%dummy%0%1%cancorr%coef%scoef%corr%scores).

print scoef /title 'Standardized coefficients' /format= f8.5.

print corr /title 'Correlations (structure matrix)' /format= f8.5.

\*----.

\*Perform varimax rotation of the structure matrix.

print /title '--- Varimax-rotated will be Structure matrix ---'.

!KO\_ortrot(corr%1%VARIMAX%{.00001,50}%0%rotcorr%rotmat1%name3).

print rotmat1 /title 'Varimax rotation (transformation) matrix' /format= f8.5.

print rotcorr /title 'Rotated structure matrix' /format= f8.5.

print (scoef\*rotmat1) /title 'Corresponding st. coefficients matrix' /format= f8.5.

print (scores\*rotmat1)(1:10,:) /title 'Corresponding discr. scores (first 10 shown)'

/format= f8.5.

\*----.

\*Another approach is to perform varimax rotation of the coefficients matrix.

print /title '--- Varimax-rotated will be St. coefficients matrix ---'.

!KO\_gcov(vars%dummy%CORR%POOL%pooledr). /\*See Note below; Get the pooled

/\*within-group corr matrix of variables

!KO\_image(pooledr%1%image). /\*Diagonal of its image matrix contains squared

/\*multiple correlation coefficients (to use in the custom

/\*Kaiser normalization)

!KO\_ortrot(scoef%diag(image)%VARIMAX%{.00001,50}%0%rotwei%rotmat2%name3).

print rotmat2 /title 'Varimax rotation (transformation) matrix' /format= f8.5.

print rotwei /title 'Rotated st. coefficients matrix' /format= f8.5.

print (corr\*rotmat2) /title 'Corresponding structure matrix' /format= f8.5.

print (scores\*rotmat2)(1:10,:) /title 'Corresponding discr. scores (first 10 shown)'

/format= f8.5.

end matrix.

Note. We follow SPSS Algorithms (DISCRIMINANT) expressing that, when varimax rotating the st. discriminant coefficients, Kaiser normalization is done by multiple correlation coefficients; therefore we first have to get to know these.

### GAUSSIAN CLASSIFIER [!KO\_gaclass]

\*/\*!KO\_gaclass(data%grmean%grcov%prior%name1%name2%name3)\*/\*.

\*Version 2.

\*Takes data DATA (rows - points, columns - axes of space) and characteristics of k (k>=2)

\*classes (groups) in the form of their centroids coordinates GRMEAN and covariance matrices GRCOV.

\*Based on squared Mahalanobis distances (z^2) of the points to the group centroids,

\*computes probabilities Prob of belonging of the points to the groups and assigns each point to a

\*group. The Gaussian classifier is an example of a Bayes classifier by scale variables,

\*and its core is the Gaussian function Prob ~ exp(-z^2/2), that is a feature, to note, behind the

\*bell-shaped curve of the normal distribution, - therefore Gaussian classifier is optimal to

\*enroll observations in normally distributed or at least bell-like populations.

\*In particular, the given classifier is often used in discriminant analysis.

\*(In discriminant analysis, classification of cases is done by data which are discriminant

\*function scores.)

\*Input:

\*DATA - n points (cases) x p variables (dimensions). These are points to classify.

\*GRMEAN - k classes x p variables. These are means (centroids) of the k classes (groups).

\*GRCOV - p x p covariance matrix or matrices of the classes. A covariance matrix/s must be

\*positive definite. There are three modes to specify GRCOV:

\*(i) k matrices stacked one under another, - i.e., to specify own cov. matrix for each class;

\*so GRCOV will be sized k\*p x p.

\*(ii) one p x p covariance matrix for all the classes, typically being the pooled within-group

\*covariance matrix.

\*(iii) specify scalar 0 instead of the matrix, - then p x p identity matrix

\*will be assumed for every group (in this situation, Mahalanobis distances will turn

\*into Euclidean ones).

\*GRMEAN and GRCOV may originate from the sample, DATA, as well as not originate. If GRMEAN and GRCOV

\*were computed on the dataset making the rows of DATA, then the task: to classify the analyzed

\*cases to the observed classes. (Group covariance matrices and their pooled matrix can be obtained

\*from data by function /\*!KO\_gcov\*/.) If GRMEAN and GRCOV were not derived from the dataset

\*of n cases DATA, that means the task: to classify "new" (or "holdout") cases to the "suggested"

\*classes.

\*PRIOR - column vector of length k; it is the apriori probabilities to assign to the classes (and

\*serving as weights when computing the posterior probabilities Prob). Specify nonnegative values,

\*their sum must be positive. The magnitude of the numbers themselves plays no role since

\*the function normalizes the vector to sum of its elements =1 at input.

\*Results:

\*NAME1 - squared Mahalanobis distances between DATA points and the class centroids, n x k.

\*NAME2 - posterior probabilities Prob to assign points to the classes, n x k (sum in each row =1).

\*NAME3 - the result of the case assignment, n x 2. The 1-st column shows the class number where

\*the classifier assigned the case to - it is the class for which Prob is maximal in that row

\*of NAME2. The 2-nd column shows the class number for which Prob is the second maximal in that

\*row of NAME2.

\*Covariance matrix errors. If GRCOV contains a covariance matrix not positive definite,

\*the function won't work correctly. Either SPSS will issue an error from the "INV" side

\*or from the "SQRT" side, or some sq. Mahalanobis distances will appear negative in the output.

\*We recommend before running the function to check the cov. matrices: eval(m), where m is a square

\*covariance matrix. All the eigenvalues should be positive.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

get data /variables= v1 v2 v3. /\*Scale variables which cases to classify

\*Two classes are suggested.

compute grmean= {-5.2, .6, 3.1;

2.4, -1.2, 3.3}. /\*The means of the classes

compute grcov= {2.1, 1.3, .4;

1.3, 1.8, -1.1;

.4, -1.1, 2.0;

2.6, 1.8, .1;

1.8, 2.3, -1.3;

.1, -1.3, 1.9}. /\*The covariances in the classes (2 stacked matrices)

compute prior= {.45;.55}. /\*Apriori probabilities (saliences) of the classes

print grmean /rlabels= 'class1' 'class2' /clabels= 'v1' 'v2' 'v3'.

print grcov /rlabels= 'v1' 'v2' 'v3' 'v1' 'v2' 'v3' /clabels= 'v1' 'v2' 'v3'.

print prior /rlabels= 'class1' 'class2'.

\*Let us make sure the cov. matrices are positive definite.

print eval(grcov(1:3,:)).

print eval(grcov(4:6,:)).

\*Classify data cases to the classes.

!KO\_gaclass(data%grmean%grcov%prior%smah%post%class).

print smah /title 'Sq. Mahalanobis distances' /clabels= 'class1' 'class2'.

print post /title 'Posterior probabilities' /clabels= 'class1' 'class2'.

print class /title 'Class of prediction' /clabels= 'TheClass' '2ndBest'.

end matrix.

EXAMPLE. Discriminant analysis with classification.

matrix.

get vars /variables= v1 v2 v3 v4. /\*Analysis variables

get group /variable= group. /\*Grouping variable (say, k=3 classes)

\*-----.

\*Let us hold out two first cases from the analysis.

compute hold= vars(1:2,:). /\*These two cases will be treated as "new cases"

/\*to classify

compute vars= vars(3:nrow(vars),:). /\*So cut them out from the data

compute group= group(3:nrow(group)). /\*and from here

\*-----.

\*Do with the analysis data.

!KO\_freq(group%1%dummy%freq%codes). /\*Grouping variable into dummies

compute k= ncol(dummy).

print {codes,freq} /title 'Groups in the analysis' /clabels 'Code' 'Count'.

!KO\_mean(vars%mean). /\*Means of the variables

compute vars= vars-make(nrow(vars),1,1)\*mean. /\*Normally SHOULD center variables

/\*before applying /\*!KO\_discrim\*/

!KO\_discrim(vars%dummy%0%1%cancorr%coef%scoef%corr%dis). /\*Discriminant analysis

/\*extracting min(3-1, 4)=2 discriminant functions; we will need

/\*their scores "dis"; because in discriminant analysis, the discr

/\*functions are used as classifiers

\*Obtain the discriminant scores also for the hold-out cases.

compute hold= hold-make(nrow(hold),1,1)\*mean. /\*Center first by

/\*the mean of the analysis variables

compute holddis= hold\*coef. /\*And compute them discriminant scores

\*Compute group means of discriminant scores for the analysis cases.

!KO\_aggr(dis%dummy%MEAN%grmean).

print grmean /title 'Means of the 2 discriminants in the 3 groups.'.

\*We need a covariance matrix or matrices of discriminant scores. This:.

!KO\_gcov(dis%dummy%COV%BOTH%grcov). /\*outputs, stacked, the pooled

/\*within-group covariance matrix and the group covariance matrices

print grcov /title 'Pooled and separate covariance matrices of discriminants'.

/\*Note that the pooled within-group covariance matrix of discriminant

/\*scores is the identity matrix

\*-----.

\*Perform classification of both the analysis and the hold-out cases.

!KO\_gaclass({holddis;dis}%grmean%0%make(k,1,1)%smah%post%class).

/\*In this run, we use the pooled within-group cov matrix

/\*and equal prior probabilities [this setting is default in DISCRIMINANT]

/\*Since the pooled matrix for discriminants is always the identity one,

/\*we could just set GRCOV argument to scalar 0

print /title 'POOLED MATRIX AND EQUAL PRIORS USED'.

print /title 'Cases 1 and 2 were hold-out: no influence, just classified.'.

print smah /title 'Sq. Mahalanobis (= sq. Euclidean) distances'

/clabels= 'class1' 'class2' 'class3' /format= f8.3.

print post /title 'Posterior probabilities'

/clabels= 'class1' 'class2' 'class3' /format= f8.5.

print class /title 'Group of assignment' /clabels= 'TheClass' '2ndBest'.

\*-----.

!KO\_gaclass({holddis;dis}%grmean%grcov(3:nrow(grcov),:)%freq%smah%post%class).

/\*In this run, we use individual within-group cov matrices,

/\*and prior probabilities proportional to the group sizes

print /title 'SEPARATE MATRIX AND UNEQUAL PRIORS USED'.

print /title 'Cases 1 and 2 were hold-out: no influence, just classified.'.

print smah /title 'Sq. Mahalanobis distances'

/clabels= 'class1' 'class2' 'class3' /format= f8.3.

print post /title 'Posterior probabilities'

/clabels= 'class1' 'class2' 'class3' /format= f8.5.

print class /title 'Group of assignment' /clabels= 'TheClass' '2ndBest'.

end matrix.

The 1st run of /\*!KO\_gaclass\*/ classification was equivalent to this run of DISCRIMINANT:

DISCRIMINANT

/GROUPS= group (1 3)

/VARIABLES= v1 v2 v3 v4

/SELECT= selvar (1) /\*Selection variable with all cases=1 but for cases 1 & 2

/ANALYSIS ALL

/SAVE= CLASS SCORES PROBS

/PRIORS EQUAL /\*Equal priors

/PLOT= CASES

/CLASSIFY= NONMISSING POOL /\*Pooled cov. matrix to use (identity one).

The 2nd run of /\*!KO\_gaclass\*/ classification was equivalent to this run of DISCRIMINANT:

DISCRIMINANT

/GROUPS= group (1 3)

/VARIABLES= v1 v2 v3 v4

/SELECT= selvar (1) /\*Selection variable with all cases=1 but for cases 1 & 2

/ANALYSIS ALL

/SAVE= CLASS SCORES PROBS

/PRIORS SIZE /\*Priors proportional to group sizes

/PLOT= CASES

/CLASSIFY= NONMISSING SEPAR /\*Separate group matrices to use.

### GAUSSIAN CLASSIFIER (WITH OPTION "LEAVE-ONE-OUT") [!KO\_gaclass2]

\*/\*!KO\_gaclass2(data%dummy%method%prior%name1%name2%name3)\*/\*.

\*Version 1.

\*Takes data DATA (rows - points, columns - axes of space) and group belonging of these

\*cases, DUMMY. Based on squared Mahalanobis distances of the points to the group centroids,

\*computes probabilities Prob of belonging of the points to the groups and assigns each point to a

\*group.

\*This function differs from /\*!KO\_gaclass\*/ - also Gaussian classifier - by the following:

\*- group means (centroid coordinates) and covariances between the variables are not input, the

\* function takes them from the data;

\*- there's no option to use separate covariance matrix for each group, only the pooled

\* within-group covariance matrix is used;

\*- all the cases must have a group membership;

\*- there's the option of leave-one-out classification (cross-validation).

\*Function /\*!KO\_gaclass2\*/ is a slightly re-written old macro "disclass", in former times

\*appended to SPSS, and it gives same results.

\*Input:

\*DATA - n points (cases) x p variables (dimensions). These are points to classify.

\*DUMMY - n x k: their observed belonging to the k classes (groups), binary dummy variables

\*(each variable corresponds to a group, with value 1 = the case belongs to it,

\*0 = doesn't belong to it). Each column in DUMMY must contain at least one 1.

\*The groups must be disjoint by membership: sum in each row in DUMMY must not exceed 1.

\*In DUMMY, there can be no rows complete of zeros - cases out of the groups.

\*(Use /\*!KO\_gaclass\*/ to classify "new" cases, which group membership is unknown.)

\*Argument METHOD - scalar. If nonpositive, the function will perform usual classification -

\*a classification without cross-validation. But if positive, the function will perform

\*cross-validation as "leave-one-out" classification (synonyms: U-method cross-validation,

\*jackknifing cross-validation). With this method, each case (data point), before it is

\*classified, is temporarily excluded from the sample and is considered a sized 1 test dataset,

\*whereas the other n-1 cases form the training dataset from which the classify rule for that

\*one case is derived. Each case thus doesn't influence its own classification.

\*PRIOR - column vector of length k; it is the apriori probabilities to assign to the classes (and

\*serving as weights when computing the posterior probabilities Prob). Specify nonnegative values,

\*their sum must be positive. The magnitude of the numbers themselves plays no role since

\*the function normalizes the vector to sum of its elements =1 at input.

\*Results:

\*NAME1 - squared Mahalanobis distances between DATA points and the class centroids, n x k.

\*NAME2 - posterior probabilities Prob to assign points to the classes, n x k (sum in each row =1).

\*NAME3 - the result of the case assignment, n x 2. The 1-st column shows the class number where

\*the classifier assigned the case to - it is the class for which Prob is maximal in that row

\*of NAME2. The 2-nd column shows the class number for which Prob is the second maximal in that

\*row of NAME2.

\*Covariance matrix error. If the pooled within-group covariance (or scatter) matrix is singular,

\*SPSS will issue an error from the "INV" side. In such situation, try to modify input values

\*of VARS slightly, for example, add tiny random noise. (You can get the said pooled matrix,

\*to test it, with function /\*!KO\_gcov\*/ or /\*!KO\_bwscat\*/.)

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your grouping is defined by a categorical variable, create dummy variables DUMMY out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE. Usual classification.

set mxloops 1E6.

matrix.

get vars /variables= v1 v2 v3 v4.

get gr /variable= class.

!KO\_freq(gr%1%dummy%freq%codes). /\*Grouping variable into dummies

print {t(1:ncol(dummy)),codes,freq} /title 'Classes (groups) observed'

/clabels 'GrNumber' 'Code' 'Count'.

compute prior= make(ncol(dummy),1,1). /\*Let priors for the classes be equal

\*Perform usual classification.

!KO\_gaclass2(vars%dummy%0%prior%smah%post%class).

print smah /title 'Sq. Mahalanobis distances, cases x classes' /format= f8.5.

print post /title 'Posterior probabilities, cases x classes' /format= f8.5.

print class /title 'Group of assignment' /clabels= 'TheClass' '2ndBest'.

\*Compare the input classification with the predicted.

compute grnum= dummy\*t(1:ncol(dummy)). /\*Dummies to categorical variable

!KO\_crosstab(grnum%class(:,1)%0%count%nums1%nums2).

print count /title 'Observed (rows) vs Predicted (cols) classification'.

print /title '(Classes go by their number "GrNumber" order)' /space= 0.

\*-----.

\*Get the same results with /\*!KO\_gaclass\*/.

!KO\_aggr(vars%dummy%MEAN%grmean). /\*Observed group means

!KO\_gcov(vars%dummy%COV%POOL%poolcov). /\*Pooled within-group cov matrix

!KO\_gaclass(vars%grmean%poolcov%prior%smah%post%class).

print smah /title 'Sq. Mahalanobis distances, cases x classes' /format= f8.5.

print post /title 'Posterior probabilities, cases x classes' /format= f8.5.

print class /title 'Group of assignment' /clabels= 'TheClass' '2ndBest'.

end matrix.

EXAMPLE. Leave-one-out classification.

set mxloops 1E6.

matrix.

get vars /variables= v1 v2 v3 v4.

get gr /variable= class.

!KO\_freq(gr%1%dummy%freq%codes). /\*Grouping variable into dummies

print {t(1:ncol(dummy)),codes,freq} /title 'Classes (groups) observed'

/clabels 'GrNumber' 'Code' 'Count'.

compute prior= make(ncol(dummy),1,1). /\*Let priors for the classes be equal

!KO\_gaclass2(vars%dummy%1%prior%smah%post%class).

print smah /title 'Sq. Mahalanobis distances, cases x classes' /format= f8.5.

print post /title 'Posterior probabilities, cases x classes' /format= f8.5.

print class /title 'Group of assignment' /clabels= 'TheClass' '2ndBest'.

\*Compare the input classification with the predicted.

compute grnum= dummy\*t(1:ncol(dummy)). /\*Dummies to categorical variable

!KO\_crosstab(grnum%class(:,1)%0%count%nums1%nums2).

print count /title 'Observed (rows) vs Predicted (cols) classification'.

print /title '(Classes go by their number "GrNumber" order)' /space= 0.

print /title 'The Predicted classification was Leave-one-out' /space= 0.

end matrix.

### CANONICAL CORRELATIONS [!KO\_cancorr]

\*/\*!KO\_cancorr(set1%set2%m%out%name1%name2%name3%name4%name5%name6)\*/\*.

\*Version 2.

\*Performs canonical correlation analysis (without tests of significance): extracts canonical

\*variates.

\*Takes two different sets of variables - SET1 (n x p1) and SET2 (n x p2), with the same cases (rows).

\*The variables (columns) must be noncollinear within the sets.

\*The variables normally should be already centered (or standardized), in order to obtain classical

\*canonical correlation analysis.

\*Argument M (scalar) - how many canonical correlations (i.e., canonical variate paires) to extract.

\*Specify positive integer from 1 to min(p1,p2). Or specify any nonpositive number: then

\*all nonzero canonical correlations will be extracted: m will equal their number.

\*Argument OUT (scalar) controls the form of results returned.

\*Results:

\*NAME1 - two-column matrix: eigenvalues in the 1st col, canonical correlations in the 2nd.

\*NAME2 - raw (unstandardized) canonical coefficients (if OUT positive) or they with normalized

\*columns (if OUT nonpositive).

\*NAME3 - standardized canonical coefficients (suitable for interpretation of the variates).

\*NAME4 - loadings, or structure matrix - it is correlations between the variables and the variates

\*inside the sets (suitable for interpretation of the variates).

\*NAME5 - cross-loadings - it is correlations between the variables and the variates across the sets.

\*Sizes of NAME2, NAME3, NAME4, NAME5 are equal, p1+p2 x m, i.e., variables of two sets x canonical

\*pairs, i.e., canonical correlations.

\*NAME6 - canonical variate scores (n x 2\*m), they are SET1\*NAME2(first p1 rows) - first their m

\*columns, and SET2\*NAME2(last p2 rows) - last their m columns. If OUT positive, standardized

\*scores (st. dev. in each variate =1) come out; if OUT nonpositive, raw scores come out

\*(but generally, there exist various versions how to compute raw canonical scores). In either case,

\*variates inside each set always don't correlate by thier scores, while between sets nonzero

\*correlations exist only in the pairs of corresponding variates - and they equal the canonical

\*correlations NAME1.

EXAMPLE.

matrix.

get set1 /variables= u1 to u5 /names= names1.

get set2 /variables= w1 w2 w3 /names= names2.

compute names= {names1,names2}.

!KO\_center(set1%set1). /\*Normally should center

!KO\_center(set2%set2). /\*all the variables

!KO\_cancorr(set1%set2%0%1%cancorr%coef%scoef%load%crload%scores).

print cancorr /title 'Eigenvalues and canonical correlations'

/clabels 'Eigen' 'CanCorr' /rlabels 'I' 'II' 'III' /format= f8.4.

print coef /title 'Canonical coefficients' /clabels 'I' 'II' 'III' /rnames= names.

print scoef /title 'Standardized canonical coefficients'

/clabels 'I' 'II' 'III' /rnames= names.

print load /title 'Canonical loadings' /clabels 'I' 'II' 'III' /rnames= names.

print crload /title 'Canonical cross-loadings'

/clabels 'I' 'II' 'III' /rnames= names.

print scores(1:10,:) /title 'Canonical scores (first 10 shown)'

/clabels 'I,set1' 'II,set1' 'III,set1' 'I,set2' 'II,set2' 'III,set2'.

print /title 'Redundancy analysis:'.

compute p1= ncol(set1).

compute p2= ncol(set2).

compute p= p1+p2.

print (cssq(load(1:p1,:))/p1)

/title "Prop. of variance in set1 explained by set1's variates".

print (cssq(load((p1+1):p,:))/p2)

/title "Prop. of variance in set2 explained by set2's variates".

print (cssq(crload(1:p1,:))/p1)

/title "Prop. of variance in set1 explained by set2's variates".

print (cssq(crload((p1+1):p,:))/p2)

/title "Prop. of variance in set2 explained by set1's variates".

end matrix.

### PRINCIPAL COORDINATES [!KO\_pcoord]

\*/\*!KO\_pcoord(dis%m%method%name1%name2)\*/\*.

\*Version 1.

\*Principal coordinate analysis (= Torgerson's metric multidimensional scaling).

\*Takes matrix of dissimilarities DIS, which it treats as squared (sic!) distances, and embeds

\*these objects in euclidean space, returning the coordinates as NAME2.

\*Dimensions (columns of NAME2) are uncorrelated variables.

\*DIS must be square symmetric, num\_objects x num\_objects.

\*M (integer scalar) - needed dimensionality of the euclidean space: number from 1 to num\_objects,

\*or a nonpositive number (then M will be = all really existing nonredundant dimensions).

\*Argument METHOD (scalar) is the method of decomposition of the double-centration matrix, the intermediate

\*product of DIS: eigen-decomposition (with METHOD positive) or svd-decomposition (with METHOD nonpositive).

\*All eigen or singular values of these decompositions are returned as vector NAME1.

\*The difference between the methods is the following: possible negative eigenvalues of the double-centration

\*matrix are discovered by eigen-decomposition, and the function discards them further, but corrects with

\*them the sum of positive eigenvalues. Negative eigenvalues are not recognized by svd-decomposition and

\*come out as junior singular values. From the point of view of accurateness of reproduction of input

\*dissimilarities DIS by resultant euclidean distances (the distances between the objects in the obtained

\*configuration NAME2), version with eigen-decomposition is preferable, as a rule.

\*If DIS is squared euclidean distances, than both methods are same, and coordinates NAME2 are then identical

\*to raw component scores which would be obtained by Principal component analysis of those data (centered)

\*from which one might compute the distances DIS. (This is an evidence of the tight kindred link between

\*Principal coordinate analysis and Principal component analysis.).

### BIPLOT [!KO\_biplot]

\*/\*!KO\_biplot(data%rw%cw%norm%m%name1%name2%name3%name4)\*/\*.

\*Version 1.

\*Biplot with option to weight rows and columns.

\*Returns coordinates of row points and column points, and other statistics.

\*This function also can be considered performing weighted Principal component analysis (see below).

\*DATA - arbitrary data matrix (at least two rows and columns). The analysis takes the data as is,

\*therefore if it is necessary do the standardization you need in advance.

\*RW - weight for each row of DATA (column vector) or a nonpositive scalar (which means no weighting:

\*the program will give equal weight to all rows).

\*CW - weight for each column of DATA (row vector) or a nonpositive scalar (which means no weighting:

\*the program will give equal weight to all columns).

\*Specified weights can be any nonnegative numbers, only ratios between weights matter.

\*Specified weight 0 for row/column means that this row/column must be passive (supplementary).

\*Every positive weight designates active row/column.

\*NORM - vector of two numbers from 0 to 1. These are coefficients of normalizing, or endowing with

\*inertia, of coordinates of row points (first number) and column points (second number). Coefficient 0

\*means "standard normalization" when scale of coordinates is standardized to 1. Coefficient 1 means

\*"principal normalization"when all the scale of eigenvalues of dimensions is given to the scale of

\*coordinates. For example, {0,1} is standard normalization for row points and principal normalization

\*for column points; {1,1} is principal normalization for these and those; {.5,.5} is so called

\*"symmetric normalization".

\*M - how many dimensions (principal axes) to extract: integer from 1 to

\*min(num\_active\_rows,num\_active\_cols) in DATA. Or specify any nonpositive number - then all nonzero

\*dimensions will be extracted.

\*Results:

\*NAME1 - two-column matrix: eigenvalues (scale, inertia) of dimensions (1st col) and proportion of

\*explained overall inertia of DATA (2nd col).

\*In NAME2, NAME3, NAME4 rows are points of biplot: first go row points of DATA, then, below them,

\*column points of DATA.

\*NAME2 - two-column matrix: point mass (1st col) and point inertia (2nd col).

\*Mass of an active row or column is proportional to its weight. Mass of a passive row or column is

\*proportional to the averaged weight of active rows or columns, respectively.

\*Inertia of a point is its scale in the normalized (weighted) matrix DATA. If there were no weighting

\*or weights for all active rows/columns were equal then the inertia of a row/column in its sum of

\*squares in DATA divided by r\*c (r = number of active rows, c = number of active columns).

\*NAME3 - coordinates of row points and column points in the space of M dimensions. Way of

\*normalization by inertia (argument NORM) affects the coordinates.

\*NAME4 - contributions of points to inertia of dimensions (first M cols) and contributions of dimensions

\*to inertia of points (last M cols). Contributions of points to dimensions are the squared elements of

\*eigenvectors. Points with high contribution to inertia of a dimension are the points to whom the

\*dimension chiefly owes its emergence. Contribution of passive points is 0 because they do not influence

\*the formation of dimensions. Contribution of dimension to inertia of point is the index of completeness

\*of explanation of the point by the given dimension; high contribution means that the point is well

\*represented via this dimension and thus it is suffice to "explain" the point.

\*RELATION to Principal component analysis (PCA). Biplot and PCA are almost identical: the only

\*difference is in that biplot brings the sum of column weights of input data to 1, and PCA - to the

\*number of (active) columns. This makes results of the two methods proportionally identical,

\*and the current function is usable to execute the principal component analysis, including PCA with

\*weighting of importancies of data rows and/or columns. Specifically, let C be the number of active

\*columns, then the following relationships are true, to convert results:

\*1) eigenvalues of PCA = eigenvalues of biplot \* C;

\*2) loadings = column coordinates under "principal normalization" of columns;

\*3) standardized component scores = row coordinates under "standard normalization" of rows;

\*4) eigenvectors of PCA = column coordinates under "standard normalization" of columns / sqrt(C);

\*5) raw component scores = row coordinates under "principal normalization" of rows \* sqrt(C).

\*On kinship of biplot, PCA, and correspondence analysis - https://stats.stackexchange.com/q/141754/3277.

EXAMPLE. Perceptual map of 13 countries (with supplementary categories).

data list list /country (a8) LIVING CLIMATE FOOD SECUR HOSPIT INFRA (6f2).

begin data

Italy 7 8 9 5 3 7

Spain 7 9 9 5 2 8

Croatia 5 6 6 6 5 6

Brazil 5 8 7 3 2 3

Russia 6 2 2 3 7 6

Germany 8 3 2 8 7 9

Turkey 5 8 9 3 1 3

Morocco 4 7 8 2 1 2

Peru 5 6 6 3 4 4

Nigeria 2 4 4 2 3 2

France 8 4 7 7 9 8

Mexico 2 5 5 2 3 3

SouthAfr 4 4 5 3 3 3

end data.

variable level all (sca) country (nom).

dataset name data13countries.

matrix.

get data /variables= living climate food secur hospit infra

/names= varname.

get country /variable= country.

compute point= {country;t(varname)}.

!KO\_center(data%data). /\*Perhaps center the features

compute rw= {1;1;0;1;1;1;1;1;1;1;1;1;0}. /\*Weights for rows: Croatia and

/\*South Africa will be declared supplementary rows (no influence

/\*on the definition of dimensions), the rest will have equal influence

!KO\_biplot(data%rw%0%{.5,.5}%2%eival%masine%coord%contr).

/\*Apply biplot to extract coordinates in 2 dimensions;

/\*Let us spread inertia (variability) half between rows and columns

print eival /title 'Inertia of dimensions' /clab 'Eigenval' 'Prop'.

print masine /title 'Points mass and inertia' /clab 'Mass' 'Inertia' /rname= point.

print coord /title 'Points coordinates in dimensions I and II'

/clab 'I' 'II' /rname= point.

print contr /title 'Contributions' /clab '->I' '->II' '<-I' '<-II' /rname= point.

save {point,coord} /outfile= \* /variables= point dim1 dim2 /string= point.

end matrix.

GRAPH /SCATTERPLOT(BIVAR)= dim1 WITH dim2 BY point (IDENTIFY).

### SIMPLE CORRESPONDENCE ANALYSIS [!KO\_corresp]

\*/\*!KO\_corresp(tab%sta%rsup%csup%norm%m%name1%name2%name3%name4)\*/\*.

\*Version 1.

\*Simple (two-way) correspondence analysis. It is an application of the biplot idea to the analysis of

\*contingency table. Returns coordinates of row points and column points, and other statistics.

\*TAB - contingency table (at least two rows and columns). It is any table with nonnegative entries,

\*in which a value in a cell has meaning of contingency, proximity between the row and the column.

\*In TAB, row sums from active columns must be all nonzero, and column sums from active rows must be all

\*nonzero (active rows/columns are not passive rows/columns).

\*STA - needed standardization for the table, integer from 1 to 6:

\*1 - two-way centering and weighted normalization

\*2 - two-way centering and simple normalization

\*3 - centering of rows and simple normalization

\*4 - centering of columns and simple normalization

\*5 - equalizing row sums, centering of rows and simple normalization

\*6 - equalizing column sums, centering of columns and simple normalization

\*Option (1) is chi-square model of analysis, it is optimal if TAB is a frequency table. The rest five

\*options to choose is euclidean model of analysis, it suits if TAB is not frequencies (but, for

\*example, sums or means). Chi-square model implies biplot with weighting.

\*RSUP - vector of row numbers that should be treated as passive (supplementary), or any nonpositive

\*scalar if no passive are needed. CSUP - analogously, vector of column numbers that should be treated

\*as passive, or any nonpositive scalar. Rows and columns that are not passive are considered active.

\*NORM - vector of two numbers from 0 to 1. These are coefficients of normalizing, or endowing with

\*inertia, of coordinates of row points (first number) and column points (second number). Coefficient 0

\*means "standard normalization" when scale of coordinates is standardized to 1. Coefficient 1 means

\*"principal normalization" when the all scale of eigenvalues of dimensions is given to the scale of

\*coordinates. For example, {0,1} is standard normalization for row points and principal normalization

\*for column points; {1,1} is principal normalization for these and those; {.5,.5} is so called

\*"symmetric normalization".

\*M - how many dimensions to extract: integer from 1 to min(num\_active\_rows-1,num\_active\_cols-1) in TAB.

\*Or specify any nonpositive number - then all nonzero dimensions will be extracted.

\*Results (they are of the same kind as in biplot /\*!KO\_biplot\*/):

\*NAME1 - two-column matrix: eigenvalues (scale, inertia) of dimensions (1st col) and proportion of

\*explained overall inertia of TAB transformed by standardization (2nd col).

\*In NAME2, NAME3, NAME4 rows are points of biplot: first go row points of TAB, then, below them, column

\*points of TAB.

\*NAME2 - two-column matrix: point mass (1st col) and point inertia (2nd col).

\*Mass of an active row or column is proportional to its weight. Mass of a passive row or column is

\*proportional to the averaged weight of active rows or columns, respectively.

\*Inertia of a point is its scale (sum of squares) in the standardized TAB.

\*NAME3 - coordinates of row points and column points in the space of M dimensions. Way of

\*normalization by inertia (argument NORM) affects the coordinates.

\*NAME4 - contributions of points to inertia of dimensions (first M cols) and contributions of

\*dimensions to inertia of points (last M cols). Contributions of points to dimensions are the squared

\*elements of eigenvectors. Points with high contribution to inertia of a dimension are the points to

\*whom the dimension chiefly owes its emergence. Contribution of passive points is 0 because they do

\*not influence the formation of dimensions.

\*Contribution of dimension to inertia of point is the index of completeness of explanation of the

\*point by the given dimension; high contribution means that the point is well represented via this

\*dimension and thus it is suffice to "explain" the point.

EXAMPLE. Correspondence analysis of a frequency table.

matrix.

get vars /variables= v1 v2. /\*take two numeric categorical variables

!KO\_crosstab(vars(:,1)%vars(:,2)%1%table%rowcat%colcat). /\*obtain the crosstabulation

print rowcat /title 'Row categories'.

print colcat /title 'Column categories'.

print table /title 'Frequency crosstabulation'.

!KO\_corresp(table%1%0%0%{.5,.5}%2%eival%masine%coord%contr).

/\*run correspondence analysis with: chi-square model, no passive rows

/\*or columns, symmetric normalization, extraction of 2 dimensions

print eival /title 'Inertia of dimensions' /clab 'Eigenval' 'Prop'.

print masine /title 'Points mass and inertia' /clab 'Mass' 'Inertia'.

print {{rowcat;t(colcat)},coord} /title 'Points coordinates in dimensions I and II'

/clab 'Point' 'I' 'II'.

print contr /title 'Contributions' /clab '->I' '->II' '<-I' '<-II'.

end matrix.

### FACTORS BY PRINCIPAL AXIS METHOD [!KO\_paf]

\*/\*!KO\_paf(cov%comm%m%conv%name1%name2%name3%name4%name5)\*/\*.

\*Version 1.

\*Linear factor analysis (only extraction of factor loadings from a matrix; the function does not

\*do factor rotations or computation of factor scores). Principal axis extraction method is used.

\*Input:

\*COV - square p x p symmetric matrix of coefficients of sscp type - i.e. correlation, covariance,

\*cosine similarity or raw sscp matrix of association coefficients between variables.

\*COMM - column vector of initial communality estimates, or nonpositive scalar. If vector, it must be

\*of length as diagonal of COV (i.e. equal the number of variables p) and contain positive values not

\*exceeding the corresponding values of the diagonal (i.e. the scales, variances of the variables).

\*If COMM is nonpositive scalar the function will use images of COV matrix as the initial

\*communalities - as a rule, they are the best canditates for initial communalities. This option

\*requires positive definite COV.

\*M - how many factors to extract, specify positive integer less than the number of variables

\*(you must decide in advance, using known methods of estimation, how many factors to extract).

\*CONV - criteria of iterative convergence: vector of 2 numbers, for example {.001,25}. The first is the

\*threshold for communality change, nonnegative number; if the change is less than it communalities will

\*be considered stabilized and iterations will stop. The second number is the maximal allowed number of

\*iterations - specify positive integer.

\*Results:

\*NAME1 - M x 4 matrix of scales (variances) of the extracted factors and the proportions of total scale

\*explained by them. The 1st column is the scales (variances) of the factors as they are, equal to the M first

\*eigenvalues of the reduced COV at last iteration. 2nd column is the proportion of these scales of

\*the total scale equal to the trace of the input COV. 3rd column is the rescaled scales (variances), i.e.

\*such from which the effect of unequal scales (variances) of variables is removed. 4th column relates to

\*the 3rd like the 2nd to the 1st.

\*If in NAME1 there occurs factor with negative variance, it is imaginary; often rising the number of

\*iterations moves it to real; or try to lessen M.

\*NAME2 - p x M matrix of factor loadings, as they are. Their column sums of squares equal the factor scales

\*and their row sums of squares equal the final communalities.

\*NAME3 - rescaled loadings. It is the loadings from which the effect of unequal scales (variances) of variables

\*is removed; they are the correlations (or cosines) between factors and variables and are convenient in

\*interpretation of factors.

\*NAME4 - initial and final communality estimates, p x 4. Columns left to right: initial as they are,

\*final as they are, initial rescaled, final rescaled. A rescaled communality is the communality

\*divided by the scale (variance) of that variable, and is the proportion of its scale explained

\*by the common factors.

\*If COV is correlation or cosine matrix then "raw" and rescaled results

\*in NAME1, in NAME2/NAME3, in NAME4 will be identical.

\*NAME5 - report of convergence, vector of 2 values which is comparable with argument CONV:

\*the 1st value is the maximal change in communalities observed on the last iteration done and

\*which, as desired, should be less than the threshold in CONV. The 2nd number is the number of iterastions

\*done; if this appears by 1 greater than the specified in CONV then it means that the maximal number of

\*iterations was done but the requested threshold of convergernce was not reached.

\*Heywood case. If on any iteration a communality value exceeds the variable's scale then iterations will

\*cut off, results are returned as existing at the moment of break. You can see Heywood case occurred by presence

\*of a value greater than one in the last column of NAME4; besides, the function will - for attention -

\*give the 2nd value in NAME5 negative sign. Results in Heywood case should be acknowledged abortive.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. See example below.

### ORTHOGONAL ANALYTIC ROTATIONS [!KO\_ortrot]

\*/\*!KO\_ortrot(load%kaiser%method%conv%reord%name1%name2%name3)\*/\*.

\*Version 2.

\*Does different kinds of orthogonal analytic rotations combinedly known as "orthomax",

\*which usually are applied in factor analysis to rotate the extracted factors.

\*LOAD - input matrix of loadings of orthogonal factors (p variables x m factors, p and m >1),

\*that need be rotated (the result of rotation is matrix NAME1). LOAD is not necessarily

\*factor loadings, it can just be coordinates of some p points in the space of m

\*perpendicular axes.

\*KAISER - scalar. If positive, then, before the rotation, LOAD will undergo Kaiser normalization

\*(and after the rotation the result will undergo the back de-normalizing). Kaiser normalization

\*brings sums of squares in the rows of LOAD to unit. If the scalar is nonpositive, then

\*no Kaiser normalization will take place.

\*KAISER - scalar or column vector of length p.

\*If nonpositive scalar, no Kaiser normalization will take place. Otherwise it will be. Kaiser

\*normalization consists in modifying sums of squares in rows of LOAD before rotation (and after

\*the rotation the result will undergo back de-normalizing).

\*If KAISER is positive scalar, usual Kaiser normalization will be done, equalizing sums of

\*squares in rows of LOAD by dividing with sqrt(communalities).

\*If KAISER is vector, it should consist of positive numbers; There will be arbitrary Kaiser

\*normalization dividing LOAD by sqrt(those numbers).

\*METHOD - method of the rotation, capitalized keyword (may put in quotes or apostrophes).

\*This is the criterion being maximized with the given method:

\*"QUARTIMAX" - quartimax; it equals the sum of the resultant loadings (final coordinates) taken

\*to power of 4: msum(NAME1^4).

\*The other methods have the formula of the criterion p\*Q-c\*V, where Q is the quartimax criterion

\*and V is the sum of squared variances of the resultant factors, the columns of NAME1:

\*rsum(cssq(NAME1)^2). The coefficient c defines the method:

\*"VARIMAX" - varimax: c=1

\*"EQUAMAX" - equamax: c=m/2

\*"PARSIMAX" - parsimax: c=p\*(m-1)/(p+m-2)

\*"FACPARS" - facpars (factor parsimony): c=p

\*"CUSTOM c" - arbitrary value of c: set a value for it (number or scalar name). Value can be any.

\*The higher is c the farther will the configuration be from a "general factor".

\*With c approaching +infinity, factor variances, i.e. cssq(NAME1), will be equal.

\*With c approaching -infinity, factor variances will be that as when you subject LOAD to rotation

\*into its principal components (without preliminary centering).

\*CONV - criteria of iterative convergence: vector of 2 numbers, for example {.0001,50}. The first is

\*the threshold for the criterion change, nonnegative number; if the change is less than it the

\*criterion will be considered stabilized and iterations will stop (see NAME3). The second number

\*is the maximal allowed number of iterations - specify positive integer.

\*REORD - scalar. If positive, then factors after the rotation, the columns of NAME1, will be

\*reordered by descend of their variances (i.e. column sums of squares), and also in each column

\*the sign of loadings will be ascribed so that the sum of loadings there be positive. If the scalar

\*is nonpositive, these two postrotational actions won't be done.

\*Results:

\*NAME1 - the resultant matrix of loadings (coordinates).

\*NAME2 - orthogonal rotation matrix, so that NAME1=LOAD\*NAME2.

\*NAME3 - iteration history, a vector of the criterion values on leaving iterations. The 1st value of

\*the vector is the magnitude of the criterion for the input LOAD; the 2nd value is the magnitude of

\*the criterion after the 1st iteration, and so on. The last value in the vector is the magnitude of

\*the criterion on exiting the last iteration used. Therefore, there was, in all, nrow(NAME3)-1

\*iterations. A criterion's values grow from iteration to iteration until they stabilize or the limit

\*of iterations is reached. Quartimax and varimax values are always positive, but the rest of the

\*methods may show negative values too, what depends on the input data.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Note: If factor/principal component analysis has been done on covariances (and not correlations),

\*then loadings can be of two kind - raw and rescaled (the latter are obtained by dividing the former

\*by the variables's st. deviations). There's more sense to rotate towards a "simple structure" just the

\*rescaled ones, because it is they which are usually interpreted. However, if you are doing the rotation

\*with Kaiser normalization, results of rotating the rescaled and the raw loadings will be equivalent,

\*for NAME2 is the same in this case.

EXAMPLE. Factor analysis with factor orthogonal rotation (varimax).

set mxloops 10000.

matrix.

get vars /variables= v1 to v30.

!KO\_corr(vars%r). /\*Correlations

\*Factor extraction.

!KO\_paf(r%0%3%{.001,25}%fvrnc%load%rload%comm%conv). /\*Extract 3 factors

print conv /title 'Extraction Convergence report'.

print fvrnc /title 'Extraction SS loadings (factor variances)'

/clab= 'Eival' 'Prop' 'RescEiv' 'Prop'.

print load /title 'Extraction Loadings'.

\*Factor rotation.

!KO\_ortrot(load%1%VARIMAX%{.0001,50}%1%rotload%rotm%history).

/\*Varimax with Kaiser normalization

print rotload /title 'Rotation Loadings (varimax with kaiser)'.

print cssq(rotload) /title 'Rotation SS loadings (factor variances)'.

print rotm /title 'Rotation (transformation) matrix'.

print history /title 'Iteration history of the rotation criterion'.

end matrix.

### OBLIQUE ROTATION PROMAX AND PROMAJ [!KO\_promax]

\*/\*!KO\_promax(ort%kaiser%k%name1%name2%name3)\*/\*.

\*Version 1.

\*Does rotation promax or promaj of an orthogonal configuration ORT, which the function takes as

\*"already close to simple structure". ORT is a matrix of factor loadings already rotated

\*towards a "simple structure" orthogonally; in the classic variant - by varimax method (use

\*function \*/\*!KO\_ortrot\*/ for that). Promax considers ORT^K as the target configuration - the more

\*ideal "simple structure" than attained in ORT (raising elements of ORT to power K>1 while retaining

\*their sign brings small loadings to zero faster than big loadings, thus "sharpening" the pattern),

\*and tries to make ORT closer to ORT^K by loosing orthogonality of the axes (factors) through

\*nonorthogonal procrustes rotation, while ORT^K is taken for the target pattern (the loadings, not

\*the correlations). The result is an oblique pattern NAME1 obtained from the orthogonal ORT.

\*Promaj is similar to promax, but creates the target configuration for rotation not by raising

\*to power K but by another method (see Mulaik, Foundations of factor analysis, 2010).

\*Input:

\*ORT - p x m (p>=m) matrix of loadings or some coordinates in the orthogonal (cartesian) frame.

\*ORT must be full rank, i.e. t(ORT)\*ORT invertible. ORT is seen as close to "simple structure"

\*yet orthogonal so far.

\*KAISER - scalar. If positive, then the target configuration will be based of ORT undergone Kaiser

\*normalization. It is the recommended way (Kaiser normalization brings sums of squares in the rows

\*to unit). If the scalar is nonpositive, the target configuration will be based on ORT as it is.

\*In the latter case it is preferable that values of ORT don't exceed 1 in abs. magnitude.

\*K - scalar, promax parameter, the power for elements of ORT. In promax, only power >1 makes sense.

\*Most frequently, K from 2 to 4 is used.

\*If you specify a nonpositive value for K, the function will realize method promaj instead of promax.

\*The specific value of K doesn't play a role then, because promaj does not raise elements to power K.

\*Results:

\*NAME3 - matrix of promax (or promaj) transformation (axes de-orthogonalization).

\*NAME1 - resultant factor pattern matrix, = ORT\*NAME3.

\*NAME2 - resultant factor correlation matrix (angle cosines between the nonorthogonal axes in

\*NAME1 configuration); it equals inv(sscp(NAME3)).

\*The factor structure matrix is NAME1\*NAME2.

EXAMPLE. Promax after a principal component analysis.

set mxloops 10000.

matrix.

get data /variables= v1 to v10. /\*Some variables

!KO\_center(data%data). /\*We will do PCA on covariances (not correlations),

/\*therefore center (not standardize) the data

!KO\_pcomp(data%1%3%1%eival%load%rload%ssccoef%scores). /\*PCA extracting 3 principal

/\*components

print load /title 'Extracted loadings' /format f8.3. /\*The raw

print rload /title 'Extracted loadings (rescaled)' /format f8.3. /\*and the rescaled

/\*loadings differ when covariances were analyzed

!KO\_ortrot(rload%1%VARIMAX%{.0001,50}%0%rotload%q%history). /\*Perform varimax

/\*(promax normally is based on it); it is better to rotate

/\*rescaled loadings because we base interpretations on rescaled ones

print rotload /title 'Varimax-rotated (rescaled) loadings - input to promax'

/format f8.3.

!KO\_promax(rotload%1%3%pattern%corr%q\_). /\*Perform promax (with power 3) based on

/\*varimax-rotated loadings

print pattern /title 'Pattern matrix (promax)' /format f8.3.

print (pattern\*corr) /title 'Structure matrix (promax)' /format f8.3.

print corr /title 'Factor (component) correlations' /format f8.3.

print (rotload\*q\_) /format f8.3. /\*(This is what the pattern matrix is)

print (rload\*q\*q\_) /format f8.3. /\*(and this) Tip: keep the REORD argument

/\*of KO\_ortrot on 0, for this expression to work properly

end matrix.

\*Equivalent job by FACTOR command:.

\*Note: sign of loadings and order of factors may differ from the results above,

\*what is normal.

FACTOR

/VARIABLES v1 to v10 /ANALYSIS v1 to v10

/PRINT INITIAL EXTRACTION ROTATION

/CRITERIA FACTORS(3) ITERATE(25) /EXTRACTION PC

/CRITERIA ITERATE(50) /ROTATION PROMAX(3)

/METHOD= COVARIANCE.

\*Note: When METHOD=COVARIANCE, SPSS rotates the rescaled loadings (and obtains

\*the raw ones by back-transforming).

### FACTOR SCORE COEFFICIENTS [!KO\_fsc]

\*/\*!KO\_fsc(cov%load%fcorr%method%name1%name2)\*/\*.

\*Version 1.

\*Computes factor score coefficient matrix by which one can obtain factor scores (estimated factor

\*values as as variables) of the factors extracted in a factor analysis. There is several methods

\*of the matrix computation (several kinds of factor scores).

\*Input necessary for all methods METHOD:

\*COV - the analyzed in the factor analysis correlation or covariance matrix, size p x p, symmetric.

\*LOAD - p x m (1<=m<p) matrix of factor loadings. If the factors underwent oblique rotation then

\*this must be factor pattern matrix. If COV is covariances, not correlations, LOAD is the "raw"

\*(not rescaled) loadings.

\*Input necessary not for all methods METHOD:

\*FCORR - m x m matrix of correlations between the factors. If factors are orthogonal or m=1 you may

\*specify an arbitrary scalar in place of the matrix.

\*Method of computation of factor scores METHOD (keyword in capital letters, optionally quoted

\*or apostrophed):

\*"REG" - regression method of Thurstone (Thompson); maximizes validity of the scores.

\*"REG2" - same method, only as off-diagonal elements of COV there will be used values reproduced

\*by the factors.

\*"REG3" - Horst's or "idealized variables" method; factor scores are computed exactly same formula

\*by which component scores are computed in principal component analysis; equivalent to method "REG"

\*using, in place of COV, the reduced reproduced matrix. Does not need FCORR (specify any scalar).

\*"BART" - Bartlett's method; minimizes bias of scores. Does not need FCORR (specify any scalar).

\*Three next methods are called correlation-preserving, because they achieve correlations between

\*factor scores that are equal to correlations between factors:

\*"AR" - Anderson-Rubin method. Does not need FCORR (specify any scalar). This method always gives

\*uncorrelated scores, even for oblique factors.

\*"MAR" - McDonald-Anderson-Rubin method is the generalization of the previous one, and is suitable

\*also for oblique factors.

\*"GREEN" - Green's method. It can be seen as an approximation to "MAR" to which it becomes identical

\*if communalities of all variables are equal.

\*For methods REG, MAR, GREEN nonsingular COV is needed.

\*Results:

\*NAME1 - coefficients to compute factor scores; the latter themselves are DATA\*NAME1, where DATA is

\*the variables analyzed by the factor analysis (centered if factor analysis was based on covariances

\*or standardized if it was based on correlations).

\*NAME2 - rescaled (standardized) coefficients, i.e. such from which the effect of inequality of

\*variable variances was removed. If COV is correlations NAME2=NAME1.

\*If instead of factor analysis proper there principal component analysis was used, then the component

\*scores, same, are returned by methods REG, REG3, GREEN.

EXAMPLE. Factor analysis (without rotation) and factor scores.

set mxloops 10000.

matrix.

get vars /variables= v1 to v30.

!KO\_corr(vars%r). /\*Correlations

!KO\_paf(r%0%3%{.001,25}%fvrnc%load%rload%comm%conv). /\*Extract 3 factors

print conv /title 'Convergence report'.

print fvrnc /title 'Extraction SS loadings (factor variances)'

/clab= 'Eival' 'Prop' 'RescEiv' 'Prop'.

print load /title 'Loadings'.

!KO\_fsc(r%load%0%REG%b%stb). /\*Regression method to compute factor scores

print b /title 'Factor score coefficients'.

!KO\_zscore(vars%vars). /\*(Correlation-based analysis implies standardized data)

save (vars\*b) /out= \*. /\*Factor scores

end matrix.

### K-MEANS CLUSTERING [!KO\_kmeans]

\*/\*!KO\_kmeans(data%ini%iter%empty%name1%name2%name3%name4)\*/\*.

\*Version 1.

\*Cluster analysis by k-means method.

\*Takes scale data DATA (n cases x p variables) and clusters the cases in specified number of

\*clusters k (k<n). Initial centres must be offered by the user.

\*INI - matrix of initial centres (means) sized k clusters x p variables (columns in DATA).

\*ITER - do this number of iterations (nonnegative integer scalar, e.g. 10). ITER=0 corresponds to

\*classification without iterations, i.e. the assignment of cases to initial centres without improving of

\*the latter.

\*EMPTY - digit (not name or expression, you may optionally quote or apostrophe the digit) 0 or 1.

\*It is specification of the way the problem of empty clusters is to be solved if such clusters emerge:

\*"1" - get rid of empty cluster by reducing k; k will not be able to return to the initial value.

\*"0" - leave empty cluster its previous centre, in hope that later it will pick cases (which may

\*sometimes not happen).

\*Empty clusters take place under bad initial centres and also with strongly discrete data.

\*Results:

\*NAME1 - the obtained cluster membership, in the form n x k binary matrix, i.e. k dummy variables

\*corresponding to the clusters. Each case is assigned to one cluster: 1 stands in that column.

\*You can create single categorical variable of cluster membership as: NAME1\*t(1:ncol(NAME1)).

\*NAME2 - k x p matrix of final, recalculated means: these are the centroids of NAME1 clusters.

\*NAME3 - k x p matrix of pre-final means: these are the latest centres to which cases were

\*classified to.

\*NAME4 - n x 1 column of squared euclidean distances from cases to centres NAME3, to which cases

\*were last time assigned. For example, if a case according to NAME1 appeared to be

\*put in cluster 3, then its distance NAME4 to the centre of that cluster is the distance to the centre

\*which is the 3rd row in NAME3, for just based on that distance the case was assigned to it.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1000.

matrix.

get data /variables= v1 to v4.

compute ini= {1,-2,3,3;

-2,2,8,-3;

4,2,0,-5}.

!KO\_kmeans(data%ini%10%1%dummy%fin%pre%dpre).

print pre /title 'Classification centres'.

print fin /title 'Recalculated centroids'.

save {dummy\*t(1:ncol(dummy)),sqrt(dpre)} /out= \* /variables= clu dpre.

end matrix.

### BLOCK-DIAGONALIZATION BY VAT/IVAT METHOD [!KO\_vat]

\*/\*!KO\_vat(mat%method%priority%name1%name2)\*/\*.

\*Version 1.

\*Reorders rows/columns of square distance matrix MAT by VAT (and IVAT) method.

\*It is one of the ways (Bezdek, Hathaway, "VAT: a tool for...", 2002; Havens, Bezdek, "An efficient

\*formulation of...", 2012) of matrix block-diagonalization which aim is to concentrate small

\*distance values near the matrix diagonal. If to render a block-diagonalized matrix on the "heatmap"

\*chart, one can get an impression about the presence of cluster structure in data and to estimate

\*tentatively the number of clusters.

\*The matrix after block-diagonalization (rows/columns permutation) - result NAME1.

\*Input:

\*MAT - square symmetric nonnegative matrix of distances (or costs) with zero diagonal.

\*METHOD - capitalized keyword (optionally in quotes or apostrophes):

\*"VAT" - do reordering by VAT algorithm. It is closely tied with Prim's algorithm of growing

\*minimal spanning tree. The obtained order will be written down to vector NAME2.

\*"IVAT" - having done the VAT reordering, then do iVAT operation. It specifically replaces

\*some distances in the matrix by other distances of it, reducing diversity of distances in the

\*matrix. The effect of iVAT-replacement is that on the "heatmap" (1) the contrast between

\*between-cluster and within-cluster distances will intensify, aiding visual detection of clusters;

\*(2) detectability of clusters of chain structure (including strongly extended, tree-like, ring-like)

\*will enhance. In its effect, iVAT operation is identical to processing the obtained in VAT matrix

\*by the Floyd-Warshall algorithm in its "determine easiest passes" version.

\*Thus, method=IVAT returns the matrix with reordered+affected (discretized) values, in comparison

\*to those of MAT.

\*PRIORITY - digit (not name or expression) 0 or 1 (optionally in quotes or apostrophes).

\*It is a technical parameter potentially affecting only if there are equal values (ties) in MAT.

\*It defines the order of search/select of an element. 0 is search "from smaller to larger": the

\*element with a smaller index will be preferred. 1 is search "from larger to smaller": the element

\*with a larger index will be preferred (this variant is slightly faster). PRIORITY doesn't change

\*validity of the result, but under ties condition the result may differ, as a consequense of that

\*the order (NAME2) may become another.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

compute d=

{0, 0.73, 0.19, 0.71, 0.16;

0.73, 0, 0.59, 0.12, 0.78;

0.19, 0.59, 0, 0.55, 0.19;

0.71, 0.12, 0.55, 0, 0.74;

0.16, 0.78, 0.19, 0.74, 0}. /\*Distance matrix

print d /format= f5.2.

\*Perform VAT.

!KO\_vat(d%VAT%1%vat%ord). /\*Apply VAT algoritm

print vat /format= f5.2. /\*VAT-reordred rows/columns

print ord. /\*The reordering followed this sequence

compute ord2= ord(ord). /\*(So you can restore

print vat(ord2,ord2) /format= f5.2. /\*the original order, if needed)

\*Perform iVAT.

!KO\_vat(d%IVAT%1%ivat%ord). /\*It is VAT reordering, then iVAT operation

print ivat /format= f5.2.

save vat /outfile= \*. /\*Save VAT- or

\*save ivat /outfile= \*. /\*iVAT-processed matrix to submit to a plotting

/\*command which can draw heatmap of a matrix

end matrix.

### K-NEAREST NEIGHBOURS (WITH DEPENDENT VARIABLE) [!KO\_knnpred]

\*/\*!KO\_knnpred(dv%dis%k%stat%name1%name2%name3%name4)\*/\*.

\*Version 1.

\*K-nearest neighbours analysis in its basic form (without tuning of K and selection of predictors).

\*The function takes, as input, not predictor variables (features) but already computed matrix

\*of distances between points (cases); therefore the distance function may be any the user chooses.

\*The identification of nearest neighbours is done like the function /\*!KO\_knnr\*/ (see) does it -

\*/\*!KO\_knnpred\*/ is based on it.

\*Input:

\*DIS - rectangular or square R x C matrix with nonnegative elements. It is the distances

\*between R row points (training set) and C column points (holdout cases aka test set).

\*If you have only training set (i.e., it is also the test one), you will enter a square

\*symmetric matrix, which diagonal must be "covered" by some knowingly larger value than

\*the rest elements of the matrix (in order the distances of points to themselves "drop

\*from the game").

\*DV - dependent (target) variable, column lengthed R. It may be categorical or quantitative.

\*The ultimate goal of the K-nearest neighbours analysis is to predict values of that variable

\*for the column points (holdout cases aka test set) based on that which values of the variable the

\*nearest to them points-neighbours from the train set have (see argument STAT).

\*K - number of nearest neighbours, positive integer scalar (usually K<<R).

\*STAT - statistic that is the prediction function of DV values of the row points: capitalized

\*keyword (may take it in quotes or apostrophes):

\*"MEAN" - mean of DV of the nearest neighbours (DV is a quantitative feature);

\*"MEDIAN" - median of DV of the nearest neighbours (DV is a quantitative feature),

\*the median is computed by the Averaging Empirical method;

\*"MODE" - mode of DV of the nearest neighbours (DV is a categorical or discrete feature).

\*There are also the weighted versions of those three statistics: "WMEAN", "WMEDIAN", "WMODE".

\*Then the statistic is computed weighted, where weights are the distances to the nearest neighbours

\*inverted into similarities by the formula 1/(d+1). And in doing so, a larger weight in the prediction

\*is given to those of the K nearest neighbours which are closer to the point of prediction.

\*Attention: in case of MODE and WMODE, variable DV must be just of positive values.

\*STAT = MODE or WMODE is the "classification by the KNN analysis".

\*If there are several tied modes (predicted categories), the function selects the one with the

\*greatest frequency in DV variable.

\*Results:

\*NAME1 - K x C matrix containing the numbers of the K nearest neighbours for each column point.

\*NAME2 - K x C matrix containing the distances themselves to the nearest neighbours. In case

\*when STAT= "WMEAN", "WMEDIAN", or "WMODE", it contains not the distances but the similarities

\*obtained from them - the weights, and those are normalized to sum 1 in each column.

\*NAME3 - K x C matrix containing values of DV of the K nearest neighbours.

\*NAME4 - row vector of length C containing the predicted DV for each column point, that is, the

\*element i is f(NAME3(:,i)), where f is the function STAT.

\*Note. The formula of obtaining similarities from distances, 1/(d+1), will yield more steep decrease

\*of weight from the closest of the K neighbours towards the farthest of them if the difference between

\*the corresponding two d is big, than if it is small. Therefore (at the weighted analysis) the result

\*may depend on the global magnitude (on the range) of distances in the data. You might want to multiply

\*DIS by a constant, thus to boost or to shrink the distances, whereby to affect the result.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. All the sample is the training set.

set mxloops 1E6.

matrix.

get y /variable= y. /\*Scale dependent variable (target)

get x /variables= x1 x2 x3. /\*Independent scale variables (features)

!KO\_seuclid(x%d). /\*Compute matrix of (squared) euclidean distances

/\*between all data points

release x. /\*(we don't need X anymore)

call setdiag(d,1E300). /\*Because a point cannot be a neighbour to itself,

/\*cover the diagonal with some supervalue

compute k= 3. /\*We're interested in K=3 nearest neighbours as predictor points

!KO\_knnpred(y%d%k%MEAN%nei%dnei%ynei%ypred). /\*Run the KNN prediction;

/\*let the prediction function be the mean

\*Prediction result:.

print {y,t(ypred)} /title 'Actual and predicted (MEAN) Y for each case'

/clabels 'Obs\_Y' 'Pred\_Y' /format= f8.4.

\*Concomitant output:.

print t(ynei) /title 'K=3 Nearest neigbours (their Y values) to each case'

/format= f8.4.

print t(nei) /title 'K=3 Nearest neigbours (their indices) to each case'.

print t(dnei) /title 'K=3 Nearest neigbours (their distances) to each case'.

end matrix.

\*Equivalent run of SPSS prodedure KNN. All the cases are the training set.

KNN y (MLEVEL=S) WITH x1 x2 x3

/RESCALE COVARIATE=NONE

/MODEL NEIGHBORS=FIXED(K=3) METRIC=EUCLID FEATURES=ALL

/CRITERIA PREDICTED=MEAN WEIGHTFEATURES=NO

/PARTITION TRAINING=100 HOLDOUT=0

/SAVE PREDVAL.

EXAMPLE. There are training cases (TRAIN=1) and holdout (TRAIN=0) cases.

set mxloops 1E6.

matrix.

get y /variable= y. /\*Scale dependent variable (target)

get x /variables= x1 x2 x3. /\*Independent scale variables (features)

get set /variable= train. /\*Set identifier;

/\*Holdout cases will be predicted by training cases

!KO\_indices2(set%set1%set0). /\*Get indices of train cases and of holdout cases

!KO\_pwminkr(x(set1,:)%x(set0,:)%1%{1,1,1}%d). /\*Compute rectangular matrix

/\*of distances

/\*between training cases (row points) and holdout cases (column points);

/\*Let the distances be Manhattan ones

compute k= 3. /\*We're interested in K=3 nearest neighbours as predictor points

!KO\_knnpred(y%d%k%"MEDIAN"%nei%dnei%ynei%ypred). /\*Run the KNN prediction;

/\*let the prediction function be the median

\*Prediction result for holdout cases:.

print {y(set0),t(ypred)}

/title 'Actual and predicted (MEDIAN) Y for each holdout case'

/clabels 'Obs\_Y' 'Pred\_Y' /format= f8.4.

print t(ynei)

/title 'K=3 Nearest training neigbours (their Y values) to each holdout case'

/format= f8.4.

print t(nei)

/title 'K=3 Nearest training neigbours (their indices) to each holdout case'.

print t(dnei)

/title 'K=3 Nearest training neigbours (their distances) to each holdout case'.

\*If needed, do analysis within training set cases as well.

!KO\_pwmink(x(set1,:)%1%{1,1,1}%d). /\*Compute square matrix of Manhattan distances

/\*between training cases only

call setdiag(d,1E300). /\*Cover diagonal entries by some supervalue

!KO\_knnpred(y%d%k%"MEDIAN"%nei%dnei%ynei%ypred). /\*Run the KNN prediction as before

\*Prediction result for training cases:.

print /title '----------------'.

print {y(set1),t(ypred)}

/title 'Actual and predicted (MEDIAN) Y for each training case'

/clabels 'Obs\_Y' 'Pred\_Y' /format= f8.4.

print t(ynei)

/title 'K=3 Nearest training neigbours (their Y values) to each training case'

/format= f8.4.

print t(nei)

/title 'K= 3 Nearest training neigbours (their indices) to each training case'.

print t(dnei) /title 'K=3 Nearest training neigbours (their distances) to each case'.

end matrix.

\*Equivalent run of SPSS prodedure KNN. Variable TRAIN splits cases in training and holdout.

KNN y (MLEVEL=S) WITH x1 x2 x3

/RESCALE COVARIATE=NONE

/MODEL NEIGHBORS=FIXED(K=3) METRIC=CITYBLOCK FEATURES=ALL

/CRITERIA PREDICTED=MEDIAN WEIGHTFEATURES=NO

/PARTITION VARIABLE=train

/SAVE PREDVAL.

EXAMPLE. Like Example first, but the dependent variable is categorical (there will be classification).

set mxloops 1E6.

matrix.

get y /variable= ypos. /\*Categorical dependent variable (target) w/ positive values

get x /variables= x1 x2 x3. /\*Independent scale variables (features)

!KO\_seuclid(x%d). /\*Compute matrix of (squared) euclidean distances

/\*between all data points

release x. /\*(we don't need X anymore)

call setdiag(d,1E300). /\*Because a point cannot be a neighbour to itself,

/\*cover the diagonal with some supervalue

compute k= 3. /\*We're interested in K=3 nearest neighbours as predictor points

!KO\_knnpred(y%d%k%MODE%nei%dnei%ynei%ypred). /\*Run the KNN prediction;

/\*the prediction function be the MODE (knn analysis performs classification)

\*Prediction result:.

print {y,t(ypred)} /title 'Actual and predicted (MEAN) Y for each case'

/clabels 'Obs\_Y' 'Pred\_Y' /format= f8.4.

\*Concomitant output:.

print t(ynei)

/title 'K=3 Nearest neigbours (their Y values) to each case' /format= f8.4.

print t(nei) /title 'K=3 Nearest neigbours (their indices) to each case'.

print t(dnei) /title 'K=3 Nearest neigbours (their distances) to each case'.

end matrix.

\*Equivalent run of SPSS prodedure KNN. All the cases are the training set.

KNN ypos (MLEVEL=N) WITH x1 x2 x3

/RESCALE COVARIATE=NONE

/MODEL NEIGHBORS=FIXED(K=3) METRIC=EUCLID FEATURES=ALL

/CRITERIA WEIGHTFEATURES=NO

/PARTITION TRAINING=100 HOLDOUT=0

/SAVE PREDVAL.

### NAIVE BAYES CLASSIFIER [!KO\_nbclass]

\*/\*!KO\_nbclass(dv%ivs%name1%name3%name3)\*/\*.

\*Version 1.

\*Naive Bayes classifier for categorical data in its basic form - without predictor selection.

\*DV - dependent categorical (or discrete) variable.

\*IVS - independent, predictor categorical (or discrete) variables.

\*DV has n1 cases (rows), IVS has n1+n2 (n2>=0) cases (rows). The first n1 cases -

\*are the training sample. Going below that n2 cases in IVS, if any - are the holdout

\*or test sample, i.e. cases which DV values we either don't know or don't submit into the

\*model building.

\*Results:

\*NAME1 - row vector of k values of the dependent variable, i.e., class codes.

\*NAME2 - n1+n2 x k matrix of predicted (posterior) probabilities of case belonging to

\*the classes; sum of probabilities =1 in each matrix row.

\*NAME3 - n1+n2 column of the predicted class (value of DV); the predicted class is the class

\*with the maximal probability in the row of NAME2. If that maximum is the same for several elements

\*in the row (tied), the selected is the class with the greater NAME1 value.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. 100 cases in the dataset. The last 10 cases are missing for Y variable.

set mxloops 1E6.

matrix.

get y /variable= y /missing= omit. /\*Dependent (target) categorical variable

/\*with, say, 3 categories (classes)

/\*It has 10 missing values in the end - those cases are holdout ones,

/\*so we discard them

get x /variables= x1 x2 x3 x4. /\*Categorical predictors, all 100 cases

!KO\_nbclass(y%x%yvals%predp%predc). /\*Run the analysis

print yvals /title "Classes' codes".

print {predc,predp} /title 'Predicted Class and classes Predicted Probabilities'

/clabels= 'PredCl' 'Prob1' 'Prob2' 'Prob3'. /\*Predictions for all cases,

/\*the 10 holdout cases are the last ones

\*Let's observe classification results for the training sample.

!KO\_crosstab(y%predc(1:90)%1%tab%name2%name3).

!KO\_classres(tab%overall%correct%ocorrect).

print {tab,correct;overall,ocorrect} /format= f8.1

/rlabels= 'Class1' 'Class2' 'Class3' 'Overall%'

/clabels= 'Class1' 'Class2' 'Class3' '%Correct'

/title 'Classification table: Observed x Predicted classes'.

end matrix.

\*Equivalent run of NAIVEBAYES command.

NAIVEBAYES y BY x1 x2 x3 x4 /SUBSET NOSELECTION /SAVE PREDVAL PREDPROB.

# RANDOM DATA FUNCTIONS

### RANDOM VALUES FROM STANDARD UNIFORM DISTRIBUTION

Use in-built function uniform().

### RANDOM VALUES FROM STANDARD NORMAL DISTRIBUTION [!KO\_normal]

\*/\*!KO\_normal(nr%nc%name)\*/\*.

\*Version 1.

\*Generates matrix NAME sized NR x NC, which elements are random

\*values from standard normal distribution.

\*NR, NC – positive integer scalars.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*To generate data from population with correlating normal variables use

\*more general function /\*!KO\_mvnorm\*/.

### RANDOM VALUES FROM NORMAL DISTRIBUTION WITH SPECIFIED PARAMETERS [!KO\_rvnorm]

\*/\*!KO\_rvnorm(mean%sd%name)\*/\*.

\*Version 1.

\*Generates random number from normal distribution with mean MEAN and standard deviation SD.

\*MEAN and SD - matrices of any, same size. Result NAME - of the same size.

\*Element ij NAME contains random value from normal distribution with parameters ij MEAN and ij SD.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

### RANDOM VALUES FROM UNIFORM DISTRIBUTION WITH SPECIFIED PARAMETERS [!KO\_rvunif]

\*/\*!KO\_rvunif(min%max%name)\*/\*.

\*Version 1.

\*Generates random number from uniform distribution with minimum MIN and maximum MAX.

\*MIN and MAX - matrices of any, same size. Result NAME - of the same size.

\*Element ij NAME contains random value from uniform distribution with parameters ij MIN and ij MAX.

\*(ij MIN must be not greater than ij MAX).

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

### RANDOM VALUES FROM CHI-SQUARE DISTRIBUTION [!KO\_chisq]

\*/\*!KO\_chisq(nr%nc%df%name)\*/\*.

\*Version 1.

\*Generates matrix NAME sized NR x NC, which elements are random

\*values from chi-square distribution with degrees of freedom DF.

\*NR, NC, DF – positive integer scalars.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

### RANDOM VALUES FROM BINOMIAL DISTRIBUTION [!KO\_binom]

\*/\*!KO\_binom(nr%ntrials%prob%name)\*/\*.

\*Version 1.

\*Generates column vector NAME of length NR, which elements are random values from

\*binomial distribution with parameters number of trials NTRIALS and success (event) probability PROB.

\*NTRIALS (positive integer) and PROB (between 0 and 1) - either two scalars or two vectors length NR.

\*If vectors, it means different values of NAME can be generated under different values of the parameters.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### RANDOM VALUES FROM CATEGORICAL DISTRIBUTION [!KO\_categ]

\*/\*!KO\_categ(nr%nc%kprob%name)\*/\*.

\*Version 2.

\*Generates matrix NAME sized NR x NC, which elements are random

\*values (codes 1 to k) from k-categorical distribution. (Categorical distribution is

\*multinomial distribution with number of trials 1. Bernoulli distribution is 2-categorical

\*distribution.)

\*NR, NC – positive integer scalars.

\*KPROB - either vector of length k, containing probabilities for the k categories, or positive integer

\*scalar equal to k.

\*If KPROB - probability vector, it must be nonnegative numbers, at least one positive. The numbers

\*need not sum to 1: the function will bring their sum to 1; it is suffice that the numbers are in

\*the same ratios as the probabilities.

\*If KPROB - scalar, it must be positive integer (fractional value will be truncated).

\*It is k, and the k propabilities are supposed to be equal. When the task is the equiprobable categorical

\*distribution this way of specifying is more convenient than by probability vector, because it will be

\*faster result.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

!KO\_categ(5%3%{.6,.3,.1}%name).

print name.

end matrix.

EXAMPLE. Random sampling of cases with replacement.

set mxloops 100000.

matrix.

get data /variables= id v1. /\*Case id and some data

loop i= 1 to 5. /\*5 random samples to draw

-!KO\_categ(10%1%nrow(data)%selcases).

/\*Generate, say, 10 indices (case numbers) to select

-print data(selcases,:). /\*The random sample of size 10

end loop.

end matrix.

For sampling without replacement use !KO\_catwor or !KO\_sample.

### SAMPLE FROM CATEGORICAL DISTRIBUTION WITHOUT REPLACEMENT [!KO\_catwor]

\*/\*!KO\_catwor(n%kprob%stock%method%name1%name2)\*/\*.

\*Version 1.

\*Collects random sample of N objects from a limited set of objects belonging to different

\*categories. This function returns random values from k-categorical distribution

\*(k is the number of object categories), like function /\*!KO\_categ\*/, but it does sampling without

\*replacement from finite populatioin, whereas /\*!KO\_categ\*/ does sampling with replacement.

\*Because objects are drawn one by one and are not returned back to "stock" (i.e. to limited

\*population) the probabilities of selection ("realization") for different categories change at

\*the process flow. These probabilities are determined, from one side, by the specified probabilities

\*of "demand", and, from other side, by the current probabilities of "availability in stock", so to say.

\*Input:

\*N - positive integer scalar: the required sample size.

\*KPROB - vector of length k, containing probabilities for the k categories of objects. It must be

\*nonnegative numbers, at least one positive. The numbers need not sum to 1: the function will bring

\*their sum to 1; it is suffice that the numbers are in the same ratios as the probabilities.

\*KPROB should be understood as the initial probabilities of "demand"; i.e. they are probabilities

\*of selection if the population had been infinite.

\*STOCK - vector of length k and same orientation as KPROB, containing frequencies for the k categories

\*of objects - it is the observed distribution of our finite population (the "stock") from where

\*objects will be drawn into the sample. The frequencies must be nonnegative integers, at least one

\*positive. If you input fractional values the function will first truncate them to integers.

\*While weights KPROB set "demand" probabilities for categories, frequencies STOCK set "availability"

\*probabilities for them.

\*METHOD - digit (not variable name or expression; you may optionally quote or apostrophe the digit).

\*It is specification of the method of determination of current probabilities of selection (realization):

\*"1" - method "Initial demand, if there left". Probability PROB(i) to select now object of category i

\*is proportional to KPROB(i)\*LEFT(i), where LEFT(i)=1, if there are objects of category i left yet in

\*the population, otherwise LEFT(i)=0. If frequencies STOCK are all big enough to be not emptied prior

\*the end of the sampling procedure then this method is equivalent to application of function /\*!KO\_categ\*/.

\*"2" - method "Initial demand \* how many left". Probability PROB(i) to select now object of category i

\*is proportional to KPROB(i)\*FLEFT(i), where FLEFT(i) is the fraction of category i among the objects yet

\*left in the population. It is classical random selection without replacement.

\*"3" - method "Floating demand \* how many left". Probability PROB(i) to select now object of category i

\*is proportional to PROB\_(i)\*FLEFT(i), where PROB\_(i) is PROB(i) computed before selection of the

\*preceding object into the sample (before selection of the 1st object PROB\_(i)=KPROB(i)). It is

\*"sharpening" method leading to soon, during selections, shift of preference toward one category,

\*typically the one with high representedness on the "stock", the limited population.

\*"4" - method "Floating non-demand \* how many left". Probability PROB(i) to select now object of

\*category i is proportional to (1-PROB\_(i))\*FLEFT(i) (before selection of the 1st object

\*1-PROB\_(i)=KPROB(i)). This method can be interesting as a "smoothing" alternative to method "2":

\*selection from low-represented categories will be somewhat enhanced and selection from high-represented

\*categories will be somewhat reduced, in comparison with that method.

\*Results:

\*NAME1 - column vector of length N containing category codes (numbers from sequence 1,2,...,k)

\*of the created random sample (and with sequence there also random). In case when there happened not

\*enough objects in the population to draw N objects without relpacement, length of NAME1 will be

\*shorter than N. If there failed to draw even a single object, NAME1=0.

\*NAME2 - column vector of length k containing the frequencies left in STOCK.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If KPROB are equal probabilities and STOCK is vector of ones or is binary (1 and 0), then equivalent

\*will be to use function /\*!KO\_sample\*/ which is faster.

EXAMPLE. Select randomly 50 cases from the dataset weighted with case frequencies. Besides, it is requested that the greater is the data value magnitude the more likely that case be selected into the sample.

set mxloops 10000000.

matrix.

get data /variables= id v1. /\*Case id and data V1

get freq /variables= freq. /\*also, frequency variable with positive integers

/\*defines how many duplicates of each case exist:

/\*this is the “stock” vector

compute kprob= abs(data(:,2)). /\*While the “demand” vector will reflect the

/\*magninude of V1 data values

!KO\_catwor(50%kprob%freq%"2"%sample%freq2).

save data(sample,:) /outfile= \* /variables= id v1.

print {data(:,1),freq2}. /\*How many case duplicates did not enter the sample.

end matrix.

\*In the saved sample if a case was selected more than once (due to duplicates

\*freq) it occupies own row. If you need to aggregate the duplicates, run:.

AGGREGATE /OUTFILE= \* /BREAK= id /v1= FIRST(v1) /freq= N.

### SELECT N RANDOM ELEMENTS FROM VECTOR [!KO\_sample]

\*/\*!KO\_sample(n%cases%name1%name2)\*/\*.

\*Version 1.

\*Randomly selects N different elements from sequence of numbers 1,2,...,k.

\*This function is convenient for forming a random subsample ("without replacement") from a dataset,

\*and also for random sorting of cases in a dataset.

\*Input:

\*N - positive integer scalar: how many different elements to select.

\*CASES - "population" of elements: either positive integer scalar or binary (0 vs 1) vector.

\*If scalar, then it is k, and selection will be from numbers 1,2,...,k. If binary vector, then its

\*length is k and its element numbers (indices) correspond to numbers 1,2,...,k; element's value 1

\*in the vector permits to select the element's number while value 0 prohibits to select. Thus, vector

\*CASES is filter. Vector consisting of just 1s is equivalent to specifying CASES as scalar.

\*Vector must contain at least one 1.

\*Results:

\*NAME1 - column vector of length N containing the randomly selected numbers, and in random order.

\*NAME2 - binary column vector of length k, where 1 flags not selected (remaining) element CASES,

\*and 0 flags selected (removed) element.

\*If N>=k length of NAME1 will be k, and NAME2 will consist only of zeros.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*The function is relatively slow. If k is great but N is small, then sampling with replacement that

\*you can accomplish with the help of function /\*!KO\_categ\*/ will be quite good approximation to the

\*sampling without replacement; but /\*!KO\_categ\*/ is faster than /\*!KO\_sample\*/.

\*If your population CASES is weighted, i.e. consists of counts (rather than 0 or 1), or if its elements

\*are not equiprobable for selection, use function /\*!KO\_catwor\*/.

EXAMPLE. Select randomly 30 cases from dataset. Then select 10 more (of not yet selected).

set mxloops 10000000.

matrix.

get data /variables= id v1. /\*Case id and data

!KO\_sample(30%nrow(data)%selected%left).

print data(selected,:).

!KO\_sample(10%left%selected%left).

print data(selected,:).

end matrix.

EXAMPLE. Equivalent to the previous, with deletion of the selected cases from data.

set mxloops 10000000.

matrix.

get data /variables= id v1. /\*Case id and data

!KO\_sample(30%nrow(data)%selected%left).

!KO\_split(data%selected%sample1%remnant).

print sample1.

!KO\_sample(10%nrow(remnant)%selected%left).

!KO\_split(remnant%selected%sample2%remnant).

print sample2.

print remnant.

end matrix.

EXAMPLE. Sort cases in random order.

set mxloops 10000000.

matrix.

get data /variables= id v1. /\*Case id and data

!KO\_sample(nrow(data)%nrow(data)%order%left).

print data(order,:).

end matrix.

### RANDOM MATRIX FROM WISHART DISTRIBUTION [!KO\_wishart]

\*/\*!KO\_wishart(chl%n%name)\*/\*.

\*Version 1.

\*Returns random matrix NAME from Wishart distribution with scale parameter C and degrees of freedom N.

\*C is a population covariance matrix (multivariate normal distribution is assumed in the

\*population). Argument CHL of the function is not C, but its Cholesky root: upper triangular

\*matrix returned by the in-built matrix function chol(C).

\*N (positive integer scalar) is the sample size for generating of the random matrix NAME.

\*The latter is a scatter matrix; to turn it into covariance matrix divide it by N.

\*If as a CHL you specify a scalar (positive) then it is counted a population standard deviation,

\*and NAME/N will be a random value of variance.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

compute cov=

{ 2.8, 1.9, 0.7,-0.4;

1.9, 4.1,-1.0, 0.5;

0.7,-1.0, 2.4, 1.3;

-0.4, 0.5, 1.3, 2.6 }.

print cov.

compute ch= chol(cov).

!KO\_wishart(ch%100%scat).

print (scat/100).

end matrix.

### RANDOM DATA FROM NORMAL POPULATION WITH SPECIFIED COVARIANCES [!KO\_mvnorm]

\*/\*!KO\_mvnorm(n%chl%mean%name)\*/\*.

\*Version 1.

\*Generates data NAME - random sample of size N from normally distributed population having

\*covariances C and centroid MEAN.

\*Argument CHL of the function is not covariance matrix C (size p x p) itself between variables,

\*but its Cholesky root: upper triangular matrix (of same size) returned by the in-built matrix

\*function chol(C).

\*N - number of cases to generate, positive integer scalar.

\*MEAN - scalar or row vector of length p: this is the means of the variables in the population.

\*Result NAME - matrix of random data n cases x p variables from normal population of the specified

\*covariances between the variables and the means of the variables.

\*In a particular case, C can be of size 1 x 1; then C is the variance (and CHL is st. deviation).

\*If matrix C is correlations, not covariances, keep argument MEAN as 0, the obtained variables NAME

\*you can then multiply by the population standard deviations and then add the population means;

\*however, more convenient way is: convert correlation matrix into covariance one by function

\*/\*!KO\_corrcov\*/, after which apply /\*!KO\_mvnorm\*/ with specifying of the means as MEAN in it.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

EXAMPLE. Generate 500-case random data from normal 4-variable correlated population.

matrix.

compute mx= {12.2, -2.3, 5.6, -3.2;

-2.3, 17.7, 1.1, -3.7;

5.6, 1.1, 23.1, 5.9;

-3.2, -3.7, 5.9, 5.8}. /\*Population covariances, positive

/\*definite matrix

print mx /title 'Population covariance matrix'.

!KO\_mvnorm(500%chol(mx)%0%vars).

!KO\_cov(vars%cov).

print cov /title 'Sample covarinces in the generated data'.

!KO\_mean(vars%mean).

print mean /title 'Means in the generated data'.

save vars /out= \*.

end matrix.

EXAMPLE. The input population matrix is MSCP. Generate 10000-case random normal data for it.

matrix.

compute mx= {21.8, 9.5, 2.2, 4.2;

9.5, 32.1, -3.1, 5.4;

2.2, -3.1, 24.3, 3.3;

4.2, 5.4, 3.3, 11.6}. /\*Population MSCP matrix, 4 variables

compute mean= {3.1,3.8,-1.1,2.4}. /\*Population means

print mx.

print mean.

!KO\_swcentr(mx%mean%{0,0,0,0}%mx). /\*Since MVNORM function needs covariance matrix,

/\*convert the MSCP matrix into the covariance

/\*(hoping it will be positive definite)

compute n= 10000.

!KO\_mvnorm(n%chol(mx)%mean%vars). /\*Generate the sample, and take to the needed

/\*population mean

print (sscp(vars)/n). /\*So that the input MSCP matrix is satisfied

save vars /out= \*.

end matrix.

### CREATE RANDOM DATA EXACTLY HAVING SPECIFIED COVARIANCES

To obtain a dataset of random data with exactly as needed covariating or correlating variables apply function /\*!KO\_tocov\*/ to randomly generated data. If you want the data to be multivariate normal generate the values from normal distribution (function /\*!KO\_normal\*/). In order to set in the end the wanted means and variances (or maximum, minimum) to the variables, use function /\*!KO\_rescale\*/. See example at function /\*!KO\_tocov\*/.

# RECODE/REPLACE FUNCTIONS

### RECODE VALUES (EXACT MATCH) [!KO\_recode1]

\*/\*!KO\_recode1(data%filter%old%new%name)\*/\*.

\*Version 1.

\*In data matrix DATA of any size replaces values of list OLD by corresponding values of list NEW.

\*Returns matrix NAME which is matrix DATA with the values thus recoded.

\*This function is similar to the out-of-MATRIX command RECODE in which the values to recode are specified

\*by listing.

\*OLD and NEW - row vectors. i-th value in OLD, if found in DATA, is recoded into i-th value of NEW.

\*Both these vectors are of equal length or NEW may be by one element longer; in the latter case all other

\*than OLD values will be recoded into that last value of NEW.

\*If some value encounters in OLD more than once it is recoded into the NEW value corresponding to the first

\*(leftmost) its encounter in the vector OLD.

\*Argument FILTER allows to prohibit recoding of values in some rows of DATA. FILTER is either a scalar or

\*a column lengthed as the number of rows in DATA. If nonzero scalar - no ban exist; if zero scalar -

\*then complete ban (function won't act, NAME will be equal to DATA). If column, then 0 in it bans

\*recoding in that row of DATA, while other than 0 value - allows recoding in the row.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Recoding in MATRIX and equivalently out of MATRIX.

matrix.

get vars /variables= v1 to v5.

!KO\_recode1(vars%1%{2,4,5}%{-2,-4,-5,999}%vars).

print vars.

end matrix.

recode v1 to v5 (2=-2) (4=-4) (5=-5) (else=999).

list v1 to v5.

EXAMPLE. Recoding in MATRIX and equivalently out of MATRIX.

matrix.

get vars /variables= v1 to v5.

get filt /variable= filter.

!KO\_recode1(vars%filt%{2,4,5}%{-2,-4,-5}%vars).

print vars.

end matrix.

do if filter<>0.

-recode v1 to v5 (2=-2) (4=-4) (5=-5).

end if.

list v1 to v5.

### RECODE VALUES (HITTING INTO RANGE) [!KO\_recode2]

\*/\*!KO\_recode2(data%filter%low%high%new%name)\*/\*.

\*Version 1.

\*In data matrix DATA of any size replaces values falling into range from LOW to HIGH (inclusive)

\*by corresponding values of list NEW.

\*Returns matrix NAME which is matrix DATA with the values thus recoded.

\*This function is similar to the out-of-MATRIX command RECODE in which the values to recode are specified

\*by range(s) from... to...

\*LOW, HIGH, and NEW - row vectors. LOW(i) and HIGH(i) is a range; HIGH(i) should normally be >= LOW(i).

\*Any value in DATA lying in the range from LOW(i) to HIGH(i) will be recoded into

\*value NEW(i). LOW and HIGH must be of equal length. NEW may be by one element longer; in the latter

\*case all other than the recoded values will be recoded into that last value of NEW.

\*If some ranges specified by vectors LOW and HIGH superimpose, their common values

\*are recoded into the NEW value corresponding to the earliest (leftmost in the vectors) of these ranges.

\*For example, if LOW is {6,5} and HIGH is {11,10}, i.e. intersecting ranges 6-11 and 5-10 were specified,

\*while NEW is {23,37}, then "disputable" data values - from 6 to 10 - will be recoded into 23,

\*not into 37, because range 6-11 is specified "earlier" (more left in the vectors) than 5-10.

\*(The same way command RECODE behaves.)

\*Argument FILTER allows to prohibit recoding of values in some rows of DATA. FILTER is either a scalar or

\*a column lengthed as the number of rows in DATA. If nonzero scalar - no ban exist; if zero scalar -

\*then complete ban (function won't act, NAME will be equal to DATA). If column, then 0 in it bans

\*recoding in that row of DATA, while other than 0 value - allows recoding in the row.

\*If vectors LOW and HIGH are identical the function will return the same result as /\*!KO\_recode1\*/.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Recoding in MATRIX and equivalently out of MATRIX.

matrix.

get vars /variables= v1 to v5.

get filt /variable= filter.

!KO\_recode2(vars%filt%{2,4,5}%{3,4.999,5}%{2.5,4.5,5}%newvars).

print newvars.

end matrix.

do if filter<>0.

-recode v1 to v5 (2 thru 3 =2.5) (4 thru 4.999 =4.5) (5=5).

end if.

list v1 to v5.

### VARIOUS UNIVARIATE STATISTICS, BY GROUPS: REPLACE ORIGINAL VALUES WITH THEM [!KO\_aggrv]

\*/\*!KO\_aggrv(data%bin%stat%check%value1%value2%name)\*/\*.

\*Version 1.

\*This function computes a requested descriptive statistic by groups - exactly as function /\*!KO\_aggr\*/ does,

\*but it returns, as result NAME, not the groups X variables table but the dataset sized as DATA, in

\*which the initial values of DATA are replaced with the computed summary statistic: if a row (case)

\*belongs to group i and only to it then its values in NAME is the row i of the mentioned table returned

\*by function /\*!KO\_aggr\*/. Therefore, /\*!KO\_aggrv\*/ performs what SPSS command AGGREGATE under option

\*MODE=ADDVARIABLES does.

\*Arguments DATA, BIN, STAT are exactly as in function /\*!KO\_aggr\*/ - see.

\*Argument CHECK (digit 0 or 1, not name or expression):

\*If you know that sums in all the rows of BIN equal 1, i.e. that each case belongs to just one group,

\*so that BIN are dummy variables, set CHECK to 0. While if you admit that sums in

\*some rows of BIN may not equal 1, set CHECK to 1.

\*Arguments VALUE1 and VALUE2 will be ignored if СHECK is 0.

\*VALUE1 (number, not name or expression): specify value which should be inserted in NAME for those cases

\*which do not belong to any group (i.e. whose sum in row of BIN =0).

\*VALUE2: specify value (number, not name or expression), which should be inserted in NAME for those cases

\*which belong to more than one group (i.e. whose sum in row of BIN >1); or specify in capital letters

\*the keyword "AVER" - which means the request, for a case belonging to more than one group

\*and thus having more than one computed value of the statistic at its disposal, to average these its

\*values of the statistic.

\*Values of arguments CHECK, VALUE1, VALUE2 may be optionnally wrapped in quotes or apostrohes.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your grouping is defined by a categorical variable, create dummy variables BIN out of it

\*with the help of function /\*!KO\_freq\*/.

EXAMPLE. Replace individual values by group means.

matrix.

get vars /vari= v1 v2 /names= names.

print data /cnames= names /format= f8.4.

get grvar /variable= group. /\*Categorical grouping variable

!KO\_freq(grvar%1%dummy%freq%codes). /\*Create dummy variables

print codes.

print dummy /title 'Group membership'.

!KO\_aggr(vars%dummy%MEAN%table).

print table /title 'Group means' /cnames= names /format= f8.4.

!KO\_aggrv(vars%dummy%MEAN%0%-999%9999%addvars).

print addvars /title 'Individual values replaced with corresponding group means'

/cnames= names /format= f8.4.

end matrix.

EXAMPLE. Same, but groups could overlap by case membership.

matrix.

get vars /vari= v1 v2 /names= names.

print data /cnames= names /format= f8.4.

compute bin= rnd(uniform(nrow(vars),3)). /\*Binary grouping variables

print bin /title 'Group memberships'.

!KO\_aggr(vars%bin%MEAN%table).

print table /title 'Group means' /cnames= names /format= f8.4.

!KO\_aggrv(vars%bin%MEAN%1%-999%AVER%addvars).

print addvars /title 'Individual values replaced with corresponding group means'

/cnames= names /format= f8.4.

print title 'If a case belongs to >1 groups the group means were averaged' /space= 0.

print title 'If a case belongs to no groups it got value -999' /space= 0.

end matrix.

# SEARCH/HIGHLIGHT FUNCTIONS

### POSITIONS OF NONZERO ELEMENTS IN VECTOR [!KO\_indices]

\*/\*!KO\_indices(vec%name)\*/\*.

\*Version 2.

\*Takes vector (row or column) VEC and returns row vector NAME containing the list of indices

\*(sequential numbers, positions) of nonzero elements in VEC. The indices go ascendingly.

\*If there is no nonzero elements in VEC the function returns scalar 0.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Some analysis separately by groups (without sorting).

set mxloops 10000. /\*Some functions might need many cycles.

matrix.

get vars /variables= v1 v2 /names= names. /\*Get some variables, memorize names

get group /variables= gr. /\*Get the grouping variable (also numeric)

!KO\_freq(group%1%dummies%freq%codes). /\*Get to know group codes and group sizes

release dummies. /\*(This we won't need in this example)

print {codes,freq} /clab 'Codes' 'Freq'.

loop i= 1 to nrow(codes). /\*For each group

-do if freq(i)>=3. /\*with at least 3 individuals

- !KO\_indices(group=codes(i)%casenums). /\*get the indices (case numbers list),

- compute data= vars(casenums,:). /\*extract the group and

- !KO\_kurtosis(data%kurt). /\*perform any analysis for it

/\*(in this example, compute kurtosis for our variables)

- print codes(i) /title 'Group code'.

- print kurt /cnames= names /title 'Kurtosis in the group' /space= 0.

-end if.

end loop.

end matrix.

### POSITIONS OF NONZERO AND ZERO ELEMENTS IN VECTOR [!KO\_indices2]

\*/\*!KO\_indices2(vec%name1%name2)\*/\*.

\*Version 1.

\*Takes vector (row or column) VEC and returns row vector NAME1 containing the list of indices

\*(sequential numbers, positions) of nonzero elements in VEC, and row vector NAME2 containing

\*the list of indices of zero elements in VEC. The indices go ascendingly.

\*If there is no nonzero elements in VEC, NAME1 will be scalar 0. If there is no zero elements

\*in VEC, NAME2 will be scalar 0.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### POSITIONS OF NONZERO ELEMENTS IN ROWS OR COLUMNS OF MATRIX

If you want to write down indices of nonzero elements in multiple vectors, i.e. in matrix, you may use /\*!KO\_indices\*/ on each its row or column. But more advantageous from the speed point of view is to use, for that, /\*!KO\_ram\*/ (extract indices of non zero elements from matrix rows) or /\*!KO\_vram\*/ (extract indices of non zero elements from matrix columns).Or use function /\*!KO\_indicesm\*/.

### POSITIONS OF NONZERO ELEMENTS IN MATRIX (AND THE ELEMENTS THEMSELVES) [!KO\_indicesm]

\*/\*!KO\_indicesm(mat%ANDVAL%name)\*/\*.

\*Version 1.

\*Takes matrix MAT of any size and returns two-column matrix NAME containing the list of indices

\*(sequential numbers, positions) of nonzero elements in MAT. The indices go ascendingly.

\*The row number is written in the 1st column of NAME and the column number is written in the

\*2nd column of NAME.

\*Argument ANDVAL allows also to write out the nonzero values themselves, as the 3rd column of NAME.

\*ANDVAL is the digit (not name or expression) 0 or 1 (you may take the digit in quotes or

\*apostrophes). If 1, the write-out of values will take place.

\*If there is no nonzero elements in MAT, the function returns scalar 0.

\*This function works somewhat faster on wide than on tall matrices.

\*This function is cognate to /\*!KO\_nzlist\*/ but returns result in somewhat different form.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 10000.

matrix.

compute mat= uniform(15,8).

compute mat= mat&\*(mat>.7).

print mat.

!KO\_indicesm(mat%1%name).

print name /clabels= 'row' 'col' 'val'.

end matrix.

### POSITION OF FIRST ENCOUNTER OF VALUE IN ROW/ROWS [!KO\_indx]

\*/\*!KO\_indx(mat%val%name)\*/\*.

\*Version 1.

\*Takes MAT - matrix or row vector. Returns index (position) of first (leftmost)

\*encounter of value VAL in each row of MAT.

\*VAL may be a scalar or a column vector of length as the number of rows in MAT; in the latter

\*instance each row of MAT is given its corresponding value VAL.

\*Result NAME - column vector with number of rows as in MAT.

\*Position 0 means that VAL is not found in the row.

### POSITION OF LAST ENCOUNTER OF VALUE IN ROW/ROWS [!KO\_rindx]

\*/\*!KO\_rindx(mat%val%name)\*/\*.

\*Version 1.

\*Takes MAT - matrix or row vector. Returns index (position) of last (rightmost)

\*encounter of value VAL in each row of MAT.

\*VAL may be a scalar or a column vector of length as the number of rows in MAT; in the latter

\*instance each row of MAT is given its corresponding value VAL.

\*Result NAME - column vector with number of rows as in MAT.

\*Position 0 means that VAL is not found in the row.

### POSITION OF RANDOM ENCOUNTER OF VALUE IN ROW/ROWS [!KO\_randindx]

\*/\*!KO\_randindx(mat%val%name)\*/\*.

\*Version 1.

\*Takes MAT - matrix or row vector. Returns index (position) of random встречи encounter

\*of value VAL in each row of MAT. That is, if VAL encounters k times in the row, one of these

\*positions will be selected randomly.

\*VAL may be a scalar or a column vector of length as the number of rows in MAT; in the latter

\*instance each row of MAT is given its corresponding value VAL.

\*Result NAME - column vector with number of rows as in MAT.

\*Position 0 means that VAL is not found in the row.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

### POSITION (PAIR OF INDICES) OF ONE VALUE IN MATRIX [!KO\_ij]

\*/\*!KO\_ij(mat%val%priority%name)\*/\*.

\*Version 2.

\*Finds in matrix MAT one element equal to value VAL (scalar).

\*Returns, as NAME, the position (indices i,j) of such element in MAT.

\*If there are more than one element equal to VAL are found, the position of one of them is

\*returned according to argument PRIORITY. If none is found, {0,0} is returned.

\*PRIORITY - capitalized keyword (may take in quotes or apostrophes):

\*"TBLR" - search top to bottom, left to right: returned is position from top-most row,

\*of left-most element.

\*"LRTB" - search left to right, top to bottom: returned is position from left-most column,

\*of top-most element.

\*"BTRL" - search bottom to top, right to left: returned is position from bottom-most row,

\*of right-most element.

\*"RLBT" - search right to left, bottom to top: returned is position from right-most column,

\*of bottom-most element.

\*"RANDOM" - of all positions of VAL, return one randomly chosen.

\*There is a tiny difference in performance speed; if you don't mind the priority itself but

\*are greedy on account of speed, select RLBT as the fastest.

EXAMPLE.

matrix.

compute x= {0,0,1,0,0;1,0,0,0,0;0,0,0,0,0;0,0,0,1,0;0,0,0,0,0;0,1,0,0,0}.

print x.

!KO\_ij(x%1%TBLR%ij).

print ij.

!KO\_ij(x%1%LRTB%ij).

print ij.

!KO\_ij(x%1%BTRL%ij).

print ij.

!KO\_ij(x%1%RLBT%ij).

print ij.

!KO\_ij(x%1%RANDOM%ij).

print ij.

end matrix.

### HIGHLIGHT IN ROW AND COLUMN NO MORE THAN ONE ELEMENT EQUAL TO VALUE [!KO\_prime]

\*/\*!KO\_prime(mat%val%name)\*/\*.

\*Version 1.

\*Takes matrix or vector MAT and value VAL.

\*Returns binary matrix NAME of size as MAT, where 1 marks elements which are equal to VAL, but the function

\*leaves in each row and column at most one such mark. In other words, the 1 in NAME

\*exclusively matches rows and columns: "one row - one column".

\*This is not the optimal matching algorithm: the result may depend on the direction of the search of the

\*elements equal to VAL. This function scans rows downward and elements in a row right to left.

\*If you wand another direction, simply change in accordance to it the order of rows and/or columns in MAT.

\*Argument VAL may be not only scalar but also a column vector of length as the number of rows in MAT.

\*In that latter case in each row of MAT there is searched and marked a different value instances, but then

\*(erasing the "superfluous" marks) the function acts same as with scalar value.

\*The function works somewhat faster with wide matrix than with high matrix.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 10000.

matrix.

compute m= rnd(uniform(9,7)\*10).

print m.

!KO\_prime(m%0%flag).

print flag /title 'Elements equal 0 are highlighted, at most one per row and col'.

end matrix.

### INDICATION OF CASES HAVING SPECIFIED PROFILE OF VALUES [!KO\_indic]

\*/\*!KO\_indic(mat%valvecs%name)\*/\*.

\*Version 1.

\*Data MAT - n cases x p variables (n>=1, p>=1).

\*Vectors of values VALVECS - m vectors x p values (m>=1), the 1-st value in a vector corresponds

\*to the 1-st variable of the data, the 2-nd value - to the 2-nd variable, and so on.

\*Returns binary indicator matrix NAME, n x m, in which a one on position (i,j) means that

\*the i-th case (row) in MAT coincides with the j-th vector (row) in VALVECS.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

matrix.

compute data=

{3, 2, 1, 0;

3, 1, 0, 3;

2, 2, 2, 4;

0, 1, 3, 0;

1, 1, 1, 1;

1, 0, 3, 2;

2, 3, 2, 2;

3, 2, 2, 2;

1, 1, 2, 1;

2, 4, 2, 4;

3, 3, 2, 2;

0, 2, 4, 4;

4, 2, 3, 2;

3, 0, 0, 1;

4, 2, 3, 1;

1, 1, 1, 1;

1, 3, 2, 3;

3, 3, 1, 2;

0, 4, 4, 1;

1, 3, 3, 2}.

print data.

compute valvecs=

{1,1,1,1;

1,2,3,4;

0,2,4,4;

1,0,3,2;

2,2,2,4}.

print valvecs.

!KO\_indic(data%valvecs%name).

print name.

end matrix.

### HIGHLIGHT DIRECT CHAINS [!KO\_runs]

\*/\*!KO\_runs(mat%maxw%dir%hl%name)\*/\*.

\*Version 2.

\*In a binary matrix MAT highlights direct chains of 1s going in succession.

\*Chains of what length will be highlighted by what "weight" value depends on argument

\*"maximal weight" MAXW (nonnegative integer scalar).

\*If MAXW>0, chain of length >=MAXW will be highlighted by number MAXW,

\*chain of length MAXW-1 will be highlighted by number MAXW-1,

\*chain of length MAXW-2 will be highlighted by number MAXW-2, and so on.

\*if MAXW is specified nonpositive (any such number), chain will be highlighted by the

\*number equal to its length, whatever long it comes to be. In other words, nonpositive

\*MAXW is the refusal of the "maximal weight" limit.

\*Argument DIR (scalar) - chains of what direction interest you: vertical, in columns

\*(positive argument), or horizontal, in rows (nonpositive argument).

\*Argument HL - digit 1 or 0 (not name or expression, you may take the digit in quotes

\*or apostrophes). This argument is about how to highlight. If HL=1, all members of a chain

\*will be highlighted by one number; if HL=0, they will be highlighted by ordinal numbers.

\*For example, if MAXW=0, the chain 1 1 1 1 1 will be highlighted as 5 5 5 5 5 under HL=1

\*and as 1 2 3 4 5 under HL=0. And if MAXW=3, it will be highlighted as 3 3 3 3 3 under HL=1

\*and as 1 2 3 3 3 under HL=0.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_runs2\*/ does the same job as this function, but is realized differently.

### HIGHLIGHT DIRECT CHAINS (ANOTHER ALGORITHM) [!KO\_runs2]

\*/\*!KO\_runs2(mat%maxw%dir%name)\*/\*.

\*Version 1.

\*In a binary matrix MAT highlights direct chains of 1s going in succession.

\*Chains of what length will be highlighted by what "weight" value depends on argument

\*"maximal weight" MAXW (positive integer scalar from 2 to num\_rows

\*or num\_columns of the matrix - depending on DIR):

\*ones in chains lengthed >=MAXW will turn into value MAXW; ones in chains

\*lengthed MAXW-1 will turn into value MAXW-1; ones in chains lengthed

\*MAXW-2 will turn ito value MAXW-2, etc.

\*Argument DIR (scalar) - chains of what direction interest you: vertical, in columns

\*(positive argument), or horizontal, in rows (nonpositive argument).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_runs\*/ does the same job as this function, but is realized differently.

\*/\*!KO\_runs2\*/ differs from /\*!KO\_runs\*/ by the following:

\*- argument MAXW must be positive;

\*- argument HL is absent;

\*- speed of /\*!KO\_runs2\*/ depends on MAXW, but of /\*!KO\_runs\*/ - not.

\*- speed of /\*!KO\_runs2\*/ is a bit higher than of /\*!KO\_runs\*/ if MAXW is not above 3 and MAT

\*is small; in all other cases /\*!KO\_runs\*/ is faster and therefore commonly preferable.

### HIGHLIGHT SLANT CHAINS [!KO\_slant]

\*/\*!KO\_slant(mat%maxw%dir%hl%name)\*/\*.

\*Version 2.

\*In a binary matrix MAT highlights slant chains of 1s going in succession.

\*Chains of what length will be highlighted by what "weight" value depends on argument

\*"maximal weight" MAXW (nonnegative integer scalar).

\*If MAXW>0, chain of length >=MAXW will be highlighted by number MAXW,

\*chain of length MAXW-1 will be highlighted by number MAXW-1,

\*chain of length MAXW-2 will be highlighted by number MAXW-2, and so on.

\*if MAXW is specified nonpositive (any such number), chain will be highlighted by the

\*number equal to its length, whatever long it comes to be. In other words, nonpositive

\*MAXW is the refusal of the "maximal weight" limit.

\*Argument DIR (scalar) - chains of what direction interest you: top-left/bottom-right

\*(positive argument), top-right/bottom-left (nonpositive argument).

\*Argument HL - digit 1 or 0 (not name or expression, you may take the digit in quotes

\*or apostrophes). This argument is about how to highlight. If HL=1, all members of a chain

\*will be highlighted by one number; if HL=0, they will be highlighted by ordinal numbers.

\*For example, if MAXW=0, the chain 1 1 1 1 1 will be highlighted as 5 5 5 5 5 under HL=1

\*and as 1 2 3 4 5 under HL=0. And if MAXW=3, it will be highlighted as 3 3 3 3 3 under HL=1

\*and as 1 2 3 3 3 under HL=0.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_slant2\*/ does the same job as this function, but is realized differently.

EXAMPLE.

matrix.

compute mat= rnd(uniform(9,11)). /\*random binary matrix

print mat.

!KO\_slant(mat%2%1%1%chains).

print chains /\*All slant chains highlighted as 2.

!KO\_slant(mat%4%1%1%chains).

print chains. /\*Slant chains of length 2 highlighted as 2,

/\*of length 3 highlighted as 3,

/\*of length 4+ highlighted as 4

!KO\_slant(mat%0%1%0%chains).

print chains. /\*Slant chains were highlighted as

/\*1 2 3 … chain\_length

end matrix.

### HIGHLIGHT SLANT CHAINS (ANOTHER ALGORITHM) [!KO\_slant2]

\*/\*!KO\_slant2(mat%maxw%dir%name)\*/\*.

\*Version 1.

\*In a binary matrix MAT highlights slant chains of 1s going in succession.

\*Chains of what length will be highlighted by what "weight" value depends on argument

\*"maximal weight" MAXW (positive integer scalar from 2 to

\*min(num\_rows,num\_columns) of the matrix): ones in chains lengthed >=MAXW will

\*turn into value MAXW; ones in chains lengthed MAXW-1 will turn into value MAXW-1;

\*ones in chains lengthed MAXW-2 will turn ito value MAXW-2, etc.

\*Argument DIR (scalar) - chains of what direction interest you: top-left/bottom-right

\*(positive argument), or top-right/bottom-left (nonpositive argument).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_slant\*/ does the same job as this function, but is realized differently.

\*/\*!KO\_slant2\*/ differs from /\*!KO\_slant\*/ by the following:

\*- argument MAXW must be positive;

\*- argument HL is absent;

\*- speed of /\*!KO\_slant2\*/ depends on MAXW, but of /\*!KO\_slant\*/ - not.

\*- speed of /\*!KO\_slant2\*/ is a bit higher than of /\*!KO\_slant\*/ if MAXW=2, or if MAXW= 3-4 and

\*MAT is small; in all other cases /\*!KO\_slant\*/ is faster and therefore commonly preferable.

### DEPTH-FIRST SEARCH IN UNIPARTITE GRAPH [!KO\_dfs]

\*/\*!KO\_dfs(adj%start%stop%forest%out%name)\*/\*.

\*Version 1.

\*Performs depth-first traversal/search of connected vertices in a unipartite unweighted graph ADJ,

\*starting from vertex START. Optional vertex STOP - point of stopping the traversal: coming up to it,

\*visiting of vertices ceases. START and STOP are scalars, vertex numbers, and must be different.

\*If you don't want to specify STOP, i.e. traversal of the graph must go maximally far, set this

\*argument to value 0.

\*Argument OUT - digit 1 or 0 (not name or expression; you may put the digit in quotes or apostrophes).

\*If 0, result NAME will contain only one column, showing, top to bottom, the order vertices were

\*visited. If 1, there will be additional computations and NAME will contain more columns:

\*2nd column shows the parent of the given visited (indicated in column 1) vertex. By "parent"

\*here is meant the most early visited vertex directing to (i.e., having it as adjacent) our vertex.

\*3rd column shows the distance of our visited vertex from vertex START (the root vertex); this

\*distance is the length of the shortest path from START to our vertex through vertices already visited

\*(in DFS, this distance from START to our vertex is not necessarily the globally minimal).

\*4th column marks, with unit, the beginning of a new chain in the visits. A chain is a continuous

\*sequence into the graph depth, where each next visited vertex is an adjacent one to the previous and

\*this edge (arc) was passed (accounted) by the algorithm. New chain begins when no further move into

\*depth is possible (for there are no more unvisited adjacent vertices left); then a new chain is

\*conceived from a yet not visited vertex lying "under" some "fork" of paths; the algorithm returns to

\*the fork to start from that vertex a new, another path into depth. DFS algorithm goals to maximize

\*the length of a chain.

\*5th column is added if FOREST=1 (see). Here is being unit-marked a starting of a new tree.

\*Argument FOREST - digit 1 or 0 (not name or expression; you may take the digit in quotes or

\*apostrophes). If 0, the algorithm will limit itself traversing into depth starting only from one

\*vertex - the one specified as START. NAME will represent a single DF tree (Depth-first tree), going

\*from the root START. If argument is 1, then - in case the tree from root START could not visit all

\*vertices of the graph - an additional root vertex is taken (from the vertices yet unvisited), and a

\*tree of yet unvisited vertices is built from it. So is repeated until lastly all vertices of the graph

\*are visited. NAME in such a case will represent not one tree but a collection of trees called

\*DF forest (Depth-first forest). The start (root vertex) of a new tree of the forest is marked with

\*unit in the 5th column of NAME (you must specify OUT=1, for NAME to have several columns). The

\*difference of a new tree from a new chain is that a tree begins from a new start, and the distances

\*in a new tree are measured from that new start, while the distances in a new chain continue to be

\*measured from the old start - from the root of the tree that chain belongs to. I.e., chains are

\*paths nested in trees.

\*Argument ADJ - input graph. The graph can be any - undirected or directed, with or without cycles,

\*with or without loops, be tree or network. It is specified via adjacency list ADJ.

\*Each row in ADJ is the graph vertex, and the row contains the numbers of the vertices adjacent to it

\*(with a directed graph, these are those it indicates to); the numbers must be listed left to right,

\*rammed. For example, if from the 2nd vertex there go edges to vertices 1,4 и 5, the 2nd row should

\*look like {1,4,5,0,...,0}, where emty spaces are padded with zeros. The order of numbers may be any:

\*{4,5,1,0,...,0}, for example, is also correct. The number of columns of ADJ must be enough to carry

\*all the information about the graph's edges. Adjacency list corresponds to a binary matrix in which

\*unit in cell (i,j) signifies the presence of the edge (arc) from vertex i to vertex j. Adjacency list

\*is another mode to represent such a matrix. If your input data are the binary matrix, you can turn it

\*into the adjacency list by function /\*!KO\_ram\*/.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. DFS and BFS.

set mxloops 1E6.

matrix.

compute mat=

{0, 0, 0, 1, 0, 1, 0, 1;

1, 1, 0, 0, 0, 0, 0, 1;

0, 0, 1, 0, 0, 0, 0, 1;

1, 0, 0, 0, 0, 0, 1, 0;

1, 0, 0, 0, 0, 1, 0, 0;

0, 1, 0, 0, 1, 0, 1, 0;

0, 0, 0, 0, 0, 0, 0, 0;

0, 0, 0, 1, 0, 0, 1, 0}. /\*A directed graph given by a asymmetric matrix

/\*(1 means the edge is directed from the vertex-row to the vertex-column)

print mat.

!KO\_ram(mat%IND%0%0%Adj%name). /\*Produce corresponding adjacency list

print Adj.

!KO\_dfs(Adj%1%0%'1'%'1'%dfs\_order). /\*Run DFS starting from vertex 1;

/\*forest=1, which means all vertices sooner or later will sure be visited

print dfs\_order. /\*The results: Tree one started with vertex 1 and comprised

/\*seven vertices; Tree two is the remaining vertex 3;

/\*The 1st chain was the 1->4->7 pass into depth, then broke;

/\*The 2nd chain was the 6->2->8 pass into depth, then broke

/\*(and note that the "parent" displayed for vertex 8 is vertex 1 despite

/\*that the actual, chain order was 2->8, not 1->8);

/\*The 3rd chain was vertex 5, one more child of 6

!KO\_bfs(Adj%1%0%'1'%'1'%bfs\_order). /\*Run BFS starting from vertex 1;

/\*forest=1, which means all vertices sooner or later will sure be visited

print bfs\_order. /\*The results: Tree one started with vertex 1 and comprised

/\*seven vertices; Tree two is the remaining vertex 3;

/\*The 1st fan of siblings was the 4,6,8 pass into breadth, then broke;

/\*The 2nd fan was 7, the only yet unvisited child of vertex 4, then broke;

/\*The 3rd fan was siblings 2,5, children of vertex 6

end matrix.

EXAMPLE.

set mxloops 1E6.

matrix.

compute Adj=

{4, 0, 0, 0;

1, 3, 7, 8;

1, 2, 3, 4;

6, 0, 0, 0;

7, 0, 0, 0;

2, 3, 0, 0;

7, 0, 0, 0;

4, 7, 0, 0}. /\*Adjacency list

print Adj.

!KO\_dfs(Adj%1%0%0%1%order). /\*Run DFS starting from vertex 1;

/\*forest=0, which means we are interested in specifically one DF tree

print order. /\*The results:

/\*The 1st chain was the 1->4->6->2->3 pass into depth, then broke

/\*(and note that the "parent" displayed for vertex 3 is vertex 6 despite

/\*that the actual, chain order was 2->3, not 6->3, and the shortest

/\*distance captured from 1 to 3 is thus length 3, not 4);

/\*The 2nd chain was just 7 (child of 2, and the dead end);

/\*The 3rd chain was just 8 (child of 2, and the dead end since vertices

/\*4 and 7 were already visited);

/\*All vertices except 5 (which is nobody’s child) got visited

end matrix.

### BREADTH-FIRST SEARCH IN UNIPARTITE GRAPH [!KO\_bfs]

\*/\*!KO\_bfs(adj%start%stop%forest%out%name)\*/\*.

\*Version 1.

\*Performs breadth-first traversal/search of connected vertices in a unipartite unweighted graph ADJ,

\*starting from vertex START. Optional vertex STOP - point of stopping the traversal: coming up to it,

\*visiting of vertices ceases. START and STOP are scalars, vertex numbers, and must be different.

\*If you don't want to specify STOP, i.e. traversal of the graph must go maximally far, set this

\*argument to value 0.

\*Argument OUT - digit 1 or 0 (not name or expression; you may put the digit in quotes or apostrophes).

\*If 0, result NAME will contain only one column, showing, top to bottom, the order vertices were

\*visited. If 1, there will be additional computations and NAME will contain more columns:

\*2nd column shows the parent of the given visited (indicated in column 1) vertex. By "parent"

\*here is meant the most early visited vertex directing to (i.e., having it as adjacent) our vertex.

\*3rd column shows the distance of our visited vertex from vertex START (the root vertex); this

\*distance is the length of the shortest path from START to our vertex through vertices visited

\*(in BFS, this distance from START to our vertex is necessarily the globally minimal).

\*4th column marks, with unit, the beginning of a new fan in the visits. A fan is a continuous

\*sequence across the graph breadth, i.e., enumeration of siblings (children of the same parent

\*visited earlier). New chain begins when there are no more siblings. DFS algorithm goals to maximize

\*the spread of a fan.

\*5th column is added if FOREST=1 (see). Here is being unit-marked a starting of a new tree.

\*Argument FOREST - digit 1 or 0 (not name or expression; you may take the digit in quotes or

\*apostrophes). If 0, the algorithm will limit itself traversing into breadth starting only from one

\*vertex - the one specified as START. NAME will represent a single BF tree (Depth-first tree), going

\*from the root START. If argument is 1, then - in case the tree from root START could not visit all

\*vertices of the graph - an additional root vertex is taken (from the vertices yet unvisited), and a

\*tree of yet unvisited vertices is built from it. So is repeated until lastly all vertices of the graph

\*are visited. NAME in such a case will represent not one tree but a collection of trees called

\*BF forest (Breadth-first forest). The start (root vertex) of a new tree of the forest is marked with

\*unit in the 5th column of NAME (you must specify OUT=1, for NAME to have several columns). The

\*difference of a new tree from a new fan is that a tree begins from a new start, and the distances

\*in a new tree are measured from that new start, while the distances in a new fan continue to be

\*measured from the old start - from the root of the tree that fan belongs to. I.e., fans are

\*paths nested in trees.

\*Argument ADJ - input graph. The graph can be any - undirected or directed, with or without cycles,

\*with or without loops, be tree or network. It is specified via adjacency list ADJ.

\*Each row in ADJ is the graph vertex, and the row contains the numbers of the vertices adjacent to it

\*(with a directed graph, these are those it indicates to); the numbers must be listed left to right,

\*rammed. For example, if from the 2nd vertex there go edges to vertices 1,4 и 5, the 2nd row should

\*look like {1,4,5,0,...,0}, where emty spaces are padded with zeros. The order of numbers may be any:

\*{4,5,1,0,...,0}, for example, is also correct. The number of columns of ADJ must be enough to carry

\*all the information about the graph's edges. Adjacency list corresponds to a binary matrix in which

\*unit in cell (i,j) signifies the presence of the edge (arc) from vertex i to vertex j. Adjacency list

\*is another mode to represent such a matrix. If your input data are the binary matrix, you can turn it

\*into the adjacency list by function /\*!KO\_ram\*/.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. See right above.

### BRON-KERBOSCH MAXIMAL CLIQUES ALGORITHM (PIVOT VERSION) [!KO\_bronkerb]

\*/\*!KO\_bronkerb(adj%return%name)\*/\*.

\*Version 1.

\*This function performs Bron-Kerbosch algorithm for enumerating maximal cliques, in its pivot

\*version. (Equivalent but a slower, base version of the algorithm - see function /\*!KO\_bronkerb2\*/.)

\*Input matrix ADJ is an undirected unweighted graph in the form of "adjacency list". Each row

\*of ADJ is a vertex in the graph, and the row contains the vertex numbers of the vertices with which

\*that vertex is adjacent (connected, neighbour). The numbers go left-to-right, rammed. For example,

\*let there vertex with number 2 be connected with vertices 1, 4, 5. Then the 2-nd row of ADJ must

\*look as: {1,4,5[,0...]}, where [0...] stands for padding of the row with zeros if necessary.

\*The number of columns in ADJ may be any sufficient to accomodate the list of all the neighbours in

\*the graph. The order of vertex numbers in a row is arbitrary: for example, that same row could be

\*written as {5,1,4[,0...]}. A vertex in a graph cannot be adjacent with itself (i.e., say vertex 2

\*cannot be found in the 2-nd row). Isolated vertices are also prohibited in the graph (i.e., a row

\*cannot consist of just zeros - but see EXAMPLE). Adjacency list ADJ corresponds to the square

\*symmetric binary adjacency matrix with zero diagonal, in which units flag the presence of a link;

\*ADJ is another mode to represent the matrix. If your input data are such binary matrix, you can

\*turn it into the adjacency list by function /\*!KO\_ram\*/.

\*Result name NAME - two-column matrix. Its 1-st column is the vertex numbers of the vertices of the

\*maximal cliques. Its 2-nd column is the clique identifier - the clique number in the order of their

\*detection by the algorithm.

\*Argument RETURN - capitalized keyword (may also take in quotes or apostrophes):

\*"ALL" - allow the algorithm to run to its end; NAME will enumerate all maximal cliques existing

\*in the graph;

\*"COUNT n" - return n maximal cliques found first, and stop; n is an integer >0, may be a digit,

\*a name, or an expression (examples of specification: COUNT 10, COUNT x, COUNT abs(z)\*2);

\*"SIZEEQ n" - return one maximal clique consisting of n vertices, and stop; n an integer >1,

\*may be a digit, a name, or an expression;

\*"SIZEGE n" - return one maximal clique consisting of >=n vertices, and stop; n an integer >1,

\*may be a digit, a name, or an expression.

\*In the two last cases - if no cliques of the requested size were found, the result will be {0,0}.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E9.

matrix.

compute Adj=

{2,9,0,0,0;

1,3,0,0,0;

2,9,4,8,0;

3,5,6,7,8;

4,6,0,0,0;

4,5,7,8,0;

4,6,8,0,0;

3,4,6,7,0;

1,2,3,0,0}. /\*Adjacency list

print Adj.

!KO\_bronkerb(adj%ALL%name).

print name /title 'All maximal cliques'.

!KO\_bronkerb(adj%SIZEGE 4%name).

print name /title 'One maximal clique of size>=4'.

end matrix.

EXAMPLE.

set mxloops 1E9.

matrix.

compute mat=

{0, 1, 1, 1, 1, 0, 0, 0;

1, 0, 0, 0, 0, 0, 0, 1;

1, 0, 0, 0, 1, 0, 1, 1;

1, 0, 0, 0, 0, 1, 0, 0;

1, 0, 1, 0, 0, 1, 1, 0;

0, 0, 0, 1, 1, 0, 1, 0;

0, 0, 1, 0, 1, 1, 0, 0;

0, 1, 1, 0, 0, 0, 0, 0}. /\*A symmetric adjacency matrix w/ zero diagonal

/\*and no empty rows/columns

print mat.

!KO\_ram(mat%IND%0%0%Adj%name). /\*Produce corresponding adjacency list

print Adj.

!KO\_bronkerb(adj%ALL%name).

print name /title 'All maximal cliques'.

end matrix.

EXAMPLE. When there are isolated vertices in the graph, they could be removed, but the more convenient is to assign them a nonexistent vertex as the neighbour (some number above the number of vertices in the graph).

set mxloops 1E9.

matrix.

compute Adj=

{2,0,0,0,0;

1,3,0,0,0;

2,4,8,0,0;

3,6,7,8,0;

0,0,0,0,0;

4,7,8,0,0;

4,6,8,0,0;

3,4,6,7,0;

0,0,0,0,0}. /\*Graph with 2 isolated vertices, 5th and 9th

print Adj.

compute Adj(5,1)= 99. /\*Assign them a nonexistent vertex

compute Adj(9,1)= 99. /\*as the only neighbour

print Adj.

!KO\_bronkerb(adj%ALL%name).

print name. /\*Verices 5 and 9 came out as "cliques" of their own

end matrix.

### BRON-KERBOSCH MAXIMAL CLIQUES ALGORITHM (BASE VERSION) [!KO\_bronkerb2]

\*/\*!KO\_bronkerb2(adj%return%name)\*/\*.

\*Version 1.

\*This function performs Bron-Kerbosch algorithm for enumerating maximal cliques, in its basic

\*version, and it is equivalent to function /\*!KO\_bronkerb\*/, but is slower than it, therefore is not

\*preferable - except the cases of very sparse graph, when it uses to be somewhat faster.

\*The arguments are the same as in /\*!KO\_bronkerb\*/. The order of cliques finding and the order of

\*vertices inside the cliques, i.e., the order of rows in the result NAME, will usually be different

\*from when /\*!KO\_bronkerb\*/ is used, but the list and composition of the cliques under RETURN=ALL is

\*always be the same as then. The vertex order in the cliques here is always one and the

\*same - ascending their number.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### CONVEX HULL: 2D JARVIS ALGORITHM [!KO\_chjarv]

\*/\*!KO\_chjarv(data%name)\*/\*.

\*Version 1.

\*Outlines convex hull (convex polygon of all peripheral points) of point cloud DATA.

\*The basic "gift-wrapping" algorithm, also called Jarvis algorithm, is designed for

\*the 2-dimensional case. DATA is n x 2 data matrix, coordinates of n points (n>=3) on plane.

\*Result NAME - list of point numbers constituting the convex hull. Points in NAME

\*go counter-clockwise of the data.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*To outline convex hull, you may also use function /\*!KO\_chgrah\*/.

EXAMPLE. See right below.

### CONVEX HULL: 2D GRAHAM ALGORITHM [!KO\_chgrah]

\*/\*!KO\_chgrah(data%name)\*/\*.

\*Version 1.

\*Outlines convex hull (convex polygon of all peripheral points) of point cloud DATA.

\*Graham's algorithm is designed for the 2-dimensional case.

\*DATA is n x 2 data matrix, coordinates of n points (n>=3) on plane. There must be no less than

\*three non-coinciding points in the data.

\*The 1-st column of NAME - the list of point numbers constituting the convex hull. If there are

\*coinciding (tied) points in the convex hull, only one instance of them is displayed in NAME;

\*however, there is the 2-nd column in NAME which informs of the number of instances each

\*point exists. Points in NAME go clockwise of the data.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*To outline convex hull, you may also use function /\*!KO\_chjarv\*/. These two functions are

\*equivalent in result, though the algorithms are different. /\*!KO\_chjarv\*/ shows all the points

\*of the hull, including tied, in the single column of the result.

EXAMPLE.

set mxloops 1E9.

matrix.

!KO\_normal(35%2%data). /\*Some points 2D coordinates

\*compute data= rnd(data\*3). /\*[If you want data to be discrete, with some

/\*points potentially coinciding (tied) and/or some points collinear\*]

!KO\_chjarv(data%jarv). /\*Run Jarvis algorithm

print jarv. /\*Result: points of the convex hull

!KO\_chgrah(data%grah). /\*Run Graham algorithm

print grah. /\*Result: points of the convex hull

save data /out= \*.

end matrix.

GRAPH /SCATTERPLOT(BIVAR)= COL1 WITH COL2. /\*You may check yourself that the results

/\*are indeed the polygon outlining the data cloud (activate data point labels)

\*Collinear points - points lying exactly on a straight line.

### MAGNITUDES OF ELEMENTS OFFSET FROM MATRIX DIAGONAL [!KO\_diagoff]

\*/\*!KO\_diagoff(nr%nc%diag%name)\*/\*.

\*Version 1.

\*The function marks the diagonal of NR x NC sized matrix with zero, while other elements

\*are marked with the magnitude of their distance (offset) from the diagonal. Under-diagonal

\*marks have (for convenience) negative sign.

\*When a matrix is nonsquare, we may say it has more than one diagonal.

\*Argument DIAG (capitalized word, optionally in quotes/apostrophes) allows

\*to choose which diagonal is to be zero-marked and offset measured off:

\*"FIRST" - first, left aka main diagonal (starts in the top-left corner of the matrix)

\*"LAST" - last aka right diagonal (ends in the bottom-right corner of the matrix)

\*"MIDDLE" - middle diagonal, it is in-between the first and the last;

\*in case abs(NR-NC) is odd number, there are two middle diagonals.

\*If the matrix is square (NR=NC), its diagonal is single, so it is unimportant then

\*how DIAG is specified.

### MAGNITUDES OF ELEMENTS OFFSET FROM MATRIX DIAGONAL BAND [!KO\_diagboff]

\*/\*!KO\_diagboff(nr%nc%name)\*/\*.

\*Version 1.

\*When a matrix is nonsquare, we may say it has more than one diagonal.

\*The first diagonal is left aka main diagonal: it starts from the top-right corner.

\*The last diagonal or right diagonal ends in the bottom-right corner. There are abs(NR-NC)+1

\*diagonals in all, they form the diagonal band.

\*The function marks all the diagonal band of NR x NC sized matrix with zero, while other

\*elements are marked with the magnitude of their distance (offset) from the band. Under-band

\*marks have (for convenience) negative sign.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

# RESTRUCTURING FUNCTIONS

### READ INLINE DATA [!KO\_read]

\*/\*!KO\_read(nr nc name) data\*/\*.

\*Version 2.

\*This function is of special form of specification. It reads values offered by you within syntax

\*and constructs matrix NAME of them. Use it to construct not big matrices.

\*The function is equivalent to the explicit construction of a matrix of the submitted elements with

\*the help of operators {,;}. Its convenience is exactly that it does not require you to write these

\*operators (which takes human time). Therefore you can simply serve a set of values separated by just blanks,

\*and the function will create matrix out of them.

\*Arguments (note, arguments in parentheses go via blank, not via %):

\*NR - number of rows in the matrix to built (positive integer: number or name, not expression)

\*NC - number of columns in the matrix to built (positive integer: number or name, not expression).

\*You are also in right to specify one of these two arguments by symbol ?. Then the function will itself

\*calculate it, dividing the number of the input values by the magnitude of the other, specified argument

\*(the division must be without the remainder).

\*The inline data themselves DATA must go after the parenthesis. Don't put period between the parenthesis

\*and DATA. Put period after DATA.

\*Data is an array of values via blank(s) or tabulation. It doesn't matter in what number of rows and columns

\*you have broken the set of values: the order of the matrix being created is defined by arguments NR, NC.

\*The function reads values as free (unstructured) text, left to right, and lines top to bottom.

\*The number of values must equal the product NRxNC.

\*What a value being read can appear?:

\*-positive, zero or negative number (examples: 2.5 -3 +4 -.001 0.15);

\*-fraction or division (examples: 1/3 -3/16 3/-16);

\*-variable name (example: x);

\*-expression in quotes or apostrohes (examples: 'x\*2' "2/3" '-ln(4)').

\* Reading values, the function removes quotes/apostrophes off them (use function /\*!KO\_read2()\*/,

\* if you want to keep quotes/apostrophes, i.e. read the values as string).

\*Don't use the current function to read any big dataset (thousands of elements) because it is slow

\*(faster will be to take the data from syntax by command DATA LIST and then open them in MATRIX).

\*This function is useful for routine work, not for programming algorithms.

EXAMPLE.

matrix.

!KO\_read(6 4 mx)

.63042603368 -.3811789 -.37968575 -.3978886

.31876472299 .2149328 -.4382288945 .237998

.267827629 .51122279 .1124206621 -.27705817

-.41313506 .042735859 .699697012798 -.031

.7608927337 .1262283 -.18491719 .622383

.72269674851 -.2762052 .10438647 -.219443.

print mx.

end matrix.

EXAMPLE.

matrix.

compute nr= 6.

compute x= 10.

!KO\_read(nr ? mx)

.63042603368 x -.37968575 -1/3

.31876472299 .2149328 -.4382288945 .237998

.267827629 .51122279 .1124206621 -.27705817

-.41313506 .042735859 x -.031

.7608927337 .1262283 -.18491719 -x

.72269674851 -.2762052 .10438647 "-.219443\*x".

print mx.

end matrix.

### READ INLINE DATA (NO UNQUOTING) [!KO\_read2]

\*/\*!KO\_read2(nr nc name) data\*/\*.

\*Version 2.

\*This function is identical to /\*!KO\_read()\*/ with the only distinction that it does not take

\*quotes/apostrophes off the value being read. Therefore if the value is quoted or surrounded by

\*apostrophes, that value is taken as a string element into matrix NAME.

### SPARSE MATRIX INTO LIST OF NONZERO VALUES [!KO\_nzlist]

\*/\*!KO\_nzlist(mat%name1%name2)\*/\*.

\*Version 1.

\*Takes matrix (or vector) MAT of any size. Copies out from MAT all nonzero elements in one list.

\*That list is the 2nd column of NAME2, and in the 1st column of NAME2 there the column numbers are listed in

\*which the copied values are from. NAME2 has as many rows as the number of nonzero elements in MAT.

\*Column vector NAME1 has the length as the number of rows in MAT, and there is written how many nonzero

\*elements has each row of MAT.

\*If MAT has no nonzero elements the function returns NAME1 and NAME2 as zero scalars.

\*If you transpose input MAT the results will be different still informationally equivalent.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_nzunlist\*/ is opposite to this and can restore MAT from NAME1 and NAME2.

EXAMPLE. See right below.

### LIST INTO SPARSE MATRIX [!KO\_nzunlist]

\*/\*!KO\_nzunlist(nnz%list%name)\*/\*.

\*Version 1.

\*Creates (sparse) matrix by writing in values from a list into the empty matrix.

\*Takes nonzero column vector NNZ which values must be nonnegative integers.

\*These values are the numbers of elements being written in each row of the being created matrix NAME (NAME will

\*have as many rows as NNZ has).

\*The values to write into NAME are borrowed by the function from the 2nd column of the 2-column matrix LIST,

\*and information to which column should a given value be put - is taken from the corresponding

\*element of the 1st column of LIST. (Values in 1st col of LIST must be all positive integers. The values being

\*written down themselves - 2nd col of LIST - may be any.)

\*In 1st row of NAME there will be written NNZ(1) values of LIST, in 2nd row of NAME there will be written

\*NNZ(2) next values of LIST, and so on.

\*The created matrix NAME has as many columns as the maximal value in the 1st column of LIST.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_nzlist\*/ is opposite to this and can copy out values from NAME, creating NNZ and LIST, if

\*the values to be copy are all the nonzero values existing in NAME.

EXAMPLE.

matrix.

!KO\_normal(6%8%mat). /\*Generate some random matrix of values

compute mat= mat&\*(abs(mat)>1.5). /\*Small values turn to 0 (make sparsity)

print mat /title 'MAT'.

!KO\_nzlist(mat%nnz%list).

print nnz /title 'NNZ: Number of nonzero values in each row'.

print list /title 'LIST: All the nonzero values (with corresponding column number)'.

!KO\_nzunlist(nnz%list%newmat).

print newmat /title 'NEWMAT: Back compilation of the matrix'.

end matrix.

### RAMMING HORIZONTALLY NONZERO ELEMENTS OR THEIR INDICES [!KO\_ram]

\*/\*!KO\_ram(mat%indval%lr%rand%name1%name2)\*/\*.

\*Version 2.

\*Takes matrix or vector MAT and shifts all nonzero elements in each row of MAT to left or right

\*border, compacting them (no zeros will be among them).

\*Argument INDVAL - capitalized keyword (may take in quotes or apostrophes):

\*"VAL" - shift nonzero values themselves

\*"IND" - shift indices of nonzero values (i.e., their column numbers)

\*"BOTH" - do both tasks

\*Argument LR - scalar. If nonpositive, the shift will be to the left; if positive, then to the right.

\*Result NAME1 is the matrix with the ramming (shifting) of values or their indices done,

\*sized as MAT. With INDVAL=BOTH, both results are returned, stacked: the matrix with indices joint

\*below the matrix with values.

\*Result NAME2 is column of length as the number of rows of MAT, showing number of nonzero elements in

\*each row of MAT/NAME1.

\*Argument RAND - digit (not name or expression) 0 or 1 (may put in quotes or apostrophes).

\*If 1, action of randomizing will be added to the shifting: rammed values (and their indices)

\*will go in random order, own in each row of NAME1.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1000000.

matrix.

compute m= rnd(uniform(12,10)\*10-4).

compute m= m&\*(abs(m)>3).

print m /title 'Matrix with zero and nonzero elements'.

!KO\_ram(m%VAL%0%0%name1%name2).

print name1 /title 'Rammed to the left (nonzero values)'.

print name2.

!KO\_ram(m%IND%0%0%name1%name2).

print name1 /title 'Rammed to the left (indices of nonzero values))'.

end matrix.

EXAMPLE. Obtain categorical multiple response set (with left-aligned structure) from dichotomous multiple response set (i.e. from a set of binary variables).

set mxloops 1000000.

matrix.

compute bin= rnd(uniform(20,8)).

print bin. /\*Binary data (dichotomous MR set)

!KO\_ram(bin%IND%0%0%name1%name2).

print name1. /\*Categorical MR set

print name1(:,1:cmax(name2)). /\*with empty columns dropped

end matrix.

### RAMMING VERTICALLY NONZERO ELEMENTS OR THEIR INDICES [!KO\_vram]

\*/\*!KO\_vram(mat%indval%tb%rand%name1%name2)\*/\*.

\*Version 1.

\*This function is like /\*!KO\_ram\*/, but shifts elements vertically, not horizontally.

\*Takes matrix or vector MAT and shifts all nonzero elements in each column of MAT to top or bottom

\*border, compacting them (no zeros will be among them).

\*Argument INDVAL - capitalized keyword (may take in quotes or apostrophes):

\*"VAL" - shift nonzero values themselves

\*"IND" - shift indices of nonzero values (i.e., their row numbers)

\*"BOTH" - do both tasks

\*Argument TB - scalar. If nonpositive, the shift will be to the top; if positive, then to the bottom.

\*Result NAME1 is the matrix with the ramming (shifting) of values or their indices done,

\*sized as MAT. With INDVAL=BOTH, both results are returned, harped: the matrix with indices joint

\*right of the matrix with values.

\*Result NAME2 is row of length as the number of columns of MAT, showing number of nonzero elements in

\*each column of MAT/NAME1.

\*Argument RAND - digit (not name or expression) 0 or 1 (may put in quotes or apostrophes).

\*If 1, action of randomizing will be added to the shifting: rammed values (and their indices)

\*will go in random order, own in each column of NAME1.

\*To manage random numbers seed use out-of-matrix generator

\*of random numbers (corresponds to commands SET MTINDEX or SET SEED).

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### PROPAGATE ROWS [!KO\_propag]

\*/\*!KO\_propag(mat%times%name)\*/\*.

\*Version 1.

\*Takes matrix or vector MAT of any size. Propagates its rows in the number of copies

\*specified in TIMES. In output matrix NAME the copies of each row go adjacently.

\*TIMES - either positive integer, or vector lengthed as the number of rows of MAT where there is given

\*the number of copies for each row separately. Values in the vector - nonnegative integers, at least one

\*value greater than zero. Zero value indicates not to take that row of MAT into NAME at all.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1000000.

matrix.

compute m= rnd(uniform(6,4)\*10).

print m.

!KO\_propag(m%2%mmm).

print mmm.

!KO\_propag(m%{1,2,3,0,1,4}%mmm).

print mmm.

end matrix.

### SELECT ROWS (FILTER CASES)

If you have data matrix DATA and a binary column vector FILT lengthed as the number of rows therein, which serves as filter of rows, “cases” (1= select the row, 0= not select the row), and you want to do something (analysis, transformation) only with the “subsample” of selected cases or you want to retain only these selected cases of the dataset, – then you may go either way: (A) write out indices of the 1s, their positions in FILT, by function /\*!KO\_indices\*/ (that will yield vector IND) and get the needed subsample as DATA(IND,:); or (B) use function /\*!KO\_propag\*/ with vector FILT as its TIMES argument. Variant (A) usually is somewhat faster; besides, it allows to insert the selected subsample of rows (cases) back into matrix DATA because vector of positions IND has remained after function /\*!KO\_indices\*/. See also function /\*!KO\_split\*/.

### ADD A CONSTANT TO ELEMENTS OF A MATRIX TRIANGLE [!KO\_tradd]

\*/\*!KO\_tradd(mat%tr%val%name)\*/\*.

\*Version 1.

\*Adds number VAL (scalar) to elements of the upper or the lower triangle of MAT.

\*Matrix MAT needs not be square, but must be at least size 2x2.

\*TR - scalar; if positive, VAL is added to elements above the main diagonal;

\*if nonpositive, VAL is added to elements below the main diagonal.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### MULTIPLY ELEMENTS OF A MATRIX TRIANGLE BY A CONSTANT [!KO\_trmult]

\*/\*!KO\_trmult(mat%tr%val%name)\*/\*.

\*Version 1.

\*Multiplies elements of the upper or the lower triangle of MAT by number VAL (scalar).

\*Matrix MAT needs not be square, but must be at least size 2x2.

\*TR - scalar; if positive, elements above the main diagonal are multiplied by VAL;

\*if nonpositive, elements below the main diagonal are multiplied by VAL.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### EXPONENTIATE ELEMENTS OF A MATRIX TRIANGLE BY A CONSTANT [!KO\_trexp]

\*/\*!KO\_trexp(mat%tr%power%name)\*/\*.

\*Version 1.

\*Exponentiates (raises to a power) elements of the upper or the lower triangle of MAT.

\*The exponent - POWER (scalar). Limitations:

\*If negative values are present in the triangle of MAT, POWER must be integer number.

\*If zero values are present in the triangle of MAT, POWER must be positive number.

\*Matrix MAT needs not be square, but must be at least size 2x2.

\*TR - scalar; if positive, elements above the main diagonal are exponentiated;

\*if nonpositive, elements below the main diagonal are exponentiated.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### UNFOLD THE TRIANGLES OF SQUARE MATRIX INTO VECTORS [!KO\_unftri]

\*/\*!KO\_unftri(mat%name1%name2%name3)\*/\*.

\*Version 2.

\*Takes matrix MAT which must be square, and returns 2-column matrix NAME1,

\*where the 1st column is the unfolding of below-diagonal (lower-left) triangle of MAT

\*and the 2nd column is the unfolding of above-diagonal (upper-right) triangle of MAT.

\*A pair of elements in the same row of NAME1 are symmetrical-positioned elements in MAT.

\*NAME2 are the corresponding to NAME1 indices (positions) of off-diagonal elements in the

\*vector that is the unfold (vectorizing) of matrix MAT. 1-st row of NAME2 pertains to

\*elements of below-diagonal triangle, and 2-nd row - to elements of above-diagonal triangle.

\*With the help of NAME2, you can quickly unfold triangles of any matrix sized as MAT,

\*without resort a second time to this function.

\*NAME3 contains memo - the numbers of rows and columns of MAT where the elements came from.

\*Thus, the 1st pair of indices in NAME3 is 2 and 1, which reminds that the 1st pair of values

\*of NAME1 are: the 2nd element from the 1st column and the 2nd element from the 1st row.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_foltri\*/ is opposite to this. It folds a pair of vectors NAME1 into the triangles

\*of the square matrix MAT.

EXAMPLE.

set mxloops 10000.

matrix.

compute m= {11,12,13,14,15;21,22,23,24,25;31,32,33,34,35;41,42,43,44,45;51,52,53,54,55}.

print m.

!KO\_unftri(m%unfold%posinvec%memo).

print unfold /clabels= 'LowTr' 'UppTr'.

print t(memo) /title 'Where pairs of symmetric elements came from:'.

print t(posinvec) /title 'Positions of elements after turning M to vector'.

print /title '------------'.

\*You can use indices "posinvec" to unfold quickly another matrix of the

\*same size as M.

compute mat= uniform(5,5).

print mat /format= f8.3.

compute mat= reshape(mat,25,1). /\*Reshape it into vector first

print mat /format= f8.3.

print {mat(posinvec(1,:)),mat(posinvec(2,:))}

/format= f8.3 /clabels= 'LowTr' 'UppTr'. /\*The unfolded triangles

end matrix.

### FOLD TWO VECTORS INTO THE TRIANGLES OF SQUARE MATRIX [!KO\_foltri]

\*/\*!KO\_foltri(vecs%name1%name2%name3)\*/\*.

\*Version 2.

\*Takes matrix VECS, which must have 2 columns (the vectors), and returns square

\*matrix NAME1, in which the below-diagonal (lower-left) triangle is the folded 1st vector

\*and the above-diagonal (upper-right) triangle is the folded 2nd vector.

\*A pair of elements in the same row of VECS become symmetrical-positioned elements in NAME1.

\*The diagonal of NAME1 consists of zeros.

\*NAME2 are the corresponding to NAME1 indices (positions) of off-diagonal elements in the

\*vector that is the unfold (vectorizing) of matrix NAME1. 1-st row of NAME2 pertains to

\*elements of below-diagonal triangle, and 2-nd row - to elements of above-diagonal triangle.

\*NAME3 is a scalar, the size of matrix NAME1.

\*If the number of rows in VECS does not permit to create square matrix NAME1, then NAME3

\*will appear a noninteger number, and NAME1 and NAME2 will be returned as zero scalars.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_unftri\*/ is opposite to this. It unfolds the triangles of the square matrix NAME1

\*into the pair of vectors VECS.

EXAMPLE.

set mxloops 10000.

matrix.

compute vecs= uniform(15,2).

print vecs.

!KO\_foltri(vecs%mat%posit%size).

print mat.

print posit.

\*You can use indices "posit" to fold quickly some other unfolded

\*triangles of the same size as "vecs".

compute vecs2= rnd(uniform(15,2)\*10).

print vecs2.

compute mat2= make(size\*\*2,1,0). /\*Initialize mat2 as empty vector of length size^2

compute mat2(posit(1,:))= vecs2(:,1). /\*Populate it with values of lower triangle

compute mat2(posit(2,:))= vecs2(:,2). /\*and of upper triangle

compute mat2= reshape(mat2,size,size). /\*Reshape into square matrix

print mat2.

end matrix.

### SYMMETRIZATION OF SQUARE MATRIX: REPLACE ONE TRIANGLE BY THE OTHER [!KO\_symtri1]

\*/\*!KO\_symtri1(mat%tr%name)\*/\*.

\*Version 1.

\*Takes matrix MAT which must be square, and replaces one triangle in it by the other one;

\*Returns square symmetric matrix NAME.

\*Argument TR orders, which triangle to replace:

\*TR non-positive scalar => below-diagonal triangle be replaced by above-diagonal triangle;

\*TR positive scalar => above-diagonal triangle be replaced by below-diagonal triangle.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### SYMMETRIZATION OF SQUARE MATRIX: REPLACE GREATER ELEMENT BY LESSER OR LESSER BY GREATER [!KO\_symtri2]

\*/\*!KO\_symtri2(mat%el%name)\*/\*.

\*Version 2.

\*Takes matrix MAT which must be square, and replaces in it each off-diagonal element

\*by the one position-symmetric to it, if the latter is greater (or lesser) than it;

\*returns square symmetric matrix NAME.

\*Argument EL orders, when to do the replacement:

\*EL non-positive scalar => replace element by its symmetric, if the latter is lesser than it;

\*EL positive scalar => replace element by its symmetric, if the latter is greater than it.

EXAMPLE.

matrix.

compute mat= {1,2,3,4;5,6,7,8;9,10,11,12;13,14,15,16}.

print mat.

!KO\_symtri2(mat%0%mat).

print mat.

end matrix.

### SORTING OF VECTOR [!KO\_sort]

\*/\*!KO\_sort(vec%ord%name1%name2)\*/\*.

\*Version 2.

\*Sorts elements in vector (row or column) VEC by magnitude of the values.

\*Returns sorted vector NAME1.

\*And also vector NAME2 which is the element numbers in NAME1 arranged in the sequence

\*in what they were prior the sorting (with the help of NAME2 you can restore the initial order

\*of elements). NAME2 is the integer ranks of values of VEC.

\*Argument ORD orders the order of sorting:

\*ORD non-positive scalar => descending; ORD positive scalar => ascending.

\*If you need then to reorder rows of an external array the same way the elements in VEC appeared to be

\*sorted, use NAME2 as COL in /\*!KO\_rsortc\*/ and set there ORD as 0.

EXAMPLE.

matrix.

compute vec= uniform(10,1).

print vec /title 'Vector'.

!KO\_sort(vec%0%svec%rnk).

print svec /title 'Sorted vector'.

print rnk /title 'Ranks which defined the sort'.

print svec(rnk) /title 'Restored order'.

end matrix.

EXAMPLE. Sorting values in the 2nd column of matrix.

matrix.

compute mat= {4,3,2;3,1,1;6,3,4;2,1,5;1,2,3;3,4,5;6,2,7;7,4,6;3,4,1;5,1,2;1,2,7}.

print mat.

!KO\_sort(mat(:,2)%-1%col%name2).

compute mat(:,2)= col.

print mat.

end matrix.

### SIMPLE SORTING OF ROWS IN MATRIX [!KO\_rsort]

\*/\*!KO\_rsort(mat%cn%ord%name1%name2)\*/\*.

\*Version 2.

\*Sorts rows in matrix MAT by values of its column CN.

\*Returns matrix NAME1.

\*And also column NAME2 which is the row numbers in NAME1 arranged in the sequence in what

\*the rows were prior the sorting (with the help of NAME2 you can restore the initial order

\*of rows). NAME2 is the integer ranks of values of column CN (prior sorting).

\*CN is the column number, positive integer scalar.

\*Argument ORD orders the order of sorting:

\*ORD non-positive scalar => descending; ORD positive scalar => ascending.

\*If you need then to reorder rows of an external array the same way the rows in MAT appeared to be

\*sorted, use NAME2 as COL in /\*!KO\_rsortc\*/ and set there ORD as 0.

EXAMPLE. Sorting rows of matrix by its 2nd column.

matrix.

compute mat= {4,3,2;3,1,1;6,3,4;2,1,5;1,2,3;3,4,5;6,2,7;7,4,6;3,4,1;5,1,2;1,2,7}.

print mat.

!KO\_rsort(mat%2%1%mat%rnk).

print mat.

print rnk. /\*Ranks which defined the sort

print mat(rnk,:). /\*Restore order

end matrix.

### SORTING OF ROWS IN MATRIX BY VALUES OF EXTERNAL COLUMN [!KO\_rsortc]

\*/\*!KO\_rsortc(mat%col%ord%name1%name2)\*/\*.

\*Version 2.

\*Sorts rows in matrix MAT by values of external column COL.

\*COL is a column vector with length as the number of rows in MAT.

\*Returns matrix NAME1.

\*And also column NAME2 which is the row numbers in NAME1 arranged in the sequence in what

\*the rows were prior the sorting (with the help of NAME2 you can restore the initial order

\*of rows). NAME2 is the integer ranks of values of COL.

\*Argument ORD orders the order of sorting:

\*ORD negative scalar => descending; ORD positive scalar => ascending.

\*If ORD is 0 then COL must contain integer ranks, and the order of sorting is defined now by them.

\*At that, the function spends no time at ranking. This way of specification is advantageous in the sense of

\*speed, when you want to sort out rows in several matrices (with the same number of rows) in identical

\*order but you don't want to joint temporarily the matrices in one for that. Then you may use

\*NAME2 obtained at sorting of the first matrix as COL in sorting of the rest of the matrices.

EXAMPLE.

set seed 453473.

matrix.

compute mat= {4,3,2;3,1,1;6,3,4;2,1,5;1,2,3;3,4,5;6,2,7;7,4,6;3,4,1;5,1,2;1,2,7}.

print mat.

compute sortby= uniform(nrow(mat),1).

print sortby. /\*External column to sort by its values

!KO\_rsortc(mat%sortby%1%smat%rnk).

print smat. /\*Matrix with rows sorted by values of the SORTBY column

print rnk. /\*Ranks which defined the sort

print smat(rnk,:). /\*(Restored initial sort order)

compute mat2= {112,67;84,209;314,475;24,199;251,87;146,403;274,88;97,150;251,282;333,460;72,302}.

print mat2. /\*One more matrix with the same number of rows as MAT, to sort samely

!KO\_rsortc(mat2%rnk%0%smat2%rnk). /\*To sort its rows by SORTBY quicker, you may use

/\*ranks RNK (instead of SORTBY) and set argument ORD to 0

print smat2.

end matrix.

### HIERARCHICAL SORTING OF ROWS IN MATRIX [!KO\_hiesort]

\*/\*!KO\_hiesort(mat%ord%name1%name2)\*/\*.

\*Version 2.

\*Sorts rows in matrix MAT by values of all its columns.

\*Rows are sorted hierarchically: by values of the 1st column; within its identical

\*values – by values of the 2nd column; within its identical values - by

\*values of the 3rd column; etc. This function is like out-of-matrix SORT CASES command

\*when you specify several variables in that command.

\*Returns matrix NAME.

\*And also column NAME2 which is the row numbers in NAME1 arranged in the sequence in what

\*the rows were prior the sorting (with the help of NAME2 you can restore the initial order

\*of rows).

\*Argument ORD is a vector lengthed as the number of columns in MAT or is a scalar;

\*it requests the order of sorting in each separate column:

\*n-th element of ORD non-positive => descending sorting in n-th column;

\*n-th element of ORD positive => ascending sorting in n-th column.

\*Scalar ORD means same order for all the columns.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*The function is relatively slow.

\*If you need then to reorder rows of an external array the same way the rows in MAT appeared to be

\*sorted, use NAME2 as COL in /\*!KO\_rsortc\*/ and set there ORD as 0.

EXAMPLE. Hierarchical sorting in MATRIX and outside MATRIX.

matrix.

compute mat= {4,3,2;3,1,1;6,3,4;2,1,5;1,2,3;3,4,5;6,2,7;7,4,6;3,4,1;5,1,2;1,2,7}.

print mat.

save mat /outfile= \* /variables= v1 v2 v3.

!KO\_hiesort(mat%{-1,1,1}%mat%name2).

print mat.

end matrix.

sort cases v1(D) v2(A) v3(A).

list v1 v2 v3.

### SORT IN RANDOM ORDER

Use function /\*!KO\_sample\*/.

### SPLITTING OF MATRIX INTO TWO [!KO\_split]

\*/\*!KO\_split(whole%dim%ind%name1%name2)\*/\*.

\*Version 2.

\*From matrix WHOLE, either its rows or its columns with indices IND are cut out,

\*forming cutting matrix NAME1. The remaining rows (columns) of WHOLE form

\*remnant matrix NAME2. The function returns both matrices NAME1 and NAME2.

\*Argument DIM - keyword (optionally quoted or in apostrophes):

\*"ROWS" - to cut out rows (word "arrays" below stands for rows);

\*"COLS" - to cut out columns (word "arrays" below stands for columns).

\*WHOLE must have the number of arrays more than one.

\*IND is a vector (row or column, doesn't matter) with length less than there

\*are arrays in WHOLE, and containing positive integers: indices of arrays in WHOLE

\*that should be cut out. These numbers do not have to go in ascending order; their sequence

\*defines the sequence of arrays how it will be in the cutting matrix NAME1.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_merge\*/ is opposite to this one and can insert cut-in NAME1 into base NAME2

\*restoring matrix WHOLE.

EXAMPLE. Temporary split a matrix into two for separate operations on those.

matrix.

compute mat= {4,3,2;3,1,1;6,3,4;2,1,5;1,2,3;3,4,5;6,2,7;7,4,6;3,4,1;5,1,2;1,2,7}.

print mat.

compute cutrows= {2,3,5,9}.

!KO\_split(mat%ROWS%cutrows%cutting%remnant).

print cutting.

print remnant.

compute remnant= remnant\*10.

!KO\_rsort(cutting%3%1%cutting%name2).

print remnant.

print cutting.

!KO\_merge(remnant%ROWS%cutting%cutrows%mat).

print mat.

end matrix.

### MERGING OF TWO MATRICES [!KO\_merge]

\*/\*!KO\_merge(base%dim%slice%ind%name)\*/\*.

\*Version 2.

\*Into base matrix BASE, all rows or columns of cut-in matrix SLICE are inserted, forming the

\*united matrix NAME which is returned. Rows (columns) constituting matrix SLICE will occupy in NAME

\*positions specified in IND.

\*Argument DIM - keyword (optionally quoted or in apostrophes):

\*"ROWS" - insert SLICE rows as rows in BASE (word "arrays" below stands for rows), number of columns

\*in BASE and SLICE must be equal;

\*"COLS" - insert SLICE columns as columns in BASE (word "arrays" below stands for columns), number of

\*rows in BASE and SLICE must be equal.

\*IND is a vector (row or column, doesn't matter) with length as the number of arrays in SLICE,

\*and containing positive integers: indices of arrays in NAME which must be occupied by arrays of

\*the cut-in matrix SLICE. These numbers do not have to go in ascending order; their sequence defines

\*the sequence of cutting-in of arrays of SLICE between arrays of BASE.

\*The maximal value in IND cannot exceed the sum of the number of arrays in BASE and SLICE.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Function /\*!KO\_split\*/ is opposite to this one and can split NAME back into

\*cutting SLICE and remnant BASE.

### SPLITTING OF MATRIX HORIZONTALLY IN SEVERAL, BY GROUPS [!KO\_msplit]

\*/\*!KO\_msplit(mat%group%'maxcode')\*/\*.

\*Version 1.

\*Takes matrix MAT (some name, not expression) and grouping variable GROUP (sorted).

\*Splits MAT horizontally in matrices corresponding to groups in GROUP. Names of the resulting

\*matrices are defined automatically.

\*The function is partly similar to command SPLIT FILE.

\*Details:

\* GROUP - case-grouping variable, column with integer group codes: 1, 2, 3,...

\*(Fractional codes and values less than 1 not allowed.) Cases (rows) must be already sorted by

\*the codes ascendingly.

\*MAT - matrix (data) with num-of-rows as in GROUP, already sorted correspondingly with GROUP, which

\*you need to break horizontally: one resultant matrix for each code of GROUP.

\*The names will be concatenated from name of MAT and the group code: MAT1, MAT2,...

\*'MAXCODE' - code of the last observed or of your interest group (maximal 999). It must be number

\*(not variable or expression), you may put it in apostrophes or quotes.

\*If some codes between 1 and maxcode are absent in GROUP the matrices for these missing groups

\*will not be created.

\*This function is useful for routine work, not for programming algorithms.

EXAMPLE.

sort cases by group. /\*Group codes must be 1, 2,..., sorted

matrix.

get data /variables= v1 to v3.

get gr /variable= group.

!KO\_msplit(data%gr%'4').

print data1.

print data2.

print data3.

print data4.

end matrix.

### RESTRUCTURING "VARIABLES INTO CASES" [!KO\_vartocas]

\*/\*!KO\_vartocas(data%k%ord%name1%name2)\*/\*.

\*Version 1.

\*Restructures each group of K variables (columns) of DATA into single column.

\*This way each case (row) gets broken in K rows.

\*Returns restructured data NAME1.

\*The function is similar to SPSS command VARSTOCASES.

\*DATA - matrix with number of columns multiple of K.

\*K - positive integer scalar; each K columns of DATA (each "group") will produce one column in NAME1.

\*Scalar ORD indicates in which order the variables producing single column are situated in DATA:

\*situated adjacently (ORD positive) or situated with step number\_columns\_DATA/K (ORD nonpositive).

\*For example, if number\_columns\_DATA=6 and K=2 then adjacent situation means that columns

\*1 and 2 will produce column, 3 and 4 will produce column, 5 and 6 will produce column. And with situation via step 3

\*columns 1 and 4 will produce column, 2 and 5 will produce column, 3 and 6 will produce column.

\*Two-column matrix NAME2 reflects that order: it contains case numbers of DATA in its 1st column and

\*the sequential index of a variable in "group" (1,2,...,K) of variables in its 2nd column.

\*Function \*/\*!KO\_castovar\*/\* is opposite to this one and restructures NAME1 back in DATA.

### RESTRUCTURING "CASES INTO VARIABLES" [!KO\_castovar]

\*/\*!KO\_castovar(data%k%ord%name1%name2)\*/\*.

\*Version 1.

\*Restructures each group of K cases (rows) of DATA into single row.

\*This way each variable (column) gets broken in K columns.

\*Returns restructured data NAME1.

\*The function is similar to SPSS command CASESTOVARS.

\*DATA - matrix with number of rows multiple of K.

\*K - positive integer scalar; each K rows of DATA (each "group") will produce one row in NAME1.

\*Rows in DATA must be already sorted so that K rows of each group appear adjacent.

\*Scalar ORD specifies in which order should the variables produced of single variable of DATA be situated:

\*adjacently (ORD positive) or with step number\_columns\_DATA (ORD nonpositive).

\*For example, if number\_columns\_DATA=3 и K=2 then adjacent situation means

\*that column 1 will produce columns 1 and 2, column 2 will produce columns 3 and 4, column 3 will produce columns 5 and 6.

\*And with situation via step 3 column 1 will produce columns 1 and 4, column 2 will produce columns 2 and 5,

\*column 3 will produce columns 3 and 6.

\*Two-row matrix NAME2 reflects that order: it contains variable numbers of DATA

\*in its 1st row and the sequential index of a case in "group" (1,2,...,K) of cases in its 2nd row.

\*Function \*/\*!KO\_vartocas\*/\* is opposite to this one and restructures NAME1 back in DATA.

EXAMPLE.

set seed 34534753.

matrix.

compute data= uniform(10,15)\*10.

print data /format= f6.2 /title '10 cases X 15 variables Data'.

!KO\_vartocas(data%3%1%vartocas%ord).

print {ord,vartocas} /format= f6.2 /clab= 'CaseNo' 'Index'

/title 'Vars-to-Cases, K=3'.

!KO\_castovar(vartocas%3%1%castovar%ord).

print {ord;castovar} /format= f6.2 /rlab= 'VarNo' 'Index'

/title 'Cases-to-Vars, K=3, back into Data'.

end matrix.

### SHIFT ROWS HORIZONTALLY [!KO\_shift]

\*/\*!KO\_shift(mat%rel%displ%name)\*/\*.

\*Version 1.

\*Takes matrix MAT with no less than two rows. Returns matrix NAME where rows were

\*shifted horizontally - to right or to left by the amounts (number of elements) specified in column

\*vector DISPL. DISPL must be of the length 1 less than the number of rows in MAT. Its values - integers.

\*Negative value is shift by this number of elements to the left, positive value is shift by this number

\*of elements to the right. The 1st element is the shift for row 2, 2nd element is the shift of

\*row 3, etc.

\*Argument REL (scalar) explains relative what the shifts in DISPL are specified. If REL is nonpositive

\*then shift of every row is the shift relative the 1st row. If REL is positive then shift of every

\*row is the shift relative the previous, overlying row.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Sequential shifting of cases by 1 case down.

matrix.

get vars /variables= v1 v2 v3 v4.

print vars.

compute shifts= make(1,ncol(vars)-1,1).

print shifts.

!KO\_shift(t(vars)%1%t(shifts)%vars2).

compute vars2= t(vars2).

print vars2.

end matrix.

### BLOCK-INCUT [!KO\_blockic]

\*/\*!KO\_blockic(mat1%mat2%pos%name)\*/\*.

\*Version 1.

\*Takes matrices MAT1 and MAT2 of any size and unites them, in-cutting MAT2 from bottom-right

\*into MAT1, docking it to position POS in MAT1. Specifically, first, matrix NAME is created

\*as block matrix: block(MAT1(1:I,1:J),MAT2). Then, if there are left cells in NAME not yet

\*received values from their corresponding cells of MAT1, they receive them.

\*POS - vector of length 2, containing positions I and J in MAT1. I can be from 1 to num\_rows

\*of MAT1. J can be from 1 to num\_columns of MAT1. When I and J are both maximal, the function

\*turns into function block().

\*If you want to "revert" the task and in-cut MAT2 into MAT1 from top-left, then first

\*change the order of rows and columns in both matrices to the opposite, and in the end change

\*the order of rows and columns in NAME to the opposite.

EXAMPLE.

matrix.

compute m1=

{46,25,15;

3,77,75;

50,40,19;

25,28,67;

57,25,88}.

compute m2=

{-70,-90;

-69,-35;

-8,-21}.

print m1.

print m2.

print {3,2}.

\*Block-incut (from bottom-right).

!KO\_blockic(m1%m2%{3,2}%m).

print m /title 'Block-incut (to position 3,2)'.

\*Now block-incut from top-left.

compute m1= m1(nrow(m1):1,ncol(m1):1).

compute m2= m2(nrow(m2):1,ncol(m2):1).

!KO\_blockic(m1%m2%{3,2}%m).

compute m= m(nrow(m):1,ncol(m):1).

print m

/title 'Reversed Block-incut (to position 3,2 meas. off the bottom-right corner)'.

end matrix.

# COMBINATORIAL FUNCTIONS AND SETS

### DYADIC SET OPERATIONS [!KO\_setdo]

\*/\*!KO\_setdo(set1%set2%oper%name)\*/\*.

\*Version 1.

\*Takes two sets SET1 and SET2 (arbitrary matrices) and returns their union, intersection,

\*difference, or symmetric difference. Value 0 in SET1 and SET2 is reserved for emptiness, i.e.,

\*it is not considered an element of a set. Any other value is an element and is accounted

\*once in the resulting row vector NAME (that is, for example, intersection between {1,2,2,2,3}

\*and {1,2,2,3,4} will be {1,2,3}, and not {1,2,2,3} or {1,2,2,2,3}).

\*OPER - capitalized keyword (may be quoted or in apostrophes):

\*"UNION" - union of two sets (values present at least in one of the sets);

\*"INTER" - intersection of two sets (values present in both sets);

\*"DIFF" - difference of two sets (values present in SET1 but not in SET2);

\*"SYMDIFF" - symmetric difference of two sets (values present in SET1 but not in SET2, or

\*in SET2 but not in SET1).

\*NAME = scalar 0 means empty set at output.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E6.

matrix.

compute x= {-1;0;1;2;3}.

compute y= {1;2;3;4;5}.

!KO\_setdo(x%y%SYMDIFF%result).

print result.

end matrix.

### CARTESIAN PRODUCT BETWEEN TWO SETS [!KO\_cartp]

\*/\*!KO\_cartp(set1%set2%ii%name)\*/\*.

\*Version 1.

\*Combines elements between two sets, creating the cartesian product. There is an option not to pair

\*elements with same index numbers.

\*Sets SET1 and SET2 are matrices of same or different size with >1 rows.

\*Their rows are the elements in the sets.

\*Result NAME is the matrix each row of which is a unique pair of elements, the 1st is from SET1,

\*the 2nd is from SET2. Cartesian product is all possible paired combinations.

\*Columns of NAME are the concatenation of columns of the input matrices.

\*Argument II is scalar. If it is nonpositive then usual, complete cartesian product is returned.

\*If II is positive then pairs representing elements having identical index numbers are excluded from the result.

\*For example, if SET1 is A,B,C and SET2 is 1,2,3,4, then pairs A1, B2 and C3 will be omitted.

EXAMPLE.

matrix.

compute x= {1;2;3}.

compute y= {1;2;3;4;5}.

print x.

print y.

!KO\_cartp(x%y%0%prod1).

print prod1.

!KO\_cartp(y%x%0%prod2).

print prod2.

!KO\_cartp(x%y%1%prod3).

print prod3.

end matrix.

### CARTESIAN PRODUCT BETWEEN N SETS [!KO\_ncartp]

\*/\*!KO\_ncartp(setlist%name)\*/\*.

\*Version 1.

\*Combines elements between two or more sets, creating the cartesian product.

\*The sets are matrices of any size, rows of a matrix are considered the elements of that set.

\*SETLIST is the list (via blanks) of the matrix names. The list may be optionally quoted/apostrophed.

\*Returns matrix NAME which rows are the elements of the catresian product of those matrices, and columns

\*are the concatenated columns of the matrices.

EXAMPLE.

matrix.

compute x= {'a';'b';'c'}.

compute y= {'1';'2';'3';'4'}.

compute z= {'+';'-'}.

!KO\_ncartp(x y z%prod).

print prod /format= a4.

end matrix.

### COMBINATIONS BY K ELEMENTS [!KO\_combk]

\*/\*!KO\_combk(set%k%name)\*/\*.

\*Version 1.

\*Forms combinations by k elements from a set of n elements.

\*Set SET is a column or matrix with n rows: a row is an element in the set.

\*K (integer scalar from 2 to n) - the needed number of elements in a combination.

\*Result NAME - matrix where each row is a combination of elements.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### ALL COMBINATIONS BY K=2,3... ELEMENTS [!KO\_allcomb]

\*/\*!KO\_allcomb(set%maxk%prefix)\*/\*.

\*Version 1.

\*Forms, from a set of n elements, combinations by k=2, k=3, ..., k=MAXK elements.

\*Returns all these combinations respectively as matrices PREFIX2, PREFIX3, etc.

\*I.e. this function is like \*/\*!KO\_combk\*/\*, but it returns at once all combinations with different k.

\*Set SET is a column or matrix with n rows: a row is an element in the set.

\*MAXK - maximal number of elements in a combination. That must be integer number (not name or expression)

\*from 2 to n. Optionally it can be taken in quotes or apostrophes.

\*PREFIX - prefix (optionally in quotes or apostrophes) in the names of the output matrices, the sets of

\*combinations.

\*In the returned matrices each row is a combination of elements.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Tip: if you want to set MAXK=n but number n is not known to you in advance, specify MAXK as a number

\*knowingly exceeding possible n. Matrices above PREFIXn won't be able to create and no error will occur.

EXAMPLE.

set mxloops 100000.

matrix.

compute set= {'a';'b';'c';'d';'e'}.

print set /format a2.

!KO\_allcomb(set%4%'name').

print name2 /format= a2.

print name3 /format= a2.

print name4 /format= a2.

end matrix.

### COMBINATIONS BY K ELEMENTS FROM DIFFERENT SETS [!KO\_dscombk]

\*/\*!KO\_dscombk(sets%last%k%name)\*/\*.

\*Version 1.

\*Forms combinations by k elements belonging to different sets.

\*Sets SETS is a column or matrix with n rows: a row is an element in a set.

\*The sets must be gathered stacked in SETS: first go all elements (rows) of one set,

\*below it all elements of a second set, and so on. There may be any number of sets and the number

\*of elements in them may be different.

\*LAST - vector (row or column) lengthed as the number of sets in SETS and in correspondence to them,

\*containing indices of last rows in each set. For example, {3,6,11} means: the are three sets in SETS,

\*with elemens occupying rows 1:3, 4:6, 7:11. 11 (last element of the last set) must equal the number

\*of rows in SETS.

\*K (integer scalar from 2 to nc, where nc is the number of sets, length of vector LAST) - number of

\*elements in a combination.

\*Result NAME - matrix where each row is a combination of k elements from different sets, no more than

\*one element from a set. If K = number of sets then a combination is "by one element from all sets".

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### ALL COMBINATIONS BY K=2,3... ELEMENTS FROM DIFFERENT SETS [!KO\_dsallcomb]

\*/\*!KO\_dsallcomb(sets%last%maxk%prefix)\*/\*.

\*Version 1.

\*Forms, from several sets, combinations by k=2, k=3, ..., k=MAXK elements belonging to different sets.

\*Returns all these combinations respectively as matrices PREFIX2, PREFIX3, etc.

\*I.e. this function is like \*/\*!KO\_dscombk\*/\*, but it returns at once all combinations with different k.

\*Sets SETS is a column or matrix with n rows: a row is an element in a set.

\*The sets must be gathered stacked in SETS: first go all elements (rows) of one set,

\*below it all elements of a second set, and so on. There may be any number of sets and the number

\*of elements in them may be different.

\*LAST - vector (row or column) lengthed as the number of sets in SETS and in correspondence to them,

\*containing indices of last rows in each set. For example, {3,6,11} means: the are three sets in SETS,

\*with elemens occupying rows 1:3, 4:6, 7:11. 11 (last element of the last set) must equal the number

\*of rows in SETS.

\*MAXK - maximal number of elements in a combination. That must be integer number (not name or expression)

\*from 2 to ns, where ns is the number of sets, length of vector LAST. Optionally it can be taken in

\*quotes or apostrophes.

\*PREFIX - prefix (optionally in quotes or apostrophes) in the names of the output matrices, the sets of

\*combinations.

\*In the returned matrices each row is a combination of k elements from different sets, no more than

\*one element from a set.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Tip: if you want to set MAXK=ns but number ns is not known to you in advance, specify MAXK as a number

\*knowingly exceeding possible ns. Matrices above PREFIXns won't be able to create and no error will occur.

EXAMPLE.

set mxloops 100000.

matrix.

compute sets= {'a1';'a2';'b1';'b2';'b3';'c1'}.

print sets /format= a2.

compute last= {2;5;6}.

!KO\_dsallcomb(sets%last%'3'%'name').

print name2 /format= a2.

print name3 /format= a2.

end matrix.

### HORIZONTAL ARITHMETIC OPERATIONS IN COMBINATIONS OF VARIABLES [!KO\_comboper]

/\*!KO\_comboper(vars%last%maxk%return%oper%weights%alpha%prefix%prefix\_res)\*/\*.

\*Version 1.

\*Columns of matrix VARS are quantitative variables (minimum 2 variables).

\*The function considers all combinations of variables of K variables in a combination; K runs over

\*values from 2 to MAXK. In each variable combination, quantity OPER is computed horizontally,

\*i.e., within each case. Which is the result.

\*Argument OPER - arithmetic action that needs to be done in each combination of variables;

\*capitalized keyword (optionally in quotes/apostrophes):

\*"SUM" - sum, operation of adding variables: V(1)+V(2)+...+V(K)

\*"DIF" - difference, operation of subtracting variables V(1)-V(2)-...-V(K)

\*"PROD" - product, operation of multiplying variables V(1)\*V(2)\*...\*V(K)

\*"QUOT" - quotient, operation of dividing variables V(1)/V(2)/.../V(K) (needs nonzero data)

\*"WSUM" - weighted sum: w(1)\*V(1)+w(2)\*V(2)+...+w(K)\*V(K) (weights w - argument WEIGHTS)

\*"WDIF" - weighted difference: w(1)\*V(1)-w(2)\*V(2)-...-w(K)\*V(K) (weights w - argument WEIGHTS)

\*"EWMA" - terminal (K-th) value of exponentially weighted moving average: S(K);

\* S(K)=(1-a)\*S(K-1)+a\*V(K), where S(K-1)=(1-a)\*S(K-2)+a\*V(K-1), where S(K-2)=...; and S(1)=V(1)

\* (exponential weight a - argument ALPHA).

\*Argument WEIGHTS - weights for WSUM and WDIF operations. Either row vector lengthed as the number

\*of columns in VARS, or matrix sized as VARS; the second instance means that for different cases

\*(VARS rows) weights for variables are set different. Other actions - not WSUM or WDIF - don't use

\*the argument: specify any value, e.g. 0.

\*Argument ALPHA - exponential weight for EWMA operation, value between 0 and 1. It is either scalar

\*or column vector lengthed as the number of rows in VARS; the second instance means that for

\*different cases alpha weight is set different. Other actions - not EWMA - don't use the argument:

\*specify any value, e.g. 0.

\*Arguments LAST, MAXK, as well as PREFIX are the same as in function /\*!KO\_dsallcomb\*/, on which this

\*function is built.

\*Argument LAST - parameter to break the variables in sets. Only variables belonging to different

\*sets will be combining together. It is a vector (row or column) lengthed as the number of sets

\*and in corresponding to them, containing indices (ordinal numbers) of the last variables in the sets.

\*For example, {3,6,11} means: there are three sets in VARS, the variable groups consisting of columns

\*1:3, 4:6, 7:11. 11 (the last value in LAST) must be equal to the number of columns in VARS.

\*If you don't have groups of variables, i.e., each variable is its own set, specify LAST

\*as 1:ncol(VARS), i.e. the vector of numbers 1,2,...,number\_of\_variables.

\*Argument MAXK - maximal needed to you value of K, the number of variables in a combination. That must

\*be integer number (not name or expression) from 2 to ns, where ns is the number of sets, length

\*of vector LAST.

\*Argument RETURN - for which values of K you want to obtain results. If for all K from 2 to

\*min(MAXK,ns), i.e. all results, specify RETURN as keyword ALL (capitalized).

\*Otherwise, request a list of K you need, spaced integers, for example: 2 3 5. Then the function

\*will compute results for those K only.

\*Results: PREFIX, PREFIX\_RES. Specify in place of these arguments 2 prefixes for the names of

\*the matrices to return, the names consisting of these prefixes and number K as the ending.

\*For each K, a pair of arrays will be output:

\*- Indices of variables forming combinations - matrix with prefix PREFIX.

\* Rows in PREFIX are various combinations of VARS variables, elements in a row are indices of

\* variables in the combination.

\*- Results of operation OPER - matrix with prefix PREFIX\_RES. Rows in it correspond to the data cases

\* (rows of VARS), and columns correspond to the variable combinations (rows of PREFIX).

\*Specifications of arguments MAXK, RETURN, PREFIX, PREFIX\_RES can be optionally surrounded with

\*quotes/apostrophes.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Tip: if you want to set MAXK=ns but number ns is not known to you in advance, specify MAXK as a number

\*knowingly exceeding possible ns. Matrices above PREFIXns won't be able to create and no error will occur.

EXAMPLE. Multiplication of quantitative variables: obtaining their “interactions” up to 3-way.

set mxloops 1E9.

matrix.

get vars /variables= v1 to v10.

!KO\_comboper(vars%1:ncol(vars)%3%ALL%PROD%0%0%Comb%Inter).

print vars.

print Comb2.

print Inter2.

print Comb3.

print Inter3.

save {Comb2,Comb3} /out= \*.

end matrix.

EXAMPLE. Weighted addition of quantitative variables from different sets of them.

set mxloops 1E9.

matrix.

get vars /variables= v1 to v10.

compute weights= {2,.5,1.3,1.8,3,.6,1,1.1,.5,0} /\*10 weights for the 10 variables

!KO\_comboper(vars%{3,7,8,10}%4%3 4%WSUM%weights%0%Comb%wSum). /\*The sets are:

/\*v1-v3, v4-v7, v8, v9-v10 (positition of the last variable in each set

/\*was specified as LAST argument); combining will be only between sets;

/\*Combinations up to K=4 variables are requested, however, results for K=2

/\*will not be produced, because RETURN is "3 4"

print vars.

print Comb3.

print wSum3.

print Comb4.

print wSum4.

save {wSum3,wSum4} /out= \*.

end matrix.

### QUANTITIES ON THE BASIS OF HORIZONTAL SUMS OR MEANS IN COMBINATIONS OF VARIABLES [!KO\_turflike]

\*/\*!KO\_turflike(vars%last%maxk%return%tosum%tocnt%cond%prefix%prefix\_sum%prefix\_cnt)\*/\*.

\*Version 1.

\*Columns of matrix VARS are quantitative variables (minimum 2 variables).

\*The function considers all combinations of variables of K variables in a combination; K runs over

\*values from 2 to MAXK. In each variable combination, statistics hs (see below) are computed

\*horizontally, i.e., within each case. Then the function (1) sums hs across cases, returning

\*result PREFIX\_SUM; (2) counts cases where hs satisfies condition COND, returning result PREFIX\_CNT.

\*So, for each combination of variables, columns from VARS, you'll get value "column\_sum(hs)"

\*and "column\_count(hs COND = True)".

\*Arguments TOSUM and TOCNT. Here you need to specify a statistic hs, in a form of capitalized keyword.

\*May request the same or different statistics in these two arguments. The choice is among the following

\*statistics, to compute within each case (row of VARS):

\*"SUM" - sum of data values

\*"MEAN" - mean of data values

\*"SUMVLD" - sum of valid data values

\*"MEANVLD" - mean of valid data values

\*"NVLD" - count of valid data values

\*In the last three statistics, value: -999 is treated as the nonvalid value.

\*If there is no valid data in a case, MEANVLD of the case evaluates as 0.

\*Argument COND - the condition for counting number of cases in which the TOCNT-statistic satisfies it.

\*Consists of one operator and one operand. Operators can be: GT (or >), LT (or <), EQ (or =),

\*GE, LE, NE. (>=, <=, <> are not allowed to use in this function.) The operand can be a

\*number or some matrix name or an expression; the negative sign may precede the operand.

\*Examples of specifying COND: GT 0, EQ -4.12, LE v3, LT x(:,1).

\*Operand with a name. The name can be of a scalar (number), a column vector of length =nrow(VARS),

\*a row vector of length >=min(MAXK,ns) [see below of ns], or a matrix sized [nrow(VARS),>=min(MAXK,ns)].

\*Column vector lets you give different values of the operand for different data cases (rows of VARS).

\*Row vector lets you give different values of the operand for different K (number of variables in

\*combination); K-th element in the row vector will correspond to the given K. Operand in the form of a

\*matrix (of the size mentioned above) lets you give different values of the operand both for different

\*cases and for different K.

\*Arguments LAST, MAXK, as well as PREFIX are the same as in function /\*!KO\_dsallcomb\*/, on which this

\*function is built.

\*Argument LAST - parameter to break the variables in sets. Only variables belonging to different

\*sets will be combining together. It is a vector (row or column) lengthed as the number of sets

\*and in corresponding to them, containing indices (ordinal numbers) of the last variables in the sets.

\*For example, {3,6,11} means: there are three sets in VARS, the variable groups consisting of columns

\*1:3, 4:6, 7:11. 11 (the last value in LAST) must be equal to the number of columns in VARS.

\*If you don't have groups of variables, i.e., each variable is its own set, specify LAST

\*as 1:ncol(VARS), i.e. the vector of numbers 1,2,...,number\_of\_variables.

\*Argument MAXK - maximal needed to you value of K, the number of variables in a combination. That must

\*be integer number (not name or expression) from 2 to ns, where ns is the number of sets, length

\*of vector LAST.

\*Argument RETURN - for which values of K you want to obtain results. If for all K from 2 to

\*min(MAXK,ns), i.e. all results, specify RETURN as keyword ALL (capitalized).

\*Otherwise, request a list of K you need, spaced integers, for example: 2 3 5. Then the function

\*will compute results for those K only.

\*Results: PREFIX, PREFIX\_SUM, PREFIX\_CNT. Specify in place of these arguments 3 prefixes for

\*the names of the matrices to return, the names consisting of these prefixes and number K as the ending.

\*For each K, a series of 3 arrays will be output:

\*- Indices of variables forming combinations - matrix with prefix PREFIX.

\* Rows in PREFIX are various combinations of VARS variables, elements in a row are indices of

\* variables in the combination.

\*- Results of summing TOSUM - column vector with prefix PREFIX\_SUM. Rows in it correspond to the rows

\* in PREFIX.

\*- Results of counting TOCNT - column vector with prefix PREFIX\_CNT. Rows in it correspond to the rows

\* in PREFIX.

\*If you don't want to have result PREFIX\_SUM or PREFIX\_CNT, put digit 0 in place of that agrument.

\*Specifications of arguments TOSUM, TOCNT, COND, MAXK, RETURN, PREFIX, PREFIX\_SUM, PREFIX\_CNT can be

\*optionally surrounded with quotes/apostrophes.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*Tip: if you want to set MAXK=ns but number ns is not known to you in advance, specify MAXK as a number

\*knowingly exceeding possible ns. Matrices above PREFIXns won't be able to create and no error will occur.

\*Particular case: if VARS are binary variables (0 vs 1), TOSUM=TOCNT="SUM", and condition COND="GT 0",

\*then the function in fact yields quantities that are computed in the so-called TURF analysis.

\*PREFIX\_SUM will give counts of "Frequency", and PREFIX\_CNT will give counts "Reach".

EXAMPLE. TURF analysis.

set mxloops 1E9.

matrix.

get vars /variables= v1 to v10. /\*Ten binary variables (products)

!KO\_turflike(vars%1:nrow(vars)%10%ALL%SUM%SUM%gt 0%Comb%Freq%Reach).

/\*All variables can be combined with each other;

/\*K runs from 2 to 10; and results for every K is requested;

/\*Statistic “within-cases SUM” specified for the TOSUM arg, that

/\*will yield TURF “Frequency” counts (output with prefix Freq);

/\*Statistic “within-cases SUM” specified for the TOCNT arg with

/\*the condition COND to count if the sum is above zero, and that

/\*will yield TURF “Reach” (output with prefix Reach)

\*The results are: variable combinations as matrices Comb2 to Comb10;

\*“Reach” values as vectors Reach2 to Reach10;

\*“Frequency” values as vectors Freq2 to Freq10.

\*Organize the results:

\*For each K, concatenate the results and sort by “Reach”,

\*and within it - by “Frequency”;

!KO\_hiesort({Reach2,Freq2,Comb2}%0%K2%ind).

!KO\_hiesort({Reach3,Freq3,Comb3}%0%K3%ind).

!KO\_hiesort({Reach4,Freq4,Comb4}%0%K4%ind).

!KO\_hiesort({Reach5,Freq5,Comb5}%0%K5%ind).

!KO\_hiesort({Reach6,Freq6,Comb6}%0%K6%ind).

!KO\_hiesort({Reach7,Freq7,Comb7}%0%K7%ind).

!KO\_hiesort({Reach8,Freq8,Comb8}%0%K8%ind).

!KO\_hiesort({Reach9,Freq9,Comb9}%0%K9%ind).

!KO\_hiesort({Reach10,Freq10,Comb10}%0%K10%ind).

\*Print.

print /title '------ As Counts ------'.

print K2 /format= f6 /title 'K=2' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K3 /format= f6 /title 'K=3' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K4 /format= f6 /title 'K=4' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K5 /format= f6 /title 'K=5' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K6 /format= f6 /title 'K=6' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K7 /format= f6 /title 'K=7' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K8 /format= f6 /title 'K=8' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K9 /format= f6 /title 'K=9' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

print K10 /format= f6 /title 'K=10' /clabels 'Reach' 'Frequ' 'VarComb' '...'.

\*Also, transform counts to percents (“Reach” as % of cases,

\*“Frequency” as % of total responses):.

compute K2(:,1)= K2(:,1)/nrow(vars)\*100.

compute K3(:,1)= K3(:,1)/nrow(vars)\*100.

compute K4(:,1)= K4(:,1)/nrow(vars)\*100.

compute K5(:,1)= K5(:,1)/nrow(vars)\*100.

compute K6(:,1)= K6(:,1)/nrow(vars)\*100.

compute K7(:,1)= K7(:,1)/nrow(vars)\*100.

compute K8(:,1)= K8(:,1)/nrow(vars)\*100.

compute K9(:,1)= K9(:,1)/nrow(vars)\*100.

compute K10(:,1)= K10(:,1)/nrow(vars)\*100.

compute totalcnt= msum(vars).

compute K2(:,2)= K2(:,2)/totalcnt\*100.

compute K3(:,2)= K3(:,2)/totalcnt\*100.

compute K4(:,2)= K4(:,2)/totalcnt\*100.

compute K5(:,2)= K5(:,2)/totalcnt\*100.

compute K6(:,2)= K6(:,2)/totalcnt\*100.

compute K7(:,2)= K7(:,2)/totalcnt\*100.

compute K8(:,2)= K8(:,2)/totalcnt\*100.

compute K9(:,2)= K9(:,2)/totalcnt\*100.

compute K10(:,2)= K10(:,2)/totalcnt\*100.

\*Print.

print /title '------ As Percents ------'.

print K2 /format= f6.1 /title 'K=2' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K3 /format= f6.1 /title 'K=3' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K4 /format= f6.1 /title 'K=4' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K5 /format= f6.1 /title 'K=5' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K6 /format= f6.1 /title 'K=6' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K7 /format= f6.1 /title 'K=7' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K8 /format= f6.1 /title 'K=8' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K9 /format= f6.1 /title 'K=9' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

print K10 /format= f6.1 /title 'K=10' /clabels 'Reach%' 'Frequ%' 'VarComb' '...'.

end matrix.

EXAMPLE. Somewhat sophisticated example.

set mxloops 1E9.

set seed 74194798.

matrix.

\*Simulate some data with value -999 designating missing data.

compute n= 15.

compute p= 8.

compute vars= rnd(uniform(n,p)\*8).

compute miss= rnd(uniform(n,p)\*.6).

compute vars= vars&\*(not miss)+miss&\*(-999).

print vars. /\*The data, 8 variables (columns), 15 cases (rows)

\*Let our variables be split in 5 sets; variables will combine only between sets.

compute last= {1,4,6,7,8}. /\*The sets are:

/\*var 1, vars 2-4, vars 5-6, var 7, var 8

\*Let us have some operand condition variable X.

compute x= rnd(uniform(n,1)\*10).

print x.

\*Run the function.

!KO\_turflike(vars%last%8%3 4 5%MEANVLD%NVLD%ge x%K%SM%CSC).

/\*Variables belonging only to different sets will be combined together;

/\*Combinations from K=2 to K=8 variables are requested, however the number of

/\*sets 5 is less, so min(8,5)=5 is the upper value of K really possible;

/\*Moreover, results for K=2 won't be produced because instead of "ALL", just

/\*K= 3, 4, and 5 were demanded as RETURN argument;

/\*(The TOSUM argument) For each variable combination, mean of valid values

/\*will be computed in each case, and then summed across cases;

/\*the results are stored as vectors with prefix SM ("sum of means")

/\*(The TOCNT argument) For each variable combination, number of valid values

/\*will be counted in each case, and then the number of cases where this

/\*count is not below the threshold contained in variable X (COND argument)

/\*is counted; the results are stored as the vectors with prefix CSC

/\*("count of satisfying counts");

/\*The combinations themselves (variables' indices) are stored

/\*in matrices prefixed K.

\*Print results.

print {SM3,CSC3,K3} /format f8.4 /title 'K=3' /clabels 'TOSUM' 'TOCNT' 'Comb' '...'.

print {SM4,CSC4,K4} /format f8.4 /title 'K=4' /clabels 'TOSUM' 'TOCNT' 'Comb' '...'.

print {SM5,CSC5,K5} /format f8.4 /title 'K=5' /clabels 'TOSUM' 'TOCNT' 'Comb' '...'.

end matrix.

### APRIORI ALGORITHM [!KO\_apriori]

\*/\*!KO\_apriori(mrc%minsup%name1%name2)\*/\*.

\*Version 1.

\*In data like multiple response, Apriori algorithm counts item sets (individual items and

\*combinations of those) that are at least frequent as the threshold MINSUP ("minimal support",

\*positive integer). The returned count is called "support". Support is the subset frequency,

\*that is, the number of cases where the item or the item combination is found as a subset.

\*For example, in these four cases: {1,3}, {1,2,5}, {1,3,5}, {1,2,3,4} the support for

\*combination {1,3} is 3, and the support for item {5} is 2. The support for {1,3,5} is 1.

\*MRC - data in the form of categorical multiple response set, one column minimum.

\*Each case (row) is a respondent or a basket, and the values in it are codes of responses

\*or items, listed. Value 0 is reserved to signify an empty (not used) cell. Complete zero rows

\*or columns are allowed. If a code encounters more than once in a case, it is recognized just once.

\*Results are the frequent item sets (NAME1) and their corresponding support (NAME2).

\*This function is not a fast realization of Apriori algorithm, so you would not

\*use it with many thousands or millions of cases.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E6.

matrix.

compute data=

{1,2,0,0;

2,4,0,0;

2,3,0,0;

1,2,4,0;

1,3,0,0;

2,3,0,0;

1,3,0,0;

1,2,3,5;

1,2,3,0}.

print data.

!KO\_apriori(data%3%name1%name2).

print {name1,name2} /title 'Results for minsupport=3'

/clabels 'ItemSet' ' ' 'Support'.

end matrix.

# COMBINATORIAL OPTIMIZATION FUNCTIONS

### MATCHING BY HUNGARIAN ALGORITHM [!KO\_hungar]

\*/\*!KO\_hungar(cost%name)\*/\*.

\*Version 1.

\*Hungarian or Kuhn-Munkres algorithm matches elements one-to-one between two sets optimally so

\*that the summed cost of pairs (or summed dissimilarity in pairs) become minimal.

\*In graph terminology, it constructs optimal matching from a bipartite weighted or unweighted graph.

\*(To note that in case of unweighted graph you can use also Hopcroft-Karp /\*!KO\_hopckarp\*/ algorithm

\*to solve the same problem.)

\*The function accepts as input a n x m matrix of costs COST with nonnegative elements. The number of

\*rows must be not less than the number of columns. The rows are one set of elements and the columns are the

\*other set of elements. Values in matrix are taken as "cost" of a pair row-column or as "dissimilarity"

\*between a row and a column. The algorithm finds for each column one matching row.

\*Result NAME is 3-column matrix with number of rows as the number of columns in COST; the rows there are

\*the matched pairs. The 2nd column there is the column number of the input matrix COST, the 1st column is

\*the row number in COST, and the 3rd column is the value in COST on their intersection. The algorithm

\*minimizes the sum in the 3rd column.

\*If your input matrix represent not costs (dissimilarities) but gains (similarities), transform it to

\*costs (dissimilarities) by sign-negating and then adding a constant to matrix to make all the values

\*nonnegative.

\*The algorithm speed is higher when the number of distinct values in COST is not great, so there is

\*the reason to consider beforehand a discretization of the "cost" values.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1000000.

matrix.

compute cost= rnd(uniform(10,8)\*100).

print cost /title 'Cost matrix'.

!KO\_hungar(cost%pairs).

print pairs /title 'Matching'.

print csum(pairs(:,3)) /title 'Overall cost (minimized)'.

end matrix.

### MATCHING BY HOPCROFT-KARP ALGORITHM [!KO\_hopckarp]

\*/\*!KO\_hopckarp(adj%name)\*/\*.

\*Version 1.

\*Hopcroft-Karp algorithm matches elements one-to-one between two sets having links between them

\*so as to maximize the number of pairs, i.e., the number of links accounted for.

\*In graph terminology, it constructs maximal matching from a bipartite unweighted graph.

\*(To note that you can also use Hungarian /\*!KO\_hungar\*/ algorithm to solve the same problem.)

\*Input matrix ADJ is an "adjacency list" sized n x m, where n and m are the numbers of elements

\*in the two sets. Each row in ADJ is an element from the "n" set, and it contains the list of element

\*numbers from the "m" set, going left-to-right, rammed. For example, let there be sets "n"

\*(1,2,...,n) and "m" (1,2,...,m=7), where, particularly, the 2nd element from "n" has links with the

\*elements 1,4, and 5 from the "m" set. Then, the 2nd row of ADJ should look like: {1,4,5,0,0,0,0}.

\*The order of the numbers plays no role: {4,5,1,0,0,0,0}, for instance, is also correct. The number of

\*columns in ADJ must be m, therefore empty trails are padded with zeros up to column 7. Adjacency list

\*corresponds to a binary matrix sized n x m, in which units flag the presence of a link; and it is

\*another mode to represent the matrix. If your input data are the binary matrix, you can turn it into

\*the adjacency list by function /\*!KO\_ram\*/.

\*Empty (zero filled) rows in ADJ are allowed - they correspond to isolated, having no links,

\*elements of "n", - but you might want first to remove them, in the speed performance view.

\*ADJ must bear at least one nonzero element.

\*If n and m differ strongly, it is advantageous, from the speed performance point of view, to have ADJ

\*wide (n<m) rather than high (n>m).

\*Result NAME is 2-column matrix giving a maximal matching; the 1st column is the elements from

\*the "n" set, 2nd column is the elements from the "m" set; each row is a done pairing.

\*Number of rows in NAME is, therefore, the size of the matching.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE. Hopcroft–Karp and Hungarian algorithms on the same data.

set mxloops 1E6.

matrix.

compute n= 10.

compute m= 8.

compute mat= rnd(uniform(n,m)\*0.8). /\*Let this random binary NxM matrix represent

/\*a bipartite graph

print mat.

!KO\_ram(mat%IND%0%0%adj%name). /\*Turn it into corresponding NxM adjacency list

/\*!KO\_ram(mat%IND%0%1%adj%name). /\*[or might additionally randomize the order]

print adj.

!KO\_hopckarp(adj%match). /\*Perform matching by Hopcroft-Karp

print match. /\*The optimal (maximal) matching

print nrow(match). /\*Its size

\*-----.

!KO\_hungar(not mat%match2). /\*Now use Hungarian for the same task;

/\*Hungarian uses MAT as the input, and note that we reversed

/\*it (1->0, 0->1), to make Hungarian to maximize gain rather

/\*than minimize cost; also mind that Hungarian needs n>=m to

/\*process a matrix correctly

print match2. /\*The optimal (maximal) matching

/\*If the maximal matching is not unique for the input data,

/\*Hungarian and Hopcroft-Karp may return different matchings;

print csum(match2(:,3)=0). /\*but the size of the maximal matching is always the same

/\*(cost "0" in Hungarian corresponded to link "1" in MAT)

end matrix.

Speed considerations. If data are very sparse (i.e., there are quite few links between the two sets of elements), Hopcroft–Karp is faster than Hungarian algorithm. In other cases it is usually slower. And its speed drops when n>>m.

### SIMPLE GREEDY MATCHING [!KO\_greedm]

\*/\*!KO\_greedm(mat%cg%name)\*/\*.

\*Version 1.

\*Simple greedy matching is a stepwise algorithm which pairs elements between two sets so that

\*to minimize (or maximize) the sum of values within pairs. This is the same task that Hungarian

\*matching (function /\*!KO\_hungar\*/) accomplishes, but /\*!KO\_greedm\*/ is faster and does not guarantee

\*the global optimum in the sense of the final sum. The algorithm finds the least (or the largest - as

\*requested) value in the matrix and records its row and column a pair (match), then cuts off the two

\*from the matrix, in the remnant matrix it then again finds the least value, pairs the row and the

\*column, deletes them, and so on.

\*Input:

\*MAT - matrix sized at least 2x2, with any numbers. If the number of columns much exceeds the number

\*of rows it might have sense to transpose the matrix first, which could give win in speed.

\*CG - digit (not name or expression) 0, 1, or 2 (the digit may optionally be put in quotes or

\*apostrophes). If "0", the function will seek minimal values, i.e. will treat MAT as cost which

\*overall must be minimized at matching. If "1", the function will seek maximal values, i.e. will

\*treat MAT as gain which overall must be maximized at matching. If "2", then MAT is a sparse

\*nonnegative gain matrix: with positive values as the gains and zero elements prohibiting to

\*do the pairing; at least one value must be greater than 0.

\*Result NAME: 3-column matrix where rows are matched pairs row-column chosen by the algorithm:

\*the row number in MAT is written in the 1st NAME's column, the column number in MAT is written in

\*the 2nd NAME's column, and the value on their intersection in MAT is in the 3rd NAME's column.

\*The sum in this 3rd column is what the function attempts to minimize (maximize).

\*In case CG="2", some last rows in NAME can be empty, because sparse MAT may not allow to pair

\*all the rows and columns.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1000000.

matrix.

compute gain= rnd(uniform(10,8)\*30).

print gain /title 'Gain matrix'.

!KO\_greedm(gain%'1'%pairs).

print pairs /title 'Matching'.

print csum(pairs(:,3)) /title 'Overall gain (attempted to maximize)'.

end matrix.

### SIMPLE GREEDY MATCHING (DISCRETE DATA) [!KO\_greedm2]

\*/\*!KO\_greedm2(mat%cg%name)\*/\*.

\*Version 1.

\*This function gives the same result as /\*!KO\_greedm\*/, but it is faster in the situation when the

\*number of distinct values in MAT is small relative the size of MAT. For example, when MAT is

\*integers from a considerably limited range. And conversely, if there are many distinct values in MAT,

\*such as, of instance, in case of continuous data, then /\*!KO\_greedm2\*/ will be slower

\*than /\*!KO\_greedm\*/.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### FLOYD-WARSHALL ALGORITHM: SHORTEST PATHS / EASIEST PASSES [!KO\_flowar]

\*/\*!KO\_flowar(cost%task%path%name1%name2)\*/.

\*Version 1.

\*Takes square matrix COST with nonnegative elements and zero diagonal;

\*the matrix may be symmetric or asymmetric. Values of COST can be understood as weights in a graph

\*or links in a network, and rows/columns of the matrix are the graph's vertices or net's nodes.

\*TASK (keyword, capitalized, may quote or put in apostrophes) is the algorithm's goal:

\*"SHWAY" - find, from each point to each point, the shortest way (Shortest path aka Shortest

\*way mode of the algo). Values in COST are taken for distances between the points:

\*COST(i,j) is the distance from i into j.

\*From every point i to every point j, the shortest way will be found. Result NAME1 will

\*contain the summative length of that shortest path from each point-row i to each point-column j.

\*"EPASS" - find, from each point to each point, the path of easiest passes (Optimal routing

\*aka Easiest passes mode of the algo). Values in COST are taken for difficulties to transit

\*a pass from a point to a point: COST(i,j) is the difficulty from i into j.

\*From every point i to every point j, the easiest way will be found:

\*all the passes of which it will consist, are minimally difficult. Result NAME1 will contain

\*the difficulty of the most difficult pass on that easiest way from each point-row i to each

\*point-column j.

\*The found (i.e., the optimal) paths, the trajectories, will be written "wrapped" in matrix NAME2.

\*Element (i,j) in NAME2 is the number of the point which follows the point i in every found path

\*that leads to point j as the endpoint of the path. See EXAMPLE, showing how to extract the whole

\*trajectory of the path from some starting point U to some ending point V=j.

\*Whether to register a trajectory or not - depends on argument PATH. Argument PATH is a digit 1 or 0

\*(not name or expression, you may take the digit in quotes or apostrophes). If 0, trajectories won't

\*be written down and matrix NAME2 will come out to be zero scalar.

\*Missing values in COST and impossible path. If in the input matrix there should be no links between

\*some points - for example, the pass from point i to point j does not exist, then set the

\*cell (i,j) to number X not less than the sum of all existing values in the matrix, i.e. X>=msum(COST).

\*So then, if in some cell (U,V) of NAME1 you see number X, that will signal that a path from starting

\*point U to ending point V is impossible.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E6.

matrix.

compute dist= rnd(uniform(8,8)\*50).

call setdiag(dist,0).

print dist /title 'Asymmetric Distance matrix'.

!KO\_flowar(dist%SHWAY%1%length%next). /\*Shortest ways will be searched for

print length /title 'Shortest path length from point-row to point-column'.

print next /title 'Paths are written here'.

\*Show extraction of a path.

compute u= 2. /\*Select two

compute v= 5. /\*arbitrary points

print u /title 'Extract the path starting from point U='.

print v /title 'and ending at point V=' /space= 0.

print length(u,v) /title '(the path length is:)' /space= 0.

compute path= u.

loop if u<>v.

-compute u= next(u,v).

-compute path= {path;u}.

end loop.

print path /title 'The path (point numbers):'.

end matrix.

### FLOYD-WARSHALL ALGORITHM (SYMMETRIC MATRIX) [!KO\_sflowar]

\*/\*!KO\_sflowar(cost%task%name)\*/.

\*Version 1.

\*This function is identical to /\*!KO\_flowar\*/, but it is only for symmetric input matrix COST.

\*This function is somewhat faster than /\*!KO\_flowar\*/.

\*Argument PATH and result NAME2 are not provided.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

### DIJKSTRA ALGORITHM: SHORTEST PATH / EASIEST PASS [!KO\_dijkstra]

\*/\*!KO\_dijkstra(cost%start%stop%task%path%name1%name2)\*/.

\*Version 1.

\*Takes square matrix COST with nonnegative elements; the matrix may be symmetric or asymmetric.

\*Values of COST can be understood as weights in a graph or links in a network, and rows/columns of

\*the matrix are the graph's vertices or net's nodes. Diagonal elements in COST are ignored by the

\*function (they are considered as 0).

\*START - scalar; the point (vertex) number from where to find paths to the other points.

\*TASK (keyword, capitalized, may quote or put in apostrophes) is the algorithm's goal:

\*"SHWAY" - find, from point START to each point, the shortest way (Shortest path aka Shortest

\*way mode of the algo). Values in COST are taken for distances between the points:

\*COST(i,j) is the distance from i into j.

\*From point START to every point j, the shortest way will be found. Result NAME1 will

\*contain the summative length of that shortest path from point-row START to each point-column j.

\*"EPASS" - find, from point START to each point, the path of easiest passes (Optimal routing

\*aka Easiest passes mode of the algo). Values in COST are taken for difficulties to transit

\*a pass from a point to a point: COST(i,j) is the difficulty from i into j.

\*From point START to every point j, the easiest way will be found:

\*all the passes of which it will consist, are minimally difficult. Result NAME1 will contain

\*the difficulty of the most difficult pass on that easiest way from point-row START to each

\*point-column j.

\*The found (i.e., the optimal) paths, the trajectories, will be written "wrapped" in vector NAME2.

\*The j-th element's value in NAME2 is the number of the point which precedes the point j in every

\*found path that leads to point j as the endpoint of the path. See EXAMPLE, showing how to extract

\*the whole trajectory of the path from the point START to some ending point V=j.

\*Whether to register a trajectory or not - depends on argument PATH. Argument PATH is a digit 1 or 0

\*(not name or expression, you may take the digit in quotes or apostrophes). If 0, trajectories won't

\*be written down and NAME2 will come out to be zero scalar.

\*Missing values in COST and impossible path. If in the input matrix there should be no links between

\*some points - for example, the pass from point i to point j does not exist, then set the

\*cell (i,j) to value: -1. So then, if some V-th element of NAME1 has turned out as -1, that will

\*signal that a path from START to ending point V is impossible.

\*Argument STOP. If you need paths from START to all the points, set STOP on value 0.

\*But if you are interested in the path from START to one specific point only, indicate its number

\*here. Then the algorithm will cease working upon coming to that point; you may save time through

\*this. Result NAME1 will then output as scalar, not vector. STOP must be different from START.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If you need paths from every point to every point, use function /\*!KO\_flowar\*/.

EXAMPLE. Shortest way from one vertex to one another vertex.

set mxloops 1E6.

matrix.

compute dist= rnd(uniform(8,8)\*50).

call setdiag(dist,0).

print dist /title 'Asymmetric Distance matrix'.

!KO\_dijkstra(dist%4%2%SHWAY%1%length%prev). /\*Shortest way from pt 4 to pt 2 will

/\*be searched for

print t(length) /title 'Shortest path length from point 4 to point 2'.

print t(prev) /title 'The path is written here'.

\*Show extraction of the path.

compute v= 2. /\*Indicate the STOP point: it was our specific goal in the function

compute path= 0. /\*(A prime to concatenate)

do if prev(v).

-loop if v.

- compute path= {v;path}.

- compute v= prev(v).

-end loop.

end if.

compute path= path(1:(nrow(path)-1)). /\*(Drop the priming zero)

print path /title 'The path (point numbers):'.

end matrix.

EXAMPLE. Equivalence with Floyd–Warshall algorithm.

set mxloops 1E6.

matrix.

compute dist= rnd(uniform(8,8)\*50).

call setdiag(dist,0).

print dist /title 'Asymmetric Distance matrix'.

compute start= 1. /\*Let this be the start point

!KO\_dijkstra(dist%start%0%SHWAY%1%length%prev). /\*Shortest way from pt 1 to every

/\*other point

print t(length)

/title 'Shortest path lengths from point 1 to to every other point'.

print t(prev) /title 'The paths are written here in the "previous point" mode'.

\*Show extraction of the path to point, say, 5.

compute v= 5. /\*The end point: 5

print v /title 'Extract the path from the start point (1) to point V='.

print length(5) /title '(the path length is:)' /space= 0.

compute path= 0. /\*(A prime to concatenate)

do if prev(v).

-loop if v.

- compute path= {v;path}.

- compute v= prev(v).

-end loop.

end if.

compute path= path(1:(nrow(path)-1)). /\*(Drop the priming zero)

print path /title 'The path (point numbers):'.

\*----.

\*Now perform the same search by means of Floyd-Warshall algorithm.

print /title '---- Now try Floyd-Warshall ----'.

!KO\_flowar(dist%SHWAY%1%lengths%next). /\*Search shortest way from each point

/\*to each point

print t(lengths(start,:))

/title 'Shortest path lengths from point 1 to to every other point'.

/\*The lengths of the optimal routes returned by Floyd-Warshall are

/\*always the same as by Dijkstra, so the algorithms are equivalent

/\*optimizers

print t(next(start,:)) /title 'The paths are written here in the "next point" mode'.

\*Show extraction of a path.

compute u= start. /\*From our start point

compute v= 5. /\*to, say, point 5

print u /title 'Extract the path starting from point U='.

print v /title 'and ending at point V=' /space= 0.

print lengths(u,v) /title '(the path length is:)' /space= 0.

compute path= u.

loop if u<>v.

-compute u= next(u,v).

-compute path= {path;u}.

end loop.

print path /title 'The path (point numbers):'.

/\*The paths themselves (the points of the optimal routes) returned by

/\*Floyd-Warshall and Dijkstra may occasionally differ, because, though

/\*equivalent, these are different algorithms

end matrix.

If your graph is unweighted, the shortest paths from one vertex to the other can be obtained also by function /\*!KO\_bfs\*/.

### PRIM'S MINIMUM SPANNING TREE ALGORITHM [!KO\_prim]

\*/\*!KO\_prim(mat%cg%start%name)\*/\*.

\*Version 1.

\*In a unipartite weighted undirected graph specified by adjacency matrix MAT, finds

\*the minimum (or maximum) spanning tree, starting from vertex START (scalar, the vertex number).

\*MAT - square n x n symmetric matrix. Its diagonal elements are ignored by the function.

\*Argument CG sets the interpretation of MAT. CG is the digit (not name or expression) 0, 1, or 2

\*(may optionally take it in quotes or apostrophes).

\*If CG=0, MAT is understood as a cost or distance matrix between the vertices of a complete

\*weighted graph. Result NAME is the minimum spanning tree.

\*If CG=1, MAT is understood as a gain or similarity matrix between the vertices of a complete

\*weighted graph. Result NAME is the maximum spanning tree. In both instances values in MAT

\*may have any sign.

\*If CG=2, MAT is understood as a sparse nonnegative gain matrix:

\*with positive values as gains and zero elements corresponding to absent edges (incomplete graph).

\*Result NAME is then the maximum spanning tree from vertex start under the condition

\*of incomplete graph. If the incomplete graph is not a connected graph (i.e., not a graph where

\*there exists a path from every vertex to every vertex), then the spanning tree will cover

\*only its part - the subgraph containing the start vertex, - and as a result, not all n-1

\*rows of NAME will be filled (some last will remain empty). Thus Prim's algorithm

\*is usable with CG=2 to diagnose connectedness in a incomplete graph.

\*When a graph is complete or is incomplete but connected, the spanning tree will cover all the graph

\*and the size (sum of weights) of the tree does not depend on the start vertex.

\*Result NAME: has three columns and n-1 rows; the rows correspond to the edges written down to the

\*spanning tree. The first two elements in a row are the vertices i and j, adjacent of the edge,

\*and the third value is the edge weight, i.e. the value (i,j) in MAT. Prim's algorithm

\*minimizes (if MAT is cost matrix) or maximizes (if MAT is gain matrix) the sum in the 3rd column

\*of NAME.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

\*If your MAT is cost matrix and the graph is incomplete (some edges nonexistent), transform it

\*to the positive gain matrix by negating, and then adding a positive constant, and the absent

\*edges to mark with zeros; then use CG=2.

EXAMPLE. Complete graph of costs (or distances).

set mxloops 1E6.

matrix.

compute dist= uniform(8,8)\*50.

compute dist= rnd((dist+t(dist))/2).

call setdiag(dist,0).

print dist /title 'Square symmetric distance matrix'.

!KO\_prim(dist%0%4%mst). /\*Build minimum spanning tree, starting from point 4

print mst /title 'Minimum spanning tree includes:' /clab= 'Point' 'Point' 'Dist'.

print msum(mst(:,3)) /title 'The sum (minimized)'.

end matrix.

EXAMPLE. Is this graph connected?

matrix.

compute g=

{0,1,0,0,1,0,0,1;

1,0,0,0,0,0,1,1;

0,0,0,0,0,1,0,0;

0,0,0,0,0,1,0,0;

1,0,0,0,0,0,1,0;

0,0,1,1,0,0,0,0;

0,1,0,0,1,0,0,0;

1,1,0,0,0,0,0,0}. /\*An incomplete (sparse) graph

/\*(all existing edge weights are equal, 1, in this example)

print g /title 'Square symmetric sparse gain matrix'.

!KO\_prim(g%2%1%mst). /\*Build maximum spanning tree, starting from point 1

print mst /title 'Maximum spanning tree:' /clab= 'Vert' 'Vert' 'Gain'.

/\*The algo exited before filling in all 7 rows of mst: G is disconnected

end matrix.

# SEQUENCES

### LEVENSHTEIN DISTANCE (WAGNER-FISCHER ALGORITHM) [!KO\_levenshtein]

\*/\*!KO\_levenshtein(s1%s2%editcosts%name)\*/\*.

\*Version 1.

\*Levenshtein distance between two sequences.

\*This is the amount of edit operations of insertion, deletion, and substitution, minimally

\*necessary to convert one sequence into the other.

\*S1 and S2 - two sequences, row vectors of length n and m (vectors can be numeric and/or string).

\*EDITCOSTS - costs of three edit operations: insertion, deletion, substitution; vector of three

\*nonnegative values. Levenshtein distance minimazes the sum cost of all the edit operations done.

\*The 1st value - cost of insertion in S1 (= deletion from S2);

\*2nd value - cost of deletion from S1 (= insertion in S2); 3rd value - cost of substitution

\*(substitution is by effect equivalent to deletion and insertion into that place).

\*Result NAME - matrix sized m+1 x n+1. Its last (bottom right) element is the

\*Levenshtein distance between S1 and S2.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E6.

matrix.

compute s1= {'s','i','t','t','i','n','g'}.

compute s2= {'k','i','t','t','e','n'}.

!KO\_levenshtein(s1%s2%{1,1,1}%d).

print d(nrow(d),ncol(d)) /title 'Levenshtein distance'.

end matrix.

### LONGEST COMMON SUBSEQUENCE [!KO\_lcs]

\*/\*!KO\_lcs(s1%s2%subseq%name1%name2)\*/\*.

\*Version 1.

\*S1 and S2 - two sequences, row vectors of length n and m (the vectors can be numeric and/or string).

\*Result NAME - matrix sized m+1 x n+1. Its last (bottom right) element is the length of

\*the maximal common subsequence between S1 and S2.

\*The subsequence itself is the row vector NAME2.

\*SUBSEQ - digit (not name or expression) 1 or 0 (you may put the digit in quotes or apostrophes).

\*If 0, the maximal common subsequence NAME2 is not returned, NAME2 will be scalar 0.

\*If the length of the maximal common subsequence =0, NAME2 will be scalar 0.

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E8.

matrix.

compute s1= {'s','i','t','t','i','n','g'}.

compute s2= {'k','i','t','t','e','n'}.

!KO\_lcs(s1%s2%1%mat%sub).

print mat.

compute length= mat(nrow(mat),ncol(mat)).

print length /title 'Length of the longest common subsequence'.

do if length>0. /\*Print the subsequence if it exists

print sub /format= a2 /title 'The longest common subsequence'.

end if.

end matrix.

### NEEDLEMAN-WUNSCH ALGORITHM [!KO\_needlwun]

\*/\*!KO\_needlwun(s1%s2%matchscore%mismatchscore%gapscore%gapsymb%name1%name2)\*/\*.

\*Version 1.

\*An algorithm for "optimal matching" or "optimal alignment" of two sequences

\*vis-a-vis each other. The sequences are allowed to shift along each other, as well as

\*be stretched through insertion of gaps between elements, in order to reach counter-position

\*with the maximal sum of scores (usually positive score is appointed for symbol match,

\*and negative score for symbol mismatch or for insertion of gap).

\*S1 and S2 - row vectors of length m and n, respectively, these are two being compared sequences.

\*The sequences may be numeric or string.

\*MATCHSCORE - score for matching of symbols, numeric scalar.

\*MISMATCHSCORE - score for matching (difference) of symbols, numeric scalar.

\*GAPSCORE - score for insertion of gap, numeric scalar.

\*GAPSYMB - suggest a symbol for the gap (to use it in NAME2).

\*Results:

\*NAME1 - m+1 x n+1 matrix of scores of all alternative alignments. Its last (bottom-right)

\*element is the sum of scores of the alignment NAME2.

\*NAME2 - two-row matrix, that is the "optimal" alignment. Its first row is S1 (with insertions of

\*gaps, where needed). Its second row is S2 (with insertions of gaps, where needed). Each pair of

\*opposite matching symbols is scored by the MATCHSCORE score, each pair of opposite unmatching

\*symbols is scored by the MISMATCHSCORE score, each pair symbol-gap is scored by the GAPSCORE score.

\*The sum of scores across all pairs equals the last element of NAME1.

\*(Often, more than one alignment have this maximal sum: NAME2 displays just one of these

\*equivalent ways to align.)

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E6.

matrix.

compute s1= {'g','a','a','c'}.

compute s2= {'c','a','a','g','a','c'}.

print s1 /format= a1.

print s2 /format= a1.

compute match= 1.

compute mismatch= -10.

compute gap= -1.

print {match,mismatch,gap} /title "User's Match, Mismatch, and Gap scores".

!KO\_needlwun(s1%s2%match%mismatch%gap%'-'%name1%name2).

print name1

/title 'Alignments scoring matrix (The last value is the optimal alignment score)'.

print name2 /title 'The optimal alignment' /format= a1.

print /title 'gap+gap+gap+match+gap+match+match = -1 is the maximized score'.

end matrix.

### NEEDLEMAN-WUNSCH ALGORITHM (WITH SIMILARITY MATRIX) [!KO\_needlwun2]

\*/\*!KO\_needlwun2(s1%s2%symblist%symbsim%gapscore%gapsymb%name1%name2)\*/\*.

\*Version 1.

\*This function is like /\*!KO\_needlwun\*/, but allows to specify individual scores of matches and

\*mismatches to specific symbols of the language on which the sequences are written. Of these scores,

\*the input similarity matrix SYMBSIM between the language symbols is comprised.

\*S1 and S2 - row vectors of length m and n, respectively, these are two being compared sequences.

\*The sequences may be numeric or string.

\*SYMBLIST - row vector, the list of symbols used in S1 and S2, it is the alphabet on which the

\*sequences are written.

\*SYMBSIM - corresponding to SYMBLIST square symmetric matrix of pairwise similarities between the

\*symbols. Scores for matches stand on the matrix diagonal, and scores for mismatches are the

\*off-diagonal elements.

\*If in S1 or S2 there encounters a symbol absent in SYMBLIST, it counts as if it is present

\*in SYMBLIST but its corresponding entries in matrix SYMBSIM are zero.

\*GAPSCORE - score for insertion of gap, numeric scalar.

\*GAPSYMB - suggest a symbol for the gap (to use it in NAME2).

\*Results:

\*NAME1 - m+1 x n+1 matrix of scores of all alternative alignments. Its last (bottom-right)

\*element is the sum of scores of the alignment NAME2.

\*NAME2 - two-row matrix, that is the "optimal" alignment. Its first row is S1 (with insertions of

\*gaps, where needed). Its second row is S2 (with insertions of gaps, where needed). Each pair of

\*opposite matching symbols is scored by the corresponding score from matrix SYMBSIM, and each pair

\*symbol-gap is scored by the GAPSCORE score.

\*The sum of scores across all pairs equals the last element of NAME1.

\*(Often, more than one alignment have this maximal sum: NAME2 displays just one of these

\*equivalent ways to align.)

\*This function may require prior setting of limit for number of cycles to enough big value

\*by command SET MXLOOPS.

EXAMPLE.

set mxloops 1E6.

matrix.

compute s1= {'g','a','a','c'}.

compute s2= {'c','a','a','g','a','c'}.

print s1 /format= a1.

print s2 /format= a1.

compute symblist= {'a','g','c','t'}.

compute symbsim= make(4,4,-10).

call setdiag(symbsim,1).

print symbsim /rnames= symblist /cnames= symblist

/title= "User's Match/Mismatch scores as a similarity matrix between symbols".

compute gap= -1.

print gap /title "User's Gap score".

!KO\_needlwun2(s1%s2%symblist%symbsim%gap%'-'%name1%name2).

print name1

/title 'Alignments scoring matrix (The last value is the optimal alignment score)'.

print name2 /title 'The optimal alignment' /format= a1.

print /title 'gap+gap+gap+match+gap+match+match = -1 is the maximized score'.

end matrix.