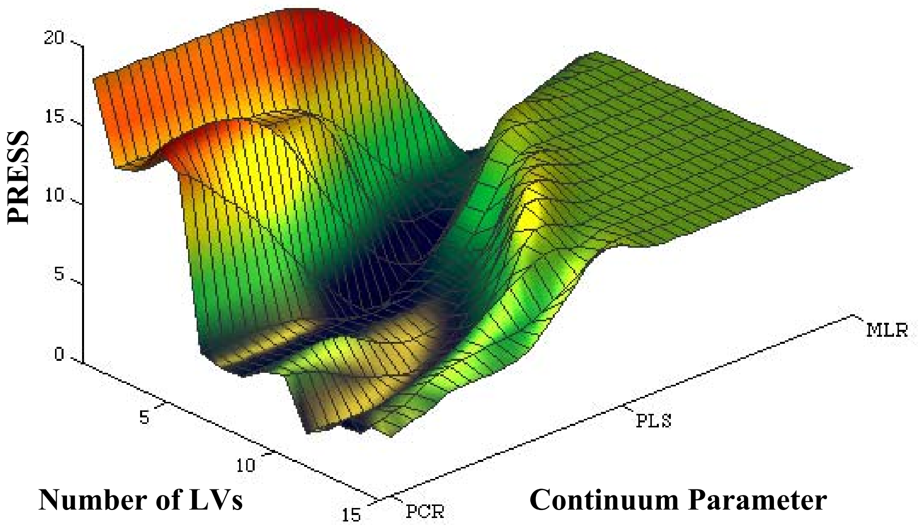


PLS_Toolbox 4.2 Reference Manualf

for use with MATLAB™

Barry M. Wise
Neal B. Gallagher
Rasmus Bro
Jeremy M. Shaver
Willem Windig
R. Scott Koch

EIGENVECTOR RESEARCH INCORPORATED

Eigenvector Research, Inc., Software License Agreement

READ THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT CAREFULLY BEFORE USING THIS SOFTWARE. THIS LICENSE AGREEMENT REPRESENTS THE ENTIRE AGREEMENT BETWEEN YOU (THE "LICENSEE" - EITHER AN INDIVIDUAL OR AN ENTITY) AND EIGENVECTOR RESEARCH, INC., ("EVRI") CONCERNING THE PLS_TOOLBOX COMPUTER SOFTWARE CONTAINED HEREIN ("PROGRAM"), AND THE ACCOMPANYING USER DOCUMENTATION.

BY USING THE SOFTWARE, YOU ACCEPT THE TERMS OF THIS AGREEMENT. IF YOU ARE NOT WILLING TO DO SO, RETURN THE UNOPENED SOFTWARE IMMEDIATELY FOR A FULL REFUND.

LICENSE GRANT: This license permits licensee to install and use one copy of the Program on a single computer. If licensee has multiple licenses for the Program, then Licensee may at any time have as many copies of the Program and its Electronic Documentation in use as it has licenses. "Use" means that a copy is loaded into temporary memory or installed into the permanent memory of a computer, except that a copy installed on a network server for the sole purpose of distribution to other computers is not in "use". Licensee is responsible for limiting the number of possible concurrent users to the number licensed. Each copy of the Program may be used on a backup computer (when the original is disabled) or a replacement computer. Replacements may be either permanent or temporary, at the same or different site as the original computer. The Hardcopy documentation provided with the Program may not be copied.

Licensee shall use the Program only for its internal operations. "Internal operations" shall include use of the Program in the performance of consulting or research for third parties who engage Licensee as an employee or independent contractor. Licensee may allow use of the Program by employees, consultants, students and/or (in the case of individual licensees) colleagues, but Licensee may not make the Program available for use by third parties generally on a "time sharing" basis.

Licensee may make copies of the Program only for backup or archival purposes. All copies of Program, Electronic Documentation and Hardcopy Documentation shall contain all copyright and proprietary notices in the originals. Licensee shall not re-compile, translate or convert "M-files" contained in the Program for use with any software other than MATLAB®, which is a product of The MathWorks, Inc. 3 Apple Hill Drive, Natick, MA 01760-2098, without express written consent of EVRI. Licensee shall not re-distribute "M-files" contained in the Program, or any derivative thereof, without express written consent of EVRI.

Licensee shall take appropriate action by instruction, agreement, or otherwise with any persons permitted access to the Program, so as to enable Licensee to satisfy the obligations under this agreement.

TERM OF AGREEMENT. This Agreement shall continue until terminated by EVRI or Licensee as provided below.

TERMINATION. EVRI may terminate this license by written notice to Licensee if Licensee (a) breaches any material term of this Agreement, (b) fails to pay the amount charged for this license within Thirty (30) days after the date due, or (c) ceases conducting business in the normal course, becomes insolvent or bankrupt, or avails itself of or becomes subject to any proceedings pertaining to insolvency or protection of creditors. Licensee may terminate this Agreement at any time by written notice to EVRI. Licensee shall not be entitled to any refund if this Agreement is terminated, except of license fees paid for any Licensed Product for which the testing period has not expired at the time of termination. Upon termination, Licensee shall promptly return all copies of the Programs and Documentation in Licensee's possession or control, or promptly provide written certification of their destruction.

LIMITED WARRANTY; LIMITATION OF REMEDIES. For a period of ninety (90) days from delivery, EVRI warrants that (a) the media shall be free of defects, or replaced at no cost to Licensee, and (b) the Program will conform in all material respects to the description of such Program's operation in the Documentation. In the event that the Program does not materially operate as warranted, licensees exclusive remedy and EVRI's sole liability under this warranty shall be (a) the correction or workaround by EVRI of major defects within a reasonable time or (b) should such correction or workaround prove neither satisfactory nor practical, termination of the License and refund of the license fee paid to EVRI for the Program. THE FOREGOING WARRANTY IS IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. EVRI SHALL NOT BE LIABLE FOR ANY SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING WITHOUT LIMITATION LOST PROFITS. Licensee accepts responsibility for its use of the Program and the results obtained therefrom.

Table of Contents

Formats and Conventions

The manual for the PLS_Toolbox uses a format consistent with that used for MATLAB. For additional information on usage see the main PLS_Toolbox manual. The following format is used in the Reference section:

Purpose	Provides short concise descriptions of a PLS_Toolbox command or function.
Synopsis	Shows the input/output format of the command or function.
Description	Describes what the command or function does and any rules or restrictions that apply.
Examples	Provides examples of how the command or function can be used.
Options	Describes advanced options of the command or function.
Algorithm	Describes algorithms and routines used within the command or function.
See Also	Refers to other related commands or functions in the PLS_Toolbox.

and the following conventions:

`Monospace`	Commands, function names, and screen displays; for example, `pca`.
Italics	Book titles, names of sections in this book, MATLAB toolbox names, and for introduction of new terms; for example, *Chemometrics*.
`Monospace`	Optional input variables from PLS_Toolbox functions.

Routines in the PLS_Toolbox follow the convention of having samples in rows and variables in columns.

PLS_Toolbox Functions

abline

Purpose

Adds a line on the current axes with a given slope and intercept.

Synopsis

```
h = abline(slope,intercept)
h = abline(slope,intercept,...)  %additional linestyle information
```

Description

`ABLINE` draws a line on on an existing axes with a given slope, `slope`, and intercept, `intercept`, using the existing x-axis range for values. If a 3D plot is shown, slope and intercept can be 2-element vectors describing the slope and intercept of the line in the y and z dimensions. Optional line style information can also be included. For more information on linestyle information, see the manual page on the `line` command. The handle of the new line object is returned.

Examples

```
abline( 3, -1, 'color', 'r', 'linestyle', '--')
```

plots a dashed red line with a slope of 3 and an intercept of -1 on the axes.

See Also

`dp, hline, line, vline`

alignmat

Purpose

Alignment of matrices and N-way arrays

Synopsis

```
[bi,itst] = alignmat(amodel,b);
[bi,itst] = alignmat(a,b,nocomp);
```

Description

In some cases, data arrays require alignment to aid the performance of the three-way (e.g. GRAM, or PARAFAC) or unfold models such as MPCA. For example, sometimes GC peaks or data from batch operations can be shifted on a sample-to-sample basis (each sample is a M_b by N matrix). In these cases, it is advantageous to choose a sub-matrix of a single matrix **A** as a standard and find the sub-matrix of subsequent samples **B** that best align or match the standard matrix. It is also possible to use a model of one or more standard matrices $\mathbf{A}_{model}$ and find the sub-matrix of subsequent samples **B** that best align or match the model. In the latter case, it is also possible to find the sub-array of **B** that best aligns with the model of a N-way data set ($\mathbf{A}_{model}$). This can be performed along multiple modes using `ALIGNMAT`.

`ALIGNMAT` finds the subarray of `b`, `bi`, that most matches `a` using two different algorithms. For input:

```
[bi,itst] = alignmat(amodel,b);
```

the sub-array `bi` is found using a projection method. In this case, `bi` is the sub-array of `b` that has the lowest residuals on a model of `a` called `amodel`. Models for `amodel` are standard model structures from PCA, PCR, GRAM, TLD, or PARAFAC. Input `b` can be class "double" or "dataset" and must have the same number of modes/dimensions as `a` with each element of `size(b)` ≥ `size(a)`. Alignment is performed for modes with `size(b)` > `size(a)`.

For input:

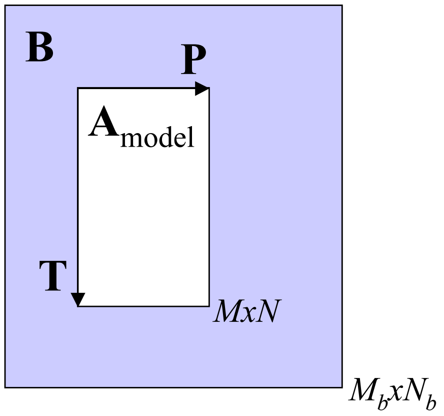
```
[bi,itst] = alignmat(a,b,ncomp);
```

both `a` and `b` can be class "double" or "dataset", but both are two-way arrays (matrices). For `a` `M` by `N` then `b` must be `Mb` by `N` where `Mb` ≥ `M` (when `Mb` = `M` no alignment is performed). The output `bi` is the sub-array of `b` that best matches the matrix `a`. Optional input `ncomp` is a scalar of the number of components to use in the decomposition {default: `ncomp = 1`}.

Output `bi` is an array of class "double", `itst` is a cell array containing the indices of `b` that match `bi`. Note that since interpolation is used the indices in `itst` are *not* in general integers.

Algorithm

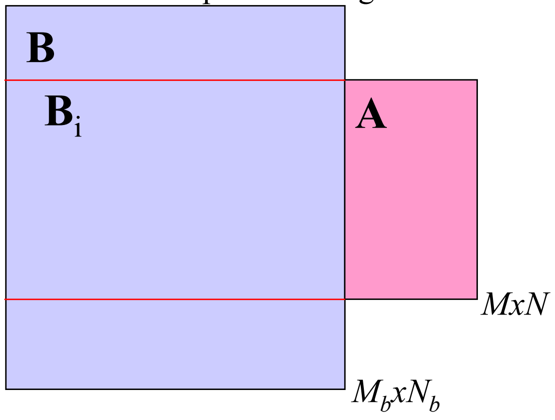
For the projection method, $\mathbf{A}_{\text{model}}$ is a model of array $\mathbf{A}$. This can be a model from PCA, GRAM, TLD, or PARAFAC. For example, if $\mathbf{A}$ is a *M* by *N* matrix then the PCA model of $\mathbf{A}$ is $\mathbf{A} = \mathbf{TP}^T + \mathbf{E}$ where $\mathbf{T}$ is *M* by *K* and $\mathbf{P}$ is *N* by *K*. Alignmat finds the submatrix of $\mathbf{B}$, $\mathbf{B}_i$, that has the lowest residuals on the model of $\mathbf{A}$ i.e. $\mathbf{B}_i = \min\left(\sum_{n=i}^{N+i}\sum_{m=j}^{M+j}\left[\mathbf{B}_{i,j}\left(\mathbf{I}-\mathbf{PP}^T\right)\right]^2 \mid i,j\right)$. This can be used to find the data "cube" within N-way arrays.



In the figure, this is represented as having each of the *M* by *N* sub-matrices of $\mathbf{B}$ projected onto the model of the *M* by *N* model of $\mathbf{A}$. Note that in the figure that the size of $\mathbf{B}$ is M_b by N_b with $M_b > M$ and $N_b > N$.

The projection method was presented in Gallagher, N.B. and Wise, B.M., "Standardization for Three-Way Analysis", *TRICAP 2000: Three-way Methods in Chemistry and Psychology*, Hvedholm Castle, Faaborg, Denmark, July (2000). In that study, it was found that the projection method was faster and more robust than the SVD-based algorithm discussed below.

In the SVD method, the standard matrix $\mathbf{A}$ and a sub-matrix of $\mathbf{B}$, $\mathbf{B}_i$, are aumented and a singular value decomposition of the result is performed such that [u,s,v] = svd([$\mathbf{A}_{MxN}$|$\mathbf{B}_{iMxNb}$]). The sub-matrix is incremented and the SVD is performed again. The sub-matrix that minimizes the rank is selected as matching best. The objective function is $R = \left(\sum_{j=ncomp+1}^{\min(M,N+N_b)} s_j\right)\left(\sum_{j=1}^{ncomp} s_j\right)^{-1}$. Note that in this algorithm *N* and N_b do not have to be equal. The algorithm is discussed in Prazen, et al., *Anal. Chem.*, **70**, 218-225, 1998.



See Also

`analysis, gram, parafac, pca, tld`

alignpeaks

Purpose

Calibrates wavelength scale using standard peaks.

Synopsis

```
s = alignpeaks(x0,x1,ax,options)
y = alignpeaks(s,y1)
```

Description

`ALIGNPEAKS` calibrates a wavelength scale using standard peak positions. Ideally, the axis scale `x0` would apply to a single instrument at time $t = 0$ and $t > 0$ or for two different instruments. However, `x1` at $t > 0$ doesn't typically match `x0` at $t = 0$ even though the numbers in the scales are identical. The result is that a plot of (`x0,y0`) and (`x0,y1`) appear shifted from one another.

The inputs to `ALIGNPEAKS` are `x0` a 1x*K* vector containing the axis locations of *K* peaks on the standard instrument at $t = 0$ (e.g., the true wavelengths), `x1` a 1x*K* vector containing the axis locations of the corresponding peaks on the field / test instrument at $t > 0$ (e.g., the peak positions on the field instrument), and `ax` a 1x*N* vector containing the axis scale where $N > K$. `ALIGNPEAKS` finds a polynomial fit between `x0` and `x1` and outputs the result in the structure array `s`. The output y is a fit of x1.

Options

Optional input `options` is a structure array with the following fields:

`name`:	'`options`', name indicating that this is an options structure,
`plots`:	['none' | {'final'}] governs level of plotting, and
`order`:	[{2}] integer giving the polynomial order.

Executing `options = alignpeaks('options');` gives an empty options structure.

Example

A measurements at $t = 0$ gives a spectrum `y0` with axis `ax`, and measurements at $t > 0$ of the same sample yields a spectrum `y1` with the same axis `ax` but with peaks shifted. Therefore

```
plot(ax,y0,'b',ax,y1,'r')
```

shows a shift in the peaks. The peak positions at $t = 0$ are listed in `x0` and the peak positions at $t > 0$ are listed in `x1`. The polynomial fit is given by

```
s    = alignpeaks(x0,x1,ax);
```

and the transformed spectrum is obtained with

```
      y10  = alignpeaks(s,y1);
```

so that

```
      plot(ax,y0,'b',ax,y1,'r')
```

shows less of a peak shift. See `alignpeaksdemo`.

See Also

`alignmat`, `alignspectra`, `registerspec`, `stdgen`

alignspectra

Purpose

Calibrates wavelength scale using a standard spectrum.

Synopsis

```
[s,y] = alignspectra(x0,y0,y1,win,mx2,options)
y  = alignspectra(s,y1);
```

Description

ALIGNSPECTRA calibrates a wavelength scale using a standard spectrum and a piece-wise shifting that maximizes correlation between windows on the standard spectrum to windows on the test spectrum. Ideally, the axis scale would be the same for all time and all instruments, however it can be necessary to calibrate the axis scale. This calibration is often done somewhat manually using known standard peak positions (see ALIGNPEAKS). In the ALIGNSPECTRA function a standard is measured on both the standard instrument with spectrum y0 and the field instrument with spectrum y1. The transform is based on a polynomial fit of the center channel of a window of channels (window size win) on the field instrument that best correlates with a similar sized window of channels on the standard instrument. The window on the field instrument is allowed to shift a maximum of mx2 channels.

The inputs to ALIGNSPECTRA are x0 a 1x*N* vector containing the axis scale of the standard instrument at $t = 0$ (e.g., the true wavelengths), y0 a 1x*N* spectrum measured on the standard instrument at $t = 0$, y1 a 1x*N* spectrum measured on the field instrument at $t > 0$, a window width of channels on the axis scale win, and the maximum number of channels to shift mx2.

Options

Optional input options is a structure array with the following fields:

name:	'options', name indicating that this is an options structure.
plots:	['none' | {'final'}] governs level of plotting.
interpolate:	['none' | {'linear'} | 'cubic'] dictates the interpolation scheme used when shifting the window. 'none' uses the coarse scale given by x0. Using other interpolation schemes can significantly increase the time required for computation (the algorithm calls the function INTERP1).
order:	[{2}] integer giving the polynomial order.

Executing options = alignspectra('options'); gives an empty options structure.

Example

A measurements at $t = 0$ gives a spectrum y0 with axis ax, and measurements at $t > 0$ of the same sample yields a spectrum y1 with the same axis ax but with peaks shifted. Therefore

```
plot(ax,y0,'b',ax,y1,'r')
```

shows a shift in the peaks. The peak positions at $t = 0$ are listed in `x0` and the peak positions at $t > 0$ are listed in `x1`. The polynomial fit is given by

```
s       = alignspectra(x0,y0,y1,25,7); %or
[s,y10] = alignspectra(x0,y0,y1,25,7);
```

and the transformed spectrum is obtained with

```
y10  = alignspectra(s,y1);
```

so that

```
plot(ax,y0,'b',ax,y1,'r')
```

shows less of a peak shift. See `alignspectrademo`.

See Also

`alignmat`, `alignpeaks`, `registerspec`, `stdgen`

als

Purpose

Alternating Least Squares computational engine for multivariate curve resolution (MCR).

Synopsis

```
[c,s] = als(x,c0,options);
```

Description

ALS decomposes a matrix X as CS such that X = CS + E where E is minimized in a least squares sense.

Inputs are the matrix to be decomposed `x` (size `m` by `n`), and the initial guess `c0`. If `c0` is size `m` by `k`, where `k` is the number of factors, then it is assumed to be the initial guess for C. If `c0` is size `k` by `n` then it is assumed to be the initial guess for S (If `m`=`n` then, `c0` is assumed to be the initial guess for C).

An optional input `options` is described below.

The outputs are the estimated matrix `c` (m by k) and `s` (k by n). Usually `c` is a matrix of contributionss and `s` is a matrix of spectra. The function

```
[c,s] = als(x,c0)
```

will decompose `x` using an non-negatively constrained alternating least squares calculation. To include other constraints, use the options described below.

Note that if no non-zero equality constraints are imposed on a factor the spectra are normalized to unit length. This can lead to significant scaling differences between factors that have non-zero equality constraints and those that do not.

Options

`display`:	['off' | {'on'}] governs level of display to command window,
plots:	['none' | {'final'}] governs level of plotting,
`ccon`:	['none' | 'reset' | {'fastnnls'}] non-negativity on contributionss, (fastnnls = true least-squares solution)
`scon`:	['none' | 'reset' | {'fastnnls'}] non-negativity on spectra, (fastnnls = true least-squares solution)
`cc`:	[ ] contributions equality constraints, must be a matrix with M rows and up to K columns with NaN where equality constraints are not applied and real value of the constraint where they are applied. If fewer than K columns are supplied, the missing columns will be filled in as unconstrained,
`ccwts`:	[inf] a scalar value or a 1xK vector with elements corresponding to weightings on constraints (0, no constraint, 0<wt<inf imposes constraint "softly", and inf is hard constrained). If a scalar value is passed for ccwts, that value is applied for all K factors,
`sc`:	[ ] spectra equality constraints, must be a matrix with N columns and up to K rows with NaN where equality contraints are not applied and real value of the constraint where they are applied. If fewer than K rows are supplied, the missing rows will be filled in as unconstrained.
`scwts`:	[inf] weighting for spectral equality constraints (see ccwts)
`sclc`:	[ ] contributions scale axis, vector with M elements otherwise 1:M is used,
`scls`:	[ ] spectra scale axis, vector with N elements otherwise 1:N is used,
`condition`:	[{'none'}| 'norm'] type of conditioning to perform on S and C before each regression step. 'norm' conditions each spectrum or contributions to its own norm. Conditioning can help stabilize the regression when factors are significantly different in magnitude.
`tolc`:	[{1e-5}] tolerance on non-negativity for contributionss,
`tols`:	[{1e-5}] tolerance on non-negativity for spectra,
`ittol`:	[{1e-8}] convergence tolerance,
`itmax`:	[{100}] maximum number of iterations,
`timemax`:	[{3600}] maximum time for iterations,
`rankfail`:	['drop' |{'reset'}| 'random' | 'fail'] how are rank deficiencies handled: `drop` - drop deficient components from model `reset` - reset deficient components to initial guess `random` - replace deficient components with random vector `fail` - stop analysis, give error

Examples

To decompose a matrix x without non-negativity constraints use:

```
options = als('options');
options.ccon = 'none';
options.scon = 'none';
[c,s] = als(x,c0,options);
```

The following shows an example of using soft-constraints on the second spectral component of a three-component solution assuming that the variable `softs` contains the spectrum to which component two should be constrained.

```
[m,n] = size(x);
options = als('options');
options.sc = NaN*ones(3,n);   %all 3 unconstrained
options.sc(2,:) = softs;      %constrain component 2
options.scwts = 0.5;          %consider as ½ of total signal in X
[c,s] = als(x,c0,options);
```

See Also

`mcr, parafac, pca`

analysis

Purpose

Graphical user interface for data analysis.

Synopsis

```
analysis
```

Description

Performs various analysis methods including PCA, MCR, PARAFAC, Cluster, PLS, PCR, PLSDA, and SIMCA using a graphical user interface. Typical operations for file manipulation, preprocessing, and Analysis selection can be found in the menu items of the figure. Once data has been loaded and an Analysis selected, the Toolbar will populate with appropriate buttons for the Analysis. Plots created by the Toolbar buttons will bring up a plot figure window as well as a plot controls window. Use the plot controls window to manipulate the plot figure.

Note: For more information see Chapter 5 of the PLS_Toolbox Manual.

See Also

`browse, cluster, mcr, parafac, pca, pcr, pls`

anova1w

Purpose

One way analysis of variance.

Synopsis

```
anova1w(dat,alpha)
```

Description

Calculates one way ANOVA table and tests significance of between factors variation (it is assumed that each column of the data represents a different treatment). Inputs are the data table dat and the desired confidence level alpha, expressed as a fraction (*e.g.* 0.95, 0.99, etc.). The output is an ANOVA table written to the command window.

See Also

`anova2w, ftest, statdemo`

anova2w

Purpose

Two way analysis of variance.

Synopsis

```
anova2w(dat,alpha)
```

Description

Calculates two way ANOVA table and tests significance of between factors variation (it is assumed that each column of the data represents a different treatment) and between blocks variation (it is assumed that each row represents a block). Inputs are the data table dat and the desired confidence level alpha, expressed as a fraction (*e.g.* 0.95, 0.99, etc.). The output is an ANOVA table written to the command window.

See Also

`anova1w, ftest, statdemo`

areadr

Purpose

Reads ASCII text file into workspace and strips off header.

Synopsis

```
      out = areadr1(file,nline,nvar,flag)
```

Description

Inputs are (`file`) an ASCII string containing the file name to be read, (`nline`) the number of rows to skip before reading or a character string containing the last few characters before the first number to be read (used to skip the header information), (`nvar`) the number of rows or columns in the matrix to be read, and (`flag`) which indicates whether (`nvar`) is the number of rows (`flag=1`) or the number of columns (`flag=2`) in the matrix.

`AREADR` can be incorporated into other routines to read data directly from groups of files. For example, to read the file `mydata.txt` with a 5 line header and 8 columns in the data into the matrix `mymatrix`:

```
mymatrix = areadr('mydata.txt',5,8,2)
```

Given header information in a text file with the following contents:

```
HEADER INFORMATION
HEADER ONE
HEADER TWO
END OF HEADER INFORMATION

1 2 1 2
2 3 2 3
3 4 3 4
4 5 4 5
```

The following command will read the 4 rows of data following the character string "END OF HEADER INFORMATION":

```
mymatrix = areadr('mydata.txt','END OF HEADER INFORMATION',4,1)
```

For an automatic text file parser which can handle this type of file without knowing the format, see `xclreadr`.

See Also

`dlmread, import, spcreadr, xclgetdata, xclputdata, xclreadr, xlsreadr`

auto

Purpose

Autoscales a matrix to mean zero and unit variance.

Synopsis

```
[ax,mx,stdx,msg] = auto(x,options)
[ax,mx,stdx,msg] = auto(x,offset)
options = auto('options')
```

Description

`[ax,mx,stdx] = auto(x)` autoscales a matrix `x` and returns the resulting matrix `ax` with mean-zero unit variance columns, a vector of means `mx` and a vector of standard deviations `stdx` used in the scaling. Output `msg` returns any warning messages. If missing data `NaNs` are found, the available data is autoscaled if the fraction missing is not above the thresholds specified below. `mx` and `stdx` can be used to scale new data (see `SCALE`).

Options

`options` =	a structure array with the following fields:
`offset`:	scaling can use standard deviation plus an offset {default = 0},
`display`:	`[ {'off'}| 'on' ]` governs level of display to the command window,
`matrix_threshold`:	fraction of missing data allowed based on entire matrix (x) {default = 0.15}, and
`column_threshold`:	fraction of missing data allowed base on a single column {default = 0.25}.
`algorithm`:	`[ {'standard'} | 'robust']` scaling algorithm. 'robust' uses MADC for scaling and median instead of mean. Should be used for robust techniques,
`stdthreshold`:	`[ 0 ]` scalar or vector of standard deviation threshold values. If a standard deviation is below its corresponding threshold value, the threshold value will be used in lieu of the actual value. Note that the actual standard deviation is always returned, whether or not it exceedes the threshold. A scalar value is used as a threshold for all variables,
`badreplacement`:	[0] value to use in place of standard deviation values of 0 (zero). Typical values used with the following effects: 0 = Any value in given variable is set to zero. Variable is effectively excluded (but still expected by model). This is also the behavior when badreplacement = inf. 1 = Values different from mean of the given variable are flagged in Q residuals with no reweighting.

Values >0 and <inf give the variable different weighting in the Q residuals (values >1 down-weight the bad variables for Q residual calculations, values <1 up-weight the bad variables.).

If the input (offset) is a scalar then, this is used as the offset value with other options set at their default values.

The optional input *offset* is added to the standard deviations before scaling and can be used to suppress low-level variables that would otherwise have standard deviations near zero.

The default options can be retreived using: `options = auto('options');`.

See Also

`gscale, medcn, mncn, normaliz, npreprocess, regcon, rescale, scale, snv`

autoimport

Purpose

Automatically reads specified file. Handles all standard filetypes.

Synopsis

```
autoimport(filename,methodname,options)
[data,name,source] = autoimport(filename,methodname,options)
```

Description

Automatically identifies a filetype and calls the appropriate reader. If no filename is provided, the user is prompted for a desired filetype to browse for. If no filename is provided but a specific filetype is provided, the user is prompted for a file of the given type.

If output is requested, the loaded item(s) is/are returned as a single output. If no outputs are requested, the items are loaded into the base workspace or other action as defined by the options structure.

Options

`options` =	a structure array with the following fields:
`target`:	`[{'workspace'}	'analysis'	'editds']` Target for file load. If 'workspace', file contents are loaded into base workspace (the default behavior). If 'analysis', file contents are automatically dropped into an empty Analysis GUI interface. If 'editds', file contents are loaded into a DataSet editor.
`defaultmethod`:	`[{'prompt'}	'string'	'error'	methodname]` governs how to handle input (filename) when no recognizable file extension can be found. 'prompt' prompts the user to identify the appropriate importer, 'string' interprets the input as a string, 'error' returns an error. Any other valid methodname can also be provided (use autoimport('methods') to get list of valid methods),
`error`:	`['error'	{'gui'}]` governs how to handle errors during imports. 'error' returns an untrapped error, 'gui' traps the error and presents an error dialog to the user.

See Also

`imageload, jcampreadr, parsexml, spcreadr, xclreadr, xyreadr`

autocor

Purpose

Calculates the autocorrelation function of a time series.

Synopsis

```
    acor = autocor(x,n,period,plots)
```

Description

`acor = autocor(x,n)` returns the autocorrelation function `acor` of a time series `x` for a maximum time shift of `n` sample periods.

`acor = autocor(x,n,`*period*`)` uses the sampling period *period* to scale the x-axis on the output plot. *period* can be empty `[]`.

The optional input *plots* suppresses plotting if set to 0.

See Also

`corrmap, crosscor`

b3spline

Purpose

Univariate spline fit and prediction.

Synopsis

```
modl = b3spline(x,y,t,options);
pred = b3spline(x,modl,options);
valid = b3spline(x,y,modl,options);
```

Description

Curve fitting using second order splines where

yi = f(xi) for i=1,...,M.

See (options.algorithm) for more information.

INPUTS:

`x` =	Mx1 vector of independent variable values.
`y` =	Mx1 vector of corresponding dependendent variable values.
t =	defines the number of knots or knot positions.
	= 1x1 scalar integer defining the number of uniformly distributed INTERIOR knots. There will be t+2 knots positioned at:
	modl.t = linspace(min(x),max(x),t+2)';
	= Kx1 vector defining manually placed knot positions,
	where modl.t = sort(t);
	Note that knot positions need not be uniform, and that t(1) can be <min(x) and t(K) can be >max(x).

Note that knot positions must be such that there are at least 3 unique data points between each knot: tk,tk+1 for k=1,...,K.

OUTPUTS:

`modl` =	standard model structure containing the spline model (See MODELSTRUCT).
`pred` =	structure array with predictions.
valid =	structure array with predictions.

Options

`options` =	a structure array with the following fields:
`display`:	`[{'on'}	'off']` level of display to command window.

`plots`: `[ {'final'} | 'none' ]` governs level of plotting. If 'final' and calibrating a model, the plot shows plot(xi,yi) and plot(xi,f(xi),'-') with knots.

`algorithm`: `[ {'b3spline'} | 'b3_0' | 'b3_01' ]` fitting algorithm

'b3spline': fits quadradic polynomials f{k,k+1} to the data between knots tk, k=1,...,K, subject to:

f{k,k+1}(tk+1) = f{k+1,k+2}(tk+1) and

f'{k,k+1}(tk+1) = f'{k+1,k+2}(tk+1) for k=1,...,K-1.

'b3_0': is the same as 'b3spline' but also constrains the ends to 0: f{1,2}(t1) = 0 and f{K-1,K}(tK) = 0.

'b3_01': is 'b3_0' but also constrains the derivatives at the ends to 0: f'{1,2}(t1) = 0 and f'{K-1,K}(tK) = 0.

`order`: positive integer for polynomial order {default = 1}.

The default options can be retreived using: `options = baseline('options');`.

See Also

baseline

Purpose

Subtracts a baseline offset from spectra.

Synopsis

```
[newspec,b] = baseline(spec,freqs,range,options);
spec = baseline(newspec,freqs,b,options);
```

Description

This function baselines spectra with a polynomial baseline function. The baseline function is fit to user-specified regions (regions free of peaks), which is then subtracted from the original spectra.

Inputs are `spec` class "double" or "dataset" containing the spectra, `freqs` the wavenumber or frequency axis vector, and `range` which specifies the baselining regions (see below). If `freqs` is omitted and `spec` is a dataset, the axissscale from the dataset will be used; otherwise a linear vector will be used.

`range` can be either an m by 2 matrix which specifies m baselining regions or a logical vector equal in length to the spectra with a 1 (one) at each point to be used as baseline and 0 (zero) elsewhere.

The output `newspec` contains the baselined spectra and `b` the polynomial coefficients.

If `b` is input instead of `range` with baselined spectra `newspec` then the output `spec` is a matrix original "unbaselined" spectra.

Options

`options` = a structure array with the following fields:

- `plots`: `[ {'none'} | 'final' ]` governs plotting of results, and
- `order`: positive integer for polynomial order {default = 1}.

The default options can be retreived using: `options = baseline('options');`.

See Also

`baselinew, deresolv, lamsel, lsq2top, normaliz, polyinterp, savgol, savgolcv, specedit, stdgen, wlsbaseline`

baselinew

Purpose

Baseline using windowed polynomial filter.

Synopsis

```
[y_b,b_b]= baselinew(y,x,width,order,res,options)
```

Description

BASELINEW fits a polynomial "baseline" to the bottom (or top) of a curve (e.g. a spectrum) by recursively calling LSQ2TOP. It uses a windowed approach and can be considered a filter or baseline (low frequency) removal algorithm. The window width required depends on the frequency of the low frequency component (baseline). Wide windows and low order polynomials are often used. See LSQ2TOP for more details on the polynomial fit algorithm.

Inputs include the curve(s) to be fit (dependent variable) y, the axis to fit against (the independent variable) x [e.g. y = P(x)], the window width width (an odd integer), the polynomial order order, and an approximate noise level in the curve res. Note that y can be *M*x*N* where x is *1*x*N*. The optional input options is discussed below.

Output y_b is a *M*x*N* matrix of ROW vectors that have had the baselines removed, and output b_b is a matrix of baselines. Therefore, y_b is the high frequency component and b_b is the low frequency component.

INPUTS:

y =	matrix of ROW vectors to be baselined, MxN [class double].
x =	axis scale, 1xN vector {if empty it is set to 1:N}.
width =	window width specifying the number of points in the filter {if (width) is empty no windowing is used}.
order =	order of polynomial [scalar] to fit {if (order) is empty (options.p) must not be empty; see below}.
res =	approximate fit residual [scalar] {if empty it is set to 5Found of fit of all data to x}.

Examples

If y is a 5 by 100 matrix then
```
y_b = baselinew(y,[],25,3,0.01);
```

gives a 5 by 100 matrix y_b of row vectors that have had the baseline removed using a 25-point cubic polynomial fit of each row of y.

If y is a 2 by 100 matrix then
```
y_b = baselinew(y,x,51,3,0.01);
```

gives a 2 by 100 matrix y_b of row vectors that have had the baseline removed using a 51-point second order polynomial fit of each row of y to x.

Options

options	= structure array with the following fields:
display :	['off' | {'on'}] governs level of display to command window.
trbflag :	['top' | {'bottom'}] top or bottom flag, tells algorithm to fit the polynomials, y = P(x), to the top or bottom of the data cloud.
tsqlim:	[0.99] limit that governs whether a data point is significantly outside the fit residual defined by input res.
stopcrit:	[1e-4 1e-4 1000 360] stopping criteria, iteration is continued until one of the stopping criterion is met: [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time [seconds])].

See Also

baseline, lamsel, lsq2top, mscorr, savgol, stdfir, wlsbaseline

batchdigester

Purpose

Parse wafer or batch data into MPCA or Summary PCA form.

Synopsis

```
[out,options] = batchdigester(data,options);
batchdigester     %prompt user for input and output
```

Description

Rearranges and optionally summarizes two-way dataset of batch or wafer data. Input `data` must be a DataSet object containing labels which identify different wafers or batches which should be split out of the data. Classes in data are (optionally) used to split each time profile of the batch/wafer into steps which can then be selected for inclusion in the output.

MPCA mode: If data is rearranged into MPCA data, each wafer/batch is arranged as one slab of a 3-way matrix. Each row is a time point and each column is one of the original variables. Only selected steps are included in the output.

Summary PCA mode: If data is summarized into Summary PCA data, all time points for a given step in a given wafer are summarized using one or more statistics:

Mean
Standard Deviation
Minimum
Maximum
Range
Slope
Length (of step)

The time profile for each original variable is summarized using the given statistic(s) and turned into a single variable (column) of the output data. If steps are used, this is repeated for each step segment (each creating a new, separate variable in the output). Each wafer/batch is thus a single row of the output data with all of the steps and original variables summarized as new variables.

Outputs are the digested data, `out`, and the options which can be used to reproduce the digestion process, `options` (see below).

Options

`options` = structure with one or more of the following fields:

`display` : [`'off'` | {`'on'`}] governs level of display to command window.

`object` :	`{ 'batch' | 'wafer' }` A string specifying the type of object being digested. This is used for display ONLY. The same algorithms are used in both cases but this option allows customization of the wording in the user prompts.
`stepclassname` :	A string specifying the name of the class which should be used to indicate steps in the process.
`stepsdesired` :	A vector of steps which should be included in the digestion.
`labelname` :	A string specifying the name of the label set which should be used to split data into batches/wafers. Use the keyword `'fixed'` to specify that the batches are of fixed length and can be split using the `nbatches` option.
`nbatches` :	The number of equally-sized batches to split the data into. Used ONLY when labelname is `'fixed'`.
`digestiontype` :	`[ 'mpca' | 'spca' ]` Specifies which digestion algorithm to use on the data.
`statistics` :	A cell specifying the statistics to be used on the data. Used ONLY when `digestiontype = 'spca';`

If sufficent information is provided in these options, the processing of data will be automatic and the user will not have to answer any responses in the GUIs. Otherwise, only prompts for missing information will be given. The options which can be used to re-process using a given digestion "recipe" will be returned as the second output to any digestion request.

See Also

`mpca, pca`

browse

Purpose

PLS_Toolbox Toolbar and Workspace browser.

Synopsis

```
browse
```

Description

`BROWSE` provides a graphical interface for tools, variables and figures used by PLS_Toolbox. Data items can be dragged onto shortcuts, or into other windows to "load" the data. Data can be dragged to other data items to "augment" these items or can be double-clicked to open in an editor.

See Also

`analysis, editds`

builddbstr

Purpose

Builds a database connection string.

Synopsis

```
str = builddbstr(dbstruct,options)
```

Description

This function is unsupported and is meant as a "simple" database connection tool. For more sophisticated connection tools and full support please see the Matlab Database Toolbox.

It is generally recommended that one use a Microsoft DSN (Data Source Name) to establish connection on Window platforms. These types of connections tend to be easier to maintain and more secure. For more information on DSN, see the Windows help entry for "ODBC". Unix platforms should use JDBC, JDBC with MySQL is a "predefined" method and is known to work with the MySQL JDBC 3.51 Driver.

Input (`dbstruct`) can be:

1) A structure containing necessary information to construct one of the predefined connections listed below. The output will be a properly formatted connection string.

2) A string indicating a predefined structure to return. The output will be a structure containing predefined values along with empty fields that may need to be filled in. Fill in the EMPTY fields as needed and the connection should work. The 'user' and 'pw' fields are always present but may not be needed. This structure can be passed directly to querydb.m.

3) A structure with additional arg.value substructure fields necessary for a connection to a non-predefined database connection. The output will be a properly formatted connection string.

Input: (structure containing the following fields)

A connection will require one of more of the following fields. Empty values are not used.

provider :	only used by ADODB object so this will always be 'MSDASQL'.
driver :	driver to be used for connection (these must be currently installed on the machine, use the ODBC Manager from Administrative Tools to view currently available drivers on your machine. JDBC must have driver installed on Matlab class path
dbname :	database name (or service name).
user :	user to connect in as, if empty not used.
pw :	password for user, if empty not used.

location :	File location on local system (e.g. c:\temp\mydb.mdb). Used for connecting to local Access databases.
server :	IP address for database (default location is 'localhost').
dsn :	Data Source Name (set up on local computer using ODBC Manager from Administrative Tools). If the database connection remains static, this can be a simple way to manage the connection. See the "ODBC" topic in Windows help for more information on DSN.
arg.name :	sub structure of additional arguments. This value must be a sting of exactly what is required in the database connection string.
arg.value :	sub structure of additional arguments. This value must be a sting of exactly what is required in the database connection string.

EXAMPLE:

```
cnn.arg(1).name  = 'PORT';
cnn.arg(1).value = '3306';
cnn.arg(2).name  = 'SOCKET';
cnn.arg(2).value = '123';
```

Predefined Database Connections:

1) Microsoft Access : 'access' Uses standard connection provided with windows (Microsoft Access Driver (*.mdb)) and doesn't require UserID or PW if database doesn't have them defined.

2) Microsoft SQL Server : 'mssql' Not tested.

3) MySQL : 'mysql' Uses (MySQL ODBC 3.51 Driver) form mysql website. Must be downloaded and installed before making connection.

4) Data Source Name : 'dsn' Uses a Data Source Name defined in Windows ODBC Data Source Administrator dialog box. Although 'user' and 'pw' are returned in the structure they are generally not needed for DSN connections, this information is usually resides in the DSN itself.

5) MySQL(JDBC) : 'jmysql' Uses (MySQL JDBC 3.51 Driver) form mysql website. Must be downloaded and installed before making connection. The driver jar file must be added to the Matlab java classpath.

6) All : 'all' Show all available fields.

Options

`isodbc`:	[{ 1 } | 0] Use ODBC connection string formatting. This should be set to 0 if using JDBC.

Examples

Examples of building connection strings on a Windows machine for use with the `querydb` function. For Oracle and other database connections, try using DSN.

Microsoft Access on local machine:

```
>> cnstr = builddbstr('access')
cnstr =
    provider: 'MSDASQL'
      driver: '{Microsoft Access Driver (*.mdb)}'
    location: ''
        user: ''
          pw: ''

>> cnstr.location = 'c:\temp\mydb.mdb';
```

MySQL on remote machine:

```
>> cnstr = builddbstr('mysql')
cnstr =
    provider: 'MSDASQL'
      driver: 'MySQL ODBC 3.51 Driver'
      server: ''
      dbname: ''
        user: ''
          pw: ''
>> cnstr.server = 'mydatabase.mywebsite.com';
>> cnstr.dbname = 'mydatabase';
>> cnstr.user = 'myname';
>> cnstr.pw = 'mypw';
```

MySQL on remote machine (JDBC on Windows):

```
>> cnstr = builddbstr('jmysql')
cnstr =
    driver: 'com.mysql.jdbc.Driver'
    server: ''
    dbname: ''
      user: ''
        pw: ''
>> cnstr.server = 'mydatabase.mywebsite.com';
>> cnstr.dbname = 'mydatabase';
>> cnstr.user = 'myname';
>> cnstr.pw = 'mypw';
```

DSN (Data Source Name):

>> cnstr = builddbstr('dsn')

```
cnstr = 
    provider: 'MSDASQL'
         dsn: ''
        user: ''
          pw: ''
>> cnstr.dsn = 'dsnname';
```

See Also

`querydb, parsemixed`

calibsel

Purpose

Stepwise variable selection (user contributed).

Synopsis

```
channel = calibsel(x,y,alpha,flag)
```

Description

`CALIBSEL` performs the variable selection procedure described in Brown, P.J., Spiegelman, C.H., and Denham, M.C., "Chemometrics and spectral frequency selection", *Phil. Trans. R. Soc. Land. A* 337, 311-322, (1991).

Inputs are the calibration spectra `x` and concentrations `y`, significance level for chi-square test `alpha`, and a variable `flag` that allows the user to modify how the routine iterates. The output `channel` is a vector of indices corresponding to selected channels/wavelengths in `y`.

See Also

`fullsearch`, `gaselctr`, `genalg`

caltransfer

Purpose

Create or apply calibration and instrument transfer models.

Synopsis

```
[transfermodel,x1t,x2t] = caltransfer(x1,x2,method,options)
x2t = caltransfer(x2,transfermodel,options)
[transfermodel,x1t,{x2t_1 x2t_2 x2t_3}] =
   caltransfer(x1,{{x2_1 x2_2 x2_3},method,options)
{x2t_1 x2t_2 x2t_3} =
   caltransfer({x2_1 x2_2 x2_3},transfermodel,options)
```

Description

`CALTRANSFER` uses one of the several transfer functions (methods) available in PLS_Toolbox to return a model and transformed data. The exact I/O is dictated by the transfer function (method) used.

INPUTS:

`x1` =	(2-way array class "double" or "dataset") calibration data (e.g., spectra from the standard instrument).
`x2` =	(2-way array class "double" or "dataset") data to be transformed (e.g., spectra from the instrument to be standardized).
method =	(string) indicating which calibration transfer function (method) to use. Choices are: 'ds' : Direct Standardization 'pds' : Piecewise Direct Standardization 'dwpds' : Double Window Piecewise Direct Standardization 'glsw' : Generalized Least-Squares Weighting 'osc' : Orthogonal Signal Correction 'alignmat' : Matrix Alignment

OUTPUTS:

transfermodel =	standard model structure containing the Calibration Transfervmodel (See MODELSTRUCT).
`x1t` =	Calibration data returned. Depending on the type of calibration function (method) used this may or may not be transformed from the input data (x1).
`x2t` =	Transformed data.

Options

`options`	= structure array with the following fields:
`display :`	[`'off'` | {`'on'`}] governs level of display to command window.
`blockdetails :`	`[ 'compact' | {'standard'} | 'all' ]` extent of data included in model. 'standard' = none, 'all' x-block.
`preprocessing :`	`{[] []}` Preprocessing structures for x and y blocks (see PREPROCESS).

NOTE: There are sub structures for each 'method'. These sub structures include both the input parameters (any additional inputs needed by the function) as well as optional inputs (the options structure for that particular function). For more information on inputs to each method see the help for that function (e.g., help stdgen). Examples of using the substructures:

Example:	OSC requires a "y" block in addition to x1 and x2. The y-block should be assigned via the options structure: opts.osc.y = yblock;
Example:	To assign window widths for DWPDS: options.dwpds.win = [5 3];

See Also

`alignmat, glsw, oscapp, osccalc, stdgen, stdize`

cellne

Purpose

Element by element comparison of two cells for inequality.

Synopsis

```
out = cellne(c1,c2)
```

Description

`CELLNE` compares the two cell inputs, `c1` and `c2`, for inequality. If the cell arrays are the same size, the corresponding cell elements are compared and a similarly sized array of logical (boolean) values, `out`, is returned. The array `out` contains a one if the two cell elements were not equal (different variable type or contents) and a zero if the two cell elements were equal.

If the cell sizes do not match, the function returns a single logical value of 1.

See Also

`comparevars`

centerfigure

Purpose

Places a given figure into a centered default position.

Synopsis

```
centerfigure(fig)
centerfigure(fig,targfig)
```

Description

Given a figure handle, CENTERFIGURE positions the figure based on the height and width of the figure and the default figure position.

If second input 'targfig' is given then CENTERFIGURE tries to place the fig centered on top of targfig.

See Also

`positionmanager`

chilimit

Purpose

Chi-squared confidence limits from sum-of-squares residuals.

Synopsis

```
[lim,scl,dof] = chilimit(ssqr,cl)
lim = chilimit(scl,dof,cl)
```

Description

`CHILIMIT` determines a confidence limit for sum-of-squares residuals, `ssqr`, by fitting the residuals to the g Chi-squared h distribution. If the sum-squared residuals are reasonably approximated by a Chi-squared distribution this gives a very good estimate of the confidence level. However, it has been observed that outliers can significantly bias the estimate.

The standard call to `CHILIMIT` uses the sum of squares residuals `ssqr`, and the optional fractional confidence level requested, *cl* {default = 0.95}. Outputs are the calculated limit `lim`, the scaling determined from the residuals `scl`, and the degrees of freedom determined from the residuals `dof`.

The scaling, `scl`, and number of degrees of freedom, `dof`, returned from a previous call to `CHILIMIT` can be used in subsequent calls to `CHILIMIT` to obtain new limits without the original residuals.

See Also

`jmlimit, pca, pcr, pls, residuallimit`

choosencomp

Purpose

GUI to select number of components from a PCA sum-of-squares captured table.

Synopsis

```
    ncomp = choosencomp(model)
```

Description

The input `model` can be a standard PCA model structure or just a sum-of-squares (SSQ) captured table from a PCA model. `CHOOSENCOMP` creates a GUI that displays the SSQ table and allows the user to select the number of principal components (`ncomp`) from the list.

The returned value, `ncomp`, is the number of selected components or an empty value `[]` if the user selected **Cancel** in the GUI.

See Also

`analysis, pca, pcaengine, simca`

class2logical

Purpose

Create a PLSDA logical block from class assignments.

Synopsis

```
[y,nonzero] = class2logical(class,groups)
```

Description

Given a list of sample classes or a DataSet object with class assignments for samples (mode 1), `CLASS2LOGICAL` creates a logical array in which each column of `y` contains the logical class membership (i.e. 1 or 0) for each class. This logical block can be used as the input `y` in PLS or PCR to perform discriminate analysis. Similarly, the output can be used with crossval to perform PLSDA cross-validation. Classes can optionally be grouped together by providing class groupings.

Inputs are `class` a list of class assignments, or a dataset with classes for first mode, and `groups` an optional input containing either:

[1 2 3 …] a vector of classes to model OR

{[1 2] [3 4] ...} a cell array containing groups of classes to consider as one class. Each cell element will be one class (see e.g. below)

Any classes in `class` which are not listed in `groups` are considered part of no group and will be assigned zero for all columns in the output.

Outputs are `y` a logical array in which each column represents one of the classes in the input class list or one of the groups in `groups` and `nonzero` the indices of samples with non-zero class assignment.

Examples

(A) Given DataSet "arch" with classes 0-5, the following creates a logical block with two columns consisting of "true" only for class 3 in the first column and "true" only for class 2 in the second column.

y = class2logical(arch,[3 2])

(B) Given DataSet "arch" with classes 0-5, the following creates a logical block with two columns consisting of "true" only for classes 0 and 1 in the first column and "true" only for classes 2 and 4 in the second column.

y = class2logical(arch,{[1 0] [2 4]})

See Also

`crossval`, `plsda`, `plsdthres`

cluster

Purpose

Agglomerative and K-means cluster analysis with dendrograms..

Synopsis

```
[results,fig] = cluster(data,labels,options)
[results,fig] = cluster(data,options)
options = cluster('options')
```

Description

`cluster(data)` performs a cluster analysis using either one of six different agglomerative methods (including K-Nearest-Neighbor (KNN), furthest neighbor, and Ward's method) or K-means clustering algorithm and plots a dendrogram. The input is `data` (class double or dataset).

Optional input *labels* can be used to put labels on the dendrogram plots. For data *M* by *N* then *labels* must be a character array with *M* rows. When *labels* is not specified and `data` is class "double", the dendrogram is plotted using sample numbers. When *labels* is not specified and `data` is class "dataset", the dendrogram is plotted using sample labels. If the `labels` field is empty it will use sample numbers.

The output is a dendrogram showing the sample distances.

Note: Calling `cluster` with no inputs starts the graphical user interface (GUI) for this analysis method.

OUTPUTS:

The outputs are (results) a structure containing results of the clustering (defined below) and the handle (fig) to any plot created. The results structure will contain the following fields:

`dist` :	the distance threshold at which each cluster forms.
`class` :	the classes of each sample (columns of class) for each distance (rows of class).
`order` :	the order of the samples which locates similar samples nearest to each other (this is the order used for the plots).
`linkage` :	a table of linkages where each row indicates a linkage of one group to another. Each row in the matrix represents one group. The first two columns indicate the sample or group numbers which were linked to form the group. The final column indicates the distance between linked items. Group numbers start at m+1 (where m is the number of samples in the input dat matrix) thus, row j of this matrix is group number m+j. This matrix can be used with the statistics toolbox dendogram function.

The (results.class) matrix can be used with the (results.dist) matrix to determine clusters of samples for any distance using:

```
results   = cluster(data);   %do cluster
ind       = max(find(results.dist<threshold));  %user-desired threshold
thisclass = results.class(ind,:);   %grab arbitrary classes
```

Options

options =	a structure array with the following fields:
`plots`:	`['none'	{'final'}]` Governs plotting. When set to 'none', the distance/cluster matrix is returned, 'final' returns a dendrogram plot showing sample distances.
`algorithm`:	`[ ]` clustering algorithm,
	'knn' {DEFAULT}: K-Nearest Neighbor
	'fn' : Furthest Neighbor
	'avgpair' : Average Paired Distance
	'med' : Median
	'cnt' : Centroid
	'ward' : Ward's Method
	'kmeans' : K-means
`preprocessing`:	`{[]}` Preprocessing structure or keyword (see PREPROCESS),
`pca`:	`[{'off'}	'on']` if 'on' then `CLUSTER` performs PCA first and clustering on the scores,
`ncomp`:	`[]` number of PCA factors to use {default = [], the user is prompted to select the number of factors from the SSQ table},
`mahalanobis`:	`[{'off'}	'on']` if 'on' then a Mahalanobis distance on the scores is used,
`slack`:	`[0]` integer number indicating how many samples can be "overridden" when two class branches merge. If the smaller of the two classes has no more than this number of samples, the branch will be absorbed into the larger class. This feature is only valid when classes are supplied in the input data. A value of 0 (zero) disables this feature.

The default options can be retreived using: `options = cluster('options');`.

See Also

`analysis, corrmap, gcluster, simca`

coadd

Purpose

Reduce resolution through combination of adjacent variables or samples.

Synopsis

```
databin = coadd(data,bins,options)
databin = coadd(data,bins,dim)
```

Description

`COADD` is used to combine ("bin") adjacent variables, samples, or slabs of a matrix. Inputs include the original array `data`, the number of elements to combine together `bins` {default: 2}, and an optional options structure *`options`*. Alternatively, the input *`options`* can be replaced with a scalar value of *`dim`* which will be used for `options.dim` (see below) and all other options will be the default values.

The mode of co-adding (defined by the options value `mode`) defines how items within each bin are combined mathematically. See options below for details.

Unpaired values at the end of the matrix are padded with the least biased value to complete the bin. Output is the co-added `data`. Unlike DERESOLV, COADD reduces the size of the data matrix by a factor of 1/bins for the dimension.

Example

Given a matrix, `data`, size 300 by 1000, the following would coadd variables in groups of three:

```
databin = coadd(data,3);
```

and the following would coadd samples in groups of two:

```
options.dim = 1;
databin = coadd(data,2,options);
```

The following is equivalent to the previous two lines using the "shortcut" input of `dim`.

```
databin = coadd(data,2,1);
```

Options

`dim`:	Dimension in which to do combination {default = 2},
`mode`:	['sum' | {'mean'} | 'prod'] method of combination. See algorithm notes for details of these modes.

Algorithm

The three modes, sum, mean and prod behave according to the following (described in terms of variables):

SUM: groups of variables are added together and stored. The resulting values will be larger in magnitude than the original values by a factor equal to the number of variables binned.

MEAN: groups of variables are added together and that sum is divided by the number of variables binned. The resulting values will be similar in magnitude to the original values.

PROD: groups of variables are multiplied together.

See Also

`deresolv`

coda_dw

Purpose

Variable selection method for hyphenated methods with a mass spectropmeter as a detector. The variables (mass chromatograms) are selected on the basis of smoothness.

Synopsis

```
    [dw_value,dw_index] = coda_dw(data,level);
```

Description

CODA_DW the Durbin Watson values of the first derivative of the chromatograms in data. The optional argument level defines the limitit of Durbin Watson value used for a plot of the results. If level is an integer it is used to plot the best level chromatograms. Low values for Durbin Watson indicate good quality chromatograms. The Durbin Watson values (`dw_values`) as wel as their ranking indices (`dw_index`) (low to high, so good to low quality) . For more information the Durbin Watson method see the function DURBIN_WATSON. Input `data` can be a matrix with the data or a datasetobject

Examples

Plotting the chromatograms with a Durbin Watson value less than 2.2.

```
    coda_dw(data,2.2);
```

Plotting the best 40 chromatograms.

```
    coda_dw(data,40);
```

Algorithm

The algorithm calculates the Durbin Watson values of the first derivative of the mass chromatograms.

See Also

`durbin_watson`

comparelcms_sim_interactive

Purpose

Select variables that are different between related data sets, e.g. mass chromatograms from LC/MS data of different batches.

Synopsis

```
comparelcms_sim_interactive
```

Description

`COMPARELCMS_SIM_INTERACTIVE` Performs the variable (mass chromatogram) selection using `comparelcms_simengine`, but with added interactivity

See Also

`comparelcms_simengine`

comparelcms_simengine

Purpose

Select variables that are different between related data sets, e.g. mass chromatograms from LC/MS data of different batches.

Synopsis

```
y=comparelcms_simengine(data,filter_width)
```

Description

`COMPARELCMS_SIMENGINE` determines which variables are different between different data sets. For example, after applying `coda_dw` to LC/MS data sets of highly related samples, such as the data of a good and a bad batch, the results will be very similar. `comparelcms_engine` takes the next step and extracts the mass chromatograms that are different. This function is normally not called by itself but by the function `comparelcms_sim_interactive`. The input argument `data` is a data cube with mode 1 the number of samples, mode two the number of spectra and mode 3 the number of variables, The optional input argument `filter_width` is used to smooth the columns of the data set in order to minimize the effect of small shifts, The output argument `y` contains the similarity indices of the variables. Variables with a low similarity index show the differences between the data sets.

Examples

Determination of similarity indices with a filter of 7 data points.

```
y=comparelcms_simengine(data,7)
```

Algorithm

The calculations are based on a similarity index of the minimum of the chromatograms (across the samples) and the maximum of the chromatograms.

See Also

`comparelcms_sim_interactive`

comparevars

Purpose

Compares two variables of any type and returns differences.

Synopsis

```
[status,msg] = comparevars(a,b,options)
```

Description

Given any two variables `a` and `b`, `COMPAREVARS` looks for any differences. This function operates on any standard Matlab data type or a DataSet object and does not give an error when variables of two different types are passed..

With no outputs, the differences between the variables (or "None Found") is displayed. With one output, the boolean result of the comparison `status` is returned (1 = variables are completely equivalent). With two outputs, the comparison result is returned and a cell array of strings is returned listing the differences as a description `msg`.

Options

`ignoreclass` :	`{}` Cell array of classes which should be ignored during the comparison. If a structure or cell contains any objects of these classes, the values will not be compared. NOTE: any numeric class (double, uint8, single) should be referred to as 'numeric' to ignore comparisons.
`ignorefield` :	`{}` Specifies one or more structure fields which should be ignored (not compared) in any structure.
`missingfield` :	`[ 'ignore' | {'difference'} ]` specifies how to handle when one of two input structures does not contain the same fields as the other. 'ignore' simply ignores missing fields. 'difference' returns this mismatch as a noted difference.

See Also

`cellne`

compressmodel

Purpose

Remove references to unused variables from a model.

Synopsis

```
[cmodel,msg] = compressmodel(model)
```

Description

COMPRESSMODEL will remove any references in a model to excluded variables. This permits the application of the model to new data in which unused variables have been hard-excluded (i.e. previously removed or not collected). Input is `model` the model to compress. Outputs are `cmodel` the compressed model and `msg` any warning messages reported during compression. Although compression will work on most models, some preprocessing methods and some model types may not compress correctly. In these cases, a warning will be given and reported in the output `msg`.

See Also

`pca, pcr, pls, plsda`

conload

Purpose

Congruence loadings for PARAFAC, TUCKER and NPLS.

Synopsis

```
Bcon = conload(X,model,options)
```

Description

Determines congruence (earlier known as correlation) loadings for a specific mode of a model. Congruence loadings look at "non-average correlations", hence take differences in offset into account.

Note that due to non-orthogonal loadings in PARAFAC, individual correlations can add to more than 1. Therefore, such loadings are not drawn with ellipses but squares added. Use options.force = 'ellipse' or 'square' to force one or the other on the plot.

INPUTS:

X =	modeled data
model =	standard model structure
mode =	loading mode to investigate (i.e. mode = 1 for samples if they are in the first mode)

OUTPUTS:

Bcon = Congruence loadings

Options

`plots` :	`['none'	{'final'}]` Governs the creation of plot of the results.
`force` :	`[{'off'}	'ellipse'	'square']` Forces a given type of limit on the plots (if plot is given).

See Also

`npls, parafac, tucker`

copydsfields

Purpose

Copies informational fields between datasets and/or model structures.

Synopsis

```
to = copydsfields(from,to,modes,block)
```

Description

Copies all informational fields from one dataset to another, one model structure to another, or between datasets and models. This function copies the fields: `label`, `class`, `classlookup`, `title`, `axisscale`, and `includ` as well as the "`<field>name`" assosciated with each (e.g. `classname`). If copying to or from a model structure, the fields to be copied from/to are sub-fields of the `detail` field.

INPUTS:

`from` =	dataset or model from which fields should be copied, and
`to` =	dataset or model to which fields should be copied.

OPTIONAL INPUTS:

`modes` =	modes (dims) which should be copied {default: all modes}. (modes) can be a cell of {[from_modes] [to_modes]} to allow cross-mode copying.
`block` =	data block of model from/to which information should be copied. Default: block 1. Can also be a cell of {[from_modes] [to_block]} to allow cross-block copying. This setting has noeffect with two DataSet objects.Output is: `to`, the updated dataset or model.

OUTPUT:

to =	the updated dataset or model.

Examples

```
mydataset2 = copydsfields(mydataset1, mydataset2);
```

copies all fields for all modes of `mydataset1` into `mydataset2` (copies set 1 only).

```
mydataset2 = copydsfields(modl, mydataset2, {2 1});
```

copies all fields from mode 2 (variables) of `modl` into mode 1 of `mydataset2`.

```
modl = copydsfields(mydataset,modl,1,{1 2});
```

copies all fields for mode 1 (samples) from set 1 of `mydataset` into block 2 (e.g. y-block) of `modl`.

See Also

`dataset/dataset`, `modelstruct`, `pca`, `pcr`, `pls`

corcondia

Purpose

Evaluate consistency of PARAFAC model.

Synopsis

```
CoreConsist = corcondia(X,loads,Weights,plots);
```

Description

`PARAFAC` can be written as a special Tucker3 model where the core is superdiagonal with ones on the diagonal. This special way of writing the model can be used to check the adequacy of a PARAFAC model by estimating what Tucker3 core is found if estimated unconstrained from the PARAFAC loadings. The core consistency is given as the percentage of variation in this core array consistent with the theoretical superdiagonal array. The maximum core consistency is thus 100Found. Consistencies well below 70-90Found indicate that either too many components are used or the model is otherwise mis-specified. The consistency can also become negative which means that the model is not reasonable. Note that core consistency is an ad hoc method. It often works well on real data, but not as well with simulated data. `CORCONDIA` does not provide proof of dimensionality, but it can give a good indication.

Inputs are the multi-way array `X` and `loads` which can be a) a cell array with PARAFAC model loadings or b) a PARAFAC model structure.

Optional inputs are *Weights* which can be used to update the core in a weighted least squares sence and *plots* which suppress plotting of the results when set to zero (0).

See Also

`corecalc, parafac, tucker`

coreanal

Purpose

Evaluate, display, and rotate core from a Tucker model.

Synopsis

```
      result = coreanal(core,action,param)
```

Description

Performs an analysis of the input core array of a Tucker model `core`. Results are returned in the output `result`.

Optional input *action* is a text string used to customize the analysis.

`action = 'list'`, the output `result` contains text describing the main properties of the core. If `coreanal` is called without outputs, the text is printed to the command window. If optional input *param* is included, the number of core entries shown can be controlled.

`action = 'plot'`, the core array is plotted and output `result` is not assigned.

`action = 'maxvar'`, Rotates the core to maximum variance. This is the same as maximum simplicity as defined by Andersson & Henrion, Chemometrics & Intelligent Laboratory Systems, 1999,47,189-204.The output `result` is a structure array containing the rotated core in the field `core` and the rotation matrices to achieve this rotation in the field `transformation`.

The loadings of the Tucker model should also be rotated correspondingly which can also be done using `coreanal`.

Examples

```
      result = coreanal(model,'list');
      result = coreanal(model.core,'list');
```

will list information on the core-entries (explained variance etc).

```
      result = coreanal(model.core,'list',10);
      coreanal(model.core,'list',10);
```

will do the same but only for the ten most significant core-entries with the second version (with no output) printing the information to the command window.

```
      result = coreanal(model,'plot');
```

will make a plot of the core where the size of each core-entry shows the variance explained. If the core is of higher order than three, it is first rearranged to a three-way array.

```
rotatedcore = coreanal(model,'maxvar');
```

will rotate the core to maximal variance.

```
rotatedmodel = coreanal(oldmodel,rotatedcore);
```

where the input `oldmodel` is the original Tucker model structure and `rotatedcore` is the output from above. The rotation can be achieved in one step using:

```
rotatedmodel = coreanal(oldmodel,coreanal(oldmodel,'maxvar'));
```

See Also

`corecalc`, `tucker`

corecalc

Purpose

Calculates the Tucker3 core array given the data array and loadings

Synopsis

```
    Core = corecalc(X,loads,orth,Weights,OldCore);
```

Description

Caculates the core array given the data `X` and the loadings `loads` (component matrices) which are held in a cell (see `TUCKER`).

Optional input *orth* is set to 0 to tell `CORECALC` that the loadings are NOT orthogonal.

Optional input *Weights* allows a weighted least squares solution to be sought.

Optional input *OldCore* provides a prior estimate of the core to speed up calculations.

The output `Core` is the Tucker3 core.

See Also

`corcondia, coreanal, parafac, tucker`

corrmap

Purpose

Correlation map with variable regrouping.

Synopsis

```
order = corrmap(data,labels,reord)
order = corrmap(data,reord)
```

Description

`CORRMAP` produces a pseudocolor map that shows the correlation between variables (columns) in a data set. The function will reorder the variables by KNN clustering if desired.

The input is the data `data` class "double" or "dataset".

Optional input *labels* contains the variable labels when the data is class "double".

Optional input *reord* will cause `CORRMAP` to keep the original ordering of the variables if set to 0.

The output `order` is a vector of indices with the variable ordering.

`corrmap(data,labels)` produces a psuedocolor correlation map with variable reordering.

`corrmap(data,labels,0)` produces a psuedocolor correlation map without variable reordering.

See Also

`autocor, crosscor`

corrspec

Purpose

Resolves correlation spectroscopy maps.

Synopsis

```
[model] = corrspec(xspec,yspec,ncomp,options)
[purintx,purinty,purspecx,purspecy,maps] =
   corrspec(xspec,yspec,idex,options)
[purintx,purinty,purspecx,purspecy,maps] =
   corrspec(xspec,yspec,model,options)
```

Description

CORRSPEC resolves a correlation map of two spectroscopies into the maps of individual components, their associated resolved spectra and the contributions ("concentrations") of the components in the original mixture spectra.

INPUTS:

`xspec`	: (2-way array class "double" or "dataset") x-matrix for dispersion matrix.
`yspec`	: (2-way array class "double" or "dataset") y-matrix for dispersion matrix.
`ncomp`	: (scalar or n x 2 matrix) if ncomp = scalar then function will calculate first n resolved pure purity components. If ncomp = n x 2 matrix, each row indicates the x and y position (index) to calculate the purity solution. If empty, the initial matrices will be calculated.

OUTPUTS:

`purintx`	: resolved x contributions('concentrations').
`purinty`	: resolved y contributions('concentrations').
`purspecx`	: resolved x pure component spectra.
`purspecy`	: resolved y pure component spectra.
`map`	: cell with ncomp resolved dispersion matrixes, each with size: size(yspec,2)*size(xspec,2)
`model`	: standard model structure, used for prediction (same pure variables on other data set) and add components to the model. The series of correlation maps resulting from the sequential elimination of components is stored in the field `detail.matrix`. See function `corrspecengine` for detailed description of matrix. The series of resolved correlation maps is stored in field `detail.maps`. Once a model has been calculated it can be used to predict x spectra from y spectra and vice versa.

Options

`plots_spectra`	: ['off'|{'on'}] governs level of plotting for spectra.
`plots_maps`	: ['off'|{'on'}] governs level of plotting for maps.
`offset`	: noise correction factor. One element defines offset for both x and y, two elements separately for x and y.
`inactivate`	: [ ] logical matrix of indices not to be used in purity calculation.
`dispersion`	: [1] See max (below).
`max`	: [3] If not given, only weight matrix will be calculated, otherwise select one of the options below: 1: standardized, offset corrected 2: length sqrt(nrows), offset corrected 3: purity about mean, offset corrected 4: purity about origin, offset corrected 5: asynchronous, offset corrected

Examples

```
load data_mid_IR
load data_near_IR
corrspec(data_mid_IR,data_near_IR,4)
```

See Also

`corrspecengine, dispmat, purity`

corrspecengine

Purpose

This function is the primary calculational engine for the function corrspec. It calculates the correlation maps and related matrices corrected for previously determined pure variables.

Synopsis

```
matrix = corrspecengine(data_x,data_y,purvar_index,offset,
  matrix_options);
```

Description

Calculates the matrices (weigh matrix, dispersion matrix and max matrix) needed for corrspec corrected for previously determined pure variables.

INPUTS:

`data_x`	: (2-way array class "double" or "dataset") x-matrix for dispersion matrix.
`data_y`	: (2-way array class "double" or "dataset") y-matrix for dispersion matrix.
`purvar_index`	: indices of maximum value in purity_values, i.e. the index of the pure variables. First column for x data, second column for y data. Empty when no pure variables have been chosen yet. When base_x is a single number n, the program calculates the first n pure purity_indices.
`offset`	: noise correction factor. One element defines offset for both x and y, two elements separately for x and y.
`max`	: if not given, only weight matrix will be calculated, otherwise it contains 2 elements: the options the dispersion_matrix and the max_matrix: 1: standardized, offset corrected 2: length sqrt(nrows), offset corrected 3: purity about mean, offset corrected 4: purity about origin, offset corrected 5: asynchronous, offset corrected

OUTPUTS:

`matrix`	: cell array with either one or three matrices, with size [ncols_y ncols_x] (ncols_y represents number of spectra in y, etc.). matrix{1}: weight_matrix, matrix used to correct for previously selected pure variables.

matrix{2}: dispersion_matrix, matrix of interest, generally correlation matrix, corrected for previously selected pure variables.

matrix{3}: max_matrix, matrix from which pure variables are chosen, generally a co-purity matrix corrected for previously selected pure variables.

See Also

`corrspec, dispmat`

cr

Purpose

Continuum regression for multivariate y.

Synopsis

```
    b = cr(x,y,lv,powers)
```

Description

`CR` develops continuum regression models for a matrix of predictor variables (x-block) `x`, and vector or matrix of predicted variables (y-block) `y`. Models are calculated for 1 to `lv` latent variables for each value of the continuum parameter specified in the row vector `powers`. The output is the matrix of regression vectors `b`.

For a y-block with `ny` variables, x-block with `nx` variables, and `np` powers (size of `powers` is `1` by `np`) `b` is size (`lv*ny*np`) by `nx`. The first block in `b` corresponds to the first power in `powers` and is (`lv*ny`) by `nx` with the first row corresponding to a `1` latent variable model for the first `y` variable.

`CR` uses the de Jong, Wise & Ricker method for continuum regression (S. de Jong, B. M. Wise and N. L. Ricker, "Canonical Partial Least Squares and Continuum Power Regression," *J. Chemo.*, **15**, 85-100, 2001). It is a drastically faster implementation of the Wise and Ricker method used in the previous `powerpls`. Note that results are identical for both methods for the univariate `y` case but not for the multivariate `y`, where the results from `CR` are typically slightly better.

The algorithm used here is usually stable up to a continuum parameter of about 6-8, sometimes as high as 10 depending upon the problem. At powers this high, however, the models have essentially converged to the PCR solution. No instabilities at small powers have been noted.

See Also

`crcvrnd, pcr, pls`

crcvrnd

Purpose

Cross-validation for continuum regression models using SDEP.

Synopsis

```
[press,fiterr,mlvp,b] = crcvrnd(x,y,splt,itr,lv,pwrs,ss,mc)
```

Description

`crcvrnd` is used to cross-validate continuum regression models given a matrix of predictor variables (x-block) `x`, matrix or vector of predicted variables (y-block) `y`, the number of divisions into which to split the data `splt`, the number of iterations of the cross-validation procedure using different re-orderings of the data set `itr`, maximum number of latent variables `lv` and the row vector of continuum regression parameters to consider `powers`.

The outputs are the predictive residual error sum of squares (PRESS) matrix `press` where each element of the matrix represents the PRESS for a given combination of LVs and continuum parameter, the corresponding fit error `fiterr`, the number of LVs and power at minimum PRESS `mlvp` and the final regression vector or matrix `b`.

The optional input *ss* causes the routine to choose contiguous blocks of data during cross-validation when set to 1. If the optional input *mc* is set to 0 the subsets are not mean-centered during cross-validation.

A good smooth PRESS surface can usuall be obtained by calculating about 20 models spaced logarithmically between 4 and 1/4 and using 10 to 30 iterations of the cross-validation. A good rule of thumb for dividing the data is to use either the square root of the number of samples or 10, which ever is smaller.

See Also

`cr, pcr, pls`

crosscor

Purpose

Calculates the crosscorrelation function of two time series.

Synopsis

```
crcor = crosscor(x,y,n,period,flag,plots)
```

Description

`crcor = crosscor(x,y,n)` returns the crosscorrelation function `crcor` of two time series `x` and `y` for a maximum time shift of `n` sample periods.

`crcor = crosscor(x,y,n,`*period*`)` uses the sampling period `period` to scale the x-axis on the output plot.

`crcor = crosscor(x,y,n,`*period*,*flag*`)` with `flag` set to 1 changes the routine from cross correlation to cross covariance.

Optional input *plots* suppresses plotting when set to 0.

See Also

`autocor, corrmap, wrtpulse`

crossval

Purpose

Cross-validation for PCA, PLS, MLR, and PCR.

Synopsis

```
results = crossval(x,y,rm,cvi,ncomp,options)
[press,cumpress,rmsecv,rmsec,cvpred,misclassed] =
   crossval(x,y,rm,cvi,ncomp,options)
```

Description

`CROSSVAL` performs cross-validation for linear regression (PCR, PLS, MLR, CorrelationPCR, and Locally Weighted Regression) and principal components analysis (PCA). Inputs are the predictor variable matrix `x`, predicted variable `y` (`y` is empty [] for `rm = 'pca'`), regression method `rm`, cross-validation method `cvi`, and maximum number of latent variables / components `ncomp`.

`rm = 'pca'` performs cross-validation for PCA,
`rm = 'mlr'` performs cross-validation for MLR,
`rm = 'pcr'` performs cross-validation for PCR,
`rm = 'nip'` performs cross-validation for PLS using NIPALS,
`rm = 'sim'` or `'pls'` performs cross-validation for PLS using SIMPLS,
`rm = 'correlationpcr'` performs cross-validation for CorrelationPCR, and
`rm = 'lwr'` performs cross-validation for Locally Weighted Regression (see `LWRPRED`).

`cvi` can be 1) a cell containing one of the cross-validation methods below with the appropriate parameters `{method splits iterations}`, or 2) a vector representing user-defined cross-validation groups.

`loo` :	leave one out cross-validation (each sample left out on its own; does not take splits or iterations as inputs),
`vet` :	`{splits}` venetian blinds (every n-th sample together),
`con` :	`{splits}` contiguous blocks, and
`rnd` :	`{splits iter}` random subsets.

Except for leave-one-out, all methods require the number of data splits `splits` to be provided. Random data subsets (`'rnd'`) also requires number of iterations `iter` where "iterations" defines the number of replicate splits to perform. For 'con' and 'vet', iterations randomly moves the starting point for the first (and subsequent) blocks.

E.g. cvi = {'con' 5}; for 5 contiguous blocks (one iteration)

For user-defined cross-validation, `cvi` is a vector with the same number of elements as x has rows (i.e. `length(cvi) = size(x,1);` when `x` is class “double”, or `length(cvi) = size(x.data,1);` when `x` is class “dataset”) with integer elements, defining test subsets. Each `cvi(i)` is defined as:

`cvi(i) = -2` the sample is always in the test set,
`cvi(i) = -1` the sample is always in the calibration set,
`cvi(i) =  0` the sample is always never used, and
`cvi(i) =  1,2,3…` defines each subset.

Options

Optional input *options* is an options structure containing one or more of the following fields:

`display:`	`['off'	{'on'}]` Governs output to command window,
`plots:`	`['none'	{'final'}]` Governs plotting,
`preprocessing:`	`{[1]}` Controls preprocessing. Default is mean centering (1). Can be input in two ways: a) As a single value: 0 = none, 1 = mean centering, 2 = autoscaling, or b) As {xp yp}, a cell array containing a preprocessing structure(s) for the X- and Y-blocks (see PREPROCESS). E.g. `pre = {xp []};` for PCA. To include preprocessing of each subset use `pre = {xp yp};` or `pre = {xp []}` for PCA. To avoid preprocessing of each subset use `pre = {[] []};` or `pre = 0` (zero).
`threshold:`	`{[]}` Alternative PLSDA threshold level (default = [] = automatic)
`prior:`	`{[]}` Used with PLSDA only. Vector of fractional prior probabilities. This is the probability (0-1) of observing a "1" for each column of y (i.e. each class). E.g. [.25 .50] defines that only 25Found and 50Found of future samples will likely be "true" for the classes identified by columns 1 and 2 of the y-block. [] (Empty) = equal priors.
`structureoutput:`	`[{'no'}	'yes']` Governs output variables. 'Yes' returns a structure instead of individual variables. 'Yes' is default if only one output is requested.
`jackknife:`	`[{'no'}	'yes']` Governs storing of jackknifed regression vectors. Jack-knifing may slow performance significantly or cause out-of-memory errors when both x and y blocks have many variables.
`rmsec:`	`['no'	{'yes'}]` Governs calculation of RMSEC. When set to 'no', calculation of "all variables" model is skipped (unless specifically required for plots or requested with multiple outputs)
`pcacvi:`	`{'loo'}` Cell describing how PCA cross-validation should perform variable replacement. Variable replacement options are similar to cross-validation CVI options and include: {'loo'} leave one variable out at a time {'con' splits} contiguous blocks (total of splits groups) {'vet' splits} venetian blinds (every n'th variable), or

{'rnd' splits} random subsets (note: no iterations)

`fastpca:` `[ 'off' | {'auto'} ]` Governs use of "fast" PCA Cross-validation algorithm. 'off' never uses fast algorithm, 'auto' uses fast algorithm when other options permit. Fast pca can only be used with pcacvi set to 'loo'

`lwr:` Sub-structure of options to use for locally-weighted regression cross-validation. Most of these options are used as defined in the LWRPRED function (see LWRPRED for more details) but there are two additional options defined for cross-validation:

`lwr.minimumpts : [20]` the minimum number of points (samples) to use in any LWR sub-model.

`lwr.ptsperterm : [20]` the number of points to use per term (LV) in the LWR model. For example, when set to 20, 20 samples will be use for a 1 LV model, 40 samples will be used for a 2 LV model, etc. If set to zero, the number of points defined by lwr.minimumpts will be used for all models - that is, the number of samples used will be independent from the number of LVs in the model.

In all cases, the number of samples in an individual test set will be the upper limit of samples to include in any LWR prediction.

Output:

`press:` predictive residual error sum of squares PRESS for each subset (subsets are rows of this matrix, number of components are columns)

`cumpress:` cumulative PRESS (sum of columns of press).

`rmsecv:` root mean square error of cross-validation.

`rmsec:` root mean square error of calibration.

`cvpred:` cross-validation y-predictions (regression methods only). If cross-validation method was random, this is the average prediction of all replicates.

`misclassed:` fractional misclassifications for each class (valid for regression methods only and only when y is a logical, (i.e. discrete-value) vector.

`reg:` jack-knifed regression vectors from each sub-set. This will be size [k*ny nx splits] such that reg(1,:,:) will be the regression vectors for 1 component model of the first column of y for all sub sets (a 1 by nx by splits matrix). Use squeeze to reduce to an nx by splits matrix. (note: options.jackknife must be 'yes' to use reg)

If options.structureoutput is 'yes', a single output (`results`) will return all the above outputs as fields in a structure. If `options.rmsec` is 'no', then RMSEC is not returned (provides faster iterative calculation)

Note that for multivariate (y) the output (press) is grouped by output variable, i.e. all of the PRESS values for the first variable are followed by all of the PRESS values for the second variable, etc.

When `options.plots` is not ‘none’ plots both RMSECV and RMSEC are provided.

Examples

```
[press,cumpress] = crossval(x,y,'nip',{'loo'},10);
[press,cumpress] = crossval(x,y,'pcr',{'vet',3},10);
[press,cumpress] = crossval(x,y,'nip',{'con',5},10);
[press,cumpress] = crossval(x,y,'sim',{'rnd',3,20},10);
res = crossval(x,y,'sim',{'rnd',3,20},10);

pre = {preprocess('autoscale') preprocess('autoscale')};
opts.preprocessing = pre;
opts.plots = ‘none’;
[press,cumpress] = crossval(x,y,'sim',{'rnd',3,20},10,opts);
res = crossval(x,y,'sim',{'rnd',3,20},10,opts);

[press,cumpress] = crossval(x,[],'pca',{'loo'},10);
[press,cumpress] = crossval(x,[],'pca',{'vet',3},10);
res = crossval(x,[],'pca',{'con',5},10);
```

See Also

`encodemethod, pca, pcr, pls, preprocess, ncrossval, ncrossval`

datahat

Purpose

Calculates the model estimate and residuals of the data.

Synopsis

```
    xhat = datahat(model);
    [xhat,resids] = datahat(model,data);
```

Description

Given a standard model structure `model` DATAHAT computes the model estimate of the data `xhat`. For example, if `model` is a PCA model of a matrix $\mathbf{X}_{cal}$ such that $\mathbf{X}_{cal} = \mathbf{TP}^T + \mathbf{E}$, then $\mathbf{X}_{hat} = \mathbf{TP}^T$. (i.e. $\mathbf{X}_{cal} = \mathbf{TP}^T + \mathbf{E} = \mathbf{X}_{hat} + \mathbf{E}$).

If optional input *data* is supplied then `DATAHAT` computes the model estimate of *data* that is output in `xhat`. For the PCA model of matrix $\mathbf{X}_{cal}$, and *data* is a data matrix $\mathbf{X}_{new}$ then $\mathbf{X}_{hat} = \mathbf{X}_{new}\mathbf{PP}^T = \mathbf{T}_{new}\mathbf{P}^T$. The output `resids` is a matrix with the corresponding residuals $\mathbf{E}$ [$\mathbf{E} = \mathbf{X}_{new}\text{-}\mathbf{X}_{new}\mathbf{PP}^T = \mathbf{X}_{new}(\mathbf{I}\text{-}\mathbf{PP}^T)$]. If *data* is $\mathbf{X}_{cal}$ then $\mathbf{X}_{hat} = \mathbf{TP}^T$ and `resids` is $\mathbf{E} = \mathbf{X}_{cal}(\mathbf{I}\text{-}\mathbf{PP}^T)$].

Note that preprocessing in `model` will be performed before the residuals are calculated. If `data` is not provided, only `xhat` is available.

Note that `DATAHAT` works with almost all standard model structures.

See Also

`analysis, parafac`

datasetdemo

Purpose

Demonstrates use of the dataset object.

Synopsis

```
datasetdemo
```

Description

This demonstration illustrates the creation and manipulation of dataset objects. Functions that are demonstrated include: DATASET, GET, SET, ISA, and EXPLODE.

For more information see help on DATASET, DATASET/SET, DATASET/GET, and DATASET/EXPLODE.

See Also

`editds, plotgui`

delsamps

Purpose

Delete samples (rows) from data matrices.

Synopsis

```
eddata = delsamps(data,samps)
eddata = delsamps(data',vars)'
```

Description

`eddata = delsamps(data,samps)` deletes `samps` row numbers (samples) from a data matrix `data` and saves the edited results to data matrix `eddata`.

`eddata = delsamps(data',vars)'` deletes `vars` column numbers (variables) from a data matrix `data` and saves the edited results to data matrix `eddata`.

See Also

`shuffle, specedit`

demos

Purpose

Demo list for the PLS_Toolbox.

Synopsis

```
demos
```

Description

`DEMOS` brings up the Matlab help browser with a list of functions that have demonstration scripts. Clicking on a listed function will display a brief description and information about the function. Along with the description are highlighted text that, when clicked, will run the demo, connect to related information, or open the function in the mfile editor.

See Also

`helppls`

deresolv

Purpose

Changes high resolution spectra to low resolution.

Synopsis

```
      lrspec = deresolv(hrspec,a)
```

Description

`DERESOLV` uses a FFT to convolve spectra with a resolution function to make it appear as if it had been taken on a lower resolution instrument. Inputs are the high resolution spectra to be de-resolved `hrspec` and the number of channels to convolve them over `a`.

The output is the estimate of the lower resolution spectra `lrspec`.

`deresolv` is useful for standardizing two instruments of different resolution. It can also be used to smooth spectra.

See Also

`baseline, savgol, stdfir, stdgen`

discrimprob

Purpose

Calculate discriminate probabilities of discrete classes for continuous predicted values.

Synopsis

```
[prob,classes] = discrimprob(y,ypred,prior)
```

Description

`DISCRIMPROB` examines the predictions of a PLS-D model (PLS-D models are trained on a standard x-block but with a y-block containing discrete class assignments for each sample). The predicted y-value from the PLS-D model will be a continuous variable that can be interpreted as a class similarity index. `DISCRIMPROB` uses the actual class asignments and the model y-value predictions to create a probability table that indicates, for a given predicted y-value, the probability that the given value belongs to each of the original classes.

Inputs are `y` the original logical classes for each sample, `ypred` the observed continuous predicted values for those samples and `prior` an optional input of the prior probabilities for each class. `prior` should be a vector representing the probabitily of observing each class in the entire population. Default prior probabilities is 1.

Output prob is a lookup matrix consisting of an index of observed y-values in the first column, and the probability of that value being of each class in the subsequent columns. The second output `classes` is the discrete classes observed in `y`, corresponding to the additional columns of `prob`.

To predict a probability that the observed value `ypred` is in class `classes(n)` use:

```
classprob = interp1(prob(:,1),prob(:,n+1),ypred)
```

See Also

`pls, plsdthres, simca`

dispmat

Purpose

Calculates the dispersion matrix of two spectral data sets.

Synopsis

```
[c,meansx,meansy,stdsx,stdsy] = dispmat(x,y,options);
```

Description

Calculates a dispersion matrix, as defined by the options, of datasets x and y.

INPUTS:

`x`	: (2-way array class "double" or "dataset") x-matrix for dispersion matrix.
`y`	: (2-way array class "double" or "dataset") y-matrix for dispersion matrix.

OUTPUTS:

`c`	: dispersion matrix, as defined by options.
`meansx`	: mean of x.
`meansy`	: mean of y.
`stdsx`	: standard deviation of x.
`stdsy`	: standard deviation of y.

Options

`offsetx`	: [0] offset for x.
`offsety`	: [0] offset for y.
`dispersion`	: [1] dispersion matrix calculated: 1: standardized, offset corrected 2: length sqrt(nrows), offset corrected 3: purity about mean, offset corrected 4: purity about origin, offset corrected 5: asynchronous, offset corrected

See Also

`corrspec, corrspecengine, purity`

distslct

Purpose

Select samples on the exterior of a data space based on a Euclidean distance.

Synopsis

```
isel = distslct(x,nosamps,flag)
```

Description

`DISTSLCT` first identifies a sample in the *M* by *N* data set `x` furthest from the data set mean. Subsequent samples are selected to be simultaneously the furthest from the mean and the selected samples for a total of `nosamps` selected samples. `DISTSLCT` calls `STDSSLCT` to find the number of samples up to the rank of the data and uses a distance measure to find additional samples if `nosamps>rank(x)`.

Optional intput tells `DISTSLCT` how many samples `STDSLCT` should estimate when `nosamps>`*N*:

1 =	`STDSLCT` selectes *N*-1, or
2 =	`STDSLCT` selects *N* {default}.

Output `isel` is a vector of length `nosamps` containing the indices of the selected samples.

This routine is used to initialize the selection of samples in the `DOPTIMAL` function. Altough it does not satisfy the d-optimality condition, it is an alternative to doptimal that does not require an inverse or calculation of a determinant.

See Also

`doptimal, stdsslct`

doptimal

Purpose

Selects samples from a candidate matrix that satisfy the d-optimal condition.

Synopsis

```
isel = doptimal(x,nosamps,iint,tol)
```

Description

`DOPTIMAL` selects a number (`nosamps`) of samples from a candidate matrix `x` that maximizes the determinant of `det(x(isel,:)'*x(isel,:))` where `isel` is a vector of indices of the selected samples.

The optional input *iint* is a vector of indices to initialize the optimization algorithm. If *iint* is not input the algorithm is initialized using samples identified as on the exterior of the data set using the `DISTSLCT` function. This is in contrast to initializing with a random subset used in many algorithms. The reason is that the routine is based on Fedorov's algorithm (de Aguiar, P.F., Bourguignon, B., Khots, M.S., Massart, D.L., and Phan-Than-Luu, R., “D-optimal designs”, *Chemo. Intell. Lab. Sys.*, **30**, 199–210, 1995) which requires calculating `inv(x(isel,:)'*x(isel,:))`, and it is possible that the inverse of a random set will not exist. The routine then exchanges the 'least informative' sample in the selected set with a 'more informative' sample in the candidate set. The optional input *tol* sets the tolerance for minimum increase in the determinant {default = $1x10^{-4}$}.

Note that `nosamps` must be ≥ `rank(x)` (it is necessary but not sufficient that `nosamps` ≥ `size(x,2)`) for a good solution to be found. This is required so that a good estimate of `inv(x(isel,:)'*x(isel,:))` can be obtained. When `nosamps < size(x,2)` the scores from PCA or PLS can be used where `nosamps` ≥ than the number of factors (principal components or latent variables) used. Also, note that the solution can depend on the initial guess and that `isel` does not necessarily represent a global optimum.

Examples

For an input matrix `x` that is m by 5

```
isel5 = doptimal(x,5);
isel6 = doptimal(x,6);
```

See Also

`distslct, stdsslct`

dp

Purpose

Adds a diagonal line at 45 degrees (slope of 1) to the current plot

Synopsis

```
h = dp(lc, flag)
```

Description

`DP` can be used to add a line of perfect prediction to plots of actual versus predicted values. Optional input *lc* can be used to change the line style as in normal plotting (*e.g.* `lc = 'b'`). Returns handle of line object.

See Also

`ellps, hline, plttern, vline, zline`

durbin_watson

Purpose

Criterion for measure of continuity.

Synopsis

```
y = durbin_watson(x)
```

Description

The durbin watson criteria for the columns of x are calculated as the ratio of the sum of the first derivative of a vector to the sum of the vector itself. Low values means correlation in variables, high values indicates randomness. Input `x` is a column vector or array in which each column represents a vector of interest. Output `y` is a scalar or vector of Durbin Watson measures.

See Also

`coda_dw`

editds

Purpose

Editor for DataSet Objects.

Synopsis

```
editds(dataset)
editds(command,fig,auxdata)
```

Description

`EDITDS` is a graphical user interface (GUI) for creating and editing dataset objects. Typing `editds` at the command line with no inputs will display the GUI. To create a new dataset, select New… from the File menu. Calling it with a dataset will display that dataset in a new GUI.

Use menu items to perform common tasks such as Saving and Including/Excluding data. Many of these tasks can also be performed graphically by clicking on the appropriate tab and editing the given control. Most heading controls have mouse-over tool tips to further help identify a particular control or column.

Data can also be plotted from the dataset editor via the View > Plot menu item or using the plot icon on the left side of the Info tab. Data can be edited directly via the Data tab and Variable labels and information can be manipulated vie their respective tabs.

See Also

`plotgui`

ellps

Purpose

Plots an ellipse on an existing figure.

Synopsis

```
ellps(cnt,a,lc,ang,pax,zh)
```

Description

`ELLPS` plots an ellipse on an existing figure e.g. an ellipse of constant Hotelling's T^2. The inputs are a 2 element vector containing the ellipse center `cnt`, and a 2 element vector containing the ellipse axes sizes `a`. Optional inputs are *lc* which defines the line color (*e.g.* `'-g'`), and *ang* which defines the angle of rotation from the x-axis {default: `ang = 0` radians}.

`ellps([4 5],[3 1.5],':g')` plots a dotted green ellipse with center (4,5), semimajor axis 3 parallel to the x-axis and semiminor 1.5 parallel to the y-axis.

Optional inputs *pax* and *zh* are used when plotting in a 3D figure. *pax* defines the axis perpindicular to the plane of the ellipse [1 = x-axis, 2 = y-axis, 3 = z-axis], and *zh* defines the distance along the *pax* axis to plot the ellipse.

`ellps([2 3],[4 1.5],'-b',pi/4,3,2)` plots an ellipse in a plane perpindicular to the z-axis at a heightof z = 2.

See Also

`dp, hline, vline, zline`

encode

Purpose

Translates a variable into matlab-executable code.

Synopsis

```
str = encode(item,varname)
str = encode(item,varname,options)
```

Description

The created code can be eval'd or included in an m-file to reproduce the variable. This is essentially an inverse function of "eval" for variables.

Input is a variable (item) and an optional name for that variable (varname). If (varname) is omitted, the input variable name will be used. If varname is empty, leading code which does assignment is omitted.

Output is a string (str) which can be inserted into an m-file or passed to eval for execution.

Options

`max_array_size :`	`[10000]` Maximum size allowed for any array dimension. Arrays with any size larger than this will be returned as simply [NaN]
`structformat :`	`['struct'	{'dot'}]` defines how structures are encoded. 'struct' uses a "struct('a',val)" style (but can get very complex with large structures). 'dot' uses "x.a = val" format which is easier to read, but less compact.
`forceoneline :`	`[{'off'}	'on']` remove all line breaks and ellipses from output. WARNING: this can cause a VERY long line on big objects and may exceed the maximum line length of editors or even MATLAB.

Example

Create code to reproduce a preprocessing structure

```
>> p = preprocess('default','meancenter');
>> encode(p)
```

See Also

`encodexml, parsexml`

encodexml

Purpose

Convert standard data types into XML-encoded text.

Synopsis

```
xml = encodexml(var)
xml = encodexml(var,'name')
xml = encodexml(var,'name','outputfile.xml')
```

Description

Converts a standard Matlab variable (var) into a human-readable XML format. The optional second input ('name') gives the name for the object's outer wrapper and the optional third input ('filename.xml') gives the name for the output file (if omitted, the XML is only returned in the output variable). For more information on the format, see the PARSEXML function.

Example

```
>> z.a = 1;
>> z.b = { 'this' ; 'that' };
>> z.c.sub1 = 'one field';
>> z.c.sub2 = 'second field';

>> z = encodexml(z,'mystruct')

z =
<mystruct>
  <a class="numeric" size="[1,1]">1</a>
  <b class="cell" size="[2,1]">
    <tr>
      <td class="string">this</td>
    </tr>
    <tr>
      <td class="string">that</td>
    </tr>
  </b>
  <c>
    <sub1 class="string">one field</sub1>
    <sub2 class="string">second field</sub2>
  </c>
</mystruct>
```

See Also

`encode, parsexml`

estimatefactors

Purpose

Estimate number of significant factors in multivariate data.

Synopsis

```
    S = estimatefactors (x,options)
```

Description

Given a bilinear dataset, ESTIMATEFACTORS estimates the number of significant factors required to describe the data. The algorithm uses PCA bootstrapping (resampling) of the data. The PCA loadings determined for each resampling are compared for changes. Principal components which change significantly from one resampling to the next are probably due mostly to noise rather than signal.

The output is an estimate of the signal to noize ratio for each principal component. Ratios of 2 or below are dominated by noise, above 3 are OK, and between 2 and 3 are a jugement call. The number of factors needed to describe the data is the number of eigenvectors with signal to noise ratios greater than about 2.

This function is based on an algorithm developed and Copyrighted 1997 by Ronald C. Henry, Eun Sug Park, and Clifford H. Spiegelman and used by permission of the authors. For reference see:

* Henry, R.C., Park, E.S., & Spiegelman, C.H. (1999). Comparing A New Algorithm With The Classic Methods For Estimating The Number Of Factors. Chemometrics and Intelligent Laboratory Systems, 48(1), 91-97.

* Park, E.S., Henry, R.C., & Spiegelman C.H. (2000). Estimating The Number Of Factors To Include In A Height Dimensional Multivaraite Bilinear Model. Communications in Statistics-Theory and Methods, 29(3), 723-746.

Options

options =	a structure array with the following fields:
`plots`:	`['none' | {'final'} ]` Governs plotting.
`resample`:	`[ {42} ]` number of times the data is to be resampled. Generally, values of 40 or 50 are sufficient. Values greater than several hundred are not required.
`maxfactors`:	`[ {30} ]` maximum number of factors to plot (if plots are selected by options.plots).
`preprocessing`:	`{[]}` Preprocessing structure or keyword (see PREPROCESS), to apply before analyzing data.

The default options can be retreived using: `options = estimatefactors('options');`.

See Also

`pca, pcaengine`

evolvfa

Purpose

Perform forward and reverse evolving factor analysis.

Synopsis

```
      [egf,egr] = evolvfa(xdat,plot,tdat)
```

Description

`[egf,egr] = evolvfa(xdat)` calculates eigenvalues of sub-matrices of `xdat` and returns results of the forward analysis in `egf` and reverse analysis in `egr`.

`[egf,egr] = evolvfa(xdat,`*plot*`)` allows the user to control plotting options. When *plot* is set to 0 the plot of the results is suppressed. Setting *plot* equal to 1 {default} plots the results.

`[egf,egr] = evolvfa(xdat,`*plot*`,`*tdat*`)` gives the routine an optional vector *tdat* to plot results against.

See Also

`ewfa, pca, wtfa`

evridebug

Purpose

Checks the PLS_Toolbox installation for problems.

Synopsis

```
problems = evridebug
```

Description

`EVRIDEBUG` runs various tests on the PLS_Toolbox installation to assure that all necessary files are present and not "shadowed" by other functions of the same name. This utility should be run if you experience problems with the PLS_Toolbox.

`EVRIDEBUG` tests for:

* Missing PLS_Toolbox folders in path,

* Multiple versions of PLS_Toolbox,

* "Shadowed" files (duplicate named files), and

* Duplicate definitions of Dataset object.

The single output `problems` is a cell containing the text of the problems encountered. If no problems are encountered, `problems` will be empty.

Examples

```
>> evridebug

No PLS_Toolbox installation problems were identified.
```

See Also

`evriinstall, evriupdate`

evriinstall

Purpose

Install and verify PLS_Toolbox

Synopsis

```
evriinstall
```

Description

`EVRIINSTALL` automates the installation and verification of the PLS_Toolbox. To run `evriinstall`:

1. Unzip PLS_Toolbox to a local directory (typically C:\MATLAB7\toolbox\).

2. Open Matlab and navigate to the directory created above in the Current Directory window.

3. Type `evriinstall` at the command line and press Enter.

Installation involves first setting the Matlab Path to include the PLS_Toolbox directory and its subdirectories. The script then runs `evridebug` to check for potential problems after installation.

See Also

`evridebug`, `evriupdate`

evriupdate

Purpose

Check the Eigenvector Research web site for PLS_Toolbox updates.

Synopsis

```
outofdate = evriupdate(umode, product)
```

Description

Check Eigenvector.com for available PLS_Toolbox updates. EVRIUPDATE checks the Eigenvector Research web site for the most current PLS_Toolbox release version number. This is compared to the currently installed version. A message reporting the availability of an update is given as necessary. Input (product) will check for an individual product for umodes 0-2.

The optional input (umode) can be any of the following:

`'auto'`:	perform an automatic check based on Auto Check settings
`'settings'`:	Gives GUI to modify the automatic check settings
`'prompt'`:	prompt user before performing check - includes prompt to allow user to modify settings.

or (umode) one of the following levels of automatic reports:

0 : give dialog stating if new version is available or not

1 : give dialog ONLY if a new version is available

2 : gives no dialog messages - only returns output flag (see below)

3 : give dialog of all products installed and version info.

4 : give dialog of all products from EVRI and versions.

5 : give dialog of all products but ONLY if a new version is available

The default mode is 4.

The output (outofdate) will be 0 (zero) if the installed PLS_Toolbox is current, 1 (one) if the installed version is out of date and -1 if evriupdate could not retreive the most current version number.

See Also

`evridebug, evriinstall`

ewfa

Purpose

Evolving window factor analysis.

Synopsis

```
      [eigs,skl] = ewfa(dat,window,plots,scl)
```

Description

The inputs are the data matrix `dat` and the window witdth `window`. The output `eigs` is the eigenvalues for each window. The windowed eigenvalues vs. sample number is also plotted. Note that the eigenvalues on the ends of the record (less than the half width of the window) are plotted as dashed lines. The output `skl` is a scale that can be used to plot `eigs` against.

Optional input `plots` can be used to suppress plotting when set to 0 {default `plots = 1`}. Optional input `scl` is a scale to plot against. It is also used to construct a new `skl`.

See Also

`evolvfa, pca, wtfa`

excludemissing

Purpose

Automatically exclude too-much missing data in a matrix.

Synopsis

```
    [newx,bad] = excludemissing(x,threshold)
```

Description

Excludes rows, columns, or n-dim elements of input `x` which have too much missing based on the input `threshold` which is a fraction of allowed missing data. If omitted, threshold will be equal to the default max_missing value of the function MDCHECK (typically 0.40).

Outputs are a dataset object with excluded elements `newx` and a cell holding the indices of the bad elements for each mode of data `bad`.

See Also

`mdcheck, replace`

explode

Purpose

Extracts variables from a structure array.

Synopsis

```
      explode(sdat,mod,txt,out)
      options = explode('options')
```

Description

`EXPLODE` writes the fields of the input structure `sdat` to variables in the workspace with the same variable names as the field names. If `sdat` is a standard model structure, only selected information is written to the workspace.

Optional string input *txt* appends a string to the variable output names.

Options

options = a structure array with the following fields:

`model:` `[ 'no' | {'yes'} ]` interpret `sdat` as model if possible, and

`display:` `[ 'off' | {'on'} ]}` display model information.

The default options can be retreived using: `options = explode('options');`.

Examples

For the structure array `x`

```
>> x.field1 = 2;
>> x.field2 = 3;
>> explode(x)
Input (sdat) is not a recognized model. Exploding as regular structure
>> whos
  Name         Size                    Bytes  Class

  field1       1x1                         8  double array
  field2       1x1                         8  double array
  x            1x1                       264  struct array
```

the variables `field1` and `field2` have been written to the base workspace.

See Also

`analysis, modelstruct`

exportfigure

Purpose

Automatically export figures to an external program.

Synopsis

```
exportfigure
exportfigure(target,sourcefigs)
```

Description

Exports one or more open figures into a new blank document in an external program. No inputs are required.

OPTIONAL INPUTS:

target =	The target program to export figures to, `target` can have the following values:
	'powerpoint' : Microsoft PowerPoint {default}
	'word' : Microsoft Word
	'clipboard' : System Clipboard (to paste into other program)
sourcefigs =	A vector of figure numbers to export {default is the current open figure (see GCF)}.
	sourcefigs = 'all', exports all open figures.

Note: "clipboard" export can only operate on one figure at a time.

See Also

factdes

Purpose

Output a full factorial design matrix.

Synopsis

```
desgn = factdes(fact,levl)
```

Description

The input `fact` is the number of factors in the design and the output `desgn` is the experimental design matrix.

`desgn = factdes(fact);` provides a full factorial two level design.

Optional input *levl* allows for multiple level designs.

`desgn = factdes(fact,` *levl*`);` provides a full factorial`levl` level design {default `levl = 2`}.

See Also

`distslct, doptimal, ffacdes1, stdsslct`

fastnnls

Purpose

Fast non-negative least squares.

Synopsis

```
[b,xi] = fastnnls(x,y,tol,b0,eqconst,xi);
```

Description

Solves the equation `xb = y` subject to the constraint that `b` is non-negative. The inputs are the matrix of predictor variables `x`, vector or matrix of predicted variables `y`. Optional inputs include: tolerance on the size of a regression coefficient that is considered zero (if `tol = 0` the default is used `tol = max(size(x))*norm(x,1)*eps`), `tol`, initial guess for the regression vectors, `b0`, and the equality constraints matrix, `eqconst`, equal in size to b0 and containing a value of NaN to indicate an unconstrained value or any finite value to indicate a constrained value. The optional input `xi` is the cached inverses output by a previous run of fastnnls (see outputs) or 0 (zero) to disable caching.

The outputs are the non-negatively constrained least squares solution, `b`, and the cache of x inverses, `xi`. If input `y` is a matrix, the result is the solution for each column of `y` calculated independently.

If tol is set to 0 or [], the default tolerance will be used. If xi is set to 0, caching will be disabled.

`FASTNNLS` is fastest when a good estimate of the regression vector `b0` is input. This eliminates much of the computation involved in determining which coefficients will be nonzero in the final regression vector. This makes it very useful in alternating least squares routines. Note that the input `b0` must be a feasible (`i.e.` nonnegative) solution.

The `FASTNNLS` algorithm is based on work by Bro and de Jong, *J. Chemo.*, **11**(5), 393-401, 1997.

INPUTS:

`x` =	the matrix of predictor variables,
`y` =	vector or matrix of predicted variables. If (y) is a matrix, the result is the solution for each column calculated independently.

OPTIONAL INPUTS:

`tol` =	tolerance on the size of a regression coefficient that is considered zero. Not supplied or empty matrix is implies the default value (based on x and eps),
`b0` =	initial guess for the regression vectors. Default or empty matrix is interpreted as no known intial guess,

`eqconst` = equality constraints matrix equal in size to b0 and containing a value of NaN to indicate an value not equality-constrained or any finite value to indicate an equality-constrained value. An empty matrix indicates no equality constraints on any elements.

`xi` = cached inverses output by a previous run of fastnnls (see outputs) or 0 (zero) to disable caching. An empty matrix is valid as a placeholder in the inputs.

See Also

`lsq2top, mcr, parafac`

ffacdes1

Purpose

Output a fractional factorial design matrix.

Synopsis

```
desgn = ffacdes1(k,p)
```

Description

`FFACDES1` outputs a $2^{(k-p)}$ fractional factorial design of experiments. The design is constructed such that the highest order interaction term is confounded. This is one way to select a fractional factorial. Input `k` is the total number of factors in the design and `p` is the number of confounded factors {default: `p = 1`}. Note that it is required that `p < k`. Output `desgn` is the experimental design matrix.

See Also

`distslct, doptimal, factdes, stdsslct`

figbrowser

Purpose

Browser with icons of all Matlab figures.

Synopsis

```
figbrowser(varargin)
```

Description

The figbrowser function creates a figure containing thumbnail images of all visible Matlab figures. Clicking on an icon will instantly make that figure the current figure and bring to the front.

INPUTS

`''(empty)` =	Creates or updates current figbrowser window
`'focus'` =	Brings the figbrowser window to the front and updates if figures have been created or deleted since last update
'hide' =	Hides the figbrowser window
`['addmenu',target_figure]` =	Adds figbrowser trigger menu to current or specified figure
`'on'` =	Turns on automatic addition of figbrowser menu to all figures. NOTE: menu addition can be permanently disabled by modifying the enableautoadd option in figbrowser. This option can be set using setplspref. When set to 'off', figbrowser will only show up on GUIs which specifically add it themselves, no matter what figbrowser command is issued. This option can also be modified through the "Figbrowser on All" menu item in all Figbrowser menus.
`'off'` =	Removes figbrowser menus from all figures.
`['autodock','on']` =	Adds figbrowser trigger menu to current or specified figure
`['autodock','off']` =	Adds figbrowser trigger menu to current or specified figure Controls auto-docking of standard figures on creation (figbrowser must be "on"). Auto-docking forces any standard figure to be opened in the Figure window.

See Also

figmerit

Purpose

Analytical figures of merit for multivariate calibration.

Synopsis

```
[nas,nnas,sens,sel] = figmerit(x,y,b);
```

Description

Calculates analytical figures of merit for PLS and PCR standard model structures. Inputs are the preprocessed (usually centered and scaled) spectral data `x`, the preprocessed analyte data `y`, and the regression vector, `b`. Note that for standard PLS and PCR structures `b = model.reg`.

The outputs are the matrix of net analyte signals `nas` for each row of `x`, the norm of the net analyte signal for each row `nnas` (this is corrected to include the sign of the prediction), the matrix of sensitivities for each sample `sens`, and the vector of selectivities for each sample `sel` (`sel` is always non-negative).

Note that the "noise-filtered" estimate present in previous versions is no longer used because an improved method for calculating the net analyte vector makes it redundant

Examples

Given the 7 LV PLS model:

```
modl = pls(x,y,7);
Rhat = modl.loads{1,1}*modl.loads{2,1}';
[nas,nnas,sens,sel,nfnas] = figmerit(x,y,Rhat);
```

Given the 5 PC PCR model:

```
modl = pcr(auto(x),auto(y),5);
Rhat = modl.loads{1,1}*modl.loads{2,1}';
[nas,nnas,sens,sel,nfnas] = figmerit(auto(x),auto(y),Rhat);
```

See Also

`pcr, pls`

findindx

Purpose

Finds the index of the array element closest to value r.

Synopsis

```
      index = findindx(array,r)
```

Description

Inputs are an array of values (array) and a value to locate (r). Output (index) is the linear index into array which will return the closest value to r.

Examples

```
index = findindx(array,r);       %get an index

nearest_value = array(index);    %find the value
```

See Also

`lamsel`

fir2ss

Purpose

Convert a finite impulse response model into an equivalent state-space model.

Synopsis

```
[phi,gamma,c,d] = fir2ss(b)
```

Description

`[phi,gamma,c,d] = fir2ss(b)` takes a vector of FIR coefficients `b` and outputs the `phi`, `gamma`, `c` and `d` matrices for a equivalent discrete state-space model.

See Also

`autocor`, `crosscor`, `plspulsm`, `wrtpulse`

fitpeaks

Purpose

Peak fitting routine.

Synopsis

```
[peakdefo,fval,exitflag,out,fit,res] = fitpeaks(peakdef,y,ax,options)
```

Description

Based on the initial guess in input `peakdef`, `FITPEAKS` estimates the peak fit (also the Jacobian and Hessian), and makes a call to `LMOPTIMIZEBND` to find the best fit of the peaks to the data. (See `LMOPTIMIZEBND` for additional information.) Results are output to `peakdefo`.

Information about individual peaks is stored in standard peak structures (see `PEAKSTRUCT`). Information on multiple peaks is stored in a multi-record structure. Given a standard peak structure (`peakdef`) that contains an initial guess of peak locations and widths, `FITPEAKS` finds new parameters that best fits peaks to the rows of the *MxN* data matrix (`y`). Results are output to a standard peak structure (`peakdefo`).

Fields of (`peakdef`) required in the initial guess for each peak are (`.fun`), (`.param`), (`.lb`), (`.penlb`), (`.ub`), and (`.penub`).

INPUTS:

`peakdef`	= multi-record standard peak structure with the following fields:
`name`:	`'Peak'`, name indicating that this is a standard peak structure.
`id`:	`''`, double or character string peak identification.
`fun`:	[{`'Gaussian'`} | `'Lorentzian'` | `'PVoigt1'` | `'PVoigt2'`], defines the peak function (see definitions in the Algorithm section).
`param`:	Parameter list for each peak function. The number of parameters depends on the peak function used: `'Gaussian'`: [height, location, width], `'Lorentzian'`: [height, location, width], `'PVoigt1'`: [height, location, width, fraction Gaussian], `'PVoigt2'`: [height, location, width, fraction Gaussian].
`lb`:	[ ], Lower bounds on the function parameters. This is a row vector with the same number of elements as `peakdef.param`.
`penlb`:	[ ], Penalty wt for lower bounds, >=0. This is a row vector with the same number of elements as `peakdef.param`. If set to 0 this constraint is not employed.
`ub`:	[ ], Upper bounds on the function parameters. This is a row vector with the same number of elements as `peakdef.param`.

`penub`: [ ], Penalty wt for upper bounds, >=0. This is a row vector with the same number of elements as `peakdef.param`. If set to 0 this constraint is not employed.

`area`: [ ], Estimated peak area.

`y` = *M*x*N* measured responses with peaks to fit. Each row of (`y`) is fit to the peaks given in (`peakdef`).

OPTIONAL INPUTS:

`ax` = 1x*N* x-axis to fit to {default `ax=1:N`}.

`options` = discussed below in the Options Section.

OUTPUTS:

`peakdefo` = The input peak structure (`peakdef`) with parameters changed to correspond to the best fit values.

`fval` = Scalar value of the objective function evaluated at termination of `FITPEAKS`.

`exitflag` = Describes the exit condition (see `LMOPTIMIZEBND`).

`out` = Structure array with information on the optimization/fitting (see `LMOPTIMIZEBND`).

`fit` = Model fit of the peaks, i.e it is the best fit to (`y`).

`res` = Residuals of fit of the peaks.

Algorithm

Peaks are fit to the functions defined below based on the definitions in the field (`peakdef.fun`). The functions can be evaluated using independent functions or a wrapper function `PEAKFUNCTION`. See `PEAKFUNCTION` for more help.

For `peakdef.fun = 'Gaussian'` the function is

$$f(a_i,\mathbf{x}) = x_1\,\mathrm{e}^{\frac{-(a_i-x_2)^2}{2x_3^2}}$$

where $a_i,\ i=1,\ldots,N$ is the i^{th} element of optional input (`ax`), and $\mathbf{x}=\begin{bmatrix}x_1 & x_2 & x_3\end{bmatrix}$ corresponds to the peak parameters in the three-element vector (`peakdef.param`). Constraints that should be used are (bounds in `peakdef`) are $x_1 \ge 0$ and $x_3 \ge 0$.

For `peakdef.fun = 'Lorentzian'` the function is

$$f(a_i,\mathbf{x}) = x_1\left[1+\left(\frac{a_i-x_2}{x_3}\right)^2\right]^{-1} = x_1\left[\frac{x_3^2}{x_3^2+(a_i-x_2)^2}\right].$$

Constraints that should be used are (bounds in peakdef) are $x_1 \geq 0$ and $x_3 \geq 0$.

For peakdef.fun = 'PVoigt1' the function is

$$f(a_i, \mathbf{x}) = x_1 \left[x_4 \, \mathrm{e}^{\frac{-4\ln(2)(a_i - x_2)^2}{x_3^2}} + (1 - x_4) \left[\frac{x_3^2}{(a_i - x_2)^2 + x_3^2} \right] \right]$$

where $\mathbf{x} = [x_1 \quad x_2 \quad x_3 \quad x_4]$ corresponds to the peak parameters in the four-element vector (peakdef.param). Constraints that should be used are (bounds in peakdef) are $x_1 \geq 0$ and $x_3 \geq 0$, while $1 \geq x_4 \geq 0$. The Pseudo-Voigt peak shape is an estimate of the Gaussian and Lorentzian peak shapes convolved.

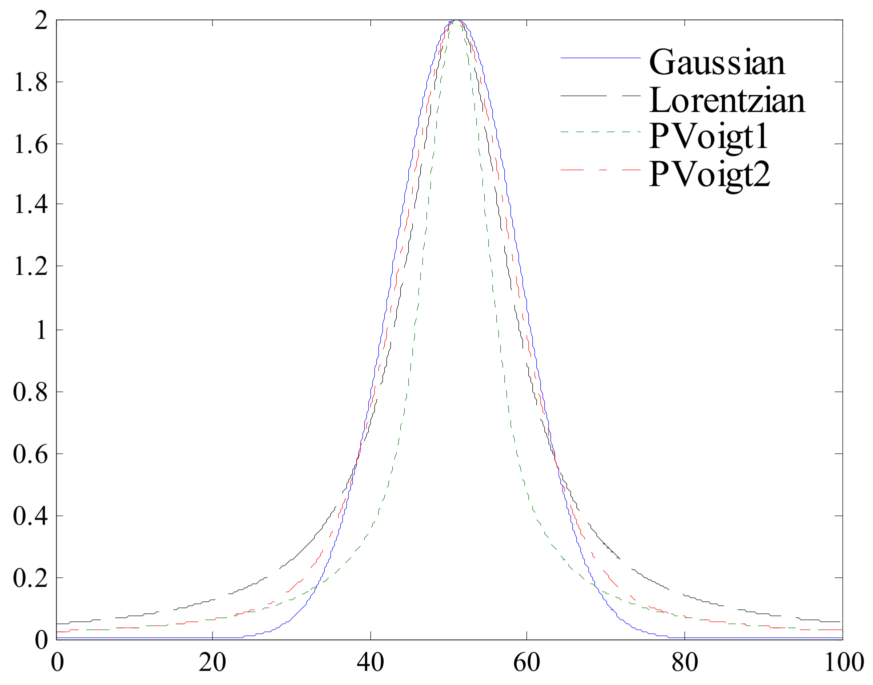
For peakdef.fun = 'PVoigt2' the function is

$$f(a_i, \mathbf{x}) = x_1 \left[x_4 \, \mathrm{e}^{\frac{-(a_i - x_2)^2}{2x_3^2}} + (1 - x_4) \left[\frac{x_3^2}{(a_i - x_2)^2 + x_3^2} \right] \right]$$

where $\mathbf{x} = [x_1 \quad x_2 \quad x_3 \quad x_4]$ corresponds to the peak parameters in the four-element vector (peakdef.param). Constraints that should be used (bounds in peakdef) are $x_1 \geq 0$ and $x_3 \geq 0$, while $1 \geq x_4 \geq 0$. The Pseudo-Voigt peak shape is an estimate of the Gaussian and Lorentzian peak shapes convolved.

A comparison of the four peaks is given in the figure below, and was generated using the following code:

```
ax     = 0:0.1:100;
y      = zeros(4,length(ax));
plot(ax,peakgaussian([2 51 8],ax),'-b', ...
     ax,peaklorentzian([2 51 8],ax),'--k', ...
     ax,peakpvoigt1([2 51 8 0.5],ax),':g', ...
     ax,peakpvoigt2([2 51 8 0.5],ax),'-.r')
legend('Gaussian','Lorentzian','PVoigt1','PVoigt2')
```



Options

`options` = structure array with the following fields:

- `name:` '`options`', name indicating that this is an options structure.
- `display:` [`'off'` | {`'on'`}] governs level of display to the command window.
- `optimopts:` options structure from `LMOPTIMIZEBND`. This field is passed to `LMOPTIMIZEBND` and can be used to control the optimization / fitting.

Examples

```
%Make a single known peak
  ax            = 0:0.1:100;
  y             = peakgaussian([2 51 8],ax);

%Define first estimate and peak type
  peakdef       = peakstruct;
  peakdef.param = [0.1  43   5]; %coef, position, spread
  peakdef.lb    = [0     0  0.0001]; %lower bounds on param
  peakdef.penlb = [1e-6 1e-6 1e-6];
  peakdef.ub    = [10 99.9 40]; %upper bounds on params
  peakdef.penub = [1e-6 1e-6 1e-6];

%Estimate fit and plot
  yint   = peakfunction(peakdef,ax);
  [peakdef,fval,exitflag,out] = fitpeaks(peakdef,y,ax);
  yfit   = peakfunction(peakdef,ax); figure
  plot(ax,yint,'m',ax,y,'b',ax,yfit,'r--')
  legend('Initial','Actual','Fit')
```

See Also

`peakfind`, `lmoptimizebnd`, `peakfunction`, `peakgaussian`, `peaklorentzian`, `peakpvoigt1`, `peakpvoigt2`, `peakstruct`

frpcr

Purpose

Full-ratio PCR calibration and prediction.

Synopsis

```
model = frpcr(x,y,ncomp,options)     %calibration
pred  = frpcr(x,model,options)       %prediction
valid = frpcr(x,y,model,options)     %validation
options = frpcr('options')
```

Description

FRPCR calculates a single full-ratio PCR model using the given number of components ncomp to predict y from measurements x. Random multiplicative scaling of each sample can be used to aid model stability. Full-Ratio PCR models are based on the simultaneous regression for both y-block prediction and scaling variations (such as those due to pathlength and collection efficiency variations). The resulting PCR model is insensitive to absolute scaling errors.

NOTE: For best results, the x-block should not be mean-centered.

Inputs are x the predictor block (2-way array or DataSet Object), y the predicted block (2-way array or DataSet Object), ncomp the number of components to to be calculated (positive integer scalar) and the optional options structure, *options*.

The output of the function is a standard model structure model. In prediction and validation modes, the same model structure is used but predictions are provided in the model.detail.pred field.

Although the full-ratio method uses a different method for determination of the regression vector, the fundamental idea is very similar to the optimized scaling 2 method as described in:

T.V. Karstang and R. Manne, "Optimized scaling: A novel approach to linear calibration with close data sets", Chemom. Intell. Lab. Syst., **14**, 165-173 (1992).

Options

options =	a structure with the following fields:
pathvar:	[{0.5}] standard deviation for random multiplicative scaling. A value of zero will disable the random sample scaling but may increase model sensitivity to scaling errors,
useoffset:	[{'off'} | 'on'] flag determining use of offset term in regression equations (may be necessary for mean-centered x-block),
display:	[{'off'} | 'on'] governs level of display to command window,

`plots:`	`[ {'none'} | 'intermediate' | 'final' ]` governs level of plotting,
`preprocessing:`	`{[ ] [ ]}` cell of two preprocessing structures (see `PREPROCESS`) defining preprocessing for the x- and y-blocks.
`algorithm:`	`[ {'direct'} | 'empirical' ]` governs solution algorithm. Direct solution is fastest and most stable. Only `empirical` will work on single-factor models when useoffset is 'on', and
`blockdetails:`	`[ 'compact' | {'standard'} | 'all' ]` extent of predictions and raw residuals included in model. `'standard'` only uses y-block, and `'all'` uses x- and y-blocks.
`confidencelimit:`	`[ {'0.95'} ]` Confidence level for Q and T2 limits. A value of zero (0) disables calculation of confidence limits.

In addition, there are several options relating to the algorithm. See FRPCRENGINE.

The default options can be retreived using: `options = frpcr('options');`.

See Also

`frpcrengine, mscorr, pcr`

frpcrengine

Purpose

Engine for full-ratio PCR; also known as optimized scaling 2 PCR.

Synopsis

```
[b,ssq,u,sampscales,msg,options] =
frpcrengine(x,y,ncomp,options);  %calibration
[yhat] = frpcrengine(x,b);  %prediction
```

Description

Calculates a single full-ratio, FR, PCR model using the given number of components `ncomp` to predict `y` from measurements `x`. Random multiplicative scaling of each sample can be used to aid model stability. Full-Ratio PCR models are based on the simultaneous regression for both y-block prediction and scaling variations (such as those due to pathlength and collection efficiency variations). The resulting PCR model is insensitive to scaling errors.

NOTE: For best results, the x-block should not be mean-centered.

Although the full-ratio method uses a different method for determination of the regression vector, the fundamental idea is very similar to the optimized scaling 2 method as described in:

T.V. Karstang and R. Manne, "Optimized scaling: A novel approach to linear calibration with close data sets", Chemom. Intell. Lab. Syst., **14**, 165-173 (1992).

For calibration mode, inputs include the x-block data, *x*, y-block data, *y*, and number of components *ncomp*. The optional input *options* is described below. Calibration mode outputs include:

`b`	=	the full-ratio regression vector for a SINGLE MODEL at the given number of PCs,
`ssq`	=	PCA variance information,
`u`	=	the x-block loadings,
`sampscales`	=	random scaling used on the samples,
`msg`	=	warning messages, and
`options`	=	the modified options structure.

For prediction mode, inputs are the x-block data, *x*, and the full-ration regression vectors, *b*. The one output is the predicted y, *yhat*.

Options

`options` =	a structure with the following fields:
`pathvar`:	`[ {0.5} ]` standard deviation for random multiplicative scaling. A value of zero will disable the random sample scaling but may increase model sensitivity to scaling errors,
`useoffset`:	`[ {'off'} | 'on' ]` flag determining use of offset term in regression equations (may be necessary for mean-centered x-block),
`display`:	`[ 'off' | {'on'} ]` governs level of display to command window,
`plots`:	`[ {'none'} | 'intermediate' ]` governs level of plotting,
`algorithm`:	`[ {'direct'} | 'empirical' ]` governs solution algorithm. Direct solution is fastest and most stable. Only `empirical` will work on single-factor models when useoffset is 'on', and
`tolerance`:	`[ {5e-5} ]` extent of predictions and raw residuals included in model. `'standard'` only uses y-block, and `'all'` uses x- and y-blocks, and
`maxiter`:	`[ {100} ]` maximum number of iterations.

The default options can be retreived using: `options = frpcrengine('options');`.

See Also

`frpcr, mscorr, pcr`

ftest

Purpose

Inverse F test and F test.

Synopsis

```
      fstat = ftest(p,n,d,flag)
```

Description

`fstat = ftest(p,n,d)` or `fstat = ftest(p,n,d,1)` calculates the F statistic `fstat` given the probability point `p` and the number of degrees of freedom in the numerator `n` and denomenator `d`.

`fstat = ftest(p,n,d,2)` calculates the probability point `fstat` given the F statistic `p` and the number of degrees of freedom in the numerator `n` and denomenator `d`.

Examples

`a = ftest(0.05,5,8);` returns the value `3.6875` for `a`, and

`a = ftest(3.6875,5,8,2);` returns the value `0.050` for `a`.

See Also

`chilimit, statdemo, ttestp`

fullsearch

Purpose

Exhaustive Search Algorithm.

Synopsis

```
[desgn,fval] = fullsearch(fun,X,Nx_sub,P1,P2, ...);
```

Description

Fullsearch selects the `Nx_sub` variables in the *M* by *Nx* matrix `X` that minimizes fun. This can be used for variable selection. The algorithm should only be used for small problems because calculation time increases significantly with the size of the problem. fun is the name of the function (defined as a character string of an inline object) to be minimized. The function is called with the `FEVAL` function as follows: `feval(fun,X,P1,P2,....)`, where `X` is the first argument for `fun` and `P1, P2, ...` the additional arguments of `fun`.

The output `desgn` is a matrix (class "logical") with the same size as `X` (*M* by *Nx*) with 1's where the variables where selected and 0's otherwise. Output `fval` has the *M* corresponding values of the objective function sorted in ascending order.

Examples

find which 2of 3 variables minimizes the inline function `g`:

```
   x = [0:10]';
   x = [x x.^2 randn(11,1)*10];
   y = x*[1 1 0]';
   g = inline('sum((y-x*(x\y)).^2)');
   [d,fv] = fullsearch(g,x,2,y);
```

find the 2 variables that minimize the cross-validation error for PCR, noting that the output from `CROSSVAL` is a vector and `g` should return a scalar

```
load plsdatad
   x = xblock1.data;
   y = yblock1.data;
   g = inline('min(sum(crossval(x,y,''pcr'',{''con'' 3},1,0)))','x','y');
   [d,fv] = fullsearch(g,x,2,y);  %takes a while if Nx_sub is > 2
```

See Also

`calibsel, crossval, genalg`

gaselctr

Purpose

Genetic algorithm for variable selection with PLS.

Synopsis

```
model = gaselctr(x,y,options)
[fit,pop,avefit,bstfit] = gaselctr(x,y,options)
options = gaselctr('options')
```

Description

GASELCTR uses a genetic algorithm optimization to minimize cross validation error for variable selection.

INPUTS:

x =	the predictor block (x-block), and
y =	the predicted block (y-block) (note that all scaling should be done prior to running GASELCTR).

Options

options =	a structure array with the following fields:
plots:	['none' | {'intermediate'} | 'replicates' | 'final'] Governs plots. 'final' gives only a final summary plot. 'replicates' gives plots at the end of each replicate. 'intermediate' gives plots during analysis. 'none' gives no plots.
popsize:	{64} the population size (16≤popsize≤256 and popsize must be divisible by 4),
maxgenerations:	{100} the maximum number of generations (25≤mg≤500),
mutationrate:	{0.005} the mutation rate (typically 0.001≤mt≤0.01),
windowwidth:	{1} the number of variables in a window (integer window width),
convergence:	{50} percent of population the same at convergence (typically cn=80),
initialterms:	{30} percent terms included at initiation (10≤bf≤50),
crossover:	{2} breeding cross-over rule (cr = 1: single cross-over; cr = 2: double cross-over),
algorithm:	['mlr' | {'pls'}] regression algorithm,
ncomp:	{10} maximum number of latent variables for PLS models,
cv:	['rnd' | {'con'}] cross-validation option ('rnd': random subset cross-validation; 'con': contiguous block subset cross-validation),

`split`:	`{5}` number of subsets to divide data into for cross-validation,
`iter`:	`{1}` number of iterations for cross-validation at each generation,
`preprocessing`:	`{[] []}` a cell containing standard preprocessing structures for the X- and Y-blocks respectively (see PREPROCESS),
`preapply`:	`[ {0} | 1 }` If 1, preprocessing is applied to data prior to GA. This speeds up the performance of the selection, but my reduce the accuracy of the cross-validation results. Output "fit" values should only be compared to each other. A full cross-validation should be run after analysis to get more accurate RMSECV values.
`reps`:	`{1}` the number of replicate runs to perform,
`target`:	a two element vector `[target_min target_max]` describing the target range for number of variables/terms included in a model `n`. Outside of this range, the `penaltyslope` option is applied by multiplying the fitness for each member of the population by: `penaltyslope*(target_min-n)` when `n<target_min`, or `penaltyslope*(n-target_max)` when `n>target_max`. Field `target` is used to bias models towards a given range of included variables (see `penaltyslope` below),
`targetpct`:	`{1}` flag indicating if values in field `target` are given in percent of variables (1) or in absolute number of variables (0), and
`penaltyslope`:	`{0}` the slope of the penalty function (see `target` above).

The default options can be retreived using: `options = gaslctr('options');`.

OUTPUT:

model =	a standard GENALG model structure with the following fields:
`modeltype`:	`'GENALG'` This field will always have this value,
`datasource`:	`{[1x1 struct] [1x1 struct]}`, structures defining where the X- and Y-blocks came from
`date`:	date stamp for when GASELCTR was run,
`time`:	time stamp for when GASELCTR was run,
`info`:	`'Fit results in "rmsecv", population included variables in "icol"'`, information field describing where the fitness results for each member of the population are contained,
`rmsecv`:	fitness results for each member of the population, for X *M*x*N* and *Mp* unique populations at convergence then `rmsecv` will be *1xMp*,
`icol`:	each row of `icol` corresponds to the variables used for that member of the population (a 1 [one] means that variable was used and a 0 [zero] means that it was not), for X *M*x*N* and *Mp* unique populations at convergence then `icol` will be *Mp*x*N*, and
`detail`:	`[1x1 struct]`, a structure array containing model details including the following fields: `avefit`: the average fitness at each generation,

`bestfit`: the best fitness at each generation, and

`options`: a structure corresponding to the options discussed above.

Examples

To use mean centering outside the genetic algorithm (no additional centering will be performed within the algorithm) do the following:

```
x2 = mncn(x);
      y2 = mncn(y);
[fit,pop] = gaselctr(x2,y2);
```

To use mean centering inside the genetic algorithm (centering will be performed for each cross-validation subset) do the following:

```
options = gaselctr('options');
      options.preprocessing{1} = preprocess('default', 'mean center');
      options.preprocessing{2} = preprocess('default', 'mean center');
[fit,pop] = gaselctr(x2,y2,options);
```

See Also

`calibsel, fullsearch, genalg, genalgplot`

gcluster

Purpose

K-means and K-nearest neighbor cluster analysis with dendrograms.

Synopsis

```
gcluster(data,labels)
```

Description

`gclster(data)` performs a cluster analysis on the data matrix `data` using K-means or K-nearest neighbor clustering and plots a dendrogram showing distances between the samples. `gcluster` is a graphical user interface that calls the function `cluster`. The user can choose cluster method (K-means or KNN), and data scaling options. PCA can also be used on the data with distances based on raw scores or on a Mahalanobis distance measure.

`gclster(data,`*labels*`)` plots on the dendrogram sample names contained in the matrix of text *labels*. *labels* can be entered as a matrix where each row is a label in single quotes and each label has the same number of characters.

Note: Calling `gclster` with no inputs starts the graphical user interface (GUI) for this analysis method.

See Also

`cluster, simca`

genalg

Purpose

Genetic algorithm for variable selection to optimize model predictive ability with graphical user interface.

Synopsis

```
    genalg(xdat,ydat)
```

Description

`GENALG` performs variable selection using a genetic algorithm. The function creates a graphical user interface that allows the user to load data from the workspace and select all of the GA algorihtm optional parameters (`GASELCTR` is a command-line version). A wide range of GA settings can be selected from the GUI. Please see `GASELCTR` for a description of each option.

Optional inputs are the training data consisting of a matrix of predictor variables *xdat* and column vector of predicted variable *ydat*. (The number of rows in *xdat* and *ydat* must be the same). If GENALG is called with no inputs, *xdat* and *ydat* can be loaded using the `File` menu.

In addition to various plots, the GUI can produce and save the results in a model structure that is the same as that returned by `GASELCTR.` Please see `GASELCTR` for a description of the model. Also, if "settings" are saved from `GENALG` this is the same as the options structure discussed in `GASELCTR`.

Examples

```
>> x2 = mncn(x);
>> y2 = mncn(y);
>> genalg(x2,y2)
```

See Also

`calibsel, fullsearch, gaselctr, genalgplot`

genalgplot

Purpose

Selected variable plot, color-coded by RMSECV for GA results.

Synopsis

```
indicies = genalgplot(fit,pop,spectrum,xaxis,xtitle)
indicies = genalgplot(results,spectrum,xaxis,xtitle)
```

Description

An interactive plotting routine which displays the results of a genetic algorithm (GA) analysis. `GENALGPLOT` can aid in identifying patterns of variables that improve model prediction (as estimated by RMSECV). The results of GA analysis include the final unique "population" which is a *M* by *N* matrix where *M* is the number of members in the population and *N* is the number of original variables in the predictor block. Each row (member) of the population corresponds to a regression model where a column with a "1" indicates that variable was included in the model and a "0" indicates that the variable was not included. The RMSECV for each model characterized its prediction performance.

The user selects a subset of the population from a plot of RMSECV versus the total number of included variables for each member of the population. The selected results are displayed in a plot that shows which variables were included for each member in the subset and its corresponding RMSECV. The plot is sorted with the best-performing individuals at the bottom of the plot and the worst at the top.

`GENALGPLOT` is most useful when many replicate GA runs have been performed (see `GENALG` and `GASELCTR`) with low settings on the maximum number of generations (`maxgenerations`) or Found at convergence (`convergence`).

Required inputs are `fit`, the RMSECV fit results from `GASELCTR` (or `rmsecv` from a `GENALG` results structure), and pop, the logical matrix of included variables for all individuals in the final population (or `icol` from a `GENALG` results structure). Optional inputs include `spectrum`, a spectrum to plot on the final "included variables" plot for reference, `xaxis`, the variable axis scale, and `xtitle`, the x-axis label for the final plot (e.g. xaxis units).

The one output is the indicies of the selected individuals (rows of `pop`).

Examples

Given the `GENALG` results structure `gamodel`, the following would plot the results:

```
genalgplot(gamodel.rmsecv,gamodel.icol)
```

See Also

`genalg, gaselctr`

getdatasource

Purpose

Extract summary information about a DataSet.

Synopsis

```
[out1, out2,...] = getdatasource(dataset1, dataset2,...)
```

Description

The input(s) `dataset1`, *dataset2*,... are dataset objects. `GETDATASOURCE` returns structures containing useful summary information about each DataSet including the contents of the DataSet fields: `name`, `author`, `date`, and `moddate`. Also returned in the structure is the size of the data field.

See Also

`dataset/dataset, dataset/subsref, modelstruct`

getpidata

Purpose

Uses the current PI connection to construct a DSO from 'taglist'.

Synopsis

```
[pidso, warnlog] = getpidata(taglist,startdate,enddate,options)
```

Description

This function requires the PI SDK (software developer kit) be installed. If only taglist is submitted and or date inputs are empty then a "snapshot" of the data is returned. Date inputs can be any PI supported value.

INPUTS

`taglist` =	Cell array of strings containing tags to query or excel file with one column of tag names.
`startdate` =	Start date/time to query or excel file with 2 columns (start and end dates). Each row will indicate a unique start/end and will be appended according to appenddir option setting.
`endtdate` =	End date/time to query.

OUTPUTS

pidso = dataset object of queried values or (if rawdata = 'on') a 1xn structure array with the following fields:

.tagname

.time

.value

With DSO returned queries, timestamps are returned in the .axisscale field. Matlab adjusted timestamps are reported in .axisscale{1,1}. The original UTC timestamps are reported in .axisscale{1,2}.

Options

options = structure array with the following fields:

`tagsearch:`	`[{'off'}	'on']` Show PI tag search gui.
`interpolate:`	`[{'interval'}	'total']` Governs interpolate settings, 'interval' is the time between data points in seconds. 'total' is the total number of points to retrieve.
`interpolateval:`	`{60}` Default is interval if 60 seconds.
`timeout:`	`{10}` Seconds to wait for server to return for each column of data.
`savefile:`	`{''}` File name to save output to.

`diplaywarnings:`	`[ 'off' | {'on'} ]` Show warning at command line after calculation.
`timecorrection:`	`{0}` Time in seconds to be added when converting PI timestamps to Matlab time.
`rawdata:`	`[ {'off'} | 'on' ]` Retrieve PI "compressed data" (actual Archive events) for given taglist. This will not use any interpolation and because data will likely be of different length, the result will be returned in a structure, not a dso.
`userservertime:`	`[ 'off' | {'on'} | local]` Governs how to convert Matlab timestamps (axisscale{1,1}). 'on' creates timestamps with timezone settings (e.g., daylight savings rules) applied. If set to 'off' then server time is used with no timezone rules applied. If set to 'local', local timezone is applied.
`appenddir:`	`[ {'mode 1'} | 'mode 3']` Mode to append to when using multiple time range inputs.
`lengthmatch:`	`[ 'min' | {'max'} | 'stretch' | 'fixed' ]` Defines how slabs should be concatenated (used only when appenddir = 'mode 3'): 'min' truncates all slabs to the shortest slab length. 'max' adds NaN's to the end of each slab to match the longest slab length. 'stretch' interpolates all slabs to match the length of the FIRST read slab. 'fixed' either truncates or infills all slabs to match a specific length specified in targetlength, below. All modes can also be adapted to match a minimum or maximum length using the "targetlength" option, below.
`targetlength:`	[] Optional target length (used only when appenddir = 'mode 3'). A non-empty value will be used in place of the default length defined by the lengthmatch option. If lengthmatch is 'min', this option defines the MAXIMUM length slab to allow. If lengthmatch is 'max', this option defines the MINIMUM length slab to allow. If lengthmatch is 'stretch', this option defines the target length. If lengthmatch is 'fixed' then this option defines the target length.

Examples

```
>> dso = getpidata('tagnames.xls','y-2d','t',options);

>> dso = getpidata('tagnames.xls','dates.xls',options);

>> dso = getpidata({'SINUSOID' 'BA:PHASE.1' 'BA:TEMP.1'},'y-
2d','t',options);
```

See Also

piconnectgui

glsw

Purpose

Calculate or apply Generalized Least Squares weighting.

Synopsis

```
modl = glsw(x,a);               %GLS on matrix
modl = glsw(x1,x2,a);           %GLS between two data sets
modl = glsw(x,y,a);             %GLS on matrix in groups based on y
modl = glsw(modl,a);            %Update model to use a new value
xt   = glsw(newx,modl,options);        %apply correction
xt   = glsw(newx,modl,a);              %apply correction
```

Description

Uses Generalized Least Squares to down-weight variable features identified from the singular value decomposition of a data matrix. The input data usually represents two or more measured populations which should otherwise be the same (e.g. the same samples measured on two different analyzers or using two different solvents) and can be input in one of several forms, as explained below. In all cases, the downweighting is performed by taking the eigenvectors and eigenvalues of the differences.

If the singular value decomposition (SVD) of the input matrix x is $\mathbf{X}=\mathbf{USV}^T$ then the deweighting matrix is estimated with the following pseudo-inverse $\mathbf{W}=\mathbf{U}\text{diag}(\text{sqrt}(1/(\text{diag}(\mathbf{S})/a^2+1)))\mathbf{V}^T$, where the center term defines $\mathbf{S}_{\text{inv}}$. The adjustable parameter a is used to scale the singular values prior to calculating their inverse. As a gets larger, the extent of deweighting decreases (because $\mathbf{S}_{\text{inv}}$ approaches 1). As a gets smaller (e.g. 0.1 to 0.001) the extent of deweighting increases (because $\mathbf{S}_{\text{inv}}$ approaches 0) and the deweighting includes increasing amounts of the the directions represented by smaller singular values.

A good initial guess for a is $1\text{x}10^{-2}$ but will vary depending on the covariance structure of **X** and the specific application. It is recommended that a number of different values be investigated using some external cross-validated metric for performance evalution.

An alternative method to use GLSW is in quantitative analysis where a continuous y-variable is used to develop pseudo-groupings of samples in X by comparing the differences in the corresponding y values. This is referred to as the "gradient method" because it utilizes a gradient of the sorted X and y blocks to calculate a covariance matrix. For more information on this method, see the chapter discussing Preprocessing in the PLS_Toolbox Manual.

For calibration, inputs can be provided by one of three methods:

1)	`x` =	data matrix containing features to be downweighted, and
	`a` =	scalar parameter limiting downweighting {default = `1e-2`}.
	Note:	If x is a dataset with classes, the differences within *each class* will be downweighted rather than the entire matrix. This reduces the within-class variation ignoring the between-class variation.
2)	`x1` =	a *M* by *N* data matrix and
	`x2` =	a *M* by *N* data matrix.
		The row-by-row differences between `x1` and `x2` will be used to estimate the downweighting.
	`a` =	scalar parameter limiting downweighting {`default = 1e-2`}.
3)	`x` =	a *M*x*N* data matrix,
	`y` =	column vector with *M* rows which specifies sample groups in `x` within which differences should be downweighted. Note that this method is identical to method (1) when classes of the X block are used to identify groups. The only difference is that these groupings are passed as a separate input. In fact, if y is empty, this defaults to method (1) above.
	`a` =	scalar parameter limiting downweighting {default = `1e-2`}.
4)	`x` =	a *M*x*N* data matrix,
	`y` =	column vector with *M* rows specifying a y-block continuous variable. In this input, the "gradient method" is used to identify similar samples and downweight differences between them. See also the gradientthreshold option below.
	`a` =	scalar parameter limiting downweighting {default = `1e-2`}.

An options structure can be used in place of (a) for any call or as the third output in an apply call. This structure consists of any of the fields:

`a`:	`[ 0.02 ]` scalar parameter limiting downweighting {default = 1e-2},
`applymean`:	`[ 'no' | {'yes'} ]` governs the use of the mean difference calculated between two instruments (difference between two instruments mode). When appling a GLS filter to data collected on the x1 instrument, the mean should NOT be applied. Data collected on the SECOND instrument should have the mean applied.
`gradientthreshold`:	`[ .25 ]` "continuous variable" threshold fraction above which the column gradient method will be used with a continuous y. Usually, when (y) is supplied, it is assumed to be the identification of discrete groups of samples. However, when calibrating, the number of samples in each "group" is calculated and the fraction of samples in "singleton" groups (i.e. in thier own group) is determined. fraction = (# Samples in Singleton Groups) / Total Samples If this fraction is above the value specified by this option, (y) is considered a continuous variable (such as a concentration or other property to predict). In these cases, the "sample similarity" (a.k.a. "column gradient") method of calculating the covariance matrix will be used. Sample similarity method determines the down-weighting required based mostly on samples which are the most similar (on the specified y-scale). Set to >=1 to disable and to 0 (zero) to always use.
`maxpcs`:	`[ 50 ]` maximum number of components (factors) to allow in the GLSW model. Typically, the number of factors in incuded in a model will be the smallest of this number, the number of variables or the number of samples. Having a limit set here is useful when derriving a GLSW model from a large number of samples and variables. Often, a GLSW model effectively uses fewer than 20 components. Thus, this option can be used to keep the GLSW model smaller in size. It may, however, decrease its effectiveness if critical factors are not included in the model.

When applying a `GLSW` model the inputs are `newx`, the x-block to be deweighted, and `modl`, a `GLSW` model structure.

Outputs are `modl`, a `GLSW` model structure, and `xt`, the deweighted x-block.

See Also

`pca, pls, preprocess, osccalc`

gram

Purpose

Generalized rank anihilation method.

Synopsis

```
[ord1,ord2,ssq,aeigs,beigs] = gram(a,b,tol,scl1,scl2,out)
```

Description

`GRAM` determines the joint invariant subspaces common to the two input matrices `a` and `b`, the ratio of their magnitudes `ssq`, and the response in both modes/orders `ord1` and `ord2`. `GRAM` assumes that the input matrices `a` and `b` are bilinear, *i.e.* are the summation over outer products.

Inputs are the two response matrices `a` and `b`, and the number of factors to calculate or tolerance on the ratio of smallest to largest singular value `tol`. Optional inputs *scl1* and *scl2* are scales to plot against when producing plots of the reponse in each mode/order. Optional input *out* suppresses plotting and printing of results to the command window when set to 0 {default `out = 1`}.

Outputs are the pure component responses in each mode `ord1` and `ord2`, the table of eigenvalues and their ratios `ssq`, and the eigenvalues for each matrix `aeigs` and `beigs`.

See Also

`mpca, parafac, parafac2, tld`

gscale

Purpose

Group/block scaling for a single or multiple blocks.

Synopsis

```
[gxs,mxs,stdxs] = gscale(xin,numblocks)
```

Description

`GSCALE` scales an input matrix `xin` such that the columns have mean zero, and variance in each block/sub-matrix relative to the total variance in `xin` equal to one. The purpose is to provide equal sum-of-squares weighting to each block in `xin`.

Inputs are a matrix `xin` (class "double") and the number of sub-matrices or blocks `numblocks`. Note that `size(xin,2)/numblocks` must be an integer. If `numblocks` is not included, it is assumed to 1 i.e. the matrix `xin` is treated as a single block.

If (numblocks) is 0 (zero) then automatic mode is used based on the dimensions of the (xin) matrix:

> If (xin) is a three-way array, it is unfolded (combining the first two modes as variables) and the size of the original second mode (size(xin,2)) is used as (numblocks). The output is re-folded back into the original three-way array.
>
> Note that the unfold operation is: xin = unfoldmw(xin,3);
>
> If (xin) is a two-way array, each variable is treated on its own and GSCALE is equivalent to autoscale (see the AUTO function).

Outputs are the scaled matrix (gxs), a rowvector of means (mxs), and a row vector of "block standard deviations" `stdxs`.

Examples

Scale a matrix `a` that has two blocks augmented together:

```
>> a = [[1 2 3; 4 5 6; 7 8 9] [11 12 13; 14 15 16; 17 18 19]]
a =
     1     2     3    11    12    13
     4     5     6    14    15    16
     7     8     9    17    18    19
>> [gxs,mxs,stdxs] = gscale(a,2);
>> gxs
gxs =
   -0.5774   -0.5774   -0.5774   -0.5774   -0.5774   -0.5774
         0         0         0         0         0         0
    0.5774    0.5774    0.5774    0.5774    0.5774    0.5774
>> mxs
mxs =
     4     5     6    14    15    16
>> stdxs
stdxs =
     3     3     3     3     3     3
```

See Also

`auto, gscaler, mncn, mpca, scale, unfoldm`

gscaler

Purpose

GSCALER Applies group/block scaling to submatrices of a single matrix.

Synopsis

```
gys = gscaler(xin,numblocks,mxs,stdxs)
xin = gscaler(gys,numblocks,mxs,stdxs,undo)
```

Description

Inputs are a matrix (xin) (class "double"), the number of sub-matrices/ blocks (numblocks), an offset vector (mxs), and a scale vector (stdxs).

See GSCALE for descriptions of (mxs) and (stdxs).

Note that size(xin,2)/numblocks must be a whole number.

When numblocks = 1, all variables are scaled as a single block.

When numblocks = 0, each variable is handled on its own and gscaler is equivalent to the SCALE function.

If the optional input (undo) is included with a value of 1 (one), then the input is assumed to be (gys) and is unscaled and uncentered to give the original (xin) matrix.

In a standard call, the output is the scaled matrix (gys). When undo is provided, the output is the unscaled original matrix (xin).

Examples

Scale a matrix `a` that has two blocks augmented together using `GSCALE`:

```
>> a = [[1 2 3; 4 5 6; 7 8 9] [11 12 13; 14 15 16; 17 18 19]];
>> [gxs,mxs,stdxs] = gscale(a,2);
>> gxs
gxs =
   -0.5774   -0.5774   -0.5774   -0.5774   -0.5774   -0.5774
         0         0         0         0         0         0
    0.5774    0.5774    0.5774    0.5774    0.5774    0.5774
>> mxs
mxs =
     4     5     6    14    15    16
>> stdxs
stdxs =
     3     3     3     3     3     3
```

Now scale a new matrix b that has two blocks augmented together:
```
>> b = [[2 3 4; 4 5 6; 6 7 8] [10 11 12; 14 15 16; 18 19 20]]
b =
     2     3     4    10    11    12
     4     5     6    14    15    16
     6     7     8    18    19    20
>> gys = gscaler(b,2,mxs,stdxs)
gys =
   -0.3849   -0.3849   -0.3849   -0.7698   -0.7698   -0.7698
         0         0         0         0         0         0
    0.3849    0.3849    0.3849    0.7698    0.7698    0.7698
```

See Also

auto, gscale, mncn, mpca, scale, unfoldm

gselect

Purpose

Selects objects in a figure (various selection styles).

Synopsis

```
selected = gselect(mode,TargetHandle,options)
[x,y]    = gselect(mode,TargetHandle,options)
```

Description

`GSELECT` is a general utility which allows user-selection of plotted objects (points, line segments, areas of images, etc.). A variety of selection modes can be used on various types of plots. Each mode allows the user to select an area or range of the current axes. After selection is complete, the function returns a cell array that contains one cell for each line or image object on the axes. These cells contain a binary (true/false) array representing the selected points of each object.

The input *mode* is a string representing the selection mode. This governs how `GSELECT` selects objects in a figure. *mode* can be one of the following strings {default = 'rbbox'}:

`'x'`:	select a single x-axis position (snaps-to line x-data),
`'y'`:	select a single y-axis position (snaps-to line y-data),
`'xs'`:	select range of x-axis positions (snaps-to line x-data),
`'ys'`:	select range of y-axis positions (snaps-to line y-data),
`'rbbox'`:	select points inside a standard rubber-band box {default },
`'polygon'`:	select points inside a polygon (user selects corners),
`'circle'`:	select points inside a circle,
`'ellipse'`:	select points inside an ellipse,
`'lasso'`:	select points inside a lasso,
`'paint'`:	drag a broad line across points for selection,
`'nearest'`:	select single nearest point,
`'nearests'`:	select multiple single (nearest) points,
`'all'`:	selects all points (no user interaction required), and
`'none'`:	selects no points (no user interaction required).

Optional input *TargetHandle* is the handle or handles of objects to test for selection. The default is all lines, patches, surfaces, and images.

The output is a cell array `selection`. Each cell in `selection` will be equal in length to the data used to create the corresponding object. For example, if a vector containing 30 points was plotted, the resulting cell will be a vector of 30 binary values. Each selected point on that object will be represented by a value of 1 (one) in the cell, unselected objects by a value of 0 (zero).

If two outputs `[x,y]` are requested, `GSELECT` does not test objects for selection and simply returns the x and y points defining the selected area.

Options

`options` =	a structure array with the following fields:
`modal:`	`[ {'Flase'} | 'True' ]` Governs window's "modal" nature. Note that some systems will not allow modal windows.
`btndown:`	`[ {'Flase'} | 'True' ]` Should button be considered "down" at start?
`demo:`	`[ {'Flase'} | 'True' ]` Is this a demo call to gselect? (do not wait to exit)
`poslabel:`	`[ 'none' | {'xy'} ]` Governs what kind of axis position labels will be shown.
`helpbox:`	`[ 'off' | {'on'} ]` Governs display of the helpbox.
`helptextpre:`	`[ '' ]` Specifies text to prepend to helpbox message.
`helptextpost:`	`[ '' ]` Specifies text to append to end of helpbox message.
`helptext:`	`[ '' ]` Specifies alternate text to replace default helpbox message.
`modalwindow` =	optional flag which can be passed in place of "options" input. Controls window modal setting during the selection process (Keeps other windows from interrupting process) A value of 1 sets options.modal to 'true'.

Examples

Example 1. Plot a vector of 10 random values and let the user select from these points using the standard rubber-band box.

```
plot(randn(10,3), randn(10,3), '.'); slct = gselect('rbbox')
```

The output will be something like:

```
slct =
    [1x10 uint8]
>> slct{1}
ans =
     0     0     0     0     1     1     0     1     0     0
>> find(slct{1})
ans =
     5     6     8
```

indicating that points 5, 6 and 8 were selected by the user.

Example 2. Plot a small image and let the user select a sub-range using the polygon tool.

```
imagesc(randn(6,6)); slct = gselect('polygon')
```

The output will be something like:

```
slct =
    [6x6 uint8]
>> slct{1}
ans =
     0     0     0     0     0     0
     0     1     0     0     0     0
     0     1     1     1     0     0
     0     1     1     1     0     0
     0     1     0     1     1     0
     0     1     0     1     0     0
```

indicating the "n" shaped region selected by the user.

See Also

`plotgui`

helppls

Purpose

Starts the MATLAB help browser with PLS_Toolbox topics.

Synopsis

```
helppls
```

Description

`HELPPLS` brings up the MATLAB help browser with a list of topics for installing and using the PLS_Toolbox. To access a particular topic simply click on its text.

Use the arrow buttons in the upper left corner of the window to navigate forward and backward (similar to a web browser). Some of the Topics may link you to a Documentation page about a particular function in the PLS_Toolbox. From here you can navigate to related topics by clicking on See Also items or to the next topic (in alphabetical order) by clicking its text in the yellow highlighted header/footer section.

See Also

`readme`

hline

Purpose

Place a horizontal line in an existing figure.

Synopsis

```
hline(y,lc)
h = hline(y,lc)
```

Description

`HLINE` draws a horizontal line on an existing figure from the left axis to the right axis at a height, or heights, defined by `y` which can be a scalar or vector. If no input is used for `y` the default vaule is zero. The optional input variable *lc* can be used to define the line style and color as in normal plotting.

Examples

```
hline(1.4,'--b')
```

plots a horizontal dashed blue line at `y = 1.4`.

See Also

`dp, ellps, plot, plttern, vline, zline`

ipls

Purpose

IPLS Interval PLS and forward/reverse MLR variable selection.

Synopsis

```
results = ipls(X,Y,int_width,maxlv,options)
results = ipls(X,Y,int_width,maxlv,numintervals,options)
[use,fit,lvs,intervals,intcv,intlv] =
   ipls(X,Y,int_width,maxlv,options)
```

Description

Performs forward or reverse selection of variable windows based on the RMSECV obtained for each individual window ("intervals") of variables. Multiple windows can also be selected iteratively by modifying the options.numintervals options. The "algorithm" option allows this function to behave as an IPLS or IPCR algorithm or a forward/reverse MLR variable selection algorithm. The default is PLS but options.algorithm = 'mlr' changes to MLR mode. See other options below.

Inputs are (X,Y) the X and Y data, (int_width) the interval i.e. window width in variables and (maxlv) the maximum number of latent variables to use in any model (maxlv has no impact if options.algorithm = 'mlr'). Note that excluding a variable in X will prevent it from being used in any model.

If options.plots is 'final', a plot is given of the minimum RMSECV versus window center. Windows which were used are indicated in blue, windows which were excluded are indicated in red. The number of latent variables (LVs) used to assess each interval (the model size that gives the indicated RMSECV) is shown at the bottom of each interval's bar, inside the axes. The best RMSECV that can be obtained using all intervals is shown as a dashed red line (all-interval RMSECV). The number of LVs used in this model is shown on the right of the axes. If this number of LVs (all-interval model) is different from the number used for the best model of the selected interval(s) (selected-interval model) then a dashed magenta line will indicate the RMSECV obtained when using all intervals but at the selected-interval model size. The mean sample is superimposed on the plot for reference.

INPUTS:

`X` =	X-block,
`Y` =	Y-block, and
`int_width` =	the interval (window width in variables)
`maxlv` =	the maximum number of latent variables to use in any model.

NOTE that excluding a variable in X will prevent it from being used in any model.

OUTPUTS:

When a single output is requested, the output is a structure with the following fields:

`use:`	the final selected indices which gave the best model,
`fit:`	the RMSECV for the selected indicies,
`lvs:`	the number of latent variables which gives the best fit,
`intervals:`	a matrix containing the indicies used for each interval.
`intcv:`	the RMSECV in the last selection cycle for all intervals (these values were used to select the last interval).
`intlv:`	the number of latent variables used in the model which gave the RMSECV values returned in intcv.

Optionally, with multiple outputs, these vaiables will be returned as single outputs (not in structure format) in the order shown above.

Options

`options` =	options structure containing the fields:
`display:`	`[ 'off' | {'on'} ]`, governs level of display to command window,
`plots:`	`[ 'none' | {'final'} ]`, governs level of plotting,
`mode:`	`[{'forward'} | 'reverse' ]` Defines action to be performed with each interval. 'forward' mode: the RMSECV calculated for each interval represents how well the y-block can be predicted using ONLY the variables included in the interval. 'reverse' mode: the RMSECV calculated for each interval represents how well the y-block can be predicted when the given interval of variables are removed from the range of included X variables. NOTE that excluding a variable in X will prevent it from being used in any model.
`algorithm:`	`[{'pls'} | 'pcr' | 'mlr' ]` Defines regression algorithm to use. Selection is done for the specific algorithm. Note that when MLR is used, input (int_width) is most often = 1 (single variable per window).
`numintervals:`	`{ [1] }` Number of intervals to select or remove. If (num_intervals) is Inf, intervals are iteratively selected and added/removed until no improvement in RMSECV is observed. NOTE: this can also be set by passing as a scalar value before, or in place of, the options structure. When passed this way, any value passed in the options structure will be ignored.
`mustuse:`	`[ ]` A vector of variable indices which MUST be used in all models. These variables will always be included in any model, whether or not they are included in the current interval.

`stepsize:` `[ ]` Distance between interval centers. An empty matrix gives the default spacing in which intervals do not overlap (stepsize = int_width).

`preprocessing:` defines preprocessing and can be one of the following:

(a) One of the following strings:

'none' : no preprocessing {default}

'meancenter' : mean centering

'autoscale' : autoscaling

(b) A single preprocessing structure defined using the function preprocess. The same preprocessing structure will be used on both the X and Y blocks.

(c) A cell containing two preprocessing structures {pre pre} one for the X block and one for the Y block.

`cvi:` `{'vet' [ ] 1}` Three element cell indicating the cross-validation leave-out settings to use {method splits iterations}. For valid modes, see the "cvi" input to crossval. If splits (the second element in the cell) is empty, the square root of the number of samples will be used. cvi can also be a vector (non-cell) of indices indicating leave-out groupings (see crossval for more info).

See Also

`gaselctr, genalg`

jcampreadr

Purpose

Reads a JCAMP file into a DataSet object.

Synopsis

```
      data = jcampreadr('filename.dx')
```

Description

Input is the filename of a JCAMP file to read. If omitted, the user is prompted for a file. Currently this reader will only read files of type:

INFRARED SPECTRUM

LINK

Output (data) is a DataSet object containing the spectrum or spectra from the file (or an empty array if no data could be read)

See Also

`spcreadr, xclreadr`

jmlimit

Purpose

Confidence limits for Q residuals via Jackson-Mudholkar.

Synopsis

```
rescl = jmlimit(pc,s,cl)
```

Description

`JMLIMIT` estimates confidence limits for Q residuals based on the Jackson-Mudholkar method. See Jackson, J.E., “A User’s Guide to Principal Components”, John Wiley & Sons, New York, NY (1991), and the discussion in the Chemometrics Tutorial on PCA.

Inputs are the number of PCs used `pc`, the vector of eigenvalues `s`, and the confidence limit `cl` expressed as a fraction (e.g. 0.95). Note that for a PCA model structure, `model`, that the eigenvalues can be found in `model.detail.ssq(:,2)`.

The output `rescl` is the confidence limit based on the method of Jackson and Mudholkar. See `CHILIMIT` for an alternate method of residual limit calculation based on chi squared.

Examples

```
rescl = jmlimit(2,ssq(:,2),0.95);
```

For a PCA model contained in the structure `model`:

```
rescl = jmlimit(4,model.detail.ssq(:,2),0.99);
```

See Also

`chilimit, analysis, pca, residuallimit`

knn

Purpose

K-nearest neighbor classifier.

Synopsis

```
pclass = knn(xref,xtest,k,options); %make prediction without model
pclass = knn(xref,xtest,options); %use default k

model = knn(xref,k,options) %create model
pclass = knn(xref,xtest,k,options) %apply model to xtest
pclass = knn(xtest,model,options)
```

Description

Performs kNN classification where the "k" closest samples in a reference set vote on the class of an unknown sample based on distance to the reference samples. If no majority is found, the unknown is assigned the class of the closest sample (see input options for other no-majority behaviors).

INPUTS:

`xref` =	a DataSet object of reference data,
`xtest` =	a DataSet object or Double containing the unknown test data.

OPTIONAL INPUTS:

`model` =	an optional standard KNN model structure which can be passed instead of xref (note order of inputs: (xtest,model)) to apply model to test data.
`k` =	number of components {default = rank of X-block}.

OUTPUTS:

`pclass` =	an optional number of neighbors to use in vote for class of unknown {default = 3}. If k=1, only the nearest sample will define the class of the unknown.
`model` =	if no test data (xtest) is supplied, a standard model structure is returned which can be used with test data in the future to perform a prediction.

Options

`options`	= structure array with the following fields :
`display`:	[`'off'` | {`'on'`}] governs level of display to screen.

`preprocessing`: { [ ] } A cell containing a preprocessing structure or keyword (see PREPROCESS). Use {'autoscale'} to perform autoscaling on reference and test data.

`nomajority`: ['error' | {'closest'} | class_number] Behavior when no majority is found in the votes. 'closest' = return class of closest sample. 'error' = give error message. class_number (i.e. any numerical value) = return this value for no-majority votes (e.g. use 0 to return zero for all no-majority votes)

See Also

`analysis, cluster, plsda, simca`

lamsel

Purpose

Determine indices of wavelength axes in specified ranges.

Synopsis

```
    inds = lamsel(freqs,ranges,out)
```

Description

`LAMSEL` determines the indices of the elements of a wavelength or wavenumber axis within the ranges specified. Inputs are the wavelength or wavenumber axis `freqs` and an *m* by 2 matrix defining the wavelength ranges to select `ranges`.

An optional input *out* suppresses displaying information to the command window when set to 0.

The output `inds` is a vector of indices of channels in the specified range(s) inclusive.

Examples

```
inds = lamsel(lamda,[840 860; 1380 1400]);
```

outputs the indices of the elements of lamda between 840 and 860 and between 1380 and 1400.

See Also

`baseline, savgol, specedit`

lddlgpls

Purpose

Provide an "load" dialog box for use with GUIs.

Synopsis

```
    [value,name,source] = lddlgpls(klass,message)
```

Description

`LDDLPLS` creates a dialog box that allows a function to load variables from the workspace or a MATLAB "mat" file into the function workspace. The location of the file to load from can be selecetd from the folders listed in the file list and from the "Look in" menu at the top of the dialog box. Optional input *klass* allows the user to select the workspace variable of class to load. Valid values for *klass* are:

`'double'`:	loads 2-way DOUBLE variable {default},
`'cell'`:	loads CELL variable,
`'char'`:	loads 2-way CHAR variable,
`'struct'`:	loads a STRUCT variable,
`'dataset'`:	loads a DATASET object,
`'doubdataset'`:	loads a 2-way DOUBLE or DATASET, or
`'*'`:	loads any class and size variable.

Optional text input *message* places a message in the load dialog box.

Outputs include `value` the value of the selected variable, `name` the original name of the variable, and `location` the filename from which the variable was loaded (will be empty if loaded from the base workspace).

See Also

`erdlgpls, svdlgpls`

leverag

Purpose

Calculates sample leverage.

Synopsis

```
      lev = leverag(x,rinv)
```

Description

`LEVERAG` calculates the sample leverage according to

`lev(i,1) = x(i,:)*inv(x'*x)*x(i,:)'.`

Note that the leverage calculation should include a term for calculation of the offset (*e.g.* see Draper, N. and Smith, H., “Applied Regression Analysis, Second Edition”, John Wiley & Sons, New York, N.Y., 1981), but the above formula contains the salient information. This, in effect, assumes that the data have been mean centered and the constant term related to estimating the offset has been ignored. If `x'*x` is replaced by `x'*x/(m-1)` where `m` is the number of rows of `x`, and `x` has been mean centered then this is the equation for Hotelling's T^2 statistic.

Note that if `x` is not of full rank then `inv(x'*x)` won't exist, or if `x` is nearly rank deficient then calculation of the inverse will be unstable. In these cases, the scores from principal components analysis can be used.

If the optional input *rinv* is supplied then the leverage is calculated as

`lev(i,1) = x(i,:)*rinv*x(i,:)'.`

See Also

`doptimal, figmerit, pls, pcr`

lmoptimize

Purpose

Levenberg-Marquardt non-linear optimization.

Synopsis

```
[x,fval,exitflag,out] = lmoptimize(fun,x0,options,params)
```

Description

Starting at (`x0`) `LMOPTIMIZE` finds (`x`) that minimizes the function defined by the function handle (`fun`) where (`x`) has N parameters. The function (`fun`) must supply the Jacobian and Hessian i.e. they are not estimated by `LMOPTIMIZE` (an example is provided in the Algorithm Section below).

INPUTS:

`fun` =	function handle, the call to `fun` is `[fval,jacobian,hessian] = fun(x)` [see the Algorithm Section for tips on writing (`fun`)] (`fval`) is a scalar objective function value, (`jacobian`) is a N x1 vector of Jacobian values, and (`hessian`) is a N x N matrix of Hessian values.
`x0` =	N x1 initial guess of the function parameters.

OPTIONAL INPUTS:

`options` =	discussed below in the Options Section.
`params` =	comma separated list of additional parameters passed to the objective function (`fun`), the call to (`fun`) is `[fval,jacobian,hessian] = fun(x,params1,params2,...)`.

OUTPUTS:

`x` =	N x1 vector of parameter value(s) at the function minimum.
`fval` =	scalar value of the function evaluated at (`x`).
`exitflag` =	describes the exit condition with the following values
`1`:	converged to a solution (`x`) based on one of the tolerance criteria
`0`:	convergence terminated based on maximum iterations or maximum time.
`out` =	structure array with the following fields:
`critfinal`:	final values of the stopping criteria (see `options.stopcrit` below).
`x`:	intermediate values of (`x`) if `options.x=='on'`.
`fval`:	intermediate values of (`fval`) if `options.fval=='on'`.

`Jacobian`: last evaluation of the Jacobian if `options.Jacobian=='on'`.

`Hessian`: last evaluation of the Hessian if `options.Hessian=='on'`.

Algorithm

The objective function is defined as $f(\mathbf{x})$, where $\mathbf{x}$ is a N x1 vector. The Jacobian $\mathbf{J}$ and the symmetric Hessian $\mathbf{H}$ are defined as

$$\mathbf{J}=\frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}}=\begin{bmatrix}\partial f/\partial x_1\\ \partial f/\partial x_2\\ \vdots\\ \partial f/\partial x_N\end{bmatrix} \qquad \mathbf{H}=\frac{\mathrm{d}}{\mathrm{d}\mathbf{x}}\left(\frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}}\right)^T=\begin{bmatrix}\partial^2 f/\partial x_1^2 & \partial^2 f/\partial x_1\partial x_2 & \cdots & \partial^2 f/\partial x_1\partial x_N\\ \partial^2 f/\partial x_2\partial x_1 & \partial^2 f/\partial x_2^2 & \cdots & \partial^2 f/\partial x_2\partial x_N\\ \vdots & \vdots & \ddots & \vdots\\ \partial^2 f/\partial x_N\partial x_1 & \partial^2 f/\partial x_N\partial x_2 & \cdots & \partial^2 f/\partial x_N^2\end{bmatrix}.$$

Two types of calls to the function `fun` are made. The first type is used often and is a simple evaluation of the function at x given by

```
fval = fun(x,params1,params2,...);
```

The second type of call returns the Jacobian and Hessian

```
[fval,jacobian,hessian] = fun(x,params1,params2,...);
```

Therefore, to enhance the speed of the optimization, the M-file that evaluates the objective function should only evaluate the Jacobian and Hessian if `nargout>1` as in the following example.

```
function [p,p1,p2] = banana(x)
%BANANA Rosenbrock's function
%  INPUT:
%    x  = 2 element vector [x1 x2]
%  OUTPUTS:
%    p  = P(x)  = 100(x1^2-x2)^2 + (x1-1)^2
%    p1 = P'(x) = [400(x1^3-x1x2) + 2(x1-1); -200(x1^2-x2)]
%    p2 = P"(x) = [1200x1^2-400x2+2, -400x1; -400x1, 200]
%    p  is (fval)
%    p1 is (jacobian)
%    p2 is (Hessian)
%
%I/O: [p,p1,p2] = banana(x);

x12 = x(1)*x(1);
x13 = x(1)*x12;
x22 = x(2)*x(2);
alpha = 10; %1 is not very stiff, 10 is The stiff function

p   = 10*alpha*(x13*x(1)-2*x12*x(2)+x22) + x12-2*x(1)+1;
if nargout>1
  p1  = [40*alpha*(x13-x(1)*x(2)) + 2*(x(1)-1);
         -20*alpha*(x12-x(2))];
  p2  = [120*x12-40*x(2) + 2, -40*x(1);
         -40*x(1),            20]*alpha;
```

```
    end
```

This example shows that the Jacobian and Hessian are not evaluated unless explicitly called for by utilizing the `nargout` command. Since estimating $\mathbf{J}$ (output `p1`) and $\mathbf{H}$ (output `p2`) can be time consuming, this coding practice is expected to speed up the optimization.

A single step in a Gauss-Newton (G-N) optimization, $\Delta\mathbf{x}_k$ is given as

$$\Delta\mathbf{x}_k = -\mathbf{H}_k^{-1}\mathbf{J}_k$$

where the index k corresponds to the step number.

A problem with the G-N methods is that the inverse of the Hessian may not exist at every step, or it can converge to a saddle point if the Hessian is not positive definite [T.F. Edgar, D.M. Himmelblau, Optimization of Chemical Processes, 1st ed., McGraw-Hill Higher Education, New York, NY, 1988]. As an alternative, the Levenberg-Marquardt (L-M) method was used for CMF [K. Levenberg, Q. Appl. Math 2 (1944) 164; D. Marquardt, S.I.A.M. J. Appl. Math 11 (1963) 431; Edgar et al.]. A single step for the L-M method is given by

$$\Delta\mathbf{x}_k = -\left(\mathbf{H}_k + \theta\mathbf{I}\right)^{-1}\mathbf{J}_k$$

where θ is a damping parameter and $\mathbf{I}$ is a N x N identity matrix. This has a direct analogy to ridge regression [A.E. Hoerl, R.W. Kennard, K.F. Baldwin, Commun. Statist. 4 (1975) 105] with θ, the ridge parameter, constraining the size of the step. This method is also called a damped G-N method [G. Tomasi, R. Bro, Comput. Stat. Data Anal. in press (2005)]. There are several details to implementing the L-M approach [M. Lampton, Comput. Phys. 11 (1997) 110]. Details associated with the `LMOPTIMIZE` function are discussed here.

At each iteration in the algorithm, the inverse of $\mathbf{H}_k + \theta\mathbf{I}$ must be estimated. As a part of this process the singular value decomposition (SVD) of $\mathbf{H}_k$ is calculated as

$$\mathbf{VSV}^T = \mathbf{H}_k \ .$$

Note that the left and right singular vectors are the same (and equal to $\mathbf{V}$) because the Hessian is symmetric. If the optimization surface is convex, $\mathbf{H}_k$ will be positive definite and the diagonal matrix $\mathbf{S}$ will have all positive values on the diagonal. However, the optimization problem may be such that this is not the case at every step. Therefore a small number α is added to the diagonal of $\mathbf{S}$ in an effort to ensure that the Hessian will always be positive definite. In the algorithm $\alpha = \mathbf{S}_{1,1} / ncond$, where $\mathbf{S}_{1,1}$ is the largest singular value and $ncond$ is the maximum condition number desired for the Hessian [$ncond$ is input as `options.ncond`]. This can be viewed as adding a small dampening to the optimization and is always included at every step. In contrast, an additional damping factor that is allowed to adapt at each step is also included. The adapting dampening factor is given by $\theta = \lambda_1 \mathbf{S}_{1,1}$ where the initial λ_1 is input to the algorithm as `options.lamb(1)`. It is typical that θ is much larger than α . The inverse for the L-M step is then estimated as

$$\left(\mathbf{H}_k + \theta \mathbf{I}\right)^{-1} \approx \mathbf{V}\left(\mathbf{S} + \left(\theta + \alpha\right)\mathbf{I}\right)^{-1} \mathbf{V}^T$$

and is used to estimate a step distance $\Delta \mathbf{x}_k$.

The ratio $r = \left[f\left(\mathbf{x}_k\right) - f\left(\mathbf{x}_k + \Delta \mathbf{x}_k\right)\right] / \left[-\mathbf{J}\Delta \mathbf{x}_k\right]$ is a measure of the improvement in the objective function relative to the improvement if the objective function decreased linearly. If $r < r_1$ then a line search is initiated [$r_1 > 0$ is a small number input as `options.ramb(1)`]. In this case, the damping factor λ_1 is increased (so that the step size is reduced) by setting $\lambda_1 = \lambda_1 / \lambda_2$ where $\lambda_2 < 1$ [λ_2 is input as `options.lamb(2)`], and a new step distance $\Delta \mathbf{x}_k$ is estimated. The ratio r is then estimated again. The damping factor λ_1 is increased until $r \geq r_1$ or the maximum number of line search steps $k_{\max}$ is reached [$k_{\max}$ is input as `options.kmax`]. (If λ_1 increases sufficiently, the optimization resembles a damped steepest decent method.) If the maximum number of line search steps $k_{\max}$ is reached, the step is "rejected" and only a small movement is made such that $\Delta \mathbf{x}_k = r_3 \Delta \mathbf{x}_k / \left|\Delta \mathbf{x}_k\right|$ [r_3 is input as `options.ramb(3)`].

If instead, the first estimate of the ratio is large enough such that $r \geq r_1$ then the line search is not initiated. If the ratio is sufficiently large such that $r > r_2$, where $r_2 > r_1$ then the damping factor is decreased by setting $\lambda_1 = \lambda_1 / \lambda_3$ where $\lambda_3 > 1$ [r_2 is input as `options.ramb(2)`; λ_3 is input as `options.lamb(3)`].

A new value for $\mathbf{x}$ is then estimated from $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$ and the next step is repeated from that point. The process is repeated until one of the stopping criteria [`options.stopcrit`] are met.

Options

`options` = structure array with the following fields:

`name`:	`'options'`, name indicating that this is an options structure.
`display`:	[`'off'` | {`'on'`}] governs level of display to the command window.
`dispfreq`:	*N*, displays results every *N*th iteration {default *N*=10}.
`stopcrit`:	[1e-6 1e-6 10000 3600] defines the stopping criteria as [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time in seconds)].
`x`:	[{`'off'`} | `'on'`] saves (x) at each step.
`fval`:	[{`'off'`} | `'on'`] saves (`fval`) at each step.
`Jacobian`:	[{`'off'`} | `'on'`] saves last evaluation of the Jacobian.

`Hessian`:	[{`'off'`} | `'on'`] saves last evaluation of the Hessian.
`ncond` =	1e6, maximum condition number for the Hessian (see Algorithm).
`lamb` =	[0.01 0.7 1.5], 3-element vector used for damping factor control (see Algorithm Section):
`lamb(1)`:	`lamb(1)` times the biggest eigenvalue of the Hessian is added to Hessian eigenvalues when taking the inverse; the result is damping.
`lamb(2)`:	`lamb(1) = lamb(1)/lamb(2)` causes deceleration in line search.
`lamb(3)`:	`lamb(1) = lamb(1)/lamb(3)` causes acceleration in line search.
`ramb` =	[1e-4 0.5 1e-6], 3-element vector used to control the line search (see Algorithm Section):
`ramb(1)`:	if fullstep < `ramb(1)`*[linear step] back up (start line search).
`ramb(2)`:	if fullstep > `ramb(2)`*[linear step], accelerate [change `lamb(1)` by the acceleration parameter `lamb(3)`].
`ramb(3)`:	if linesearch rejected, make a small movement in direction of L-M step `ramb(3)`*[L-M step].
`kmax` =	50, maximum steps in line search (see Algorithm Section).

Examples

```
options   = lmoptimize('options');
options.x = 'on';
options.display = 'off';
[x,fval,exitflag,out] = lmoptimize(@banana,x0,options);
plot(out.x(:,1),out.x(:,2),'-o','color', ...
 [0.4 0.7 0.4],'markersize',2,'markerfacecolor', ...
 [0 0.5 0],'markeredgecolor',[0 0.5 0])
```

See Also

`function_handle, lmoptimizebnd`

lmoptimizebnd

Purpose

Levenberg-Marquardt bounded non-linear optimization.

Synopsis

```
[x,fval,exitflag,out] =
  lmoptimizebnd(fun,x0,xlow,xup,options,params)
```

Description

Starting at (`x0`) `LMOPTIMIZE` finds (`x`) that minimizes the function defined by the function handle (`fun`) where (`x`) has N parameters. Inputs (`xlow`) and (`xup`) can be used to provide lower and upper bounds on the solution (`x`). The function (`fun`) must supply the Jacobian and Hessian i.e. they are not estimated by `LMOPTIMIZEBND` (an example description is provided in the Algorithm Section of the function `LMOPTIMIZE`).

INPUTS:

`fun` =	function handle, the call to `fun` is `[fval,jacobian,hessian] = fun(x)` [see the Algorithm section for tips on writing (`fun`)] (`fval`) is a scalar objective function value, (`jacobian`) is a Nx1 vector of Jacobian values, and (`hessian`) is a NxN matrix of Hessian values.
`x0` =	Nx1 initial guess of the function parameters.
`xlow` =	Nx1 vector of corresponding lower bounds on (`x`). See `options.alow`. If an element of `xlow == -inf`, the corresponding parameter in (`x`) is unbounded on the low side.
`xup` =	Nx1 vector of corresponding upper bounds on (`x`). See `options.aup`. If an element of `xup == inf`, the corresponding parameter in (`x`) is unbounded on the high side.

OPTIONAL INPUTS:

`options` =	discussed below in the Options Section.
`params` =	comma separated list of additional parameters passed to the objective function (`fun`), the call to (`fun`) is `[fval,jacobian,hessian] = fun(x,params1,params2,...)`.

OUTPUTS:

`x` =	N x1 vector of parameter value(s) at the function minimum.
`fval` =	scalar value of the function evaluated at (`x`).
`exitflag` =	describes the exit condition with the following values
`1`:	converged to a solution (`x`) based on one of the tolerance criteria
`0`:	convergence terminated based on maximum iterations or maximum time.
`out` =	structure array with the following fields:
`critfinal`:	final values of the stopping criteria (see `options.stopcrit` above).
`x`:	intermediate values of (`x`) if `options.x=='on'`.
`fval`:	intermeidate values of (`fval`) if `options.fval=='on'`.
`Jacobian`:	last evaluation of the Jacobian if `options.Jacobian=='on'`.
`Hessian`:	last evaluation of the Hessian if `options.Hessian=='on'`.

Algorithm

The algorithm is essentially the same as that discussed in `LMOPTIMIZE` and this section discusses only the two main differences between `LMOPTIMIZEBND` and `LMOPTIMIZE`.

The first difference is the addition of penalty functions used to enforce bounding. For example, the objective function used in `LMOPTIMIZE` is $f\left(\mathbf{x}\right)$, but the objective function used by `LMOPTIMIZEBND` is $f\left(\mathbf{x}\right)+g_{low}\left(\mathbf{x}\right)+g_{up}\left(\mathbf{x}\right)$. The penalty functions for upper, $g_{up}\left(\mathbf{x}\right)$, and lower bounds, $g_{low}\left(\mathbf{x}\right)$, are similar, so only the lower penalty function is described.

Define d as the lower boundary, γ_0 a small number (e.g. 0.001) and α_0 a large number [e.g. $-\ln(10^{-8})/\gamma_0$], then for a single parameter the lower penalty function is given as

$$g_{low}\left(x_i\right)=\begin{cases} e^{-\alpha_0\left(x_i-d-\gamma_0\right)} & \left(x_i-d-\gamma_0\right)\geq 0 \\ 1-\alpha_0\left(x_i-d-\gamma_0\right)+\tfrac{1}{2}\alpha_0^2\left(x_i-d-\gamma_0\right)^2 & \left(x_i-d-\gamma_0\right)<0 \end{cases} .$$

This function can be considered an external point function because it is defined outside the feasible region (outside the boundaries). It is continuous at the boundary and also has continuous first and second derivatives. This is in contrast to internal point functions such as a log function that is not continuous at the boundary [e.g. $\ln\left(0\right)$ is not continuous]. The first and second derivatives of the penalty function are given by

$$\frac{\mathrm{d}g_{low}\left(x_i\right)}{\mathrm{d}x_i}=\begin{cases} -\alpha_0 e^{-\alpha_0\left(x_i-d-\gamma_0\right)} & \left(x_i-d-\gamma_0\right)\geq 0 \\ -\alpha_0+\alpha_0^2\left(x_i-d-\gamma_0\right) & \left(x_i-d-\gamma_0\right)<0 \end{cases} \text{ and}$$

$$\frac{\mathrm{d}^2 g_{low}(x_i)}{\mathrm{d}x_i^2} = \begin{cases} \alpha_0^2 e^{-\alpha_0(x_i-d-\gamma_0)} & (x_i-d-\gamma_0) \ge 0 \\ \alpha_0^2 & (x_i-d-\gamma_0) < 0 \end{cases} .$$

The external point penalty function does not guarantee that a step won't move outside the boundaries into the infeasible region. It does, however provide a means for getting back inside the feasible region.

A second modification is included in the `LMOPTIMIZEBND` algorithm to avoid large steps outside the feasible region. If a step $\Delta\mathbf{x}_k$ is such that any $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta\mathbf{x}_k$ are outside the feasible region, the step size for those parameters is reduced. The reduction is 90% the distance of that parameter to the boundary. This typically changes the direction of the step $\Delta\mathbf{x}_k$.

Options

`options` = structure array with the following fields:

`name`:	'`options`', name indicating that this is an options structure.
`display`:	[`'off'` | {`'on'`}] governs level of display to the command window.
`dispfreq`:	*N*, displays results every *N*th iteration {default *N*=10}.
`stopcrit`:	[1e-6 1e-6 10000 3600] defines the stopping criteria as [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time in seconds)].
`x`:	[{`'off'`} | `'on'`] saves (x) at each step.
`fval`:	[{`'off'`} | `'on'`] saves (`fval`) at each step.
`Jacobian`:	[{`'off'`} | `'on'`] saves last evaluation of the Jacobian.
`Hessian`:	[{`'off'`} | `'on'`] saves last evaluation of the Hessian.
`ncond` =	1e6, maximum condition number for the Hessian (see Algorithm).
`lamb` =	[0.01 0.7 1.5], 3-element vector used for damping factor control (see Algorithm Section):
`lamb(1)`:	`lamb(1)` times the biggest eigenvalue of the Hessian is added to Hessian eigenvalues when taking the inverse; the result is damping.
`lamb(2)`:	`lamb(1) = lamb(1)/lamb(2)` causes deceleration in line search.
`lamb(3)`:	`lamb(1) = lamb(1)/lamb(3)` causes acceleration in line search.
`ramb` =	[1e-4 0.5 1e-6], 3-element vector used to control the line search (see Algorithm Section):
`ramb(1)`:	if fullstep < `ramb(1)`*[linear step] back up (start line search).
`ramb(2)`:	if fullstep > `ramb(2)`*[linear step], accelerate [change `lamb(1)` by the acceleration parameter `lamb(3)`].
`ramb(3)`:	if linesearch rejected, make a small movement in direction of L-M step `ramb(3)`*[L-M step].
`kmax` =	50, maximum steps in line search (see Algorithm Section).

alow: [ ], Nx1 vector of penalty weights for lower bound, {default = `ones(N,1)`}. If an element is zero, the corresponding parameter in (x) is not bounded on the low side.

aup: [ ], Nx1 vector of penalty weights for upper bound, {default = `ones(N,1)`}. If an element is zero, the corresponding parameter in (x) is not bounded on the high side.

Examples

```
options   = lmoptimize('options');
options.x = 'on';
options.display = 'off';
options.alow    = [0 0]; %x(1) and x(2) unbounded on low side
options.aup     = [1 0]; %x(1) bounded on high side and x(2)
                         % unbounded on high side

[x,fval,exitflag,out] = lmoptimize(@banana,x0,[0 0], ...
                          [0.9 0],options);
plot(out.x(:,1),out.x(:,2),'-o','color', ...
 [0.4 0.7 0.4],'markersize',2,'markerfacecolor', ...
 [0 0.5 0],'markeredgecolor',[0 0.5 0])
```

See Also

`function_handle, lmoptimize`

localmax

Purpose

Automated identification of local maxima

Synopsis

```
i0 = localmax(x,w)
```

Description

Finds maxima in windows of width (`w`). Wider windowing is used to avoid local maxima that might be due to noise. The default window width is `w`=3. This function is called by `PEAKFIND`.

INPUT:

`x` =	*MxN* matrix of measured traces containing peaks each 1*xN* row of (`x`) is an individual trace.

OPTIONAL INPUT:

`w` =	odd scalar window width for determining local maxima {default: `w = 3`}.

OUTPUT:

`i0` =	*Mx*1 cell w/ indices of the location of the major peaks for each of the *M* traces in each cell.

Examples

```
load nir_data
plot(spec1.axisscale{2},spec1.data(1,:))
i0 = localmax(spec1.data(1,:));
vline(spec1.axisscale{2}(i0{1}))

i0 = localmax(spec1.data(1,:),5);
vline(spec1.axisscale{2}(i0{1}),'r')
```

See Also

`fitpeaks, peakfind`

logdecay

Purpose

Variance scales a matrix using the log decay of the variable axis.

Synopsis

```
[sx,logscl] = logdecay(x,tau)
```

Description

Inputs are data to be scaled (x), and the decay rate (tau). Outputs are the variance scaled matrix (sx) and the log decay based variance scaling parameters (logscl).

For an m x n matrix 'x' the variance scaling used for variable 'i' is exp(-(i-1)/((n-1)*tau)). This gives a scaling of 1 on the first variable (i.e. no scaling), and a scaling of 1/exp(-1/tau) on the last variable. The following table gives example values of tau and the scaling on the last variable:

tau	scaling
1	2.7183
1/2	7.3891
1/3	20.0855
1/4	54.5982
1/5	148.4132

See Also

`autoscale, scale`

lsq2top

Purpose

Fits a polynomial to the top/(bottom) of data.

Synopsis

```
[b,resnorm,residual,options] = lsq2top(x,y,order,res,options)
```

Description

`LSQ2TOP` is an iterative least squares fitting algorithm. It is based on a weighted least squares approach where the weights are determined at each step. At initialization the weights are all 1, then a polynomial is fit through the data cloud using least squares. When fitting to the top of a data cloud, data points with a residual significantly below a defined limit (i.e. the points below the polynomial fit line) are given a small weighting. Therefore, on subsequent iterations these data points are weighted less in the fit, and the fit line moves to fit to the top of the data cloud.

Input `x` is the independent variable e.g. a *M*x*1* vector corresponding to a frequency or wavelength axis. Input `y` is the dependent variable e.g. a *M*x*1* vector corresponding to a measured spectrum. Input `order` is a scalar defining the order of polynomial to be fit e.g. y = P(x), and `res` is a scalar approximation of the fit residual e.g. noise level. Input `options` is discussed below. Note that the function can be used to fit to the top or bottom of a data cloud by changing `trbflag` in `options`.

The outputs are `b`, the regression coefficients [highest order term corresponds to `b(1)` and the intercept corresponds to `b(end)`], `resnorm` is the squared 2-norm of the residual, and `residual` is the fit residuals = `y` - P(x). The `options` ouput is the input `options` echoed back, the field initwt may have been modified.

Options

`options`	= structure array with the following fields :
`display:`	[`'off'` | {`'on'`}] governs level of display to command window.
`trbflag:`	[`'top'` | {`'bottom'`}] top or bottom flag, tells algorithm to fit the polynomials, y = P(x), to the top or bottom of the data cloud.
`tsqlim:`	[`0.99`] limit that governs whether a data point is significantly outside the fit residual defined by input `res`.
`stopcrit:`	[`1e-4 1e-4 1000 360`] stopping criteria, iteration is continued until one of the stopping criterion is met: [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time [seconds])].
`initwt:`	[ ] empty or Mx1 vector of initial weights (0<=w<=1).

Algorithm

For `order = 1` and fitting to the top of a data cloud, `LSQ2TOP` finds the vector $\mathbf{b} = \begin{bmatrix} b_1 & b_2 \end{bmatrix}$ that minimizes $\left(\mathbf{y} - \mathbf{x}b_1 - \mathbf{1}b_2\right)^T \mathbf{W}\left(\mathbf{y} - \mathbf{x}b_1 - \mathbf{1}b_2\right)$ where $\mathbf{W}$ is a diagonal weighting matrix whose elements are initially 1 and then are modified on each subsequent iteration.

The weighting is determined by first estimating the residuals for each data point j as $residual_j = \mathbf{y}_j - \mathbf{x}_j b_1 - b_2$ and defining $t_j = residual_j / res$ where res is the input `res`. A corresponding t-statistic from a t-table is estimated using the following

```
      tsqst    = ttestp(1-options.tsqlim,5000,2);
```

where t_{table} is `tsqst`. The elements of $\mathbf{W}$ are then given by $w_j = 1\big/\left(0.5 + t_j / t_{table}\right)$ for data points with $t_j < t_{table}$, and is a 1 otherwise. Therefore, the weighting is smaller for points far below the fit line.

The procedure can be modified to fit to the bottom of a data cloud by changing `options.trbflag`.

See Also

`baseine, baselinew, fastnnls`

lsq2topb

Purpose

Fits a polynomial to the top/(bottom) of data.

Synopsis

```
[yi,resnorm,residual,options] = lsq2topb(x,y,order,res,options)
```

Description

For order=1 and fitting to top of data cloud, LSQ2TOPB finds (yi) that minimizes sum((W*(y - yi)).^2) where W is a diagonal weighting matrix given by:

```
>> tsq = residual/res; % (res) is an input
>> tsqst = ttestp(1-options.tsqlim,5000,2); % T-test limit from table
>> ii = find(tsq<-tsqst); % finds residuals below the line
>> w(ii) = 1./(0.5+tsq(ii)/tsqst); %de-weights pts significantly below line
```

i.e. w(ii) is smaller for residuals far below/(above) the fit line.

INPUTS:

`x` =	independent variable Mx1 vector.
`y` =	dependent variable, Mx1 vector.
`order` =	order of polynomial [scalar] for polynomial function of input (x). If (order) is empty, (options.p) must contain a MxK matrix of basis vectors to fit in lieu of polynomials of (x).
res =	approximate fit residual [scalar].

OPTIONAL INPUTS:

k =	number of components {default = rank of X-block}, and

OUTPUTS:

`yi` =	the fit to input (y).
`resnorm` =	squared 2-norm of the residual.
`residual` =	y - yi.

Options

`options`	= structure array with the following fields :
`p`:	[ ] If (options.p) is empty, input (order) must be >0. Otherwise, options.p is a MxK matrix of basis vectors.
`smooth`:	[ ] if >0 this adds smoothing by adding a penalty to the magnitude of the 2nd derivative. (empty or <=0 means no smooth).
`display`:	[`'off'` | {`'on'`}] governs level of display to command window.

`trbflag`:	[`{'top'} | 'bottom' | 'middle'`] flag that tells algorithm to fit (yi) to the top, bottom, or middle of the data cloud.
`tsqlim`:	[`0.99`] limit that govers whether a data point is outside the fit residual defined by input (res).
`stopcrit`:	[`1e-4 1e-4 1000 360`] stopping criteria, iteration is continued until one of the stopping criterion is met [(rel tol) (abs tol) (max # iterations) (max time [seconds])].
`initwt`:	[ ] empty or Mx1 vector of initial weights (0<=w<=1).

See Also

`baseine, baselinew, fastnnls`

lwrpred

Purpose

Predictions based on locally weighted regression models.

Synopsis

```
ypred = lwrpred(xnew,xold,yold,lvs,npts,out)
[ypred,extrap] = lwrpred(xnew,xold,yold,lvs,npts,out)
```

Description

`LWRPRED` makes new sample predictions `ypred` for a new matrix of independent variables `xnew` based on an existing data set of independent variables `xold`, and a vector of dependent variables `yold`. Predictions are made using a locally weighted regression model defined by the number principal components used to model the independent variables `lvs` and the number of points defined as local `npts`.

Optional input *out* suppresses printing of the results when set to 0 {default = 1}. Additional output (extrap), a vector equal in length to number of samples in xnew, is non-zero when the given sample was predicted by extrapolating outside of the range of y-values which were used in the model. The value represents the distance (in y-units) extrapolated outside of the modeled samples. For example, a value of -0.3 indicates that the given sample was predicted by extrapolating 0.3 y-units below the lowest modeled sample in yold.

Note: Be sure to use the same scaling on new and old samples *i.e.* `xnew` must be scaled the same as `xold`!

Options

`options` =	a structure array with the following fields:
`display:`	`[ 'off' | {'on'} ]` governs level of display.
`alpha:`	`[ 0-1 ]` Weighting of y-distances in selection of local points. 0 = do not consider y-distances {default}, 1 = consider ONLY y-distances,
`iter:`	`[ {5} ]` Iterations in determining local points. Used only when alpha > 0 (i.e. when using y-distance scaling),
`preprocessing:`	`{ 2 2 }` Two element cell array defining preprocessing to use on data. First element of cell defines x-block preprocessing, second element defines y-block preprocessing. Options are: 0 = no scaling or centering 1 = mean center only 2 = autoscale (default) For example: {1 2} performs mean centering on x-block and autoscaling on y-block,

`algorithm:` `[ {'globalpcr'} | 'pcr' | 'pls' ]` Method of regression after samples are selected. 'globalpcr' performs PCR based on the PCs calculated from the entire calibration data set but a regression vector calculated from only the selected samples. 'pcr' and 'pls' calculate a local PCR or PLS model based only on the selected samples.

`reglvs:` `[ ]` Used only when algorithm is 'pcr' or 'pls', this is the number of latent variables/principal components to use in the regression model, if different from the number used to select calibration samples. [] (Empty) implies LWRPRED should use the same number of latent variables in the regression as were used to select samples. NOTE: This option is NOT used when algorithm is 'globalpcr'.

See Also

`pls, polypls`

lwrxy

Purpose

Predictions based on locally weighted regression with y-distance weighting.

Synopsis

```
    ypred = lwrxy(xnew,xold,yold,lvs,npts,alpha,iter,out)
```

Description

NOTE: LWRXY is depreciated. Y-distance weighting should be accessed via the .alpha option of LWRPRED.

`LWRXY` makes new sample predictions `ypred` for a new matrix of independent variables `xnew` based on an existing data set of independent variables `xold`, and a vector of dependent variables `yold`. Predictions are made using a locally weighted regression model defined by the number principal components used to model the independent variables `lvs`, the number of points defined as local `npts`, the weighting given to the distance in y `alpha`, and the number of iterations to use `iter`.

Optional input *out* suppresses printing of the results when set to 0 {default = 1}.

Note: Be sure to use the same scaling on new and old samples *i.e.* `xnew` must be scaled the same as `xold`!

See Also

`lwpred, pls, polypls`

manrotate

Purpose

Graphical interface to manually rotate model loadings and investigate directions in the scores.

Synopsis

```
manrotate(model,lvs)
```

Description

`MANROTATE` shows a score vs. score scatter plot and model loadings and allows the user to "rotate" the loadings. The loadings (shown as two colored lines in the score/score plot) can be dragged through different angles observing the resulting loading shape in the loadings plot (Loadings are always kept orthogonal.)

This interface is useful to identify a loading "shapes" which point towards, and orthogonal to, a given sample cluster or direction.

The user clicks on the heavy lines in the scores plot and "drags" them to point in a selected direction. The loadings (shown on the right in the figure) are automatically updated to show the loading which accounts for the new direction in the scores plot. The rotated loading vectors can be saved to the workspace using the toolbar save button.

Inputs include a PCA, PLS, PCR, or other 2-way factor-based model, `model`, and an optional input, *`lvs`*, which is a two-element vector specifying which of the model factors should be plotted and rotated (default `= [1 2]` which plots factor 2 vs factor 1.)

See Also

`pca, pcr, pls, varimax`

matchvars

Purpose

Align variables of a dataset to allow prediction with a model.

Synopsis

```
[mxdata, unmap] = matchvars(model,xdata,options)
[mxdata, unmap] = matchvars(labels,xdata,options)
[mxdata, unmap] = matchvars(axisscale,xdata,options)
[mxdata, mydata, unmapx, unmapy] =
   matchvars(model,xdata,ydata,options)
rdata = matchvars(mdata,unmap)
```

Description

Given a standard model structure `model` MATCHVARS uses either the labels stored in the model or, if no labels exist, the axisscale in the model to rearrange or interpolate the variables of a dataset object so that the model can be applied to the data. If `model` is a regression model, both an X and a Y block may be passed for alignment. A Y block is not required, however.

MATCHVARS WITH LABELS: When variable labels exist in both the model and the data, the variables in `data` are rearranged to match the variable order in `model` based on the labels stored in the model. Any variables required by `model` that do not exist in `data` are returned as NaN (Not a Number). These will usually be automatically replaced by the prediction routine using REPLACE.

MATCHVARS WITH LABELS: When variable labels exist in both the model and the data, the variables in `data` are rearranged to match the variable order in `model` based on the labels stored in the model. Any variables required by `model` that do not exist in `data` are returned as NaN (Not a Number). These will usually be automatically replaced by the prediction routine using REPLACE.

When no labels exist in the supplied model, the axisscale is used to interpolate the data based on the setting of options.axismode (see below). Axis regions which require extrapolation are returned as NaN (Not a Number). These will usually be automatically replaced by the prediction routine using REPLACE.

If neither labels nor axisscales can be used to align variables, the dataset object is passed back without modification.

An ordinary cell or character array of strings representing labels to match or an ordinary vector representing an axisscale may be passed in place of `model`. Such labels or axisscale can only be used with a single dataset (i.e. xdata).

NOTE: if axisscale was used to interpolate new variables for mxdata or mydata, the unmap variable(s) will be linear vectors which simply return the original data.

INPUTS:

`model` =	a standard model structure OR a cell or character array of labels to match labels in xdata OR a vector of axisscale (e.g. wavelength, wavenumber, etc) to match xdata using axisscale.
`xdata` =	a dataset object containing the X-block data.

OPTIONAL INPUTS:

`ydata` =	a second dataset containing the Y-block data
`unmap` =	used only when performing an "undo" of a previous MATCHVARS call. This is a vector describing how to reorder the columns back to the original order, as output by the previous call to MATCHVARS. Can be used to re-order the outputs from a model, such as the T- or Q-contributions, back to the original data order.

OUTPUT:

`mxdata` =	adjusted ("matched") x-block data
`mydata` =	adjusted ("matched") y-block data (not returned if no y-data passed)
`unmapx` =	a vector describing how the original variable order can be obtained from the reordered data. This can be used on other model outputs such as residuals and T contributions rearranging them to be like the original data. Any column discarded from the original data will have an NaN in unmap. See the "reorder" type of call in I/O below.
`unmapy` =	same as unmapx but for the y-block (ydata) variable.
`rdata` =	reverted data - output only when matchvars is called with unmap as input.

Options

`options` =	a structure array with the following fields:
`axismode:`	`['discrete' |{'linear'}| 'spline']` a string defining the interpolation method to use for matching variables using axisscale. If 'discrete', axisscale values must be matched exactly by data. Any other axismode will be passed to interp1 to perform interpolation. See INTERP1 for interpolation options.

See Also

`interp1, modlpred, pcapro, replace, str2cell`

mcr

Purpose

Multivariate curve resolution with constraints.

Synopsis

```
model = mcr(x,ncomp,options)    %calibrate
model = mcr(x,c0,options)       %calibrate with explict initial guess
pred  = mcr(x,model,options)    %predict
options = mcr('options')
```

Description

MCR decomposes a matrix **X** as **CS** such that **X** = **CS** + **E** where **E** is minimized in a least squares sense. Inputs are the matrix to be decomposed x (size *m* by *n*), and either the number of components to extract, ncomp, or the explict initial guess, c0. If c0 is size *m* by *k*, where *k* is the number of factors, then it is assumed to be the initial guess for **C**. If c0 is size *k* by *n* then it is assumed to be the initial guess for **S**. If *m*=*n* then, c0 is assumed to be the initial guess for **C**. Optional input *options* is described below.

The output, model, is a standard model structure. The estimated contributionss **C** are stored in model.loads{2} and the estimated spectra **S** in model.loads{1}. Sum-squared residuals for samples and variables can be found in model.ssqresiduals{1} and model.ssqresiduals{2}, respectively. See the PLS_Toolbox manual for more information on the MCR method and models.

MCR, by default, uses the alternating least squares (ALS) algorithm. For details on the ALS algorithm and constraints available in MCR, see the ALS reference page.

When called with new data and a model structure, MCR performs a prediction (applies the model to the new data) returning the projection of the new data onto the previously recovered loadings (i.e. estimated spectra).

Options

`options` =	a structure array with the following fields:
`display:`	`[ 'off' | {'on'} ]` governs level of display to command window.
`plots:`	`[ 'none' | {'final'} ]` governs level of plotting.
`preprocessing:`	`{ [] }` preprocessing to apply to x-block (see PREPROCESS).
`blockdetails:`	`[ 'compact' | {'standard'} | 'all' ]` Extent of predictions and raw residuals included in model. 'standard' = none, 'all' x-block.
`initmethod:`	`['distslct']` initialization method.
`initmode:`	`[1 | 2]` mode of x for automatic initialization.
`confidencelimit:`	`[{0.95}]` Confidence level for Q limits.
`alsoptions:`	`['options']` options passed to ALS subroutine (see ALS).

The default options can be retreived using: `options = mcr('options');`.

See Also

`als, analysis, evolvfa, ewfa, fastnnls, mlpca, parafac, plotloads, preprocess`

mdcheck

Purpose

Missing Data Checker and infiller.

Synopsis

```
[flag,missmap,infilled] = mdcheck(data,options)
options = mdcheck('options')
```

Description

This function checks for missing data and infills it using a PCA model if desired. The input is the data to be checked `data` as either a double array or a dataset object. Optional input `options` is a structure containing options for how the function is to run (see below).

Outputs are the fraction of missing data `flag`, a map of the locations of the missing data as an unint8 variable `missmap`, and the data with the missing values filled in `infilled`. Depending on the plots option, a plot of the missing data may also be output.

Options

`options` =	a structure array with the following fields:
`frac_ssq`:	[{0.95}] desired fraction between 0 and 1 of variance to be captured by the PCA model,
`max_pcs`:	[{5}] maximum number of PCs in the model, if 0, then it uses the mean,
`meancenter`:	['no' | {'yes'}], tells whether to use mean centering in the algorithm,
`recalcmean`:	['no' | {'yes'}], recalculate mean center after each cycle of replacement (may improve results for small matricies),
`display`:	[{'off'} | 'on'], governs level of display,
`tolerance`:	[{1e-6 100}] convergence criteria, the first element is the minimum change and the second is the maximum number of iterations,
`max_missing`:	[{0.4}] maximum fraction of missing data with which MDCHECK will operate, and
`toomuch`:	[{'error'} | 'exclude'] what action should be taken if too much missing data is found. 'error' exit with error message, 'exclude' will exclude elements (rows/columns/slabs/etc) which contain too much missing data from the data before replacement. 'exclude' requires a dataset object as input for (data),
`algorithm`:	[{'svd'} | 'nipals'] specified the missing data algorithm to use, NIPALS typically used for large amounts of missing data or large multi-way arrays.

Note: MDCHECK captures up to *options.frac_ssq* of the variance using *options.max_pcs* or fewer PCA components.

The default options can be retreived using: `options = mdcheck('options');`.

See Also

`parafac, pca`

med2top

Purpose

Fits a constant to top/(bottom) of data.

Synopsis

```
      [yf,residual,options] = med2top(y,options)
```

Description

MED2TOP is similar to LSQ2TOP with a 0 order polynomial, it can be considered an asymmetric estimate of the mean.

For fitting to the bottom:

```
>> tsq = residual/res; % (res) is an input
>> tsqst = ttestp(1-options.tsqlim,5000,2); % T-test limit from table
>> ii = find(tsq>-tsqst); % finds samples below the line
```

The ii samples are kept for the next estimate of (yf):

```
>> yf = median(y(ii));
```

INPUTS:

`y`	= trace to be filtered, Mx1 vector.

OUTPUTS:

`yf`	= scalar, estimate of filtered data.
`residual`	= y - yf.
`options`	= input `options` echoed back, the field initwt may have been modified.

Options

options	= a structure array with the following fields.
display:	[{'off'} | 'on'] Governs screen display to command line.
trbflag:	[{'top'} | 'bottom' | 'middle'] flag that tells algorithm to fit to the top, bottom, or middle of the data cloud.
tsqlim:	[0.99] limit that govers whether a data point is outside the fit residual defined by input (res).
stopcrit:	[1e-4 1e-4 1000 360] stopping criteria, iteration is continued until one of the stopping criterion is met [(rel tol) (abs tol) (max # iterations) (max time [seconds])].
initwt:	[ ] empty or Mx1 vector of initial weights (0<=w<=1).

See Also

`baseline`, `baslinew`, `fastnnls`, `lsq2top`

medcn

Purpose

Median center scales matrix to median zero.

Synopsis

```
[mcx,mx,msg] = medcn(x,options)
```

Description

`MEDCN` centers a matrix `x` to it's median and returns a matrix `mcx` with median zero columns and a vector of medians `mx` used to center the data. Optional input `options` is discussed below.

The output `msg` returns any warning messages.

Options

options	= a structure array with the following fields.
display:	[{'off'} | 'on'] Governs screen display.
matrix_threshold:	{.15} Error threshold based on fraction of missing data in whole matrix.
column_threshold:	{.25} Error threshold based on fraction of missing data in single column.

See Also

`auto, mncn, rescale, scale`

mlpca

Purpose

Maximum likelihood principal components analysis (user contributed).

Synopsis

```
[U,S,V,SOBJ,ErrFlag] = mlpca(x,stdx,p)
```

Description

`MLPCA` performs maximum likelihood principal components analysis assuming uncorrelated measurement errors. This is a method that attempts to provide an optimal estimation of the *p*-dimensional subspace containing the data by taking into account uncertainties in the measurements, thereby dealing with those cases that cannot be treated by simple scaling. Inputs are `x` (*m* by *n*) the data matrix to be decomposed, `stdx` (*m* by *n*) matrix of standard deviations corresponding to the observations in x, and the number of factors into which the data is decomposed `p`. The outputs are `U` (*m* by *p*) orthonormal, `S` (*p* by *p*) diagonal, and `V` (*n* by *p*) orthonormal. The ML scores are given by `U*S`. Additional output `SOBJ` is the value of the objective function for the best model. For exact uncertainty estimates, this should follow a chi-squared distribution with (m-p)*(n-p) degrees of freedom. Additional output `ErrFlag` indicates the termination conditions of the function;

`ErrFlag = 0`: normal termination (convergence), or
`ErrFlag = 1`: maximum number of iterations exceeded.

Also see:

P.D. Wentzell and M.T. Lohnes, “Maximum Likelihood Principal Component Analysis with Correlated Measurement Errors Theoretical and Practical Considerations”, Chemom. Intell. Lab. Syst., **45**, 65-85 (1999).

P.D. Wentzell, D.T. Andrews, D.C. Hamilton, K. Faber, and B.R. Kowalski, "Maximum likelihood principal component analysis", J. Chemometrics **11**(4), 339-366 (1997).

P.D. Wentzell, D.T. Andrews, and B.R. Kowalski, "Maximum likelihood multivariate calibration", Anal. Chem., **69**, 2299-2311 (1997).

D.T. Andrews and P.D. Wentzell, "Applications of maximum likelihood principal components analysis: Incomplete data and calibration transfer", Anal. Chim. Acta, **350**, 341-352 (1997).

See Also

`analysis, mcr, parafac, pca`

mlr

Purpose

Multiple Linear Regression for multivariate Y.

Synopsis

```
model = mlr(x,y,options)
pred  = mlr(x,model,options)
valid = mlr(x,y,model,options)
```

Description

MLR identifies models of the form Xb = y + e.

INPUTS:

`y`	= X-block: predictor block (2-way array or DataSet Object)
`y`	= Y-block: predictor block (2-way array or DataSet Object)

OUTPUTS:

`model`	= scalar, estimate of filtered data.
`pred`	= structure array with predictions
`valid`	= structure array with predictions

Options

options	= a structure array with the following fields.
display:	[{'off'} | 'on'] Governs screen display to command line.
plots:	['none' | {'final'}] governs level of plotting.
preprocessing:	{ [] [] } preprocessing structure (see PREPROCESS).
blockdetails:	['compact' | {'standard'} | 'all'] Extent of predictions and raw residuals included in model. 'standard' = only y-block, 'all' x and y blocks.

See Also

`analysis, crossval, modelstruct, pcr, pls, preprocess, ridge`

mlrengine

Purpose

Multiple Linear Regression computational engine.

Synopsis

```
reg = mlrengine(x,y,options)
```

Description

Inputs are an x-block `x`, y-block `y` and optional `options` structure.

Output is the matrix of regression vectors `reg`.

Options

`options`	= a structure array with the following fields.
`display:`	[{'off'} | 'on'] Governs screen display to command line.
`ridge:`	[0] ridge parameter to use in regularizing the inverse.

See Also

`analysis, pcr, pls`

mncn

Purpose

Mean center data matrices.

Synopsis

```
[mcx,mx] = mncn(x,options)
```

Description

`MNCN` mean centers a matrix `x` and returns a matrix `mcx` with mean zero columns and a vector of means `mx` used to center the data.

See Also

`auto`, `rescale`, `scale`

modelselector

Purpose

Create or apply a model selector model.

Synopsis

```
    model =
       modelselector(triggermodel,target_1,target_2,...,target_default);
    [target_model,applymodel] = modelselector(data,model)
```

Description

A Selector Model is a special model type which, when applied to new data, selects between two or more "target" models based on a "trigger" model. It is used to implement discrete local models when a single global model is not sufficient for all possible scenarios.

For example, if a single PCA or PLS model does not perform sufficiently for all operating conditions but the operating conditions can be split into two or more easier-to-model subsets, a selector model can be used to choose between these subset models when applying the models to new data.

Selector models consist of a trigger model (trigger) which can be either a PLSDA model or a set of one or more logical test strings and a set of two or more target models (target_1, target_2, etc) which can be any type of standard model structure or an empty array [ ] to indicate a null model.

Guidelines and rules for trigger models:

(A) A PLSDA trigger model can be created using the PLSDA function. Themodel should be built with data representative of the sample types to which each target model can be applied. The number of classes separated by the PLSDA model dictates the number of target models which can be selected from. The target models should be in the same order as the numerical class numbers used with PLSDA (e.g. if classes 1, 2 and 3 are used in PLSDA, the target models should be ordered so that target_1 is appropriate if the PLSDA model finds that a sample is class 1, target_2 is for class 2, and target_3 is for class 3.)

(B) Logical test strings are specified as a trigger model by passing a cell containing one or more strings which perform a logical test on a variable from the data set. Variables are specified using either a label in double quotes (e.g. "flowrate"), or a axisscale value in quotes and square brackets (e.g. "[1530]"). The varaible can be used in any interpretable Matlab expression (including function calls) that returns a logical result. The simplest test could involve one of the Matlab logical comparison operators (< > <= >= == and ~=) and a value to which the given variable should be compared. For example, the target model:

{'"Fe">1100' '"Fe"<500'}

tests if the variable named "Fe" is greater than 1100. If true, the target_1 model is applied, if not true, "Fe" is tested for being less than 500, and if so, target_2 is selected. If neither test is true, the "default" target model (i.e. target_3) is selected.

Example 2:

{'"[1745.3]"<=500'}

tests if variable 1745.3 (on the variable axiscale) is less than or equal to 500. If true, target_1 is selected, if not true, default target model is selected. If variable 1745.3 does not exist, it is interpolated from the provided data.

When creating a selector model, there must be at least as many target models passed as there are classes (when trigger is a PLSDA model) or strings (when trigger is a cell of logical test strings). There may also be an additional target model (i.e. the "default" model) which is used if none of the classes or tests were positive.

Note that target models may be any standard model structure including another selector model (thus allowing multi-layer selector trees).

To apply a selector model, a single row of new data is passed as a dataset along with the selector model itself. The output is the selected target model (target_model) along with a unique description of the "branch(s)" taken to select the target model as a vector of branch numbers (applymodel). For example, given a multi-layer selector model containing:

```
selector_model  -> target_1 = PCA_model_A1
                   target_2 = Selector_model -> target_1 = PCA_model_B1
                                               target_2 = PCA_model_B2
                  target_3 = PCA_model_A2
```

a returned value for applymodel of [2 1] implies that the second target model was selected from the first layer of target models, and this model was another selector model. From that second selector model, the first target model (PCA_model_B1) was selected and that is what was returned.

Note that if there are multiple "branches" (trigger models) the data passed to modelselector must contain all the data necessary for all trigger models within the selector model. If some of those variables are not used by a given model, modelselector will automatically discard unneeded variables before applying each trigger model.

See Also

`lwrpred, plsda, simca`

modelstruct

Purpose

Constructs an empty model structure.

Synopsis

```
model = modelstruct(modeltype,pred)
```

Description

The output of many of the PLS_Toolbox functions is a single model structure in which the results of the analysis are contained. A structure is an organized group of variables all stored as "fields" of a single containing variable. The purpose of `MODELSTRUCT` is to create the empty model structures used by the various modeling routines. The type of structure requested is passed as the single string input `modeltype` and should be one of: 'pca', 'pcr', (for PCA or PCR models) 'nip', 'sim' (PLS models), or 'parafac' (PARAFAC model).

Once the structures created by `MODELSTRUCT` are filled-in by the appropriate function (e.g. PLS, PCR, PCA), they contain all the results of the analysis and can be used as a single object for making further predictions or plots from the modeling results. In many cases, these models can be passed whole to another function. For example:

```
opts.plots = 'none';      % turn off plots for PCA (see PCA)
modl = pca(x, 3, opts);   % create a PCA model from data X
modlrder(modl);           % display relevent model information
plotscores(modl);         % plot scores from model
```

Although the individual fields (contents) of each model vary between modeltypes, most contain at least these fields:

`modeltype:`	name of model,
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`loads:`	cell array with model loadings for each mode/dimension,
`pred:`	cell array with model predictions for input data block (the first cell is empty if `options.blockdetail = 'normal'`),
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

Note that fields such as `loads`, `tsqs` and `ssqresiduals` are cell arrays of size `[modes, blocks]` where `modes` is the dimensionality of the data (e.g. for an array, modes = 2) and `blocks` is the number of blocks used by the analysis method (e.g. for PCA, blocks = 1, for PLS, blocks = 2). Thus, for a standard PCA model, loads will be a 2x1 cell containing "scores" in `modl.loads{1,1}` and traditional "loadings" in `modl.loads{2,1}`.

Because the models are standard MATLAB structures, they can be examined using standard structure notation:

```
>> modl.modeltype
ans =
PCA
>> modl.loads
ans =
    [30x4 double]
    [10x4 double]
```

Additionally, the individual components of a model can be "exploded" into individual variables using the `EXPLODE` function.

See Also

`analysis, explode, parafac, pca, pcr, pls`

modelviewer

Purpose

Visualization of multi-way models.

Synopsis

```
model = modelviewer(model,x);
```

Description

`MODELVIEWER` provides a graphical view of a model by enabling overview of scores, loadings, residuals etc. in one overall figure. Individual modes can be assessed by clicking plots and enlarged figures created by right-clicking plots.

INPUTS:

`model`	= PARAFAC, Tucker, or NPLS model, and
`x`	= X-block: predictor block (2-way array or DataSet Object).

OUTPUT:

`model`	= standard model structure (See `MODELSTRUCT`).

See Also

`plotgui, plotloads, plotscores`

modlpred

Purpose

Predictions based on models created by ANALYSIS.

Synopsis

```
[yprdn,resn,tsqn,scoresn] = modlpred(newx,modl,plots)
[yprdn,resn,scoresn] = modlpred(newx,bin,p,q,w,lv,plots);
```

Description

MODLPRED makes Y-block predictions based on an X-block and an existing regression model created using ANALYSIS.

Inputs are the new X-block data newx in the units of the original data, the structure variable that contains the regression model modl, and an optional variable *plots* which suppresses the plots when set to 0 {default = 1}.

Outputs are the Y-block predictions yprdn, residuals resn, T^2 values tsqn, and scores scoresn.

MODLPRED can also make predictions based on an existing PLS model constructed with the NIPALS algorithm from the PLS function. Inputs are the matrix of predictor variables newx, the PLS model inner-relation coefficients bin, the x-block loadings p, the y-block loadings q, the x-block weights w, the number of latent variables to use in prediction lv, and an optional variable *plots* which suppresses the plots when set to 0 {default = 1}.

Outputs are the Y-block predictions yprdn, residuals resn, and the scores scoresn. Note that T^2 are not calculated.

See Also

analysis, explode, modlrder, pca, pcapro, pcr, pls

modlrder

Purpose

Prints model information for standard model structures.

Synopsis

```
    modlrder(modl)
```

Description

`MODLRDER` reads information contained in a standard model structure variable `modl` and prints the information to the command window. It can be used with models created by the following functions: `ANALYSIS, NPLS, PARAFAC, PCA, PCR, PLS, ANALYSIS`.

Information includes date and time created and methods used to construct the model. There is no assignable output.

See Also

`analysis, explode, modlpred, pcapro, ssqtable`

mpca

Purpose

Multi-way (unfold) principal components analysis.

Synopsis

```
model = mpca(mwa,ncomp,options)
model = mpca(mwa,ncomp,preprostring)
pred = mpca(mwa,model,options)
options = mpca('options')
```

Description

Principal Components Analysis of multi-way data using unfolding to a 2-way matrix followed by conventional PCA.

Inputs to MPCA are the multi-way array mwa (class “double” or “dataset”) and the number of components to use in the model nocomp. To make predictions with new data the inputs are the multi-way array mwa and the MPCA model model. Optional input *options* is discussed below.

The output model is a structure array with the following fields:

modeltype:	'MPCA',
datasource:	structure array with information about the x-block,
date:	date of creation,
time:	time of creation,
info:	additional model information,
loads:	1 by 2 cell array with model loadings for each mode/dimension,
pred:	cell array with model predictions for each input data block (this is empty if options.blockdetail = 'normal'),
tsqs:	cell array with T^2 values for each mode,
ssqresiduals:	cell array with sum of squares residuals for each mode,
description:	cell array with text description of model, and
detail:	sub-structure with additional model details and results.

Options

options =	a structure array with the following fields.
display:	['off' | {'on'}] governs level of display to command window,
plots:	['none' | {'final'}] governs level of plotting,
outputversion:	[2 | {3}] governs output format,

`preprocessing`:	`{ [] }` preprocessing structure, {default is mean centering i.e. `options.preprocessing = preprocess('default', 'mean center')`} (see `PREPROCESS`),
`blockdetails`:	`[ 'compact' | {'standard'} | 'all' ]` extent of detail in predictions and residuals included in model structure ('`standard`' results in sum of squared residuals, and '`all`' gives all x-block residuals), and
`samplemode`:	[{3}] mode (dimension) to use as the sample mode e.g. if it is 3 then it is assumed that mode 3 is the sample/object dimension i.e. if `mwa` is 7x9x10 then the scores `model.loads{1}` will have 10 rows (it will be 10x`ncomp`).

The default options can be retreived using: `options = mpca('options');`.

It is also possible to input just the preprocessing option as an ordinary string in place of *options* and have the remainder of options filled in with the defaults from above. The following strings are valid:

'`none`':	no scaling,
'`auto`':	unfolds array then applies autoscaling,
'`mncn`':	unfolds array then applies mean centering, or
'`grps`':	{default} unfolds array then group/block scales each variable, i.e. the same variance scaling is used for each variable along its time trajectory (see `GSCALE`).

`MPCA` will work with arrays of order 3 and higher. For higher order arrays, the last order is assumed to be the sample order, *i.e.* for an array of order *n* with the dimension of order *n* being *m*, the unfolded matrix will have *m* samples. For arrays of higher order the group scaling option will group together all data with the same order 2 index, for multiway array `mwa`, each `mwa(:,j,:, ... ,:)` will be scaled as a group.

See Also

`analysis, evolvfa, ewfa, explode, parafac, pca, preprocess`

mplot

Purpose

Automatic creation of subplots and plotting.

Synopsis

```
[rows,cols] = mplot(n,options)
[rows,cols] = mplot([rows cols],options)
[rows,cols] = mplot(rows,cols,options)
[rows,cols] = mplot(y,options)
[rows,cols] = mplot(x,y,options)
```

Description

Inputs can be one of four forms:

(1) the number of subplots requested `n`, “best fit” onto the figure

(2) the number of rows and columns for the subplot array `[rows cols]`

(3) or data to plot `y` with or without reference data for the x-axis `x`. Each column of `y` is plotted in a single subplot on the figure.

Outputs are the number of rows `rows` and columns `cols` used for the subplots.

Examples

Example 1. To automatically create a “best fit” of four empty subplots

```
mplot(4)
```

Example 2. To automatically create four subplots in a 4 x 1 arrangement

```
mplot([4 1])
```

Example 3. To automatically plot three random columns, each in its own subplot

```
mplot(rand(100,3))
```

Options

`center`:	[{'no'} | 'yes'] governs centering of "left-over" plots at bottom of figure (when an uneven number of plots are to be fit onto the screen,
`axismode` :	[{''} | 'tight'] governs axis settings

Algorithm

When mplot is doing the "best fit", it attempts to keep the number of rows and columns as close as possible in size (Except for `n`=3 which is done as a 3x1 figure). Thus, the plot progression is: 1x1, 2x1, 3x1, 2x2, 3x2, 3x3, 4x3, etc.

See Also

`plotgui, subplot`

ms_bin

Purpose

Bins Mass Spectral data into user-defined bins.

Synopsis

```
dso = ms_bin(data)
dso = ms_bin(data, options)
```

Description

Often raw Mass Spec data is output in its original profile format (e.g., 14.5, 14.5, 14.6,...) and one requires "unit" mass resolution (e.g., 14, 15, 16,...) in order to reduce the size of the data and or analyze the data properly. In its default form the MS_BIN function will bin at unit resolution and return the data in a DataSet Object. Using the two optional parameters (resolution and round_off_point) the function can be adjusted to meet different requirements.

INPUTS:

data :	a cell array with the data. Each cell will correspond to a row in the resulting dataset 'dso' and should contain nx2 numeric array of "xy" MS data: the first column contains the mass numbers, the second column contains the counts (intensities). The number of rows in the cells can be different.

OUTPUTS:

dso :	dataset object

Options

resolution :	optional, defines the resolution. The default value is 1.
round_off_point :	optional. Normally the round-off point is in the middle of the bin. For unit resolution it would be 0.5: everything below 0.5 will be rounded down, everything higher than 0.5 will be rounded up. In case the peak is asymmetrical other points are used, e.g. 0.65. The round off for the array m with the mass numbers is then: round(m+0.5-round_off_point); The asymmetric round-off is also valid for resolution lower than 1: the round_off_point is the relative position in the bin.

See Also

`frpcr, stdfir, stdgen`

mscorr

Purpose

Multiplicative scatter/signal correction (MSC).

Synopsis

```
[sx,alpha,beta,xref] = mscorr(x,xref,mc,win,specmode,subind)
```

Description

`MSCORR` performs multiplicative scatter correction (a.k.a. multiplicative signal correction) on an input matrix of spectra `x` (class "double") regressed against a reference spectra `xref` (class "double"). If (xref) is empty or omitted, the mean of (x) is used as the reference.

If the optional input *mc* is 1 {default} then an intercept is used. If *mc* is set to 0 (zero) then a force fit through zero is used.

Optional input *win* is a *NK* element cell array of indices corresponding to windows to perform MSC, i.e. MSC is performed in each window `win{i}` for *i=1:NK*. In this case, (`alpha` and `beta` are not assigned). Optional input (specmode) defines which mode of the data is the spectral mode (default = 2) and is only used when (x) contains 3 or more modes. Optional input (subind) specifies the indices within the included spectral variables that are used to calculate the MSC correction factors (alpha and beta); default is that ALL included spectral variables are used.

Outputs are the corrected spectra `sx`, the intercepts/offsets `alpha`, the multiplicative scatter factor/slope `beta`, and the reference spectrum `xref`.

Algorithm

For input spectra $\mathbf{x}$ (1xN) and reference spectra $\mathbf{x}_{\text{ref}}$ (1xN) the model is:

$$\mathbf{x}^{\text{T}}\beta + \alpha = \mathbf{x}^{\text{T}}_{\text{ref}} .$$

and the corrected spectra $\mathbf{x}_{\text{s}}$ (1xN) is given by:

$$\mathbf{x}_{\text{s}} = (\mathbf{x}_{\text{ref}} - \alpha)/\beta .$$

See Also

`frpcr, stdfir, stdgen`

mtfreadr

Purpose

Read / Import AdventaCT Multi-Trace Format (MTF) files.

Synopsis

```
data = mtfreadr(filename,combine)
[data,lotinfo] = mtfreadr(filename,combine)
```

Description

Generic reader for AdventaCT Multi-Trace Format (MTF) files. Input is an optional filename `filename` If omitted, user is prompted to locate file. An optional input `combine` is a string instructing how to combine multiple traces found in the mtf file:

'none' :	returns a cell array containing datasets formed from each of the separate traces located in the MTF file.
'truncate' :	{default} truncates all traces to the shortest trace's length.
'pad' :	pads all traces with NaN's to the longest trace's length.
'stretch' :	uses linear interpolation to stretch all traces to the longest trace's length.

The output `data` is either a DSO (3-way DSO if multiple traces were found) or a cell array containing all the trace DSOs. Note that if a given trace does not have a sufficient number of columns in all rows, column contents may be scrambled from the dropped point down. In this situation, a warning will be given.

See Also

`areadr, spcreadr, xclgetdata, xclputdata, xclreadr`

ncrossval

Purpose

Cross-validation for multilinear PLS (`NPLS`).

Synopsis

```
    [press,cumpress,rmsecv,rmsec,cvpred,misclassed] =
       ncrossval(x,y,rm,cvi,ncomp,out,pre)
```

Description

Performs cross-validation of NPLS. If two-way unfold-PLS is desired convert input x to two-way x. By default, the data are centered across the first mode, but no scaling is applied. This can be changed by using additional input arguments.

INPUTS:

`x` =	X-block matrix,
`y` =	Y-block matrix, and
`rm` =	regression method (must be 'npl')
`cvi` =	see CROSSVAL
`ncomp` =	maximum number of factors.
`out` =	see CROSSVAL
`pre` =	see CROSSVAL

OUTPUT:
See CROSSVAL

See Also

`crossval, npls`

nippls

Purpose

NIPALS Partial Least Squares computational engine.

Synopsis

```
[reg,ssq,xlds,ylds,wts,xscrs,yscrs,bin] = nippls(x,y,ncomp,options)
options = nippls('options')
```

Description

Performs PLS regression using NIPALS algorithm.

INPUTS:

`x` =	X-block (*M* by *Nx*) and
`y` =	Y-block (*M* by *Ny*).

OPTIONAL INPUTS:

`nocomp` =	number of components {default = rank of X-block}, and
`options` =	discussed below.

The default options can be retreived using: `options = nippls('options');`.

OUTPUTS:

`reg` =	matrix of regression vectors,
`ssq` =	the sum of squares captured (ssq),
`xlds` =	X-block loadings,
`ylds` =	Y-block loadings,
`wts` =	X-block weights,
`xscrs` =	X-block scores,
`yscrs` =	Y-block scores, and
`bin` =	the inner relation coefficients.

Note: The regression matrices are ordered in `reg` such that each *Ny* (number of y variables) rows correspond to the regression matrix for that particular number of latent variables.

Options

options =	a structure containing the fields:
`display`:	['off' |{'on'}], governs display to command window.

See Also

`pls, analysis, simpls`

normaliz

Purpose

Normalizes rows of matrix to unit vectors.

Synopsis

```
[ndat,norms] = normaliz(dat)
[ndat,norms] = normaliz(dat,out,normtype)
```

Description

NORMALIZ can be used for pattern normalization, which is useful for preprocessing in some pattern recognition applications and also for correction of pathlength effects for some quantification applications.

The input is the data matrix dat. Optional input *out* suppresses warnings when set to 0 (zero) {default = 1} (warnings are given if the norm of a vector is zero). Optional input *normtype* can be used to specify the type of norm {default = 2}. If *normtype* is specified then *out* must be included, *out* can be empty [].

The output is the matrix of normalized data ndat where the *rows* have been normalized, and the vector of norms used in the normalization norms. Warnings are given for any vectors with zero norm.

Algorithm

For a 1 by N vector **x**, the norm n_x is given by $n_x = \left(\sum_{j=1}^{N} \left| x_i^p \right| \right)^{1/p}$ where p is *normtype*. The normalized 1 by N vector $\mathbf{x}_n$ is given by $\mathbf{x}/n_x$.

See Also

auto, baseline, mncn, mscorr, snv

npls

Purpose

Multilinear-PLS (N-PLS) for true multi-way regression.

Synopsis

```
model = npls(x,y,ncomp,options)
pred  = npls(x,ncomp,model,options)
options = npls('options')
```

Description

`NPLS` fits a multilinear PLS1 or PLS2 regression model to `x` and `y` [R. Bro, J. Chemom., 1996, 10(1), 47-62]. The NPLS function also can be used for calibration and prediction.

INPUTS:

`x` =	X-block,
`y` =	Y-block, and
`ncomp` =	the number of factors to compute, or
`model` =	in prediction mode, this is a structure containing a NPLS model.

OPTIONAL INPUTS:

`options` =	`discussed below.`

OUTPUT:

`model` =	standard model structure (see: `MODELSTRUCT`) with the following fields:
`modeltype:`	'NPLS',
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
reg:	cell array with regression coefficients,
`loads:`	cell array with model loadings for each mode/dimension,
core:	cell array with the NPLS core,
`pred:`	cell array with model predictions for each input data block,
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

Options

options = options structure containing the fields:

`display`: `[ 'off' | {'on'} ]`, governs level of display to command window,

`plots`: `[ 'none' | {'final'} ]`, governs level of plotting,

`outputregrescoef`: if this is set to 0 no regressions coefficients associated with the X-block directly are calculated (relevant for large arrays), and

`blockdetails`: `[ {'standard'} | 'all' ]`, level of detail included in the model for predictions and residuals.

See Also

`datahat, explode, gram, mpca, outerm, parafac, pls, tld, unfoldm`

npreprocess

Purpose

Preprocessing of multi-way arrays.

Synopsis

```
[prex,prepar] = npreprocess(x,prepar,undo,options)
prex = npreprocess(x,setting)
prex = npreprocess(x,prepar)
prex = npreprocess(x,prepar,1)
options = npreprocess('options')
```

Description

NPREPROCESS is used for three different purposes:

1) for centering and scaling multi-way arrays in which case the parameters (offsets and scales) are first calculated and then applied to the data,

2) for preprocessing another data set according to (1), and

3) for transforming preprocessed data back (undo preprocessing).

INPUTS:

`x` =	data array, and
`settings` =	a two-row matrix (class "double") indicating which modes to center and scale. The matrix is: `settings = [cent; scal]`. E.g. `settings(1,:) = [1 0 1]` => center across mode one and three, and `settings(2,:) = [1 1 0]` => scale to unit variance within mode one and two.

OPTIONAL INPUTS:

prepar =	contains earlier defined mean and scale parameters, this data is required for applying or undoing preprocessing,
undo =	when set to 1 this flag tells to undo/transform back, and
options =	`discussed below.`

OUTPUTS:

`prex` =	the preprocessed data, and
`prepar` =	a structure containing the necessary parameters to pre- and post-process other arrays.

Options

options =	a structure array with the following fields:
`display`:	`[ {'on'} | 'off' ]`, governs level of display,
`iterproc`:	`[ 'on' | {'off'} ]`, allows iterative preprocessing which is necessary for some combinations of centering and scaling (see User Manual),
`scalefirst`:	`[ {'on'} | 'off' ]`, defines that scaling is done before centering which may have implications in complex combinations of preprocessing (see User Manual), and
`usemse`:	`[ {'on'} | 'off' ]`, defines that mean square scaling is used instead of scaling by standard deviations as is common in two-way analysis.

Examples

To apply preprocessing with options:

```
    [prex,prepar] = npreprocess(x,settings,[],0,options);
```

See Also

`auto, mncn, preprocess, rescale, scale`

oscapp

Purpose

Applies orthogonal signal correction model to new data.

Synopsis

```
    newx = oscapp(x,nw,np,nofact)
```

Description

Inputs are the new data matrix `x`, weights from the OSC model `nw`, and loadings from the OSC `np`.

Optional input *nofact* can be used to restrict the correction to a smaller of factors than originally calculated.

The output is is the corrected data matrix `newx`.

Note: input data `x` must be centered and scaled like the original data!

See Also

`crossval, osccalc`

osccalc

Purpose

Calculates orthogonal signal correction.

Synopsis

```
[nx,nw,np,nt] = osccalc(x,y,nocomp,iter,tol)
```

Description

Inputs are the matrix of scaled predictor variables `x`, scaled predicted variable(s) `y`, and the number of OSC components `nocomp`.

Optional inputs are the maximum number of iterations used in attempting to maximize the variance captured by othogonal components `iter` {default = 0}, and the tolerance on percent of `x` variance to consider when forming the final w vector `tol` {default = 99.9}.

Outputs are the OSC corrected predictor matrix `nx`, and the x-block weigths `nw`, loads `np`, and scores `nt` that were used in making the correction.

Once the calibration is done, new (scaled) X data can be corrected by `newx = x - x*nw*inv(np'*nw)*np';`. See `OSCAPP`.

See Also

`crossval, oscapp`

outerm

Purpose

Computes the outer product of any number of vectors with multiple factors.

Synopsis

```
mwa = outerm(facts,lo,vect)
```

Description

The input to outer is a 1 by *N* cell array `facts`, where each cell contains a matrix of factors for one of the modes (a.k.a. ways, dimensions, or orders), with each factor being a column in the matrix.

Optional inputs are `lo` the number of a mode to leave out in the formation of the outer product, and a flag `vect` which causes the function to not sum and reshape the final factors when set to 1. (This option is used in the alternating least squares steps in PARAFAC.)

The output is the multiway array resulting from multiplying the factors together `mwa`, or the strung out individual factors.

Examples:

```
a = [[1:7]' [2 4 1 3 5 7 6]'];          %  7x2
b = [sin([1:.5:5]') cos([1:.5:5]')];   %  9x2
c = [[1:8 0 0]', [0 0 1:8]'];          % 10x2
x = outerm({a,b,c});                   % 7x9x10
```

See Also

`gram, mpca, parafac, tld`

parafac

Purpose

PARAFAC (PARAllel FACtor analysis) for multi-way arrays

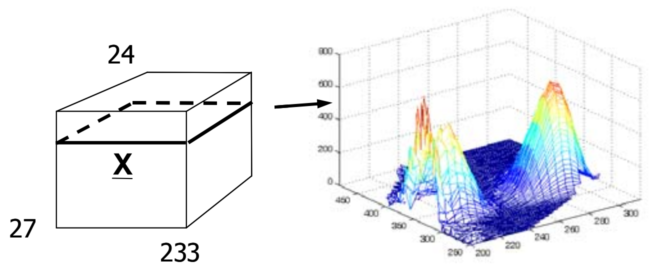
Synopsis

```
model   = parafac(X,initval,options)
pred    = parafac(Xnew,model)
options = parafac('options')
```

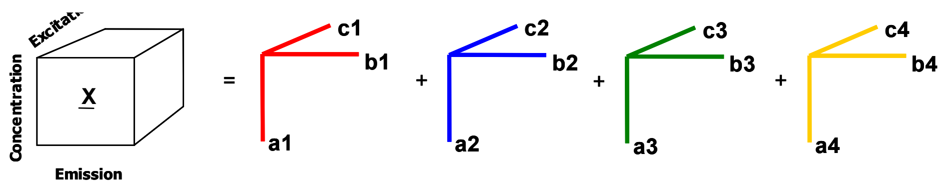
Description

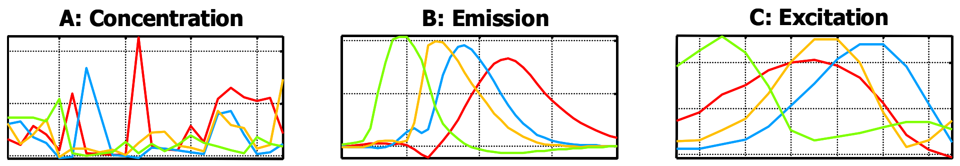
PARAFAC will decompose an array of order *N* (where *N* ≥ 3) into the summation over the outer product of *N* vectors (a low-rank model). E.g. if *N*=3 then the array is size *I* by *J* by *K*. An example of three-way fluorescence data is shown below..

For example, twenty-seven samples containing different amounts of dissolved hydroquinone, tryptophan, phenylalanine, and dopa are measured spectrofluoremetrically using 233 emission wavelengths (250-482 nm) and 24 excitation wavelengths (200-315 nm each 5 nm). A typical sample is also shown.



A four-component PARAFAC model of these data will give four factors, each corresponding to one of the chemical analytes. This is illustrated graphically below. The first mode scores (loadings in mode 1) in the matrix **A** (27×4) contain estimated relative concentrations of the four analytes in the 27 samples. The second mode loadings **B** (233×4) are estimated emission loadings and the third mode loadings **C** (24×4) are estimated excitation loadings.





In the PARAFAC algorithm, any missing values must be set to NaN or Inf and are then automatically handled by expectation maximization. This routine employs an alternating least squares (ALS) algorithm in combination with a line search. For 3-way data, the initial estimate of the loadings is usually obtained from the tri-linear decomposition (TLD).

INPUTS:

`x` =	the multiway array to be decomposed, and
`ncomp` =	the number of factors (components) to use, or
`model` =	a PARAFAC model structure (new data are fit to the model i.e. sample mode scores are calculated).

OPTIONAL INPUTS:

`initval` =	cell array of initial values (initial guess) for the loadings (e.g. `model.loads` from a previous fit). If not used it can be 0 or [], and
`options` =	discussed below.

OUTPUTS:

The output `model` is a structure array with the following fields:

`modeltype:`	'PARAFAC',
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`loads:`	1 by *K* cell array with model loadings for each mode/dimension,
`pred:`	cell array with model predictions for each input data block,
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

The output `pred` is a structure array that contains the approximation of the data if the options field blockdetails is set to `'all'` (see next).

Options

options =	a structure array with the following fields:
`display:`	`[ {'on'} | 'off' ]`, governs level of display,
`plots:`	`[ {'final'} | 'all' | 'none' ]`, governs level of plotting,
`weights:`	`[]`, used for fitting a weighted loss function (discussed below),
`stopcrit:`	`[1e-6 1e-6 10000 3600]` defines the stopping criteria as [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time in seconds)],
`init:`	`[ 0 ]`, defines how parameters are initialized (discussed below),
`line:`	`[ 0 | {1}]` defines whether to use the line search {default uses it},
`algo:`	`[ {'ALS'} | 'tld' | 'swatld' ]` governs algorithm used,
`iterative:`	settings for iterative reweighted least squares fitting (see help on weights below),
`blockdetails:`	'standard'
`missdat:`	this option is not yet active,
`samplemode:`	`[1]`, defines which mode should be considered the sample or object mode,
`constraints:`	{3x1 cell}, defines constraints on parameters (discussed below), and
`coreconsist:`	`[ {'on'} | 'off' ]`, governs calculation of core consistency (turning off may save time with large data sets and many components).

The default options can be retrieved using: `options = parafac('options');`.

WEIGHTS

Through the use of the *options* field `weights` it is possible to fit a PARAFAC model in a weighted least squares sense The input is an array of the same size as the input data `X` holding individual weights for each element. The PARAFAC model is then fit in a weighted least squares sense. Instead of minimizing the frobenius norm $||x\text{-}M||^2$ where M is the PARAFAC model, the norm $||(x\text{-}M).*weights||^2$ is minimized. The algorithm used for weighted regression is based on a majorization step according to Kiers, *Psychometrika*, **62**, 251-266, 1997 which has the advantage of being computationally inexpensive. If alternatively, the field `weights` is set to `‘iterative’` then iteratively reweighted least squares fitting is used. The settings of this can be modified in the field `iterative.cutoff_residuals` which defines the cutoff for large residuals in terms of the number of robust standard deviations. The lower the number, the more subtle outliers will be ignored.

INIT

The *options* field `init` is used to govern how the initial guess for the loadings is obtained. If optional input *initval* is input then `options.init` is not used. The following choices for init are available.

Generally, `options.init = 0`, will do for well-behaved data whereas `options.init = 10`, will be suitable for difficult models. Difficult models are typically those with many components, with very correlated loadings, or models where there are indications that local minima are present.

`init = 0`,	PARAFAC chooses initialization {default},
`init = 1`,	uses TLD (unless there are missing values then random is used),
`init = 2`,	initializes loadings with random values,
`init = 3`,	based on orthogonalization of random values (preferred over 2),
`init = 4`,	based on singular value decomposition,
`init = 5`,	based on compression which may be useful for large data, and
`init > 5`,	based on best fit of many (the value `options.init`) small runs.

CONSTRAINTS

The *options* field `constraints` is used to employ constraints on the parameters. It is a cell array with number of elements equal to the number of modes of the input data X. Each cell contains a structure array with the following fields:

`nonnegativity:`	`[ {0} | 1 ]`, a 1 imposes non-negativity.
`unimodality:`	`[ {0} | 1 ]`, a 1 imposes unimodality (1 local maxima).
`orthogonal:`	`[ {0} | 1 ]`, constrain factors in this mode to be orthogonal.
`orthonormal:`	`[ {0} | 1 ]`, constrain factors in this mode to be orthonormal.
`exponential:`	`[ {0} | 1 ]`, a 1 fits an exponential function to the factors in this mode.

`smoothness.weight:[0 to 1], imposes smoothness using B-splines, values near 1 impose high smoothness and values close to 0, impose less smoothness.`

`fixed.position:`	`[ ]`, a matrix containing 1's and 0's of the same size as the corresponding loading matrix, with a 1 indicating where parameters are fixed.
`fixed.value:`	`[ ]`, a vector containing the fixed values. Thus, if `B` is the loading matrix, then we seek `B(find(fixed.position)) = fixed.value`. Therefore, fixed.value must be a matrix of the same size as the loadings matrix and with the corresponding elements to be fixed at their appropriate values. All other elements of fixed.value are disregarded.
`fixed.weight:`	[ ], a scalar (0 ≤ `fixed.weight` ≤ 1) indicating how strongly the `fixed.value` is imposed. A value of 0 (zero) does not impose the constraint at all, whereas a value of 1 (one) fixes the constraint.
`ridge.weight:`	`[ ]`, a scalar value between 0 and 1 that introduces a ridging in the update of the loading matrix. It is a penalty on the size of the esimated loadings. The closer to 1, the higher the ridge. Ridging is useful when a problem is difficult to fit.
`equality.G:`	[ ], matrix with *N* columns, where *N* is the number of factors, used with `equality.H`. If **A** is the loadings for this mode then the constraint is

	imposed such that $\mathbf{GA}^T = \mathbf{H}$. For example, if **G** is a row vector of ones and **H** is a vector of ones (1's), this would impose closure.
`equality.H`:	[ ], matrix of size consistent with the constriant imposed by `equality.G`.
`equality.weight`:	[ ], a scalar ($0 \leq$ equality.weight ≤ 1) indicating how strongly the `equality.H and equality.G` is imposed. A value of 0 (zero) does not impose the constraint at all, whereas a value of 1 (one) fixes the constraint.
`leftprod`:	`[0]`, If the loading matrix, **B** is of size JxR, the leftprod is a matrx **G** of size JxM. The loading **B** is then constrained to be of the form **B = GH**, where only **H** is updated. For example, **G** may be a certain JxJ subspace, if the loadings are to be within a certain subspace.
`rightprod`:	`[0]`, If the loading matrix, **B** is of size JxR, the rightprod is a matrx **G** of size MxR. The loading **B** is then constrained to be of the form **B** = **HG**, where only **H** is updated. For example, if rightprod is `[1 1 0;0 0 1]`, then the first two components in **B** are forced to be the same.
`iterate_to_conv`:	`[0]`, Usually the constraints are imposed within an iterative algorithm. Some of the constraints use iterative algorithms themselves. Setting `iterate_to_conv` to one, will force the iterative constraint algorithms to continue until convergence.
`timeaxis`:	`[]`, This field (if supplied) is used as the time axis when fitting loadings to a function (e.g. see `exponential`). Therefore, it must have the same number of elements as one of the loading vectors for this mode.
`description`:	`[1x1592 char]`,

If the constraint in a mode is set as fixed, then the loadings of that mode will not be updated, hence the initial loadings stay fixed.

Examples

`parafac demo` gives a demonstration of the use of the PARAFAC algorithm.

`model = parafac(X,5)` fits a five-component PARAFAC model to the array `X` using default settings.

`pred = parafac(Z,model)` fits a parafac model to new data `Z`. The scores will be taken to be in the first mode, but you can change this by setting `options.samplemodex` to the mode which is the sample mode. Note, that the sample-mode dimension may be different for the old model and the new data, but all other dimensions must be the same.

`options = parafac('options');` generates a set of default settings for PARAFAC. `options.plots = 0;` sets the plotting off.

`options.init = 3;` sets the initialization of PARAFAC to orthogonalized random numbers.

`options.samplemodex = 2;` Defines the second mode to be the sample-mode. Useful, for example, when fitting an existing model to new data has to provide the scores in the second mode.

`model = parafac(X,2,options);` fits a two-component PARAFAC model with the settings defined in options.

`parafac io` shows the I/O of the algorithm.

See Also

`datahat, explode, gram, mpca, outerm, parafac2, tld, tucker, unfoldm`

parafac2

Purpose

PARAFAC2 (PARAllel FACtor analysis2) for multi-way arrays

Synopsis

```
    model   = parafac2(X,ncomp);       %decomposition
    model   = parafac2(X,ncomp,options);
    model   = parafac2(X,initval);
    pred    = parafac2(Xnew,model);    %application
    options = parafac2('options');
```

Description

The three-way PARAFAC2 model is best perceived as a model close to the ordinary PARAFAC model. The major difference is that strict trilinearity is no longer required, so PARAFAC2 can sometimes handle elution time shifts, varying batch trajectories etc. The ordinary PARAFAC model is also sometimes called the PARAFAC1 model to distinguish it from the PARAFAC2 model.

In the PARAFAC1 model, one loading matrix is found for each mode. That implies that this loading matrix is the same across all levels for the other modes. For example, in a PARAFAC1 model of a data set with chromatographic spectrally detected experiments, the PARAFAC1 model ideally provides a loading matrix for e.g. the chromatographic mode which holds the true elution profiles of the chemical analytes. Thus, the PARAFAC1 model assumes that these elution profiles do not change shape in different experiments (only their magnitude). Such an assumption may be too strict and invalid. A little model error is seldom problematic, but if the structure of the data deviates considerably from the assumptions of the model, it can be impossible to fit a reasonable model. In the PARAFAC2 model, this trilinearity assumption is relaxed in one mode. A PARAFAC1 model of a three-way array is given by **A**, **B** and **C** (loading matrices in first, second and third mode). In PARAFAC2, the loadings in one mode can change from level to level. That is, assume that the third mode (**C**) of dimension *K* holds different samples (it is common practice, to have samples in the last mode for PARAFAC2). Instead of having a fixed first mode loading **A** for all samples, **A** may now vary from sample to sample. Thus for each sample, *k*, there is an individual **A** called $\mathbf{A}_k$. The only restriction on $\mathbf{A}_k$ is that the cross-product $\mathbf{A}_k^T\mathbf{A}_k$ remains constant. This is in contrast to PARAFAC1 where **A** is simply the same for all *k*.

Another way of imposing this constraint ($\mathbf{A}_k^T\mathbf{A}_k$ constant) is to say that each $\mathbf{A}_k$ is modeled as $\mathbf{P}_k\mathbf{H}$ where $\mathbf{P}_k$ is an orthogonal matrix of the same size as $\mathbf{A}_k$ and where **H** is a small quadratic matrix with dimension equal to the number of components. This different interpretation of the concept shows that the individual components $\mathbf{A}_k$ only differ up to a rotation. Hence, the latent variables are the same for all samples but may manifest themselves through different rotations.

The situations in which the PARAFAC2 model is valid can be difficult to understand because the flexibility compared to the PARAFAC1 model is somewhat abstract. However, one simple way to see the applicability of the PARAFAC2 model is that PARAFAC2 is worth considering in situations in which PARAFAC1 should ideally be valid, but where practical applications show that it is not. For example, it is often observed that the differences in elution profiles from experiment to experiment in chromatography makes the PARAFAC1 model difficult to fit. Many times PARAFAC2 can still handle such deviations even when the shifts in retention times are quite severe.

It is possible to fit both the PARAFAC1 and the PARAFAC2 model. If both models give the same results (approximately), then PARAFAC1 is likely valid and then PARAFAC1 is preferred because it uses fewer degrees of freedom. If there are large deviations, PARAFAC2 may be preferred. Note, though, that the K matrices $\mathbf{A}_k$ may have a larger variability than the corresponding $\mathbf{A}$ from the PARAFAC1 model because of the smaller amount of data that it is estimated from. This does not imply inadequacy but simply that there are differences in the way that the parameters are estimated.

Another interesting type of application of PARAFAC2 follows from the insight that the constraint that $\mathbf{A}_k^T\mathbf{A}_k$ is constant. This directly implies that the individual slabs, $\mathbf{X}_k$, of the array can have different lengths, hence different size $\mathbf{A}_k$, yet still fulfill the constraint that $\mathbf{A}_k^T\mathbf{A}_k$ is constant. Thus, PARAFAC2 can also handle e.g. batch data where the data from each batch are obtained at different sampling rates or different sampling duration. This is a very powerful feature of the PARAFAC2 model compared to the PARAFAC1 model.

The three-way PARAFAC2 model is given

$$\mathbf{X}_k = \mathbf{A}_k\mathbf{D}_k\mathbf{B}^T + \mathbf{E}_k = \mathbf{P}_k\mathbf{H}\mathbf{D}_k\mathbf{B}^T + \mathbf{E}_k,\ k = 1, .., K$$

$\mathbf{X}_k$ is a slab of data (I×J) in which I may actually vary with K. K is the number of slabs and $\mathbf{A}_k$ (I×ncomp) are the first-mode loadings for the kth sample. $\mathbf{D}_k$ is a diagonal matrix that holds the kth row of $\mathbf{C}$ in its diagonal. $\mathbf{C}$ (K×comp) is the third mode loadings, $\mathbf{H}$ is an (ncomp×ncomp) matrix, and $\mathbf{P}_k$ is an (I×ncomp) orthogonal matrix. The output $\mathbf{P}$ is given as a cell array of length K where the kth cell element holds the (I×ncomp) matrix $\mathbf{P}_k$. Thus, to get e.g. the second sample $\mathbf{P}$, write **P**{2}, and to get the estimate of the first mode loadings, $\mathbf{A}_k$, at this second frontal slab (k = 2), write **P**{2}***H**.

The model can also be fitted to more than three-way data. It is important then to be aware which mode is supposed to be fitted by separate loadings for each sample. The convention is that the first mode is the mode that has individual loadings and that these are defined across the last (the sample) mode. For example, chromatographic data with spectral detection can be arranged as the first mode being elution, the second spectral and the third mode being different experiments. Then different elution profiles (mode one) are found for each experiment (mode three). For multivariate batch process data, the array is typically arranged as time × variables × batches, meaning that the time trajectories (mode one) can vary from batch to batch (mode three).

INPUTS:

`x` =	the multiway array to be decomposed, If all slabs have similar size, x is an array. For example, for three-way data where the matrix of measurements for sample one is held in x1, for sample 2 in x2 etc. then X(:,:,1) = X1; X(:,:,2) = X2; etc. If the slabs have different size, X is a cell array (type <help cell> for more info on cells). Then X{1} = X1; X{2} = X2; etc., and
`ncomp` =	the number of factors (components) to use, or
`model` =	a PARAFAC model structure (new data are fit to the model i.e. sample mode scores are calculated).

OPTIONAL INPUTS:

`initval` =	cell array of initial values (initial guess) for the loadings (e.g. `model.loads` from a previous fit). If not used it can be 0 or [], and
`options` =	discussed below.

OUTPUTS:

Data that are input as a cell-array in PARAFAC2 are converted to an array by zero-padding each samples first mode dimension in case of different first mode dimensions for different samples. Residuals etc. are also output as arrays. The output `model` is a structure array with the following fields:

`modeltype:`	'PARAFAC2',
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`loads:`	1 by *K* cell array with model loadings for each mode/dimension,
`pred:`	cell array with model predictions for each input data block,
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

The output `pred` is a structure array that contains the approximation of the data if the options field blockdetails is set to `'all'` (see options).

Options

options =	a structure array with the following fields:
`display:`	`[ {'on'} | 'off' ]`, governs level of display,
`plots:`	`[ {'final'} | 'all' | 'none' ]`, governs level of plotting,
`weights:`	`[]`, used for fitting a weighted loss function,
`stopcrit:`	`[1e-6 1e-6 10000 3600]` defines the stopping criteria as [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time in seconds)],
`init:`	`[ 0 ]`, defines how parameters are initialized (discussed below),
`line:`	`[ 0 | {1}]` defines whether to use the line search {default uses it},
`algo:`	`not applicable for PARAFAC2 as ALS is always used,`
`iterative:`	settings for iterative reweighted least squares fitting,
`blockdetails:`	'standard'
`missdat:`	this option is not yet active,
`samplemode:`	`[3]`, defines which mode should be considered the sample or object mode (do not change in PARAFAC2),
`constraints:`	{3x1 cell}, defines constraints on parameters (see PARAFAC), and
`coreconsist:`	`[ {'on'} | 'off' ]`, governs calculation of core consistency (turning off may save time with large data sets and many components).

The default options can be retrieved using: `options = parafac('options');`.

Note that samplemode should not be altered in PARAFAC2. See help on PARAFAC for help on the use of options for PARAFAC2. One important difference from PARAFAC is that constraints in the first mode *do not* apply to the estimated profiles, $\mathbf{A}_k$, themselves but only to **H**. It is generally adviced not to use constraints in the first mode.

Examples

`parafac2 demo` for a demonstration of the use of the PARAFAC2 algorithm.

`model = parafac2(X,5)` fits a five-component PARAFAC2 model to the array X using default settings.

`options = parafac2('options');` generates a set of default settings for PARAFAC2. `options.plots = 0;` sets the plotting off.

`options.init = 3;` sets the initialization of PARAFAC2 to orthogonalized random numbers.

`model = parafac2(X,2,options);` fits a two-component PARAFAC2 model with the settings defined in options.

`parafac2 io` shows the I/O of the algorithm.

See Also

`datahat`, `explode`, `gram`, `mpca`, `outerm`, `parafac`, `tld`, `tucker`, `unfoldm`

parsemixed

Purpose

Parse numerical and text data into a DataSet Object.

Synopsis

```
data = parsemixed(a,b)
```

Description

Given two inputs containing a numerical array `a` and a matching cell array containing text `b`, PARSEMIXED outputs a DataSet object with a "logical" interpretation of the numerical and text data. It identifies contiguous block of numbers and then attempts to interpret text as labels and label names for that block of data.

INPUTS:

`a` =	numerical array containing the numerical portion of the data to parse (NOTE: NaN's are OK).
`b` =	a cell array of the same size as (a) but containing any strings which were not interpretable as numbers.

OUTPUT:

`data` =	a DataSet object formed from the parsing of the input data.

Options

`options` =	a structure array with the following fields:
`labelcols:`	[] specifies one or more columns of the file which should be interpreted as text labels for rows even if parsable as numbers,
`labelrows:`	[] specifies one or more rows of the file which should be interpreted as text labels for columns even if parsable as numbers,
`includecols:`	[] Specifies one or more columns of the file which should be interpreted as the "include" field for ROWS of the matrix (i.e. this column specifies which rows should be included). Multiple items in this list will be combined using a logical "and" (all must be "1" to include field.
`includerows:`	[] Specifies one or more rows of the file which should be interpreted as the "include" field for COLUMNS of the matrix (see above notes about includecols).
`classcols:`	[] Specifies one or more columns of the file which should be interpreted as classes for rows of the data.
`classrows:`	[] Specifies one or more rows of the file which should be interpreted as classes for columns of the data.
`axisscalecols:`	[] Specifies one or more columns of the file which should be interpreted as axisscales for rows of the data.

`axisscalerows:`	`[]` Specifies one or more rows of the file which should be interpreted as axisscales for columns of the data.
`compactdata:`	`[ 'no' | {'yes'} ]` Specifies if columns and rows which are entirely excluded should be permanently removed from the table.
`waitbar:`	`[ 'off' | {'on'} ]` Specifies whether waitbars should be shown while the data is being processed.

See Also

`areadr, dataset, xclreadr, xlsreadr`

parseXML

Purpose

Convert XML file to a MATLAB structure.

Synopsis

```
object = parseXML(filename)
```

Description

Creates Matlab object from XML file. The format of the file must follow that used by ENCODEXML. Each XML tag will be encoded as a field in a Matlab structure. The top-level tag will be the single field in the top-level of the returned structure and all sub-tags will be sub-fields therein. Contents of those fields can be specified using the following attributes:

Tags with the attribute 'class' will be encoded using these rules:

class="numeric" : Contents of tag must be comma-delimited list of values with rows delimited by semicolons. Each row must have the same number of values (equal in length) or an error will result. Multi-way matricies can be encapulated in <tn mode="i"> tags where i is the mode that the enclosed item expands on (i>=3).

class="cell" : Contents encoded as Matlab cell. Format of contents is same as HTML table tags (<tr> for new row, <td> for new container/column) with the added tag of <tn mode="i"> to describe an multi-dimensional cell (see class="numeric").

class="string" : Contents encoded as string or padded string array. If multiple row string, each row should be enclosed in <sr> tags.

class="structure" : Used for struture arrays ONLY. Contents encoded into a structure array using array notation identical to that described for class="cell". If a structure is size [1 1] then it does not need to use array notation and must not be marked with this class attribute. Instead, the contents of the structure should simply be enclosed within the tag as sub-tags.

class="dataset" : Contents will be interpreted as a DataSet Object. Any tags which do not map to valid DataSet Object fields will be ignored. See the DataSet definition for details on valid fields and ENCODEXML for example of DataSet XML format.

When class is omitted, a single-entry (non-array) structure is assumed.

"Size" attribute: Tags of class "numeric", "cell", or "structure" (structure-array only) should also include the attribute size="[...]" which gives the size of the tag's contents. Value for size must be enclosed in square brackets and must be at least two elements long (use [0,0] for empty). For example <myvalue class="numeric" size="[3,4]"> says that the field myvalue will be numeric with 3 rows and 4 columns. Size can be multi-dimensional as needed (size="[2,4,6,2]" implies that the contents of the tag will give a 4-dimensional array of the given sizes)

If input (filename) is omitted, the user will be prompted for a file name to read.

See Also

`encodexml, xclreadr`

pca

Purpose

Perform principal components analysis.

Synopsis

```
pca
model = pca(data,ncomp,options);      %decomposition
pred  = pca(newdata,model,options);   %application
options = pca('options')
```

Description

Performs a principal component analysis decomposition of the input array `data` returning `ncomp` principal components. E.g. for an M by N matrix $\mathbf{X}$ the PCA model is $\mathbf{X} = \mathbf{TP}^T + \mathbf{E}$, where the scores matrix $\mathbf{T}$ is M by K, the loadings matrix $\mathbf{P}$ is N by K, the residuals matrix $\mathbf{E}$ is M by N, and K is the number of factors or principal components `ncomp`. The output `model` is a PCA model structure. This model can be applied to new data by passing the model structure to `PCA` along with new data `newdata` or by using `PCAPRO`. The output of PCA is a model structure with the following fields (see `MODELSTRUCT` for additional information):

`modeltype:`	`'PCA'`,
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`loads:`	cell array with model loadings for each mode/dimension,
`pred:`	cell array with model predictions for the input block (when `blockdetail='normal'` x-block predictions are not saved and this will be an empty array)
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

If the inputs are a M_{new} by N matrix `newdata` and and a PCA model `model`, then PCA applies the model to the new data. Preprocessing included in `model` will be applied to `newdata`. The output `pred` is structure, similar to `model`, that contains the new scores, and other predictions for newdata.

Note: Calling `pca` with no inputs starts the graphical user interface (GUI) for this analysis method.

Options

options =	a structure array with the following fields:
display:	`['off'	{'on'}]`, governs level of display to command window,
plots:	`['none'	{'final'}]`, governs level of plotting.
outputversion:	`[2	{3}]`, governs output format (discussed below),
algorithm:	`[{'svd'}	'maf'	'robustpca']`, algorithm for decomposition, Algorithm 'maf' requires Eigenvector's MIA_Toolbox.
preprocessing:	`{[]}`, cell array containing a preprocessing structure (see `PREPROCESS`) defining preprocessing to use on the data (discussed below),
blockdetails:	`[{'standard'}	'all']`, level of detail included in the model for predictions and residuals.
confidencelimit:	`[ {'0.95'} ]`, confidence level for Q and T2 limits. A value of zero (0) disables calculation of confidencelimits.
roptions:	structure of options to pass to robpca (robust PCA engine from the Libra Toolbox). `alpha:[ {0.75} ]`, (1-alpha) measures the number of outliers the algorithcarbuggym should resist. Any value between 0.5 and 1 may be specified. These options are only used when algorithm is 'robustpca'.

The default options can be retreived using: `options = pca('options');`.

OUTPUTVERSION

By default (`options.outputversion` = 3) the output of the function is a standard model structure `model`. If `options.outputversion = 2`, the output format is:

```
[scores,loads,ssq,res,reslm,tsqlm,tsq] = pca(xblock1,2,options);
```

where the outputs are

`scores` =	x-block scores,
`loads` =	x-block loadings
`ssq` =	the sum of squares information,
`res` =	the Q residuals,
`reslim` =	the estimated 95Found limit line for Q residuals,
`tsqlim` =	the estimated 95Found limit line for T^2, and
`tsq` =	the Hotelling's T^2 values.

PREPROCESSING

The `preprocessing` field can be empty [] (indicating that no preprocessing of the data should be used), or it can contain a preprocessing structure output from the `PREPROCESS` function. For example `options.preprocessing = {preprocess('default', 'autoscale')}`. This information is echoed in the output `model` in the `model.detail.preprocessing` field and is used when applying the PCA model to new data.

See Also

analysis, evolvfa, ewfa, explode, parafac, plotloads, plotscores, preprocess, ssqtable

pcaengine

Purpose

Principal components analysis computational engine.

Synopsis

```
[ssq,datarank,loads,scores,msg] = pcaengine(data,ncomp,options)
options = pcaengine('options')
```

Description

This function is intended primarily for use as the engine behind other more full featured PCA programs. The only required input is the data matrix `data`.

Optional inputs include the number of principal components desired in the output *ncomp*, and a structure containing optional inputs *options*. If the number of components *ncomp* is not specified, the routine will return components up to the rank of the data `datarank`.

The outputs are the variance or sum-of-squares captured table `ssq`, mathematical rank of the data `datarank`, principal component loadings `loads`, principal component scores `scores`, and a text variable containing any warning messages `msg`.

To enhance speed, the routine is written so that only the specified outputs are computed.

Options

`options` =	a structure array with the following fields:
`display:`	`[ 'off' | {'on'} ]`, governs level of display to command window,
`algorithm:`	`[ {'regular'} | 'big' | 'auto']`, tells which algorithm to use,
	`'regular'`, uses an SVD and calculates all eigenvectors and eigenvalues,
	`'big'`, calculates the “economy size” SVD, and
	`'auto'`, checks the size of the data matrix and automatically chooses between `'regular' and 'big'`

The default options can be retreived using: `options = pcaengine('options');`.

See Also

`analysis, evolvfa, ewfa, explode, parafac, pca, ssqtable`

pcapro

Purpose

Project new data onto an existing principal components model.

Synopsis

```
[scoresn,resn,tsqn] = pcapro(newdata,loads,ssq,reslm,tsqlm,plots)
[scoresn,resn,tsqn] = pcapro(newdata,pcamod,plots)
```

Description

Inputs can be in two forms: 1) as a list of input variables, or 2) as a single model structure variable returned by `ANALYSIS` or `PCA`.

1) If a list of input variables is used the inputs are the new data `newdata` scaled the same as the original data used to construct the model, the model loadings `loads`, the model variance info `ssq`, the limit for Q `reslm`, the limit for T^2 `tsqlm`, and an optional variable *plots* which suppresses plotting when set to 0 {default *plots* = 1}.

WARNING: Scaling for `newdata` should be the same as original data used to create the PCA model!

The I/O format is:

```
[scoresn,resn,tsqn] = pcapro(newdata,loads,ssq,q,tsq,plots)
```

2) If the PCA model is input as the single model structure variable returned by `ANALYSIS` or `PCA` then the inputs are the new data `newdata` in the units of the original data, the structure variable that contains the PCA model `pcamod`, and an optional variable *plots* which suppresses the plots when set to 0 {default *plots* = 1}.

NOTE: `newdata` will be preprocessed in `PCAPRO` using information stored in `pcamod` (`pcamod.detail.preprocessing`).

The I/O format is:

```
[scoresn,resn,tsqn] = pcapro(newdata,pcamod,plots)
```

Outputs are the new scores `scoresn`, residuals `resn`, and T^2 values `tsqn`. These are plotted if plots = 1 {default}.

See Also

`datahat, analysis, explode, modlpred, pca, simca, tsqmtx`

pcolormap

Purpose

Produces a pseudocolor map with labels.

Synopsis

```
pcolormap(data,maxdat,mindat)
pcolormap(data,xlbl,ylbl,maxdat,mindat)
```

Description

`PCOLORMAP` produces a pseudocolor map of the *M* by *N* input matrix `data`.

If data is class "double" the I/O format is:

```
pcolormap(data,xlbl,ylbl,maxdat,mindat)
```

If data is class "dataset" the I/O format is:

```
pcolormap(data,maxdat,mindat)
```

Optional inputs:

(xlbl) a character array with m rows of sample labels if empty no labels are included, if == 1 then xlbl = int2str([1:m]'); [xlbl = int2str([1:m]') used when size(xlbl,1)~=m],

(ylbl) a character array with n rows of variable labels if empty no labels are included, if ==1 then ylbl = int2str([1:n]'); [ylbl = int2str([1:n]') used when size(ylbl,1)~=n],

(maxdat) a user defined maximum for scaling the color scale {default = max(max(data))},

(mindat) a user defined minimum for scaling the color scale {default = min(min(data))}.

See Also

`corrmap, pcolor, rwb`

pcr

Purpose

Principal components regression: multivariate inverse least squares regession.

Synopsis

```
model = pcr(x,y,ncomp,options)      %calibration
pred  = pcr(x,model,options)        %prediction
valid = pcr(x,y,model,options)      %validation
options = pcr('options')
```

Description

`PCR` calculates a single principal components regression model using the given number of components `ncomp` to predict `y` from measurements `x`.

To construct a PCR model, the inputs are `x` the predictor x-block (2-way array class "double" or "dataset"), `y` the predicted y-block (2-way array class "double" or "dataset"), `ncomp` the number of components to to be calculated (positive integer scalar) and the optional structure, *options*. The output is a standard model structure `model` with the following fields (see `MODELSTRUCT`):

`modeltype:`	`'PCR'`,
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`reg:`	regression vector,
`loads:`	cell array with model loadings for each mode/dimension,
`pred:`	2 element cell array with model predictions for each input block (when `options.blockdetail='normal'` x-block predictions are not saved and this will be an empty array) and the y-block predictions.
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

To make predictions the inputs are `x` the new predictor x-block (2-way array class "double" or "dataset"), and `model` the PCR model. The output `pred` is a structure, similar to `model`, that contains scores, predictions, etc. for the new data.

If new y-block measurements are also available then the inputs are `x` the new predictor x-block (2-way array class "double" or "dataset"), `y` the new predicted block (2-way array class "double" or "dataset"), and `model` the PCR model. The output `valid` is a structure, similar to `model`, that contains scores, predictions, and additional y-block statistics etc. for the new data.

In prediction and validation modes, the same model structure is used but predictions are provided in the `model.detail.pred` field.

Note: Calling `pcr` with no inputs starts the graphical user interface (GUI) for this analysis method.

Options

`options` =	a structure array with the following fields:
`display:`	`['off'	{'on'}]`, governs level of display to command window,
`plots:`	`['none'	{'final'}]`, governs level of plotting,
`outputversion:`	`[2	{3}]`, governs output format (discussed below),
`preprocessing:`	`{[] []}`, two element cell array containing preprocessing structures (see `PREPROCESS`) defining preprocessing to use on the x- and y-blocks (first and second elements respectively),
`algorithm:`	`[{'svd'}	' robustpcr'	' correlationpcr']`, governs which algorithm to use. 'svd' is standard algorithm. 'robustpcr' is robust algorithm with automatic outlier detection. 'correlationpcr' is standard PCR with re-ordering of factors in order of y-variance captured.
`blockdetails:`	`['compact'	{'standard'}	'all']`, extent of predictions and raw residuals included in model. 'standard' = only y-block, 'all' x and y blocks.
`confidencelimit:`	`[ {'0.95'} ]`, confidence level for Q and T2 limits. A value of zero (0) disables calculation of confidence limits,
`roptions:`	structure of options to pass to rpcr (robust PCR engine from the Libra Toolbox). Only used when algorithm is 'robustpcr',
	`alpha : [ {0.75} ]`, (1-alpha) measures the number of outliers the algorithm should resist. Any value between 0.5 and 1 may be specified. These options are only used when algorithm is 'robustpcr'.
	`intadjust : [ {0} ]`, if equal to one, the intercept adjustment for the LTS-regression will be calculated. See ltsregres.m for details (Libra Toolbox).

The default options can be retreived using: `options = pcr('options');`.

OUTPUTVERSION

By default (`options.outputversion` = 3) the output of the function is a standard model structure `model. If options.outputversion = 2`, the output format is:

```
[b,ssq,t,p] = pcr(x,y,ncomp,options)
```

where the outputs are

`b` =	matrix of regression vectors or matrices for each number of principal components up to `ncomp`,
`ssq` =	the sum of squares information,
`t` =	x-block scores, and
`p` =	x-block loadings.

Note: The regression matrices are ordered in `b` such that each *Ny* (number of y-block variables) rows correspond to the regression matrix for that particular number of principal components.

See Also

`analysis`, `crossval`, `frpcr`, `modelstruct`, `pca`, `pls`, `preprocess`, `analysis`, `ridge`

pcrengine

Purpose

Principal components regression computational engine.

Synopsis

```
[reg,ssq,loads,scores,pcassq] = pcrengine(x,y,ncomp,options)
```

Description

`PCRENGINE` calculates the basic elements of a PCR model (see `PCR`).

Inputs are `x` the predictor x-block, and `y` the predicted y-block.

Optional input *ncomp* is the number of components to to be calculated (positive integer scalar). If the number of components *ncomp* is not specified, the routine will return components up to the rank of the x-block. Optional input *options*. is discussed below.

Outputs are the matrix of regression vectors `reg`, the sum of squares captured `ssq`, x-block loadings `loads`, x-block scores `scores`, and the PCA ssqtable (pcassq).

Note: The regression matrices are ordered in `b` such that each *Ny* (number of y-block variables) rows correspond to the regression matrix for that particular number of principal components.

Options

options =	a structure array with the following fields:
`display:`	`[ 'off' | {'on'} ]`, governs level of display to command window,
`sortorder:`	`[ {'x'} | 'y' ]`, governs order of factors in outputs. 'x' is standard PCR sort order (ordered in terms of X block variance captured). 'y' is Correlation PCR sort order (ordered in terms of Y block variance captured).

The default options can be retreived using: `options = pcrengine('options');`.

See Also

`analysis, pcr, pls`

peakfind

Purpose

Automated identification of peaks.

Synopsis

```
[i0,iw] = peakfind(x,width,tolfac,w,options)
[i0,iw] = peakfind(x,width,options)
```

Description

Given a set of measured traces (x) `PEAKFIND` attempts to find the location of the peaks. Different algorithms are available and each is discussed in the Algorithm Section.

INPUTS:

`x`	= MxN matrix of measured traces. Each $1xN$ row of (x) is an individual trace with potential peaks.
`width`	= number of points in Savitzky-Golay filter.

OPTIONAL INPUTS:

`tolfac`	= tolerance on the estimated residuals, peaks heights are estimated to be > tolfac*residuals {default: tolfac = 3}.
`w`	= odd scalar window width for determining local maxima {default: w = 3} (see LOCALMAXIMA).
`options`	= discussed below in the Options Section.

OUTPUTS:

`i0`	= $Mx1$ cell array with each cell containing the indices of the location of the major peaks for each of the M traces.
`iw`	= $Mx1$ cell array with each cell containing the indices of the location of the windows containing each peak in (i0). (If not included in the output argument list, it is not calculated and the algorithm is slightly faster.) .

Algorithm

Each peak finding algorithm uses the smoothed and second derivative data (see `SAVGOL`) and an estimate of the residuals. The smoothed and second derivative are estimated as:

```
d0 = savgol(x,width,2,0);
d2 = savgol(x,width,2,2);
```

The residuals are defined for the i^{th} row/trace as

```
residuals = sqrt(mean((x(i,:)-d0(i,:)).^2));
```

For `options.algorithm = 'd0'`, locates a candidate set of peaks (`pks`) by identifying local maxima (within the specified window size) in the smoothed data:

```
      pks = localmax(d0(i,:),w);
```

Next, the input (`tolfac`) is used to estimate two thresholds (`tol0`) and (`tol2`) using the smoothed and second derivative data:

```
      tol0 = tolfac*sqrt(mean((x(i,:)-d0(i,:)).^2));
      tol2 = tol0*(max(d2(i,:))-min(d2(i,:)))/ …
                 (max(d0(i,:))-min(d0(i,:)));
```

Finally, the set of major peaks are selected from the initial candidate set of peaks . To be accepted, the value of d0 and d2 at the peak location must surpass the estimated noise level of both d0 and d2 by the tolerance factor (tolfac).

```
      i0{i} = pks(d0(i,pks)>tol0 & d2(i,pks)<-tol2);
```

For `options.algorithm = 'd2'`, the algorithm operates similarly to what is described for d0 except that it locates candidate peaks as the local maxima on the second derivative data and to be accepted, a peak must only surpass the estimated noise level of d2 by the tolerance factor. That is, d0 is not considered at all in the calculation except to estimate the noise level.

For `options.algorithm = 'd2r'`, as with 'd2', 'd2r' locates peaks in the second derivative data, d2, but selects the final set as those peaks which have a "relative" height (difference between closest d2 peak valley and d2 peak top) which surpasses the estimated noise level of d2 by the tolerance factor, tolfac.

Options

`options` = structure array with the following fields:

`name:`	'`options`', name indicating that this is an options structure.
`algorithm:`	[{`'d0'`} | `'d2'` | `'d2r'`] selects an algorithm used to identify peak location. These algorithms are complimentary and may work differently in the presense of backgrounds and other peak shape effects. 'd0' : locates a candidate set of peaks by identifying local maxima (within the specified window size) in the smoothed data (d0). Next, a threshold on d0 and the second derivative (d2) is used to select a final set of peaks from this candidate set. To be accepted, the value of d0 and d2 at the peak location must surpass the estimated noise level of both d0 and d2 by the tolerance factor (tolfac). 'd2' : locates candidate peaks as local maxima in the smoothed 2nd derivative data (d2) and selects a final set of peaks as those candidate peaks which surpass (by the tolerance factor, tolfac) the estimated noise level of d2. d0 position or value is not considered in any part of the selection except to estimate the noise level. 'd2r' : as with 'd2', 'd2r' locates peaks in d2, but selects the final set as those peaks which have a "relative" height (difference between closest d2 peak valley and d2 peak top) which surpasses (by the tolerance factor, tolfac) the estimated noise level of d2.
`npeaks:`	The maximum number of peaks to find.

{'all'} chooses all peaks that are > tolfac.
1,2,3, ... integer maximum number of peaks.

See Also

`fitpeaks`, `localmax`

peakfunction

Purpose

Outputs the estimated peaks from parameters in (`peakdef`)

Synopsis

```
[y,peakdef] = peakfunction(peakdef,ax)
```

Description

Given the multi-record standard peak structure (`peakdef`) and the corresponding wavelength/frequency axis (`ax`), the peak parameters in the field (`peakdef.param`) are used to generate peaks. This function is called by `PEAKFITS` and the result is the output (`fit`), and the peak area estimates in (`peakdef`) are updated. See `PEAKFITS` for more information. This function calls `PEAKGAUSSIAN`, `PEAKLORENTZIAN`, `PEAKPVOIGT1`, and `PEAKVOIGT2`.

INPUTS:

`peakdef`	= standard peak structure (see `PEAKSTRUCT`) output by `fitpeaks`.
`ax`	= corresponding wavelength/frequency axis. This is also input to the function `FITPEAKS`. Peak positions are based on this axis.

OUTPUTS:

`y`	= estimated peaks based on the parameters in the input (`peakdef`).
`peakdef`	= the original input (`peakdef`) with the area field estimated.

Examples

```
  ax            = 0:0.1:100;
  y             = peakgaussian([2 51 8],ax);%Make known peak
%Define first estimate and peak type
  peakdef       = peakstruct;
  peakdef.param = [0.1  43   5];      %coef, position, spread
  peakdef.lb    = [0.0   0   0.0001]; %lower bounds on param
  peakdef.penlb = [1 1 1];
  peakdef.ub    = [10 99.9  40];      %upper bounds on params
  peakdef.penub = [1 1 1];
%Estimate fit and plot
  yint    = peakfunction(peakdef,ax);
  [peakdef,fval,exitflag,out] = fitpeaks(peakdef,y,ax);
  yfit    = peakfunction(peakdef,ax); figure
  plot(ax,yint,'m',ax,y,'b',ax,yfit,'r--')
  legend('Initial','Actual','Fit')
```

See Also

`fitpeaks`, `peakgaussian`, `peaklorentzian`, `peakpvoigt1`, `peakpvoigt2`, `peakstruct`

peakgaussian

Purpose

Outputs a Gaussian function, Jacobian, and Hessian for a given set of input parameters and axis.

Synopsis

```
[y,y1,y2] = peakgaussian(x,ax)
```

Description

Given a 3-element vector of parameters (x) and a $1xN$ vector of independent variables e.g. a wavelength or frequency axis (ax), PEAKGAUSSIAN outputs a Gaussian peak (y). If more than one output is requested, it also outputs the Jacobian (y1) and Hessian (y2). Derivatives are with respect to the parameters and are evaluated at (x). This function is called by PEAKFUNCTION.

INPUTS:

x	= 3 element vector with parameters x(1) = coefficient x_1, x(2) = mean x_2, and x(3) = spread x_3.
ax	= $1xN$ vector of independent variables e.g. a wavelength or frequency axis with elements a_i, $i = 1,\ldots,N$.

OUTPUTS:

y	= $1xN$ vector with the Gaussian function, $y_i = f\left(a_i, \mathbf{x}\right)$.
y1	= $3xN$ matrix of the Jacobian of f evaluated at (x).
y2	= $3x3xN$ matrix of the Hessian of f evaluated at (x).

Algorithm

The function is

$$f\left(a_i, \mathbf{x}\right) = x_1 \, e^{\frac{-\left(a_i - x_2\right)^2}{2x_3^2}}$$

Examples

```
%Make a single known peak
  ax            = 0:0.1:100;
  y             = peakgaussian([2 51 8],ax);
  plot(ax,y)
```

See Also

peakfunction, peaklorentzian, peakpvoigt1, peakpvoigt2, peakstruct

peakidtext

Purpose

Writes peak ID information on present graph of a set of peaks.

Synopsis

```
h = peakidtext(peakdef)
```

Description

When a set of peaks is plotted, `PEAKIDTEXT` can be used to put the peak id (`peakdef.id`) on the graph (see `PEAKSTRUCT`). For example, if (`ax`) is the wavelength, frequency, or time axis and (`y`) is a set of peaks then, for an initial guess given in (`peakdef`) the fit parameters are obtained using:

```
peakdefo = fitpeaks(peakdef,y,ax);
```

A plot can be made using:

```
plot(ax,y,'b',ax,peakfunction(peakdefo,ax),'r')
```

Next, labels are put on the graph using:

```
peakidtext(peakdefo)
```

This also puts a vertical line at the peak center and puts the text label, based on the contents of the (`peakdefo.id`) field, near the peak maximum.

INPUT:

`peakdef`	= a standard peak structure (see `PEAKSTRUCT`).

OUTPUT:

`h`	= vector of handles corresponding to the individual text labels.

See Also

`fitpeaks, peakfunction, peakstruct`

peaklorentzian

Purpose

Outputs a Lorentzian function, Jacobian, and Hessian for a given set of input parameters and axis.

Synopsis

```
[y,y1,y2] = peaklorentzian(x,ax)
```

Description

Given a 3-element vector of parameters (x) and a $1xN$ vector of independent variables e.g. a wavelength or frequency axis (ax), PEAKLORENTZIAN outputs a Lorentzian peak (y). If more than one output is requested, it also outputs the Jacobian (y1) and Hessian (y2). Derivatives are with respect to the parameters and are evaluated at (x). This function is called by PEAKFUNCTION.

INPUTS:

x	= 3 element vector with parameters
	x(1) = coefficient x_1,
	x(2) = mean x_2, and
	x(3) = spread x_3.
ax	= $1xN$ vector of independent variables e.g. a wavelength or frequency axis with elements a_i, $i = 1,\ldots,N$.

OUTPUTS:

y	= $1xN$ vector with the Lorentzian function, $y_i = f\left(a_i, \mathbf{x}\right)$.
y1	= $3xN$ matrix of the Jacobian of f evaluated at (x).
y2	= $3x3xN$ matrix of the Hessian of f evaluated at (x).

Algorithm

The function is

$$f\left(a_i, \mathbf{x}\right) = x_1\left[1 + \left(\frac{a_i - x_2}{x_3}\right)^2\right]^{-1} = x_1\left[\frac{x_3^2}{x_3^2 + \left(a_i - x_2\right)^2}\right]$$

Examples

```
%Make a single known peak
  ax             = 0:0.1:100;
  y              = peaklorentzian([2 51 8],ax);
  plot(ax,y)
```

See Also

peakfunction, peakgaussian, peakpvoigt1, peakpvoigt2, peakstruct

peakpvoigt1

Purpose

Outputs a pseudo-Voigt function, Jacobian, and Hessian for a given set of input parameters and axis.

Synopsis

```
[y,y1,y2] = peakpvoigt1(x,ax)
```

Description

Given a 4-element vector of parameters (x) and a $1xN$ vector of independent variables e.g. a wavelength or frequency axis (ax), PEAKPVOIGT1 outputs a pseudo-voit peak (y). If more than one output is requested, it also outputs the Jacobian (y1) and Hessian (y2). Derivatives are with respect to the parameters and are evaluated at (x). This function is called by PEAKFUNCTION.

INPUTS:

x = 4 element vector with parameters
x(1) = coefficient x_1,
x(2) = mean x_2,
x(3) = spread x_3, and
x(4) = fraction Gaussian x_4.

ax = $1xN$ vector of independent variables e.g. a wavelength or frequency axis with elements a_i, $i = 1,\ldots,N$.

OUTPUTS:

y = $1xN$ vector with the Lorentzian function, $y_i = f\left(a_i, \mathbf{x}\right)$.

y1 = $4xN$ matrix of the Jacobian of f evaluated at (x).

y2 = $4x4xN$ matrix of the Hessian of f evaluated at (x).

Algorithm

The function is

$$f\left(a_i, \mathbf{x}\right) = x_1 \left[x_4 \, \mathrm{e}^{\frac{-4\ln(2)\left(a_i - x_2\right)^2}{x_3^2}} + \left(1 - x_4\right) \left[\frac{x_3^2}{\left(a_i - x_2\right)^2 + x_3^2} \right] \right]$$

Examples

```
%Make a single known peak
  ax            = 0:0.1:100;
  y             = peakpvoigt1([2 51 8 0.5],ax);
  plot(ax,y)
```

See Also

peakfunction, peakgaussian, peaklorentzian, peakpvoigt2, peakstruct

peakpvoigt2

Purpose

Outputs a pseudo-Voigt function, Jacobian, and Hessian for a given set of input parameters and axis.

Synopsis

```
        [y,y1,y2] = peakpvoigt2(x,ax);
```

Description

Given a 4-element vector of parameters (`x`) and a $1xN$ vector of independent variables e.g. a wavelength or frequency axis (`ax`), `PEAKPVOIGT2` outputs a pseudo-voigt peak (`y`). If more than one output is requested, it also outputs the Jacobian (`y1`) and Hessian (`y2`). Derivatives are with respect to the parameters and are evaluated at (`x`). This function is called by `PEAKFUNCTION`.

INPUTS:

`x`	= 4 element vector with parameters `x(1)` = coefficient x_1, `x(2)` = mean x_2, `x(3)` = spread x_3, and `x(4)` = fraction Gaussian x_4.
`ax`	= $1xN$ vector of independent variables e.g. a wavelength or frequency axis with elements a_i, $i = 1,\ldots,N$.

OUTPUTS:

`y`	= $1xN$ vector with the Lorentzian function, $y_i = f\left(a_i, \mathbf{x}\right)$.
`y1`	= $4xN$ matrix of the Jacobian of f evaluated at (`x`).
`y2`	= $4x4xN$ matrix of the Hessian of f evaluated at (`x`).

Algorithm

The function is

$$f\left(a_i, \mathbf{x}\right) = x_1 \left[x_4 \, \mathrm{e}^{\frac{-\left(a_i - x_2\right)^2}{2x_3^2}} + \left(1 - x_4\right) \left[\frac{x_3^2}{\left(a_i - x_2\right)^2 + x_3^2} \right] \right]$$

Examples

```
%Make a single known peak
  ax            = 0:0.1:100;
  y             = peakpvoigt2([2 51 8 0.5],ax)
  plot(ax,y)
```

See Also

peakfunction, peakgaussian, peaklorentzian, peakpvoigt1, peakstruct

peakstruct

Purpose

Makes an empty standard peak definition structure.

Synopsis

```
peakdef = peakstruct(fun,n)
```

Description

The output of `PEAKSTRUCT` is an empty standard peak structure, or multi-record peak structure.

No input is required. Optional inputs can be used to create different types of default peak definitions in each of the structure records.

OPTIONAL INPUTS:

`fun`	= Peak function name {default = `'Gaussian'`}. Available peak names (shapes) are: `'Gaussian'`, `'Lorentzian'`, `'PVoigt1'`, and `'PVoigt2'`.
`n`	= Number of records to include in the (`peakdef`) structure.

OUTPUTS:

`peakdef`	= A structure array with the following fields:
`name`:	`'Peak'`, indentifies (`peakdef`) as a peak definition structure.
`id`:	integer or character string peak identifier.
`fun`:	peak function name {e.g.`'Gaussian'`}.
`param`:	$1xP$ vector of parameters for each peak function:
`fun`	= `'Gaussian'`; `param` = [height, position, width].
`fun`	= `'Lorenzian'`; `param` = [height, position, width].
`fun`	= `'PVoigt1'`; `param` = [height, position, width, fraction Gaussian], where $0 \leq$ fraction Gaussian ≤ 1.
`fun`	= `'PVoigt2'`; `param` = [height, position, width, fraction Gaussian], where $0 \leq$ fraction Gaussian ≤ 1. Descriptions of the functions and parameters are given in the Algorithm section of the `FITPEAKS` entry in the reference manual. Also see `PEAKFUNCTION`.
`lb`:	$1xP$ vector of lower bounds on (`.param`).
`penlb`:	$1xP$ vector of penalties for lower bounds. If an entry is 0, then the corresponding lower bound is not active.
`ub`:	$1xP$ vector of upper bounds on (`.param`).

penub: $1xP$ vector of penalties for upper bounds. If an entry is 0, then the corresponding upper bound is not active.

Examples

```
peakdef = peakstruct('',3);
disp(peakdef(2))

peakdef(2) = peakstruct('PVoigt1');
peakdef(2).id = '2: Voigt';
disp(peakdef(2))
```

See Also

`fitpeaks`, `peakfunction`, `peakgaussian`, `peaklorentzian`, `peakstruct`, `peakpvoigt1`, `peakpvoigt2`

percentile

Purpose

Finds percentile point (similar to MEDIAN).

Synopsis

```
    s = percentile(x,y)
```

Description

`PERCENTILE` finds the point in the data `x` where the fraction `y` has lower values. Input `x` is a *M*x*N* data array, and `y` is a percentile where 0<y<1.

The output is a *1* by *N* vector `s` of percentile points (`PERCENTILE` works on the columns of `x`.

See Also

`median`

ploteigen

Purpose

Extracts information from a model needed to construct a dataset object for PLOTGUI.

Synopsis

```
a = ploteigen(modl, options)
```

Description

Extracts the variance captured, eigenvalue, and RMSE (root-mean-squared error) information from a model structure for viewing using PLOTGUI. The inputs are a standard model structure, *modl*, and an optional options structure, *options*, described below. The output, *a*, is a DataSet object which can be passed to PLOTGUI for viewing.

Options

`plots:`	['none' | 'final' | {'auto'}] governs plotting behavior, 'auto' makes plots if no output is requested {default}.
`figure:`	`['off'	{'on'}]`, governs level of display to command window.

See Also

`analysis, modelstruct, pca, pcr, plotgui, plotloads, pls`

plotgui

Purpose

Interactive data viewer.

Synopsis

```
fig = plotgui(data)
fig = plotgui(data,'PropertyName',PropertyValue,...)
fig = plotgui('update','PropertyName',PropertyValue,...)
```

Description

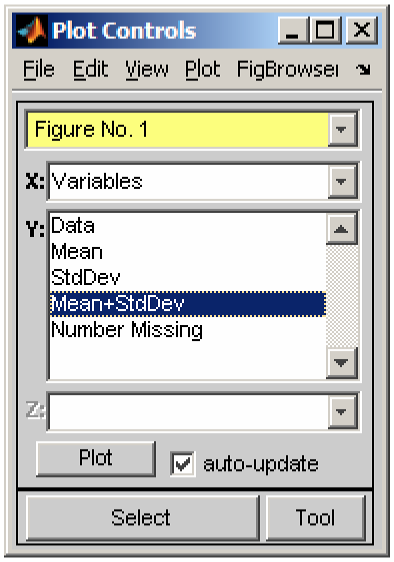
Plots input data `dat` and provides a control toolbar in the **Plot Controls** window to select portions of the data to view. The toolbar allows interactive selection, exclusion, and classing of rows or columns of data. The `PLOTGUI` command has various display options that are given as *'PropertyName', PropertyValue* pairs or as a single keyword. Properties and Keywords are discussed below. To modify options for an existing `PLOTGUI` figure without providing new data, use the `'update'` keyword.

`PLOTGUI` returns the handle of the figure in which the data is displayed (`fig`).

Input `dat` can be class “double” or “dataset”. The description given below is generally listed for two-way data arrays. Options specific to data that are three-way or image are noted explicitly. `PLOTGUI` uses the dataset labels, classes, etc. when `dat` is class “dataset”.

Plot Controls Toolbar

The toolbar consists of 1) a menu bar with **File**, **Edit**, and **View** menus, 2) a figure selection dropdown menu, 3) three axis menus (labeled **x**, **y**, and **z**), 4) plot update controls **Plot** button and **auto-update** checkbox, and **Select** button.



Each figure in the figure selection dropdown menu menu can be modified by the `PLOTGUI` controls. Selecting a figure from this menu will bring that figure into view and indicate the selected axis menu settings. A "+" or a "*" next to a figure's name indicates that it is linked with another figure (see Duplicate Figure below).

The axis menus (labeled **x**, **y**, and **z**) select what parts of the data should be used for the plot. Each column or row selected in the y-axis menu will be plotted against the column, row or index selected in the x-axis menu. If any selection is made on the z-axis menu, then each y-axis selection is also plotted against the column or row selected in the z-axis menu to make a three-dimensional plot.

If the input `dat` is three-way it is assumed to be a multivariate image, and the y-axis is slice or slab and the figure default is `imagesc(dat(:,:,1))`. This is also true if `dat` is class “dataset” with the `type` field set to `'image'` or `'image'`.

If the **auto-update** checkbox is selected, figures are updated automatically when new axis-menu selections are made. Otherwise, the **Plot** button must be pressed before any changes are reflected in the figure.

View Menu

Various options associated with the viewed data are contained in the **View** menu. The specific options depend on the data being plotted. The **View** menu options are listed below.

Table:	Opens a **Plotted Data** window that lists the numerical values of the plotted data.
Numbers:	Displays the index number next to each plotted point.
Labels:	Displays available lables next to each plotted point. If no labels are available this option is greyed out.
Classes:	Uses available class information to give each plotted point a different symbol. If no class information is available this option is greyed out. The fly-out menu includes any class sets defined in the dataset as well as options to "Outline Class Groups". Group outlining allows drawing of lines to either enclose all samples in a group ("border points") or as a confidence boundry ("confidence ellipse").
Declutter Labels:	Controls the label/number decluttering options. Automatic modes remove labels when they overlap. "Selected Only" removes labels on all points except those which have been selected using the standard selection tools.
Label Angle:	Changes the angle of (i.e. rotates) all labels in a plot.
Excluded Data:	Shows any points which have been “excluded” from the data set.
Axis Lines:	Places lines through the origin.
Log Scales:	Switches axes between log and linear scaling.
Auto y-scale:	When enabled, all plotted data items are scaled so that their y-axis values are on a similar scale (that is, they are each baselined and normalized). The different methods for y-scaling include: Sum, Length, Max. In each case, the given property is set equal to 1 for each plotted data item. In addition, if the plot has been zoomed, the y-scaling method is based only on the currently visible data. The scaling can be recalculated for any given zoomed view by selecting "Scale from current zoom".
Auto Contrast:	Contrast enhancement for a slice/slab for multivariate images (only available when the data are 3-way or type image).
Duplicate Figure:	Creates a duplicate copy of the current figure that is linked to the current figure i.e. if one figure is modified the other automatically changes to reflect the modification. The parent figure will have a "+" next to its

name in the figure selection dropdown menu and the child figure will have a "*".

Spawn Figure:	Creates a duplicate copy of the current figure that is not controled by the **Plot Controls** toolbar. This is a simple MATLAB figure.
Dock Controls:	When checked, the **Plot Controls** toolbar are "docked" next to the controled figure.
Settings:	Allows the user to modify other view settings.

Plot Menu

Selects the "mode" in which the current data should be viewed. This can be either a summary of any given mode (Data Summary mode) or one of the standard modes of a data matrix including the rows, columns, or slabs.

Data Summary:	Plots all the data, the mean, the standard deviation, or the mean ± the standard deviation. For Variables (columns) or Samples (rows) depending on what is selected in the x-axis.
Rows:	Plots the data across rows selecting which rows (usually samples) to view.
Columns:	Plots the data down the columns selecting which columns (usually variables) to view.
Slabs:	Uses `IMAGESC` to view a slice/slab of a 3-way array (only available when the data are 3-way).

Selection using the Select button

The **Select** button allows the user to select plotted points in the current figure. After clicking **Select**, the current figure will be brought to the front and points are selected using the current selection tool (selected using the **Tool** button; see also Edit/Selection Mode menu). To extend a selection (i.e. add new points to the already selected points), use the **shift-key** while pressing the mouse button. To remove points from the selection, use the **control-key** while pressing the mouse button. To keep from making any selection, press "**Esc**" or "**Escape**".

Edit Menu

The **Edit** menu contains various actions relating to selections. The specific actions available depends on the current selection and `PLOTGUI` mode. The **Edit** menu options are listed below.

Select All:	Selects all plotted points.
Deselect All:	Deselects all plotted points.

Select Class: Select all points of a given class or classes in the data (if any classes are defined).

Select Excluded: Selects all points which are currently excluded (see **View/Excluded Data**).

Selection Mode: Menu used to choose selection mode from the following:

Box: Click and drag a rubber band box around points,

Polygon: Click to mark the corners of a polygon around points and click on the intial point or press [Enter] to close the polygon,

Circle: Click to mark center of a circle, then click to mark the outside edge of the circle,

Ellipse: Click to mark center of an ellipse, click again to mark the minor axis size for the ellipse, then complete the selection by clicking to mark the size and direction of the major axis for the ellipse

Paintbrush: Click and drag to "paint" a selection onto points,

Lasso: Click and drag a free-form line to "ensnare" the points,

Single X: Click to select a single point on the x-axis,

Single Y: Click to select a single point on the y-axis,

X Range: Click and drag to select a range of points on the x-axis,

Y Range: Click and drag to select a range of points on the y-axis, and

Nearest: Click to select the nearest point.

Multiple Nearest: Click to select the nearest point, repeated until the [Enter] key is pressed.

Include All: Includes all excluded points (whether or not they are selected).

Exclude Selection: Excludes (soft deletes) the selected points from the data set. See **View/Excluded Data**.

Include Selection: Includes the selected points in the data set. See **View/Excluded Data**.

Include Only Selection: Exclude all *unselected* points from the data set i.e. keep only the selected points.

Info on Selection: Get information on selected point (only available when a single point is selected).

Set Class: Set the class of the selected points.

Exclude Plotted Data: Excludes all items currently selected in the y-axis menu for plotting. Note that unlike the other exclusion options in this menu, this and the next two options act on the mode selected in the **Plot** menu.

Include Plotted Data: Includes all items currently selected in the y-axis menu for plotting.

Include Only Plotted: Includes all items currently selected in the y-axis menu for plotting and only those items (all others are excluded).

File Menu

The **File** menu contains various actions relating to files. The **File** menu options are listed below.

Load Data: Creates an interface for the user to load data into PLOTGUI from the base workspace or a file.

Save Data: Creates an interface for the user to save data from PLOTGUI to the base workspace or a file.

Open in Editor: Opens the given dataset in a linked DataSet Editor window.

Export Figure: Allows exporting the current figure to Various external programs (exporting will not function correctly if the given program is not installed on the computer).

Save Selected Indices: Saves the current selection as a vector of indices. This can be used with the Load Selected Indices command to quickly store and reload different selections.

Load Selected Indices: Load a vector of indices to use as a selection.

Reset Controls: Refreshes Plot Controls. Useful if graphical objects are not correctly aligned.

Properties and Keywords

The following is a list of available properties. Each should be included as a *'PropertyName', PropertyValue* pair in an initial PLOTGUI call or a PLOTGUI 'update' call. Note that calls to PLOTGUI for *'PropertyName'* and *PropertyValue* are case insensitive.

The current value of almost all properties can be retrieved using the getappdata function on the PLOTGUI figure and requesting the property of interest. Note that calls to GETAPPDATA are case sensitive and *'PropertyName'* must be in all lower-case. The I/O format is:

```
currentvalue = getappdata(fig,'propertyname')
```

where fig is the handle of the PLOTGUI figure. If 'propertyname' is not included getappdata(fig) will list all the properties and their current values. Properties and their possible values follow:

```
AxisMenuValues:{[x] [y] [z]}, Two or three element cell containing
indices or strings indicating which item, or items, to select in
each of the three axis pull down menus. In [x] or [y] a value of 0
(zero) means to select index number. In [z] a value of 'none'
means to not use the z-axis.
AxisMenuDefaults:Axis menu defaults are axis menu values used if
the axis menu values can not be restored. The input format is the
same as axismenuvalues.
Figure:[scalar integer], Figure on which data should be plotted
{default is current figure}.
```

New:Key word – no associated *PropertyValue*. Creates a new figure for display of data. This is equivalent to an initial PLOTGUI call.
PlotBy:[scalar integer], Dimension (mode) for the axis menu selections: 0 = special "data browser", 1 = rows, 2 = columns, etc. (see **View** menu). The default is 2 or the number of modes in the data if larger than 2-way.
VSIndex:[1 1] {default}, Two element vector indicating if "Index" should be offered on x and y axis menus. A 1 indicates that it should be offered as a selection and a 0 indicates that it should not e.g. [1 1] indicates that it should be offered for both the x-axis and y-axis.

The following are **image specific** properties:
Image:Key word – no associated *PropertyValue*. Unfolds a 2 or 3-way array and displays it as and image, allowing selection, classing, and exclusion of individual pixels.
Unfold:Key word – no associated *PropertyValue*. Pseudonym for "image".
AsImage:Key word – no associated *PropertyValue*. Display 3-way data that have already been unfolded as an image allowing selection, classing, and exclusion of individual pixels.

The following are view properties:

ViewClasses: [1] {default}, Turns on **View/Classes** menu. A 0 (zero) turns it off.
ViewExcludedData:[1] {default}, Turns on **View/Excluded Data** menu. A 0 (zero) turns it off.
ViewLabels: [1] {default}, Turns on **View/Labels** menu. A 0 (zero) turns it off.
ViewNumbers: [1] {default}, Turns on **View/Numbers** menu. A 0 (zero) turns it off.

The following are plot properties:

LineStyle: <string>, Defines line style (see PLOT).
PlotType: <string>, String used to select plot type {default [ ] is atuomatic selection}. Other values are 'scatter', 'bar', 'none' ('none' = do no plotting).
SelectionMarker:<string>, Defines marker style for selected points (see PLOT).

The following are selection properties:

SelectionMode: <string>, Defines the selection mode. This can be any string listed under **View/Selection Mode** above. Also see GSELECT.
BrushWidth: [scalar integer number of pixels], This defines the brush width for use when selectionmode = 'paintbrush'. See **View/Selection Mode/Paintbrush**.
NoSelect: [0] {default}, When set to 0 this allows selections. When set to 1 no selection is allowed.

`NoInclud:`	`[0]` {default}, When set to 0 this allows changes to the `inlclud` field (i.e. it allows data to be excluded). When set to 1 no changes to the `inlclud` field are allowed (i.e. data can not be excluded).

The following are on-event properties:

```
CloseGUICallback:Command(s) to execute when the figure is closed.
IncludChangeCallback:Command executed when includ field of the
dataset is modified.
InfoReqCallback:Command executed when information on a selected
point is requested.
PlotCommand:Command executed after plotting (e.g. draw limits,
assign ButtonDownFcns, modifiy axes, …).
SelectionChangeCallback:Command executed when a selction is made.
SetClassCallback:Command executed when the class field of the
dataset is changed.
```

The following are confidence limit properties:

`ConfLimits:`	Boolean flag to make "Conf. Limits" controls visible. 1 = show controls (`PLOTGUI` does nothing with these controls, thus the routine specified in 'plotcommand' must be set to use values).
`LimitsValue:`	Value for Conf. Limits editbox.
`ShowLimits:`	Value for "Conf. Limits" checkbox (1 = checked).

The following are figure linking properties (WARNING! Modifying these settings can lead to unexpected results!):

`Children:`	Add new child of the current `PLOTGUI` figure (all child figures are updated when their parent is updated and closed when their parent is closed). Note: this property will only allow adding of additional children. Other modifications must be made using `setappdata`.
`ControlBy:`	Reassign control for `PLOTGUI` figure.
`Parent:`	Assign a parental link (Forces the parent figure to update if this figure is updated, also see `'Children'`).
`TimeStamp:`	Time-stamp of last time this figure was updated (can be set to any string to isolate figure from updating by parents).

The following are other miscellaneous properties:

`UIControl:` Add extra uicontrol(s) to `PLOTGUI` control toolbar for use with current figure (buttons, sliders, etc.). The value passed to `UIControl` should be a cell in which each entry is the tag of a new object to create and the value of that field should contain a cell of uicontrol property / value pairs to set for that object. For example:

```
myobj.mybtn = {'style', 'pushbutton', 'string', 'new fig',
'callback', 'figure'};
plotgui('update','uicontrol',myobj)
```

creates a button with the tag `'mybtn'` on the controls for the current figure.

If the cell for any object does not contain a `'position'` property for the object, `PLOTGUI` will manage the object's position.

The following are read-only properties. These properties can only be viewed and are only accessible through the MATLAB getappdata command.

`Selection:`	Cell array of currently selected values. Usually the same format as "`includ`" field of DataSet object where each cell represents the index of selected items in each dimension {rows, columns, slabs, ...}.
	When selecting elements in greater than 2-dimensional data (and without the use of the `'image'` keyword), two cells of this field will be pairs of selected indices: `{x,y,[]}` or `{[],y,z}`.
`FigureType:`	`'PlotGUI'`
`DataSet:`	DataSet used in figure (or pointer to figure with actual dataset)
	Note: This is set by calling `PLOTGUI` with a new dataset as an input. The actual DataSet can be retrieved using the `getdataset` command (see below).

The following are other valid figure properties. See the MATLAB doc umentation on FIGURE properties for additional information.

HandleVisibility, MenuBar, Name, NumberTitle, Position, Resize, Tag, ToolBar, Units, UserData, Visible, WindowStyle

Examples

`fig = plotgui(mydata)` plots `mydata` allowing user to select which column(s) of `mydata` to plot using pull-down menus. Figure number of plot is returned.

`plotgui(mydata,'plotby',1)` or `plotgui(mydata,'plotby','rows')` plots `mydata` as in first example except that rows of `mydata` (dimension 1) are used for pull-down menus instead of columns. Note: When a `PLOTGUI` property is set for a given figure, the new value will be retained until a new value for that property is provided, even if new data is plotted on the same `PLOTGUI` figure.

`fig = plotgui(mydata,'plotby',1,'axismenuvalues',{[1] [2 3]})` plots rows of `mydata`; sets controls with row 1 selected for the x-axis and rows 2 and 3 selected for the y-axis. Use:

```
getappdata(fig,'axismenuvalues')
```

to retrieve current axis menu settings. axispulldown

`plotgui(mydata,'viewclasses',1)` plots `mydata` using symbols to identify the classes stored in dataset `mydata`. Use a value of 0 (zero) to turn off viewclasses.

`plotgui('update','viewclasses',1)` Turns on `viewclasses` property for current figure without having to pass data to plot (substitute string `'update'` for data) `mydata = plotgui('getdataset',fig)` Retrieves `mydata` from figure `fig`.

`plotgui(myimage,'image')` plots 3-way image `myimage` selecting slabs of the image for display. The keyword `'image'` allows selection, classing and exclusion of pixels in the image.

See Also

`dataset`, `analysis`, `plotloads`, `plotscores`

plotloads

Purpose

Extract and display loadings information from model.

Synopsis

```
a = plotloads(modl,options)
a = plotloads(loads,labels,classes)
options = plotloads('options')
```

Description

Given a standard model structure, relevant loading information (e.g. labels) is collected and passed to `PLOTGUI` for plotting. The input is the model containing loadings to plot `modl`. (e.g. see `MODELSTRUCT`). Optional input *options* is discussed below.

Input `loads` is a *N* by *K* loadings matrix (class "double"). Optional input *labels* is a character or cell array with *N* rows containing sample labels, and optional input *classes* is a vector with *N* integer elements of class identifiers.

If no output is requested then `PLOTLOADS` initiates an interactive plotting utility to make loadings plots. If an output is requested, no plots are made, and the output `a` is a dataset object containing the loadings and labels, etc.

Options

options =	a structure array with the following fields:
`display:`	`[{'on'}	'off']`, governs level of display,
`plots:`	`['none'	'final'	{'auto'} |]`, governs plotting behavior, `'auto'` makes plots if no output is requested {default}, and
`figure:`	`[]`,governs where plots are made, when `figure = []` plots are made in a new figure window {default}, this can also be a valid figure number (i.e. figure handle).

The default options can be retreived using: `options = plotloads('options');`.

See Also

`analysis, modelstruct, pca, pcr, plotgui, plotscores, pls`

plotscores

Purpose

Extract and display scores information from model.

Synopsis

```
a = scoresplot(modl,options)
a = scoresplot(modl,pred,options)
a = plotscores(scores,labels,classes)
options = plotscores('options')
```

Description

Given a standard model structure, relevant scores information (e.g. labels) is collected and passed to `PLOTGUI` for plotting. The input is the model containing scores to plot `modl`. (e.g. see `MODELSTRUCT`). A second input pred contains a test or validation structrure (see `PCA`) that can be plotted with scores in `modl`. Optional input *options* is discussed below.

Input `scores` is a *M* by *K* scores matrix (class "double"). Optional input *labels* is a character or cell array with *M* rows containing sample labels, and optional input *classes* is a vector with *M* integer elements of class identifiers.

If no output is requested then `PLOTSCORES` initiates an interactive plotting utility to make scores plots. If an output is requested, no plots are made, and the output `a` is a dataset object containing the scores and labels, etc.

Options

`options` =	a structure array with the following fields:
`display:`	`[{'on'}	'off']`, governs level of display,
`plots:`	`['none'	'final'	{'auto'} |]`, governs plotting behavior, `'auto'` makes plots if no output is requested {default},
`figure:`	`[]`,governs where plots are made, when `figure = []` plots are made in a new figure window {default}, this can also be a valid figure number (i.e. figure handle), and
`sct:`	`[0	{1}]`, tells whether to plot cal (`modl` scores) with test (`pred` scores), `sct = 1` plots original calibration data with prediction set {default}.

The default options can be retreived using: `options = plotscores('options');`.

See Also

`analysis, modelstruct, pca, pcr, plotgui, plotloads, pls`

pls

Purpose

Partial least squares regression for univariate or multivariate y-block.

Synopsis

```
model = pls(x,y,ncomp,options)       %calibration
pred  = pls(x,model,options)         %prediction
valid = pls(x,y,model,options)       %validation
options = pls('options')
```

Description

`PLS` calculates a single partial least squares regression model using the given number of components `ncomp` to predict `y` from measurements `x`.

To construct a PLS model, the inputs are `x` the predictor block (2-way array class "double" or class "datadet"), `y` the predicted block (2-way array class "double" or class "datadet"), `ncomp` the number of components to to be calculated (positive integer scalar), and the optional structure, *options*. The output is a standard model structure `model` with the following fields (see `MODELSTRUCT`):

`modeltype:`	`'PLS'`,
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`reg:`	regression vector,
`loads:`	cell array with model loadings for each mode/dimension,
`pred:`	2 element cell array with model predictions for each input block (when `options.blockdetail='normal'` x-block predictions are not saved and this will be an empty array) and the y-block predictions.
`wts:`	double array with X-block weights,
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and
`detail:`	sub-structure with additional model details and results.

To make predictions the inputs are `x` the new predictor x-block (2-way array class "double" or "dataset"), and `model` the PLS model. The output `pred` is a structure, similar to `model`, that contains scores, predictions, etc. for the new data.

If new y-block measurements are also available then the inputs are `x` the new predictor x-block (2-way array class "double" or "dataset"), `y` the new predicted block (2-way array class "double" or "dataset"), and `model` the PLS model. The output `valid` is a structure, similar to `model`, that contains scores, predictions, and additional y-block statistics etc. for the new data.

Note: Calling `pls` with no inputs starts the graphical user interface (GUI) for this analysis method.

Options

`options` =	a structure array with the following fields:
`display:`	`[ 'off' | {'on'} ]`, governs level of display to command window,
`plots`	`[ 'none' | {'final'} ]`, governs level of plotting,
`outputversion:`	`[ 2 | {3} ]`, governs output format (see below),
`preprocessing:`	`{[] []}`, two element cell array containing preprocessing structures (see `PREPROCESS`) defining preprocessing to use on the x- and y-blocks (first and second elements respectively)
`algorithm:`	`[ 'nip' | {'sim'} | 'robustpls' ]`, PLS algorithm to use: NIPALS or SIMPLS {default}, and
`blockdetails:`	`[ {'standard'} | 'all' ]`, extent of predictions and residuals included in model, `'standard'` = only y-block, `'all'` x- and y-blocks.
`confidencelimit:`	`[ {'0.95'} ]`, confidence level for Q and T2 limits, a value of zero (0) disables calculation of confidence limits,
`roptions:`	structure of options to pass to rsimpls (robust PLS engine from the Libra Toolbox).
	`alpha:[ {0.75} ]`, (1-alpha) measures the number of outliers the algorithm should resist. Any value between 0.5 and 1 may be specified. These options are only used when algorithm is 'robustpls'.

The default options can be retreived using: `options = pls('options');`.

OUTPUTVERSION

By default (`options.outputversion = 3`) the output of the function is a standard model structure `model`. `If options.outputversion = 2`, the output format is:

```
[b,ssq,p,q,w,t,u,bin] = pls(x,y,ncomp,options)
```

where the outputs are

`b` =	matrix of regression vectors or matrices for each number of principal components up to `ncomp`,
`ssq` =	the sum of squares information,
`p` =	x-block loadings,
`q` =	y-block loadings,
`w` =	x-block weights,

`t` =	x-block scores
`u` =	y-block scores, and
`bin` =	inner relation coefficients.

Note: The regression matrices are ordered in `b` such that each *Ny* (number of y-block variables) rows correspond to the regression matrix for that particular number of principal components.

Algorithm

Note that unlike previous versions of the `PLS` function, the default algorithm (see Options, above) is the faster SIMPLS algorithm. If the alternate NIPALS algorithm is to be used, the `options.algorithm` field should be set to `'nip'`.

See Also

`analysis, crossval, modelstruct, nippls, pcr, plsda, preprocess, ridge, simpls`

plsda

Purpose

Partial least squares discriminate analysis.

Synopsis

```
model = plsda(x,y,ncomp,options)
model = plsda(x,ncomp,options)
pred  = plsda(x,model,options)
valid = plsda(x,y,model,options)
options = plsda('options')
```

Description

PLSDA is a multivariate inverse least squares discrimination method used to classify samples. The y-block in a PLSDA model indicates which samples are in the class(es) of interest through either:

(A) a column vector of class numbers indicating class asignments:

```
y = [1 1 3 2]';
```

(B) a matrix of one or more columns containing a logical zero (= not in class) or one (= in class) for each sample (row):

```
y = [1 0 0;
     1 0 0;
     0 0 1;
     0 1 0]
```

NOTE: When a vector of class numbers is used (case A, above), class zero (0) is reserved for "unknown" samples and, thus, samples of class zero are never used when calibrating a PLSDA model. The model will include predictions for these samples.

The prediction from a PLSDA model is a value of nominally zero or one. A value closer to zero indicates the new sample is NOT in the modeled class; a value of one indicates a sample is in the modeled class. In practice a threshold between zero and one is determined above which a sample is in the class and below which a sample is not in the class (See, for example, PLSDTHRES). Similarly, a probability of a sample being inside or outside the class can be calculated using DISCRIMPROB. The predicted probability of each class is included in the output model structure in the field:

```
model.details.predprobability
```

INPUTS

x = X-block (predictor block) class "double" or "dataset",

`y` =	Y-block - OPTIONAL if `x` is a dataset containing classes for sample mode (mode 1) otherwise, `y` is one of:
	(A) column vector of sample classes for each sample in x -OPTIONAL if `x` is a dataset containing classes for sample mode (mode 1)
or	(B) a logical array with 1 indicating class membership for each sample (rows) in one or more classes (columns)
`ncomp` =	the number of latent variables to be calculated (positive integer scalar).

OUTPUT

`model` =	standard model structure containing the PLSDA model (See MODELSTRUCT).
`pred` =	structure array with predictions
`valid` =	structure array with predictionsz

Note: Calling `plsda` with no inputs starts the graphical user interface (GUI) for this analysis method.

Options

`display`: ['off' | {'on'}] governs level of display to command window.

`plots`: ['none' | {'final'}] governs level of plotting.

`preprocessing`: {[] []} preprocessing structures for x and y blocks (see PREPROCESS).

`algorithm`: ['nip' | {'sim'}] PLS algorithm to use: NIPALS or SIMPLS

`blockdetails`: ['compact' | {'standard'} | 'all'] Extent of detail included in model.

'standard' keeps only y-block, 'all' keeps both x- and y- blocks

See Also

`class2logical, crossval, pls, plsdthres, simca`

plsdaroc

Purpose

Calculate and display ROC curves for PLSDA model.

Synopsis

```
roc = plsdaroc(model,ycol,options)
```

Description

ROC curves can be used to assess the specificity and sensitivity possible with different predicted y-value thresholds for a PLSDA model. Inputs are a PLSDA model `model`, an optional index into the y-columns used in the model `ycol` [default = all columns], and an options structure. Output is a dataset with the sensitivity/specificity data `roc`.

Options

`plots` :	['none' | {'final'}]	governs plotting on/off

See Also

`discrimprob`, `plsda`, `plsdthres`, `simca`

plsdthres

Purpose

Bayesian threshold determination for PLS Discriminate Analysis.

Synopsis

```
[threshold,misclassed,prob] = plsdthres(model,options)
[threshold,misclassed,prob] = plsdthres(y,ypred,options)
```

Description

PLSDTHRES uses the distribution of calibration-sample predictions obtained from a PLS model built for two or more logical classes to automatically determine a threshold value which will best split those classes with the least probability of false classifications for future predictions. It is assumed that the predicted values for each class are approximately normally distributed. The calibration can contain more than 2 classes, in which case thresholds to distinguish all classes will be determined. It is assumed that with more than 2 classes the primary misclassification threat is from the adjacent class(es).

Inputs

y =	measured Y-block values used in PLS, and
ypred =	PLS predicted Y values for calibration samples.
model =	a PLS/PLSDA model structure from which y and ypred should be obtained automatically.

Outputs

threshold =	[], vector of thresholds. If y consists of more than two classes, threshold will be a vector giving the upper bound y-value for each class.
misclassed =	[], array containing the fraction of misclassifications for each class (rows): Column 1 = false negatives and Column 2 = false positives.
prob =	lookup matrix of predicted y (column 1) vs. probability of each class (columns 2 to end).

Options

options is a structure array with the following fields:

display:	[{'on'} | 'off'], governs level of display,
plots:	['none' | 'final' | {'auto'} |], governs plotting behavior, 'auto' makes plots if no output is requested {default},
cost:	[], vector of logarithmic cost biases for each class in y, cost is used to bias against misclassification of a particular class or classes {default = [] uses all zeros i.e. equal cost}.
prior:	[], vector of prior probabilities of observing each class. If any class prior is Inf, the frequency of observation of that class in the calibration is used as its prior probability. If all priors are Inf, this has the effect of providing the fewest incorrect predictions assuming that the probability of observing a given class in future samples is similar to the frequency that class in the calibration set. {default = [] uses all ones i.e. equal priors.}

See Also

crossval, discrimprob, pls, simca

plsnipal

Purpose

Calculate single latent variables for partial least squares regression.

Synopsis

```
[p,q,w,t,u] = plsnipal(x,y)
```

Description

`PLSNIPAL` is called by the routine `pls` to calculate each latent variable in a partial least squares regression.

Inputs `x` and `y` are either the x-block and y-block for calculation of the first latent variable, or the x-block and y-block residuals for calculation of subsequent latent variables.

The outputs are `p` the x-block latent variable loadings, `q` the y-block variable loadings, `w` the x-block latent variable weights, `t` the x-block latent variable scores, and `u` the y-block latent variable scores.

See Also

`nippls, pls, analysis, simpls`

plspulsm

Purpose

Builds finite impulse response (FIR) models for multi-input single (MISO) output systems using partial least squares regression.

Synopsis

```
    b = plspulsm(u,y,n,maxlv,split,delay)
```

Description

`plspulsm` calculates a vector of FIR coefficients `b` using PLS regression. Inputs are a matrix of process input vectors `u`, and a process output vector `y`. `n` is a row vector with the number of FIR coefficents to use for each input, `maxlv` is the maximum number of latent variables to consider, `split` is the number of times the model is rebuilt and tested during cross-validation, and `delay` is a row vector containing the number of time units of delay for each input.

Note: `plspulsm` uses contiguous blocks of data for cross-validation.

Examples

```
b = plspulsm([u1 u2],y,[25 15],5,10,[0 3])
```

This system has 2 inputs as column vectors `u1` and `u2` and a single output vector `y`. The FIR model will use 25 coefficients for input variable `u1` and 15 coefficients for input variable `u2`. For this model a maximum of 5 latent variables will be considered. The cross validation split the data into 10 subsets. The number of time units of delay for the first input variable `u1` is 0 and for the second input variable `u2` it is 3.

See Also

`autocor, crosscor, fir2ss, wrtpulse`

plsrsgcv

Purpose

Generates a matrix used to calculate residuals from a single data block using partial least squares regression models with cross vaildation.

Synopsis

```
coeff = plsrsgcv(data,lv,cvit,cvnum,out)
```

Description

`coeff = plsrsgncv(data,lv,cvit,cvnum)` calculates a matrix `coeff` from a single data block data. `plsrsgncv` calculates partial least squares regression models of each variable in the matrix data using the remaining variables and cross-validation with random test data blocks. The maximum number of latent variables to consider is lv, the number of test sets is cvit, and the number of samples in each test set is cvnum. Multiplying a new data matrix by the matrix coeff yields a matrix whose values are the difference between the new data and it's prediction based on the PLS regressions created by plsrsgncv.

See Also

`plsrsgn, replace`

plsrsgn

Purpose

Generates a matrix used to calculate residuals from a single data block using partial least squares regression models.

Synopsis

```
      coeff = plsrsgn(data,lv,out)
```

Description

`coeff = plsrsgn(data,lv)` calculates a matrix `coeff` from a single data block `data`. `plsrsgn` calculates partial least squares regression models of each variable in the matrix data using the remaining variables and the number of latent variables `lv`. Multiplying a new data matrix by the matrix `coeff` yields a matrix whose values are the difference between the new data and it's prediction based on the PLS regressions created by `plsrsgn`.

See Also

`plsrsgcv, replace`

plttern

Purpose

Plots a 2D ternary diagram.

Synopsis

```
[tdata,h] = plttern(data,linestyle,x1lab,x2lab,x3lab)
```

Description

`PLTTERN` makes 2-D ternary plots of the data contained in the three column input matrix `data`. The columns of `data` correspond to concentrations (≥ 0 and real) and are normalized to fit in the range 0 to 100. Optional inputs *x1lab*, *x2lab*, *x3lab* are row vectors of text containing labels for the axes. The output `tdata` is the normalized concentration data.

See Also

`dp, ellps, hline, pan, pltternf, vline, zline`

pltternf

Purpose

Plots a 3D ternary diagram with frequency of occurrence.

Synopsis

```
tdata = plttern(data,x1lab,x2lab,x3lab);
```

Description

`PLTTERN` makes 3-D ternary plots of the data contained in the four column input matrix `data`. The first three columns of `data` correspond to concentrations (≥ 0 and real) and are normalized to fit in the range 0 to 100. The fourth column of data corresponds to the frequency of occurrence (≥ 0 and real). Optional inputs *x1lab*, *x2lab*, *x3lab* are row vectors of text containing labels for the axes. The output `tdata` is the normalized concentration data.

See Also

`dp, ellps, hline, pan, plttern, vline, zline`

polyinterp

Purpose

Polynomial interpolation, smoothing, and differentiation.

Synopsis

```
    yi = polyinterp(x,y,xi,width,order,deriv);
```

Description

Estimates (yi) which is the smoothed values of (y) at the points in the vector (x). (If the points are evenly spaced use the SAVGOL function instead.)

INPUTS:

y =	(M by N) matrix. Note that (y) is a matrix of ROW vectors to be smoothed.
x =	(1 by N) corresponding axis vector at the points at which (y) is given.

OPTIONAL INPUTS:

xi =	a vector of points to interpolate to.
width =	specifies the number of points in the filter {default = 15}.
order =	the order of the polynomial {default = 2}.
deriv =	the derivative {default = 0}.

Examples

If `y` is a 5 by 100 matrix, `x` is a 1 by 100 vector, and `xi` is a 1 by 91 vector then `polyinterp(x,y,xi,11,3,1)` gives the 5 by 91 matrix of first-derivative row vectors resulting from an 11-point cubic interpolation to the 91 points in `xi`.

See Also

`baseline, lamsel, mscorr, savgol, stdfir`

polypls

Purpose

Calculate partial least squares regression models with polynomial inner relations.

Synopsis

```
    [p,q,w,t,u,b,ssqdif] = polypls(x,y,lv,n)
```

Description

`POLYPLS` creates a partial least squares regression model with polynomial fit for the inner relation. Inputs are a matrix of predictor variables (x-block) `x`, a matrix of predicted variables (y-block) `y`, the number of latent variables `lv`, and the order of the polynomial `n`.

Outputs are `p` the x-block latent variable loadings, `q` the y-block variable loadings, `w` the x-block latent variable weights, `t` the x-block latent variable scores, `u` the y-block latent variable scores, `b` a matrix of polynomial coefficients for the inner relationship, and `ssqdif` a table of x- and y-block variance captured by the PLS model.

Use `POLYPRED` to make predictions with new data.

See Also

`lwrxy, pls, polypred`

polypred

Purpose

Make predictions for partial least squares regression models with polynomial inner relations.

Synopsis

```
    ypred = polypred(x,b,p,q,w,lv)
```

Description

`POLYPRED` uses parameters created by the routine `POLYPLS` to make predictions from a new x-block matrix of predictor variables `x`. Inputs are `b` a matrix of polynomial coefficients for the inner relationship, `p` the x-block latent variable loadings, `q` the y-block variable loadings, `w` the x-block latent variable weights, and the number of latent variables `lv`.

Note: It is important that the scaling of the new data `x` is the same as that used to create the model parameters in `POLYPLS`.

See Also

`lwrxy, polypls, pls`

preprocess

Purpose

Selection and application of preprocessing methods.

Synopsis

```
s = preprocess(s)                              %GUI preprocessing selection
s = preprocess('default','methodname')                  %Non-GUI selection
[datap,sp] = preprocess('calibrate',s,data)   %single block calibrate
[datap,sp] = preprocess('calibrate',s,xblock,yblock)     %multi-block
datap = preprocess('apply',sp,data)                     %apply to new data
data = preprocess('undo',sp,datap)                     %undo preprocessing
```

Description

PREPROCESS is a general tool to choose preprocessing steps and to perform these steps on data. See PREPROUSER for a description on how custom preprpocessing can be added to the standard proprocessings listed below. PREPROCESS has four basic command-line forms which include:

1) SELECTION OF PREPROCESSING.

The purpose of the following calls to PREPROCESS is to generate standard structure arrays that contain the desired preprocessing steps.

```
s = preprocess;
```

generates a GUI and allows the user to select preprocessing steps interactively. The output s is a standard preprocessing structure.

```
s = preprocess(s);
```

allows the user to interactively edit a previously identified preprocessing structure s. The output s is the edited preprocessing structure.

```
s = preprocess('default','methodname');
```

returns the default structure for method 'methodname'. A list of strings that can be used for 'methodname' can be viewed using the command:

```
preprocess('keywords')
```

A list of standard methods `'methodname'` follow:

`'abs'`:	takes the absolute value of the data (see `ABS`),
`'autoscale'`:	centers columns to zero mean and scales to unit variance (see `AUTO`),
`'detrend'`:	remove a linear trend (see `BASELINE`),
`'gls weighting'`:	generalized least squares weighting (see `GLSW`),
`'groupscale'`:	group/block scaling (see `GSCALE`),
`'mean center'`:	center columns to have zero mean (see `MNCN`),
`'msc (mean)'`:	multiplicative scatter correction with offset, the mean is the reference spectrum (see `MSCORR`),
`'median center'`:	center columns to have zero median (see `MEDIAN`),
`'normalize'`:	normalization of the rows (see `NORMALIZ`),
`'osc'`:	orthogonal signal correction (see `OSCCALC` and `OSCAPP`),
`'sg'`:	Savitsky-Golay smoothing and deriviatives (see `SAVGOL`), and
`'snv'`:	standard normal deviate (autoscale the rows, see `SNV`).

The output is a standard preprocessing structure array `s` where each method to apply is a separate record.

2) CALIBRATE.

The objective of the following calls to `PREPROCESS` is to estimate preprocessing parameters, if any, from a calibration data set and perform preprocessing on the calibration data set. The I/O format is:

```
[datap,sp] = preprocess('calibrate',s,data);
```

The inputs are `s` a standard preprocessing structure and `data` the calibration data. The preprocessed data is returned in `datap`, and preprocessing parameters are returned in a modified preprocessing structure `sp`. Note that `sp` is used as an input with the `'apply'` and `'undo'` commands described below.

Short cuts for each method can also be used. Examples for `'mean center'` and `'autoscale'` are

```
[datap,sp] = preprocess('calibrate','mean center',data);
[datap,sp] = preprocess('calibrate','autoscale',data);
```

Preprocessing for some multi-block methods require that the y-block be passed also. The I/O format in these cases is:

```
[datap,sp] = preprocess('calibrate',s,xblock,yblock);
```

Preprocessing `'methodname'` that require a y-block are:

```
'osc'
'gls weighting'
```

3) APPLY.

The objective of the following call to PREPROCESS

```
      datap = preprocess('apply',sp,data)
```

is to apply the calibrated preprocessing in `sp` to new data. Inputs are `sp` the modified preprocessing structure (See 2 above) and the data, `data`, to apply the preprocessing to. The output is preprocessed data `datap` that is class “dataset”.

4) UNDO.

The inverse of applying preprocessing is perfromed in the following call to PREPROCESS

```
      data = preprocess('undo',sp,datap);
```

Inputs are `sp` the modified preprocessing structure (See 2 above) and the data, `datap`, (class “double” or “dataset”) from which the preprocessing is removed. Note that for some preprocessing methods an inverse does not exist or has not been defined and an 'undo' call will cause an error to occur. For example, `'osc'` and `'sg'`. One reason for not defining an inverse, or undo, is because it would require a significant amount of memory storage when data sets get large.

See Also

`crossval, pca, pcr, pls, preprouser`

preprouser

Purpose

User defined items for preprocess catalog.

Synopsis

```
preprouser(fig)
```

Description

Each method available in the `preprocess` function has an associated `'methodname'` such as those listed in the help for `preprocess`. Each method is defined using a preprocessing structure that contains all the necessary information to perform calculations for that method. The standard methods are defined in the `preprocatalog` file, which should not be edited by the user. Additional user-defined methods can be defined in the `preprouser` file and the following text describes how the user to add custom preprocessing methods. A few example methods already exist in the `preprouser` file to guide the user.

To add a custom user-defined preprocessing method, the user must 1) open the `PREPROUSER.M` file, 2) edit the file to create a structure with the fields described below, 3) after defining the structure add the line `preprocess('addtocatalog',fig,usermethod)`, and 4) save and close the `PREPROUSER.M` file.

The line added in Step 3

```
preprocess('addtocatalog',fig,usermethod)
```

makes the new custom method available to `PREPROCESS`. The input `usermethod` is the preprocessing structure containing the user-defined method, and `fig` is a figure handle passed to `preprouser` by `preprocess`.

The methods defined in the `preprocatalog` and `preprouser` files are available to all functions making use of the `preprocess` function.

The fields in a preprocessing structure are listed here. Detailed descriptions and examples follow this list.

`description:`	text string containing a description for the method,
`calibrate:`	cell containing the line(s) of code to execute during a calibration operation (see command-line form 2 of `PREPROCESS`),
`apply:`	cell containing the line(s) of code to execute during an apply operation (see command-line form 3 of `PREPROCESS`),
`undo:`	cell containing the line(s) of code to execute during an undo operation (see command-line form 4 of `PREPROCESS`),
`out:`	cell used to hold calibration-phase results for use in apply or undo (these are the parameters estimated from the calibration data and used to preprocess new data),
`settingsgui:`	text string containing the function name of a method-specific GUI to invoke when the **Settings** button is pressed in the preprocessing GUI,
`settingsonadd:`	`[ 0 | {1} ]`, boolean: 1 = indicates that the settings GUI should be automatically brought up when method is "added" in the preprocessing GUI,
`usesdataset:`	`[ {0} | 1 ]`, boolean: indicates if this method should be passed a dataset object (1) or a an array (0) (e.g. class “double” or “uint8”),
`caloutputs:`	integer: number of expected items in field `out` after calibration has been performed. This field is set by the user to tell `PREPROCESS` what the length of the cell in field `out` will be after calibration,
`keyword:`	text string containing the `'methodname'`, this string is used in the call to `PREPROCESS` so that it will return the custom preprocessing structure (see command-line form 1 of `PREPROCESS`), and
`userdata:`	user-defined variable often used to store method options.

Detailed descriptions and examples for each field follow:

DESCRIPTION:

The description is a short (1-2 word) text string containing a description for the preprocessing method. The string will be displayed in the GUI and can also be used as a string keyword (see also `keyword`) to refer to this method.

Example:

```
pp.description = 'Mean Center';
```

CALIBRATE, APPLY, UNDO:

Each of these "command" fields contains a single cell consisting of a command string to be executed by PREPROCESS when performing calibration, apply, or undo operations (see command-line forms 2, 3, and 4 of PREPROCESS). Calibrate actions operate on original calibration data with the output parameters stored in the out field, whereas apply actions operate on new data using parameters stored in the out field as input(s). For methods which act on a single sample at a time, the calibrate and apply operations are often identical (for example, see the normalize example below). The undo action uses parameters stored in the out field as input(s) to remove preprocessing from previously preprocessed data. However, the undo action may be undefined for certain methods. If this is the case, the undo field should be an empty cell.

To assure that all samples (rows) in the data have been appropriately preprocessed, an apply command is automatically performed following a calibrate call. Note that excluded variables are replaced with NaN.

The command strings should be one or more valid MATLAB commands, each separated by a semicolon ';' (e.g. see EVAL). Each command will be executed inside the PREPROCESS environment in which the following variables are available:

data:	The data field contains the data on which to operate and in which to return modified results. If the field usesdataset is 1 (one) then data will be a DataSet object. In this case, it is expected that the function will calibrate using only included rows but apply and undo the preprocessing to all rows. If the field usesdataset is 0 (zero) then data will be an array (e.g. class "double"). In this case, the function will calibrate using all rows and columns and will apply and undo the preprocessing to all rows and columns.
out:	Contents of the preprocessing structure field out (described below). Any changes will be stored in the preprocessing structure for use in subsequent apply and undo commands.
userdata:	Contents of the preprocessing structure field userdata (described below). Any changes will be stored in the preprocessing structure for later retrieval.

Several variables are available for use during command operations (calibarate, apply, and undo). However, these variables should not be changed by the commands and are considered "read-only".

`include:`	When the field `usesdataset = 1`, the data is passed as a dataset object. In this case, `include` contains the contents of the original dataset object's `includ` field.
`otherdata:`	Cell array of any inputs to `PREPROCESS` which followed the data in the input list. For example, it is used by PLS_Toolbox regression functions to pass the y-block for use in methods which require that information.
`originaldata:`	A dataset object which contains the original data unmodified by any preprocessing steps. For example, `originaldata` can be used to retrieve axis scale or class information even when `usesdataset` is 0 (zero).

Examples:

The following `calibrate` field performs mean-centering on `data`, returning both the mean-centered data as well as the mean values which are stored in `out{1}`:

```
pp.calibrate   = { '[data,out{1}] = mncn(data);' };
```

The following `apply` and `undo` fields use the `scale` and `rescale` functions to apply and undo the previously determined mean values (stored by the calibrate operation in `out{1}`) with new data:

```
pp.apply       = { 'data = scale(data,out{1});' };
pp.undo        = { 'data = rescale(data,out{1});' };
```

OUT:

The out field is a cell array that contains the output parameters returned during the calibration operation. For example, if the following commands are run

```
load wine
s = preprocess('default','autoscale');
[dp,sp] = preprocess('calibrate',s,wine);
```

then the out field of sp is a 1 by 2 cell array with the first cell, out{1}, containing the means of the variables in the dataset wine, and the second cell, out{2}, contains the standard deviations. These parameters are used in subsequent apply and undo commands. See the related field caloutputs. Prior to the calibration operation both the out and caloutputs fields are empty.

SETTINGSGUI:

The name of a graphical user interface (GUI) function that allows the user to set options for this method. The function is expected to take as its only input a standard preprocessing structure from which it should take the current settings. The function should output the same preprocessing structure modified to meet the user's specification. Typically, these changes are made to the userdata field and the commands in the calibrate, apply and undo fields use that field’s contents as input options.

The design of GUIs for selection of options is beyond the scope of this document and the user is directed to the following example files, both of which use GUIs to modify the userdata field of a preprocessing structure: autoset.m savgolset.m .

Example:

```
pp.settingsgui   = 'autoset';
```

SETTINGSONADD:

The settingsonadd field contains a boolean (1=true, 0=false) value. If it is 1=true, then when the user adds the method in the PREPROCESS GUI, the method's settingsgui will be automatically invoked. If a method requires the user to make a selection of options, settingsonadd=1 will guarantee that the user has an opportunity to modify the options or at least choose the default settings.

Example:

```
pp.settingsonadd   = 1;
```

USESDATASET:

The `usesdataset` field contains a boolean (1=true, 0=false) value.

If it is 1=true, the preprocessing method is capable of handling dataset objects and `PREPROCESS` will pass the data as a dataset. It is the responsibility of the function(s) called by the method to appropriately handle the dataset's `includ` field.

If it is 0=false, the preprocssing method expects standard MATLAB classes (double, uint8, etc). `PREPROCESS`, which uses a dataset object internally to hold the data, will extract data from the dataset ojbect prior to calling this method. It will then reinsert the preprocessed data back into the dataset object after the method has been invoked.

Although excluded columns are never extracted and excluded rows are not extracted when performing `calibration` operations, excluded rows are passed when performing `apply` and `undo` operations.

Example:

```
pp.usesdataset   = 0;
```

CALOUTPUTS:

For functions which require a calibrate operation prior to an apply or undo (see the fields: `calibrate` and `out`), this field indicates how many values are expected in the `out` field. For example, in the case of mean centering the mean values stored in the field `out` are required to apply or undo the operation. Initially, `out` is an empty cell ({}). Following the calibration operation for mean centering, it becomes a single-item cell (length of one). For other calibration operations `out` may be a cell of length greater than one.

By examining this cell's length, `PREPROCESS` can determine if a preprocessing structure has already been calibrated and contains the necessary information. The `caloutputs` field, when greater than zero, indicates to `PREPROCESS` that it should test the `out` field prior to attempting an apply or undo.

Example: in the case of mean-centering, the length of `out` should be 1 (one) after calibration.

```
pp.caloutputs    = 1;
```

KEYWORD:

The field `keyword` is a string that can be used to retrieve the default preprocessing structure for this method. When retrieving a structure by keyword, `PREPROCESS` ignores any spaces and is case-insensitive. The `keyword` field (or the `description` string, discussed above) can be used in place of any preprocessing structure in `calibrate` and `default` calls to preprocess:

```
pp = preprocess('default','meancenter');
```

Example:

```
pp.keyword     = 'mncn';
```

USERDATA:

The field userdata contains additional user-defined data that can be changed during a calibration operation and retrieved for use in apply and undo operations. This field is often used to hold options for the preprocessing method which are then used by the commands in the calibrate, apply, and undo fields.

Example: in SAVGOL several input variables are defined with various method options, then they are assembled into a vector in userdata:

```
      pp.userdata    = [windowsize order derivative];
```

Examples

The following is the preprocessing structure used for sample normalization (see NORMALIZ). The calibrate and apply commands are identical and there is no information that is stored during the calibration phase, thus caloutputs is zero. There is no undo defined for this operation (this is because the normalization information required to undo the action is not being stored anywhere). The norm type (e.g. a 2-norm) of the normalization is set in userdata and is used in both calibrate and apply steps.

```
      pp.description = 'Normalize';
      pp.calibrate   = {'data = normaliz(data,0,userdata(1));'};
      pp.apply       = {'data = normaliz(data,0,userdata(1));'};
      pp.undo        = {};
      pp.out         = {};
      pp.settingsgui   = 'normset';
      pp.settingsonadd = 0;
      pp.usesdataset   = 0;
      pp.caloutputs    = 0;
      pp.keyword       = 'Normalize';
      pp.userdata      = 2;
```

The following is the preprocessing structure used for Savitsky-Golay smoothing and derivatives (see SAVGOL). In many ways this structure is similar to the normalize structure except that SAVGOL takes a dataset object as input and, thus, usesdataset is set to 1. Also note that because of the various settings required by savgol, this method uses of the settingsonadd feature to bring up the settings GUI as soon as the method is added.

```
pp.description = 'SG Smooth/Derivative';
pp.calibrate =
   {'data=savgol(data,userdata(1),userdata(2),userdata(3));'};
pp.apply     =
   {'data=savgol(data,userdata(1),userdata(2),userdata(3));'};
pp.undo      = {};
pp.out       = {};
pp.settingsgui   = 'savgolset';
pp.settingsonadd = 1;
pp.usesdataset   = 1;
pp.caloutputs    = 0;
pp.keyword       = 'sg';
pp.userdata      = [ 15 2 0 ];
```

The following example creates a preprocessing structure to invoke multiplicative scatter correction (MSC, see MSCORR) using the mean of the calibration data as the target spectrum. The calibrate cell here contains two separate operations. The first calculates the mean spectrum and the second performs the MSC. The third input to the MSCORR function is a flag indicating whether an offset should also be removed. This flag is stored in the userdata field so that the settingsgui (mscorrset) can change the value easily. Note that there is no undo defined for this function.

```
pp.description = 'MSC (mean)';
pp.calibrate   = { 'out{1}=mean(data);
   data=mscorr(data,out{1},userdata);' };
pp.apply       = { 'data = mscorr(data,out{1});' };
pp.undo        = {};
pp.out         = {};
pp.settingsgui = 'mscorrset';
pp.settingsonadd = 0;
pp.usesdataset   = 0;
pp.caloutputs    = 1;
pp.keyword       = 'MSC (mean)';
pp.userdata      = 1;
```

See Also

preprocess

purity

Purpose

Calculation of pure variables.

Synopsis

```
[purint,purspec] = purity(data,ncomp,options);
[model]          = purity(data,ncomp);
[purint,purspec] = purity(data,ncomp,model);
[model]          = purity(data,model);
```

Description

PURITY calculates pure variables and resolves data into ncomp spectra of the pure components (purspec) and their contributions (purint). For more information about the algorithm see PURITYENGINE. Data can be a matrix with the data or a dataset object.

The output arguments `purity_values` contains the purity values for all the variables and can be plotted as the "purity spectrum". The argument `length_values` contains the `purity_values` multiplied by the length of the variables. This results in a "length spectrum" that is easier to relate to the original data than the purity spectrum

The optional input `options` is a structure with the following fields

display:	['off'|{'on'}] display to command window.
plot:	['off'|{'on'}] plotting of result.
axistype:	{2x1} [char] Mode 1: [{continuous}|'discrete'|'bar'] Mode 2: [{continuous}|'discrete'|'bar'] defines plots. if emtpy the values of the (future) DSO field will be used in case they are not defined, the 'continuous' defaults will be used.
select:	[{[]},[1 2]] if empty, pure rows/columns will be selected from last slab, otherwise, the numbers identify from which slab(s) the pure rows/columns are selected.
offset:	[3 10] default noise correction factor for the two slabs.
offset_row2col:	3 scalar value row2col offset, default is offset(1).
mode:	['rows',{'cols'},'row2col'] determines if pure rows, cols are selected. row2col 2 is row-to-column solution.
algorithm:	'purityengine' defines algorithm used.
interactive:	['on',{'off'}, defines interactivity; 'on', 'cursor','inactivate','reactivate'] 'reactivate', 'cursor', 'inactivate', 'reactivate' are used for higher level calls for interactivity,'off' is used for demos and command mode applications.
resolve:	['off'|{'on'}] indicates if the resolved results are required or not.

Examples

Resolving 4 components in a data set:

```
    [purint,purspec]=purity(data,4)
```

Algorithm

The core algorithm is the function `purityengine`.

See Also

`purityengine`

purityengine

Purpose

Calculation of pure variables.

Synopsis

```
[purity_index,purity_values,length_values]=purityengine(data,...
base,offset)
```

Description

`PURITYENGINE` calculates the column index (`purity_index`) of the variable in data that has the largest angle with respect to `base`. For the first pure variable `base` should be empty: the program then substitutes a vector of ones for `base.` `base` generally contains previously determined pure variables. The argument `offset` gives a lower weight to variables with low values. Its value is based on a percentage of the maximum value of the mean of data. A typical value is 3.

The output arguments `purity_values` contains the purity values for all the variables and can be plotted as the "purity spectrum". The argument `length_values` contains the `purity_values` multiplied by the length of the variables. This results in a "length spectrum" that is easier to relate to the original data than the purity spectrum

Examples

Determination of three pure variables of a matrix data for an offset of 3

```
[purity_index,purity_values,length_values]=purityengine(data,[],3);
purity_array=[purity_index];
[purity_index,purity_values,length_values]=purityengine(data,...
data(:,purity_array),3);
purity_array=[ purity_array purity_index];
[purity_index,purity_values,length_values]=purityengine(data,...
data(:,purity_array),3);
purity_array=[ purity_array purity_index];
```

The indices of the three pure variables are in purity_array. A plot of purity_values and length_values shows the successive stages of the pure variable extraction.

Algorithm

The calculations are based on the MATLAB function subspace. The angle of every variable in the data is calculated with respect to the base: `subspace(base,data(:,i))`

See Also

`purity`

qconcalc

Purpose

Calculate Q residuals contributions for predictions on a model.

Synopsis

```
qcon = qconcalc(newx,model)
qcon = qconcalc(model);   %requires that model contains residuals
```

Description

Inputs are the new data `newx` and the 2-way PCA or regression model for which Q contributions should be calculated `model`.

If the model was created using the "blockdetails = 'all'" option in PLS or PCA (or whatever function was used to create the model), then `newx` can be omitted to retrieve the Q contributions for the calibration data. Note that this option is not the default so it is unlikely this call will work unless you have specifically created the model with the appropriate call.

See Also

`datahat, pca, pcr, pls, tconcalc`

querydb

Purpose

Executes a query on a database defined by connection string.

Synopsis

```
out = querydb(connstr,sqlstr,options);
```

Description

This function is unsupported and is meant as a "simple" database connection tool. For more sophisticated connection tools and full support please see the Matlab Database Toolbox.

JDBC connections require that the jdbc driver ".jar" file be added to the Matlab java classpath. See the documentation for the Matlab commands 'javaaddpath' and 'javaclasspath' for more information. For example, using the MySQL Connector/J 3.1 driver you'll need to add the "mysql-connector-java-3.1.12-bin.jar" file to your java class path.

INPUTS

`connstr` :	A connection string or a structure created using builddbstr. See BUILDDBSTR for more information.
`sqlstr` :	A SQL statement to be executed on the connection. The SQL statement must be of proper syntax or it will fail. Default behavior is geared toward SELECT statements that return values. If attempting to execute a SQL command that doesn't return a value (e.g., CREATE TABLE) set the 'rtype' option to 'none'. NOTE: Use a seperate program like Microsoft Access to formulate the SQL statement. Access queries can require some small changes in syntax.

OUTPUTS

`out` :	DataSet Object, Cell Array, or Scalar depending on 'rtype'.

Options

`rtype` :	[{'dso'} | 'cell' | 'none'] Return type, default is return SQL recordset as a DataSet Object using parsemixed.m to parse data in. If 'cell' then a cell array is returned with all values. If 'insert' then function will execute an "INSERT" type query and attempt to return the Auto Number ID (as a scalar) of the row created. If 'none' function will execute query and return an empty.
`varlabels` :	[{'none'} | 'fieldnames'] Defines what should be used as variable labels on output DataSet Object (only used when rtype is 'dso'). 'fieldnames' uses the SQL field names for variable labels.

conntype : ['jdbc' | {'odbc'}] Determines type of connection. ODBC uses a Windows ADO with Matlab (descibed above). JDBC connections only work when jdbc class files are on static java path.

getaccesstables : ['on' | {'off'}] Short circuit to retrieve list of tables in Access database, similar to SHOW TABLES query in MySQL. Input 'sqlstr' will not be called when option is 'on'.

Examples

Assuming there is a connection string named ‘mydbconn’ already created using the `builddbstr` command. To return a DSO:

```
>> sqlstr = ‘SELECT * FROM myTable’;
>> mydso = querydb(mydbconn,sqlstr);
```

To return a cell array:

```
>> opts = querydb(‘options’);
>> opts.rtype = ‘cell’;
>> mycell = querydb(mydbconn,sqlstr,opts);
```

See Also

`builddbstr, parsemixed`

regcon

Purpose

Converts a regression model to y = ax + b form.

Synopsis

```
[a,b] = regcon(mod)
[a,b] = regcon(regv,xmn,ymn)
[a,b] = regcon(regv,xmn,ymn,xst,yst)
```

Description

`REGCON` can be used to convert a model `mod` generated by the `PCR`, `PLS`, or `ANALYSIS` functions. The outputs are the regression coefficients `a` and the intercept `b` such that y = ax + b. In this case the I/O syntax is:

```
[a,b] = regcon(mod)
```

Notes:

(1) REGCON can will convert a regression model which uses Mean Centering, Autoscaling, or None as the preprocessing. Any other preprocessing will be rejected and cause an error.

(2) If the model was built with some variables excluded, REGCON will infill with zeros as appropriate so that the output can be used on the original X-block with all variables present.

`REGCON` can also be used to convert the individual parts of a regression model, including the column vector of regression coefficients `regv`, predictor variable means `xmn`, predicted variable means `ymn`, predictor variable scaling `xst`, and predicted variable scaling `yst`. If `xmn` or `ymn` is not supplied or is set equal to 0 or [], then it is assumed to be zero (*i.e.* no centering was used in the model). If `xst` or `yst` is not supplied or is set equal to 0 or [], then it is assumed to be one (*i.e.* no scaling was used in the model). In this case the I/O syntax is:

```
[a,b] = regcon(regv,xmn,ymn,xst,yst)
```

Examples

`[a,b] = regcon(mod);`	using REGRESSION model
`[a,b] = regcon(regv,xmn,ymn);`	mean centered only
`[a,b] = regcon(regv,xmn,ymn,xst,yst);`	mean centered and scaled
`[a,b] = regcon(regv,xmn,ymn,[],yst);`	x data centered but not scaled
`[a,b] = regcon(regv,0,0,xst,yst);`	x and y scaled by not centered

See Also

`analysis, auto, mncn, modlpred, modlrder, pcr, pls, ridge`

registerspec

Purpose

Shift spectra based on expected peak locations.

Synopsis

```
[data_i,axaxis,foundat] = registerspec(data,xaxis,peaks,options)
peaks = registerspec(data,xaxis,options)
```

Description

REGISTERSPEC is used to correct spectra for shifts in x-axis (e.g. wavelength or frequency) registration. The alignment is based on either a polynomial or constrained-spline fit of reference peaks' observed position to their expected position. In contrast to other alignment methods (e.g. piecewise direct standardization or dynamic time warping), REGISTERSPEC may be more useful when 1) x-axis shifts are small and potentially non-linear, 2) only a few consistant reference peaks exist, and/or 3) when some of the spectral bands are expected to undergo significant shape changes in the normal range of observations.

There are two modes used to call REGISTERSPEC. The first mode is used to align new spectra given a set of reference peaks. The second mode is used to help identify peaks in a calibration set that might be useful as reference peaks:

Spectral Alignment:

```
[data_i,axaxis,foundat] = registerspec(data,xaxis,peaks,options)
```

When aligning new spectra to known reference peak positions, REGISTERSPEC takes as input a matrix or DataSet object containing spectra to be aligned, data, an x-axis reference for those spectra, xaxis, and a vector containing the expected positions of previously-identified reference peaks, peaks. Outputs are the spectra aligned to the reference peaks, data_i, the x-axis scale for those spectra, axaxis (generally the same as xaxis, except as discussed below) and an array, foundat, of the observed shifts for each reference peak (columns) and each spectra in data (rows).

If the input xaxis is omitted and data is a DataSet object containing axisscale information for the variables (data.axisscale{2}), this axis will be used as xaxis. Otherwise, a lack of input for xaxis will cause REGISTERSPEC to assume that the spectral channels are evenly spaced starting from a value of 1.

In addition to correcting peak shifts, the sampling rate of the output spectra can be increased through cubic-spline interpolation. The options.interpolate setting (see below) controls the sampling rate of the output spectra. Generally the output axaxis is the same as the input xaxis. However, when interpolation is performed, the output axaxis will contain the x-axis values that correspond to the interpolated spectra in the input data.

Various options can be set through the optional input structure options. These are described in detail below. It is recommended that options.order, options.maxshift, and options.window be reviewed prior to use. Note that options.maxshift and options.window are input in absolute x-axis units and the desired input values will vary depending on the original x-axis interval (i.e. data-point spacing) and expected peak widths. In addition, the order of polynomial used to correct for shifts should be reviewed (options.order). It is generally best to keep the order as low as possible (<3 is preferable) to avoid over-fitting and unusual shifting at the ends of the spectrum.

Reference Peak Identification:

```
peaks = registerspec(data,xaxis,options)
```

When using `REGISTERSPEC` to identify reference peaks, the spectral data and x-axis information is supplied alone without a list of reference peaks. In this mode, a set of spectra (often those used for a multivariate calibration model) are searched for peaks which show relatively consistant maxima. The algorithm first locates peaks on the mean spectrum by automatically identifying positions that show a clear inflection point as a peak maximum. Peaks located in the first step are then tested on the individual spectra and must meet the following criteria:

(1) For all obesrved spectra, the peak must contain a maximum value (i.e. the peak cannot be a shoulder without an inflection point).

(2) For all observed spectra, the peak must not shift more than the value set by `options.maxshift` (default is 4 x-axis units) from the peak's position in the mean spectrum.

The output is a list of potential reference peaks. These should be examined carefully. There is no constraint that a peak have a signal to noise or signal to background level above that which permits the maximum to be found. Thus, very low-signal peaks could be returned as stable but not be observable in future spectra. Additionally, it may be useful to take the list of reference peaks and execute `REGISTERSPEC` on the calibration data itself to examine the extent and nature of shifting on the calibration data itself.

Often this routine is used as a preprocessing step for a calibration model. In these cases, `REGISTERSPEC` should be run both on the original calibration data (first to locate reference bands, then a second time to subject the calibration data to the shift algorithm), as well as on future data prior to prediction.

INPUTS

`data`	=	matrix or DataSet of spectra
`xaxis`	=	optional frequencies or energies associated with each variable in data {optional; default = use DataSet values, otherwise use 1:n}
`peaks`	=	expected locations of peaks to use for shifting. If omitted, 'findpeaks' mode will be invoked and stable peaks will be found in the data (see below).

OUTPUTS

`data_i` =	shifted, interpolated data
`axaxis` =	interpolated xaxis (will be equal to xaxis if no interpolation is requested)
`foundat` =	matrix of peak shifts found for each peak (columns) in each spectrum (rows)
`peaks` =	(only for 'findpeaks' mode) Locations of found peaks in xaxis units.

Notes: If input (`peaks`) is omitted, the algorithm identifies peaks in the mean spectrum by setting peaks at every variable and allowing these to drift to the nearest maximum. It then locates the same peaks in each of the individual spectra and keeps only those peaks which could be located in all spectra with less shift than specified in options.maxshift.

Examples

To locate stable peaks in (unshifted) calibration data
```
      peaks = registerspec(calibrationdata);
```

To correct x-axis shift in new data using previously identified peaks
```
      newdata_unshifted = registerspec(newdata,peaks);
```

Options

`display`:	`[{'on'}	'off']` governs command-line output
`plots`:	`[{'none'}	'fit'	'final']` governs plotting options
`nopeaks`:	`['none'	{'warning'}	'error']` governs behavior when none of the reference peaks can be located.
`shiftby`:	`[{-0.1}]` minimum shifting interval. A positive value is interpreted as being in absolute xaxis units and a negative value as relative to the smallest xaxis interval.
`interpolate`:	`[{[]}]` interpolation interval for output spectra. Empty [] does no interpolation. A positive value is interpreted as being in absolute xaxis units and a negative value as relative to the smallest xaxis interval.
`maxshift`:	`[in xaxis units, {4}]` maximum allowed peak shift (peaks which require more shift than this will NOT be used for xaxis correction).
`window`:	`[in xaxis units, {[]}]` size of window to search for each peak. Empty [] uses automatic window based on maxshift.
`order`:	order of polynomial (only used for polynomial algorithms)
`algorithm`:	xaxis correction algorithm. One of:

`'pchip'`:	constrained picewise spline (well behaved)
`'poly'`:	{default} standard polynomial fit to found peaks
`'iterativepoly'`:	iterative polynomial fitting (order increased in each cycle - works better for badly shifted spectra)
`'findpeaks'`:	locate non-moving peaks in whole dataset. Triggered by omission of the (peaks) input.
`smoothing`:	`['off'	{'on'}]` governs use of smoothing algorithm during peak location. If 'on' each sub-window is smoothed prior to locating maximum in window.
`smoothinfo`:	`[width order]` smoothing parameters to be passed to smoothing function (savgol) if enabled by smoothing option above. `width` is width of window in number of variables, `order` is order of polynomial. Default is width of 5 and order 2: [5 2].

Algorithm

Correction of x-axis shift in a given spectrum is achieved by first locating the maximum value nearest to the expected peak locations using localized spline interpolation nearby the expected location (within `options.maxshift` axis units from the expected position). The observed peak locations are then compared to the expected peak locations and the difference is fit with the desired function (see options). The difference is finally removed from the spectrum using interpolation back to the expected frequency or wavelength values.

Automatic peak location is achieved by attempting to locate peaks across the entire spectrum, then searching those peaks which show less than `options.maxshift` change in position throughout the set of calibration spectra.

See Also

`alignmat, coadd, deresolv, stdfir, stdgen`

replace

Purpose

Replace variables based on principal component analysis (PCA) or partial least squares (PLS) regression models.

Synopsis

```
rm = replace(model,vars)
[rm,repdata] = replace(model,vars,data)
repdata = replace(model,data)
```

Description

`REPLACE` replaces variables from data matrices with values most consistent with the given PCA or PLS model. Input `model` can be any of the following:

1) a standard model structure generated by the PCA or PLS functions or the Anlysis GUI

2) a set of loading column vectors (*e.g.*, `loads` returned by the `pca` routine, or `model.loads{2}` if the output is a model structure)

3) the PCA residual generating matrix (`I-loads*loads´`), or

4) the PLS residuals generating matrix `coeff` returned by the `plsrsgn` routine.

Optional input `vars` is a row vector containing the indices of the variables (columns) to be replaced. If omitted, the input data is searched for non-finite values (NaN, Inf) and these values are replaced.

When `vars` in input, the outputs are the replacement matrix `rm` and the replaced data (if data was provided), `repdata`. Multiplication of a data matrix `xnew` by `rm` will replace variables with values most consistent with the given PCA or PLS model. If `vars` was not supplied, only `repdata` is output.

Examples

A PCA model was created on a data matrix `xold` giving a model structure `model`. The loadings, a set of loadings column vectors, were extracted to a variable `loads` using `loads = model.loads{2};`. It was found that the sensor measuring variable 9 has gone "bad" and we would like to replace it in the new data matrix `xnew`. A replacement matrix `rm` is first created using `replace`.

```
rm = replace(loads,9);
```

The new data matrix with variable 9 replaced `rxnew` is then calculated by multiplying `xnew` by `rm`.

```
rxnew = xnew*rm;
```

See Also

mdcheck, pca, plsrsgcv, plsrsgn

rescale

Purpose

Scales data back to original scaling.

Synopsis

```
rx = rescale(x,means,stds,options)
```

Description

Rescales a matrix `x` using the means `means` and standard deviation `stds` vectors specified. An optional input `options` is an options structure with the field:

`rx = rescale(x,means)` rescales a mean centered matrix `x` using a vector of `means`.

`rx = rescale(x,means,`*stds*`)` rescales an autoscaled matrix `x` using a vector of `means`, and vector of standard deviations `stds`.

Options

`stdthreshold`:	`[ 0 ]` scalar value or vector of standard deviation threshold values. If a standard deviation is below its corresponding threshold value, the threshold value will be used in lieu of the actual value. A scalar value is used as a threshold for all variables.

See Also

`auto, medcn, mncn, npreprocess, preprocess, scale`

residuallimit

Purpose

Esitmates confidence limits for sum squared residuals.

Synopsis

```
[rescl,s] = residuallimit(residuals,cl,options)
[rescl,s] = residuallimit(model,cl,options)
rescl     = residuallimit(s,cl,options)
options = residuallimit('options');.
```

Description

Inputs are a matrix of residuals, `residuals`, and a frational confidence limit, `cl`, where `0<cl<1` {default = 0.95}. For example, for a PCA model $\mathbf{X} = \mathbf{TP}^T + \mathbf{E}$, the input `residuals` is the matrix **E** which can be calculated using the datahat function or a standard model structure (model). Optional input *options* is discussed below. To calculate multiple confidence limits, `cl` can be a vector of fractional confidence limits.

Two alternate methods of calling `RESIDUALLIMIT` are:

(a) When using the Jackson-Mudholkar method (see options) the eigenvalues of the residuals, `s`, can be passed in place of `residuals`. This is typically faster than passing the residuals themselves.

(b) A standard model structure, `model`, can be passed in place of `residuals`. In this case, `RESIDUALLIMIT` will locate valid residual information within the model and use that to calculate the limit.

The output is the estimated residual limit `rescl`. When using the Jackson-Mudholkar algorithm, an additional output, s, is also returned containing eigenvalues of **E**. To improve speed, `s` can be used in place of `residuals` in subsequent calls to `RESIDUALLIMIT` for the same data.

See Jackson (1991) for the details of the calculation.

Options

options =	a structure array with the following fields:
`algorithm:`	`[{'jm'}	'chi2'	'auto']`, governs choice of algorithm: `'jm'`, uses Jackson-Mudholkar method (slower, more robust), `'chi2'`, uses chi-squared moment method (faster, less robust with outliers), and `'auto'` automatically selects based on data size (<300 rows or columns, use 'jm', otherwise, use 'chi2')

The default options can be retreived using: `options = residuallimit('options');`.

Examples

The following example will calculate the 95Found residuals confidence limit for a model, `model`, using the residual eigenvalues stored in the model:

```
      rescl = residuallimit(model,0.95);
```

The following example will also calculate the 95Found residuals confidence limit for a model, `model`, but by using the actual residuals calculated from the calibration data, `data`, using the `datahat` function:

```
      [xhat,residuals] = datahat(model,data);
      rescl = residuallimit(residuals,0.95);
```

See Also

`chilimit, analysis, datahat, pca`

reversebytes

Purpose

Flips order of bytes in a word.

Synopsis

```
res = reversebytes(y,totalbytes,base)
```

Description

Generalized reversal of bytes. Inputs are `y`, the value(s) to operate on, the total number of bytes to swap `totalbytes` {default = 2} in each word, and the number base to work in `base` {default = 2^8 = 256 = 1 hex byte}. Note that the default is to swap 2 hex bytes in a 16 bit number.

Examples

To swap 4 BYTES in a 32 bit number:

```
reversebytes(y,4)
```

To swap 2 WORDS in a 32 bit number:

```
reversebytes(y,2,2^16)
```

reviewmodel

Purpose

Examines a standard model structure for typical problems.

Synopsis

```
[warn,color,warningid] = reviewmodel(model,single)
```

Description

Given a standard PLS_Toolbox model structure, REVIEWMODEL examines the numerical and build information and returns textual warnings to advise the user of possible issues.

INPUTS:

model :	a standard model structure (or the handle to an Analysis GUI).
single :	a flag where a value of 1 (one) indicates that only the single most urgent issue should be returned.

OUTPUTS:

issues :	A structure array containing one or more issues identified in the model. The structure contains the following fields and may contain one or more records, or may be empty if no issues were identified.
	issue - the text describing the issue.
	color - a "color code" identifying the sevrity of the issue.
	issueid - a unique ID identifying the issue.

If no outputs are requested, any issues are simply displayed in the Command Window.

See Also

ridge

Purpose

Ridge regression by Hoerl-Kennard-Baldwin.

Synopsis

```
[b,theta] = ridge(x,y,thetamax,divs,tf)
```

Description

`RIDGE` creates a ridge regression model for a matrix of predictor variables (x-block) `x`, and a vector of predicted variable (y-block) `y`. The maximum value of the ridge parameter to consider is given by `thetamax` (`thetamax` > 0). `divs` specifies the number of values of the ridge parameter between 0 and `thetamax` to be used for calculating the regression vector shown in the plots created by the `ridge` routine.

The optional variable *tf* allows the user to position text on the plot when `tf` is set to 1. The text identifies the optimum of the ridge parameter `theta` and can be positioned with cursors or the mouse.

Outputs are `b` the regression column vector at optimum ridge parameter `theta`.

In most instances the optimum ridge parameter will be less than 0.1, often as low as 0.01. A good starting guess when working with the method is to specify `thetamax = 0.1` with `divs = 20`.

See Also

`pcr, pls, analysis, ridgecv`

ridgecv

Purpose

Ridge regression with cross validation.

Synopsis

```
[b,theta,cumpress] = ridge(x,y,thetamax,divs,split)
```

Description

The function `ridgecv` uses cross-validation to create a ridge regression model for a matrix of predictor variables (x-block) `x`, and a matrix of predicted variables (y-block) `y`. The maximum value of the ridge parameter to consider is given by `thetamax` (0 < `thetamax`). `divs` specifies the number of values of the ridge parameter between 0 and `thetamax` to be used for calculating models used in the cross validation and shown in plots created by the routine, and `split` is the number of times the model is rebuilt on a different subset of samples.

Outputs are `b` the regression column vector at optimum ridge parameter `theta` as determined by cross-validation.

In most instances the optimum ridge parameter will be less than 0.1, often as low as 0.01. A good starting guess when working with the method is to specify `thetamax = 0.1` with `divs = 20`.

Note: `RIDGECV` uses the venetian blinds cross-validation method.

See Also

`crossval, pcr, pls, analysis, ridge`

rinverse

Purpose

Calculates pseudo inverse for PLS, PCR and RR models.

Synopsis

```
rinv = rinverse(mod,ncomp)
rinv = rinverse(p,t,w,ncomp)
rinv = rinverse(p,t,ncomp)
rinv = rinverse(sx,theta)
```

Description

For the following I/O format:

```
rinv = rinverse(mod,ncomp)
```

The input `mod` is a model structure from `PCR`, `PLS`, or `ANALYSIS` and `ncomp` is the number of factors in the model (number of principal components or latent variables).

For PLS models, the inputs are the loadings `p`, scores `t`, weights `w` and number of latent variables `ncomp`. For this case the I/O syntax is:

```
rinv = rinverse(p,t,w,ncomp)
```

For PCR models, the inputs are the loadings `p`, scores `t`, and number of principal components `ncomp`. For this case the I/O syntax is:

```
rinv = rinverse(p,t,ncomp)
```

For ridge regression (RR) models, the inputs are the scaled predictor x matrix `sx` and ridge parameter `theta`.

```
rinv = rinverse(sx,theta)
```

See Also

`pcr, pls, ridge, stdsslct`

rmse

Purpose

Calculate Root Mean Square Difference(Error).

Synopsis

```
err = rmse(y1,y2)
```

Description

RMSE is used to calculate the root mean square difference between two vectors or matrices. If the vector or matrix is from a model estimation and measurements then the output is the Root Mean Square Error (RMSE).

Output depends on the input:

A) `y1` is a matrix or vector

```
err = rmse(y1);
```

The output `err` is the root mean square of the elements of `y1`.

B) `y1` is a matrix or vector, `y2` the same size as `y1`

```
err = rmse(y1,y2);
```

The output `err` is the root mean square of the difference between `y1` and `y2`.

C) `y1` is a matrix or vector, `y2` a column vector.

```
err = rmse(y1,y2);
```

The output `err` is the root mean square of the difference between each column of `y1` and `y2`.

For example, `y2` is a reference and the RMSE is calculated between each column of `y1` and the vector `y2`.

See Also

`crossval`

rwb

Purpose

Red-white-blue color map.

Synopsis

```
map = rwb(m)
```

Description

Creates a red to white to blue colormap, useful for plotting values that range from -1 to 1, such as those generated by `CORRMAP`. Optional input *m* specifies the length of the colormap. With no inputs, RWB returns a colormap the same length as the current colormap. The output `map` is the m by 3 colormap matrix.

See Also

`bone, colormap, cool, copper, corrcoef, corrmap, flag, gray, hot, hsv, pink`

savgol

Purpose

Savitzky-Golay smoothing and differentiation.

Synopsis

```
[y_hat,cm] = savgol(y,width,order,deriv,options)
```

Description

SAVGOL performs Savitzky-Golay smoothing on a matrix of row vectors y. At each increment (column) a polynomial of order *order* is fitted to the number of points *width* surrounding the increment. An estimate for the value of the function (*deriv = 0*) or derivative of the function (*deriv > 0*) at the increment is calulated from the fit resulting in a smoothed function `y_hat`. E.g. see A. Savitzky and M.J.E. Golay, Anal. Chem. **36**, 1627 (1964).

`[y_hat,cm] = savgol(y,width,order,deriv)` allows the user to select the number of points in the filter `width` {default = 15}, the order of the polynomial to fit to the points `order` {default = 2}, and the order of the derivative `deriv` {default = 0}.

Output `cm` allows the user to apply smoothing to additional matrices of the same size as `y`, *e.g.* `y_hat2 = y2*cm` where `y2` is the same size as `y` used to determine `cm`.

Note: `width` must be ≥ 3 and odd, and and `deriv` must be ≤ `order`.

Options

options =	a structure array with the following fields:
`useexcluded:`	`[{'true'}	'false']`, governs how excluded data is handled by the algorithm. If 'true', excluded data is used when handling data on the edges of the excluded region (unusual excluded data may influence nearby non-excluded points). When 'false', excluded data is never used and edges of excluded regions are handled like edges of the spectrum (may introduce edge artifacts for some derivatives).
`useexcluded:`	`[{'fast'}	'polyinterp']`, governs how edges of data and excluded regions are handled. 'fast' is standard SavGol approach. 'polyinterp' uses slower, but more stable polynomial interpolation algorithm.

Examples

If y is 3 by 100 then

```
y_hat = savgol(y,11,4,2);
```

yields a 3 by 100 matrix `y_hat` that contains row vectors of the second derivative of rows of `y` resulting from an 11-point quartic Savitzky-Golay smooth of each row of `y`.

See Also

`baseline`, `baselinew`, `deresolv`, `lamsel`, `mscorr`, `polyinterp`, `savgolcv`, `stdfir`, `wlsbaseline`

savgolcv

Purpose

Cross-validation for Savitzky-Golay smoothing and differentiation.

Synopsis

```
cumpress = savgolcv(x,y,lv,width,order,deriv,ind,rm,cvi,pre);  %for x
   class "double"
cumpress = savgolcv(x,y,lv,width,order,deriv,[],rm,cvi,pre);   %for x
   class "dataset"
```

Description

`SAVGOLCV` performs cross-validation of Savitzky-Golay parameters: filter width, polynomial order, and derviative order.

INPUT:

`x` =	*M* by *N* matrix of predictor variables with ROW vectors to be smoothed (*e.g.* spectra), and
`y` =	*M* by *P* matrix of predicted variables.

OPTIONAL INPUTS:

`ind` =	indices of columns of `x` to be used for calibration {default `ind = [1:n]` *i.e.* all `x` columns}.

The following are optional Savitzky-Golay parameters (calls `SAVGOL`). By entering a vector, instead of a scalar, these variables are cross-validated.

`width` =	number of points in filter {default `width = [11 17 23]`}.
`order` =	polynomial order {default `order = [2 3]`}.
`deriv` =	derivative order {default `deriv = [0 1 2]`}.

The following are optional cross-validation parameters (calls `CROSSVAL`).

`lv` =	maximum number of LVs {default `lv = min(size(x))`}.
`rm` =	regression method. Options are: rm = 'nip', PLS via NIPALS algorithm; `rm = 'sim'`, PLS via SIMPLS algorithm {default}, and `rm = 'pcr'`, uses PCR.
`cvi` =	cross-validation method. Options are: `cvi = 'loo'`, leave-one-out, `cvi = 'vet'`, venetian blinds {default}, `cvi = 'con'`, contiguous blocks, and `cvi = 'rnd'`, repeated random test sets.
`split` =	number of subsets to split the data into {default = 5} and is required for `cvi = 'vet'`, `'con'`, or `'rnd'`.
`iter` =	number of iterations {default = 5} and is required for `cvi = 'rnd'`.
`mc` =	0 supresses mean centering of subsets {default `mc = 1`}.

OUTPUT:

The output is a 4 dimensional array with each dimension corresponding to one of the directions cross-validated over.

```
          cumpress(i,:,:,:) =derivative dimension,
          cumpress(:,j,:,:) =latent variable dimension,
          cumpress(:,:,k,:) =window width dimension, and
          cumpress(:,:,:,l) =polynomial order dimension.
```

See Also

`baseline, crossval, lamsel, mscorr, savgol, specedit, stdfir`

scale

Purpose

Scales data using specified means and std. devs.

Synopsis

```
      sx = scale(x,means,stds,options)
```

Description

`sx = scale(x,means)` subtracts a vector `means` from a matrix `x` and returns the result as `sx`. If `means` is the vector of means this routine mean centers `x`.

`sx = scale(x,means,`*stds*`)` subtracts a vector `means` from a matrix `x`, divides each column by the corresponding element in the vector *stds* and returns the result as `sx`. If `means` is the vector of means and *stds* is the vector of standard deviations this routine atuo-scales `x` so that each column of `sx` has zero mean and unit variance.

The optional input `options` is an options structure contianing the field "stdthreshold" which defines a threshold value for standard deviation below which the threshold value will be used in lieu of the actual value. A scalar value is used as a threshold for all variables. A vector is assumed to be equal in length to `stds` and describes the threshold to use on each individual element.

See Also

`auto, gscaler, medcn, mncn, npreprocess, preprocess, rescale`

setpath

Purpose

Modifies and saves current directory

Synopsis

```
setpath(*flag*)
```

Description

`SETPATH` will modify the MATLAB path to include the current directory and all subdirectories and will save the path to the `pathdef.m` file.

If the optional input *flag* i s set to 0 then only the current directory is saved

See Also

`evriinstall`

shuffle

Purpose

Randomly re-order matrix rows.

Synopsis

```
xr = shuffle(x)
[xr,x2r,x3r,x4r...] = shuffle(x,x2,x3,x4...)
[xr,x2r,x3r,...] = shuffle(x,x2,x3,...,'groups')
```

Description

`SHUFFLE` randomly re-orders the rows of the input matrix `x` and returns the results as `xr`.

All additional inputs (*x2*, *x3*, ...) must have same number of rows as `x`, and will have their rows re-ordered to the same random order as `xr`. If the final input is the string `groups` then the first input is sorted into groups of matching rows and the order of the groups is randomly shuffled, keeping group members together. This is useful for random reordering of measurement replicates. If all the rows of the first input are unique, `groups` will have no effect on the behavior of shuffle.

See Also

`delsamps`

simca

Purpose

Create soft independent method of class analogy models for classification.

Synopsis

```
model = simca(x,ncomp,options)       %creates simca model on dataset
   x
model = simca(x,classid,labels)      %models double x with class id
pred  = simca(x,model,options);      %predictions on x with model
options = simca('options');.
```

Description

The function `SIMCA` develops a SIMCA model, which is really a collection of PCA models, one for each class of data in the data set and is used for supervised pattern recognition.

`SIMCA` cross-validates the PCA model of each class using leave-one-out cross-validation if the number of samples in the class is <= 20. If there are more than 20 samples, the data is split into 10 contiguous blocks.

INPUTS:

`x` =	*M* x *N* matrix of class "dataset" where class information is extracted from `x.class{1,1}` and labels from `x.label{1,1}`, or
`x` =	*M* x *N* data matrix of class "double" and
`classid` =	*M* x 1 vector of class identifiers where each element is an integer identifying the class number of the corresponding sample.
`model` =	when making predictions, input `model` is a SIMCA model structure.

OPIONAL INPUTS:

`ncomp` =	integer, number of PCs to use in each model. This is rarely known *a priori*. When `ncomp=[]` {default} the user is querried for number of PCs for each class.
`labels` =	a character array with *M* rows that is used to label samples on Q vs. T^2 plots, otherwise the class identifiers are used.
`options` =	a structure array discussed below.

OUPUT:

`model` =	model structure array with the following fields:
`modeltype:`	`'SIMCA'`,
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,

`info:`	additional model information,
`description:`	cell array with text description of model,
`submodel:`	structure array with each record containing the PCA model of each class (see `PCA`), and
`detail:`	sub-structure with additional model details and results.
`pred` =	is a structure, similar to `model`, that contains the SIMCA predictions. Additional, or other, fields in `pred` are:
`rtsq:`	the reduced T^2 (T^2 divided by it's 95Found confidence limit line) where each column corresponds to each class in the SIMCA model,
`rq:`	the reduced Q (Q divided by it's 95Found confidence limit line) where each column corresponds to each class in the SIMCA model,
`nclass:`	the predicted class number (class to which the sample was closest when considering T^2 and Q combined), and
`submodelpred:`	structure array with each record containing the PCA model predictions for each class (see `PCA`).

Note: Calling `simca` with no inputs starts the graphical user interface (GUI) for this analysis method.

Options

`options` =	a structure array with the following fields:
`display:`	`[ {'on'} | 'off' ]`, governs level of display,
`plots:`	`['none' | {'final'} ]`, governs level of plotting,
`staticplots:`	`['no' | {'yes'} ]`, produce ole-style "static" plots,
`rule:`	`[{'combined'} | 'final' | 'T2' | 'Q']`, decision rule,
`preprocessing:`	`{ [ ] }`, a preprocessing structure (see `PREPROCESS`) that is used to preprocess data in each class.

The default options can be retreived using: `options = simca('options');`.

Note: with `display='off'`, `plots='none'`, `nocomp=`(>0 integer) and preprocessing specified that `SIMCA` can be run without command line interaction.

See Also

`cluster, crossval, pca, plsdthres, discrimprob, plsdaroc, plsdthres`

simpls

Purpose

Partial Least Squares regression using the SIMPLS algorithm.

Synopsis

```
[reg,ssq,xlds,ylds,wts,xscrs,yscrs,basis] = simpls(x,y,ncomp,options)
options = simpls('options');.
```

Description

`SIMPLS` performs PLS regression using SIMPLS algorithm.

INPUTS:

`x` =	X-block (predictor block) class “double” or “dataset”, and
`y` =	Y-block (predicted block) class “double” or “dataset”.

OPIONAL INPUTS:

`ncomp` =	integer, number of latent variables to use in {default = rank of X-block}, and
`options` =	a structure array discussed below.

OUPUTS:

`reg` =	matrix of regression vectors,
`ssq` =	the sum of squares captured (ssq),
`xlds` =	X-block loadings,
`ylds` =	Y-block loadings,
`wts` =	X-block weights,
`xscrs` =	X-block scores,
`yscrs` =	Y-block scores, and
`basis` =	the basis of X-block loadings.

Note: The regression matrices are ordered in `reg` such that each *Ny* (number of Y-block variables) rows correspond to the regression matrix for that particular number of latent variables.

NOTE: in previous versions of SIMPLS, the X-block scores were unit length and the X-block loadings contained the variance. As of Version 3.0, this algorithm now uses standard convention in which the X-block scores contain the variance.

Options

`options` = a structure array with the following fields:

`display:` `[ {'on'} | 'off' ]`, governs level of display, and

`ranktest:` `[ 'none' | 'data' | 'scores' | {'auto'} ]`, governs type of rank test to perform.

`'data'` = single test on X-block (faster with smaller data blocks and more components),

`'scores'` = test during regression on scores matrix (faster with larger data matricies),

`'auto'` = automatic selection, or

`'none'` = assumes X-block has sufficient rank.

The default options can be retreived using: `options = simpls('options');`.

See Also

`crossval, modelstruct, pcr, plsnipal, preprocess, analysis`

snv

Purpose

Standard Normal Variate scaling.

Synopsis

```
[xcorr,mns,sds] = snv(x,options);       %perform snv scaling
x = snv(xcorr,mns,sds);         %undo snv
```

Description

Scales rows of the input x to be mean zero and unit standard deviation. This is the same as autoscaling the transpose of x.

INPUT:

x =	*M* by *N* matrix of data to be scaled (class "double" or "dataset").

OPTIONAL INPUTS:

options =	options structure passed to function "auto" when performing SNV scaling. See auto.m for available options (not valid for undo operation).
mns =	a vector of length *M* of means, and
sds =	vector of length *M* of standard deviations.

OUTPUTS:

xcorr =	the scaled data (xcorr will be the same class as x),
mns =	vector of means for each row, and
sds =	vector of standard deviations for each row.

To rescale or "undo" SNV, inputs are xcorr, mns, and sds from a previous SNV call. The output will be the original x.

See Also

auto, normaliz, preprocess

spcreadr

Purpose

Reads a Galactic SPC file.

Synopsis

```
x = spcreadr(filename,subs,wlrange,options)
[data,xaxis,auditlog] = spcreadr(filename,subs,wlrange,options)
```

Description

SPCREADR reads a Galactic SPC file.

INPUT:

filename =	a text string with the name of a SPC file or a cell of strings of SPC filenames. If filename is omitted or blank, the user will be prompted to select a file graphically. If filename is an empty cell {}, the user will be prompted to select a folder and then one or more SPC files in the folder the identified folder.

OPTIONAL INPUTS:

subs =	[], scalar or vector indicating the sub-files to read, e.g. [3] reads sub-file 3, [3:9] reads sub-files 3 to 9, {default reads all sub-files} and
wlrange =	[], two element vector (inclusive endpoints) of the wavelength range to return {default returns the entire wavelength range}.

OUTPUTS:

x =	a dataset object containing the spectrum, or
data =	a data array with measured intensities,
xaxis =	vector containing the wavelength axis, and
auditlog =	char array with the log from the file.

Options

options =	a structure array with the following fields:
axismatching:	['none' | 'intersect' |{'interpolate'}], defines action taken when the x-axes of two spectra being read do not match. The options are: 'intersect' returns only the points where the spectral x-axis values overlap excatly.

'interpolate' returns the overlapping portions with linear interpolation to match spectral points exactly. As no extrapolation will be done, the returned spectra will cover the smallest common spectral range.

'none' ignores x-axis differences as long as the number of data points is the same in all spectra.

`textauditlog:` `[ {'no'} | 'yes' ]`, governs output of audit log contents. When 'yes', the auditlog is returned as a raw text array. Otherwise, the auditlog is returned as a structure with field names taken from auditlog keys.

See Also

`areadr, xclgetdata, xclputdata, xclreadr`

specedit

Purpose

GUI for selecting spectral regions on a plot.

Synopsis

```
specedit(x,f)
```

Description

If input variable (x) is a vector `SPECEDIT` plots `x` (*e.g.* spectra) versus an optional input *f* e.g. wavelengths. If `x` is a matrix of spectra then `SPECEDIT` plots the mean of `x` where the rows of `x` correspond to different sample spectra and the columns of `x` correspond to different wavelengths. Regions of `x` can be selected using push buttons. The edited matrix input and column indices can be saved to the workspace interactively.

See Also

`baseline`, `lamsel`

ssqtable

Purpose

Prints variance captured table to the command window.

Synopsis

```
    ssqtable(ssq,ncomp)
```

Description

SSQTABLE prints the variance captured table from input ssq to the command window for the desired number of factors *ncomp*. If ssq is a standard model structure (e.g. from ANALYSIS), the model information is displayed along with the variance captured table (see MODLRDER). If *ncomp* is omitted, the entire available tabe is displayed.

Examples

For a standard model structure called modl (e.g. as returned by, ANALYSIS, PCA, or PLS functions)

```
ssqtable(modl.detail.ssq,5)
```

will print the variance captured table *only* for the first 5 factors to the command window. Alternatively,

```
ssqtable(modl,5)
```

will print *both* the model information and the variance captured table for first 5 factors.

See Also

analysis, modlrder, pca, pcr, pls

stdfir

Purpose

Standardization using FIR filtering.

Synopsis

```
sspec = stdfir(nspec,rspec,win,mc)
```

Description

STDFIR is a moving window multiplicative scatter correction with a fixed window size. This algorithm uses an inverse least squares regression. (Also see `MSCORR`.)

Inputs are `nspec` the new spectra to be standardized, `rspec` the standard spectra from the standard instrument (a row vector that is a reference spectrum), and `win` is the window width (must be an odd number).

If the optional input *mc* is 1 {default} the regression allows for an offset and a slope, if *mc* is set to 0 only the slope is used (no offset is used i.e. it is a force fit through zero).

The output is `sspec` the standardized spectra. This routine is based on the method discussed in

Blank, T.B., Sum, S.T., Brown, S.D., and Monfre, S.L., "Transfer of Near-Infrared Multivariate Calibrations without Standards", *Anal. Chem.*, 68(17), 2987-2995, 1996.

See Also

`mscorr, stdgen`

stdgen

Purpose

Piecewise and direct standardization transform generator.

Synopsis

```
[stdmat,stdvect] = stdgen(spec1,spec2,win,options)
options = stdgen('options')
```

Description

`STDGEN` can be used to generate direct or piecewise direct standardization matrix with or without additive background correction. It can also be used to generate the transform using the "double window" method. The transform is based on spectra from two instruments, or original calibration spectra and drifted spectra from a single instrument.

INPUTS:

`spec1` =	*M* by *N1* spectra from the standard instrument, and
`spec2` =	*M* by *N2* spectra from the instrument to be standarized.

OPTIONAL INPUTS:

`win` =	`[]`, empty or a 1 or 2 element vector. If `win` is a scalar then `STDGEN` uses a single window algorithm, and if `win` is a 2 element vector it uses a double window algorithm. `win(1)` = (odd) is the number of channels to be used for each transform, and `win(2)` = (odd) is the number of channels to base the transform on. If `win` is not input it is set to zero and direct standardization is used.
`options` =	a structure array discussed below.

OUTPUTS:

stdmat =	the transform matrix, and
stdvect =	the additive background correction.

Note: if only one output argument is given, no background correction is used.

Options

`options` =	a structure array with the following fields:
`tol:`	`[ {0.01} ]`, tolerance used in forming local models (it equals the minimum relative size of singular values to include in each model), and

maxpc: [ ], specifies the maximum number of PCs to be retained for each local model {default: []}. maxpc must be ≤ the number of transfer samples. If maxpc is not empty it supersedes tol.

The default options can be retreived using: `options = stdgen('options');`.

See Also

baseline, distslct, mscorr, stdfir, stdize, stdsslct

stdize

Purpose

Standardizes new spectra using transform from `STDGEN`.

Synopsis

```
stdspec = stdize(nspec,stdmat,stdvect)
```

Description

Inputs are the new spectra to be standardized `nspec`, and the standardization matrix `stdmat` (output from `STDGEN`).

Optional input `stdvect` is the offset vector (output from `STDGEN`). Note that if `stdvect` was calculated when generating the transform with `STDGEN`, then it should be input when applying the transform with `STDIZE`.

The output is a matrix of the standardized spectra `stdspec`.

See Also

`stdgen, stdsslct`

stdsslct

Purpose

Selects subsets of spectra for use in instrument standardization based on sample leverage.

Synopsis

```
    [specsub,specnos] = stdsslct(spec,nosamps,rinv)
```

Description

STDSSLCT selects samples for use in instrument standardization transform development based on their multivariate leverage.

The inputs are the spectra to be used in generating the transform `spec`, and the number of samples to be selected for the subset `nosamps`. The optional input *rinv* uses the pseudo inverse from a calibration regression model to determine sample leverages.

The outputs are the subset of spectra selected `specsub`, and the sample numbers (indices) of the selected spectra `specnos`.

See Also

`distslct, doptimal, stdgen, stdize, rinverse`

svdlgpls

Purpose

Dialog to save variable to workspace or MAT file.

Synopsis

```
[name,location] = svdlgpls(varin,message)
```

Description

`SVDLPLS` creates a dialog box to save a variable to the base workspace or a MATLAB file from a function (*e.g.* a GUI). Input `varin` is the variable to be saved. The dialog box allows the user to name `varin` to a new variable and select between saving into the base workspace or a file. Variables can be appended onto existing files by selecting the file from the file list or written into new files by providing a new file name. The location for the file can be selecetd from the folders listed in the file list and from the **Look in** menu at the top of the dialog box. Files are always MATLAB "mat" files. The optional text variable *messag* allows a message to be printed in the dialog box.

Optional outputs give information about the variable name `name` and file location `location` used to save the variable. Location will be empty if saved to the base workspace.

See Also

`erdlgpls, lddlgpls`

tconcalc

Purpose

Calculate Hotellings T2 contributions for predictions on a model.

Synopsis

```
tcon = tconcalc(newx,model)
tcon = tconcalc(pred,model)
tcon = tconcalc(model)
```

Description

Inputs are the new data `newx` and the 2-way PCA or regression model for which T2 contributions should be calculated `model`. Alternatively, the prediction structure `pred` calculated with new data can be used in place of the new data itself or both can be omitted (passing model only) to get T2 contributions for the calibration data.

See Also

`datahat, pca, pcr, pls, qconcalc`

testfitpeaks

Purpose

Demo calls to the FITPEAKS function.

Synopsis

```
[peakdef,fval,exitflag,output] = testfitpeaks(test)
```

Description

TESTFITPEAKS is a set of example calls to FITPEAKS. Editing this M-file provides some insight into how the peak fitting utilities can be used.

No input is required.

OPTIONAL INPUT:

test	= calls different peak fitting examples.
	test = 1 fits a single Gaussian peak.
	test = 2 fits two Gaussian peaks.
	test = 3 fits a single Lorentzian peak.
	test = 4 fits two Lorentzian peaks.
	test = 5 fits a Gaussian and Lorentzian peak.
	test = 6 fits a single PVoigt2 peak.
	test = 7 fits a Gaussian and a PVoigt2 peak.
	test = 8 fits a Gaussian and a PVoigt1 peak.
	test = 9 fits a single PVoigt1 peak.

OUTPUTS:

peakdef	= The input peak structure (peakdef) with parameters changed to correspond to the best fit values.
fval	= Scalar value of the objective function evaluated at termination of FITPEAKS.
exitflag	= Describes the exit condition (see LMOPTIMIZEBND).
out	= Structure array with information on the optimization/fitting (see LMOPTIMIZEBND).

See Also

fitpeaks, peakfunction, peakstruct

testpeakdefs

Purpose

Checks peak parameters in a peak definition structure.

Synopsis

```
[out,msg,loc] = testpeakdefs(peakdef)
```

Description

`TESTPEAKDEFS` checks the consistency of the peak definitions in a peak definition structure and is useful for checking the initial guess for (`peakdef`). This function examines each record of a peak definition structure (`peakdef`) and determines:

1) if the lower bounds are lower than the initial guess (any parameters lower than the lower bounds is an error),

2) if the upper bounds are higher than the initial guess (any parameters higher than the upper bounds is an error), and

3) if the number of parameters in each peak definition are consistent with the corresponding peak function (`peakdef.fun` field).

INPUT:

`peakdef.fun`	= a multi-record peak definition structure array where each record is a peak definition.

OUTPUTS:

`out`	= output status code: 0 = no problems discovered. -1 = problem encountered.
`msg`	= error message (last error detected).
`loc`	= location of detected problems. This is a two-column matrix with column one corresponding to a peak with an inconsistent definition, and column two corresponding to the inconsistent parameter definition (e.g. a paramter is < its lower bound). If column two has a zero, this means that there is a peak definition with an inaccurate number of parameters for the specific peak shape (e.g. for `peakdef.fun = Gaussian` there are 3 parameters).

See Also

`peakstruct`

tld

Purpose

Trilinear decomposition.

Synopsis

```
    model = tld(x,ncomp,scl,plots)
```

Description

The trilinear decomposition can be used to decompose a 3-way array as the summation over the outer product of triads of vectors. Inputs are the 3 way array `x` and the number of components to estimate `ncomp`. Optional input variables include scales for each of of the array axes, (*scl1, scl2, scl3*). These axes can be entered as `0` or `[]` placeholders. The output of TLD is a structured array (model) containing all of the model elements in the following fields:

`date`: model creation date stamp

`time`: model creation time stamp

`size`: size of the original input array

`loads`: 1 by 3 cell array of the loadings in each dimension

`res`: 1 by 3 cell array residuals summed over each dimension

`scl`: 1 by 3 cell array with scales for plotting loads

Note that the model loadings are presented as unit vectors for the first two dimensions, remaining scale information is incorporated into the final (third) dimension.

See Also

`gram, outerm, parafac`

trendtool

Purpose

Univariate trend analysis tool.

Synopsis

```
      trendtool(axis,data)
      trendtool(data)
      trendtool
```

Description

TRENDTOOL allows the user to graphically perform univariate analysis of two-way data. Inputs are `axis` which is the variable scale to plot against [can be omitted] and `data` the data to plot in which rows are samples. If `data` is omitted, the user is prompted to load a dataset to analyze.

Right-clicking on the trend data plot allows placement of "markers". Markers return either the height at a point or integrated area between two points. Reference markers can be added to each marker to subtract the height at a point or subtract a two-point baseline from the associated marker. Markers can be saved or loaded using the toolbar buttons. A Waterfall plot (linked to axis range shown in data plot) can be created using the waterfall toolbar button.

The results of the analysis are plotted in the trend results plot which shows a color-coded results of the univariate analysis and allows saving of the analysis results and selection of points to show in the trend data figure.

See Also

`pca, plotgui`

tsqlim

Purpose

Calculates PCA confidence limits for Hotelling's T^2.

Synopsis

```
tsqcl = tsqlim(m,pc,cl)
tsqcl = tsqlim(model,cl)
```

Description

Inputs can be in one of two forms:

(a) the number of samples `m`, the number of principal components used `pc`, and the fractional confidence limit, `cl` (0 < `cl` < 1) which can be a scalar or a vector (to calculate multiple confidence limits simultaneously).

or (b) a standard model structure, `model`, and the fractional confidence limit, `cl` (0 < `cl` < 1).

The output `tsqcl` is the confidence limit. See Jackson (1991).

Examples

```
tsqcl = tsqlim(15,2,0.95)

model = pca(data,pc); tsqcl = tsqlim(model,0.95)
```

See Also

`analysis, pca, pcr, pls`

tsqmtx

Purpose

Calculates the Hotelling's T^2 contributions for PCA.

Synopsis

```
[tsqmat,tsqs] = tsqmtx(x,model)
[tsqmat,tsqs] = tsqmtx(x,p,ssq)
```

Description

`TSQMTX` calculates the Hotelling's T^2 contributions for PCA.

INPUTS:

`x` =	data matrix (class "double" or "dataset), and
`model` =	model structure returned from `ANALYSIS` or `PCA`, or
`p` =	PCA loadings, and
`ssq` =	variance captured table.

If a PCA model structure `model` is input, the loadings and variance captured table are extracted from the model. Additionally, the preprocessing from the model is applied to the data prior to estimating the scores. However, if the loadings `p` and variance captured table `ssq` are passed as inputs then the data must be preprocessed in a manner similar to the data used to calibrate the PCA model.

OUTPUTS:

`tsqmat` =	indivual variable contributions to Hotelling's T^2, and
`tsqs` =	Hotelling's T^2 for each sample.

ALGORITHM

If **P** is the loadings matrix and **T** is the scores matrix from the calibration data that had *M* samples, then **S** is a diagonal matrix defined as $\mathbf{S} = \mathbf{T}^T\mathbf{T}/(M\text{-}1)$. For a new sample $\mathbf{x}_{new}$ (row vector that has been appropriately scaled) the T^2 contribution $\mathbf{t}_{con}$ is calculated as $\mathbf{t}_{con} = \mathbf{x}_{new}\mathbf{P}\mathbf{S}^{-1/2}\mathbf{P}^T$.

See Also

`datahat, pca, pcr, pls`

ttestp

Purpose

Evaluates t-distribution and its inverse.

Synopsis

```
y = ttestp(x,a,z)
```

Description

Evaluates a t-distribution with input flag `z`. For `z = 1` the output `y` is the probability point for given t-statistic `x` with `a` degrees of freedom. For `z = 2` the output `y` is the t-statistic for given probability point `x` with `a` degrees of freedom.

Examples

```
y = ttestp(1.9606,5000,1)

y = 0.025

y = ttestp(0.005,5000,2)

y = 2.533
```

See Also

`ftest`, `statdemo`

tucker

Purpose

TUCKER analysis for n-way arrays.

Synopsis

```
model = tucker(x,ncomp,initval,options)          %tucker model
pred  = tucker(x,model)                           %application
options = tucker('options')
```

Description

TUCKER decomposes an array of order K (where $K \geq 3$) into the summation over the outer product of K vectors. As opposed to PARAFAC every combination of factors in each mode are included (subspaces). Missing values must be NaN or Inf.

INPUTS:

`x` =	the multi-way array to be decomposed and
`ncomp` =	the number of components to estimate, or
`model` =	a `TUCKER` model structure.

OPTIONAL INPUTS:

`initval` =	if *`initval`* is the loadings from a previous TUCKER model are then these are used as the initial starting values to estimate a final model,
	if *`initval`* is a TUCKER model structure then mode 1 loadings (scores) are estimated from x and the loadings in the other modes (see output `pred`),
`options` =	discussed below.

OUTPUTS:

`model` =	a structure array with the following fields:
`modeltype:`	'TUCKER',
`datasource:`	structure array with information about input data,
`date:`	date of creation,
`time:`	time of creation,
`info:`	additional model information,
`loads:`	1 by K+1 cell array with model loadings for each mode/dimension,
`pred:`	cell array with model predictions for each input data block,
`tsqs:`	`cell array with` T^2 values for each mode,
`ssqresiduals:`	`cell array with` sum of squares residuals for each mode,
`description:`	cell array with text description of model, and

`detail:`	sub-structure with additional model details and results.
`pred` =	is a structure array, similar to `model`, that contains prediction results for new data fit to the TUCKER model.

Options

`options` =	a structure array with the following fields:
`display:`	`[ {'on'} | 'off' ]`, governs level of display,
`plots:`	`[ {'final'} | 'all' | 'none' ]`, governs level of plotting,
`weights:`	`[]`, used for fitting a weighted loss function (discussed below),
`stopcrit:`	`[1e-6 1e-6 10000 3600]` defines the stopping criteria as [(relative tolerance) (absolute tolerance) (maximum number of iterations) (maximum time in seconds)],
`init:`	`[ 0 ]`, defines how parameters are initialized (see `PARAFAC`),
`line:`	`[ 0 | {1}]` defines whether to use the line search {default uses it},
`algo:`	this option is not yet active,
`blockdetails:`	'standard'
`missdat:`	this option is not yet active,
`samplemode:`	`[1]`, defines which mode should be considered the sample or object mode and
`constraints:`	{4x1 cell}, defines constraints on parameters (see PARAFAC). The first three cells define constraints on loadings whereas the last cell defines constraints on the core.

The default options can be retreived using: `options = tucker('options');`.

See Also

`datahat, gram, mpca, outerm, parafac, parafac2, tld, unfoldm`

unfoldm

Purpose

Unfolds an augmented matrix for MPCA.

Synopsis

```
      xmpca = unfoldm(xaug,nsamp)
```

Description

`UNFOLDM` unfolds the input matrix `xaug` to create a matrix of unfolded row vectors `xmpca` for MPCA. `xaug` contains `nsamp` matrices $\mathbf{A}_j$ augmented such that `[xaug] = [`A_1`; `A_2`; ...; `A_{nsamp}`]`. For example, for `xaug` of size (nsamp*m by n) each matrix $\mathbf{A}_j$ is of size m by n. For $\mathbf{A}_j$ each m by 1 column a_i is transposed and augmented such that `[`b_j`] = [`a_1`', `a_2`', ..., `a_n`']` and `[xmpca] = [`b_1`; `b_2`; ...; `b_{nsamp}`]`. Note: the $\mathbf{A}_j$ should all be the same size.

Examples

```
a = [1     2     3
           4     5     6
          -1    -2    -3
          -4    -5    -6]

xmpca = unfoldm(a,2)

xmpca = [1     4     2     5     3     6
        -1    -4    -2    -5    -3    -6]
```

See Also

`gscale, mpca, pca, reshape`

unfoldmw

Purpose

Unfolds multiway arrays along specified order.

Synopsis

```
mwauf = unfoldmw(mwa,order)
```

Description

Inputs are the multiway array to be unfolded `mwa` (class "double" or "dataset"), and the dimension (or mode) number along which to perform the unfolding `order`.

The output is the unfolded array `mwauf` (class "double" or "dataset" depending on the input class).

When working with dataset objects, `unfoldmw` will create `label` and `includ` fields consistent with the input. This function is used in the development of PARAFAC models in the alternating least squares steps.

See Also

`mpca`, `outerm`, `parafac`, `reshape`, `tld`, `unfoldm`

updatemod

Purpose

Update model structure to be PLS_Toolbox 3.0 compatible.

Synopsis

```
umodl = updatemod(modl,data)
```

Description

The input `modl` is the PLS_Toolbox Version 2 PLS, PCR, or PCA model to be updated to Version 3.

Optional input *data* is required if the model was constructed using a version older than Version 2.0.1c.

The output is an updated Version 3.0 model `umodl`.

See Also

`analysis, pca, pcr, pls`

varcap

Purpose

Variance captured for each variable in PCA model.

Synopsis

```
vc = varcap(x,loads,scl,plots)
```

Description

VARCAP calculates and displays the percent variance captured for each variable and number of principal components in a PCA model.

Inputs are the properly scaled *M* by *N* data x (*i.e.* scaled using the same scaling used when creating the PCA model) with associated *N* by *K* loadings matrix loads.

Optional input *scl* (1 by *N*) specifies the x-axis for plotting. Optional input *plots* suppresses plotting when set to 0 {default = 1}.

The output is a *K* by *N* matrix of variance captured vc for each variable and each number of PCs considered (vc is number of PCs by number of variables). A stacked bar chart of vc is also plotted. Optional input *plots* suppresses plotting when set to 0 {default = 1}.

See Also

analysis, pca

varcapy

Purpose

Calculate percent y-block variance captured by a PLS regression model.

Synopsis

```
    vc = varcapy(model,options)
```

Description

VARCAPY Calculate percent y-block variance captured by a PLS regression model. Given a PLS regression model, VARCAPY calculates the percent of y-block variance captured by each latent variable of the model for each column of the y-block.

Input is a standard PLS model structure. Outupt is a matrix containing the variance captured by each latent variable (rows) for each column of y (columns).

Options

plots : ['none' |{'final'}] Governs plotting of results.

See Also

`analysis, pca`

varimax

Purpose

Orthogonal rotation of loadings.

Synopsis

```
    vloads = varimax(loads,options);
```

Description

Input `loads` is a N by K matrix with orthogonal columns and the output `vloads` is a N by K matrix with orthogonal columns rotated to maximize the "raw varimax criterion". Optional input *options* is discussed below.

Algorithm

Under varimax the total simplicity S is maximized where $S = \sum_{k=1}^{K} S_k$, and the simplicty for each factor (column) is $S_k = \overline{(a_k - \overline{a}_k)^2}$ where the overbar indicates the mean and a_k is the kth column of `vloads`.

The algorithm is based on Kaiser's VARIMAX Method (J.R. Magnus and H. Neudecker, *Matrix Differential Calculus with Applications in Statistics and Econometrics*, Revised Ed., pp 373-376, 1999). They note that if the algorithm converges, "which is not guaranteed, then a (local) maximum … has been found."

See Also

`analysis, pca`

vip

Purpose

Calculate Variable Importance in Projection from regression model.

Synopsis

```
vip_scores = vip(mode)
```

Description

Variable Importance in Projection (VIP) scores estimate the importance of each variable in the projection used in a PLS model and is often used for variable selection. A variable with a VIP Score close to or greater than 1 (one) can be considered important in given model. Variables with VIP scores significantly less than 1 (one) are less important and might be good candidates for exclusion from the model.

The input is a PLS model structure (model). The output (vip_scores) is a set of column vectors equal in length to the number of variables included in the model. It contains one column of VIP scores for each column of the original calibration y-block.

See Chong & Jun, Chemo. Intell. Lab. Sys. 78 (2005) 103–112.

See Also

`plotloads, pls, plsda`

vline

Purpose

Place a vertical line in an existing figure.

Synopsis

```
      h = vline(x,lc)
```

Description

`VLINE` draws a vertical line on an existing figure from the bottom axis to the top axis at at postions defined by `x` which can be a scalar or vector. If no input is used for *x* the default vaule is zero {default `x = 0`}.

Optional input *lc* is used to define the line style and color as in normal plotting (see `PLOT`). If not inputs are supplied, `VLINE` draws a vertical green line at 0.

Output `h` is the handle(s) of line(s) drawn.

Examples

```
vline([2.5 3],'-r')
```

plots a vertical red line at `x = 2.5 and 3`.

See Also

`dp, ellps, hline, pan, plot, plttern`

wlsbaseline

Purpose

Weighted least squares baseline function.

Synopsis

```
[bldata,wts] = wlsbaseline(data,baseline,options)
[bldata,wts] = wlsbaseline(data,order,options)
```

Description

Subtracts a baseline (or other signal) from a spectrum with the constraint that residuals below zero be weighted more heavily than those above zero. This achieves a robust "non-negaitve" residual fit when residuals of significant amplitude (e.g. signals on a background) are present.

Inputs are `data` the spectral data, `baseline` the reference spectrum/spectra to use for baseline OR an integer value representing the order of polynomial baselining to use and `options` an optional options structure.

Outputs are the baselined data `bldata` and the weightings `wts` indicating the amount of baseline which was removed from each spectrum in data. (i.e. `bldata = data - wts*baseline`)

Polynomial baseline Option: If a positive scalar value is given instead of the input `baseline`, then a polynomial baseline of that order will be used. In this mode, any row of the output `wts` can be used with the polyval function to obtain the baseline removed from the corresponding row of data.

Options

`plots` :	[{'none'} | 'debug' | 'intermediate' | 'final'] governs plots
`weightmode` :	[{1} | 2] flag indicating which weighting mode to use. Mode 1 = Power method. Negative residuals are weighted up by the power of 10.^(option.negw). All residuals are then raised to the power of (option.power) Mode 2 = T squared method. Negative residuals are weighted up by the extent to which the surpass an estimate of the noise limit and the approximate t-limit defined by (option.tsqlim)
`trbflag` :	['bottom' | 'top'] baseline to top or bottom of data
`negw` :	{1} deweighting scale of negative values (10^negw) (used only for weightmode = 1),
`power` :	{2} exponential amplification of residuals (used only for weightmode = 1),

`tsqlim` :	[0.99] t-test confidence limit for significant negative residuals which need to be up-weighted. (used only for weightmode = 2),
`nonneg` :	['no'|{'yes'}] flag to force non-negative baseline weighting, most often used when "real" spectra are used for baslineing and they should not be "flipped" by a negative weighting. Using nonneg = 'yes', WLSBASELINE an be used as a partial CLS prediction to estimate the concentration of a species when not all species' pure component spectra are known,
`delta` :	[1e-4] change-of-fit convergence criterion,
`maxiter` :	[100] maximum iterations allowed per spectrum,
`maxtime` :	[600] maximum time (in seconds) permitted for baselining of all data.

Examples

To swap 4 BYTES in a 32 bit number:

See Also

`baseline, baselinew`

wrtpulse

Purpose

Creates input and output matrices for finite impulse response (FIR) dynamic model identification and prediction.

Synopsis

```
[newu,newy] = wrtpulse(u,y,n,delay)
```

Description

`WRTPULSE` is used to write time series data with muliple inputs and a single output into a form to obtain finite impulse response (FIR) and ARX models. Inputs are a matrix of input vectors `u`, and an output vector `y`. `n` is a row vector with the number of coefficents to use for each input, and `delay` is a row vector containing the number of time units of delay for each input. The output is a matrix of lagged input variables `newu` and the corresponding output vector `newy`.

See Also

`autocor, crosscor, fir2ss, plspulsm`

wtfa

Purpose

Window target factor analysis.

Synopsis

```
[rho,angl,q,skl] = wtfa(spec,tspec,window,p,options)
```

Description

Inputs are a *M* by *N* data matrix `spec`, a *K* by *N* matrix of target spectra `tspec`, the window width `window` > 1, and the number of principal components, PCs, for modelling each window of spectra, `p`. The input `p` is used to govern the PCA model in each window:

`p >= 1`:	(integer) number PCs is a constant `p`,
`0 < p < 1`:	sets a relative criterion for selecting number of PCs in each window i.e. only the first set of PCs that together capture >=p*100Found of the variance in the window are used, or
`p < 0`:	sets an absolute value for number of PCs i.e. factors with singular values <|p| are not used. EWFA (see `EWFA`) can be used as a guide for setting `p` when `p<0`.

Outputs are the cosines `rho` between `tspec` and a `p` component PCA model of `spec` in each window, `angl` `[= acos(rho)]`, and Q residuals `q`. Note that the output values near the end of the record (less than the half width of the window) are plotted as dashed lines and the window center is output in the variable `skl`.

This routine is based on work in: Lohnes, M.T., Guy, R.D., and Wentzell, P.D., "Window Target-Testing Factor Analysis: Theory and Application to the Chromatographic Analysis of Complex Mixtures with Multiwavelength Flourescence Detection", Anal. Chim. Acta, 389, 95-113 (1999).

Options

options = a structure array with the following fields:

`plots`: `[ 'none' | {'angle'} | 'rho' | 'q' ]`, governs plotting,
`'angle'`, plots projection angle {default},
`'rho'`, plots direction cosine, and
`'q'`, plots Q residuals.

`scale`: `[ ]`, is a M element time scale to plot against

The default options can be retreived using: `options = wtfa('options');`.

See Also

`evolvfa, ewfa, pca`

xclgetdata

Purpose

Extract a data table from an Excel spreadsheet.

Synopsis

```
xmat = xclgetdata(filename,datarange,formt)
```

Description

`XCLGETDATA` extracts a data table from an Excel spreadsheet using dynamic data exchange (DDE) and writes it to the variable `xdat`. This function only works on a PC, the spreadsheet must be open in Office 97 or higher, and character arrays can't be extracted.

It has been observed that `XCLGETDATA` won't work unless a copy of the open spreadsheet is saved to the hard drive and the name in `filename` is exact. Also, if the function doesn't work check the Excel menu **tools/options/general** and ensure that the **ignore other applications** check box is unchecked.

Examples

To get a table data from the range C2 to T25 from the open workbook 'book1.xls':

```
data =  xclgetdata('book1.xls','r2c3:r25c20');
```

To get a table data from 'Sheet2' the range D4 to F16 from the open workbook 'book1.xls':

```
data =  xclgetdata('c:\book1.xls\sheet2','r4c4:r16c6');
```

See Also

`areadr`, `spcreadr`, `xclputdata`, `xclreadr`

xclputdata

Purpose

Fill a data table in an Excel spreadsheet.

Synopsis

```
xclputdata(filename,datarange,xmat,formt)
```

Description

`XCLPUTDATA` fills a a range in an Excel spreadsheet using dynamic data exchange (DDE) with a data table contained in the variable `xdat`. This function only works on a PC, the spreadsheet must be open in Office 97 or higher.

If the function doesn't work check the Excel menu **tools/options/general** and ensure that the **ignore other applications** check box is unchecked.

Examples

To place a 3x5 data table contained in the workspace variable `xdat` into the spreadsheet 'book1.xls' in the range B2 to F4:

```
xclputdata('book1.xls','r2c2:r4c6','xdat');
```

See Also

`areadr, spcreadr, xclgetdata, xclreadr`

xclreadr

Purpose

Reads ASCII flat files from MS Excel and other spreadsheets as a DataSet Object.

Synopsis

```
out = xclreadr(file,delim,options)
```

Description

`XCLREADR` reads tab, space, comma, semicolon or bar delimited files with names on the columns (variables) and rows (samples).

If `XCLREADR` is called with no input, or an empty matrix for file name *file*, a dialog box allows the user to select a file to read from the hard disk.

INPUTS:

file = One of the following identifications of files to read:

a) a single string identifying the file to read ('example.txt')
b) a cell array of strings giving multiple files to read ({'example_a' 'example_b' 'example_c'})
c) an empty array indicating that the user should be prompted to locate the file(s) to read ([])

delim = An optional string used to specify the delimiter character.

Supported delimiters include:

'tab'	or	'\t' or sprintf('\t')
'space'	or	' '
'comma'	or	','
'semi'	or	';'
'bar'	or	'|'

If (delim) is omitted, the file will be searched for a delimiter common to all rows of the file and producing an equal number of columns in the result.

OUTPUTS:

out = A DataSet object with date, time, info (data from cell (1,1)) the variable names `vars`, sample names `samps`, and data matrix `data`. Note that the primary difference between this function and the Mathworks function xlsread is the parsing of labels and output of a dataset object.

Note that the primary difference between this function and the Mathworks function xlsread is the parsing of labels and output of a dataset object.

Options

options =	a structure array with the following fields:
`parsing`:	`[ 'manual' | {'automatic'} | 'auto_strict' ]` determines the type of parsing to perform: 'automatic' : the file is automatically parsed for labels and header information. This works on many standard arrangements with different numbers of rows and column labels. May take some time to complete with larger files. See note below regarding additional options available with 'automatic' parsing. 'auto_strict' : faster automatic parsing which does not handle header lines, and expects that all row labels will be on the left-hand side of the data and all column labels will be on the top of the columns. If this returns the wrong result, try 'automatic'. 'manual' : the options below are used to determine the number of labels and header information. Note that when the file type is XLS, 'automatic' parsing is always performed. (the following options are only used when options.parsing='manual')
commentcharacter:	[''] any line that starts with the given character will be considered a comment and parsed into the"comment" field of the DataSet object. Deafult is no comment character. Example: '%' uses % as a commentcharacter. NOTE: Only used with 'automatic' and 'manual' parsing, NOT with 'auto_strict' parsing.
`headerrows`:	[{0}] number of header rows to expect in the file.
`rowlabels`:	[{1}] number of row labels to expect in the file.
`collabels`:	[{1}] number of column labels to expect in the file.

The default options can be retreived using: `options = xclreadr('options');`

In addition to the above options, if option parsing is set to 'automatic', any option used by the PARSEMIXED function can be input to XCLREADR. These options will be passed directly to PARSEMIXED for use in parsing the file. See PARSEMIXED for details.

See Also

`areadr, dataset, spcreadr, xclgetdata, xclputdata, xlsreadr`

xlsreadr

Purpose

Reads .XLS files from MS Excel and other spreadsheets.

Synopsis

```
out = xlsreadr(file,sheets,options)
```

Description

This function reads Microsoft XLS files, parses the contents into a DataSet object. If called with no input a dialog box allows the user to select a file to read from the hard disk. Optional input `file` is a text string with the file name. Optional input (file) is a text string with the file name. Optional input (sheets) is a cell array containing the names of one or more sheets in XLS file to read. Optional input (options) specifies the parsing options. For details on these options, see PARSEMIXED.

Note that the primary difference between this function and the Mathworks function `xlsread` is the parsing of labels and output of a dataset object.

See Also

`areadr`, `dataset`, `xclgetdata`, `xclreadr`

xyreadr

Purpose

Reads one or more ASCII XY or XY... files into a DataSet object.

Synopsis

```
out = xyreadr(file,delim,options)
```

Description

Reads standard XY ASCII files in which the first column is a column of axisscale values (wavelengths, retention times, etc) and the second and possibly subsequent column(s) are values measured at the corresponding axisscale values. Returns a DataSet object with the X as the axisscale in the file and all Y columns (both in the same file and in multiple files) concatenated and transposed as rows.

It is REQUIRED that, if multiple files are being read, they must all have the same X range. If this is not true, the import may fail.

INPUTS:

file = One of the following identifications of files to read:

a) a single string identifying the file to read
('example.txt')

b) a cell array of strings giving multiple files to read
({'example_a' 'example_b' 'example_c'})

c) an empty array indicating that the user should be prompted to locate the file(s) to read
([])

delim = An optional string used to specify the delimiter character.

Supported delimiters include:

'tab'	or	'\t' or sprintf('\t')
'space'	or	' '
'comma'	or	','
'semi'	or	';'
'bar'	or	'|'

If (delim) is omitted, the file will be searched for a delimiter common to all rows of the file and producing an equal number of columns in the result.

OUTPUTS:

out = a DataSet object with the first column of the file(s) stored as the axisscale{2} values and all subsequent column(s) stored as rows of data.

Options:

commentcharacter:	[''] any line that starts with the given character will be considered a comment and parsed into the"comment" field of the DataSet object. Deafult is no comment character. Example: '%' uses % as a commentcharacter.
`headerrows`:	[{0}] number of header rows to expect in the file.
`waitbar`:	['off' |{'on'}] governs use of waitbars to show progress.

See Also

`areadr, dataset, xclgetdata, xclreadr`

yscale

Purpose

Rescale the y-axis limits on each subplot in a figure.

Synopsis

```
yscale(infscale,xrange,allaxes)
ax = yscale(infscale,xrange,allaxes)
```

Description

Each axes on a subplot is rescaled so that the y-scale tightly fits the maximum and minimum of the displayed data. The input `infscale`, when set to 1 (one), also rescales each line object on each axes to tightly fit the new limits (i.e. inf-scales each line object relative to one another). Default is 0 `scale axis to data`. The input `xrange` uses the specified x-axis range for scaling rather than the current axis settings.

If the single output `ax` is requested, the plots are not rescaled, but the axis which would have been used is returned.

The optional third input `allaxes` rescales the specified axis or axes handles. Default is to rescale all axes.

zline

Purpose

Adds vertical lines to 3D figure at specified locations.

Synopsis

```
h = zline(x,y,lc)
```

Description

`ZLINE` draws a vertical line on an existing 3D figure from the bottom axis to the top axis at at postions defined by `x` and `y` which can be a scalar or vector. If no input is used for *x* and `y` the default vaule is zero {default = `0`}.

Optional input *lc* is used to define the line style and color as in normal plotting (see `PLOT`). If not inputs are supplied, `ZLINE` draws a vertical green line at 0.

Output `h` is the handle(s) of line(s) drawn.

Examples

```
zline(2.5, 1.2,'-r')
```

plots a vertical red line at `x = 2.5 and y = 1.2`.

See Also

`dp, ellps, hline, pan, plot, plttern, vline`

Distribution Fitting Tool Set - General Functions

chitest

Purpose

Uses chi-squared to test if sample has a specific distribution.

Synopsis

```
    vals = chitest(x,distname,classes)
```

Description

Assesses how well a particular distribution fits the data (x).

INPUTS:

x = The name of a matrix (column vector) in which the sample data is stored.

distribution = Optional distribution name to assume as the parent distribution for thesample. If this argument is missing, then 'normal' is assumed. This argument must be in single quotes and the name may be abbreviated.

classes = Optional argument naming the number of equal probability intervals for which counts should be collected for the test. If this argument is missing, then the number of classes is taken to be

$$\left\lceil \frac{\max\{x\} - \min\{x\}}{3.5\,\mathrm{var}\{x\}} \, length\{x\} \right\rfloor + 1$$

where {x} is the smallest integer z such that z ≤ x. If specified, the number of classes may not be greater than the length of the data vector.

OUTPUTS:

The return value is a structure with fields:

`chi2` =	value of the test statistic (x^2)
`pval` =	p-value associated with the test statistic
`df` =	degrees of freedom of the test
`classes` =	number of intervals for which counts are obtained
`parameters` =	maximum likelihood estimates
`E` =	expected counts for the classes
`O` =	observed counts for the classes

Note: If a sample contains all negative values, then some of the overlay distributions will not be drawn as they are not applicable. If only some of the sample is made up of negative values, these values are ignored in obtaining the maximum likelihood estimates and subsequent results.

Examples

```
chitest(x)
chitest(x,'exp')
chitest(x,'logistic',12)
```

See Also

`distfit, kstest, plotcqq, plotkd, plotqq`

ck_function

Purpose

Validates distribution function string.

Synopsis

```
    string = ck_function(string)
```

Description

Translates various function string names into internal keyword. Abbreviations can be used with distribution function. For instance, the following example will produce the density distribution at x:

```
>> n = normdf('d',x);
```

INPUTS:

'cumulative'	'c'	'cdf'	
'density'	'd'	'pdf'	
'quantile'	'q'	'inv'	'inverse'
'random'	'r'		

OUTPUTS:
'cumulative'
'density'
'quantile'
'random'

Examples

```
string = ck_function(string);
```

See Also

ensurep

cqtool

Purpose

Interactive conditional quantile-quantile plot gui.

Synopsis

```
        cqtool(x)
```

Description

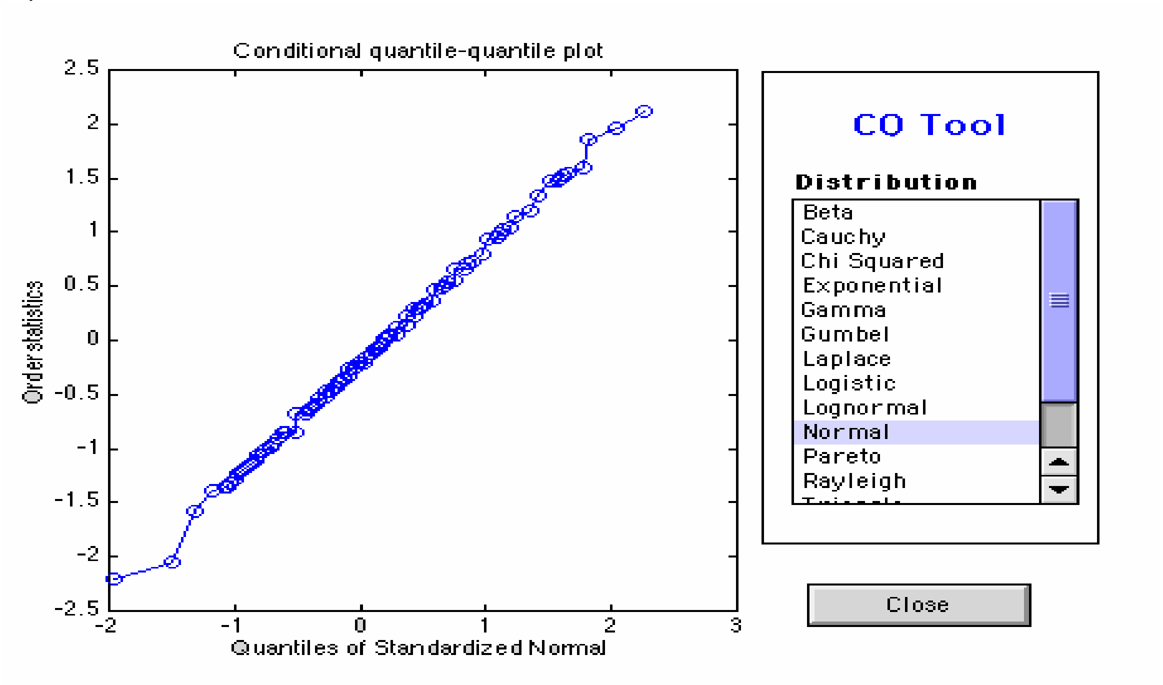
Assesses how well a particular distribution fits the data (x). Conditional quantile plots as described in the 1986 Kafadar and Spiegelman article "An alternative to ordinary q-q plots" in Computational Statistics & Data Analysis are also available in this toolbox

INPUTS:

x = The name of a matrix (column vector) in which the sample data is stored.

Examples

```
cqtool(x)
```



Note: If a sample contains all negative values, then some of the overlay distributions will not be drawn as they are not applicable. If only some of the sample is made up of negative values, these values are ignored in obtaining the maximum likelihood estimates and subsequent results.

See Also

`plotedf`, `plotkd`, `plotcq`, `plotqq`, `plotsym`

distfit

Purpose

Chitest for all distributions.

Synopsis

```
res = distfit(x,options)
```

Description

This command will perform the chi-squared test for all supported distributions and then present a list of the supported distributions from the most likely parent distribution to the least likely (along with the associated p-values).The default behavior is to display a figure containing the results. This can be disabled using `options`.

NOTE: Some distributions will ignore parts of the sample that are not part of the supported range.

INPUTS:

x =	The name of a matrix (column vector) in which the sample data is stored.

OUTPUTS:

The return value is a structure with fields:

`dist` =	names of candidate distributions.
`pval` =	*p*-value associated with the test statistic.

Options:

`name` :	'`options`', name indicating that this is an options structure,
`plots` :	['none' | {'final'}] governs level of plotting,

Examples

```
distfit(x)
```

See Also

`chitest`

ensurep

Purpose

Verifies that input contains only probabilities in [0,1].

Synopsis

```
    prob = ensurep(prob)
```

Description

The input is a real (x) and the output is (prob):

If $x > 1$, then prob = 1.

If $x < 0$, then prob = 0.

If x imaginary, inf, or NaN, then prob = NaN.

Examples

```
prob = ensurep(prob);
```

See Also

`ck_function`

kdensity

Purpose

Calculates the kernel density estimate.

Synopsis

```
[kde, newx] = kdensity(x,code,width,n,at)
```

Description

Produces the kernel density estimate of the data contained in the input vector (x) which must be real.

INPUTS:

x =	The name of a matrix (column vector) in which the sample data is stored.
code =	Integer between 1 and 7 indicating which kernel to use. 1 - Bivwight 2 - Cosine 3 - Epanechnikov {default} 4 - Gaussian 5 - Parzen 6 - Triangle
width =	scalar, optional window width to use in the kernel calculation. If not specified, then the optimal window width is used according to the calculation: $\min\left\{\sigma_x, \frac{p_{75} - p_{25}}{1.349}\right\}\left(\frac{0.9}{n}\right)^{0.20}$
n =	scalar, number of points at which to estimate the density.
at =	vector, allows the user to specify a vector of points at which the density should be estimated. By using this option, it makes it easier to overlay density estimates for different samples on the same graph.

OUTPUTS:

newx =	x input returned.
kde =	The return value is a structure with fields.
x =	vector of points where density was estimated. Will be the same as 'at' input if used.
fx =	?
n =	number of points at which to estimate density. Same as 'n' input if used.
width =	window width used. Same as 'width' input if used.

kernel = name of kernel used.

Examples

```
kde = kdensity(x,2);
kde = kdensity(x,2,22.4);
kde = kdensity(x,2,22.4,50);
kde = kdensity(x,2,22.4,50,y);
```

See Also

`plotkd`

kstest

Purpose

Kolmogorov-Smirnov test that a sample has a specified distribution.

Synopsis

```
      vals = kstest(x,distname)
```

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
	distname =string, optional distribution name to assume as the parent distribution for the sample. Default value is 'normal'.

OUTPUTS:

The return value is a structure with fields (larger values indicate rejecting the named distribution as a candidate parent distribution for the sample). The `ks` is the value of the Kolmogorov-Smirnov statistic and is $\sqrt{n}$ times the maximum difference of the distributions. The maximum difference in the distributions is returned as `Dn`.

`Ks` =	value of the adjusted test statistic.
`Dn` =	unadjusted test statistic.
`parameters` =	maximum likelihood estimates.

Examples

```
kstest(x)
kstest(x,'exp')
```

See Also

```
CHITEST, DISTFIT
```

ktool

Purpose

GUI tool for investigating the kernel density of a sample.

Synopsis

```
ktool(x)
```

Description

Investigate density estimates interactively with various kernel density estimates. Kernel densities are calculated using the kernel with an overlaid best-fit density.

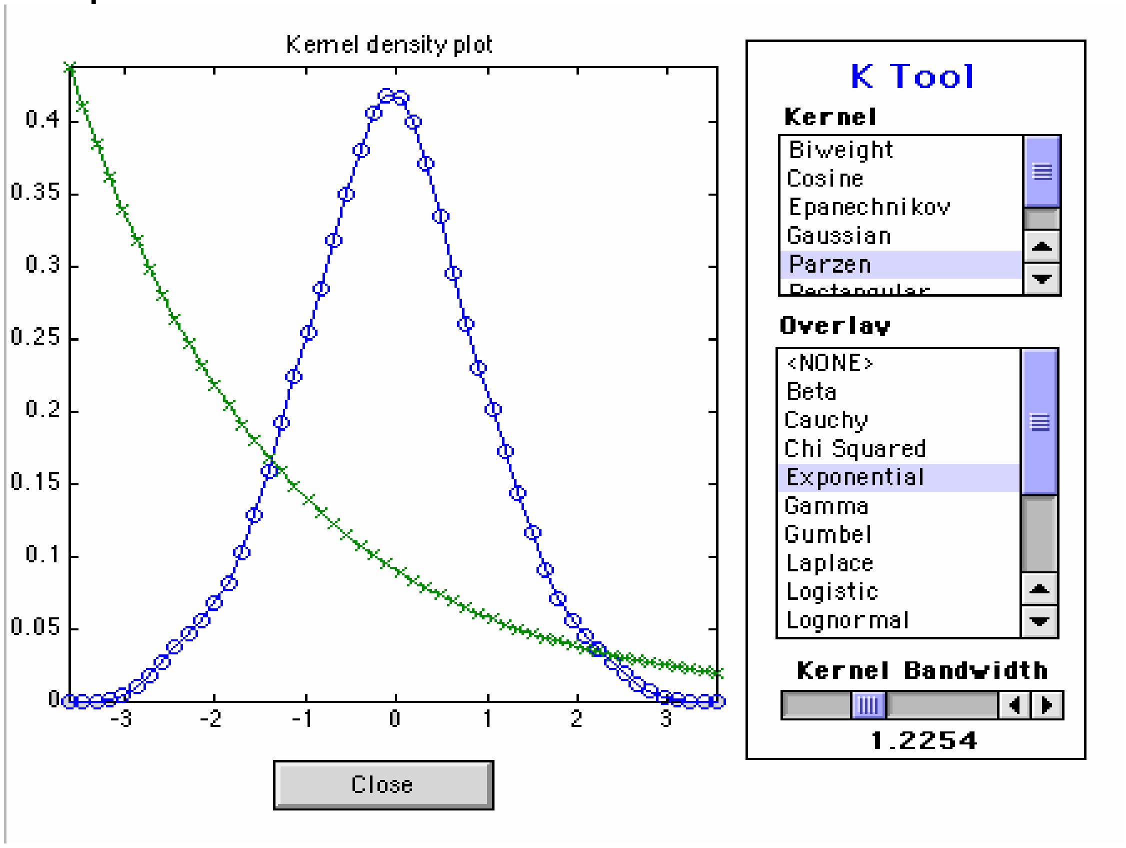
INPUTS:

x = matrix (column vector) in which the sample data is stored.

OUTPUTS:

No outputs.

Examples



Note: If a sample contains all negative values, then some of the overlay distributions will not be drawn as they are not applicable. If only some of the sample is made up of negative values, these values are ignored in obtaining the maximum likelihood estimates and subsequent results.

See Also

`cqtool, plotcqq, plotkd, plotqq, qtool`

means

Purpose

Calculates the algebraic, harmonic, and geometric mean of a vector.

Synopsis

```
vals = means(x)
```

INPUTS:

x =	matrix (column vector) in which the sample data is stored.

OUTPUTS:

The return value is a structure with fields:

`amean` =	arithmetic mean.
`na` =	number of obs used in amean calculation.
`hmean` =	harmonic mean.
`nh` =	number of obs used in hmean calculation.
`gmean` =	geometric mean.
`ng` =	number of obs used in gmean calculation.

Examples

```
mns = means(x);
```

See Also

`summary`

newtondf

Purpose

Newton's root finder.

Synopsis

```
[quantile,exitflag] = newtondf(q,distfun,x,a,b,maxits,tol)
```

Description

Newton's root finder for a given quantile

INPUTS:

q =	matrix, the quantile point of interest
distfun =	string, distribution function name.
x =	matrix, original input matrix
a =	matrix, scale parameter
b =	matrix, shape parameter
maxits =	scalar, maximum number of iterations
tol =	scalar, tolerance

OUTPUTS:

quantile =	matrix, quantile
exitflag =	0 if no error, 1 if maximum iterations is exceeded

Examples

```
[quantile,exitflag] = newtondf(q,distfun,x,a,b);
```

parammle

Purpose

Maximum likelihood parameter estimates.

Synopsis

```
params = parammle(x,distname)
```

Description

Use `parammle` to obtain the best fit parameter estimates for a supported distribution.

Note: Some distributions (beta, Cauchy, gamma, Gumbel, and Weibull) will take longer to find the maximum likelihood estimates as the estimators are not analytically known. They are solved for by optimizing the likelihood.

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
distname =	string, optional distribution name to assume as the parent distribution for the sample. Default value is 'normal'.

OUTPUTS:

The return value is a structure with up to 3 fields depending on the distribution (`distname`).

a =	first paramter.
b =	second parameter (if necessary).
c =	third parameter (if necessary).

Examples

```
params = parammle(x,'exponential')
```

See Also

`chitest`

pctile1

Purpose

Returns the Pth percentile of a data vector.

Synopsis

```
pctile = pctile1(x,p)
```

Description

The return value (`pctile`) is the specified percentile of the sample. This is the function used by the summary command.

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
p =	integer (1,100), percentile to calculate.

Examples

```
pctl = pctile1(x,50)
```

See Also

`pctile2`

pctile2

Purpose

Returns the Pth percentile of a data vector.

Synopsis

```
pctile = pctile2(x,p)
```

Description

The return value (`pctile`) is the specified percentile of the sample. This is an alternative to the `pctile1` command used by the `summary` command.

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
p =	integer (1,100), percentile to calculate.

Examples

```
pctl = pctile2(x,50)
```

See Also

`pctile1`

plotcqq

Purpose

Conditional quantile-quantile plot.

Synopsis

```
      vals = plotcqq(x,distname,translate)
```

Description

Plots a conditional QQplot of a sample in vector (x). Conditional quantile plots as described in the 1986 Kafadar and Spiegelman article “An alternative to ordinary q-q plots” in Computational Statistics & Data Analysis are also available in this toolbox.

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
distname =	string, optional distribution name to assume as the parent distribution for the sample. Default value is 'normal'.
translate =	scalar, axis translation.

OUTPUTS:

The return value is a structure with the following fields:

q =	quantile of the named distribution.
u =	values at which the quantiles were evaluated.

Examples

```
vals = plotcqq(x)
vals = plotcqq(x,'normal')
vals = plotcqq(x,'beta')
```

See Also

plotedf, plotkd, plotqq, plotsym

plotedf

Purpose

Empirical distribution fuction plot.

Synopsis

```
    plotedf(x)
```

Description

Displays a plot of the estimated cumulative distribution..

INPUTS:

x =	matrix (column vector) in which the sample data is stored.

Examples

```
plotedf(x)
```

See Also

`plotcqq, plotpct, plotqq, plotkd`

plotkd

Purpose

Kernel density plot.

Synopsis

```
    plotkd(x,distname,kernel,userw,translate)
```

Description

Provides a kernel density plot of the input x and an overlay.

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
distname =	string, optional distribution name to assume as the parent distribution for the sample. Default value is 'normal'.

kernel =	Integer between 1 and 7 indicating which kernel to use.
	`1 - Bivwight`
	`2 - Cosine`
	`3 - Epanechnikov {default}`
	`4 - Gaussian`
	`5 - Parzen`
	`6 - Triangle`

userw =	scalar, the optional window width to use in the kernel calculation. If not specified, then the optimal window width is used according to the calculation: $\min\left\{\sigma_x, \frac{p_{75} - p_{25}}{1.349}\right\}\left(\frac{0.9}{n}\right)^{0.20}$
translate =	scalar, axis translation.

Examples

```
plotkd(x)
plotkd(x,'normal')
```

See Also

`plotcqq, plotedf, plotqq, plotsym`

plotpct

Purpose

Percentile plot.

Synopsis

```
plotpct(x)
```

Description

Creates a percentile plot of the input (x). Plotted percentiles of centered and scaled x(i) versus i/(N+1).

INPUTS:

x =	matrix (column vector) in which the sample data is stored.

Examples

```
plotpct(x)
```

See Also

`plotcqq, plotedf, plotqq, plotkd`

plotqq

Purpose

Quantile-quantile plot.

Synopsis

```
vals = plotqq(x,distname,options)
```

Description

Makes a quantile-quantile plot of a sample in the input (x) against the optional input (distname). A 45 degree line is also plotted. The larger the deviation from the reference line the more likely it is the input (x) does not come from the distribution (distname).

INPUTS:

x =	matrix (column vector) in which the sample data is stored.
distname =	string, optional distribution name to assume as the parent distribution for the sample. Default value is 'normal'. If distname = 'select' or = '', the user is prompted to select one of the valid distribution types to use. If distname = 'auto' or 'automatic' then the best fitting distribution is used as determined by DISTFIT.
translate =	scalar, axis translation.

OUTPUTS:

The return value is a structure with the following fields:

q =	quantile of the named distribution.
u =	values at which the quantiles were evaluated.

Options

`plots:`	`[ 'none' | {'final'} ]` Governs plotting. If 'none', no plot is created and the function simply returns the fit (see outputs).
`histogram:`	`[ {'off'} | 'on' ]` Governs the plotting of a histogram of the measured and reference distribution below the main QQ plot.
`translate:`	`[ 0 ]` translate the x axis by this offset {default = 0}.
`varname:`	`[ '' ]` label name to use on x-axis and title. Default is empty which uses the actual input variable name.
`color:`	`[ 'b' ]` symbol color to use for the plot(s).

Examples

```
vals = plotqq(x)
vals = plotqq(x,'normal')
vals = plotqq(x,'beta')
```

See Also

plotedf, plotkd, plotcqq, plotsym

plotsym

Purpose

Symmetry plot.

Synopsis

```
    vals = plotsym(x)
```

Description

Plotted are the distances above the median versus the distances below the median. In other words $\text{median} - x_{(i)}$ versus $x_{(n+1-i)} - \text{median}$. If the distribution is symmetric, then all points should lie on a diagonal line.

INPUTS:

x =	matrix (column vector) in which the sample data is stored.

Examples

```
plotsym(x)
```

See Also

`plotedf, plotkd, plotcqq, plotqq`

qtool

Purpose

Interactive quantile-quantile plot gui.

Synopsis

```
qtool(x)
```

Description

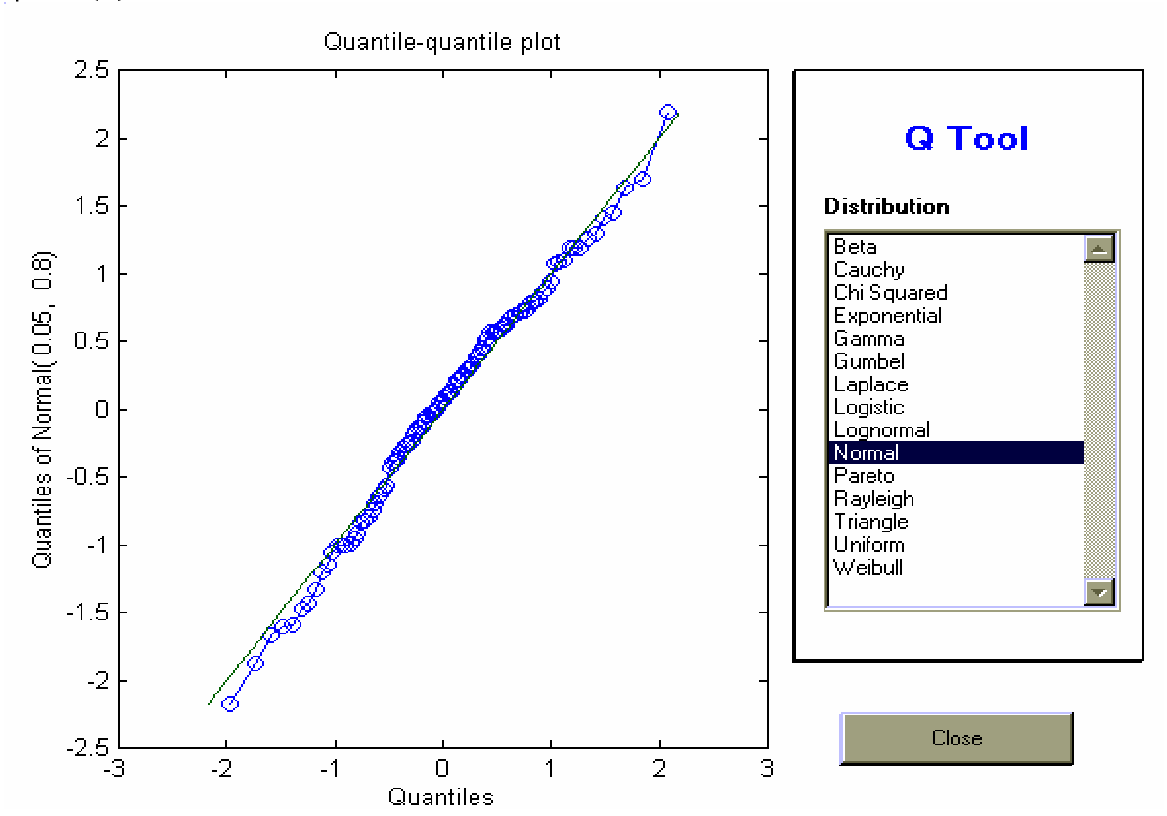
Assesses how well a particular distribution fits the data (x).

INPUTS:

x = The name of a matrix (column vector) in which the sample data is stored.

Examples

```
qtool(x)
```



Note: If a sample contains all negative values, then some of the overlay distributions will not be drawn as they are not applicable. If only some of the sample is made up of negative values, these values are ignored in obtaining the maximum likelihood estimates and subsequent results.

See Also

`plotedf`, `plotkd`, `plotqq`, `plotsym`

resize

Purpose

Resizes arguments to same length.

Synopsis

```
[xout,varargout] = resize(x,varargin)
```

Description

Inputs (x) and (v) can be scalars, vectors, matrices, or multidimensional arrays. The function will attempt to resize all inputs to the largest size of each dimension for any given input as repeated multiple of itself. If input is a scalar, the function will return that scalar.

Examples

```
(newx,newv1,newv2) = resize(x,v1,v2,v3);
```

original sizes are:

x - 2x2x2

v1 - 2x6

v2 - 4x1

v3 - 1x1

new sizes are:

newx - 4x6x2

newv1 - 4x6x2

newv2 - 4x6x2

newv3 - 1x1

See Also

`repmat`

summary

Purpose

Summarizing statistics for sample data.

Synopsis

```
summ = sumary(x)
```

INPUTS:

x =	matrix (column vector) in which the sample data is stored.

Outputs:

The return value is a structure with fields:

mean =	mean of the sample
std =	standard deviation of the sample
n =	number of observations
min =	minimum value in the sample
max =	maximum value in the sample
p10 =	tenth percentile
p25 =	twenty-fifth percentile (lower quartile)
p50 =	fiftieth percentile (median)
p75 =	seventy-fifth percentile (upper quartile)
p90 =	nintieth percentile
skew =	skewness
kurt =	kurtosis

Examples

```
summ = summary(x);
```

See Also

`means`

ttest1

Purpose

One sample t-test.

Synopsis

```
      result = ttest1(x,mu,test)
```

Description

Calculates a one sample t-test for sample (x).

INPUTS:

x =	The name of a matrix (column vector) in which the sample data is stored.
mu =	scalar, the null hypthesis value for the mean {default = 0}.
ttest =	[-1,{0},1] indicates what ttest is for:
	-1 - lower tail H0: mean(x) <= mean(y)
	0 - wo-tail H0: mean(x) ~= mean(y) {default}
	1 - upper tail H0: mean(x) >= mean(y)

OUTPUTS:

The output (result) a structure with the following fields:

t =	test statistic.
p =	probability value
mean =	mean of x
var =	variance of x
n =	length of x
se =	standard error
df =	degress of freedom
hyp =	hypothesis being tested

Examples

```
result = ttest1(x);
result = ttest1(x,mu);
result = ttest1(x,mu,test);
```

See Also

```
ttest2e, ttest2u, ttest2p
```

ttest2e

Purpose

Two sample t-test (assuming equal variance).

Synopsis

```
result = ttest2e(x,y,test)
```

Description

Calculates a two sample t-test for samples (x) and (y) assuming equal variance.

INPUTS:

`x` =	matrix (column vector) in which the sample data is stored.
`y` =	matrix (column vector) in which the sample data is stored.
`ttest` =	[-1,{0},1] indicates what ttest is for: -1 - lower tail H0: mean(x) <= mean(y) 0 - wo-tail H0: mean(x) ~= mean(y) {default} 1 - upper tail H0: mean(x) >= mean(y)

OUTPUTS:

The output (result) a structure with the following fields:

`t` =	test statistic.
`p` =	probability value
`mean1` =	mean of x
`mean2` =	mean of y
`var1` =	variance of x
`var2` =	variance of y
`n1` =	length of x
`n2` =	length of y
`pse` =	pooled standard error
`df` =	degress of freedom
`hyp` =	hypothesis being tested

Examples

```
result = ttest2e(x,y);
result = ttest2e(x,y,test);
```

See Also

ttest1, ttest2u, ttest2p

ttest2p

Purpose

Two sample paired t-test.

Synopsis

```
result = ttest2e(x,y,test)
```

Description

Calculates a two sample paired t-test for samples (x) and (y).

INPUTS:

`x` =	matrix (column vector) in which the sample data is stored.
`y` =	matrix (column vector) in which the sample data is stored.
`ttest` =	[-1,{0},1] indicates what ttest is for: -1 - lower tail H0: mean(x) <= mean(y) 0 - wo-tail H0: mean(x) ~= mean(y) {default} 1 - upper tail H0: mean(x) >= mean(y)

OUTPUTS:

The output (result) a structure with the following fields:

`t` =	test statistic.
`p` =	probability value
`mean` =	mean of x - y
`var` =	variance of x - y
`n` =	length of x - y
`se` =	standard error
`df` =	degress of freedom
`hyp` =	hypothesis being tested

Examples

```
result = ttest2p(x,y);
result = ttest2p(x,y,test);
```

See Also

```
ttest1, ttest2e, ttest2u
```

ttest2u

Purpose

Two sample t-test (assuming unequal variance).

Synopsis

```
result = ttest2u(x,y,test,dfapp)
```

Description

Calculates a two sample t-test for samples (x) and (y) assuming unequal variance.

INPUTS:

`x` =	matrix (column vector) in which the sample data is stored.
`y` =	matrix (column vector) in which the sample data is stored.
`ttest` =	[-1,{0},1] indicates what ttest is for: -1 - lower tail H0: mean(x) <= mean(y) 0 - wo-tail H0: mean(x) ~= mean(y) {default} 1 - upper tail H0: mean(x) >= mean(y)
`dfapp` =	[{-1}, 1] indicates which degree of freedom calculation to use. -1 - indicates Welch's approximate degrees of freedom {default} 1 - indicates Satterthwaite's approximate degrees of freedom

OUTPUTS:

The output (result) a structure with the following fields:

`t` =	test statistic.
`p` =	probability value
`mean1` =	mean of x
`mean2` =	mean of y
`var1` =	variance of x
`var2` =	variance of y
`n1` =	length of x
`n2` =	length of y
`pse` =	pooled standard error
`df` =	degress of freedom
`app` =	'Satterthwaite' or 'Welch'
`hyp` =	hypothesis being tested

Examples

```
result = ttest2u(x,y);
result = ttest2u(x,y,test);
result = ttest2e(x,y,test,dfapp);
```

See Also

`ttest1, ttest2u, ttest2p`

Distribution Fitting Tool Set - Distribution Functions

betadf

Purpose

Beta distribution.

Synopsis

```
prob = betadf(function,x,a,b,options)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Beta distribution.

This distribution is commonly used to model activity time. In its usual form, the data must be in (0,1), but this toolbox will allow both a location and scale parameter (in addition to the `a` and `b` above). This may be symmetric or asymmetric.

$$B(a,b)=\int_0^1 u^{a-1}(1-u)^{b-1}\,\mathrm{d}u$$

$$f(x)=\frac{x^{a-1}(1-x)^{b-1}}{B(a,b)}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (0,1). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	scale parameter (real and nonnegative).
`b` =	shape parameter (real and nonnegative).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Options

options is a structure array with the following fields:

`name`:	`'options'`, name indicating that this is an options structure,
`scale`:	`{1}`, scale for the ordinate, and
`offset`:	`{0}`, offset for the ordinate.

The default options structure can be retrieved using: `options = betadf('options')`.

Examples

Cumulative:

```
>> prob = betadf('c', [0.85 0.9],1,2)
prob =
    0.9775    0.9900

>> x    = [0:0.01:1];
>> plot(x,betadf('c',x,1,2),'b-',x,betadf('c',x,0.5,0.5),'r-')
```

Density:

```
>> prob = betadf('d', 0.9, 1, 2)
prob =
0.2000

>> x    = [0:0.01:1];
>> plot(x,betadf('d',x,1,2),'b-',x,betadf('d',x,0.5,0.5),'r-')
```

Quantile:

```
>> prob = betadf('q',[0.9775 0.9900]',1,2)
prob =
    0.8500
    0.9000
```

Random:

```
>> prob = betadf('r',[5 1],1,2)
prob =
    0.3791
    0.2549
    0.8169
    0.0216
    0.1516
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

cauchydf

Purpose

Cauchy distribution.

Synopsis

```
prob = cauchydf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Cauchy distribution.

This distribution is equivalent to a t-distribution with zero degrees of freedom and is symmetric.

From: http://www.brighton-webs.co.uk/distributions/cauchy.asp

(The Cauchy distribution is a symmetrical, and to use a technical term, heavy tailed. Heavy tailed means that a high proportion of the population is comprised of extreme values.

There is no analytical definition of moment based properties (e.g. mean, variance etc.) thus the parameters are typically described as the location parameter and a scale factor. The most easily derived property is the median for this reason and for consistency with the rest of the site, the parameters have been defined as the median and a scale factor.

The moment based properties derived from a set of random numbers do not provide any useful information on the properties of the distribution.

The Cauchy distribution is also known as the Lorentzian Distribution.

An application of the Cauchy distribution is in software testing where it is necessary to use datasets which contain a few extreme values which might trigger some adverse reaction.)

$$f(x)=\left\{\pi b\left[1+\left(\tfrac{x-a}{b}\right)^2\right]\right\}^{-1}$$

$$F(x)=\tfrac{1}{2}+\tfrac{1}{\pi}\arctan\left(\tfrac{x-a}{b}\right)$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	median or location parameter (real).
`b` =	scale parameter (real and positive). Describes distribution of data around the mode.

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> x    = [-5:0.1:5];
>> prob = cauchydf('c',x);
>> plot(x,prob), vline

>> x    = [-8:0.1:8];
>> prob = cauchydf('c',x);
>> plot(x,prob), vline([0; cauchydf('q',[0.9 0.95])'])
```

Density:

```
>> prob = cauchydf('d',x);
>> plot(x,prob), vline

>> x    = [-8:0.1:8];
>> prob = cauchydf('d',x);
>> plot(x,prob), vline([0; cauchydf('q',[0.9 0.95])'])
```

Quantile:

```
>> x2 = cauchydf('q',cauchydf('c',x));
>> plot(x,x2,'.'), dp
```

Random:

```
>> prob = cauchydf('r',[4 1])
prob =
    0.0480
   -1.0204
    5.7400
   -0.2175
```

See Also

betadr, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

chidf

Purpose

Chi-squared distribution.

Synopsis

```
    prob = chidf(function,x,a)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Chi-sqared distribution.

The chi-squared distribution usually models data that are positive (such as the sum of physical measurements). With integer degrees of freedom parameter v, it is equal to the sum of v normally distributed variates. This toolbox does not require that the degrees of freedom be integral and will ignore negative values in a sample. Chi-squared distributions have variance equal to twice the mean.

$$f(x) = \frac{x^{(a-2)/2}\exp(-x/2)}{2^{a/2}\Gamma(a/2)}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (0,inf).
	for function=quantile - matrix with values in the interval (0,1).
	for function=random - vector indicating the size of the random matrix to create.
`a` =	degrees of freedom parameter (positive integer).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = chidf('c',[3.7942 4.6052],2)
prob =
    0.8500    0.9000

>> x = 0:0.1:8;
>> plot(x,chidf('c',x,2),'b',x,chidf('c',x,0.5),'r')
```

Density:

```
>> prob = chidf('d',[3.7942 4.6052],2)
prob =
    0.0750    0.0500

>> x = 0:0.1:8;
>> plot(x,chidf('d',x,2),'b',x,chidf('d',x,0.5),'r')
```

Quantile:

```
>> prob = chidf('q',[0.85 0.9],2)
prob =
     3.7942    4.6052
```

Random:

```
>> prob = chidf('r',[4 1],2)
prob =
    0.1023
    2.9295
    0.9990
    1.4432
```

See Also

betadr, cauchydf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

expdf

Purpose

Exponential distribution.

Synopsis

```
prob = expdf(function,x,a)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for an Exponential distribution.

The exponential distribution is commonly used to measure lifetime data (time to failure of light bulbs, time to failure of a particular resistor on a circuit board, etc.). It may also measure time between events. The distribution is skewed to the right. The variance is equal to the square of the mean in this distribution. Negative values in the sample are ignored.

$$f(x) = a\exp(-ax)$$

$$F(x) = 1 - \exp(-ax)$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	mean/scale parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = expdf('c',[3.7942 4.6052],2)
prob =
    0.8500    0.9000

>> x = 0:0.1:8;
>> plot(x,expdf('c',x,2),'b',x,expdf('c',x,0.5),'r')
```

Density:

```
>> prob = expdf('d',[3.7942 4.6052],2)
prob =
    0.0750    0.0500

>> x = 0:0.1:8;
>> plot(x,expdf('d',x,2),'b',x,expdf('d',x,0.5),'r')
```

Quantile:

```
>> prob = expdf('q',[0.85 0.9],2)
prob =
     3.7942    4.6052
```

Random:

```
>> prob = expdf('r',[4 1],2)
prob =
    0.3271
    2.3940
    0.9508
    3.9324
```

See Also

betadr, cauchydf, chidf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

gammadf

Purpose

Gamma distribution.

Synopsis

```
prob = gammadf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Gamma distribution.

This distribution is commonly used to measure lifetime data (like the exponential distribution). The variance may be smaller, equal, or larger than the mean for this distribution and may also be symmetric or asymmetric. Negative values in the sample are ignored.

$$f(x) = \frac{(x/a)^{b-1}\exp(-x/a)}{a\Gamma(b)}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (0,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	scale parameter (real and nonnegative).
`b` =	shape parameter (real and nonnegative).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = gammadf('c',0.99,0.5)
prob =
    0.8406

>> x    = [0:0.1:10];
>> plot(x,gammadf('c',x,2),'b-',x,gammadf('c',x,0.5),'r-')
```

Density:

```
>> prob = gammadf('d',0.99,0.5)
prob =
0.2107

>> x    = [0:0.1:10];
>> plot(x,gammadf('d',x,2),'b-',x,gammadf('d',x,0.5),'r-')
```

Quantile:

```
>> prob = gammadf('q',0.99,0.5)
prob =
    3.3174
```

Random:

```
>> prob = gammadf('r',[4 1],2)
ans =
    0.4549
    0.4638
    0.3426
    0.5011
```

See Also

betadr, cauchydf, chidf, expdf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

gumbeldf

Purpose

Gumbel distribution.

Synopsis

```
prob = gumbeldf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Gumbel distribution.

This distribution is also known as the Type I extreme value distribution. It is an alternative to the Weibull distribution.

$$f(x)=\frac{\Gamma\left[\frac{a+b}{2}\right]\left(a/b\right)^{a/2}x^{(a-2)/2}}{\Gamma\left(\frac{a}{2}\right)\Gamma\left(\frac{b}{2}\right)\left[1+\frac{a}{b}x\right]^{(a+b)/2}}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf).
	for function=quantile - matrix with values in the interval (0,1).
	for function=random - vector indicating the size of the random matrix to create.
`a` =	mode/location parameter (real).
`b` =	scale parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = gumbeldf('c',0.99,0.5,1)
prob =
    0.5419
>> x    = [0:0.1:10];
>> plot(x,gumbeldf('c',x,2),'b-',x,gumbeldf('c',x,0.5),'r-')
```

Density:

```
>> prob = gumbeldf('d',0.99,0.5,1)
prob =
0.3320

>> x    = [0:0.1:10];
>> plot(x,gumbeldf('d',x,2),'b-',x,gumbeldf('d',x,0.5),'r-')
```

Quantile:

```
>> prob = gumbeldf('q',0.99,0.5,1)
prob =
    5.1001
```

Random:

```
>> prob = gumbeldf('r',[4 1],2,1)
ans =

    3.8437
    2.6508
    2.3566
    4.2479
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

laplacedf

Purpose

Laplace distribution.

Synopsis

```
prob = laplacedf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Laplace distribution.

This distribution is a symmetric distribution also known as the double exponential distribution. It is more peaked than the normal distribution Leptokurtic rather than mesokurtic means that it has a sharper peak at the mean in the density plot than a similar normal density

$$f(x) = \tfrac{1}{2b}\exp\left(-\tfrac{|x-a|}{b}\right)$$

$$F(x) = \tfrac{1}{2}\exp\left[-\tfrac{a-x}{b}\right]\mathrm{I}(x < a) + 1 - \tfrac{1}{2}\exp\left[-\tfrac{x-a}{b}\right]\mathrm{I}(x \geq a)$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (0,1).
	for function=quantile - matrix with values in the interval (0,1).
	for function=random - vector indicating the size of the random matrix to create.
`a` =	scale parameter (real and positive).
`b` =	shape parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = laplacedf('c',0.99,1,2)
prob =
    0.4975

>> x    = [0:0.1:10];
>> plot(x,laplacedf('c',x,1,2),'b-',x,laplacedf('c',x,3,7),'r-')
```

Density:

```
>> prob = laplacedf('d',0.99,1,1)
prob =
    0.4950

>> x    = [0:0.1:10];
>> plot(x,laplacedf('d',x,2,1),'b-',x,laplacedf('d',x,0.5,1),'r-')
```

Quantile:

```
>> prob = laplacedf('q',0.99,0.5,1)
prob =
    4.4120
```

Random:

```
>> prob = laplacedf('r',[4 1],2,1)
ans =
    0.4549
    0.4638
    0.3426
    0.5011
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

logisdf

Purpose

Logistic distribution.

Synopsis

```
prob = logisdf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Logistic distribution.

This distribution is a common alternative to the normal distribution. It is symmetric and many times used when data represents midpoints of interval data (data collected in such a way that a range instead or an exact value is collected). The variance may be smaller, equal, or larger than the mean for this distribution.

$$f(x) = \frac{\exp\left[-(x-a)/b\right]}{b\left\{1+\exp\left[-(x-a)/b\right]\right\}^2}$$

$$F(x) = \tfrac{1}{2}\left\{1+\tanh\left[\tfrac{1}{2}(x-a)/b\right]\right\}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	mean parameter (real).
`b` =	standard deviation parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = logisdf('c',0.99,1,2)
prob =
    0.4988

>> x    = [0:0.1:10];
>> plot(x,logisdf('c',x,1,2),'b-',x,logisdf('c',x,3,.5),'r-')
```

Density:

```
>> prob = logisdf('d',0.99,1,2)
prob =
    0.1250

>> x    = [0:0.1:10];
>> plot(x,logisdf('d',x,2,1),'b-',x,logisdf('d',x,0.5,1),'r-')
```

Quantile:

```
>> prob = logisdf('q',0.99,1,2)
prob =
   10.1902
```

Random:

```
>> prob = logisdf('r',[4 1],2,1)
ans =
    0.4549
    0.4638
    0.3426
    0.5011
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

lognormdf

Purpose

Lognormal distribution.

Synopsis

```
prob = lognormdf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Lognormal distribution.

This distribution may be used to characterize data that are themselves products or attribute data (square footage, acreage, etc.). The distribution is skewed to the right, but for very large means, may look nearly symmetric. Negative values in the sample are ignored.

$$f(x) = \frac{1}{xb(2\pi)^{1/2}} \exp\left\{ \frac{-(\log x - a)^2}{2b^2} \right\}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	mean parameter (real and positive).
`b` =	standard deviation parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = lognormdf('c',0.99,1,2)
prob =
    0.3068

>> x    = [0:0.1:10];
>> plot(x,lognormdf('c',x,1,2),'b-',x,lognormdf('c',x,3,7),'r-')
```

Density:

```
>> prob = lognormdf('d',0.99,1,1)
prob =
    0.2420

>> x    = [0:0.1:10];
>> plot(x,lognormdf('d',x,2,1),'b-',x,lognormdf('d',x,0.5,1),'r-')
```

Quantile:

```
>> prob = lognormdf('q',0.99,0.5,1)
prob =
   16.8837
```

Random:

```
>> prob = lognormdf('r',[4 1],2,1)
ans =
   13.5191
    4.4913
   19.8518
    8.7712
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf

normdf

Purpose

Normal / Gaussian distribution.

Synopsis

```
prob = normdf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Normal distribution.

This distribution is used for many data types including physical attributes and sums of quantities. It is a symmetric distribution and the variance can be smaller, equal, or larger than the mean.

$$f\left(x\right)=\frac{1}{b\left(2\pi\right)^{1/2}}\exp\left[-\frac{\left(x-a\right)^2}{2b^2}\right]$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	mode/location parameter (real).
`b` =	scale parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = normdf('c',[1.9600 2.5758])
ans =
    0.9750    0.9950

>> x = -5:.1:5;
>> plot(x,normdf('c',x,0,1)), vline([ 0 ; normdf('q',[0.975; 0.995],0,1)])
```

Density:

```
>> prob = normdf('d',[1.9600 2.5758],0,1)
ans =
    0.0584    0.0145

>> x = -5:.1:5;
>> plot(x,normdf('d',x,0,1)), vline([0; normdf('q',[0.975; 0.995],0,1)])
```

Quantile:

```
>>
ans =
    1.9600    2.5758
```

Random:

```
>> prob = normdf('r',[4 1],0,1)
ans =
   -0.4326
   -1.6656
    0.1253
    0.2877
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, paretodf, raydf, triangledf, unifdf, weibulldf

paretodf

Purpose

Pareto distribution.

Synopsis

```
    prob = paretodf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Pareto distribution.

This distribution is commonly used to model financial data (especially insurance data). It is skewed to the right and the variance may be smaller, equal, or larger than the mean. Negative values in the sample are ignored.

$$f(x) = ba^b / x^{b+1}$$

$$F(x) = 1-(a/x)^b$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	scale parameter (real and positive).
`b` =	shape parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = paretodf('c',2,1,2)
prob =
    0.7500

>> x    = [0:0.1:10];
>> plot(x,paretodf('c',x,1,2),'b-',x,paretodf('c',x,3,7),'r-')
```

Density:

```
>> prob = paretodf('d',2,1,1)
prob =
    0.2500

>> x    = [0:0.1:10];
>> plot(x,paretodf('d',x,2,1),'b-',x,paretodf('d',x,0.5,1),'r-')
```

Quantile:

```
>> prob = paretodf('q',0.5,1,2)
prob =
    1.4142
```

Random:

```
>> prob = paretodf('r',[4 1],2,1)
ans =
   40.1037
    2.6012
    5.0870
    3.8909
```

See Also

`betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, raydf, triangledf, unifdf, weibulldf`

raydf

Purpose

Rayleigh distribution.

Synopsis

```
prob = raydf(function,x,a)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Rayleigh distribution.

This distribution is commonly used to model lifetime data (time to failure). It is skewed to the right and the variance is usually larger than the mean (though it can be smaller or equal). Negative values in the sample are ignored.

$$f(x) = \left(x/a^2\right)\exp\left[-x^2/\left(2a^2\right)\right]$$

$$F(x) = 1 - \exp\left[-x^2/\left(2a^2\right)\right]$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf).
	for function=quantile - matrix with values in the interval (0,1).
	for function=random - vector indicating the size of the random matrix to create.
`a` =	scale parameter (real).

Note: If inputs (`x` and `a`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = raydf('c',2,1)
prob =
    0.8647

>> x    = [0:0.1:10];
>> plot(x,raydf('c',x,1),'b-',x,raydf('c',x,3),'r-')
```

Density:

```
>> prob = raydf('d',2,1)
prob =
    0.2707

>> x    = [0:0.1:10];
>> plot(x,raydf('d',x,2),'b-',x,raydf('d',x,0.5),'r-')
```

Quantile:

```
>> prob = raydf('q',0.5,1)
prob =
    1.1774
```

Random:

```
>> prob = raydf('r',[4 1],2)
ans =
    4.2135
    3.3893
    2.2085
    0.3865
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, triangledf, unifdf, weibulldf

tdf

Purpose

Student's t distribution.

Synopsis

```
prob = tdf(function,x,a)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Student's t distribution.

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (0,1). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
a =	scale parameter (real)

Note: If inputs (`x` and `a`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

See Also

`betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf, weibulldf`

triangledf

Purpose

Triangle distribution.

Synopsis

```
prob = triangledf(function,x,a,b,c)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Triangle distribution.

This distribution is usually used for rough models of data and is triangular in shape (hence the name).

$$f(x) = \frac{2(x-a)}{(b-a)(c-a)} \mathrm{I}(a \le x \le c) + \frac{2(b-x)}{(b-a)(b-c)} \mathrm{I}(c \le x \le b)$$

$$F(x) = \frac{(x-a)^2}{(b-a)(c-a)} \mathrm{I}(a \le x \le c) + 1 - \frac{(b-x)^2}{(b-c)(c-a)} \mathrm{I}(c \le x \le b)$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored (-inf,inf). quantile - interval (0,1). random - vector indicating the size of the random matrix to create.
`a` =	"min" parameter (real, <= mode).
`b` =	"max" parameter (real, >= mode).
`c` =	"mode" parameter (real, >= min and <=max).

Note: If inputs (`x`, `a`, `b`, and `c`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but will convert them to NaN.

Examples

Cumulative:

```
>> prob = triangledf('c',2,1,3,2)
prob =
    0.5000

>> x    = [0:0.1:10];
>> plot(x,triangledf('c',x,1,3,2),'b-',x,triangledf('c',x,1,5,3),'r-')
```

Density:

```
>> prob = triangledf('d',2,1,3,2)
prob =
    1.0000

>> x    = [0:0.1:10];
>> plot(x,triangledf('d',x,0,3,0),'b-',x,triangledf('d',x,1,3,2),'r-')
```

Quantile:

```
>> prob = triangledf('q',0.5,1,3,2)
prob =
    2.0000
```

Random:

```
>> prob = triangledf('r',[4 1],1,3,2)
ans =
    2.2817
    1.9431
    2.1094
    2.2585
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, unifdf, weibulldf

unifdf

Purpose

Uniform distribution.

Synopsis

```
prob = unifdf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Uniform distribution.

This distribution is used when all possible outcomes of an experiment are equally likely. The distribution is flat with no peak.

$$f\left(x\right)=\frac{1}{b-a}$$

$$F\left(x\right)=\frac{x-a}{b-a}$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	"min" parameter (real).
`b` =	"max" parameter (real and >= min).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = unifdf('c',1.5,1,2)
prob =
    0.5000

>> x    = [0:0.1:10];
>> plot(x,unifdf('c',x,1,2),'b-',x,unifdf('c',x,3,7),'r-')
```

Density:

```
>> prob = unifdf('d',1.5,1,2)
prob =
    1.0000

>> x    = [0:0.01:10];
>> plot(x,unifdf('d',x,1,3),'b-',x,unifdf('d',x,1,4),'r-')
>> ylim([0 1])
```

Quantile:

```
>> prob = unifdf('q',0.5,1,2)
prob =
    1.5
```

Random:

```
>> prob = unifdf('r',[4 1],2,1)
ans =
    1.9218
    1.7382
    1.1763
    1.4057
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, weibulldf

weibulldf

Purpose

Weibull distribution.

Synopsis

```
prob = weibulldf(function,x,a,b)
```

Description

Estimates cumulative distribution function (cumulative, cdf), probability density function (density, pdf), quantile (inverse of cdf), or random numbers for a Weibull distribution.

This distribution is used to model lifetime data (time to failure). It is skewed to the right, but may appear symmetric for data in which there are relatively no small outcomes. Negative values in the sample are ignored.

$$f\left(x\right)=\left(bx^{b-1}/a^{b}\right)\exp\left[-\left(x/a\right)^{b}\right]$$

$$F\left(x\right)=1-\exp\left[-\left(x/a\right)^{b}\right]$$

INPUTS:

`function` =	[{'cumulative'} | 'density' | 'quantile' | 'random'], defines the functionality to be used. Note that the function recognizes the first letter of each string so that the string could be: ['c' | 'd' | 'q' | 'r'].
`x` =	matrix in which the sample data is stored, in the interval (-inf,inf). for function=quantile - matrix with values in the interval (0,1). for function=random - vector indicating the size of the random matrix to create.
`a` =	scale parameter (real).
`b` =	shape parameter (real and positive).

Note: If inputs (`x`, `a`, and `b`) are not equal in size, the function will attempt to resize all inputs to the largest input using the RESIZE function.

Note: Functions will typically allow input values outside of the acceptable range to be passed but such values will return NaN in the results.

Examples

Cumulative:

```
>> prob = weibulldf('c',2,1,2)
prob =
    0.9817

>> x    = [0:0.1:10];
>> plot(x,weibulldf('c',x,1,2),'b-',x,weibulldf('c',x,3,7),'r-')
```

Density:

```
>> prob = weibulldf('d',2,1,1)
prob =
    0.1353

>> x    = [0:0.1:10];
>> plot(x,weibulldf('d',x,2,1),'b-',x,weibulldf('d',x,0.5,1),'r-')
```

Quantile:

```
>> prob = weibulldf('q',0.5,1,2)
prob =
    0.8326
```

Random:

```
>> prob = weibulldf('r',[4 1],2,1)
ans =
    5.4812
    4.9755
    1.0562
    4.4820
```

See Also

betadr, cauchydf, chidf, expdf, gammadf, gumbeldf, laplacedf, logisdf, lognormdf, normdf, paretodf, raydf, triangledf, unifdf