MVAPACK, v2.1

An NMR and GC/LC-MS Chemometrics Toolbox for GNU Octave

Bradley Worley, Thao Vu, Chris Jurich

This manual is for the MVAPACK toolbox for GNU Octave.

Table of Contents

1 Overview of MVAPACK

This manual is written to be a comprehensive set of documentation that covers all important and commonly used functions in the *M*ulti*v*ariate *A*nalysis *Pack*age, or **MVAPACK**.

Chapter 1 [Overview], page 2, introduces MVAPACK and describes what it is written for, as well as quickly provides rationale for why certain design decisions were made during the creation of MVAPACK.

Chapter 2 [General concepts], page 5, introduces the most important concepts the user needs to be aware of while using MVAPACK, including how data is organized into structures, what different forms of data are called, *etc.*

Chapter 3 [MVAPACK patterns], page 12, presents the most commonly used functions in MVAPACK as programming *patterns* (code examples). The examples exhibit a recommended style for using MVAPACK that will greatly facilitate documentation and reproducibility of processing protocols.

Chapter 4 [NMR file loading], page 21, documents the functions used to load NMR data into the GNU Octave programming environment for further processing.

Chapter 5 [Data pre-processing], page 23, documents all functions that pertain to NMR data processing and manipulation. These functions are distinct from the functions in Chapter 6 [Data pre-treatment], page 31, which are mostly instrumentation-agnostic and serve a common goal of preparing data matrices for multivariate analysis.

Chapter 7 [Multivariate modeling], page 37, documents the functions used to build, visualize, validate and manipulate multivariate models, namely PCA, PLS, OPLS and LDA models.

Appendix A [Function index], page 56, lists all functions implemented in MVAPACK for reference purposes.

Appendix B [Example usage], page 91, contains a complete "*FIDs-to-models*" example MVAPACK script. The script is written in accordance with the recommended MVAPACK programming style shown in Chapter 3 [MVAPACK patterns], page 12.

1.1 What is MVAPACK?

MVAPACK is a free and open source package for GNU Octave that implements a powerful and flexible set of functions for 1D/2D NMR and GC/LC-MS chemometrics.

Most other chemometrics toolchains require multiple file format conversions, hopping between (possibly expensive) interactive software packages, and manual data manipulation in improper environments (e.g. Microsoft Excel). MVAPACK allows the data analyst to pull raw data files from the spectrometer all the way to validated and visualizable multivariate models, all within the GNU Octave programming environment.

1.2 Why GNU Octave?

GNU Octave is one of many domain-specific programming languages available for numerical computation. However, it boasts three main advantages that help it to stand apart in the context of NMR chemometrics and metabolomics.

- First, Octave is released under the GNU General Public License, an open source license.

- Second, Octave code is almost completely portable with MATLAB® code, giving it an advantage over other languages that share no syntactic commonalities with such a widely used and supported language.
- Finally, the Octave interpreter can execute natively on GNU/Linux, where many other NMR processing toolkits are also installed (*e.g.* NMRPipe, RNMRTK).

1.3 Why open source?

Providing MVAPACK to the community under an open source license was an important design decision early on. The GNU General Public License (GPL) protects the user's right to modify any part of the released software, so long as proper attribution and availability conditions are met. A package like MVAPACK should be completely available and open to the NMR chemometrics community for use, tinkering, extension, and so on. Releasing under the GPL ensures that this is so.

We want users who extend MVAPACK with new functions or amend the MVAPACK source code to feel free to send their contributions in as a patch. That way, additions and changes from all over the community can be released as part of the official MVAPACK package, available for all to use.

1.4 What am I waiting for?

We don't know! Get out there, start using MVAPACK and send us your new algorithms for inclusion into the next version release.

If you do have a new function, please provide a way for us to acknowledge your contribution in the function header (see any of the `*.m` files in MVAPACK for an example of how to do this). It's especially handy if you can also include a citation to the work that the code is based on. Also, remember that the code has to be released under the GNU GPL.

We hope you enjoy MVAPACK and find it useful.

1.5 Release updates

MVAPACK v2.1 features a variety of compatibility improvements as well as expansion of analytical functionality. Note that v2.1 is an incremental update and there will be another release later in 2021 that features further improvements for the GC/LC-MS capabilities.

To improve compatability, a number of deprecated API calls have been updated. These have no impact on functionality for the end-user but ensure that the software will be compatible with newer versions of Octave, namely v6. Likewise, it eliminates a number of existing compilation errors that used to occur during the build process.

This release also features significant expansion of GC/LC-MS data processing abilities. The newly added function enable users to build extracted ion chromatograms (EICs) from a variety of industry-standard file formats before performing peak picking and a number of data matrix preparation steps for use in principal component analysis (PCA) modelling. A complete list of the newly added functions are below:

- `baseline_noise_estimation()`
- `baseline_subtraction()`
- `create_eics()`

- `efficient_filter()`
- `gauss_baseline_noise()`
- `gauss_deriv_peak_picking()`
- `gaussian_coefficients()`
- `impute()`
- `impute_mean()`
- `load_peaks()`
- `pick_peaks()`
- `prepare_eic()`
- `quantile_normalize()`
- `save_peaks()`

The upcoming release will provide additional, more sophisticated methods for data preprocessing and matrix generation for GC/LC-MS datasets.

2 General concepts

The functions in MVAPACK follow a set of conventions for data structures that are important to be aware of during data handling. This chapter introduces and explains those conventions in detail.

2.1 Preliminary

This section describes how to download and install MVAPACK for the first time.

MVAPACK has been tested and verified to work properly on Ubuntu 20.04 , Debian 9 ("Stretch"), Centos7, and Mac OS X. GNU Octave version 4 or later is required to install MVAPACK, though version 6 is recommended when possible.

NOTE: The [loadnmr], page 71, function in MVAPACK requires `NMRPipe` to be installed, so make sure you have `NMRPipe` installed and configured before installing MVAPACK.

2.1.1 Required Octave Packages

2.1.1.1 Installing Octave

Installing GNU Octave is truthfully outside the scope of this documentation. Every system has a different means of installing Octave, but none makes it too terribly difficult. The best place to start reading about how to install Octave is the Octave download page (`https://www.gnu.org/software/octave/download.html`).

2.1.1.2 Installing Packages

However, once Octave is installed, there are packages you *must* install before all MVAPACK functions will work properly. It is recommended that these packages are installed by the root user, so every user on the machine can use them. Open a terminal and `su` into the root account, then install the following packages: *image*, *optim*, *signal*, and *io*.

For example:

```
$ su -
# octave
octave:1> pkg install -auto -forge image
octave:2> pkg install -auto -forge optim
octave:3> pkg install -auto -forge signal
octave:4> pkg install -auto -forge io
octave:4> pkg install -auto -forge econometrics
octave:5> exit
# exit
```

The above example shows how to install Octave packages directly from octave-forge, but that's only one way to do it. Many distributions (Debian, as a concrete example) provide stable versions of Octave packages inside their package repositories. So, in the case of a Debian system, the following commands are recommended instead of the above:

```
$ sudo apt-get install octave-image octave-optim        \
                       octave-signal octave-io          \
                       octave-econometrics
```

See the Octave manual for more detailed instructions on installing packages.

2.1.2 Downloading MVAPACK

The latest version of MVAPACK is available (along with the latest version of this manual) at bionmr.unl.edu (`http://bionmr.unl.edu/mvapack.php`). It is available as a `tar.gz` file that may be installed via the Octave package management system.

Save the `tar.gz` file to a location you can easily navigate to from a terminal, `/tmp` for example.

2.1.3 Installing MVAPACK

It is recommended that MVAPACK is installed by the root user, so every user on the machine can use the package. Open a terminal and `su` into the root account, then move to the directory in which the `tar.gz` file was saved.

Once in the proper directory, start an Octave interactive session and install the MVAPACK package.

As an example:

```
$ su -
# cd /tmp
# octave
octave:1> pkg install -auto mvapack-YYYYMMDD.tar.gz
octave:2> exit
# exit
```

For more information, see the Octave manual for more detailed instructions on installing packages.

2.1.4 Using MVAPACK

Once MVAPACK is installed, any Octave session opened will automatically load in the MVAPACK functions on startup. So using MVAPACK is as easy as running Octave, from *any* directory:

```
$ octave
octave:1> ... MVAPACK is ready to use! ...
```

2.2 Definitions

Certain terms in multivariate analysis are utilized in this manual, as shown below.

2.2.1 Observations

Observations are defined as individual measurements in either a univariate or multivariate space. Observations in MVAPACK are most typically 1D NMR spectra or binned spectra.

Operations that manipulate the magnitudes of observations are most often referred to as Section 6.2 [Normalization], page 32, operations.

2.2.2 Variables

Variables are defined as individual components of a measurement that may be taken. In MVAPACK, variables are typically either single data points in full-resolution 1D NMR data or bins in binned data.

Operations that manipulate the magnitudes of variables are most often referred to as Section 6.3 [Scaling], page 33, operations.

2.2.3 Data vectors

Data vectors in MVAPACK are *column vectors* and either store an abscissa (*t*, *ppm*, *idx*, *etc.*) or a single observation. Data vectors may be real (as is the case of abscissas) or complex.

For example, a single (uniformly sampled) free induction decay in MVAPACK is fully represented by two data vectors: the abscissa t and the observation f.

2.2.4 Data matrices

Data matrices in MVAPACK store multiple observations as stacked *row vectors* and are typically paired with an abscissa vector. Data matrices may be real (real spectra or bins) or complex (free induction decays or complex spectra).

For example, a set of free induction decays in MVAPACK are represented by the abscissa data vector t and the data matrix F.

2.2.5 Multiblock data matrices

Multiblock data in MVAPACK is a collection of multiple data matrices, which may have different numbers of variables but must have the same number of observations. A common example of multiblock data is when spectral data from two different instrumental sources (*e.g.* NMR and MS) need to be modeled jointly. The [mbpca], page 41, [mbpls], page 41, and [mbopls], page 42, algorithms are all designed to handle multiblock data matrices.

2.2.6 Response matrices

Response matrices are a special type of data matrix that hold 'outputs', and are represented by the symbol Y. Response matrices fall into two categories: continuous values and class labels. Response matrices composed of continuous values (*e.g.* activities, absorbances, or other measureables) are used in the context of regression (*i.e.* PLS-R, OPLS-R, *etc.*). Class matrices (in MVAPACK) are composed only of ones and zeros, where a one in row i and column m indicates that the i-th observation in X belongs to the m-th class. For more information on using class matrices, see [classes], page 52, and [loadlabels], page 52.

2.2.7 Scores and loadings

For every component of a PCA, PLS or OPLS model built by MVAPACK, at least one 'score' vector and one 'loading' vector are produced. Conventionally, score vectors are column vectors designated by t_a and loading vectors are column vectors designated by p_a, where a is the component to which the scores and loadings belong. Scores and loadings come in pairs and their outer product results in a rank-one matrix that holds fitted data of a single component. The sum of all score-loading outer products results in a low-rank approximation of the input data.

As an example, a three-component ($A = 3$) PCA model may be written as follows:

$$X = TP' + E = [t_1 \;\; t_2 \;\; t_3][p_1 \;\; p_2 \;\; p_3]' + E = t_1p_1' + t_2p_2' + t_3p_3' + E$$

If the input matrix X has N observations and K variables, each score vector will contain N elements and each loading vector will contain K. The residuals matrix E holds data in X not fitted by the model TP'.

The data contained in scores and loadings will differ based on the objective of the algorithm. Regardless, scores are considered low-dimensionality representations of observations and loadings are considered low-dimensionality representations of variables. Thus, scatter plots of scores will show how observations are related based on data trends, and scatter plots of loadings will show how/which variables contribute to those trends.

2.2.8 Internal cross-validation

All models in MVAPACK undergo 'leave n out' internal cross validation while they are being calculated. This procedure randomly divides the input matrices into a training set and a validation set. A cross-validation model is built to match the current actual model, and the degree of fit and quality of prediction (of the training set) are both evaluated to yield the R^2 and Q^2 statistics, respectively.

The R^2 statistic ranges from 0 to 1 and quantifies the fraction of variation is explained by the model. High R^2 values indicate that the model captures a large amount of variation present in the input data.

The R^2 statistic is calculated (for X, in this example) as follows:

$$R^2_X = \frac{\sum_{i=1}^{N}\sum_{k=1}^{K}(TP')^2_{ik}}{\sum_{i=1}^{N}\sum_{k=1}^{K}x^2_{ik}}$$

In short, R^2 is the ratio of the summed row sums of squares of TP' and X.

The Q^2 statistic ranges from $-\infty$ to 1 and quantifies the capability of a model to predict it's own data. For PCA models, Q^2 indicates how well the model predicts X. For PLS and OPLS models, Q^2 indicates how well the model predicts Y.

The Q^2 statistic is calculated as follows:

$$Q^2 = 1 - \frac{\sum_{i=1}^{N}\sum_{m=1}^{M}\left(Y - Y_{CV}\right)^2_{im}}{\sum_{i=1}^{N}\sum_{m=1}^{M}Y^2_{im}}$$

In short, Q^2 is the ratio of the summed row sums of squares of the cross-validation predicted Y-residual $(Y - Y_{CV})$ and Y.

The number of components a given model will have for any input dataset is largely dependent on the above cross-validation statistics above. All methods in MVAPACK require per-component R^2 statistics greater than 0.01 and cumulative per-component Q^2 statistics that are strictly increasing. (*i.e.* no individual component may have a negative Q^2). Aside from that, the number of model components is limited to half the observation count or half the variable count, whichever is less.

See [rq], page 48, [rqdiff], page 48, and [rqplot], page 48, for more information on internal cross-validation statistics in MVAPACK.

2.2.9 Permutation testing

While internal cross-validation metrics are a good first insight into model quality, supervised models from PLS or OPLS require more rigorous validation before they can be considered useful. Response permutation testing, or simply *permutation testing*, is one such validation technique that provides evidence of a model's reliability.

Response permutation testing randomly shuffles the rows of the response matrix Y and rebuilds a new (O)PLS model with the same number of components as the model under test. This procedure is repeated over 100 – 1000 iterations, resulting in a distribution of R^2 and Q^2 statistics that represent the null hypothesis: an invalid or overfit model. The R^2 and Q^2 statistics from the model under test may then be compared to this distribution to determine model significance.

See [permtest], page 49, [permscatter], page 48, and [permdensity], page 48, for more information on response permutation testing in MVAPACK.

2.2.10 CV-ANOVA testing

The CV-ANOVA validation technique relies on the residuals from internal model cross-validation to determine model significance. By comparing the cross-validated fit sum of squares to the cross-validated total sum of squares using an F-test, model validity can be measured.

See [cvanova], page 48, for more information on CV-ANOVA testing in MVAPACK.

2.2.11 More information

Every command in MVAPACK is provided with documentation on its use. You can see the documentation of a function (*e.g.* `fn_name`) by running `help fn_name` inside an Octave interactive session.

Learning how to harness the maximum potential of MVAPACK honestly requires learning how to use GNU Octave (`http://www.gnu.org/software/octave/`). Complete documentation of the GNU Octave programming language is available at the Octave website (`http://www.gnu.org/software/octave/doc/interpreter/index.html`).

2.3 Organization

2.3.1 Data vector organization

Data vectors are organized as column vectors. In Octave, vector indices begin at `1` and end at the number of vector elements. Take the example of a single free induction decay (t, f):

```
% just loading in a single spectrum, for example.
[f, parms, t] = loadnmr('my-spectra-01/1');

% total number of data points (real+imag).
parms.td
⇒ 16384

% size of the abscissa that pairs with f.
size(t)
⇒ 8192 1

% the abscissa is real.
iscomplex(t)
⇒ 0
```

```
% size of the free induction decay data vector.
size(f)
⇒ 8192 1

% the free induction decay is complex.
iscomplex(f)
⇒ 1
```

Single or multiple vector elements may be accessed using the vector indices. See the GNU Octave documentation for more information on how to use indices.

Here are a few quick examples of vector indices:

```
f(1) *= 0.5;              % scale down the first point.
f(1:10) -= mean(f(1:10)); % mean-center the first ten points.
f(end / 2:end) *= 0;      % zero the second half of the vector.
```

2.3.2 Data matrix organization

Data matrices are organized as sets of observation row vectors. The row (first) index of a data matrix is the observation index and the column (second) index of a data matrix is the variable index, so `X(i,k)` corresponds to the i-th observation and k-th variable of the data matrix X. Take the example of a set of free induction decays (t, F):

```
% just loading in a set of spectra, for example.
[F, parms, t] = loadnmr(glob('my-spectra-??/1'));

% size of the free induction decay data matrix.
size(F)
⇒ 50 8192

% the data matrix is complex, of course.
iscomplex(F)
⇒ 1
```

Accessing elements of data matrices can take any number of different shapes and sizes. The GNU Octave documentation fully details how to index matrices.

Here are a few quick examples of matrix indices:

```
% extract the first decay into a data vector.
f = F(1,:)';

% scale down the first point of every decay.
F(:,1) *= 0.5;

% mean-center all points of the first three decays.
F(1:3,:) -= ones(3,1) * mean(F(1:3,:));

% zero the second half of the first, third and fifth decays.
F([1,3,5],end / 2:end) *= 0;
```

2.3.3 Multiblock organization

Multiblock data matrices are organized as cell arrays which contain at least two data matrices. For example, if two data matrices X_1 and X_2 have been created, and both matrices have ten observations, the multiblock data matrix may be constructed using [multiblock], page 73, like so:

```
X = multiblock(X1, X2);
```

Now, X is a cell array that contains the matrices X_1 and X_2, and is ready for further processing, *e.g.* by [mbpca], page 41.

2.3.4 Structure organization

MVAPACK makes extensive use of structures to organize outputs of functions that would be completely unwieldy without doing so. Multivariate models, acquisition parameters, cross-validation results, and other forms of data are all packed into structures, to name a few.

Data inside structures is accessed by structure fields. For example, if a model *mdl* contains a field A, it is accessible as `mdl.A`.

Here is an example of accessing fields inside a model structure:

```
% build a model.
mdl = pca(X);

% number of model components.
mdl.A
⇒ 3

% total model fit sum of squares.
mdl.Rsq.X.cum
⇒ 0.92

% total cross-validated fit sum of squares.
mdl.Qsq.cum
⇒ 0.90
```

3 MVAPACK patterns

MVAPACK is designed to be flexible, but common patterns exist in its use. The following code segments detail recommended ways to accomplish the most common tasks performed within the MVAPACK software.

The discerning reader will notice a liberal use of structures in the following code examples. Organizing data into structures simplifies the task of saving all relevant Octave session data for re-use later. Use of structures also serves as a means of documenting one's data processing for reproduction later.

3.1 Loading NMR data

3.1.1 Loading Bruker DMX data without zero-filling

This code is an example of how to load free induction decay data from files generated by older versions of Topspin or XWIN-NMR. The command will read the directories that match the filename glob `my-data-???/1` in the current directory and read the `fid` and `acqus` data from each matching directory. The decays will then be automatically group delay corrected.

```
F.dirs = glob('my-data-???/1');
[F.data, F.parms, F.t] = loaddmx(F.dirs);
```

3.1.2 Loading Bruker DMX data with zero-filling

This code is similar to the above example, but the DMX free induction decays are not group delay corrected during loading. Instead, two zero fills are performed and then the group delay correction is done. The filename glob `my-data-*/1` also differs from the previous example: the splat (`*`) acts as a multiple-character wildcard, while the question mark (`?`) acts as a single-character wildcard.

```
F.nzf = 2;
F.dirs = glob('my-data-*/1');
[F.data, F.parms, F.t] = loaddmx(F.dirs, false);
F.data = zerofill(F.data, F.parms, F.nzf);
F.data = dmxcorr(F.data, F.parms);
```

If apodization is also desired, it must be done prior to both zero-filling *and* group delay correction:

```
F.nzf = 1;
F.dirs = glob('my-data-*/1');
[F.data, F.parms, F.t] = loaddmx(F.dirs, false);
F.data = apodize(F.data, F.parms, @expwindow);
F.data = zerofill(F.data, F.parms, F.nzf);
F.data = dmxcorr(F.data, F.parms);
```

3.1.3 Loading more modern Bruker or Agilent data

Free induction decay data collected on more recent spectrometers may be loaded using [loadnmr], page 71. Group delay correction is automatic.

```
F.dirs = glob('my-data-???/1');
[F.data, F.parms, F.t] = loadnmr(F.dirs);
```

If the loaded free induction decay just looks like intense noise, you may need to byte-swap the data:

```
[F.data, F.parms, F.t] = loadnmr(F.dirs, true);
```

For more information, see [loadnmr], page 71.

3.1.4 Loading classes and labels

Samples collected under automation may not be grouped by class identity during data collection, resulting in the members of each class being randomly distributed within the rows of a dataset.

Before such data matrices are used in MVAPACK, they must be reordered to group the observations by class. The simplest way to do this is to make a text file (e.g. `my-class-labels.txt`) that defines the textual class name of each observation on each line, like so:

```
WT
WT
K2A
A6P
WT
K2A
A6P
K2A
...
```

In MVAPACK, the class labels, indices and identity matrix may be loaded using [loadlabels], page 52. Once these are loaded, the data matrix `F` can be reordered to group observations by class. This is an essential step if the input data is 'out of order'.

```
cls.filename = 'my-class-labels.txt';
[cls.labels, cls.indices, cls.Y] = loadlabels(cls.filename);
F.data = F.data(cls.indices, :);
```

3.2 Processing in the time domain

3.2.1 General apodization

Apodization of free induction decays is performed by passing a line-broadening factor (*lb*, default is 0.3 Hz) to [expwindow], page 24. Like the case of zero-filling, DMX data loaded with [loaddmx], page 22, must be corrected by [dmxcorr], page 22, **after** apodization.

```
F.lb = 0.3; % hz.
F.apod = @expwindow;
F.data = apodize(F.data, F.parms, F.apod, F.lb);
```

Or, if a 0.3 Hz Gaussian apodization is desired:

```
F.lb = 0.3; % hz.
F.apod = @gausswindow;
F.data = apodize(F.data, F.parms, F.apod, F.lb);
```

3.2.2 General zero-filling

The following code is an example of how to zero-fill free induction decay data. In this example, two zero-fills are performed (`nzf = 2`).

```
F.nzf = 2;
F.data = zerofill(F.data, F.parms, F.nzf);
```

3.2.3 General Fourier transformation

Fourier transformation is performed with [nmrft], page 25. An accompanying chemical shift axis `S.ppm` will also be generated.

```
[S.data, S.ppm] = nmrft(F.data, F.parms);
```

A centered Hertz axis may also be returned, if you want to deal with that type of frequency unit:

```
[S.data, S.ppm, S.hz] = nmrft(F.data, F.parms);
```

3.3 Processing in the frequency domain

3.3.1 Removal of undesirable observations

Observations may be removed by their index in a data matrix, using the function [rmobs], page 31. The example below removes the first five and the twenty-third observations from the data matrix *F*.

```
F.rm.obs = [1 : 5, 23];
F.data = rmobs(F.data, F.rm.obs);
```

The same pattern may be performed on any data matrix, such as a set of spectra:

```
S.rm.obs = [1 : 5, 23];
S.data = rmobs(S.data, S.rm.obs);
```

3.3.2 Automatic phasing and normalization

Phase correction and normalization are interrelated, and their correction is ideally performed using an algorithm that simultaneously handles both. The following code is an example of the recommended way to automatically phase and normalize NMR spectra. First, automatic phasing of each spectrum is performed with [autophase], page 26. Then, [pscorr], page 33, phases and normalizes the dataset to bring it into the best possible agreement with its mean:

```
[S.data, S.phc0, S.phc1] = autophase(S.data, F.parms);
[S.data, S.normfactor] = pscorr(S.data);
```

3.3.3 Extraction of real spectral data

Once phase correction is finished, a purely real data matrix can be made. Most algorithms following phase correction expect the data matrix to not contain complex numbers, which is why this step is done.

```
X.data = realnmr(S.data, F.parms);
X.ppm = S.ppm;
```

3.3.4 Chemical shift referencing

The simplest form of spectral referencing is to modify only the chemical shift axis using [refadj], page 27. The following example shows how to shift the abscissa from -0.15 ppm to 0.00 ppm to produce referenced spectra.

```
X.ref.old = -0.15;
X.ref.new = 0.00;
X.ppm = refadj(X.ppm, X.ref.old, X.ref.new);
```

3.3.5 Spectral alignment

Following chemical shift referencing, variations in peak chemical shifts not caused by global reference offsets may be corrected with [icoshift], page 29.

```
X.data = icoshift(X.data, X.ppm);
```

3.3.6 Removal of undesirable variables

Removing variables is done in a similar manner to removing observations, but now the chemical shift axis must also be modified. The following example removes the most upfield and downfield portions of the spectra as well as the residual water signal (See [rmvar], page 31).

```
k0 = findnearest(X.ppm, min(X.ppm));
k1 = findnearest(X.ppm, 0.2);
k2 = findnearest(X.ppm, 4.6);
k3 = findnearest(X.ppm, 4.8);
k4 = findnearest(X.ppm, 8.5);
k5 = findnearest(X.ppm, max(X.ppm));
X.rm.var = [k5 : k4, k3 : k2, k1 : k0];
[X.data, X.ppm] = rmvar(X.data, X.ppm, X.rm.var);
```

3.3.7 Adaptive intelligent binning

Binning is an optional step, provided spectral alignment was performed. If binning is to be done, use [binadapt], page 30.

```
[B.data, B.ppm, B.widths] = binadapt(X.data, X.ppm, F.parms);
```

3.4 Handling multivariate models

3.4.1 Building unsupervised models

PCA models are built using [pca], page 37. Depending on the number of data matrix rows and columns, calculation times can range from less than a second (N=50, K=200) to an hour (N=150, K=32768). Once the model is successfully built, the class identity matrix and class labels may be added using [addclasses], page 49, and [addlabels], page 50.

```
pcaMdl = pca(X.data);
pcaMdl = addclasses(pcaMdl, cls.Y);
pcaMdl = addlabels(pcaMdl, cls.labels);
```

3.4.2 Building supervised models

While unsupervised models require only a data matrix, supervised models also need a response matrix. The response matrix Y may be a second set of continuous values (regression) or a class identity matrix (discriminant analysis). The following example builds a model with [opls], page 39. Class labels are then also added.

```
oplsMdl = opls(X.data, cls.Y);
oplsMdl = addlabels(oplsMdl, cls.labels);
```

3.4.3 Validating supervised models

Supervised models (*e.g.* PLS, OPLS) require rigorous validation to ensure the reliability of any conclusions drawn from the modeled data. The simplest means of validation is done with [rqplot], page 48:

```
rqplot(mdl);
```

Still more complicated is to run response permutation testing (See [permtest], page 49):

```
mdl.cv.perm = permtest(mdl);
permscatter(mdl.cv.perm);
```

Finally, the CV-ANOVA test (See [cvanova], page 48) may be performed to analyze the fitted residuals from internal cross-validation:

```
mdl.cv.anova = cvanova(mdl);
mdl.cv.anova
```

The lack of a semicolon in the second line of the above example is not a typographical error! If you want to see the results of the CV-ANOVA test, you have to leave it off.

3.4.4 Building models with different scaling

The default scaling function during PCA, LDA and PLS model building is [suv], page 34, and the default function for OPLS is [spareto], page 34. If you want to change the scaling function used, pass another argument to the `pca`, `lda`, `pls` or `opls` function that holds the function handle of the scaling method you wish to use. For example:

```
pcaMdlWithLevelScaling = pca(X.data, @slevel);
```

All available scaling methods are discussed in Section 6.3 [Scaling], page 33.

3.4.5 Building multiblock structures

Multiblock PCA (See [mbpca], page 41), multiblock PLS (See [mbpls], page 41), and multiblock OPLS (See [mbopls], page 42) all require a multiblock array as input, instead of a matrix. Given two data matrices A and B, creation of a multiblock array is simple:

```
X = multiblock(A, B);
```

Provided all matrices have the same number of observations (rows), as many data matrices may be included in the multiblock array as desired:

```
X = multiblock(Xhnmr, Xlcms, Xftir, Xclinical, ...);
```

3.4.6 Handling multiblock models

Multiblock models contain nested "block models" which individually approximate their respective data matrix in the multiblock array, but also relate to the other block models via

the constraints of the algorithm. For example, if an MB-PLS model is built on a three-block array:

```
X = multiblock(X1, X2, X3);
mdl = mbpls(X, Y);
```

It will contain super-scores and loadings, as well as three block models. Therefore, many commands (which normally operate on the model as a whole) will function on the blocks as well:

```
rqplot(mdl);
scoresplot(mdl.blocks{1});
backscaleplot([], mdl.blocks{2});
T3 = scores(mdl.blocks{3}, 2);
```

3.5 Plotting model results

3.5.1 Model quality plots

The PCA, PLS and OPLS modeling functions in MVAPACK automatically produce R^2 and Q^2 metrics during fitting. These values can be plotted to quickly determine how well the model fits the data (R^2) and how well the model predicts the data (Q^2):

```
rqplot(mdl);
```

For models having at least two components, the blue bars indicate cumulative R^2 values and the green bars indicate cumulative Q^2 values.

3.5.2 Scores scatter plots

Scores scatter plots may be generated in the following way:

```
scoresplot(mdl);
```

However, the `scoresplot` function may not be able to decide whether to build a 2D or 3D plot, in which case the number of plot dimensions must be passed as a second argument, *e.g.*:

```
scoresplot(mdl, 2);
```

If observation numbers are desired instead of points in the scores plot, they may be enabled by passing *numbers* as `true` to `scoresplot`:

```
scoresplot(mdl, [], [], true);
```

See [scores], page 50, and [scoresplot], page 46, for more information on plotting scores.

3.5.3 Loadings scatter plots

Two-dimensional loadings scatter plots may be generated in the following way:

```
loadingsplot(mdl);
```

If variable numbers are desired instead of points in the scores plot, they may be enabled by passing *numbers* as `true` to `loadingsplot`:

```
loadingsplot(mdl, [], true);
```

See [loadings], page 50, and [loadingsplot], page 46, for more information on plotting loadings.

3.5.4 Loadings line plots

Loadings line plots are generated using [backscaleplot], page 46, in the following way:

```
backscaleplot(X.ppm, mdl);
```

If the line plots need to be colored according to model weighting, a second argument is passed to the function:

```
backscaleplot(X.ppm, mdl, true);
```

The abscissa vector passed to `backscaleplot` must match the data passed to the model building function (*i.e.* `mdl` above was generated based on `X.data`).

Finally, the backscaled loadings may be stored into a matrix for plotting in a more efficient environment, like gnuplot:

```
G = backscaleplot(X.ppm, mdl, true);
save -ascii 'G.txt' G
```

The columns of `G` will be `X.ppm`, p_1, p_2, ... p_A, w. Here, w is the Z-factor by which the data was scaled prior to multivariate modeling. A gnuplot script to plot a single-component OPLS loading would contain the following statements:

```
unset key
set cbrange [0 : 1]
set palette rgbformula 30, 13, 10
plot 'G.txt' with lines linecolor palette
```

3.5.5 S-plots

S-plots are generated for OPLS models using [splot], page 47, in the following way:

```
splot(mdl);
```

If variable numbers are desired instead of points in the S-plot, they may be enabled by passing *numbers* as `true` to `splot`:

```
splot(mdl, [], true);
```

3.6 Plotting data matrices

3.6.1 Plotting data matrices overlaid

Plotting all observations in a data matrix is a commonly performed task during interactive data processing. The following example shows how to plot the rows of a time-domain data matrix, overlaid.

```
plot(F.t, realnmr(F.data', F.parms));
```

Spectral data matrices may be plotted in the same manner:

```
plot(S.ppm, realnmr(S.data', F.parms));
```

Real data matrices are also plotted in the same way, but without need for the `realnmr` function:

```
plot(X.ppm, X.data');
```

3.6.2 Plotting data matrices stacked

Stack plots of data matrices are also possible through the following example. Unlike the overlaid command, stack plots require no `realnmr` invocation or matrix transposition.

```
stackplot(F.data);
```

More information on stacked plots is available in [stackplot], page 46.

3.6.3 Plotting to aid phase correction

Manual phase correction is performed using [phase], page 26. This function simply accepts zero-order and first-order phase correction values and returns the corrected spectral data.

Because manual phase correction is an iterative process, a quick way of guess-and-check must be used by plugging in values for *phc0* and *phc1* below:

```
plot(S.ppm, phase(S.data, F.parms, phc0, phc1)');
```

Once a suitable result is found, the values of *phc0* and *phc1* can be used to actually correct the spectral data matrix:

```
S.phc0 += phc0;
S.phc1 += phc1;
S.data = phase(S.data, F.parms, phc0, phc1);
```

3.6.4 Plotting multiblock data matrices

The individual blocks of multiblock data matrices may be accessed from their multiblock array, like so:

```
plot(ab, X{1});
```

3.7 Saving MVAPACK data

3.7.1 Saving session data for later

The following code is an example of how to save relevant data from an open Octave session to a file for use later. Data matrices and models are stored as fields of a main structure, which may be saved to a file.

```
savedate = date();
save('-binary', '-z', 'my-session.dat.gz');
```

3.7.2 Loading saved session data

Loading a session back from a file is as simple as the following example:

```
load('my-session.dat.gz');
```

3.7.3 Saving data matrices to text

It is sometimes necessary to output data matrices from Octave for plotting or analysis in other programs. An easy way to save a data matrix and it's associated abscissa is as follows:

```
data = [t, real(F')];
save -ascii 'data.txt' data
```

Saving data to comma-separated values format ("CSV") is even easier:

```
data = [ppm, real(S')];
csvwrite('data.csv', data);
```

3.7.4 Loading data matrices from text

Loading data matrices back from text files is also sometimes required. The recommended way to do so follows:

```
data = load('data.txt');
t = data(:,1);
Fr = data(:,2:end)';
```

Notice in the above command that the data matrix *Fr* is no longer complex.

Loading data from comma-separated values format ("CSV") is accomplished through the `csvread` command:

```
data = csvread('data.csv');
ppm = data(:,1);
Sr = data(:,2:end);
```

Again, the reloaded data in *Sr* is no longer complex. Saving complex data to text requires a bit more conjuring:

```
data = [ppm, real(S'), imag(S')];
save -ascii 'data.txt' data
  ...
data = load('data.txt');
ppm = data(:,1);
S = complex(data(:,2:(end-1)/2+1), data(:,(end-1)/2+2:end));
```

In general, it's easier and faster to save the whole session (See Section 3.7.1 [Saving session data for later], page 19) than to go through the above hassle.

3.7.5 Saving plots to postscript files

Saving plots into the Enhanced PostScript file format (`*.eps`) may be accomplished using the `print` command. First, plot the data you wish to save to file. Then run the following command:

```
print -deps -color 'myplot.eps'
```

The function may give some warnings, but if it returns without errors your figure should be saved into the file.

Saving three-dimensional plots to files has an additional twist, however, because 3D plots are rotatable. Once a suitable orientation has been found, record the numbers in the bottom left corner of the plot window, which will look something like:

```
view: 35, 310
```

Before running the `print` function to save the plot, run this:

```
view(310, 90 - 35);
```

The proper orientation will now be saved such that the `print` function will save the plot correctly. Remember, don't literally use *35* and *310*; use the values shown in the corner of the plot window.

Plots may also be saved using the `saveas` function if the plot package is installed into Octave:

```
saveas(gca(), 'myplot.eps', 'eps');
```

4 NMR file loading

The first step in any data processing procedure is to load the actual data. The functions documented in this chapter do exactly that.

4.1 Loading Bruker or Agilent FID data

Data collected from Agilent and modern Bruker spectrometers is best loaded with the [loadnmr], page 71, function, which will automatically detect as many spectral parameters as possible to ensure seamless loading into MVAPACK.

`f = loadnmr (dirname)` [Function File]
`[f, parms] = loadnmr (dirname)` [Function File]
`[f, parms, t] = loadnmr (dirname)` [Function File]
`[f, parms, t] = loadnmr (dirname, doswap)` [Function File]
`F = loadbruker (dirnames)` [Function File]
`[F, parms] = loadnmr (dirnames)` [Function File]
`[F, parms, t] = loadnmr (dirnames)` [Function File]
`[F, parms, t] = loadnmr (dirnames, doswap)` [Function File]

Loads one or more Bruker or Agilent fid/ser files, automatically extracting parameters and automatically determining whether to parse Bruker or Agilent data.

Some older data needs to be byte-swapped when it is loaded in. To do this, pass *doswap* as `true` to this function. The default behavior is to skip the byte swap.

The parameters can be optionally returned if a second return value is requested. If a third optional return value is requested, the time abscissa (t) will be returned.

NOTE: Extreme care must be taken to ensure that the acquisition parameters of all experimental data specified in *dirnames* are totally identical! Only one parameter structure (*parms*) will be returned, so it is assumed that the first experiment holds parameters that are representative of the entire dataset.

`p = acquparms (dirname)` [Function File]

Reads values from the key-value pairs found in Bruker 'acqus' files and Varian/Agilent 'procpar' files. The type of acquisition parameters to parse is determined automatically.

The output p is a cell array that contains a structure for each dimension present in the spectral data. Each element (dimension) in p contains information relevant to reconstructing time- and frequency-domain axes (along that dimension) for the spectral intensities:

Universal:
`p.td`: number of complex points.
`p.obs`: transmitter base frequency.
`p.car`: carrier offset frequency.
`p.sw`: spectral width. `p.dim`: dimension.

Bruker only:
`p.decim`: decimation ratio.
`p.grpdly`: estimated group delay.

`p.dspfvs`: dsp firmware version.

4.2 Bruker-format data

The [loaddmx], page 22, function is best for older Bruker data that has no group delay correction information contained in its `acqus` file.

`f = loaddmx (dirname)` [Function File]
`[f, parms] = loaddmx (dirname)` [Function File]
`[f, parms, t] = loaddmx (dirname)` [Function File]
`[f, parms, t] = loaddmx (dirname, correct)` [Function File]
`F = loaddmx (dirnames)` [Function File]
`[F, parms] = loaddmx (dirnames)` [Function File]
`[F, parms, t] = loaddmx (dirnames)` [Function File]
`[F, parms, t] = loaddmx (dirnames, correct)` [Function File]

Loads one or more 1D Bruker DMX-format fid files, automatically extracting parameters. The parameters can be optionally returned if a second return value (*parms*) is requested. A third optional return value, *t*, can be requested that will contain the time-domain abscissa.

An optional second input argument *correct* may be supplied to enable or disable group delay correction. The default behavior is to correct for group delay, but if you plan on zero-filling or apodizing, you need to postpone the correction until after the filling operation. *F* may then be corrected explicitly for group delay (See [dmxcorr], page 22).

`Fcorr = dmxcorr (F, parms)` [Function File]

Corrects Bruker DMX-format for group delay issues. This function is usually called automatically (by [loaddmx], page 22) without the user having to, but may have to be used if zero filling or apodizing must be applied before correction.

5 Data pre-processing

Pre-processing encompasses all operations that are needed to take the raw instrumental data to properly 'sanitized' final instrumental data. In other words, pre-processing is specific to the type of instrumentation, NMR in this case.

The following sections document the important pre-processing functions that are available for use in MVAPACK.

5.1 NUS reconstruction

Two-dimensional spectra that have been nonuniformly sampled may be easily reconstructed in MVAPACK using iterative soft thresholding (IST). The IST routine must be performed prior to any other time-domain processing using the [nmrist], page 23, function.

`[recfid] = nmrist (fid, parms)` [Function File]
`[recfid] = nmrist (fid, parms, phc)` [Function File]

Performs Iterative Soft Thresholding (IST) reconstruction of a nonuniformly sampled 2D NMR time domain matrix or a 2D NMR time domain cell array. Two-dimensional data must be arranged with slices of the direct-dimension along the rows.

The product of this operation is a uniformly sampled data structure that may be processed in the same way as any other 2D data.

An optional third argument, *phc*, may be passed as a two-element vector that supplies manual phase correction values for the direct dimension. If *phc* is not provided or is empty, automatic phase correction will be applied during reconstruction.

5.2 Apodization

Time-domain multiplication of any given window function with a set of free induction decays will result in the spectra having been convolved with that window function's impulse response function. Use of different window functions results in different peak shapes in the output spectra.

`wfid = apodize (fid, parms)` [Function File]
`wfid = apodize (fid, parms, fn)` [Function File]
`wfid = apodize (fid, parms, fn, opts)` [Function File]

Performs apodization of a time-domain NMR free-induction decay in order to alleviate truncation artifacts that can arise from Fourier transformation.

In the one-dimensional case, data in *fid* may either be a column vector or a data matrix where each free induction decay is arranged as a row in the matrix.

In the two-dimensional case, data in *fid* may either be a data matrix where each direct-dimension slice is along the rows, or a cell array that contains multiple matrices, each having direct-dimension slices along its rows.

A parameter structure (or array) must be passed as a second argument. The third argument, *fn*, is used to specify the type of apodization to use ([expwindow], page 24, [gausswindow], page 24, [sinewindow], page 24). The default is an exponential window function. An optional fourth argument (*opts*) may be passed that holds the parameters needed by the apodization function *fn*.

Apodization is most commonly of the exponential kind in 1D NMR, which effects a Lorentzian lineshape and is used to ensure the FID has decayed to zero by the end of the acquisition.

Window functions available to [apodize], page 23, are as follows:

`w = expwindow (t, lb)` [Function File]

Calculate window coefficients for exponential windowing of a signal. The parameter *lb* is the line-broadening factor (in Hz) to apply to the signal having time points in *t*.

The default line-broadening factor is 0.3 Hz.

The equation applied to the free induction decay is as follows:

$$wfid(t) = fid(t) \cdot e^{-lb \cdot t}$$

Instead of using this function directly, it is recommended that you use [apodize], page 23.

`w = gausswindow (t, lb)` [Function File]

Calculate window coefficients for gaussian windowing of a signal. The parameter *lb* is the line-broadening factor (in Hz) to apply to the signal having time points in *t*.

The default line-broadening factor is 0.3 Hz.

The equation applied to the free induction decay is as follows:

$$wfid(t) = fid(t) \cdot e^{-(lb \cdot t)^2}$$

Instead of using this function directly, it is recommended that you use [apodize], page 23.

`wfid = sinewindow (t, opts)` [Function File]

Calculate window coefficients for sinusoidal windowing of a signal.

The parameters of the window are specified as fields in the *opts* structure. The parameter *offset* is the relative offset of the sine bell. The parameter *ending* is the relative ending of the sine bell. Both *offset* and *ending* span ($[0, 1]$). The final parameter *exponent* specifies the order of the sinusoid.

The default parameter values are 0.5, 0 and 2, resulting in a squared cosine window.

The equation applied to the free induction decay is as follows:

$$wfid(t) = fid(t) \cdot \sin\left[\pi(offset) + \pi(ending - offset)\left(\frac{t}{t_{max}}\right)\right]$$

Instead of using this function directly, it is recommended that you use [apodize], page 23.

5.3 Zero-filling

MVAPACK defines one zero-filling operation as a doubling of the number of time-domain data points, two operations as a quadrupling, and so on. The zero-filling operation provides a means to increase digital resolution of the sampled data, but adds no further information to the data.

zfid = zerofill (*fid*, *parms*) [Function File]
zfid = zerofill (*fid*, *parms*, *k*) [Function File]
Appends zeros at the end of a free induction decay (FID) vector, matrix or cell array by doubling the total length k times. If k is not supplied, a default number of one zero fill is performed.

For one-dimensional data, k must be a scalar value. In the case of two-dimensional data, k may be a scalar or a vector.

5.4 Fourier transformation

Fourier transformation converts time-domain free induction decay data into frequency-domain spectral data, a required operation for almost all 1D NMR processing protocols.

s = nmrft (*fid*, *parms*) [Function File]
[*s*, *ppm*] = nmrft (*fid*, *parms*) [Function File]
[*s*, *ppm*, *hz*] = nmrft (*fid*, *parms*) [Function File]
... = nmrft (*fid*, *parms*, *doshift*) [Function File]
Performs Fourier transformation and shifting to produce an NMR spectrum.

In the one-dimensional case, data in *fid* may either be a column vector or a data matrix where each free induction decay is arranged as a row in the matrix.

In the two-dimensional case, data in *fid* may either be a data matrix where each direct-dimension slice is along the rows, or a cell array that contains multiple matrices, each having direct-dimension slices along its rows.

A parameter structure (or array) must be passed as a second argument. Optionally, a second output value may be produced which contains the chemical shift abscissa vector(s) associated with *s* (*ppm*. Also, a third output value may be produced which contains the centered abscissa vector in hertz units, without the carrier offset applied (*hz*).

A final optional argument, *doshift*, may be passed to enable or disable the default behavior of shifting the Fourier-transformed data by half the number of data points. The default behavior is to shift the data, and you should really never have a reason not to.

5.5 Phasing

Offsets in the signal phase relative to the receiver phase must be corrected through the process of phasing, either manually or automatically as discussed below.

5.5.1 Simple phasing

sp = phase (*s*, *parms*, *phc0*, *phc1*) [Function File]
Corrects the phase of a Fourier-transformed spectrum with a zero order correction *phc0* and a first order correction *phc1*. The first order correction is performed as a function of chemical shift offset from the spectral center frequency. The arguments *phc0* and *phc1* may be scalars or vectors, where the *i*-th elements of the vectors correct the *i*-th row of *s*.

sp = autophase (*s*, *parms*) [Function File]
[*sp*, *phc0*, *phc1*] = autophase (*s*, *parms*) [Function File]
sp = autophase (*s*, *parms*, *objective*) [Function File]
[*sp*, *phc0*, *phc1*] = autophase (*s*, *parms*, *objective*) [Function File]
Performs automatic phase correction of a one- or two-dimensional NMR spectrum or spectral dataset, using a simplex optimization algorithm:

> M. Siegel. 'The use of the modified simplex method for automatic phase correction in Fourier-transform Nuclear Magnetic Resonance spectroscopy'. Analytica Chimica Acta, 1981. 133(1981): 103-108.

For one-dimensional spectra, the algorithm uses an entropy minimization objective during optimization (See [simplex_entropy], page 26). For two-dimensional spectra, a whitening objective is used (See [simplex_whiten], page 27).

In the one-dimensional case, data in *s* may either be a column vector or a data matrix where each spectrum is arranged as a row in the matrix.

In the two-dimensional case, data in *s* may either be a data matrix where each direct-dimension slice is along the rows, or a cell array that contains multiple matrices, each having direct-dimension slices along its rows.

A parameter structure (or array) must be passed as a second argument. The phase correction values *phc0* and *phc1* may also be returned if desired.

5.5.2 Advanced phasing

obj = simplex_minimum (*s*, *phc*) [Function File]
Objective function for [autophase], page 26, that maximizes the lowest real spectral point.

obj = simplex_integral (*s*, *phc*) [Function File]
Objective function for [autophase], page 26, that maximizes the integrated area of the real component of the spectrum.

obj = simplex_entropy (*s*, *phc*) [Function File]
Objective function for [autophase], page 26, that minimizes the entropy of the first derivative of the real spectral component:

> L. Chen, Z. Weng, L. Y. Goh, M. Garland. 'An efficient algorithm for automatic phase correction of NMR spectra based on entropy minimization'. J. Magn. Res., 2002. 158(2002): 164-168.

obj = simplex_whiten (*s*, *phc*) [Function File]

Objective function for [autophase], page 26, that minimizes the number of 'colored pixels' of the real spectral component:

> G. Balacco, C. Cobas. 'Automatic phase correction of 2D NMR spectra by a whitening method'. Mag. Res. Chem., 2009. 47: 322-327.

5.6 Referencing

Referencing globally aligns the entire dataset based on an internal reference compound with a standardized chemical shift (e.g. DSS). In MVAPACK, referencing only changes the chemical shift axis values, not the data matrix values.

ppmadj = refadj (*ppm*, *oldcs*, *newcs*) [Function File]

Shifts the values in the chemical shift vector to move a reference position to zero chemical shift. This operates only on the abscissa vector, not on the data matrix.

If you need to reference individual spectra within a data matrix to a common chemical shift axis, use [coshift], page 29.

5.7 Regions of Interest

Selecting regions of interest (ROIs) in a dataset is a requisite for any number of functions in MVAPACK that take spectral regions as inputs. Regions of interest take the form of an $R \times 2$ matrix, where each of the R rows contains two values defining a start and an end for that region.

5.7.1 Generating regions

While regions of interest may be constructed manually, MVAPACK provides some automatic routines for generating them as well.

roi = roibin (*s*, *ab*, *parms*, *wmin*) [Function File]

Generate regions of interest (ROIs) from a spectrum or set of spectra in *s* with corresponding abscissa in *ab*. The minimum viable ROI width is specified in *wmin*.

This function generates ROIs by first using [binadapt], page 30, to bin the variables of a spectrum or a set of spectra, and converts the bins into ROIs.

roi = roipeak (*s*, *ab*, *parms*, *wmin*) [Function File]

Generate regions of interest (ROIs) from a spectrum or set of spectra in *s* with corresponding abscissa in *ab*. The minimum viable ROI width is specified in *wmin*.

This function generates ROIs by first using [peakpick], page 76, to pick the peaks of a spectrum or a set of spectra, creates ROIs of width *wmin* for each peak, and then joins all overlapped ROIs together until no two ROIs overlap.

Regions of interest may also be generated from bin parameters output from any of the MVAPACK binning routines:

roi = bin2roi (*abnew*, *widths*) [Function File]

Translate bin centers and widths in *abnew* and *widths*, respectively, into regions of interest in *roi*. The resulting regions may then be plotted using [roiplot], page 28.

5.7.2 Visualizing regions

Regions of interest may be overlaid onto data using the [roiplot], page 28, function.

`roiplot (X, ab, parms, roi)` [Function File]
: Builds a line plot of one-dimensional spectral data with overlaid regions of interest as rectangles, or builds a contour plot of two-dimensional spectral data with overlaid regions of interest as rectangles.

5.7.3 Removing regions

Regions of interest may be removed from the ROI matrix using the [rmroi], page 28, function.

`roirm = rmroi (roi, rmzones)` [Function File]
: Removes any regions of interest from *roi* that overlaps the regions of interest specified in *rmzones*.

5.7.4 Binning by regions

A manual binning method is available that utilizes regions of interest as a means of dividing data matrices (See [binmanual], page 30).

5.7.5 Vectorizing regions

Data in one-dimensional and two-dimensional spectral data structures may be vectorized into a data matrix using the [roi2data], page 28, command. Regions that have been vectorized using that function may be de-vectorized using the [data2roi], page 28, function.

`Xroi = roi2data (X, ab, parms, roi)` [Function File]
`[Xroi, abroi] = roi2data (X, ab, parms, roi)` [Function File]
: Builds a data matrix from paired one- or two-dimensional spectral data and regions of interest by concatenating the spectral data inside each region. For one-dimensional data, this is effectively the logical inverse of the [rmroi], page 28, function. For two-dimensional data, regions are vectorized prior to concatenation.

`X = data2roi (Xroi, ab, parms, roi)` [Function File]
: Reconstructs all possible spectral data contained within a data matrix that was generated by [roi2data], page 28.

5.8 Integration

Integral curves or integral values are useful for determining relative signal proportions from spectral data, and may be calculated and visualized using the following functions in MVAPACK:

`Imax = integrals (X, ab, roi)` [Function File]
`[I, Iab] = integrals (X, ab, roi)` [Function File]
: Calculates integrals of a spectral dataset over the specified regions of interest in *roi*. If a single output is requested (*Imax*), it will hold the final values of the integals. If two outputs are requested (*I*, *Iab*), they will hold the integration curves.

`integralsplot (x, ab, Ix, Iab)` [Function File]
: Overlays integral curves generated from [integrals], page 28, on a spectral line plot.

5.9 Alignment

In contrast to referencing, spectral alignment changes the values in a data matrix based on a circular shift operation of each row, without changing the chemical shift axis. Alignment may either be performed globally (See [coshift], page 29) or locally (See [icoshift], page 29), as detailed below.

`Xc = coshift (X)` [Function File]
`[Xc, lags] = coshift (X)` [Function File]

Performs whole-spectrum correlation-optimized shifting to align peaks in NMR spectra to a common target, the average of the dataset. The *Xc* output is the set of aligned spectra, and *X* is the input (real) data matrix to be shifted. Optionally, the number of points each spectrum was shifted may be returned in *lags*.

If individual peaks are misaligned within spectra such that this function cannot correct them, use [icoshift], page 29. However, this function will not produce warping artifacts.

`Xc = icoshift (X, ab)` [Function File]
`Xc = icoshift (X, ab, seg)` [Function File]
`[Xc, lags] = icoshift (X, ab)` [Function File]
`[Xc, lags] = icoshift (X, ab, seg)` [Function File]
`[Xc, lags] = icoshift (X, ab, seg, cofirst)` [Function File]

Performs interval correlation-optimized shifting to align peaks in NMR spectra to a common target, by default the average of the dataset.

The *Xc* output is the set of aligned spectra, and *X* is the input (real) data matrix to be shifted. The second argument *ab* is the abscissa of the spectral data.

The optional argument *seg* may be used to either define a number of segments to use (scalar) or define one or more segments based on abscissa values (matrix). Each manually defined segment should be a two-element row in *seg*. The final optional argument *cofirst* allows the user to perform an initial global alignment ([coshift], page 29) prior to segmented alignment. The default value of *cofirst* is `false`.

This code is based on the algorithm documented in:

> F. Savorani, G. Tomasi, S. B. Engelsen, 'icoshift: A versatile tool for the rapid alignment of 1D NMR spectra.', J. Magn. Res. 2010: 190-202.

5.10 Binning

Binning is a dimensionality reduction method that groups input variables into adjacent 'bins' and produces a smaller set of output variables by integrating the original variables in each bin to yield one output. Available methods of binning in MVAPACK are detailed below.

`xnew = binunif (X, ab, parms, w)` [Function File]
`[xnew, abnew] = binunif (X, ab, parms, w)` [Function File]
`[xnew, abnew, widths] = binunif (X, ab, parms, w)` [Function File]

`xnew = binunif (X, ab, parms, w)` [Function File]

Manually bin a one- or two-dimensional spectrum or spectral dataset in *X* based on uniform bin widths provided in *w*. The *abnew* and *widths* values are optionally returnable.

`xnew = binoptim (X, ab, w, slack)` [Function File]
`[xnew, abnew] = binoptim (X, ab, w, slack)` [Function File]
`[xnew, abnew, widths] = binoptim (X, ab, w, slack)` [Function File]
`[xnew, abnew, widths, indices] = binoptim (X, ab, w, slack)` [Function File]

Optimally bin a one-dimensional spectrum or spectral dataset in *X* such that final bins have a width no greater than *w*. The optionally returnable values in *abnew* correspond to the new bin centers in abscissa units. The final optional return value (*widths*) provides the widths of all the output bins.

This code is based on the description of the Optimized Bucketing Algorithm (OBA) presented in:

> Sousa et. al., 'Optimized Bucketing of NMR spectra: Three case studies', Chemometrics and Intelligent Lab Systems, 2013.

The *w* value is optional and has a default value of *0.025*. Also optional is the *slack* variable, which must range from *0* to *1* and has a default value of *0.45*.

`xnew = binadapt (X, ab, parms, w)` [Function File]
`[xnew, abnew] = binadapt (X, ab, parms, w)` [Function File]
`[xnew, abnew, widths] = binadapt (X, ab, parms, w)` [Function File]

Adaptively bin a one- or two-dimensional spectrum or spectral dataset in *X* such that final bins have a width no less than *w*. The optionally returnable values in *abnew* correspond to the new bin centers in abscissa units. The final optional return value (*widths*) provides the widths of all the output bins.

In the one-dimensional case, data in *X* may either be a column vector or a data matrix where each observation is arranged as a row in the matrix.

In the two-dimensional case, data in *X* may either be a data matrix where each direct-dimension slice is along the rows, or a cell array that contains multiple matrices, each having direct-dimension slices along its rows.

This code is based on the description of 'AI-binning' presented in:

> De Meyer et. al., 'NMR-Based Characterization of Metabolic Alterations in Hypertension Using an Adaptive, Intelligent Binning Algorithm', Analytical Chemistry, 2008.

`xnew = binmanual (X, ab, parms, roi)` [Function File]
`[xnew, abnew] = binmanual (X, ab, parms, roi)` [Function File]
`[xnew, abnew, widths] = binmanual (X, ab, parms, roi)` [Function File]
`xnew = binmanual (X, ab, parms, centers, widths)` [Function File]

Manually bin a one-dimensional spectrum or spectral dataset in *X* based on regions of interest provided in *roi*, or centers and widths provided in *centers* and *widths*. If regions of interest are used to bin, the *abnew* and *widths* values are optionally returnable.

6 Data pre-treatment

Pre-treatment encompasses all operations that are needed to take the processed ('sanitized') instrumental data to a form that is suitable for digestion by multivariate analysis algorithms like PCA, PLS or OPLS. This chapter documents all available pre-treatment functions in MVAPACK.

6.1 Removing data

Outlying or invalid observations due to failures in sample preparation may be removed in MVAPACK. It is *highly recommended* that observations are removed *in MVAPACK* using [rmobs], page 31, and **not** by deleting input data and rows from any class label files. The former method is easier to document and trace back, whereas missing data can lead to confusion later. The recommended way to remove observations is presented in Section 3.3.1 [Removal of undesirable observations], page 14.

Variables that are known to cause issues during model building (*e.g.* baseline, reference, solvent, etc.) may also be removed using [rmvar], page 31. Alternatively, low-variation bin variables may be 'automatically' removed using [rmnoise], page 31.

Functions for removing observations and variables from MVAPACK data are described in the following subsections.

6.1.1 Removing observations

`Xrm = rmobs (X, idx)` [Function File]
Removes an observation (*idx* scalar) or observations (*idx* vector) from the dataset *X*.

6.1.2 Removing variables

`idx = findnearest (x, a)` [Function File]
Finds the nearest index in a vector *x* to a value *a*. This is similar to the `find` command, but returns the nearest match if no entries are exact.

`[Xrm, abrm] = rmvar (X, ab, idx)` [Function File]
Removes a variable (*idx* scalar) or variables (*idx* vector) from the dataset *X* and its corresponding abscissa, *ab*. Use [findnearest], page 31, to locate the indices to supply to *idx*.

`[Xrm, abrm] = rmnoise (X, ab, idx)` [Function File]
`[Xrm, abrm] = rmnoise (X, ab, idx, nstd)` [Function File]
`[Xrm, abrm, idxrm] = rmnoise (X, ab, idx)` [Function File]
`[Xrm, abrm, idxrm] = rmnoise (X, ab, idx, nstd)` [Function File]
Uses a relative standard deviation to distinguish between signal and noise in a binned NMR spectral dataset. Bins identified as noise will be removed in the output data matrix *Xrm* and abscissa *abrm*. The *idx* variable is used to defined a noise region.

An optional argument *nstd* may be supplied (default: 2) to set how many standard deviations from the mean noise value the threshold will be.

IMPORTANT: This routine is *not* designed to be used with full-resolution data matrices, *especially* when binning is a subsequent step. Such actions are almost surely guaranteed to yield strange results.

6.2 Normalization

Normalization operations are applied to the *rows* of a data matrix in order to bring observations into the same domain. Normalization is commonly applied to correct for variations in total sample concentration, spectrometer sensitivity, *etc.*

NOTE: All normalization routines below expect real (not complex) data, with the exception of [pscorr], page 33.

6.2.1 CS normalization

`Xn = csnorm (X)` [Function File]
`[Xn, s] = csnorm (X)` [Function File]

Normalize the observations of a data matrix to constant integral. The calculated normalization factors may be optionally returned in *s*.

6.2.2 PQ normalization

`Xn = pqnorm (X)` [Function File]
`[Xn, s] = pqnorm (X)` [Function File]

Normalize the observations of a data matrix using the Probabilistic Quotient Normalization method as described in:

> F. Dieterle et. al. 'Probabilistic Quotient Normalization as Robust Method to Account for Dilution of Complex Biological Mixtures. Application in 1H NMR Metabonomics.' Analytical Chemistry, 2006.

6.2.3 HM normalization

`Xn = histmatch (X)` [Function File]
`[Xn, s] = histmatch (X)` [Function File]

Normalize the observations of a data matrix using the Histogram Matching method described in:

> R. J. O. Torgrip et. al. 'A note on normalization of biofluid 1H-NMR data'. Metabolomics, 2008. 4(2008):114-121.

6.2.4 SNV normalization

`Xn = snv (X)` [Function File]
`[Xn, mu] = snv (X)` [Function File]
`[Xn, mu, s] = snv (X)` [Function File]

Normalize the observations of a data matrix using standard normal variate normalization (SNV). Row centering is also performed in the process. The variables used to center and scale X may optionally be returned.

6.2.5 MSC normalization

`Xn = mscorr (X)` [Function File]
`Xn = mscorr (X, r)` [Function File]

Use multiplicative scatter correction to normalize spectra into better alignment with each other, row-wise at least. If the optional r reference observation is not provided, the mean of the observations in X will be used.

6.2.6 PSC normalization

`Xn = pscorr (X)` [Function File]
`[Xn, b] = pscorr (X)` [Function File]
`Xn = pscorr (X, r)` [Function File]
`[Xn, b] = pscorr (X, r)` [Function File]

Use simultaneous phase-scatter correction to normalize spectra into better alignment with each other, row-wise at least. If the optional r reference observation is not provided, the mean of the observations in X will be used. Normalization factors may be requested through the second optional return value b. More information here:

> B. Worley, R. Powers, 'Simultaneous Phase and Scatter Correction in NMR Datasets', Chemometr. Intell. Lab. Syst., 2013, submitted.

6.2.7 ROI normalization

`Xn = roinorm (X, ab, roi)` [Function File]
`[Xn, s] = roinorm (X, ab, roi)` [Function File]

Normalize the observations of a data matrix to such that the integral of a spectral region specified by *roi* is one. The calculated normalization factors may be optionally returned in s. The values specified in *roi* must correspond to those in the abscissa ab.

6.3 Scaling

Scaling operations are applied to the *columns* of a data matrix in order to bring variables into the same domain.

Without scaling, algorithms such as PCA and PLS examine the *covariance* eigenstructure of X and Y. It is possible – in fact, common – for large variables to dominate the covariance eigenstructure, effectively causing PCA and PLS to dismiss smaller variables. This unequal weighting of variables is corrected through scaling. The most common scaling method, unit variance scaling, results in PCA and PLS examining the *correlation* eigenstructure of their input matrices where all variables have equal weight.

Scaling methods other than UV have different purposes and effective weightings. A more thorough discussion of scaling is discussed here:

> R. A. van den Berg et. al., 'Centering, scaling, and transformations: improving the biological information content of metabolomics data', BMC Genomics, 2006(7): 142.

In the mathematical descriptions of the scaling methods below, the scaled matrix $\tilde{X}$ is the output of the scaling function $f(X)$, and i and k are row and column indices of the data matrices, respectively:

$$\tilde{X} = [\tilde{x}_{ik}] = f(X)$$

Furthermore, $\bar{x}_k$ represents the sample mean of the k-th column of X, defined as:

$$\bar{x}_k = \frac{1}{N} \sum_{i=1}^{N} x_{ik}$$

Finally, s_k represents the sample standard deviation of the k-th column of X, defined as:

$$s_k = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_{ik} - \bar{x}_k)^2}$$

6.3.1 No scaling

Performing no scaling prior to multivariate analysis results in fitting based on covariance eigenstructure, not correlation eigenstructure. In English, large variations will be weighted much more strongly than smaller variations in the fitted models.

`Xc = snone (X)` [Function File]
`[Xc, mu] = snone (X)` [Function File]
`[Xc, mu, s] = snone (X)` [Function File]
`... = snone (X, w)` [Function File]

Performs only mean-centering, but no scaling. An optional weighting vector w may be passed during the scaling. The variables used to center and scale X may be optionally returned.

The resulting scaled elements of $\tilde{X}$ are calculated as follows:

$$\tilde{x}_{ik} = \frac{x_{ik} - \bar{x}_k}{w_k}$$

6.3.2 Unit variance scaling

Performing unit variance (UV) scaling prior to multivariate analysis results in fitting based on the correlation eigenstructure of the data, meaning that every variable is equally weighted in the fitted models.

`Xs = suv (X)` [Function File]
`[Xs, mu] = suv (X)` [Function File]
`[Xs, mu, s] = suv (X)` [Function File]
`... = suv (X, w)` [Function File]

Scale the variables of a data matrix to unit sample variance by dividing by their sample standard deviation. Centering is also performed in the process. An optional weighting vector w may be passed during the scaling. The variables used to center and scale X may optionally be returned.

The resulting scaled elements of $\tilde{X}$ are calculated as follows:

$$\tilde{x}_{ik} = \frac{x_{ik} - \bar{x}_k}{w_k s_k}$$

6.3.3 Pareto scaling

Pareto scaling is a middle-ground between applying no scaling and applying UV scaling. Large variations are still weighted more strongly than smaller variations, but they will be less likely to dominate the resulting models as would occur in unscaled data.

`Xs = spareto (X)` [Function File]
`[Xs, mu] = spareto (X)` [Function File]
`[Xs, mu, s] = spareto (X)` [Function File]
`... = spareto (X, w)` [Function File]

Scale the variables of a data matrix by the square root of their sample standard deviation. Centering is also performed in the process. An optional weighting vector w may be passed during the scaling. The variables used to center and scale X may be optionally returned.

The resulting scaled elements of $\tilde{X}$ are calculated as follows:

$$\tilde{x}_{ik} = \frac{x_{ik} - \bar{x}_k}{w_k \sqrt{s_k}}$$

6.3.4 Range scaling

Range scaling scales all variables to a unit range, effectively comparing signals relative to an underlying range of responses.

`Xs = srange (X)` [Function File]
`[Xs, mu] = srange (X)` [Function File]
`[Xs, mu, s] = srange (X)` [Function File]
`... = srange (X, w)` [Function File]

Scale the variables of a data matrix to unit range. Centering is also performed in the process. An optional weighting vector w may be passed during the scaling.

The resulting scaled elements of $\tilde{X}$ are calculated as follows:

$$\tilde{x}_{ik} = \frac{x_{ik} - \bar{x}_k}{w_k (x_{k,\max} - x_{k,\min})}$$

6.3.5 Level scaling

Level scaling focuses multivariate analysis on fitting the relative responses of all signals. Signals with larger relative responses will be weighted more heavily than those with smaller relative responses.

`Xs = slevel (X)` [Function File]
`[Xs, mu] = slevel (X)` [Function File]
`[Xs, mu, s] = slevel (X)` [Function File]
`... = slevel (X, w)` [Function File]

Scale the variables of a data matrix by their mean value, the level scaling method. Centering is also performed in the process. An optional weighting vector w may be passed during the scaling. The values used to center and scale X may optionally be returned.

The resulting scaled elements of $\tilde{X}$ are calculated as follows:

$$\tilde{x}_{ik} = \frac{x_{ik} - \bar{x}_k}{w_k \bar{x}_k}$$

6.3.6 VAST scaling

VAriable STability (VAST) scaling focuses multivariate analysis on fitting small variations in a data matrix.

`Xs = svast (X)` [Function File]
`[Xs, mu] = svast (X)` [Function File]
`[Xs, mu, s] = svast (X)` [Function File]
`... = svast (X, w)` [Function File]

Scale the variables of a data matrix using VAST scaling. Centering is also performed in the process. An optional weighting vector w may be passed during the scaling. The variables used to center and scale X may be optionally returned.

The resulting scaled elements of $\tilde{X}$ are calculated as follows:

$$\tilde{x}_{ik} = \frac{x_{ik} - \bar{x}_k}{w_k s_k} \cdot \frac{\bar{x}_k}{s_k}$$

6.4 Denoising

Methods of denoising are intended to remove variation in a data matrix that does not contribute to class distinction. This removal results in more parsimonious supervised models that exhibit lower lack-of-fit sum-of-squares.

6.4.1 Direct orthogonal signal correction

`[Z, W, P, T] = dosc (X, Y, A)` [Function File]
`[Z, W, P, T] = dosc (X, Y, A, tol)` [Function File]

Performs Direct Orthogonal Signal Correction as described in:

> Westerhuis J. A., de Jong S., Smilde A. K., 'Direct orthogonal signal correction', Chemometrics and Intelligent Laboratory Systems, 56 (2001): 13-25.

The function requires a data matrix X, a response matrix Y, and a number of OSC components to calculate A. If a fourth argument *tol* is provided, it will be used as the tolerance for calculating a pseudoinverse of the data matrix. Otherwise, a default value of 1e-3 will be used.

The function returns a corrected data matrix Z, OSC weights W, OSC loadings P and OSC score components T.

6.4.2 Per-class treatment of data

One method of "denoising" is to apply data pre-treatment algorithms to a data matrix on a per-class basis, instead of treating the data as a whole. The very act of applying the same action to each class individually can increase the between-class variation in a dataset and enhance class separations.

`Xnew = perclass (fn, X, Y)` [Function File]

Performs the operation defined in the function handle *fn* on X in a class-dependent manner. In other words, instead of treating the entire dataset as one as would occur when applying *fn* normally, *fn* is applied to each class *individually* and the results are reassembled into the matrix *Xnew*.

It is important to note that the function *fn* must not modify either the number of rows or columns of the data matrix X. Finally, the function handle *fn* must be of the following form:

```
function Xnew = fn (X), ... end
```

7 Multivariate modeling

Multivariate modeling in MVAPACK is intended to be used primarily for chemical fingerprinting applications where global chemical trends in spectral data are sought. For all algorithms discussed below, scores and loadings will be produced as low-dimensional representations of high-dimensional observations and variables, respectively.

7.1 Model training

The process of generating a model from an initial set of data is referred to as 'training' the model, where the initial dataset is called the 'training set'.

The algorithms documented below are used to form unsupervised (PCA) or supervised (PLS, OPLS, LDA) multivariate models from input data matrices X and response matrices Y. These algorithms search for variations in the high-dimensional input data that either contribute most to the total variation (PCA) or contribute most to class distinction (PLS, OPLS, LDA).

In the mathematical descriptions of the model fitting methods below, X is an $N \times K$ data matrix and Y is an $N \times M$ response matrix.

7.1.1 PCA

Principal Component Analysis (PCA) generates a linear model of an input data matrix X such that the model components are orthogonal and capture a maximal amount of variation present in X. The PCA model is structured as follows:

$$X = TP' + E = \sum_{a=1}^{A} t_a p_a' + E$$

Where T is an $N \times A$ matrix of scores, P is a $K \times A$ matrix of loadings, E is an $N \times K$ matrix of residuals. The number of components A is chosen by cross-validation such that the model's cumulative R^2 metric is greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing.

PCA model scores in T are low-rank approximations of observations in X, and model loadings in P are approximations of variables in X.

`mdl = pca (X)` [Function File]
`mdl = pca (X, scalefn)` [Function File]
`mdl = pca (X, scalefn, ncv)` [Function File]
`mdl = pca (X, scalefn, ncv, aout)` [Function File]
`mdl = pca (X, scalefn, ncv, aout, w)` [Function File]

Performs Principal Component Analysis (PCA) on the input data matrix X using the iterative NIPALS method. More information may be found here:

> P. Geladi, B. R. Kowalski. 'Partial Least Squares Regression: A Tutorial', Analytica Chimica Acta, 1986(185): 1-17.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [suv], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV

groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The optional argument *aout* may be used if a specific number of model components is desired. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

The last optional argument *w* may be used to manually weight the variables during the PCA analysis. The weights will be effectively multiplied into the scale values obtained by *scalefn*.

7.1.2 PLS

Partial Least Squares Projection to Latent Structures (PLS) generates a linear model of an input data matrix X and response matrix Y such that the model components are orthogonal and capture a maximal amount of correlation between X and Y. The PLS model is structured as follows:

$$X = TP' + E = \sum_{a=1}^{A} t_a p_a' + E$$

$$Y = UC' + F = \sum_{a=1}^{A} u_a c_a' + F$$

Where T is an $N \times A$ matrix of scores, P is a $K \times A$ matrix of X-loadings, E is an $N \times K$ matrix of X residuals, U is an $N \times A$ matrix of Y-scores, C is an $M \times A$ matrix of Y-weights, and F is an $N \times M$ matrix of Y residuals.

The number of components A is chosen by cross-validation such that the model's cumulative R_X^2 and R_Y^2 metrics are greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing.

PLS model scores in T are low-rank approximations of observations in X and are good predictors of U. Model X-loadings in P are approximations of variables in X and Y-weights in C are approximations of columns of Y.

mdl = pls (*X*, *Y*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*, *ncv*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*, *ncv*, *aout*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*, *ncv*, *aout*, *w*) [Function File]

Performs Projection to Latent Structures (PLS) on the input data matrix X and response matrix Y using the iterative NIPALS method. More information may be found here:

> S. Wold, M. Sjostrom, L. Eriksson, 'PLS-regression: A basic tool in chemometrics', Chemometr. Intell. Lab. Syst., 2001(58): 109-130.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [suv], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used

to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The final optional argument *aout* may be used if a specific number of model components is desired. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

7.1.3 OPLS

Orthogonal Projections to Latent Structures (OPLS) generates a linear model of an input data matrix X and response matrix Y such that the model components are orthogonal and capture both a maximal amount of correlation and anti-correlation between X and Y. The OPLS model is structured as follows:

$$X = T_p P_p' + T_o P_o' + E = \sum_{a=1}^{A_p} t_{p,a} p_{p,a}' + \sum_{b=1}^{A_o} t_{o,b} p_{o,b}' + E$$

$$Y = UC' + F = \sum_{a=1}^{A_p} u_a c_a' + F$$

Where T_p is an $N \times A_p$ matrix of predictive scores, T_o is an $N \times A_o$ matrix of orthogonal scores, P_p is a $K \times A_p$ matrix of predictive X-loadings, P_o is a $K \times A_o$ matrix of orthogonal X-loadings, E is an $N \times K$ matrix of X residuals, U is an $N \times A_p$ matrix of Y-scores, C is an $M \times A_p$ matrix of Y-weights, and F is an $N \times M$ matrix of Y residuals.

The number of predictive components A_p is chosen by cross-validation such that the model's cumulative $R^2_{X_p}$ and R^2_Y metrics are greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing. At least one orthogonal component is simultaneously extracted with each predictive component during fitting.

OPLS scores in T_p and T_o are low-rank approxiations of predictive and orthogonal variation in observations in X, respectively. The OPLS loadings in P_p and P_o fill similar roles for the variables in X.

`mdl = opls (X, Y)` [Function File]
`mdl = opls (X, Y, scalefn)` [Function File]
`mdl = opls (X, Y, scalefn, ncv)` [Function File]
`mdl = opls (X, Y, scalefn, ncv, aout)` [Function File]
`mdl = opls (X, Y, scalefn, ncv, aout, w)` [Function File]

Performs Orthogonal Projection to Latent Structures (OPLS) on the input data matrix X and response matrix Y using the iterative NIPALS method, described here:

> J. Trygg, S. Wold, 'Orthogonal Projections to Latent Structures (O-PLS)', J. Chemometrics, 2002(16): 119-128.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [spareto], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The final optional argument *aout* may be used if a specific number of model components is desired. If *aout* is specified as a scalar, then the model will be built with that number of predictive components. If *aout* is a vector, the first element should hold the desired number of predictive components. The remaining elements in *aout* should hold the desired number of orthogonal components for each predictive component. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

7.1.4 LDA

Linear Discriminant Analysis (LDA) generates a linear model of an input data matrix X and response matrix Y such that the model components capture a maximal amount of between-class variation. The model projects the (non-singular) input data into a discrimination space as follows:

$$T = XP$$

In other words, the model is structured as follows:

$$X = TP' + E$$

Where P is a matrix of the first A significant eigenvectors of the Fisher LDA matrix S:

$$PDP^{-1} = S = S_W^{-1} S_B$$

And S_W is the within-class covariance matrix and S_B is the between-class covariance matrix. New observations are classified based on which class mean they fall closest to after projection by P.

The number of components A is chosen by cross-validation such that the model's cumulative R_X^2 and R_Y^2 metrics are greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing.

`mdl = lda (X, Y)` [Function File]
`mdl = lda (X, Y, scalefn)` [Function File]
`mdl = lda (X, Y, scalefn, ncv)` [Function File]
`mdl = lda (X, Y, scalefn, ncv, aout)` [Function File]

Performs Linear Discriminant Analysis (LDA) on the input data matrix X and the input response matrix Y using an eigendecomposition of the within-class and between-class covariance matrix ratio. More information may be found here:

> W. Hardle, L. Simar. 'Applied Multivariate Statistical Analysis, 2nd ed.', Springer-Verlag, Berlin Heidelberg, 2007.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [suv], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The final optional argument *aout* may be used if a specific number of model components is desired. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

7.1.5 MB-PCA

Multiblock Principal Component Analysis (MBPCA) generates a linear model of a set of B input data matrices $X = [X_1 X_2 \cdots X_B]$ such that the overall model components are orthogonal and capture a maximal amount of variation present in X. The MBPCA model is structured as follows:

$$X = [X_1 \;\; X_2 \;\; \cdots \;\; X_B], \forall b \in [1, B]$$

Where each block (X_b) is itself a bilinear model:

$$X_b = T_b P_b' + E_b = \sum_{a=1}^{A} t_{b,a} p_{b,a}' + E_b$$

Where T_b is an $N \times A$ matrix of block scores, P_b is a $K_b \times A$ matrix of block loadings, E_b is an $N \times K_b$ matrix of residuals. The number of components A is chosen by cross-validation such that the model's cumulative R^2 metric is greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing.

PCA model scores in T (and T_b) are low-rank approximations of observations in X (and X_b), and model loadings in P (and P_b) are approximations of variables in X (and X_b).

`mdl = mbpca (X)` [Function File]
`mdl = mbpca (X, scalefn)` [Function File]
`mdl = mbpca (X, scalefn, ncv)` [Function File]
`mdl = mbpca (X, scalefn, ncv, aout)` [Function File]

Performs Multiblock Principal Component Analysis (MBPCA) on the set of input data matrices $X = \{X1 \ldots XB\}$ using a Consensus PCA (CPCA) method with modifications from Westerhuis. More information may be found here:

> J. Westerhuis, T. Kourti, J. F. MacGregor. 'Analysis of Multiblock and Hierarchical PCA and PLS models', Journal of Chemometrics, 1998(21): 301-321.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [suv], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The final optional argument *aout* may be used if a specific number of model components is desired. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

7.1.6 MB-PLS

Multiblock Partial Least Squares Projection to Latent Structures (MBPLS) generates a linear model of a set of B input data matrices $X = [X_1 X_2 \cdots X_B]$ and a response matrix Y such that the overall model components are orthogonal and capture a maximal amount of correlation between X and Y. The MBPLS model is structured as follows:

$$X = [X_1 \;\; X_2 \;\; \cdots \;\; X_B], \forall b \in [1, B]$$

Where each block (X_b) is itself a bilinear model:

$$X_b = T_b P_b' + E_b = \sum_{a=1}^{A} t_{b,a} p_{b,a}' + E_b$$

$$Y = UC' + F = \sum_{a=1}^{A} u_a c_a' + F$$

Where T_b is an $N \times A$ matrix of block scores, P_b is a $K_b \times A$ matrix of X_b-loadings, E_b is an $N \times K_b$ matrix of X_b residuals, U is an $N \times A$ matrix of Y-scores, C is an $M \times A$ matrix of Y-weights, and F is an $N \times M$ matrix of Y residuals.

The number of components A is chosen by cross-validation such that the model's cumulative R_X^2 and R_Y^2 metrics are greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing.

PLS model scores in T (and T_b) are low-rank approximations of observations in X (and X_b) and are good predictors of U. Model X_b-loadings in P_b are approximations of variables in X_b and Y-weights in C are approximations of columns of Y.

`mdl = mbpls (X, Y)` [Function File]
`mdl = mbpls (X, Y, scalefn)` [Function File]
`mdl = mbpls (X, Y, scalefn, ncv)` [Function File]
`mdl = mbpls (X, Y, scalefn, ncv, aout)` [Function File]

Performs Multiblock Projection to Latent Structures (MBPLS) on the set of input data matrices $X = \{X1 \ldots XB\}$ and response matrix Y using a multiblock PLS method with modifications from Westerhuis. More information may be found here:

> J. Westerhuis, T. Kourti, J. F. MacGregor. 'Analysis of Multiblock and Hierarchical PCA and PLS models', Journal of Chemometrics, 1998(21): 301-321.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [suv], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The final optional argument *aout* may be used if a specific number of model components is desired. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

7.1.7 MB-OPLS

Multiblock Orthogonal Projections to Latent Structures (MBOPLS) generates a linear model of a set of B input data matrices $X = [X_1 X_2 \cdots X_B]$ and a response matrix Y such that the overall model components are orthogonal and capture a maximal amount of correlation between X and Y. The MBOPLS model is structured as follows:

$$X = [X_1 \quad X_2 \quad \cdots \quad X_B], \forall b \in [1, B]$$

Where each block (X_b) is itself a bilinear model:

$$X_b = T_b P_b' + T_{o,b} P_{o,b}' + E_b = \sum_{a=1}^{A} t_{b,a} p_{b,a}' + E_b$$

$$Y = UC' + F = \sum_{a=1}^{A} u_a c_a' + F$$

Where T_b is an $N \times A$ matrix of block scores, P_b is a $K_b \times A$ matrix of X_b-loadings, E_b is an $N \times K_b$ matrix of X_b residuals, U is an $N \times A$ matrix of Y-scores, C is an $M \times A$ matrix of Y-weights, and F is an $N \times M$ matrix of Y residuals.

The number of components A is chosen by cross-validation such that the model's cumulative R_X^2 and R_Y^2 metrics are greater than 0.01 and the model components' cumulative Q^2 metrics are strictly increasing.

PLS model scores in T (and T_b) are low-rank approximations of observations in X (and X_b) and are good predictors of U. Model X_b-loadings in P_b are approximations of variables in X_b and Y-weights in C are approximations of columns of Y. Orthogonal scores and loadings have been identified in $T_{o,b}$ and $P_{o,b}$ and are essentially low-rank stores for variation that interferes with PLS predictive components.

`mdl = mbopls (X, Y)` [Function File]
`mdl = mbopls (X, Y, scalefn)` [Function File]
`mdl = mbopls (X, Y, scalefn, ncv)` [Function File]
`mdl = mbopls (X, Y, scalefn, ncv, aout)` [Function File]

Performs Multiblock Orthogonal Projections to Latent Structures (MBOPLS) on the set of input data matrices $X = \{X1 \ldots XB\}$ nd response matrix Y using a multiblock OPLS method. More information may be found here:

> B. Worley, R. Powers. 'A Sequential Algorithm for Multiblock Orthogonal Projections to Latent Structures', Journal of Chemometrics, In prep.

The optional argument *scalefn* may be set to the function handle of an appropriate scaling routine. The default scaling function is [suv], page 34.

The optional argument *ncv* may be used to specify the number of cross-validation iterations and/or groups. If *ncv* is a scalar, it will be used to set the number of CV groups, with 10 CV iterations. If *ncv* is a 2-element vector, its elements will be used to set the number of CV groups and iterations, respectively. The default is to use 7 groups and 10 iterations.

The final optional argument *aout* may be used if a specific number of model components is desired. If *aout* is specified as a scalar, then the model will be built with that number of predictive components. If *aout* is a vector, the first element should hold the desired number of predictive components. The remaining elements in *aout* should hold the desired number of orthogonal components for each predictive component. **CAUTION:** using *aout* will not guarantee that the resultant model components are statistically significant!

7.1.8 SVM

Support vector machines (SVM) find an optimal separating hyperplane that has a maximum margin/distance to the observations by minimizing the following loss function:

$$min(\frac{1}{2}||\omega||^2 + C\sum_{i=1}^{N}\xi_i) \equiv min(\frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N}y_i y_j (x_i y_j)\alpha_i \alpha_j - \sum_{i=1}^{N}\alpha_i + C\sum_{i=1}^{N}\xi_i)$$

where y_i is the class label of the i^{th} observation either 1 or -1, N is number observations, x_i is the vector of features for the i^{th} observation, α's are Lagrange multipliers, w is the normal vector to the hyperplane, C is the cost parameter for any observation falling into the wrong side of the hyperplane, ξ_i are slack variables denoting distance of the i^{th} observation on the wrong side of the margin. In order to minimize the loss function to solve for estimates of parameters, sequential minimal optimization was paplied. More information are available here:

> Platt, J. Sequential minimal optimization: A fast algorithm for training support vector machines; Microsoft 1998.

Kernels can be applied to project original input feature space into higher dimensional space in which now the separating boundary is linear, using different basis functions.

Linear kernel calculates the dot product of any two input matrices using this equation:

$$K(x, x') =< x, x' >$$

where $< . >$ denotes the dot product.

`[svm_mdl] = svm(X, y, kernel, gamma, d, C)` [Function File]

Function that builds a support vector machine (SVM) model. Takes the following as input:

- X: n x p data matrix

- y: n x 1 vector of labels, -1 or 1

- kernel: kernel option to transform data, the default is linear kernel, i.e. no transformation

- gamma: reflect linearity degree of the classifying hyperplane. small and large gammas correspond to straight line curvy planes, respectively

- d: degrees of the polynomial kernel

- C: best cost value obtained from *tunesvm()* function

Function returns a structure with the svm model.

7.1.9 Random Forests

Create and combine classification random-forest trees which are essentially finding the optimal partitioning of the data by asking consectuve questions at each splitting node until all samples are well classified in the leaf nodes. To measure the quality of a split, information gain is calculated as following:

$$infoGain = -\sum p(y = c) log(p(y = c)) + \sum p(y = c|X_j < t) log(p(c|X_j < t))$$

With

$$p(y = c) = \frac{1}{|D|} \sum_{i \epsilon D} I(y_i = c)$$

$$p(y = c|X_j < t) = \frac{1}{|D'|} \sum_{i \epsilon D'} I(y_i = c|X_j < t)$$

where c is each class label, D is number of samples in that c^{th} class before splitting. D' is number of samples in the c^{th} class after the splitting which is done on X_j feature at value t. Information gain is computed for every feature and every possible splitting value. The combination of splitting feature and splitting value that results in maximum information gain is chosen to split the entire data set in the head node. This process is repeated at every node down in a tree until all samples are perfectly classified. More information can be found here:

Murphy, K., Machine learning: a probablistic perspective Massachussets Institute of Technology Press: 2012.

`rf = RandomForest(datasmatrix, labelsmatrix, numbertreescreated, numbertreessaved)` [Function File]

This serves as the main function for processing the data used to create a structure of RandomForest trees. The bulk of tree creation/computation is done from the RandomForest executable file (see RandomForest.cpp for souce code). Creates necesary intermediate files for the RandomForest executable as well as reads in the output of the executable and saves it to an octave structure.

`[yhat, prob]= rfclassify(mdl, Z, pseudo_label, numbertreessaved)` [Function File]

RandomForest classificationf function where mdl is any trained random forest model, Z is the new data matrix; *psuedo_label* is a vector of pseudo labels for the new observations.

7.2 Model prediction

Once a model has been trained and suitably validated, it may be used to model new observations (a 'prediction set') in a step unsurprisingly referred to as 'prediction'.

Given a model trained on an $N \times K$ matrix of observations and an $N \times M$ matrix of classes, a new set of n K-dimensional observations may be classified by the model, resulting in an $n \times M$ matrix of predicted class memberships for the new observations.

`Y = classify (mdl, X)` [Function File]
`[Y, T] = classify (mdl, X)` [Function File]

Predicts responses Y from one or more observations X based on the discriminant analysis model *mdl*. Any model that contains a class matrix of the format generated by [classes], page 52, should support classification of new observations.

An optional second return value (T) may be requested that holds transformed scores for the new observations.

7.3 Model visualization

Relationships of observations within a dataset may be visualized by plotting scores with [scoresplot], page 46. Loadings analogously represent variables and may be drawn with [loadingsplot], page 46. All forms of plotting, coloring, *etc* are documented in the subsections below.

7.3.1 Plotting

`stackplot (X)` [Function File]
`stackplot (X, ab)` [Function File]

Builds a stacked line plot of time-domain FID data or frequency- domain spectral data. If time-domain data is plotted, *ab* must hold the time values for the abcissa. If frequency-domain values are to be plotted, *ab* must have chemical shifts. The *ab* vector is optional.

`scoresplot (mdl)` [Function File]
`scoresplot (mdl, d)` [Function File]
`scoresplot (mdl, d, coloring)` [Function File]
`scoresplot (mdl, d, coloring, numbers)` [Function File]

Builds a scores plot of PCA, PLS or OPLS modeled data. The number of components to plot may be supplied as the optional second argument *d*. An optional matrix *coloring* may be passed to define the color scheme of the plot. An optional third argument (*numbers*) may be set to true (default is false) to change the plotted points into observation numbers.

`loadingsplot (mdl)` [Function File]
`loadingsplot (mdl, coloring)` [Function File]
`loadingsplot (mdl, coloring, numbers)` [Function File]
`h = loadingsplot (mdl)` [Function File]
`h = loadingsplot (mdl, coloring)` [Function File]
`h = loadingsplot (mdl, coloring, numbers)` [Function File]

Builds a loadings plot from PCA, PLS or OPLS modeled data. An optional second argument *coloring* may be set to an appropriate function (See [nocolors], page 48, [varcolors], page 48). An optional third argument (*numbers*) may be set to true (default is false) to change the plotted points into variable numbers.

`backscaleplot (ab, mdl)` [Function File]
`backscaleplot (ab, mdl, coloring)` [Function File]
`pdata = backscaleplot (ab, mdl, coloring)` [Function File]

Builds a backscaled loadings plot from PCA, PLS or OPLS modeled data. An optional second argument *coloring* may be set to enable or disable coloring of the backscaled loadings values. The default behavior is to generate uncolored plots, because plot coloring can take a while.

Because external plotting programs like `gnuplot` are considerably quicker at plotting multicolored lines, colored backscaled data ready for plotting may be returned into *pdata*.

More information on the technique of backscaling model loadings can be found in the following two references:

O. Cloarec et. al., 'Evaluation of the Orthogonal Projection on Latent Structure Model Limitations Caused by Chemical Shift Variability and Improved Visualization of Biomarker Changes in 1H NMR Spectroscopic Metabonomic Studies', Anal. Chem., 2005(77): 517–526.

O. Cloarec et. al., 'Statistical Total Correlation Spectroscopy: An Exploratory Approach for Latent Biomarker Identification from Metabolic 1H NMR Data Sets', Anal. Chem., 2005(77): 1282–1289.

`weightsplot (`*`mdl`*`)` [Function File]
`weightsplot (`*`mdl`*`, `*`coloring`*`)` [Function File]
`h` `= weightsplot (`*`mdl`*`)` [Function File]
`h` `= weightsplot (`*`mdl`*`, `*`coloring`*`)` [Function File]

Builds a loadings plot from PCA, PLS or OPLS modeled data. An optional second argument *coloring* may be specified to color the points. See [nocolors], page 48, [varcolors], page 48.

`cvscoresplot (`*`mdl`*`)` [Function File]
`cvscoresplot (`*`mdl`*`, d)` [Function File]
`cvscoresplot (`*`mdl`*`, d, `*`coloring`*`)` [Function File]
`cvscoresplot (`*`mdl`*`, d, `*`coloring`*`, `*`numbers`*`)` [Function File]

Builds a cross-validated scores plot of PLS or OPLS modeled data. The number of components to plot may be supplied as the optional second argument *d*. An optional matrix *coloring* may be passed to define the color scheme of the plot. An optional third argument (*numbers*) may be set to true (default is false) to change the plotted points into observation numbers.

`splot (`*`mdl`*`)` [Function File]
`splot (`*`mdl`*`, a)` [Function File]
`splot (`*`mdl`*`, a, `*`numbers`*`)` [Function File]
`pdata` `= splot (...)` [Function File]

Builds an S-plot from (O)PLS modeled data to facilitate identification of variables that contribute strongly to class distinction. An optional second argument *a* may be passed to select which predictive component is to be analyzed. The default behavior is to plot the first predictive component. An optional third argument (*numbers*) may be set to true (default is false) to change the plotted points into variable numbers.

`susplot (`*`mdl1`*`, `*`mdl2`*`)` [Function File]
`susplot (`*`mdl1`*`, `*`mdl2`*`, a)` [Function File]
`susplot (`*`mdl1`*`, `*`mdl2`*`, a, `*`numbers`*`)` [Function File]
`pdata` `= susplot (...)` [Function File]

Builds a Shared and Unique Structure (SUS)-plot from (O)PLS modeled data to facilitate identification of variables that covary and contravary in two models. An optional third argument *a* may be passed to select which predictive component is to be analyzed. The default behavior is to plot the first predictive component. An optional fourth argument (*numbers*) may be set to true (default is false) to change the plotted points into variable numbers.

`rqplot (mdl)` [Function File]
`h = rqplot (mdl)` [Function File]
Builds a bar plot of R^2/Q^2 values from a PCA, PLS, OPLS or LDA model.

`permscatter (S)` [Function File]
Plots information in S that has been calculated by [permtest], page 49, in a scatter plot format.

`permdensity (S)` [Function File]
Plots information in S that has been calculated by [permtest], page 49, using kernel density estimation to reconstruct the null distributions generated by permutation.

7.3.2 Coloring

`colors = clscolors (Y)` [Function File]
Uses the Y matrix created by [classes], page 52, to build different colors for each class.

`colors = obscolors (XY)` [Function File]
Uses the XY matrix (either X or Y) to build different colors for each observation, rainbow-style.

`colors = varcolors (X)` [Function File]
Uses the X matrix to build different colors for each variable, rainbow-style.

`colors = nocolors (XY)` [Function File]
Uses the XY matrix (either X or Y) to build a matrix of blackness for a scatter plot.

7.4 Model validation

Validation is a critical step in the process of using supervised multivariate models, such as those produced by PLS or OPLS. Standard n-fold cross-validation is automatically performed by [pca], page 37, [pls], page 38, and [opls], page 39, producing R^2 and Q^2 statistics that may be extracted with [rq], page 48, or [rqdiff], page 48. Furthermore, PLS and OPLS models must be further validated by CV-ANOVA or permutation testing.

`v = rq (mdl)` [Function File]
Returns the calculated R^2/Q^2 values from a PCA, PLS OPLS or LDA model.

`v = rqdiff (mdl)` [Function File]
Returns the calculates R^2/Q^2 values from a PCA, PLS or OPLS model, but gives differential values. In other words, this function gives the R^2 and Q^2 values per-component, not cumulative.

`S = cvanova (mdl)` [Function File]
Uses an already performed internal cross-validation of a PLS or OPLS model to calculate a CV-ANOVA p-value. The output structure S contains the following information:

`S.SS`: cross-validated sum of squares.
`S.MS`: cross-validated mean square errors.
`S.DF`: degrees of freedom.

`S.F`: f-statistic from cross-validated mean square errors.
`S.p`: p-value indicating how well the model fits the data.

L. Eriksson, J. Trygg, S. Wold. 'CV-ANOVA for significance testing of of PLS and OPLS models.' J. Chemometrics 2008(22): 594-600.

`S = permtest (mdl)` [Function File]
`S = permtest (mdl, n)` [Function File]

Uses response permutation testing of a PLS or OPLS model to calculate the significance of the model parameters. The output structure S contains the following information:

`S.n`: number of permutations performed.
`S.r`: permutation Y-correlation coefficients.
`S.Rsq.orig`: original R^2 value.
`S.Qsq.orig`: original Q^2 value.
`S.Rsq.perm`: permutation R^2 values.
`S.Qsq.perm`: permutation Q^2 values.
`S.Rsq.t`: R^2 t-statistic.
`S.Qsq.t`: Q^2 t-statistic.
`S.Rsq.p`: R^2 t-test p-value.
`S.Qsq.p`: Q^2 t-test p-value.

An optional second argument n may be passed to set the number of permutations to execute. The default number of permutations is 100.

The structure S generated by this function may be passed to [permscatter], page 48, or [permdensity], page 48, for visualization of the permutation test results.

7.5 Model manipulation

Supplementary data such as class identity matrices and class labels may be added to models for use by various plotting and analysis routines.

On the other hand, scores, loadings and other model components may be extracted for use in other analyses.

The available functions for manipulating model structures are documented below.

7.5.1 Adding data

`mdlAdd = addclasses (mdl, Y)` [Function File]
`mdlAdd = addclasses (mdl, Y, overwrite)` [Function File]

Adds a supplementary class matrix to a PCA, PLS, *etc* model. This function will not work for supervised models that already contain a class matrix unless the *overwrite* argument is set to `true`.

The function is intended to be used like so:

```
mdl = addclasses(mdl, Y);
```

mdlAdd = addlabels (*mdl*, *labels*) [Function File]
Adds a supplementary string cell array (*labels*) to a PCA, PLS, *etc* model. The labels will be placed on scores plots and written to saved scores files. If *mdl* already contains a labels array, it will be replaced with *labels* without warnings.

The function is to be used as follows:
```
mdl = addlabels(mdl, labels);
```

7.5.2 Extracting data

T = scores (*mdl*) [Function File]
T = scores (*mdl*, *n*) [Function File]
Returns the calculated scores values from a PCA, PLS or OPLS model. When *n* is provided as a second argument, only *n* columns will be returned in *T*. Otherwise, the full matrix of scores will be returned.

P = loadings (*mdl*) [Function File]
P = loadings (*mdl*, *n*) [Function File]
Returns the calculated loadings values from a PCA, PLS or OPLS model. When *n* is provided as a second argument, only *n* columns will be returned in *P*. Otherwise, the full matrix of loadings will be returned.

W = weights (*mdl*) [Function File]
W = weights (*mdl*, *n*) [Function File]
Returns the calculated weights values from a PCA, PLS or OPLS model. When *n* is provided as a second argument, only *n* columns will be returned in *W*. Otherwise, the full matrix of weights will be returned.

T = cvscores (*mdl*) [Function File]
T = cvscores (*mdl*, *n*) [Function File]
Returns the cross-validated scores from a PLS model in the form of an array. Each element of the array contains the reconstructed scores from a monte carlo leave-n-out cross validation. The length of the array equals the number of cross-validation iterations.

7.6 Classes and labels

Multivariate discriminant analysis algorithms like PLS and OPLS require a class membership matrix that defines which class each data matrix observation belongs to. Moreover, textual labels of the classes in a model are required when visualization is performed. The following functions are useful for creating those data structures for addition to models.

Y = classes ([*n1*, *n2*, ..., *nM*]) [Function File]
Creates a Y-matrix suitable for discriminant analysis, where class membership is denoted by a 1 in the corresponding class column. One vector argument is required, where the number of elements in the vector equals the number of classes and each element in the vector gives the number of observations in that class.

labels = loadlabels (*fname*) [Function File]
[*labels*, *indices*] = loadlabels (*fname*) [Function File]

`[labels, indices, Y] = loadlabels (fname)` [Function File]

Loads in class label assignments for a dataset from a text file, where the n'th line in the text file contains the class label of the n'th observation.

An optional second return value (*indices*) can be requested that contains the row indices that will bring the associated data matrix into sync with the class membership matrix, like so: `X = X(indices,:);`

An optional third return value (Y) can be requested that contains the binary class membership matrix used during PLS.

7.7 Separations

Methods of quantifying class separations in (*validated*) multivariate models are available in MVAPACK, documented below.

`j2v = j2 (mdl)` [Function File]
`j2v = j2 (mdl, Y)` [Function File]

Computes the J_2 clustering statistic (ratio of the determinants of entire dataset covariance to each cluster covariance) on the computed scores of a PCA, PLS or OPLS model. PCA models need an accompanying Y-matrix to define classes. It may either be passed as a second argument to this function or added to the model (See [addclasses], page 49).

For more information on the J_2 metric, see: K. Koutroumbas, S. Theodoridis. 'Pattern Recognition'. Elsevier Press, Amsterdam, 2006.

`D = overlaps (mdl, k)` [Function File]
`D = overlaps (mdl, k, Y)` [Function File]

Computes a p-value matrix between all class scores of a PCA, PLS or OPLS model. PCA models need an accompanying Y-matrix to define classes (See [addclasses], page 49).

The default number of components (k) will be set to the model component count, unless specified in the arguments.

`D = distances (mdl)` [Function File]
`D = distances (mdl, k)` [Function File]
`D = distances (mdl, k, metric)` [Function File]
`D = distances (mdl, k, metric, Y)` [Function File]

Computes a distance matrix between all class scores of a PCA, PLS or OPLS model. PCA models need an accompanying Y-matrix to define classes.

The default number of components (k) will be set to the model component count, unless specified in the arguments. The default metric is the squared Mahalanobis distance.

`retval = euclidean (X, y)` [Function File]

Return the squared Euclidean distance between the means of the multivariate samples x and y, which must have the same number of components (columns), but may have a different number of observations (rows).

8 Metabolite identification

8.1 Metabolite identification

The process of identifying and quantifying metabolites in complex biological mixtures is challenging and time-consuming due to large number of metabolite candidates in available database and potential peak shiftings from their theoretical positions. Shifting errors are inherited in almost every metabolomics experiment due to changes in pH levels, temperatures, and instrument instability. We propose a method with a search window of size to account for the peak drifting and a regulization term to shrink concentration of metabolites not contributing to the mixture to 0. Metabolite concentration coeffients are estimated by optimizing this following objective function:

`Y = classes ([n1, n2, ..., nM])` [Function File]

Creates a Y-matrix suitable for discriminant analysis, where class membership is denoted by a 1 in the corresponding class column. One vector argument is required, where the number of elements in the vector equals the number of classes and each element in the vector gives the number of observations in that class.

`labels = loadlabels (fname)` [Function File]
`[labels, indices] = loadlabels (fname)` [Function File]
`[labels, indices, Y] = loadlabels (fname)` [Function File]

Loads in class label assignments for a dataset from a text file, where the n'th line in the text file contains the class label of the n'th observation.

An optional second return value (*indices*) can be requested that contains the row indices that will bring the associated data matrix into sync with the class membership matrix, like so: `X = X(indices,:);`

An optional third return value (Y) can be requested that contains the binary class membership matrix used during PLS.

$$\sum_{i=1}^{n}\sum_{k=i-d}^{k=i+d} w_{ik}(y_i - \sum_{j=1}^{M}\beta_j g_j(x_k))^2 + \lambda\sum_{j=1}^{M}|\beta_j|$$

Where y_{nx1} is the mixture spectrum; d is pre-defined size of search window set for each i^{th} peak; $g_j(x_k)$ is intensity of signal at k^{th} peak on spectrum of the j^{th} reference metabolite; β_j is concentration coeffient of the j^{th} reference metabolite; λ is regularized term.

Reference spectrum for the j^{th} metabolite can be generated by taking advantages of Lorentzian curve and peak information from public database using this following function:

`sim_spec = sim_cauchy(x, y, ppm, s)` [Function File]

x and y are peak locations and corresponding intensities of a given metabolite obtained from public database e.g. HMDB; ppm is all chemical shift values; s is shape parameter with default value of 0.002. Metabolite concentration coefficients β_j's are estimated using coordinate descent. In other words, we perform the algorithm by taking gradient of the above objective function with respect one β_j at a time. More information about this approach can be found here:

Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models viacoordinate descent. Journal of statistical software, 33(1):1, 2010.

The weight matrix w plays an important role in locating any potential peak shifting by assigning more weight on any search location that yields smaller difference between the observed and reference spectrum.

`[beta, weight] = coordinate_lasso1(theta, X, y, peaks_id, lambda, num_iters, d, sig, sc)` [Function File]

Estimating betas and weight matrix via coordinate descent Estimates concentration coefficients and weight matrix in beta_hat and w_matrix by provided initial values of β_j in *theta*, matrix X_{nxM} consists of all M reference spectra, each in one column, y_{nx1} is the mixture spectrum we want to identify metabolites from, peaks_id comprises all indexes of peaks in reference matrix X, *lambda* is a regularized term chosen from cross validation; niter is the number of iterations for the estimation process, default value is 5; d is search window size, *sig* is tuning parameter for the weight matrix, recommended to be in the range of $[\frac{max(y)^2}{6}, \frac{max(y)^2}{3}]$. *sc* is an option to scale input data matrix with default set to True.

Similarly, tuning parameter can also be set to be adaptive to peak location. It can be built in to the function as follows:

`[beta, weight] = coordinate_lasso2(theta, X, y, peaks_id, lambda, num_iters, d, sc)` [Function File]

Estimating beta's and weight matrix via coordinate descent with adaptive sigma. This version is similar to *coordinate_lasso1()*, but is adapative to peak location.

`mpe = cv_coordinate1(theta, X, y, peaks_id, lambda, num_iters, d, sig, nfold)` [Function File]

k-fold Cross-validation to tune penalty lambda.

Here l_seq is the sequence of lambda values to be evaluated on; nfold is number of folds that is set to split the data matrix X into, default value is 4. Similar to above, tuning parameter can be adaptive throughout the cross validation using:

`mpe = cv_coordinate2(theta, X, y, peaks_id, lambda, num_iters, d, nfold)` [Function File]

k-fold Cross-validation to tune penalty lambda

9 GC/LC-MS workflows

Gas-chromatography/liquid-chromatography mass spectrometry (GC/LC-MS) is a popular analytical method for chemometric analysis and this release of MVAPACK features an expansion of functions to work with these types of experimental results.

Note that this is an incremental release of MVAPACK and there are a number of functions currently being developed that will be integrated into the package in another release later in 2021. Because this functionality is so new, there will likely be bugs and/or unmet needs by users. As these come up, do not hesitate to reach out and make our team aware of any issues.

9.1 General workflow

A typical analysis of GC/LC-MS data contains many steps and MVAPACK's structuring of these steps is listed below. While this ordering has worked for a variety of in-house datasets, please keep in mind that the ordering is subject to change for any given dataset and that other packages (commercial and academic) will vary the ordering slightly:

1. **Extracted Ion Chromatogram (EIC) Generation**

   EICs are thin slices of signal whose mass-to-charge (m/z) ratios are meant to be narrow enough to contain a single ion. This process involves reading in data from one of the various open-source formats and creating folders of .csv files with the relevant data points.

2. **Peak Picking**

   With EICs generated, peaks can be picked using a variety of signal processing methods. These methods typically locate a retention time and exact m/z value for the peak as well as information on peak width and signal-to-noise.

3. **Peak Filtration**

   It is not uncommon for each replicate in an experiment to generate thousands or tens of thousands of peaks and it is often necessary to remove peaks based on factors like total intensity, integration, width or any other peak variable.

4. **Peak Alignment**

   GC/LC-MS spectrometers often experience drift over time and it is common for two identical peaks in different replicates. To address this issue, it is often necessary to adjust retention times between replicates so that data matrices can be generated and further analysis can be performed.

5. **Matrix Generation**

   With all peaks aligned, a feature matrix can be generated. In this step, a set of features common in the replictes are selected and a large matrix is generated wherein each row is a feature and the columns correspond to the different replicates in the experiments. The resulting matrix holds the peak integrations for each replicate at each feature.

6. **Feature Normalization**

   The significant variation in intensities across GC/LC-MS experiments necessitates normalization to extract meaningful information. Once the matrix is generated, the features can be normalized.

7. **Feature Imputation**

   Biological and technical replicates are frequently missing peaks found in other replicates of the same experimental group. To overcome this issue and avoid losing features, imputation is performed by interploating missing peaks that are found in at least 80% of other replicates in an experimental group.

8. **Principal Component Analysis (PCA) Modelling**

   Once a feature/data matrix has been generated, normalized and imputed, the matrix can be analyzed with a variety of statistical methods to infer biological meaning. A popular method is PCA analysis, which collapses variance in feature intensities across hundreds of peaks into two dimensions for convenient visualization.

9.2 Working with GC/LC-MS data

MVAPACK's GC/LC-MS functionality is designed to work with the open source .mzML, .mzXML and proteoWizard .txt formats. Most instruments save their data in proprietary formats so you will likely have to convert your datasets with the proteoWizard utility.

The application you will be primarily using from proteoWizard `msconvert`, which has both a commandline and GUI version. Examples for the commandline version can be seen here. Note that `msconvert` relies on a variety of dll's and can only be run natively on windows. It is possible to process files with WINE, but this is typically complicated and it is generally better to perform this step on a native Windows machine.

Lastly, it's worth noting that the .mzML format is generally preferable. The .mzXML format is also common but .mzML has been designated as the future format in the proteomics/metabolomics field. Both of these formats use compression to reduce file size, but it is possible to use the plaintext proteoWizard format, though it should be noted that this will result in greater storage needs.

Appendix A Function index

This appendix lists all functions implemented in the MVAPACK toolbox. If a function is mentioned in the main body of this manual, a link will be placed to that mention. Otherwise, the full function documentation is given in the appendix.

Please remember that not all functions below are directly useful during data handling, as they are typically subroutines of the truly user-friendly functions in MVAPACK. As a general rule: if it's mentioned in the main body of this manual, it's meant to be directly used.

A.1 A

p = acquparms_agilent (*dirname*, *fbase*, *dim*) [Function File]
Reads values from the key-value pairs found in the Agilent 'procpar' file from a filename provided by the [acquparms], page 21, function. It is highly recommended that you use [acquparms], page 21, instead of this function.

p = acquparms_bruker (*dirname*, *fbase*, *dim*) [Function File]
Reads values from the key-value pairs found in the Bruker 'acqus' file from a filename provided by the [acquparms], page 21, function. It is highly recommended that you use [acquparms], page 21, instead of this function.

p = acquparms (*dirname*) [Function File]
See [acquparms], page 21.

mdlAdd = addclasses (*mdl*, *Y*) [Function File]
mdlAdd = addclasses (*mdl*, *Y*, *overwrite*) [Function File]
See [addclasses], page 49.

mdlAdd = addlabels (*mdl*, *labels*) [Function File]
See [addlabels], page 50.

wfid = apodize1d (*fid*, *parms*, *fn*, *opts*) [Function File]
Performs apodization of a one-dimensional time-domain NMR free-induction decay in order to alleviate truncation artifacts that can arise from Fourier transformation.

Instead of using this function directly, it is recommended that you use [apodize], page 23.

wfid = apodize2d (*fid*, *parms*, *fn*, *opts*) [Function File]
Performs apodization of a two-dimensional time-domain NMR free-induction decay in order to alleviate truncation artifacts that can arise from Fourier transformation.

Instead of using this function directly, it is recommended that you use [apodize], page 23.

wfid = apodize (*fid*, *parms*) [Function File]
wfid = apodize (*fid*, *parms*, *fn*) [Function File]
wfid = apodize (*fid*, *parms*, *fn*, *opts*) [Function File]
See [apodize], page 23.

`[sp, phc0, phc1] = autophase1d (s, objective)` [Function File]
Corrects the phase of a one-dimensional Fourier-transformed spectrum or spectral dataset found by simplex optimization. The method uses an entropy minimization objective during optimization (See [simplex_entropy], page 26).

Instead of using this function directly, it is recommended that you use [autophase], page 26.

`[sp, phc0, phc1] = autophase2d (s, objective)` [Function File]
Corrects the phase of a two-dimensional Fourier-transformed spectrum or spectral dataset found by simplex optimization. The method uses a whitening objective during optimization (See [simplex_whiten], page 27).

Instead of using this function directly, it is recommended that you use [autophase], page 26.

`sp = autophase (s, parms)` [Function File]
`[sp, phc0, phc1] = autophase (s, parms)` [Function File]
`sp = autophase (s, parms, objective)` [Function File]
`[sp, phc0, phc1] = autophase (s, parms, objective)` [Function File]
See [autophase], page 26.

A.2 B

`Y = backscaleclasses (mdl)` [Function File]
Recover the original discriminant analysis class matrix Y see [classes], page 52, from a scaled, centered version using [backscale], page 57.

`B = backscale (A, center, scale)` [Function File]
`B = backscale (mdl)` [Function File]
`[B, m] = backscale (mdl)` [Function File]
`[B, m, s] = backscale (mdl)` [Function File]
Undo a scaling operation performed while building a model of the matrix A. The columns (variables) of A must match the lengths of the vectors *center* and *scale*. The inputs may either be a complete set of data to perform backscaling or a PCA, PLS or OPLS model. In the latter case, the extracted centering and scaling vectors used during backscaling may be optionally returned as m and s as well.

`backscaleplot (ab, mdl)` [Function File]
`backscaleplot (ab, mdl, coloring)` [Function File]
`pdata = backscaleplot (ab, mdl, coloring)` [Function File]
See [backscaleplot], page 46.

`noise = baseline_noise_estimation(raw, res=0.10)` [Function File]
Estimates the noise-to-signal (NS) ratio of a series of extracted ion chromatogram (EIC) counts. This value is useful for baseline noise subtraction as a higher NS value means more aggressive parameters should be used. Note that this function assumes the inputted values are uniformly spaced with respect to time and this should be indicated by the second parametere. This value is in seconds and the default value is 0.10.

z = *baseline_subtraction*(*y*, *lambda*, *p*) [Function File]
Performs basline subtraction from an extracted ion chromatogram (EIC) by first esitmating the baseline with an asymmetric least squares (ALS) fit. This type of fitting is designed to fit only the baseline of a series by under-penalizing high values that are likely to be signal. The parameters p and lambda correspond to asymmetry and smoothness, respectively. It is suggested that p and lambda be on the ranges of [0.001, 0.1] and [10^2, 10^9], respectively. This implementation is based off of the following paper:

> Eilers, P. H. C.; Boelens, H. F. M. Baseline Correction with Asymmetric Least Squares Smoothing, 2005.

roi = bin2roi (*abnew*, *widths*) [Function File]
See [bin2roi], page 27.

xnew = binadapt1d (*X*, *ab*, *w*) [Function File]
[*xnew*, *abnew*] = binadapt1d (*X*, *ab*, *w*) [Function File]
[*xnew*, *abnew*, *widths*] = binadapt1d (*X*, *ab*, *w*) [Function File]
Adaptively bin a one-dimensional spectrum or spectral dataset.

It is highly recommended that you use [binadapt], page 30, instead of this function directly.

xnew = binadapt2d (*X*, *ab*, *w*) [Function File]
[*xnew*, *abnew*] = binadapt2d (*X*, *ab*, *w*) [Function File]
[*xnew*, *abnew*, *widths*] = binadapt2d (*X*, *ab*, *w*) [Function File]
Adaptively bin a two-dimensional spectrum or spectral dataset.

It is highly recommended that you use [binadapt], page 30, instead of this function directly.

xnew = binadapt (*X*, *ab*, *parms*, *w*) [Function File]
[*xnew*, *abnew*] = binadapt (*X*, *ab*, *parms*, *w*) [Function File]
[*xnew*, *abnew*, *widths*] = binadapt (*X*, *ab*, *parms*, *w*) [Function File]
See [binadapt], page 30.

xnew = binmanual1d (*X*, *ab*, *roi*) [Function File]
[*xnew*, *abnew*] = binmanual1d (*X*, *ab*, *roi*) [Function File]
[*xnew*, *abnew*, *widths*] = binmanual1d (*X*, *ab*, *roi*) [Function File]
Manually bin a one-dimensional spectrum or spectral dataset in X based on regions of interest provided in *roi*, or centers and widths provided in *centers* and *widths*. If regions of interest are used to bin, the *abnew* and *widths* values are optionally returnable.

xnew = binmanual2d (*X*, *ab*, *roi*) [Function File]
[*xnew*, *abnew*] = binmanual2d (*X*, *ab*, *roi*) [Function File]
[*xnew*, *abnew*, *widths*] = binmanual2d (*X*, *ab*, *roi*) [Function File]
Manually bin a two-dimensional spectrum or spectral dataset in X based on regions of interest provided in *roi*, or centers and widths provided in *centers* and *widths*. If regions of interest are used to bin, the *abnew* and *widths* values are optionally returnable.

A.3 C

- *min_intensity* denotes the lowest max intensity an EIC must have to be saved.

- *filt_type* indicates the type of raw data filtration to be carried out. Valid options are 'base_peak' or 'absolute'.

- *cutoff* indicates the cutoff that will be used to filter the raw data. For 'base_peak' mode, this decimal will be multiplied by the base peak to get a cutoff whereas for 'absolute', it is the specified value.

`Xn = csnorm (X)` [Function File]
`[Xn, s] = csnorm (X)` [Function File]
See [csnorm], page 32.

`S = cvanova (mdl)` [Function File]
See [cvanova], page 48.

`mpe = cv_coordinate1(theta, X, y, peaks_id, lambda, num_iters, d, sig, nfold)` [Function File]
See [cv_coordinate1], page 53.

`mpe = cv_coordinate2(theta, X, y, peaks_id, lambda, num_iters, d, nfold)` [Function File]
See [cv_coordinate2], page 53.

`idx = cvindices (N, Np)` [Function File]
Creates a vector of group indices from 1 to *Np*, where each value in the vector is the index of the training set to which the point will belong during cross-validation.

`X = cvjoin (Xt, Xv, idx)` [Function File]
Rejoins a one-dimensional array of datasets back into a single dataset, where the array of indices *idx* was generated using the appropriate functions (See [cvindices], page 61).

`CV = cvldainit (mdl)` [Function File]
Initializes and returns a cell array used for internal cross-validation of LDA models.

`[Qsq, Qstd, CV] = cvlda (mdl, Amax)` [Function File]
Performs internal cross-validation of an LDA model and returns a Q^2 value for inferring model reliability. The *CV* cell array returned is a modified version of that found in the passed model.

`CV = cvoplsinit (mdl)` [Function File]
Initializes and returns a cell array used for internal cross-validation of OPLS models.

`[Qsq, Qstd, CV] = cvopls (mdl, V, Ao)` [Function File]
Performs internal cross-validation of a PLS or OPLS model and returns a Q^2 value for inferring model reliability. The *CV* cell array returned is a modified version of that found in the passed model.

`[Xt, Xv] = cvsplit (X, idx)` [Function File]
Creates a one-dimensional array of datasets from a single dataset, where the array of indices *idx* was generated using the appropriate functions (See [cvindices], page 61).

`C = cwt (x)` [Function File]
`C = cwt (x, w)` [Function File]
`C = cwt (x, w, voices)` [Function File]
`C = cwt (x, w, voices, octave)` [Function File]
`C = cwt (x, w, voices, octave, scale)` [Function File]
`C = cwt (x, w, voices, octave, scale, doifft)` [Function File]
Performs continuous wavelet transformation of a vector signal to produce a time-frequency transform (CWT) matrix, *C*. The default wavelet is the Mexican hat (See [wavelet_sombrero], page 89), but can be set to another function by passing *w* with a function handle of the form:

```
function psi = wavelet_function (t) ... end
```

The optional argument *voices* sets the number of voices per octave, and defaults to 8. The optional argument *octave* sets the initial octave, and defaults to 4. The optional argument *scale* sets the initial scale, and defaults to 8. You can tweak these numbers to your satisfaction.

The final optional argument *doifft* sets whether the function should perform a final inverse fourier transform. This is useful if you wish to calculate derivatives in frequency space. By default, *doifft* is set to `true`.

A.4 D

`X = data2roi1d (Xroi, ab, roi)` [Function File]
Uses one-dimensional spectral data inside regions of interest from a data matrix to reconstruct portions of a full spectral dataset.

`X = data2roi2d (Xroi, ab, roi)` [Function File]
Uses two-dimensional spectral data inside regions of interest from a data matrix to reconstruct portions of a full spectral dataset.

`X = data2roi (Xroi, ab, parms, roi)` [Function File]
See [data2roi], page 28.

`T = decompose1d (f, t, parms)` [Function File]
`T = decompose1d (f, t, parms, roi)` [Function File]
`T = decompose1d (f, t, parms, roi, minbw)` [Function File]
`T = decompose1d (f, t, parms, roi, minbw, fitopts)` [Function File]
Performs Complete Reduction to Amplitude and Frequency Table (CRAFT) analysis of a 1D time-domain NMR free induction decay (FID) in *f*, with a time abscissa in *t* and parameters in *parms*.

It is highly recommended that you use [decompose], page 64, instead of calling this function directly.

`T = decompose2d (f, t, parms)` [Function File]
`T = decompose2d (f, t, parms, roi)` [Function File]

`T = decompose2d (f, t, parms, roi, minbw)` [Function File]
`T = decompose2d (f, t, parms, roi, minbw, fitopts)` [Function File]

Performs Complete Reduction to Amplitude and Frequency Table (CRAFT) analysis of a 2D time-domain NMR free induction decay (FID) in *f*, with a time abscissa in *t* and parameters in *parms*.

It is highly recommended that you use [decompose], page 64, instead of calling this function directly.

`T = decompose (f, t, parms)` [Function File]
`T = decompose (f, t, parms, roi)` [Function File]
`T = decompose (f, t, parms, roi, minbw)` [Function File]
`T = decompose (f, t, parms, roi, minbw, fitopts)` [Function File]

Performs Complete Reduction to Amplitude and Frequency Table (CRAFT) analysis of a time-domain NMR free induction decay (FID) in *f*, with a time abscissa in *t* and parameters in *parms*. If multiple decays are provided in *f*, then a joined table reflecting data from all decays will be returned.

In the one-dimensional case, data in *f* may either be a column vector or a data matrix where each observation is arranged as a row in the matrix.

In the two-dimensional case, data in *f* may either be a data matrix where each direct-dimension slice is along the rows, or a cell array that contains multiple matrices, each having direct-dimension slices along its rows.

The optional argument *roi* may be passed to specify either a function handle for automated region of interest (ROI) selection, or a matrix of manually defined regions of interest. Each manually defined region should be a two-element row in *roi* containing the lower and upper frequency values (in hertz, See [nmrft], page 25). Two-dimensional data should have ROI rows with four numbers (max and min for each dimension). In the case of a function handle, *roi* must be a function defined as follows:

```
function roi = roi_function (s, ab, parms, wmin) ... end
```

The optional argument *minbw* may be passed to specify a minimum bandwidth (in Hertz) of the automatically selected ROIs. The default value of *minbw* is one one-hundredth of the total spectral width.

The CRAFT algorithm is implemented according to information reported in:

> Krishnamurthy K., 'CRAFT (complete reduction to amplitude frequency table) - robust and time-efficient Bayesian approach for quantitative mixture analysis by NMR', Magnetic Resonance in Chemistry, 2013.

`D = distances (mdl)` [Function File]
`D = distances (mdl, k)` [Function File]
`D = distances (mdl, k, metric)` [Function File]
`D = distances (mdl, k, metric, Y)` [Function File]

See [distances], page 51.

`Fcorr = dmxcorr (F, parms)` [Function File]

See [dmxcorr], page 22.

A.5 E

A.6 F

idx = findnearest (*x*, *a*) [Function File]
See [findnearest], page 31.

f = flatness (*X*) [Function File]
f = flatness (*X*, *ab*, *roi*) [Function File]
Calculate the spectral flatness, the ratio of the geometric mean to the arithmetic mean of a signal. The optional arguments *ab* and *roi* may be used to specify regions of interest for which to calculate flatness, instead of the entire data matrix (default).

If a single vector is provided in *X*, one flatness will be returned for each ROI. If a data matrix is provided, each observation (row) in the matrix will return its own set of flatness values.

A.7 G

noise = gauss_baseline_noise(*cts*, *filtered*, *sn_cutoff*) [Function File]
Estimates the noise-to-signal (NS) ratio of a series of extracted ion chromatogram (EIC) counts given the *cts* and the corresponding *filtered* values. This method of noise estimation also requires a *sn_cutoff* which should be somewhat less than the cutoff used for peak picking.

peaks = gauss_deriv_peak_picking(*pathname*, *peaks*, [Function File]
sn_cutoff=10, *peaks_per_slice=5*, *interp_res=0.25*)
Performs peak picking on a .csv file containing an extracted ion chromatogram (EIC) created by the create_eics() function. The peak picking performed here is based on the procedure used by the creators of XCMS. Below is a summary of the relevant parameters:

- *pathname*: path to the .csv file containing the EIC.

- *peaks*: existing peaks to be appended to the output matrix.

- *sn_cutoff*: cutoff used to accept/reject peaks. Default value is 10.

- *peaks_per_slice*: maximum number of peaks per EIC. Only the most intense peaks also above the signal cutoff are selected.

- *interp_res*: interpolation resolution for the EIC data. Value is in seconds.

To read more about XCMS and their peak picking strategy, see the below citation:

> Colin Smith, Elizabeth Want, Grace O'Maillle, Ruben Abagyan and Gary Sizudak. XCMS: Processing Mass Spectrometry Data for Metabolite Profiling Using Nonlinear Peak Alignment, Matching, and Identification. Anal. Chem. 2006, 78 (3), 779–787

[coefs] = gaussian_coefficients(*res*, *sigma*=10, *width=3*, [Function File]
amplitude=0.3)
Returns a second derivative guassian distribution centered at time = 0 seconds. This wavelet is used for peak picking and noise estimation. Note that for this to be useful the *res* must be the same as the EIC data on which it is later applied. Below is a summary of its input parameters:

- *res* refers to time resolution. Measure in seconds, common values are 0.05-0.25. MUST be same as data it is being used on.

- *sigma* refers to the standard deviation of the distribution. Measured in seconds.

- *width* refers to the one tailed width of the wavelet. The overall length of the wavelet in seconds is 2**width***sigma*.

- *amplitude* the amplitude that the wavelet is normalized to.

`w = gausswindow (t, lb)` [Function File]
See [gausswindow], page 24.

`ppm = genppm (n, parms)` [Function File]
Uses spectral parameters for number of real data points (n), spectral width and carrier offset in ppm to build a vector of chemical shifts for NMR spectra.

This function returns only a single dimension axis at a time. Thus, *parms* must be a structure.

`t = gentime (n, parms)` [Function File]
Uses spectral parameters for number of real data points (n), spectral width (sw) and carrier offset in ppm (car) to build a vector of chemical shifts for NMR spectra.

This function returns only a single dimension axis at a time. Thus, *parms* must be a structure.

A.8 H

`hada_prod = hadamard(A, B)` [Function File]
This function calculates element-wise Hadamard product of two equal-size matrices. This product is only defined for two matrices of equal size. It will exit with an error otherwise. More information can be found here:

> Smilde, Age K., Rasmus Bro, and Paul Geladi. Multi-way analysis with applications in the chemical sciences. Chichester, West Sussex, England Hoboken, NJ: J. Wiley, 2004.

`err = histmatch_func (alpha, Ht, Xs, Zmap)` [Function File]
This is the objective function for [histmatch], page 32. You'll never call this function directly, if you're writing sane code.

`Xn = histmatch (X)` [Function File]
`[Xn, s] = histmatch (X)` [Function File]
See [histmatch], page 32.

A.9 I

`Xc = icoshift (X, ab)` [Function File]
`Xc = icoshift (X, ab, seg)` [Function File]
`[Xc, lags] = icoshift (X, ab)` [Function File]
`[Xc, lags] = icoshift (X, ab, seg)` [Function File]
`[Xc, lags] = icoshift (X, ab, seg, cofirst)` [Function File]
See [icoshift], page 29.

idx = id_create(*id*, *d*, *n*) [Function File]
Generate neighbor indexes of a given peak within window size d

idx = id_seq(*id*, *d*, (*n*)) [Function File]
Generate neighbor indexes of a given peak within window size d

[ivals] = impute(*vals*, *groups*, *max_missing=0.20*, *how='mean'*) [Function File]
Wrapper function for imputing missing values in GC/LC-MS data matrices. The *vals* matrix is assumed to have rows which correspond to features and columns which correspond to replicates. The *groups* is a one dimensional matrix that indicates the experimental groupings for each of the replicates in the columns. *max_missing* indicates the maximum fraction of missing entries that occur for a given feature for imputation to occur. The default value is 0.20 and this is the max recommended value. *how* indicates the type of imputation at present the only allowed version is 'mean' which uses the group average for missing values.

[ivals] = impute_mean(*vals*, *groups*, *max_missing=0.20*) [Function File]
Method which performs a group-wise mean imputation for missing values in a GC/LC-MS data matrix. This function can be called independently but it is recommended that end users instead use the *impute()* function and specify the *how* method as 'mean'. The *groups* is a one dimensional matrix that indicates the experimental groupings for each of the replicates in the columns. *max_missing* indicates the maximum fraction of missing entries that occur for a given feature for imputation to occur. The default value is 0.20 and this is the max recommended value.

Imax = integrals (*X*, *ab*, *roi*) [Function File]
[*I*, *Iab*] = integrals (*X*, *ab*, *roi*) [Function File]
See [integrals], page 28.

integralsplot (*x*, *ab*, *Ix*, *Iab*) [Function File]
See [integralsplot], page 28.

tf = ismultiblock (*mdl*) [Function File]
Returns whether a model *mdl* is of the 'multiblock' variety.

tf = isnus (*x*) [Function File]
Returns whether a vector is non-uniformly sampled or not. An abscissa vector *x* is deemed non-uniformly sampled when any absolute difference between consecutive points is greater than or equal to twice the minimum difference between all its consecutive points.

y = ist (*x*, *sched*) [Function File]
y = ist (*x*, *sched*, *opts*) [Function File]
See ⟨undefined⟩ [ist], page ⟨undefined⟩.

opts = ist_options () [Function File]
Returns the default options for running IST reconstruction.

A.10 J

j2v = j2 (*mdl*) [Function File]
j2v = j2 (*mdl*, *Y*) [Function File]
See [j2], page 51.

A.11 K

krao_prod = krao(*A*, *B*) [Function File]
This function calculates the Khatri-Rao product of two matrices. Note that these two matrices must be the same size. More information can be found here:

> Smilde, Age K., Rasmus Bro, and Paul Geladi. Multi-way analysis with applications in the chemical sciences. Chichester, West Sussex, England Hoboken, NJ: J. Wiley, 2004.

A.12 L

res = lambda_max1(*X*, *y*, *peaks_id*, *d*) [Function File]
Obtain the maximum lambda value such that all beta estimates are 0.

Y = ldaclassify (*mdl*, *X*) [Function File]
Y = ldaclassify (*P*, *U*, *X*) [Function File]
[*Y*, *T*] = ldaclassify (*mdl*, *X*) [Function File]
[*Y*, *T*] = ldaclassify (*P*, *U*, *X*) [Function File]
Predicts responses Y from one or more observations X based on the LDA either provided in *mdl* or as P and U. The observations in X are transformed into the discriminant space and classified based on Euclidean distances to the model classes.

NOTE: this function is not meant to be used directly. If you want to use an LDA model to classify new observations, use [classify], page 45.

[*P*, *D*, *U*] = ldacomp (*X*, *Y*) [Function File]
Extracts a multiclass LDA decomposition from a data and response matrix. This function is not to be used directly; it is a subroutine of [lda], page 40.

mdl = lda (*X*, *Y*) [Function File]
mdl = lda (*X*, *Y*, *scalefn*) [Function File]
mdl = lda (*X*, *Y*, *scalefn*, *ncv*) [Function File]
mdl = lda (*X*, *Y*, *scalefn*, *ncv*, *aout*) [Function File]
See [lda], page 40.

[*Kresult*] = linear_kernel(*X*, *Y*) [Function File]
calculate dot product between any two matrices

f = loadnmr (*dirname*) [Function File]
[*f*, *parms*] = loadnmr (*dirname*) [Function File]
[*f*, *parms*, *t*] = loadnmr (*dirname*) [Function File]
[*f*, *parms*, *t*] = loadnmr (*dirname*, *doswap*) [Function File]
F = loadbruker (*dirnames*) [Function File]

`[F, parms] = loadnmr (dirnames)` [Function File]
`[F, parms, t] = loadnmr (dirnames)` [Function File]
`[F, parms, t] = loadnmr (dirnames, doswap)` [Function File]

Loads one or more Bruker or Agilent fid/ser files, automatically extracting parameters and automatically determining whether to parse Bruker or Agilent data.

Some older data needs to be byte-swapped when it is loaded in. To do this, pass *doswap* as `true` to this function. The default behavior is to skip the byte swap.

The parameters can be optionally returned if a second return value is requested. If a third optional return value is requested, the time abscissa (t) will be returned.

NOTE: Extreme care must be taken to ensure that the acquisition parameters of all experimental data specified in *dirnames* are totally identical! Only one parameter structure (*parms*) will be returned, so it is assumed that the first experiment holds parameters that are representative of the entire dataset.

`x = loadascii (filename)` [Function File]
`[x, ab] = loadascii (filename)` [Function File]
`X = loadascii (filenames)` [Function File]
`[X, ab] = loadascii (filenames)` [Function File]

Loads one or more ASCII files, each which contains a two-column, space-delimited format. The first column is expected to be an abscissa and the second column is expected to be the data.

`f = loaddmx (dirname)` [Function File]
`[f, parms] = loaddmx (dirname)` [Function File]
`[f, parms, t] = loaddmx (dirname)` [Function File]
`[f, parms, t] = loaddmx (dirname, correct)` [Function File]
`F = loaddmx (dirnames)` [Function File]
`[F, parms] = loaddmx (dirnames)` [Function File]
`[F, parms, t] = loaddmx (dirnames)` [Function File]
`[F, parms, t] = loaddmx (dirnames, correct)` [Function File]

See [loaddmx], page 22.

`fid = loadfid (filename, parms, doswap)` [Function File]

Loads a Bruker or Agilent fid file based on given parameters.

This function requires that nmrPipe be installed on the system and the environment variables required to run nmrPipe are set up.

`fid = loadfid2 (filename, parms, doswap, version)` [Function File]

Loads a Bruker or Agilent fid file based on given parameters.

This function requires that nmrPipe be installed on the system and the environment variables required to run nmrPipe are set up.

`fid = loadfid (filename, parms, doswap)` [Function File]

Loads a Bruker or Agilent fid file based on given parameters.

This function requires that nmrPipe be installed on the system and the environment variables required to run nmrPipe are set up.

ser = loadser (*filename*, *parms*, *doswap*) [Function File]
Loads a Bruker or Agilent ser file based on given parameters.

This function requires that nmrPipe be installed on the system and the environment variables required to run nmrPipe are set up.

s = lorentz (*ppm*, *par*) [Function File]
Simulates a perfectly phased complex Lorentzian peak over an abscissa *ppm* according to the parameter vector *par*. If *par* is a K-by-3 matrix, then K peaks will be simulated and summed.

The expected contents of each row of *par* are as follows:

`par(:,1)`: Chemical shift, in *ppm* units.
`par(:,2)`: Linewidth, in *ppm* units.
`par(:,3)`: Amplitude, in absolute units.

A.13 M

v = mad (*X*) [Function File]
calculate the median absolute deviation of a data matrix X: `MAD = med|x - med(x)|`

Y = mboplsclassify (*mdl*, *X*) [Function File]
[*Y*, *T*] = mboplsclassify (*mdl*, *X*) [Function File]
Predicts responses Y from one or more observations X based on the MBOPLS model provided in *mdl*. The observations in X are transformed by the regression coefficients (B) and classified based on sum of squares to the model classes.

NOTE: this function is not meant to be used directly. If you want to use a PLS model to classify new observations, use [classify], page 45.

mdl = mbopls (*X*, *Y*) [Function File]
mdl = mbopls (*X*, *Y*, *scalefn*) [Function File]
mdl = mbopls (*X*, *Y*, *scalefn*, *ncv*) [Function File]
mdl = mbopls (*X*, *Y*, *scalefn*, *ncv*, *aout*) [Function File]
See [mbopls], page 42.

Y = mbpcaclassify (*mdl*, *X*) [Function File]
[*Y*, *T*] = mbpcaclassify (*mdl*, *X*) [Function File]
Predicts responses Y from one or more observations X based on the MBPCA model provided in *mdl*. The observations in X are transformed into the principal component space and classified based on Mahalanobis distances to the model classes.

NOTE: this function is not meant to be used directly. If you want to use a PCA model to classify new observations, use [classify], page 45.

mdl = mbpca (*X*) [Function File]
mdl = mbpca (*X*, *scalefn*) [Function File]
mdl = mbpca (*X*, *scalefn*, *ncv*) [Function File]
mdl = mbpca (*X*, *scalefn*, *ncv*, *aout*) [Function File]
See [mbpca], page 41.

`Y = mbplsclassify (mdl, X)` [Function File]
`[Y, T] = mbplsclassify (mdl, X)` [Function File]

Predicts responses Y from one or more observations X based on the MBPLS model provided in *mdl*. The observations in X are transformed by the regression coefficients (B) and classified based on sum of squares to the model classes.

NOTE: this function is not meant to be used directly. If you want to use a PLS model to classify new observations, use [classify], page 45.

`mdl = mbpls (X, Y)` [Function File]
`mdl = mbpls (X, Y, scalefn)` [Function File]
`mdl = mbpls (X, Y, scalefn, ncv)` [Function File]
`mdl = mbpls (X, Y, scalefn, ncv, aout)` [Function File]

See [mbpls], page 41.

`f = mean2d (dirname)` [Function File]
`[f, parms] = mean2d (dirname)` [Function File]
`[f, parms, t] = mean2d (dirname)` [Function File]
`[f, parms, t] = mean2d (dirname, doswap)` [Function File]
`F = loadbruker (dirnames)` [Function File]
`[F, parms] = mean2d (dirnames)` [Function File]
`[F, parms, t] = mean2d (dirnames)` [Function File]
`[F, parms, t] = mean2d (dirnames, doswap)` [Function File]

Returns an average of a set of two-dimensional spectra.

`Xn = mscorr (X)` [Function File]
`Xn = mscorr (X, r)` [Function File]

See [mscorr], page 32.

`X = multiblock (X_1, ..., X_B)` [Function File]

Builds a multiblock data matrix from individual data matrices, ensuring that all dimensions match up.

A.14 N

`y = ndft (x, t)` [Function File]
`[y, w] = ndft (x, t)` [Function File]
`y = ndft (x, t, w)` [Function File]

Compute the non-uniform discrete Fourier transform of x using the brute-force ($O(N^2)$) NDFT algorithm.

The NDFT is calculated along the first non-singleton dimension of the array. Thus if x is a matrix, `ndft (x, t)` computes the NDFT for each column of x.

An optional second return value w may be requested that will hold the frequency domain axis, in whatever units correspond to those in t. Alternatively, a third argument (also w) may be passed to explicitly specify the frequencies at which to compute the NDFT.

`idx = nearest(v, k)` [Function File]

See ⟨undefined⟩ [nearest], page ⟨undefined⟩.

`n = nmrdims (x, parms)` [Function File]
Determines whether the dataset given by *x* and *parms* is one- or two-dimensional, based on the expected data types that each dimensionality of data may assume. The data *x* may be real or complex, time or frequency domain, *etc.*

`[s, ppm, hz] = nmrft1d (fid, parms, doshift)` [Function File]
Performs Fourier transformation and shifting to produce a 1D NMR spectrum or a 1D NMR spectral data matrix. One-dimensional data in *fid* may either be a column vector or a data matrix where each free induction decay is arranged as a row in the matrix.

Instead of using this function directly, it is recommended that you use [nmrft], page 25.

`[s, ppm, hz] = nmrft2d (fid, parms, doshift)` [Function File]
Performs Fourier transformation and shifting to produce a 2D NMR spectral matrix or a 2D NMR spectral cell array. Two-dimensional data must be arranged with slices of the direct-dimension along the rows.

Instead of using this function directly, it is recommended that you use [nmrft], page 25.

`s = nmrft (fid, parms)` [Function File]
`[s, ppm] = nmrft (fid, parms)` [Function File]
`[s, ppm, hz] = nmrft (fid, parms)` [Function File]
`... = nmrft (fid, parms, doshift)` [Function File]
See [nmrft], page 25.

`[recfid] = nmrist (fid, parms)` [Function File]
`[recfid] = nmrist (fid, parms, phc)` [Function File]
See [nmrist], page 23.

`colors = nocolors (XY)` [Function File]
See [nocolors], page 48.

`s = nusft (fid, t)` [Function File]
`[s, ppm] = nusft (fid, t, parms)` [Function File]
`[s, ppm, hz] = nusft (fid, t, parms)` [Function File]
Performs Fourier transformation and shifting to produce an NMR spectrum. The data in *fid* may either be a column vector or a data matrix where each free induction decay is arranged as a row in the matrix.

This function differs from standard NMR Fourier transformation (See [nmrft], page 25) solely because it uses a non-uniform discrete Fourier transform (NDFT) instead of the classical fast Fourier transform (FFT) to compute the spectrum, thus allowing the input samples to be arbitrarily spaced in time. However, this method of computing the spectrum suffers from serious drawbacks, and should not be used in any seriousness.

If a parameter structure is passed as a second argument, a second output value will be produced which contains the chemical shift abscissa vector that is associated with *s*. Optionally, in this case, a third output value will be produced which contains the centered abscissa vector in hertz units, without the carrier offset applied (*hz*).

A.15 O

colors = obscolors (*XY*) [Function File]

See [obscolors], page 48.

Y = oplsclassify (*mdl*, *X*) [Function File]

[*Y*, *T*] = oplsclassify (*mdl*, *X*) [Function File]

Predicts responses Y from one or more observations X based on the OPLS model provided in *mdl*. The observations in X are filtered based on orthogonal variation, transformed by the regression coefficients (B) and classified based on sum of squares to the model classes.

NOTE: this function is not meant to be used directly. If you want to use a OPLS model to classify new observations, use [classify], page 45.

[*w*, *t*, *p*, *u*, *c*, *Wo*, *To*, *Po*, *iter*] = oplscomp (*X*, *Y*, *V*, *aout*) [Function File]

Extracts a single OPLS component from a data and response matrix, as well as any accompanying significant orthogonal components. The returned w, t, p, u and c are vectors corresponding to the rank-one approximation of X and Y of the OPLS component. The OPLS components may be returned in batches of more than one at a time. The final input argument *aout* may be set to nonzero to specify a desired orthogonal component count.

mdl = opls (*X*, *Y*) [Function File]

mdl = opls (*X*, *Y*, *scalefn*) [Function File]

mdl = opls (*X*, *Y*, *scalefn*, *ncv*) [Function File]

mdl = opls (*X*, *Y*, *scalefn*, *ncv*, *aout*) [Function File]

mdl = opls (*X*, *Y*, *scalefn*, *ncv*, *aout*, *w*) [Function File]

See [opls], page 39.

V = orthspace (*X*, *Y*) [Function File]

Returns a matrix containing the Y-orthonormal subspace of the data matrix X. This is used during OPLS modeling, typically.

D = overlaps (*mdl*, *k*) [Function File]

D = overlaps (*mdl*, *k*, *Y*) [Function File]

See [overlaps], page 51.

A.16 P

[*A*, *B*, *C*, *Z*, *iter*, *diffA*] = parafac(*X*, *F*) [Function File]

This function performs a parafac model of 3-way data on input matrix X with predefined number of components F, using minimization across three modes with Khatri-rao products. The convergence is achieved when the change in frobenius is less than 10−9 between iterations. The idea is adapted from Multiway Analysis in the Chemical Sciences introduced by Smilde, Bro and Geladi. For more information see:

> Smilde, Age K., Rasmus Bro, and Paul Geladi. Multi-way analysis with applications in the chemical sciences. Chichester, West Sussex, England Hoboken, NJ: J. Wiley, 2004.

`Y = pcaclassify (mdl, X)` [Function File]
`[Y, T] = pcaclassify (mdl, X)` [Function File]

Predicts responses Y from one or more observations X based on the PCA model provided in *mdl*. The observations in X are transformed into the principal component space and classified based on Mahalanobis distances to the model classes.

NOTE: this function is not meant to be used directly. If you want to use a PCA model to classify new observations, use [classify], page 45.

`[t, p, iter] = pcacomp (X)` [Function File]
`[t, p, iter] = pcacomp (X, t0)` [Function File]

Extracts a single PCA component from a data matrix. The optional second argument *t0* can be passed to specify an initial value for t prior to iteration.

`mdl = pca (X)` [Function File]
`mdl = pca (X, scalefn)` [Function File]
`mdl = pca (X, scalefn, ncv)` [Function File]
`mdl = pca (X, scalefn, ncv, aout)` [Function File]
`mdl = pca (X, scalefn, ncv, aout, w)` [Function File]

See [pca], page 37.

`P = peakpick (s, ab)` [Function File]

Picks peaks in a 1D spectrum vector or 2D spectrum matrix. The input spectrum s and abscissa *ab* are both required. The one-dimensional peak-picking algorithm is an implementation of:

> Du et. al., 'Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching', Bioinformatics, 2006.

The two-dimensional peak-picking algorithm is not yet implemented.

`Xnew = perclass (fn, X, Y)` [Function File]

See [perclass], page 36.

`permdensity (S)` [Function File]

See [permdensity], page 48.

`permscatter (S)` [Function File]

See [permscatter], page 48.

`S = permtest (mdl)` [Function File]
`S = permtest (mdl, n)` [Function File]

See [permtest], page 49.

`sp = phase1d (s, phc0, phc1)` [Function File]

Corrects the phase of a one-dimensional Fourier-transformed spectrum or spectral dataset with a zero order correction *phc0* and a first order correction *phc1*.

Instead of using this function directly, it is recommended that you use [phase], page 26.

sp = phase2d (*s*, *phc0*, *phc1*) [Function File]
Corrects the phase of a two-dimensional Fourier-transformed spectral matrix or cell array with a zero order correction *phc0* and a first order correction *phc1*.

Instead of using this function directly, it is recommended that you use [phase], page 26.

sp = phase (*s*, *parms*, *phc0*, *phc1*) [Function File]
See [phase], page 26.

peaks = pick_peaks(*slice_dir*, *how='gaussian'*, *sn_cutoff=10*, *peaks_per_slice=5*, *interp_res=0.25*) [Function File]
Method for picking peaks from a directory of extracted ion chromatogram (EICs) .csv files. *slice_dir* corresponds to a directory of EICs generated by *create_eics()*. {how specifies the type of peak picking, with "gaussian" being the only current option. *sn_cutoff* refers to the signal-to-noise (SN) cutoff which is taken as the ratio of peak intensity to average signal in the EIC, with the default value being 10. {peaks_per_slice indicates the max number of peaks per slice and {interp_res indicates the uniform interpolation resolution of the data points in seconds. The outputted {peaks matrix stores the information for each peak as a row in the following format for each row:

[ret_time, mz, cts, integral, peak_width]

Y = plsclassify (*mdl*, *X*) [Function File]
[*Y*, *T*] = plsclassify (*mdl*, *X*) [Function File]
Predicts responses Y from one or more observations X based on the PLS model provided in *mdl*. The observations in X are transformed by the regression coefficients (B) and classified based on sum of squares to the model classes.

NOTE: this function is not meant to be used directly. If you want to use a PLS model to classify new observations, use [classify], page 45.

[*w*, *t*, *p*, *u*, *c*, *iter*] = plscomp (*X*, *Y*) [Function File]
Extracts a single PLS component from a data and response matrix.

mdl = pls (*X*, *Y*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*, *ncv*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*, *ncv*, *aout*) [Function File]
mdl = pls (*X*, *Y*, *scalefn*, *ncv*, *aout*, *w*) [Function File]
See [pls], page 38.

[*x*, *y*] = plsroc (*mdl*) [Function File]
[*x*, *y*, *auc*] = plsroc (*mdl*) [Function File]
Returns the results of cross-validating a two-class PLS-DA model in the format of a receiver operating characteristic (ROC) curve.

[*Kresult*] = poly_kernel(*X*, *Y*, *gamma*, *d*) [Function File]
calculate d-polynomial kernel between any two matrices

olynomial Kernel

Xn = pqnorm (*X*) [Function File]
[*Xn*, *s*] = pqnorm (*X*) [Function File]
See [pqnorm], page 32.

[new_rts, new_mzs, new_cts] = prepare_eic (*rts*, *mzs*, *cts*, *interp_res*) [Function File]
Prepares the data found in an extracted ion chromatogram (EIC) for peak picking. Method is typically used internally and not called by users directly. Input arrays must be of the same size. Assumes that retention times are in seconds.

prevpow2 (*x*) [Function File]
If *x* is a scalar, return the last integer N such that `2^n <= abs(x)`.
If *x* is a vector, return `prevpow2(length(x))`.

y = pscorr_build (*x*, *p*) [Function File]
Rebuilding function used by [pscorr], page 33, to calculate optimal phasing and scatter correction of a set of NMR spectra. You will never have to call this function on its own.

y = pscorr_func (*x*, *p*) [Function File]
Objective function used by [pscorr], page 33, to calculate optimal phasing and scatter correction of a set of NMR spectra. This is probably useless on its own.

Xn = pscorr (*X*) [Function File]
[*Xn*, *b*] = pscorr (*X*) [Function File]
Xn = pscorr (*X*, *r*) [Function File]
[*Xn*, *b*] = pscorr (*X*, *r*) [Function File]
See [pscorr], page 33.

[*p_value*] = ptest_svm(*mdl*, *nperm*) [Function File]
calculate d-polynomial kernel between any two matrices

A.17 Q

normed = quantile_normalize(*raw_matrix*) [Function File]
Performs normalization on a GC/LC-MS data matrix of peak features. Imputation should be performed on the data matrix prior to this step. The matrix has the format of each row being a feature and each column corresponding to a specific replicate. This normalization method is based off of that used in BioConductoR. To read more see the following citation:

> Ben Bolstad <Bmb.Com>. PreprocessCore. Bioconductor 2017. https://doi.org/10.18129/B9.BIOC.PREPROCESSCORE.

A.18 R

ca, *acc* = RandomForest_classificationAccuracy(*rf*, *newData*, *newLabels*) [Function File]
Traverses through the given tree based on the values in newData. Once a leaf is reached the result the tree came to is compared to the given newLabels. The result of the tree is returned along with a value signifying if the result was correct.

ClassificationMatrix = RandomForest_classify(*rf*, *newdata*, *newlabels*) [Function File]

Function used to return of a matrix of RandomForest predictions on a new matrix. Takes in a created RandomForest struct along with a new data set and corresponding labels. Returns a cell array with each row representing one of the new samples and each column representing a tree from the RandomForest struct. Each value in the returned cell array represents what that specific tree guessed for the type of the specific sample. (i.e. 1,1 would be tree 1's prediction for the type of sample 1) The final row represents the overall classification accuracy of a given tree on the new data set.

rf = RandomForest(*datasmatrix*, *labelsmatrix*, *numbertreescreated*, *numbertreessaved*) [Function File]

See [RandomForest], page 45.

rf, *newInfo* = RandomForest_readBinaryTree(*treeInfo*) [Function File]

Takes in a cell array of values associated with a singular tree in a RandomForest set. Recursively works through each entry in the cell array to construct a tree struct.

R = ransy1d (*X*, *k*) [Function File]

Use the Ratio Analysis SpectroscopY method (RANSY) to extract single compound spectra from spectra of complex mixtures, defined here:

> S. Wei, et. al. 'Ratio Analysis Nuclear Magnetic Resonance Spectroscopy for Selective Metabolite Identification in Complex Samples'. Analytical Chemistry 2011(83): 7616-7623.

[*Kresult*] = rbf_kernel(*X*, *Y*, *gamma*, *d*) [Function File]

Calculates the radial basis function kernel between any two matrices.

x = realnmr (*s*, *parms*) [Function File]

Discards imaginaries from all dimensions of an NMR dataset in s and returns only the real spectrum in x. A parameter structure (or array) must be passed as a second argument.

s = reconstruct (*T*, *ab*) [Function File]

[*s*, *f*] = reconstruct (*T*, *ab*, *t*) [Function File]

Reconstructs a spectrum s and (optionally) a FID f matrix from a CRAFT decomposition T (See [decompose], page 64). Reconstruction of the spectrum from T permits the automatic removal of phase errors from all signals, resulting in a spectrum that requires no phase correction. However, reconstructed time-domain data will contain the phasing.

The signal table T need not be generated by [decompose], page 64. Any real, three-column matrix T will do. Every row of T is expected to be a complex exponentially decaying sinusoid. The columns of T are expected to be the amplitudes, frequencies, widths and phases of the signals. More specifically:

`T(:,1)`: Amplitude, in absolute units.
`T(:,2)`: Frequency, in Hertz.
`T(:,3)`: Linewidth, in Hertz.
`T(:,4)`: Phase, in $[-1, 1]$.

Alternatively, T may be an M-by-1 cell array, with each value equal to a signal table (matrix). In this case, a data matrix having M rows will be generated, with each observation a reconstruction based on the m-th signal table in the cell array.

`ppmadj = refadj (ppm, oldcs, newcs)` [Function File]
See [refadj], page 27.

`Xn = refnorm (X, ab, refcs)` [Function File]
`[Xn, s] = refnorm (X, ab, refcs)` [Function File]
Normalize the observations of a data matrix to such that the maximum intensity of a spectral region centered around *refcs* is one. The calculated normalization factors may be optionally returned in *s*. The values specified in *roi* must correspond to those in the abscissa *ab*.

`[Y, Yfit, Yhat] = responses (mdl)` [Function File]
Returns the univariate backscale response values from a PLS or OPLS model. The outputs *Y*, *Yfit* and *Yhat* contain the input response values, output fitted responses and output re-estimated responses.

`responsesplot (mdl)` [Function File]
`responsesplot (mdl, coloring)` [Function File]
Builds a responses plot from PLS or OPLS modeled data. An optional second argument *coloring* may be specified to color the points. See [nocolors], page 48, [obscolors], page 48.

`[yhat, prob]= rfclassify(mdl, Z, pseudo_label,` [Function File]
`numbertreessaved)`
See [rfclassify], page 45.

`[result] = rfclass(n1, n2)` [Function File]
calculate dot product between any two matrices

`P = rjmcmc1d (y, t)` [Function File]
`P = rjmcmc1d (y, t, opts)` [Function File]
Execute a Reversible Jump Markov Chain Monte Carlo (RJ-MCMC) simulation in order to obtain estimates of the number of signals in a time-domain free induction decay *y*, and the corresponding parameters of those signals. A time axis *t* is required to be paired with *y*.

A third (optional) argument may be passed to specify options for the fitting algorithm. The argument, *opts*, must be a structure and is expected to contain any of the following option fields:

The option field *iters* may be passed as a two-element vector, whose first element is the number of desired burn-in iterations, and whose second element is the number of desired simulation iterations. *iters* may also contain a third element that indicates the modulus for which iteration to save during the simulation (*i.e.* only save every hundredth iteration).

The option field *lambda* may be passed to indicate the expected number of signals the model will contain. The default value is 20. The option field *delta* may be passed

to indicate the expected signal to noise ratio of the data. The default value is 10. Further option fields *omega0* and *rho0* may be provided to initialize the simulation at more sane parameter values (frequency and decay rate, respectively).

The RJ-MCMC algorithm is implemented according to information reported in:

> Andrieu C., 'Joint Bayesian Model Selection and Estimation of Noisy Sinusoids via Reversible Jump MCMC', IEEE Transactions on Signal Processing, 1999.
>
> Rubtsov D., Griffin J., 'Time-domain Bayesian detection and estimation of noisy damped sinusoidal signals applied to NMR spectroscopy', Journal of Magnetic Resonance, 2007.
>
> Roodaki A., 'Note on the computation of the Metropolis-Hastings ratio for Birth-or-Death moves in trans-dimensional MCMC algorithms for signal decomposition problems', arXiv:1111.6245v2, 2012.

`D = rjmcmc1d_reconstruct (t, omega, rho)` [Function File]
`yhat = rjmcmc1d_reconstruct (t, omega, rho, a)` [Function File]
`yhat = rjmcmc1d_reconstruct (t, P)` [Function File]

Reconstructs a one-dimensional time-domain free induction decay from parameters estimated by [rjmcmc1d], page 80. If only frequencies (*omega*) and decay rates (*rho*) are provided, a basis of signals (D) is returned. If amplitudes are also provided in *a*, a final signal estimate will be returned.

`P = rjmcmc2d (y, t)` [Function File]
`P = rjmcmc2d (y, t, opts)` [Function File]

Execute a Reversible Jump Markov Chain Monte Carlo (RJ-MCMC) simulation in order to obtain estimates of the number of signals in a 2D time-domain free induction decay y, and the corresponding parameters of those signals. A two-element time cell array t is required to be paired with y, the input time-domain matrix.

A third (optional) argument may be passed to specify options for the fitting algorithm. The argument, *opts*, must be a structure and is expected to contain any of the following option fields:

The option field *iters* may be passed as a two-element vector, whose first element is the number of desired burn-in iterations, and whose second element is the number of desired simulation iterations. *iters* may also contain a third element that indicates the modulus for which iteration to save during the simulation (*i.e.* only save every hundredth iteration).

The option field *lambda* may be passed to indicate the expected number of signals the model will contain. The default value is 5. The option field *delta* may be passed to indicate the expected signal to noise ratio of the data. The default value is 10. Further option fields *omega0* and *rho0* may be provided to initialize the simulation at more sane parameter values (frequency and decay rate, respectively).

The RJ-MCMC algorithm is an extension of [rjmcmc1d], page 80, to two-dimensional time-domain data matrices. See [rjmcmc1d], page 80, for more information and literature references.

roi = roipeak (*s*, *ab*, *parms*, *wmin*) [Function File]
See [roipeak], page 27.

roiplot1d (*roi*) [Function File]
roiplot1d (*X*, *ab*, *roi*) [Function File]
Overlays regions of interest (as lines) on an existing line plot of frequency-domain spectral data, or builds a line plot of data with overlaid regions of interest as rectangles.

roiplot2d (*roi*) [Function File]
roiplot2d (*X*, *ab*, *roi*) [Function File]
Overlays regions of interest (as rectangles) on an existing contour plot of frequency-domain spectral data, or builds a contour plot of data with overlaid regions of interest as rectangles.

roiplot (*X*, *ab*, *parms*, *roi*) [Function File]
See [roiplot], page 28.

v = rqdiff (*mdl*) [Function File]
See [rqdiff], page 48.

rqinfo (*mdl*) [Function File]
Prints R^2/Q^2 values from a PCA, PLS, OPLS or LDA model.

v = rq (*mdl*) [Function File]
See [rq], page 48.

rqplot (*mdl*) [Function File]
h = rqplot (*mdl*) [Function File]
See [rqplot], page 48.

A.19 S

save_peaks(*peaks*, *filename*) [Function File]
Utility method and recommended means to save picked peaks. First argument is a matrix of peaks where each row is a peak and has the following data layout for each row: [ret_time, mz, cts, integral, peak_width]

Second argument is the .csv filename to which the data will be saved.

savescores (*mdl*, *filename*) [Function File]
savescores (*mdl*, *filename*, *A*) [Function File]
savescores (*mdl*, *filename*, *A*, *Y*) [Function File]
savescores (*mdl*, *filename*, *A*, *Y*, *labels*) [Function File]
Exports scores from a PCA, PLS or OPLS model to SIMCA-P+ format text.

T = scores (*mdl*) [Function File]
T = scores (*mdl*, *n*) [Function File]
See [scores], page 50.

`r = ssratio (a, b)` [Function File]

Calculates the ratio of the row sum of squares of a and b.

`stackplot (X)` [Function File]
`stackplot (X, ab)` [Function File]

See [stackplot], page 46.

`[A, B] = states (X)` [Function File]
`A = states (X)` [Function File]
`X = states (A, B)` [Function File]

De-interlaces States/Haberkorn/Ruben cosine- and sine-modulated rows of a complex matrix X into the complex matrices A and B.

The de-interlacing procedure is as follows:

$$X_{cos} \leftarrow X_{2:2:N,\cdot}$$

$$X_{sin} \leftarrow X_{1:2:N-1,\cdot}$$

$$A \leftarrow Re\{X_{cos}\} + i \cdot Re\{X_{sin}\}$$

$$B \leftarrow Im\{X_{cos}\} + i \cdot Im\{X_{sin}\}$$

Alternatively, this function can re-interlace the matrices A and B to yield a new original matrix X.

The re-interlacing procedure is as follows:

$$X_{cos} \leftarrow Re\{A\} + i \cdot Re\{B\}$$

$$X_{sin} \leftarrow Im\{A\} + i \cdot Im\{B\}$$

$$X_{2:2:N,\cdot} \leftarrow X_{cos}$$

$$X_{1:2:N-1,\cdot} \leftarrow X_{sin}$$

`C = stocsy (X, ab)` [Function File]

Use the Statistical Total Correlation SpectroscopY (STOCSY) method to generate a contour map of correlations between spectral variables in a 1D NMR dataset, defined here:

> O. Cloarec, et. al. 'Statistical Total Correlation Spectroscopy: An Exploratory Approach for Latent Biomarker Identification from Metabolic 1H NMR Data Sets'. Analytical Chemistry 2005(77): 1282-1289.

`[fsub, tsub, dfbw, D] = subfid1d (f, t, parms, Fmin, Fmax)` [Function File]

Extracts a band of frequencies ($[F_{min}, F_{max}]$) from f into *fsub*.

It is highly recommended that you use [subfid], page 87, instead of calling this function directly.

`[fsub, tsub, dfbw, D] = subfid2d (f, t, parms, Fmin, Fmax)` [Function File]

Extracts a band of frequencies ($[F_{min}, F_{max}]$) from f into *fsub*.

It is highly recommended that you use [subfid], page 87, instead of calling this function directly.

`[fsub, tsub, dfbw, D] = subfid (f, t, parms, Fmin, Fmax)` [Function File]
Extracts a band of frequencies ($[F_{min}, F_{max}]$) from *f* into *fsub*. This function can operate on multiple free induction decays at once if *f* is supplied as a data matrix. The function also accepts two-dimensional time-domain data in the form of matrices or cell arrays.

For one-dimensional data, *Fmin* and *Fmax* must be scalar values. For two-dimensional data, they must be two-vectors containing frequencies for each dimension.

Procedurally, the extraction is as follows: modulate *f* such that the desired frequency band lies at zero frequency, FIR filter *f* to avoid aliasing, and decimate the filtered form of *f*. The extracted band will be returned in (*fsub*,*tsub*), and the decimation ratio will be returned in *D*.

`[ssub, absub] = subspect1d (s, ab, parms, Fmin, Fmax)` [Function File]
Extracts a band of frequencies ($[F_{min}, F_{max}]$) from *s* into *ssub*.

It is highly recommended that you use [subspect], page 87, instead of calling this function directly.

`[ssub, absub] = subspect2d (s, ab, parms, Fmin, Fmax)` [Function File]
Extracts a band of frequencies ($[F_{min}, F_{max}]$) from *s* into *ssub*.

It is highly recommended that you use [subspect], page 87, instead of calling this function directly.

`[ssub, absub] = subspect (s, ab, parms, Fmin, Fmax)` [Function File]
Extracts a band of frequencies ($[F_{min}, F_{max}]$) from *s* into *ssub*. This function can operate on multiple spectra at once if *s* is supplied as a data matrix. The function also accepts two-dimensional spectral data in the form of matrices or cell arrays.

The values in *Fmin* and *Fmax* should correspond to those in *ab*. For one-dimensional data, *Fmin* and *Fmax* must be scalar values. For two-dimensional data, they must be two-vectors containing frequencies for each dimension.

Unlike [subfid], page 87, which performs its extraction in the time domain, this function has it easy: it just crops out the subspectrum of interest from the input spectral data.

`susplot (mdl1, mdl2)` [Function File]
`susplot (mdl1, mdl2, a)` [Function File]
`susplot (mdl1, mdl2, a, numbers)` [Function File]
`pdata = susplot (...)` [Function File]
See [susplot], page 47.

`Xs = suv (X)` [Function File]
`[Xs, mu] = suv (X)` [Function File]
`[Xs, mu, s] = suv (X)` [Function File]
`... = suv (X, w)` [Function File]
See [suv], page 34.

A.20 T

OUTPUT:

bestC: best cost value chosen from cross-validation

CV: return cell with prediction error rate for each iteration

A.21 U

R = ussr (*F*, *P*, *ppm*) [Function File]

R = ussr (*F*, *P*, *ppm*, *alpha*) [Function File]

Perform Uncomplicated Statistical Spectral Remodeling (USSR) on two matrices F and P, where spectra are paired in the two matrices, arranged row-wise. The P matrix is remodeled to yield R according to the procedure in:

B. Worley et. al., 'Uncomplicated Statistical Spectral Remodeling', J. Biomol. NMR., in preparation.

The chemical shift values on the abscissas of all spectra are assumed to be identical to within the digital resolution of the experiment, so only a single abscissa vector *ppm*.

An optional fourth input variable *alpha* may be set to define the level of confidence required to keep a signal in the reconstructed spectrum.

A.22 V

colors = varcolors (*X*) [Function File]

See [varcolors], page 48.

[*outer_prod*] = vector_prod(*A*, *B*) [Function File]

Finds the vector product matrix of two input vectors. Assumes that A and B are of the same size and throws an error if this is not the case. More information can be found here:

> Smilde, Age K., Rasmus Bro, and Paul Geladi. Multi-way analysis with applications in the chemical sciences. Chichester, West Sussex, England Hoboken, NJ: J. Wiley, 2004.

A.23 W

y = wavelet_haar (*t*) [Function File]

Calculates the Haar mother wavelet.

y = wavelet_sombrero (*t*) [Function File]

Calculates the Mexican hat, or 'sombrero', mother wavelet.

W = weights (*mdl*) [Function File]

W = weights (*mdl*, *n*) [Function File]

See [weights], page 50.

weightsplot (*mdl*) [Function File]

weightsplot (*mdl*, *coloring*) [Function File]

h = weightsplot (*mdl*) [Function File]

h = weightsplot (*mdl*, *coloring*) [Function File]

See [weightsplot], page 47.

weight = w_weight(*s*, *sig*) [Function File]

Assigning weights as a function of residual sum of squares (proportional to Gaussian kernel)

A.24 Z

zfid = zerofill1d (*fid*, *k*) [Function File]

Appends zeros at the end of a one-dimensional free induction decay (FID) vector or matrix by doubling the total length k times.

Instead of using this function directly, it is recommended that you use [zerofill], page 25.

zfid = zerofill2d (*fid*, *k*) [Function File]

Appends zeros at the end of a two-dimensional free induction decay (FID) matrix or cell array by doubling the total length k times along each dimension.

Instead of using this function directly, it is recommended that you use [zerofill], page 25.

zfid = zerofill (*fid*, *parms*) [Function File]

zfid = zerofill (*fid*, *parms*, *k*) [Function File]

See [zerofill], page 25.

Appendix B Example usage

B.1 1D NMR Example Usage

The following script is a complete example of how to use MVAPACK to handle NMR chemometrics data.

This procedure applied to a dataset having 32 observations and 8192 real data points results in a run-time of 30 minutes, most of which is due to the color-enabled `backscaleplot` command.

It goes without saying that *your mileage may vary*. This script is a working example, and the exact series of processing commands will differ for every dataset.

```
% load in the raw time-domain data.
F.dirs = glob('my-data-???/1');
[F.data, F.parms, F.t] = loadnmr(F.dirs);

% zero-fill three times.
F.nzf = 3;
F.data = zerofill(F.data, F.parms, F.nzf);

% fourier transform the data.
[S.data, S.ppm] = nmrft(F.data, F.parms);

% autophase the spectra.
[S.data, S.phc0, S.phc1] = autophase(S.data, F.parms);

% extract the real spectral components.
X.data = realnmr(S.data, F.parms);
X.ppm = S.ppm;

% remove two failed observations.
X.rm.obs = [1, 31];
X.data = rmobs(X.data, X.rm.obs);

% identify indices of chemical regions to cut.
i0 = findnearest(X.ppm, min(X.ppm));
i1 = findnearest(X.ppm, 0.554);
i2 = findnearest(X.ppm, 4.488);
i3 = findnearest(X.ppm, max(X.ppm));

% remove the uninformative chemical shift regions.
X.rm.var = [i3 : i2, i1 : i0];
[X.data, X.ppm] = rmvar(X.data, X.ppm, X.rm.var);

% perform adaptive intelligent binning with default settings.
[B.data, B.ppm] = binadapt(X.data, X.ppm, F.parms);
```

```
% perform segmented spectral alignment with default settings.
X.data = icoshift(X.data, X.ppm);

% create pca class information.
cls.pca.Y = classes([7, 8, 7, 7]);
cls.pca.labels = {'A', 'B', 'C', 'D'};

% create opls-da class information.
cls.opls.Y = classes([7 + 8 + 7, 7]);
cls.opls.labels = {'ABC', 'D'};

% build a full-resolution pca model.
mdl.pca = pca(B.data);
mdl.pca = addclasses(mdl.pca, cls.pca.Y);
mdl.pca = addlabels(mdl.pca, cls.pca.labels);

% build a full-resolution opls-da model.
mdl.opls = opls(X.data, cls.opls.Y);
mdl.opls = addlabels(mdl.opls, cls.opls.labels);

% add cv-anova validation information to the opls-da model.
mdl.opls.cv.anova = cvanova(mdl.opls);

% add permutation test information to the opls-da model.
mdl.opls.cv.nperm = 500;
mdl.opls.cv.perm = permtest(mdl.opls, mdl.opls.cv.nperm);

% plot pca model information.
rqplot(mdl.pca);
scoresplot(mdl.pca, 3);
backscaleplot(X.ppm, mdl.pca);

% plot opls model information.
rqplot(mdl.opls);
permscatter(mdl.opls.cv.perm);
scoresplot(mdl.opls, 2);
backscaleplot(X.ppm, mdl.opls, true);

% save the session data.
savedate = date();
save('-binary', '-z', 'my-data.dat.gz');
```

B.2 Metabolite Identification in 1D NMR Mixture

```
% read in matrix of reference spectra and chemical shift vector ppm
% each column in refset matrix is a spectrum of one
% reference metabolite
```

```
refset = csvread("refset1_v3.csv");
ppm = csvread('tmsp_ppm.csv');
% read in peak indexes as a vector
peaks_id = csvread("peaks_index.csv");
% read in mixture spectrum
y = csvread(("spectrum1.csv");
n = length(ppm);
d = 10;
k = 50;
e = 0.0001;
p = columns(refset);
sig = 0.3;

% initial values for beta
beta_init = zeros(p,1);
% obtain maximum lambda
l_max = lambda_max(refset, y, d, n);
% create lambda sequence
l_seq = exp(linspace(log(e*l_max), log(l_max), length.out = k)));
l_seq = flip(l_seq);
% create an empty vector to store mean prediction error through
% cross validation
mpe = [];
for i=1:k
    mpe(i) = cv_coordinate1(beta_init, refset, y, peaks_id, l_seq(i),
    niter = 5, d, sig, nfold=4);
end
[min_e, min_idx] = min(mpe);
l_opt = l_seq(min_idx);
% identify metabolite with non-zero coefficients through beta_hat
[beta_hat, w_matrix] = coordinate_lasso1(beta_init, refset, y,
    peaks_id, l_opt, niter = 5, d, sig, sc = T);
```

B.3 2D NMR Example Usage

```
% add the mvapack path for the bionmr server
addpath('/opt/mvapack');

% load in the fid data.
F.dirs = glob('???');
[F.data, F.parms, F.t] = loadnmr(F.dirs);

% build the spectra
[S.data, S.ppm] = nmrft(F.data, F.parms);

% autophase the spectra.
[S.data, S.phc0, S.phc1] = autophase(S.data, F.parms);
```

```
[S.data, S.phc0, S.phc1] = autophase(S.data, F.parms);
[S.data, S.phc0, S.phc1] = autophase(S.data, F.parms);


% extract the real spectral components.
X.data = realnmr(S.data, F.parms);
X.ppm = S.ppm;

%Check plot to find reference standard
plot(X.ppm, X.data)


% first step, change the reference

%Reference
X.ppm = refadj(X.ppm, -0.04, 0.0);

%icoshift
X.data = icoshift(X.data, X.ppm);

%bin the data matrix

[B1.data, B1.ppm,B1.widths] = binadapt(X.data, X.ppm, F.parms);

%noise removal
i0 = findnearest(B1.ppm, 9.91);
i1 = findnearest(B1.ppm, 9.92);
B1.noise = [i1 : i0];
[B1.data, B1.ppm, B1.idxrm] = rmnoise(B1.data, B1.ppm, B1.noise);
B1.widths(B1.idxrm') = [];

%SNV normalization
B1.data = snv (B1.data);

%find the indices to remove

i0 = findnearest(B1.ppm, min(B1.ppm));
i1 = findnearest(B1.ppm, 0.1);
i2 = findnearest(B1.ppm, 4.7);
i3 = findnearest(B1.ppm, 4.8);
i6 = findnearest(B1.ppm, 10.0);
i7 = findnearest(B1.ppm, max(B1.ppm));

% remove the indices from the data matrix.
B1.rm.var = [i7:i6, i3 : i2, i1 : i0];
[B1.data, B1.ppm] = rmvar(B1.data, B1.ppm, B1.rm.var);
```

```
%change class information

% build class information.x)
cls.x = classes([3,6,6,12]);
cls.labelsx = 'cin','HPV','Cancer','control';


%build a pca model
mdlpca = pca(B1.data,[],[],[],[]);
mdlpca = addclasses(mdlpca, cls.x);
mdlpca = addlabels(mdlpca, cls.labelsx);
scoresplot(mdlpca, 2, [], true);
rqplot(mdlpca)
%to save scores to make dendrograms
savescores (mdlpca, 'scorespcasnv.txt', 3, cls.x, cls.labelsx)
%.........................................
%          Addtional processing                .
%.........................................


% zero fill three times and group delay correct.
F.nzf = 3;
F.data = zerofill(F.data, F.parms, F.nzf);

% To do PSC
S.data = pscorr(S.data);

% remove the first and last observations.
% Note Only if you want to remove OBS

S.rm.obs = [3,9,10,20,26,30,31,32];
S.data = rmobs(S.data, S.rm.obs);

% remove the first and last observations. Note Only if you want to
% remove OBS
B1.rm.obs = [15];
B1.data = rmobs(B1.data, B1.rm.obs);

% remove the first and last observations. Note Only if you want to
% remove OBS
X.rm.obs = [14,6];
X.data = rmobs(X.data, X.rm.obs);

% remove the first and last observations. Note Only if you want to
% remove OBS

S.rm.obs = [12];
```

```
S.data = rmobs(S.data, S.rm.obs);

% remove the first and last observations. Note Only if you want to
% remove OBS
X.rm.obs = [1,2,3,4,5,6,13,14,15,16,17,18];
X.data = rmobs(X.data, X.rm.obs);

%Total sum normalization

Xt.data = X.data ./ (diag(sum(X.data,2))* ones(29,14196));

%find mean of a group of spectra

X.data = mean(X.data)

%saving a spectra in a text file from mvapack

A = [X.ppm, X.data(1,:)'];
save -ascii 'A.txt' A

%printing a plot

print -deps -color 'filename.eps'

% 3d plot print, if the view at the left corner of the plot
% e.g view (74,10)
view (10, 90-74);
print -deps -color 'filename.eps'

%ROI norm
roi = [-0.06, 0.06];
X.data = roinorm(X.data, X.ppm, roi);

%find the indices to remove

i0 = findnearest(X.ppm, min(X.ppm));
i1 = findnearest(X.ppm, 0.05);
i2 = findnearest(X.ppm, 4.65);
i3 = findnearest(X.ppm, 4.75);
i4 = findnearest(X.ppm, 12.0);
i5 = findnearest(X.ppm, max(X.ppm));

% remove the indices from the data matrix.
X.rm.var = [i5 : i4, i3 : i2, i1 : i0];
[X.data, X.ppm] = rmvar(X.data, X.ppm, X.rm.var);

%find the indices to remove. These will vary for each dataset.
```

```
i0 = findnearest(S.ppm, min(S.ppm));
i1 = findnearest(S.ppm, 0.3);
i2 = findnearest(S.ppm, 3.3);
i3 = findnearest(S.ppm, 3.4);
i4 = findnearest(S.ppm, 4.7);
i5 = findnearest(S.ppm, 4.9);
i6 = findnearest(S.ppm, 10.0);
i7 = findnearest(S.ppm, max(S.ppm));

% remove the indices from the data matrix.
S.rm.var = [i7:i6,i5:i4,i3:i2,i1 : i0];
[S.data, S.ppm] = rmvar(S.data, S.ppm, S.rm.var);

% find the indices to remove

i2 = findnearest(X.ppm, 4.55);
i3 = findnearest(X.ppm, 5.0);

% remove the indices from the data matrix.
X.rm.var = [i3:i2];
[X.data, X.ppm] = rmvar(X.data, X.ppm, X.rm.var);


%generating lda

mdl.lda =lda((mdlpca,3), cls.Y)
```

B.4 GC/LC-MS Peak Picking

One of the most important steps of GC/LC-MS is peak picking. In the MVAPACK framework this is a specific step/end point. Use the below script as a starting point but keep in mind that there are many default parameters being used so it may be necessary to tune these parameters for your own datasets.

```
addpath('/opt/mvapack');

% Get the replicate paths. Here they are .mzML but other
% usable extensions are .txt and .mzXML

paths = glob('/path/to/data/*.mzML');

for ii = 1:length( paths )
    % loop through and get each path individually
    path = paths{ ii };
    % create a folder of Extracted Ion Chromatograms for
```

```
    % each replicate. The combination of sprint() and
    % regexprep() allows us to make a new folder of EICs
    % with the same name as the original replicate.
    dirname = sprintf('/path/to/eics/%s',
                regexprep( path, '^.*\/|\..*$', '')
              );
    create_eics( path, dirname );
    % now that we have the eics we can perform peak picking.
    % again, this is using default parameters but others can
    % be selected for pick_peaks()
    peaks = pick_peaks( dirname );
    % now we can save the peaks
    peakfile = sprintf('/path/to/%s-peaks.csv',
                    regexprep( path, '^.*\/|\..*$', '')
                );
    save_peaks( peaks, peakfile );
end
```