
hctsa
Manual

Table of Contents

Performing highly comparative time-series analysis in Matlab using *hctsa*

This manual outlines the steps required to set up and implement highly comparative time-series analysis using the hctsa package, as described in our papers:

- B. D. Fulcher & N. S. Jones. Automatic time-series phenotyping using massive feature extraction. *bioRxiv* 081463 (2016).
- S. S. Sethi, V. Zerbi, N. Wenderoth, A. Fornito, B. D. Fulcher. Structural connectome topology relates to regional BOLD signal dynamics in the mouse brain. *Chaos* **27**, 047405 (2017) (here for a preprint).
- B. D. Fulcher & N. S. Jones. Highly comparative feature-based time-series classification. *IEEE Trans. Knowl. Data Eng.* **26**, 3026 (2014).
- B. D. Fulcher, M. A. Little, N. S. Jones. Highly comparative time-series analysis: the empirical structure of time series and their methods. *J. Roy. Soc. Interface* **10**, 20130048 (2013).
- B. D. Fulcher, A. E. Georgieva, C. W. G. Redman, N. S. Jones. Highly comparative fetal heart rate analysis. *34th Ann. Int. Conf. IEEE EMBC* 3135 (2012).

List of code files used in *hctsa*

Here we provide a full list of Matlab code files, organized loosely into broad categories, and with brief descriptions.

Distribution

Code summarizing properties of the distribution of values in a time series (disregarding their sequence through time).

Code file	Description
DN_Burstiness	Burstiness statistic of a time series
DN_CompareKSFit	Fits a distribution to data.
DN_CustomSkewness	Custom skewness measures
DN_FitKernelSmooth	Statistics of a kernel-smoothed distribution of the data.
DN_Fit_mle	Maximum likelihood distribution fit to data.
DN_HighLowMu	The highlowmu statistic.
DN_HistogramMode	Mode of a data vector.
DN_Mean	A given measure of location of a data vector.
DN_MinMax	The maximum and minimum values of the input data vector
DN_Moments	A moment of the distribution of the input time series.
DN_OutlierInclude	How statistics depend on distributional outliers.
DN_OutlierTest	How distributional statistics depend on distributional outliers.
DN_ProportionValues	Proportion of values in a data vector.
DN_Quantile	Quantile of the data vector
DN_RemovePoints	How time-series properties change as points are removed.
DN_SimpleFit	Fit distributions or simple time-series models to the data.
DN_Spread	Measure of spread of the input time series.
DN_TrimmedMean	Mean of the trimmed time series using trimmean.
DN_Unique	The proportion of the time series that are unique values
DN_Withinp	Proportion of data points within p standard deviations of the mean.
DN_cv	Coefficient of variation
DN_pleft	Distance from the mean at which a given proportion of data are more distant.
EN_DistributionEntropy	Distributional entropy.
HT_DistributionTest	Hypothesis test for distributional fits to a data vector.

Correlation

Code summarizing basic properties of how values of a time series are correlated through time.

Code file	Description

CO_AddNoise	Changes in the automutual information with the addition of noise
CO_AutoCorr	Compute the autocorrelation of an input time series
CO_AutoCorrShape	How the autocorrelation function changes with the time lag.
CO_Embed2	Statistics of the time series in a 2-dimensional embedding space
CO_Embed2_AngleTau	Angle autocorrelation in a 2-dimensional embedding space
CO_Embed2_Basic	Point density statistics in a 2-d embedding space
CO_Embed2_Dist	Analyzes distances in a 2-d embedding space of a time series.
CO_Embed2_Shapes	Shape-based statistics in a 2-d embedding space
CO_FirstMin	Time of first minimum in a given correlation function
CO_FirstZero	The first zero-crossing of a given autocorrelation function
CO_NonlinearAutocorr	A custom nonlinear autocorrelation of a time series.
CO_StickAngles	Analysis of line-of-sight angles between time-series data points.
CO_TranslateShape	Statistics on datapoints inside geometric shapes across the time series
CO_f1ecac	The 1/e correlation length
CO_fzcglscf	The first zero-croessing of the generalized self-correlation function
CO_glscf	The generalized linear self-correlation function of a time series.
CO_tc3	Normalized nonlinear autocorrelation function, tc3.
CO_trev	Normalized nonlinear autocorrelation, trev function of a time series
DK_crinkle	Computes James Theiler's crinkle statistic
DK_theilerQ	Computes Theiler's Q statistic
DK_timerev	Time reversal asymmetry statistic
NL_embed_PCA	Principal Components analysis of a time series in an embedding space.
Automutual information:	
CO_RM_AMInformation	Automutual information (Rudy Moddemeijer implementation)
CO_CompareMinAMI	Variability in first minimum of automutual information
CO_HistogramAMI	The automutual information of the distribution using

CO_HistogramAMI	histograms.
IN_AutoMutualInfoStats	Statistics on automutual information function for a time series.

Entropy and information theory

Entropy and complexity measures for time series

Code file	Description
EN_ApEn	Approximate Entropy of a time series
EN_CID	Simple complexity measure of a time series.
EN_MS_LZcomplexity	Lempel-Ziv complexity of a n-bit encoding of a time series
EN_MS_shannon	Approximate Shannon entropy of a time series.
EN_PermEn	Permutation Entropy of a time series.
EN_RM_entropy	Entropy of a time series using Rudy Moddemeijer's code.
EN_Randomize	How time-series properties change with increasing randomization.
EN_SampEn	Sample Entropy of a time series
EN_mse	multiscale entropy for a time series
EN_rpde	Recurrence period density entropy (RPDE).
EN_wentropy	Entropy of time series using wavelets.

Time-series model fitting and forecasting

Fitting time-series models, and doing simple forecasting on time series.

Code file	Description
MF_ARMA_orders	Compares a range of ARMA models fitted to a time series.
MF_AR_arcov	Fits an AR model of a given order, p.
MF_CompareAR	Compares model fits of various orders to a time series.
MF_CompareTestSets	Robustness of test-set goodness of fit
MF_ExpSmoothing	Exponential smoothing time-series prediction model.
MF_FitSubsegments	Robustness of fitted model parameters across different
MF_GARCHcompare	Comparison of GARCH time-series models
MF_GARCHfit	GARCH time-series modeling.
MF_GP_FitAcross	Gaussian Process time-series modeling for local prediction.
MF_GP_LocalPrediction	Gaussian Process time-series model for local prediction.
MF_GP_hyperparameters	Gaussian Process time-series model parameters and goodness of fit
MF_StateSpaceCompOrder	Change in goodness of fit across different state space models.
MF_StateSpace_n4sid	State space time-series model fitting.
MF_arfit	Statistics of a fitted AR model to a time series.
MF_armax	Statistics on a fitted ARMA model.
MF_hmm_CompareNStates	Hidden Markov Model (HMM) fitting to a time series.
MF_hmm_fit	Fits a Hidden Markov Model to sequential data.
MF_steps_ahead	Goodness of model predictions across prediction lengths
FC_LocalSimple	Simple local time-series forecasting.
FC_LoopLocalSimple	How simple local forecasting depends on window length.
FC_Surprise	How surprised you would be of the next data point given recent memory.
PP_ModelFit	See if AR model fit improves with different preprocessings

Stationarity and step detection

Quantifying how properties of a time series change over time.

Code file	Description
SY_DriftingMean	Mean and variance in local time-series subsegments.
SY_DynWin	How stationarity estimates depend on the number of time-series subsegments
SY_KPSStest	The KPSS stationarity test.
SY_LocalDistributions	Compares the distribution in consecutive time-series segments
SY_LocalGlobal	Compares local statistics to global statistics of a time series.
SY_PPtest	Phillips-Peron unit root test.
SY_RangeEvolve	How the time-series range changes across time.
SY_SlidingWindow	Sliding window measures of stationarity.
SY_SpreadRandomLocal	Bootstrap-based stationarity measure.
SY_StatAv	Simple mean-stationarity metric, StatAv.
SY_StdNthDer	Standard deviation of the nth derivative of the time series.
SY_StdNthDerChange	How the output of SY_StdNthDer changes with order parameter.
SY_TISEAN_nstat_z	Cross-forecast errors of zeroth-order time-series models
SY_VarRatioTest	Variance ratio test for random walk.
Step detection:	
CP_ML_StepDetect	Analysis of discrete steps in a time series.
CP_l1pwc_sweep_lambda	Dependence of step detection on regularization parameter.
CP_wavelet_varchg	Variance change points in a time series.

Nonlinear time-series analysis and fractal scaling

Nonlinear time-series analysis methods, including embedding dimensions and fluctuation analysis.

Code file	Description
NL_BoxCorrDim	Correlation dimension of a time series.
NL_DVV	Delay Vector Variance method for real and complex signals.
NL_MS_fnn	False nearest neighbors of a time series.
NL_MS_nlpe	Normalized drop-one-out constant interpolation nonlinear prediction error.
NL_TISEAN_c1	Information dimension.
NL_TISEAN_d2	d2 routine from the TISEAN package.
NL_TISEAN_fnn	false nearest neighbors of a time series.
NL_TSTL_FractalDimensions	Fractal dimension spectrum, D(q), of a time series.
NL_TSTL_GPCorrSum	correlation sum scaling by Grassberger-Proccacia algorithm
NL_TSTL_LargestLyap	Largest Lyapunov exponent of a time series.
NL_TSTL_PoincareSection	Poincare sectino analysis of a time series.
NL_TSTL_ReturnTime	Analysis of the histogram of return times.
NL_TSTL_TakensEstimator	Taken's estimator for correlation dimension.
NL_TSTL_acp	acp function in TSTOOL
NL_TSTL_dimensions	box counting, information, and correlation dimension of a time series.
NL_crptool_fnn	Analyzes the false-nearest neighbours statistic.
SD_SurrogateTest	Analyzes test statistics obtained from surrogate time series
SD_TSTL_surrogates	Surrogate time-series analysis
TSTL_delaytime	Optimal delay time using the method of Parlitz and Wichard.
TSTL_localdensity	Local density estimates in the time-delay embedding space
Fluctuation analysis:	
SC_MMA	Physionet implementation of multiscale multifractal analysis
SC_fastdfa	Matlab wrapper for Max Little's ML_fastdfa code
SC_FluctAnal	Implements fluctuation analysis by a variety of methods.

Fourier and wavelet transforms, periodicity measures

Properties of the time-series power spectrum, wavelet spectrum, and other periodicity measures.

Code file	Description
SP_Summaries	Statistics of the power spectrum of a time series
DT_IsSeasonal	A simple test of seasonality.
PD_PeriodicityWang	Periodicity extraction measure of Wang et al.
WL_DetailCoeffs	Detail coefficients of a wavelet decomposition.
WL_coeffs	Wavelet decomposition of the time series.
WL_cwt	Continuous wavelet transform of a time series
WL_dwtcoeff	Discrete wavelet transform coefficients.
WL_fBM	Parameters of fractional Gaussian noise/Brownian motion in a time series
WL_scal2frq	Frequency components in a periodic time series

Symbolic transformations

Properties of a discrete symbolization of a time series.

Code file	Description
SB_BinaryStats	Statistics on a binary symbolization of the time series
SB_BinaryStretch	Characterizes stretches of 0/1 in time-series binarization
SB_MotifThree	Motifs in a coarse-graining of a time series to a 3-letter alphabet
SB_MotifTwo	Local motifs in a binary symbolization of the time series
SB_TransitionMatrix	transition probabilities between different time-series states
SB_TransitionpAlphabet	How transition probabilities change with alphabet size.

Statistics from biomedical signal processing

Simple time-series properties derived mostly from the heart rate variability (HRV) literature.

Code file	Description
MD_hrv_classic	Classic heart rate variability (HRV) statistics.
MD_pNN	pNNx measures of heart rate variability.
MD_polvar	The POLVARd measure of a time series.
MD_rawHRVmeas	Heart rate variability (HRV) measures of a time series.

Basic statistics, trend

Basic statistics of a time series, including measures of trend.

Code file	Description
SY_Trend	Quantifies various measures of trend in a time series.
ST_FitPolynomial	Goodness of a polynomial fit to a time series
ST_Length	Length of an input data vector.
ST_LocalExtrema	How local maximums and minimums vary across the time series.
ST_MomentCorr	Correlations between simple statistics in local windows of a time series.
ST_SimpleStats	Basic statistics about an input time series

Others

Other properties, like extreme values, visibility graphs, physics-based simulations, and dependence on pre-processings applied to a time series.

Code file	Description
EX_MovingThreshold	Moving threshold model for extreme events in a time series
HT_HypothesisTest	Statistical hypothesis test applied to a time series.
NW_VisibilityGraph	Visibility graph analysis of a time series.
PH_ForcePotential	Couples the values of the time series to a dynamical system
PH_Walker	Simulates a hypothetical walker moving through the time domain.
PP_Compare	Compare how time-series properties change after pre-processing.
PP_Iterate	How time-series properties change in response to iterative pre-processing.

Getting started

The *hctsa* package can be used *completely within Matlab*, allowing users to analyse time-series datasets quickly and easily. Here we will focus on this Matlab-based use of the software, but note that, for larger datasets requiring distributed computing set-ups, or datasets that may grow with time, *hctsa* can also be linked to a *mySQL* database, as described in a dedicated chapter.

Installing the *hctsa* package

The simplest way to get the *hctsa* package up and running is to run the `install` script, which adds the required paths to dependent time-series packages (toolboxes), and compiles *mex* binaries to work on your system architecture. Once this one-off installation step is complete, you're ready to go! (NB: to include additional functions from the TISEAN nonlinear time-series analysis package, you'll also need to compile TISEAN routines).

After installation, future use of the package can begin by opening Matlab, navigating to the *hctsa* package, and then loading the paths required by the *hctsa* package by running the `startup` script.

Structure of the *hctsa* package

Overview

The *hctsa* framework consists of three basic objects:

1. *Master Operations* specify pieces of code (Matlab functions) and their inputs to be computed. Taking in a single time series, master operations can generate a large number of outputs as a Matlab structure, each of which can be identified with a single *operation* (or 'feature').
2. *Operations* (or 'features') are a single number summarizing some measure of structure in a time series. In *hctsa*, each operation links to an output from a piece of evaluated code (a *master operation*).
3. *Time series* are univariate, uniformly sampled, time-ordered measurements.

These three different objects are summarized below:

	Master Operation	Operation	Time Series
Summary:	Code and inputs to execute	Single feature	Univariate data
Example:	`CO_AutoCorr(x,1:5,'TimeDomain')`	`AC_1`	[1.2, 33.7, -0.1, ...]

In the example above, a *master operation* specifies the code to run, `CO_AutoCorr(x,1:5,'TimeDomain')`, which outputs the autocorrelation of the input time series (*x*) at lags 1, 2, ..., 5. Each operation (or 'feature') is a single number that draws on this set of outputs, for example, the autocorrelation at lag 1, which is named `AC_1`, for example.

In the *hctsa* framework, master operations, operations, and time series are stored as structure arrays that contain all of their associated keywords and metadata (and actual time-series data in the case of time series).

For a given *hctsa* analysis, the user must specify a set of code to evaluate (*master operations*), their associated individual outputs to measure (*operations*), and a set of time series to evaluate the features on (*time series*).

We provide a default library of approximately 8,000 *operations* (derived from approximately 1,100 unique *master operations*). This can be customized, and additional pieces of code can also be added to the repository.

The results of a *hctsa* analysis

Having specified a set of master operations, operations, and time series, the results of computing these functions in the time series data are stored in three matrices:

- **TS_DataMat** is an *n* x *m* data matrix containing the results of applying *m* operations to the *n* time series.
- **TS_Quality** is an *n* x *m* matrix containing quality labels for each operation output (coding different outputs such as errors or NaNs). Quality labels are described in the section below.
- **TS_CalcTime** is an *n* x *m* matrix containing calculation times for each operation output. Note that the calculation time stored is for the corresponding master operation.

HCTSA .mat files

Each `HCTSA*.mat` file includes the structure arrays described above: for **TimeSeries** (corresponding to the rows of the **TS_** matrices), **Operations** (corresponding to columns of the **TS_** matrices), and **MasterOperations**, corresponding to the code evaluated to compute the operations. In addition, the results are stored as above: **TS_DataMat**, **TS_Quality**, and **TS_CalcTime**.

Quality labels

Quality labels are used to indicate when operations take non-real values, or when fatal errors were encountered. Quality labels are stored in the **Quality** column of the **Results** table in the *mySQL* database, and in local Matlab files as the **TS_Quality** matrix.

When the quality label is nonzero, this indicates that a *special-valued output* occurred. In this case, the output value of the operation is set to zero, as a convention, and the quality label codes the special output value:

Quality label	Description
0	No problems with calculation. Output was a real number.
1	A fatal error was encountered.
2	Output of the code was **NaN**.
3	Output of the code was **Inf**.
4	Output of the code was **-Inf**
5	Output had a non-zero imaginary component
6	Output was empty (e.g., `[]`)
7	Field specified for this operation did not exist in the master operation output structure

An example analysis workflow using *hctsa* in Matlab

At its core, *hctsa* analysis involves computing a library of time-series analysis features (which we call *operations*) on a time-series dataset.

The basic sequence of a Matlab-based *hctsa* analysis is to:

1. *Initialize* a `HCTSA.mat` file, which contains all of the information about the set of time series and operations in your analysis, as well as the results of applying all operations to all time series, using `TS_init`,
2. These operations can be computed on your time-series data using `TS_compute`. The results are structured in the local `HCTSA.mat` file containing matrices (that store the results of the computations) and structure arrays (that store information about the time-series data and operations), as described here.
3. After the computation is complete, a range of processing, analysis, and plotting functions are provided to understand and interpret the results.

Example 1: Compute a feature vector for a time series

As a quick check of your operation library, you can compute the full default code library on a time-series data vector (a column vector of real numbers) as follows:

```
    >> x = randn(500,1); % A random time-series
    >> featVector = TS_CalculateFeatureVector(x,0); % compute the default feature vect
or for x
```

Example 2: Analyze a time-series dataset

Suppose you have have a time-series dataset to analyze. You first generate a formatted `INP_ts.mat` input file containing your time series data and associated name and keyword labels, as described here. You then initialize an *hctsa* calculation using the default library of features:

```
>> TS_init('INP_ts.mat');
```

This generates a local file, `HCTSA.mat` containing the associated metadata for your time series, as well as information about the full time-series feature library (`Operations`) and the set of functions and code to call to evaluate them (`MasterOperations`), as described here.

Next you want to evaluate the code on all of the time series in your dataset. For this you can simply run:

```
>> TS_compute;
```

As described here, or, for larger datasets, using a script to regularly save back to the local file (cf. `sample_runscript_matlab`).

Having run your calculations, you may then want to label your data using the keywords you provided in the case that you have labeled groups of time series:

```
>> TS_LabelGroups;
```

and then normalize and filter the data using the default sigmoidal transformation:

```
>> TS_normalize;
```

A range of visualization scripts are then available to analyze the results, such as plotting the reordered data matrix:

```
>> TS_cluster; % compute a reordering of data and features
>> TS_plot_DataMatrix; % plot the data matrix
```

To inspect a low-dimensional representation of the data:

```
>> TS_plot_pca;
```

Or to determine which features are best at classifying the labeled groups of time series in your dataset:

```
>> TS_TopFeatures;
```

Each of these functions can be run with a range of input settings.

Compiling binaries

Some external code packages require compiled binary code to be used. Compilation of the mex code is handled by `compile_mex` as part of the `install` script, but the *TISEAN* package binaries need to be compiled separately in the command line.

Compiling mex code

Many of the operations (especially external code packages) rely on *mex* functions (pieces of code written in C or fortran), that need to be compiled to run natively on a given system architecture. To ensure that as many operations as possible run successfully on your data, you should compile these *mex* functions for your system. This requires working compilers (e.g., gcc, g++) to be installed on your system, which can be configured using `mex -setup` (cf. `doc mex` for more information).

Once mex is set up, the mex functions used in the time-series code repository can be compiled by navigating to the **Toolboxes** directory and then running `compile_mex`.

Compiling the *TISEAN* binaries

Some operations rely on the *TISEAN* nonlinear time-series analysis package, which Matlab accesses via the terminal using `system` commands, so the *TISEAN* binaries ***cannot*** be installed from within Matlab, but instead must be installed from the command line. If you are running Linux or Mac, we will assume that you are familiar with the command line, while those running Windows will require an alternate method to install *TISEAN*, as explained below.

Installing *TISEAN* on Linux or Mac

In the command line (**not within Matlab**), navigate to the **Toolboxes/Tisean_3.0.1** directory of the repository, then run the following chain of commands:

```
$ ./configure
$ make clean
$ make
$ make install
```

This should install the *TISEAN* binaries in your **~/bin/** directory (you can instead install into a system-wide directory, **/usr/bin**, for example, by running `./configure –prefix=/usr`). Additional information about the *TISEAN* installation process is provided on the *TISEAN* website.

If installation was successful then you should be able to access the newly-compiled binaries from the commandline, e.g., typing the command `which poincare` should return the path to the *TISEAN* function `poincare`. Otherwise, you should check that the install directory is in your system path, e.g., by adding the following:

```
export PATH=$PATH:$HOME/bin
```

to your **~/.bash_profile** (and running `source ~/.bash_profile` to update).

The path where *TISEAN* is installed will also have to be in Matlab's environment path, which is added by `startup.m`, assuming that the binaries are stored in **~/bin**. The `startup.m` code also adds the **DYLD_LIBRARY_PATH**, which is also required for *TISEAN* to function properly.

If you choose to use a custom location for the *TISEAN* binaries, that is not in the default Matlab system path (`getenv('PATH')` in Matlab), then you will have to add this path manually. You can test that Matlab can see the *TISEAN* binaries by typing, for example, the following into Matlab:

```
>> !which poincare
```

If Matlab's system paths are set up correctly, this command should return the path to your compiled *TISEAN* binary, `poincare`.

Installing *TISEAN* on Windows

If you are running Matlab from Windows, you will need a mechanism for Matlab to call `system` commands and find compiled TISEAN binaries. There are two options:

1. **Install Cygwin on your machine**. Cygwin provides a Linux distribution-like environment on Windows. Use this environment to compile and install TISEAN (as per the instructions above for Linux or Mac). Matlab will then also need to be launched from Cygwin, using the command: `matlab &`. This instance of Matlab should then be able to call `system` commands through cygwin, including the ability to access the *TISEAN* binaries.

2. **Sacrifice operations that rely on *TISEAN***. In total, *TISEAN*-based operations account for approximately 300 operations in the operation library. Although they provide important, well-tested implementations of nonlinear time-series analysis methods, it's not the end of the world if you decide it's too much trouble to install and are ok to miss out on these methods (see below on how to explicitly remove them from the library).

Ignoring *TISEAN* functions

If you decide not to use functions from the *TISEAN* package, they can be permanently removed from a given *hctsa* dataset.

1. To filter a local Matlab *hctsa* file (e.g., `HCTSA.mat`), you can use the following: `TS_local_clear_remove('ops',TS_getIDs('tisean','raw','ops'),1,'raw');`, which will remove all operations with the 'tisean' keyword from the *hctsa* dataset in `HCTSA.mat`.
2. If using a *mySQL* database, TISEAN functions can be removed from the database as follows: `SQL_clear_remove('ops',SQL_getids('ops',0,'tisean',{}),1)`.

Running Computations

An *hctsa* analysis requires setting a library of time series, master operations and operations, and generates a `HCTSA.mat` file (using `TS_init`), as described here. Once this is set up, computations are run using `TS_compute`.

These steps, as well as information on how to inspect the results of an *hctsa* analysis and working with HCTSA*.mat files, are provided in this chapter.

Initiating an *hctsa* analysis using a custom dataset

Formatted input files are used to set up a custom dataset of time-series data, pieces of Matlab code to run (master operations), and associated outputs from that code (operations). By default, you can simply specify a custom time-series dataset and the default operation library wil be used. In this section we describe how to initiate an *hctsa* analysis, including how to format the input files used in the *hctsa* framework.

Specifying a set of time series and operations using `TS_init`

Initiating a dataset for an *hctsa* analysis involves specifying an input file for each of:

1. the time series to analyze (`INP_ts.mat` or `INP_ts.txt`).
2. the code to run (`INP_mops.txt`).
3. the features to extract from that code (`INP_ops.txt`).

Details of how to format these input files are described below.

To use the default library of operations, you can initiate a time-series dataset (e.g., as specified in the .mat file, `INP_test_ts.mat`) using the following:

```
TS_init('INP_test_ts.mat');
```

To specify all sets of master operations and operations, you can use the following:

```
TS_init('INP_ts.mat','INP_mops.txt','INP_ops.txt');
```

`TS_init` produces a Matlab file, `HCTSA.mat` , containing all of the structures required to understand the set of time series, operations, and the results of their computation (explained here).

Through this initialization process, each time series will be assigned a unique ID, as will each master operation, and each operation.

Adding time series

When formatting a time series input file, two formats are available:

- *.mat file input*, which is suited to data that are already stored as variables in Matlab,
- *.txt file input*, which is better suited to when each time series is already stored as an individual text file.

Note that when using the .mat file input method, time-series data is stored in the database to six significant figures. However, when using the .txt file input method, time-series data values are stored in the database as written in the input text file of each time series.

Input file format 1 (.mat file)

When using a .mat file input, the .mat file should contain three variables:

- `timeSeriesData` : either a *N* x 1 cell (for *N* time series), where each element contains a vector of time-series values, or a *N* x *M* matrix, where each row specifies the values of a time series (all of length *M*).
- `labels` : a *N* x 1 cell of unique strings containing a named label for each time series.
- `keywords` : a *N* x 1 cell of strings, where each element contains a comma-delimited set of keywords (one for each time series), containing *no whitespace*.

An example involving two time series is below. In this example, we add two time series (showing only the first two values shown of each), which are labeled according to .dat files from a hypothetical EEG experiment, and assigned keywords (which are separated by commas and no whitespace). In this case, both are assigned keywords 'subject1' and 'eeg' and, additionally, the first time series is assigned 'trial1', and the second 'trial2' (these labels can be used later to retrieve individual time series). Note that the labels do not need to specify filenames, but can be any useful label for a given time series.

```
timeSeriesData = {[1.45,2.87,...],[8.53,-1.244,...]}; % (a cell of vectors)
labels = {'EEGExperiment_sub1_trail1.dat','EEGExperiment_sub1_trail2.dat'}; % data lab
els for each time series
keywords = {'subject1,trial1,eeg','subject1,trial2,eeg'}; % comma-delimited keywords f
or each time series

% Save these variables out to INP_test.mat:
save('INP_test.mat','timeSeriesData','labels','keywords');

% Initialize a new hctsa analysis using these data and the default feature library:
TS_init('INP_test.mat');
```

Input file format 2 (text file)

When using a text file input, the input file now specifies filenames of time series data files, which Matlab will then attempt to load (using `dlmread`). Data files should thus be accessible in the Matlab path. Each time-series text file should have a single real number on each row, specifying the ordered values that make up the time series. Once imported, the time-series data is stored in the database; thus the original time-series data files are no longer required, and can be removed from the Matlab path.

The input text file should be formatted as rows with each row specifying two whitespace separated entries: (i) the file name of a time-series data file and (ii) comma-delimited keywords.

For example, consider the following input file, containing three lines (one for each time series to be added to the database):

```
gaussianwhitenoise_001.dat     noise,gaussian
gaussianwhitenoise_002.dat     noise,gaussian
sinusoid_001.dat               periodic,sine
```

Using this input file, a new analysis will contain 3 time series, **gaussianwhitenoise_001.dat** and **gaussianwhitenoise_002.dat** will be assigned the keywords ‘noise’ and ‘gaussian’, and the data in **sinusoid_001.dat** will be assigned keywords ‘periodic’ and ‘sine’. Note that keywords should be separated *only* by commas (and no whitespace).

Adding master operations

In our system, a *master operation* refers to a piece of Matlab code and a set of input parameters.

Valid outputs from a master operation are:

1. A single real number,
2. A structure containing real numbers,
3. **NaN** to indicate that the input time series is not appropriate for this code.

The (potentially many) outputs from a master operation can thus be mapped to individual operations (or features), which are single real numbers summarizing a time series that make up individual columns of the resulting data matrix.

Two example lines from the input file, **INP_mops.txt** (in the **Database** directory of the repository), are as follows:

```
CO_tc3(y,1)     CO_tc3_y_1
ST_length(x)    ST_length
```

Each line in the input file specifies two pieces of information, separated by whitespace:

1. A piece of code and its input parameters.
2. A unique label for that master operation (that can be referenced by individual operations).

We use the convention that *x* refers to the input time series and *y* refers to a *z*-scored transformation of the input time series (i.e., $(x - \mu_x)/\sigma_x$). In the example above, the first line thus adds an entry in the database for running the code `CO_tc3` using a *z*-scored time series as input (*y*), with '1' as the second input with the label **CO_tc3_y_1**, and the second line will add an entry for running the code `ST_length` using the non-*z*-scored time-series *x*, with the label **length**.

When the time comes to perform computations on data using the methods in the database, Matlab needs to have path access to each of the master operations functions specified in the database. For the above example, Matlab will attempt to run both `CO_tc3(y,1)` and `ST_length(x)`, and thus the functions `CO_tc3.m` and `ST_length.m` must be in the Matlab path. Recall that the script `startup.m`, which should be run at the start of each session using *hctsa*, handles the addition of paths required for the default code library.

Adding operations (features)

The input file, e.g., `INP_ops.txt` (in the **Database** directory of the repository) should contain a row for every operation, and use labels that correspond to master operations. An example excerpt from such a file is below:

```
CO_tc3_y_1.raw     CO_tc3_1_raw      correlation,nonlinear
CO_tc3_y_1.abs     CO_tc3_1_abs      correlation,nonlinear
CO_tc3_y_1.num     CO_tc3_1_num      correlation,nonlinear
CO_tc3_y_1.absnum  CO_tc3_1_absnum   correlation,nonlinear
CO_tc3_y_1.denom   CO_tc3_1_denom    correlation,nonlinear
ST_length          length            raw,lengthDependent
```

The first column references a corresponding master label and, in the case of master operations that produce structure, the particular field of the structure to reference (after the fullstop), the second column denotes the label for the operation, and the final column is a set of comma-delimited keywords (that must not include whitespace). Whitespace is used to separate the three entries on each line of the input file. In this example, the master operation labeled `CO_tc3_y_1`, outputs is a structure, with fields that are referenced by the first five operations listed here, and the `ST_length` master operation outputs a single number (the length of the time series), which is referenced by the operation named **'length'** here. The two keywords 'correlation' and 'nonlinear' are added to the `CO_tc3_1` operations, while the

keywords ‘raw’ and ‘lengthDependent’ are added to the operation called `length`. These keywords can be used to organize and filter the set of operations used for a given analysis task.

Performing calculations using `TS_compute`

An *hctsa* dataset has been initialized (specifying details of a time-series dataset and operations to include using `TS_init`), all results entries in the resulting `HCTSA.mat` are set to `NaN`, corresponding to results that are as yet uncomputed.

Calculations are performed using the function `TS_compute`, which stores results back into the matrices in `HCTSA.mat`. This function can be run without inputs to compute all missing values in the default hctsa file, `HCTSA.mat`:

```
% Compute all missing values in HCTSA.mat:
TS_compute();
```

`TS_compute` will begin evaluating operations on time series in `HCTSA.mat` for which elements in `TS_DataMat` are `NaN` (i.e., computations that have not been run previously). Results are stored back in the matrices of `HCTSA.mat`: `TS_DataMat` (output of each operation on each time series), `TS_CalcTime` (calculation time for each operation on each time series), and `TS_Quality` (labels indicating errors or special-valued outputs).

Custom settings for running `TS_compute`

(1) Computing features in parallel across available cores using Matlab's Parallel Processing Toolbox. This can be achieved by setting the first input to `true`:

```
% Compute all missing values in HCTSA.mat using parallel processing:
TS_compute(true);
```

(2) Computing across a custom range of time-series IDs (`ts_id`) and operation IDs (`op_id`). This can be achieved by setting the second and third inputs:

```
% Compute missing values in HCTSA.mat for ts_ids from 1:10 and op_ids from 1:1000
TS_compute(false,1:10,1:1000);
```

(3) Specifying what types of values should be computed:

```
    % Compute all values that have never been computed before (default)
    TS_compute(false,[],[],'missing');
    % Compute all values that have never previous been calculated OR have previously b
een computed but returned an error:
    TS_compute(false,[],[],'error');
```

(4) Specifying a custom .mat file to operate on (`HCTSA.mat` is the default):

```
    % Compute all missing values in my_HCTSA_file.mat:
    TS_compute(false,[],[],'missing','my_HCTSA_file.mat');
```

(5) Suppress commandline output. All computations are displayed to screen by default (which can be overwhelming but is useful for error checking). This functionality can be suppressed by setting the final (6th) input to `false` :

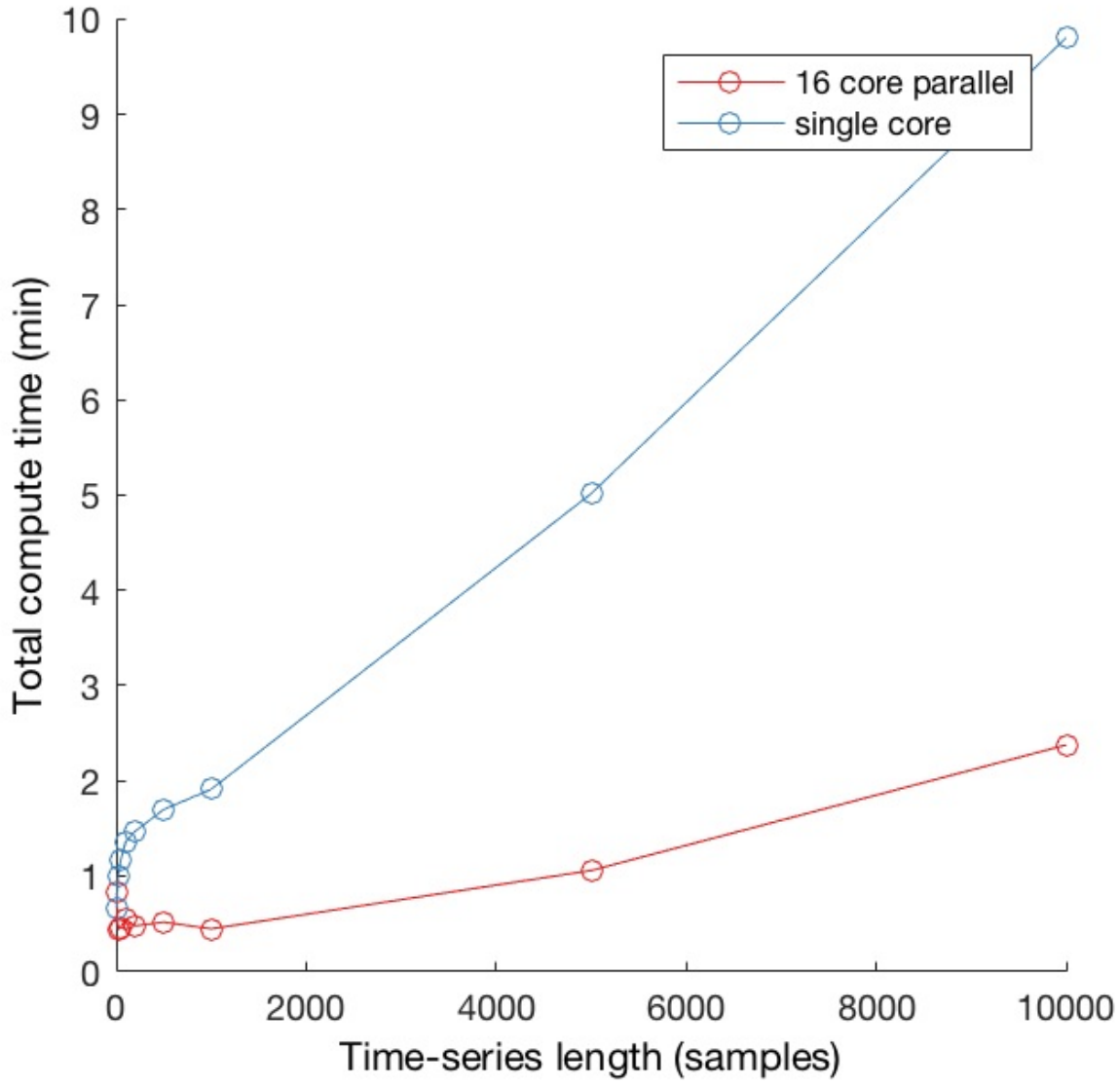
```
    % Compute all missing values in HCTSA.mat, suppressing output to screen:
    TS_compute(false,[],[],'missing','',false);
```

Computation approaches for full datasets

Computing features for full time-series datasets can be time consuming, especially for large datasets of long time series. An appropriate strategy therefore depends on the time-series length, the number of time series in the dataset, and the computational resources available. When multiple cores are available, it is always recommended to use the parallel setting (i.e., as `TS_compute(true)`).

Computation time scaling

The first thing to think about is how the time taken to compute 7749 features of v0.93 of *hctsa* scales with the length of time series in your dataset (see plot below). The figure compares results using a single core (e.g., `TS_compute(false)`) to results using a 16-core machine, with parallelization enabled (e.g., `TS_compute(true)`).



Times may vary across on individual machines, but the above plot can be used to estimate the computation time per time series, and thus help decide on an appropriate computation strategy for a given dataset.

Note that if computation times are too long for the computational resources at hand, one can always choose a reduced set of features, rather than the full set of >7000, to get a preliminary understanding of the dataset. One such reduced set of features is in `INP_ops_reduced.txt`. We plan to reduced additional reduced feature sets, determined according to different criteria, in future.

On a single machine

If only a single machine is available for computation, there are a couple of options:

1. For small datasets, when it is feasible to run all computations in a single go, it is easiest

to run computations within Matlab in a single call of `TS_compute`.

2. For larger datasets that may run for a long time on a single machine, one may wish to use something like the provided `sample_runscript_matlab` script, where `TS_compute` commands are run in a loop over time series, compute small sections of the dataset at a time (and then saving the results to file, e.g., `HCTSA.mat`), eventually covering the full dataset iteratively.

On a distributed compute cluster using Matlab

With a distributed computing setup, a local Matlab file (`HCTSA.mat`) can be split into smaller pieces using `TS_subset`, which outputs a new data file for a particular subset of your data, e.g., `TS_subset('raw',1:100)` will generate a new file, `HCTSA_subset.mat` that contains just time series with IDs from 1 to 100. Computing features for time series in each such subset can then be run on a distributed computing setup. For example, with a different compute node computing a different subset (by queuing batch jobs that each work on a given subset of time series). Once all subsets have been computed, the results can then be recombined into a single `HCTSA.mat` file using `TS_combine` commands, as described here.

Using mySQL to facilitate distributed computing

Distributing feature computations on a large-scale distributed computing setup can be better suited to a linked mySQL database, especially for datasets that grow with time, as new time series can be easily added to the database. In this case, computation proceeds similarly to above, where shell scripts on a distributed cluster computing environment can be used to distribute jobs across cores, with all individual jobs writing to a centralized *mySQL* server. A set of Matlab code that generates an appropriately formatted mySQL database and interfaces with the database to facilitate *hctsa* feature computation is included with the software package, and is described in detail here.

Visualizing special values and errors using `TS_InspectQuality`

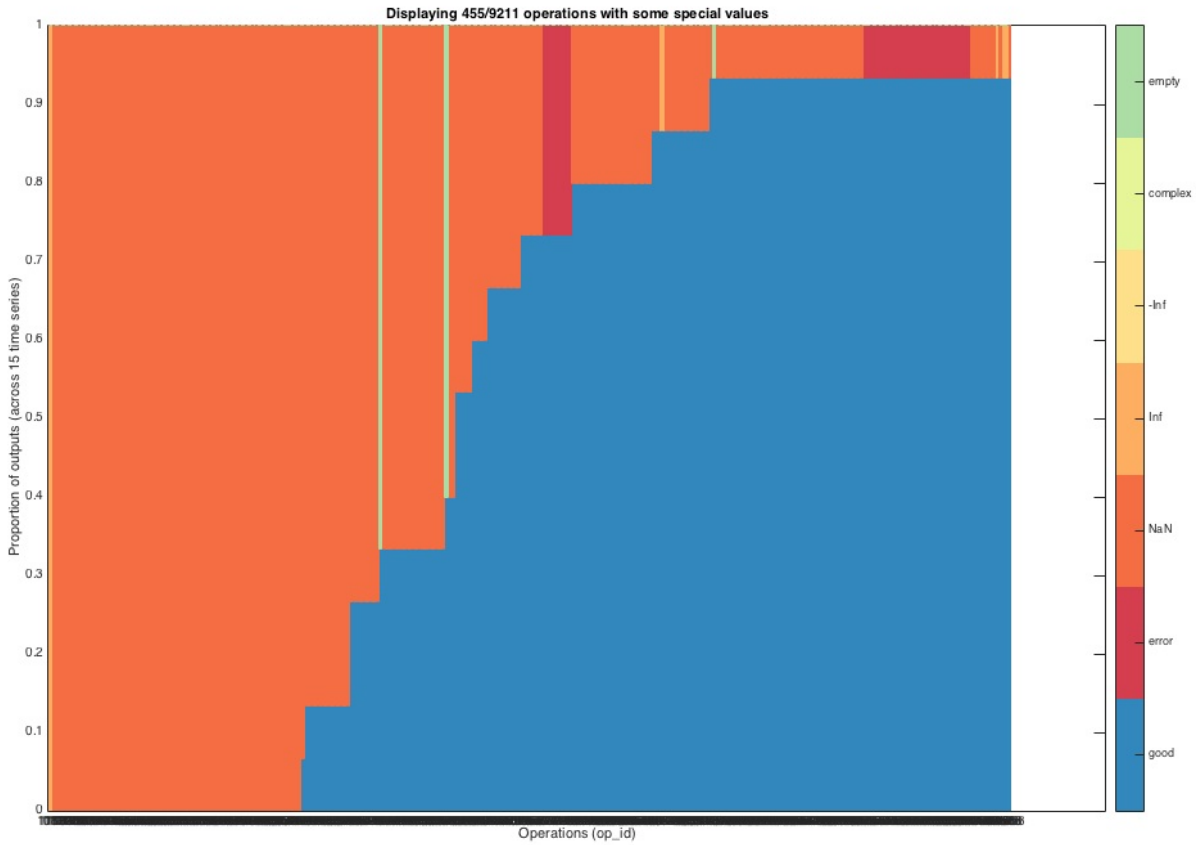
When applying thousands of time-series analysis methods to diverse datasets, many operations can give results that are not real numbers. Some time series may be inappropriate for a given operation (such as fitting a positive-only distribution to data that is not positive), or measuring stationarity across 2,000 datapoints in time series that are shorter than 2,000 samples. Other times, an optimization routine may fail, or some unknown error may be called.

Some errors are not problems with the code, but represent issues with applying particular sets of code to particular time series, such as when a Matlab fitting function reaches the maximum number of iterations and returns an error. Other errors are genuine problems with the code that need to be corrected. Both cases are labeled as errors in our framework.

It can be good practice to visualize where special values and errors are occurring after a computation to see where things might be going wrong, using `TS_InspectQuality`. This can be run in four modes:

1. `TS_InspectQuality('summary');` [default] Summarizes the proportion of special-valued outputs in each operation as a bar plot, ordered by the proportion of special-valued outputs.
2. `TS_InspectQuality('master');` Plots which types of special-valued outputs were encountered for each master operation.
3. `TS_InspectQuality('full');` Plots the full data matrix (all time series as rows and all operations as columns), and shows where each possible special-valued output can occur (including 'error', 'NaN', 'Inf', '-Inf', 'complex', 'empty', or a 'link error').
4. `TS_InspectQuality('reduced');` As `'full'`, but includes only columns where special values occurred.

For example, running `TS_InspectQuality('summary')` loads in data from **HCTSA.mat** and produces the following, which can be zoomed in on and explored to understand which features are producing problematic outputs:



Errors with compiled code

Note that errors returned from Matlab files do not halt the progress of the computation (using `try-catch` statements), but errors with compiled **mex** functions (or external commandline packages like TISEAN) can produce a fault that crashes Matlab or the system. We have performed some basic testing on all mex functions, but for some unusual time series, such faults may still occur. These situations must be dealt with by either identifying and fixing the problem in the original source code and recompiling, or by removing the problem code.

Troubleshooting errors

When getting information on operations that produce special-valued outputs (getting IDs listed from `TS_InspectQuality`), it can be useful to then test examples by re-running pieces of code with the problematic data. The function `TS_WhichProblemTS` can be used to retrieve time series from an *hctsa* dataset that caused a problem for a given operation.

Usage is as follows:

```
% Find time series that failed for the operation with ID = 684.
[ts_ind, dataCell, codeEval] = TS_WhichProblemTS(684);
```

This provides the list of time series IDs (`ts_ind`), their time-series data vectors (`dataCell`), and the code to evaluate the given operation (in this case, the master operation code corresponding to the operation with ID 684).

You can then pick an example time series (e.g., the first problem time series: `x = dataCell{1}; y = zscore(x)`), and then copy and paste the code in codeEval into the command line to evaluate the code for this time series. This method allows easy debugging and inspection of examples of time-series data that caused problems for particular operations flagged through the `TS_InspectQuality` process.

Working with *hctsa* files

When running *hctsa* analyses, often you want to take subsets of time series (to look in more detail at a subset of your data) or subsets of operations (to explore the behavior of different feature subsets), or combine multiple subsets of data together (e.g., as additional data arrive).

The *hctsa* package contains a range of functions for these types of tasks, working directly with *hctsa* .mat files, and are described below. Note that these types of tasks are easier to manage when *hctsa* data are stored in a mySQL database.

Retrieving time series (or operations) of interest by matching on assigned keywords using `TS_getIDs`

Many time-series classification problems involve filtering subsets of time series based on keyword matching, where keywords are specified in the input file provided when initializing a dataset.

Most filtering functions (such as those listed in this section), require you to specify a range of IDs of TimeSeries or Operations in order to specify them. Recall that each TimeSeries and Operation is assigned a unique ID (assed as the ID field in the corresponding structure array). To quickly get the IDs of time series that match a given keyword, the following function can be used:

```
TimeSeriesIDs = TS_getIDs(theKeyword,'HCTSA_N.mat');
```

Or the IDs of operations tagged with the 'entropy' keyword:

```
OperationIDs = TS_getIDs('entropy','norm','ops');
```

These IDs can then be used in the functions below (e.g., to clear data, or extract a subset of data).

Note that to get a quick impression of the unique time-series keywords present in a dataset, use the function `TS_WhatKeywords`, which gives a text summary of the unique keywords in an *hctsa* dataset.

Clearing or removing data from an *hctsa* dataset using `TS_local_clear_remove`

Sometimes you may want to remove a time series from an *hctsa* dataset because the data was not properly processed, for example. Or one operation may have produced errors because of a missing toolbox reference, or you may have altered the code for an operation, and want to clear the stored results from previous calculations.

For example, often you want to remove from your operation library operations that are dependent on the location of the data (e.g., its mean: `'locdep'`), that only operate on positive-only time series (`'posOnly'`), that require the TISEAN package (`'tisean'`), or that are stochastic (i.e., they give different results when repeated, `'stochastic'`).

The function `TS_local_clear_remove` achieves these tasks when working directly with .mat files (NB: if using a mySQL database, `SQL_clear_remove` should be used instead).

`TS_local_clear_remove` loads in a an *hctsa* .mat data file, clears or removes the specified time series or operations, and then writes the result back to the file.

Example 1: Clear all computed data from time series with IDs 1:5 from `HCTSA.mat` (specifying `'raw'`):

```
TS_local_clear_remove('ts',1:5,0,'raw');
```

Example 2: Remove all operations with the keyword 'tisean' (that depend on the TISEAN package) from `HCTSA.mat`:

```
TS_local_clear_remove('ops',TS_getIDs('tisean','raw','ops'),1,'raw');
```

Example 3: Remove all operations that require positive-only data (the `'posOnly'` keyword) from `HCTSA.mat`:

```
TS_local_clear_remove('ops',TS_getIDs('posOnly','raw','ops'),1,'raw');
```

Example 4: Remove all operations that are location dependent (the `'locdep'` keyword) from `HCTSA.mat`:

```
TS_local_clear_remove('ops',TS_getIDs('locdep','raw','ops'),1,'raw');
```

See the documentation in the function file for additional details about the inputs to `TS_local_clear_remove`.

Extracting a subset from an *hctsa* dataset using `TS_subset`

Sometimes it's useful to retrieve a subset of an *hctsa* dataset, when analyzing just a particular class of time series, for example, or investigating a balanced subset of data for time-series classification, or to compare the behavior of a reduced subset of features. This can be done with `TS_subset`, which takes in a *hctsa* dataset and generates the desired subset, which can be saved to a new .mat file.

Example 1: Import data from 'HCTSA_N.mat', then save a new dataset containing only time series with IDs in the range 1--100, and all operations, to 'HCTSA_N_subset.mat' (see documentation for all inputs).

```
TS_subset('norm',1:100,[],1,'HCTSA_N_subset.mat')
```

Note that the subset in this case will have be normalized using the full dataset of all time series, and just this subset (with IDs up to 100) are now being analyzed. Depending on the normalization method used, different results would be obtained if the subsetting was performed prior to normalization.

Example 2: From `HCTSA.mat` (`'raw'`), save a subset of that dataset to 'HCTSA_healthy.mat' containing only time series tagged with the 'healthy' keyword:

```
TS_subset('raw',TS_getIDs('healthy','raw'),[],1,'HCTSA_healthy.mat')
```

Combining multiple *hctsa* datasets using `TS_combine`

When analyzing a growing dataset, sometimes new data needs to be combined with computations on existing data. Alternatively, when computing a large dataset, sometimes you may wish to compute sections of it separately, and may later want to combine each section into a full dataset.

To combine *hctsa* data files, you can use the `TS_combine` function.

Example: combine *hctsa* datasets stored in the files `HCTSA_healthy.mat` and `HCTSA_disease.mat` into a new combined file, `HCTSA_combined.mat` :

```
TS_combine('HCTSA_healthy.mat','HCTSA_disease.mat',0,'HCTSA_combined.mat')
```

The third input, `compare_tsids`, controls the behavior of the function in combining time series. By setting this to 1, `TS_combine` assumes that the TimeSeries IDs are comparable between the datasets (e.g., most common when using a *mySQL* database to store *hctsa* data), and thus filters out duplicates so that the resulting *hctsa* dataset contains a unique set of time series. By setting this to 0 (default), the output will contain a union of time series present in each of the two *hctsa* datasets. In the case that duplicate TimeSeries IDs exist in the combination file, a new index will be generated in the combined file (where IDs assigned to time series are re-assigned as unique integers using `TS_ReIndex`).

In combining operations, this function works differently when data have been stored in a unified *mySQL* database, in which case operation IDs can be compared meaningfully and combined as an intersection. However, when *hctsa* datasets have been generated using `TS_init`, the function will check that the same set of operations have been used in both files.

Performing highly comparative analysis

Once a set of operations have been computed on a time-series dataset, the results are stored in a local **HCTSA.mat** file. The result can be used to perform a wide variety of highly comparative analyses, such as those outlined in our paper.

The type of analysis employed should the be motivated by the specific time-series analysis task at hand. Setting up the problem, guiding the methodology, and interpreting the results requires strong scientific input that should draw on domain knowledge, including the questions asked of the data, experience performing data analysis, and statistical considerations.

The first main component of an *hctsa* analysis involves filtering and normalizing the data using `TS_normalize`, described here, which produces a file called **HCTSA_N.mat**. Information about the similarity of pairs of time series and operations can be computed using `TS_cluster`, described here which stores this information in **HCTSA_N.mat**. The suite of plotting and analysis tools that we provide with *hctsa* work with this normalized data, stored in **HCTSA_N.mat**, by default.

Plotting and analysis functions:

- Visualizing structure in the data matrix using `TS_plot_DataMatrix`.
- Visualizing the time-series traces using `TS_plot_TimeSeries`.
- Visualizing low-dimensional structure in the data using `TS_plot_pca`.
- Exploring similar matches to a target time series using `TS_SimSearch`.
- Visualizing the behavior of a given operation across the time-series dataset using `TS_FeatureSummary`.

Tools for classification tasks:

For time-series classification tasks, groups of time series can be labeled using the `TS_LabelGroups` function described here; this group label information is stored in the local **HCTSA*.mat** file, and used by default in the various plotting and analysis functions provided. Additional analysis functions are provided for basic time-series classification tasks:

- Explore the classification performance of the full library of features using `TS_classify`
- Determine the features that (individually) best distinguish between the labeled groups using `TS_TopFeatures`

Assigning group labels to time series using `TS_LabelGroups`

Highly comparative analyses often involve classification tasks, in which each observation is assigned a (numeric) class label. Once data has been retrieved, as described above, class labels can be assigned to each time series in a dataset, and stored in the local `HCTSA*.mat` files using the function `TS_LabelGroups`.

The example below assigns labels to two groups of time series in the `HCTSA.mat` (specifying the shorthand `'raw'` for this default, un-normalized data), corresponding to those labeled as 'parkinsons' and those labeled as 'healthy':

```
TS_LabelGroups({'parkinsons','healthy'},'raw');
```

The first input is a cell specifying the keyword string to use to match each group.

To automatically detect unique keywords for labelling, `TS_LabelGroups` can be run with an empty first input, as `TS_LabelGroups([],'raw');`

By default, this function saves the group indices back to the data file (in this example, `HCTSA.mat`), by adding a new field, `Group`, to the `TimeSeries` structure array, which contains the group index of each time series.

Group indices stay with the time series they are assigned to after filtering and normalizing the data (using `TS_normalize`). Group labels can be reassigned at any time by re-running the `TS_LabelGroups` function.

Group labels are used by a range of analysis functions, including `TS_plot_pca`, `TS_TopFeatures`, and `TS_classify`.

Filtering, and normalizing data using `TS_normalize`

The first step in analyzing a dataset involves processing the data matrix, which can be done using `TS_normalize`. This involves filtering out operations or time series that produced many errors or special-valued outputs, and then normalizing of the output of all operations, which is typically done in-sample, according to an outlier-robust sigmoidal transform (although other normalizing transformations can be selected). Both of these tasks are performed using the function `TS_normalize`. The `TS_normalize` function writes the new, filtered, normalized matrix to a local file called `HCTSA_N.mat`. This contains normalized, and trimmed versions of the information in `HCTSA.mat`.

Example usage is as follows:

```
TS_normalize('scaledRobustSigmoid',[0.8,1.0]);
```

The first input controls the normalization method, in this case a scaled, outlier-robust sigmoidal transformation, specified with 'scaledRobustSigmoid'. The second input controls the filtering of time series and operations based on minimum thresholds for good values in the corresponding rows (corresponding to time series; filtered first) and columns (corresponding to operations; filtered second) of the data matrix.

In the example above, time series (rows of the data matrix) with more than 20% special values (specifying 0.8) are first filtered out, and then operations (columns of the data matrix) containing any special values (specifying 1.0) are removed. Columns with approximately constant values are also filtered out. After filtering the data matrix, the outlier-robust ‘scaledRobustSigmoid’ sigmoidal transformation is applied to all remaining operations (columns). The filtered, normalized matrix is saved to the file `HCTSA_N.mat`.

Details about what normalization is saved to the `HCTSA_N.mat` file as `normalizationInfo`, a structure that contains the normalization function, filtering options used, and the corresponding `TS_normalize` code that can be used to re-run the normalization.

Setting the normalizing transformation

It makes sense to weight each operation equally for the purposes dimensionality reduction, and thus normalize all operations to the same range using a transformation like ‘scaledRobustSigmoid’, ‘scaledSigmoid’, or ‘mixedSigmoid’. For the case of calculating mutual information distances between operations, however, one would rather not distort the

distributions and perform no normalization, using ‘raw’ or a linear transformation like ‘zscore’, for example. The list of implemented normalization transformations can be found in the function `BF_NormalizeMatrix`.

Note that the 'scaledRobustSigmoid' transformation does not tolerate distributions with an interquartile range of zero, which will be filtered out.

Setting the filtering parameters

Filtering parameters depend on the application. Some applications can allow the filtering thresholds can be relaxed. For example, setting `[0.7,0.9]`, removes time series with less than 70% good values, and then removes operations with less than 90% good values. Some applications can tolerate some special-valued outputs from operations (like some clustering methods, where distances are simply calculated using those operations that are did not produce special-valued outputs for each pair of objects), but others cannot (like Principal Components Analysis); the filtering parameters should be specified accordingly.

Analysis can be performed on the data contained in `HCTSA_N.mat` in the knowledge that different settings for filtering and normalizing the results can be applied at any time by simply rerunning `TS_normalize`, which will overwrite the existing `HCTSA_N.mat` with the results of the new normalization and filtration settings.

Clustering the data matrix using `TS_cluster`

For the purposes of visualizing the data matrix, it is often desirable to have the rows and columns reordered to put similar rows adjacent to one another, and similarly to place similar columns adjacent to one another. This reordering can be done using hierarchical linkage clustering, by the function `TS_cluster`:

```
distanceMetricRow = 'euclidean'; % time-series feature distance
linkageMethodRow = 'average'; % linkage method
distanceMetricCol = 'corr_fast'; % a (poor) approximation of correlations with NaNs
linkageMethodCol = 'average'; % linkage method

TS_cluster(distanceMetricRow, linkageMethodRow, distanceMetricCol, linkageMethodCol);
```

This function reads in the data from `HCTSA_N.mat`, and stores the re-ordering of rows and columns back into `HCTSA_N.mat` in the `ts_clust` and `op_clust` (and, if the size is manageable, also the pairwise distance information). Visualization functions (such as `TS_plot_DataMatrix` and `TS_SimSearch`) can then take advantage of this information, using the general input label `'cl'`.

Visualizing the data matrix using `TS_plot_DataMatrix`

The clustered data matrix (if clustering has been performed, otherwise the non-clustered data matrix is used) can be visualized by running

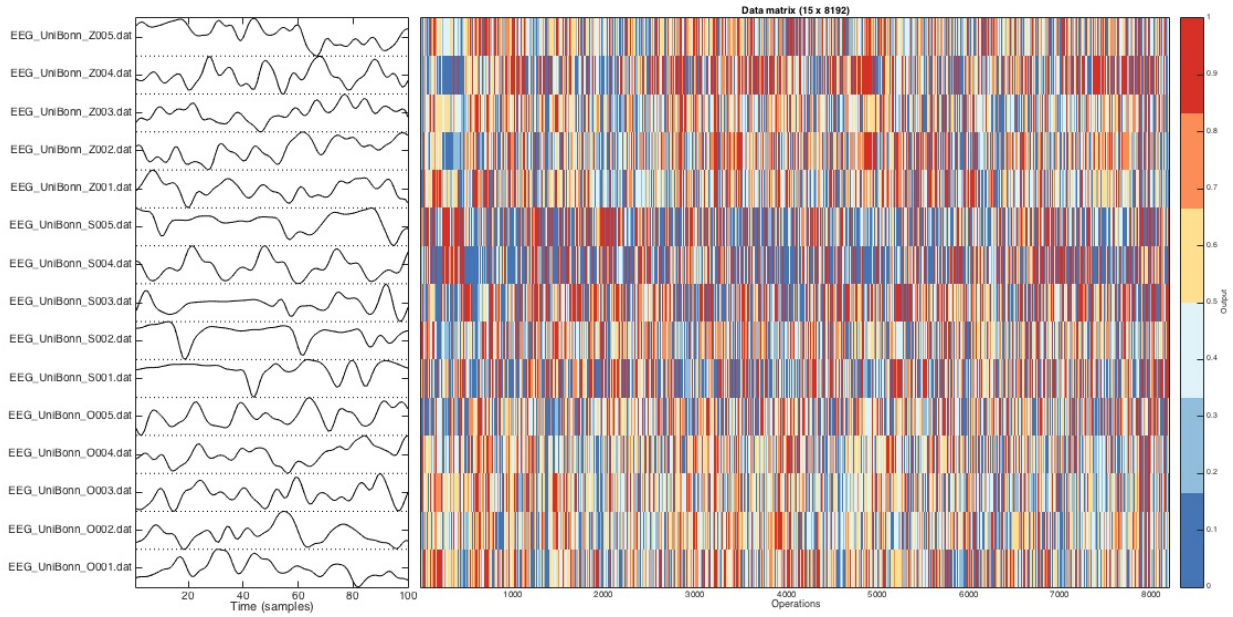
```
TS_plot_DataMatrix
```

This will produce a colored visualization of the data matrix such as that shown below.

When data is grouped according to a set of distinct keywords and stored as group metadata (using the `TS_LabelGroups` function), these can also be visualized using `TS_plot_DataMatrix('colorGroups',1)`.

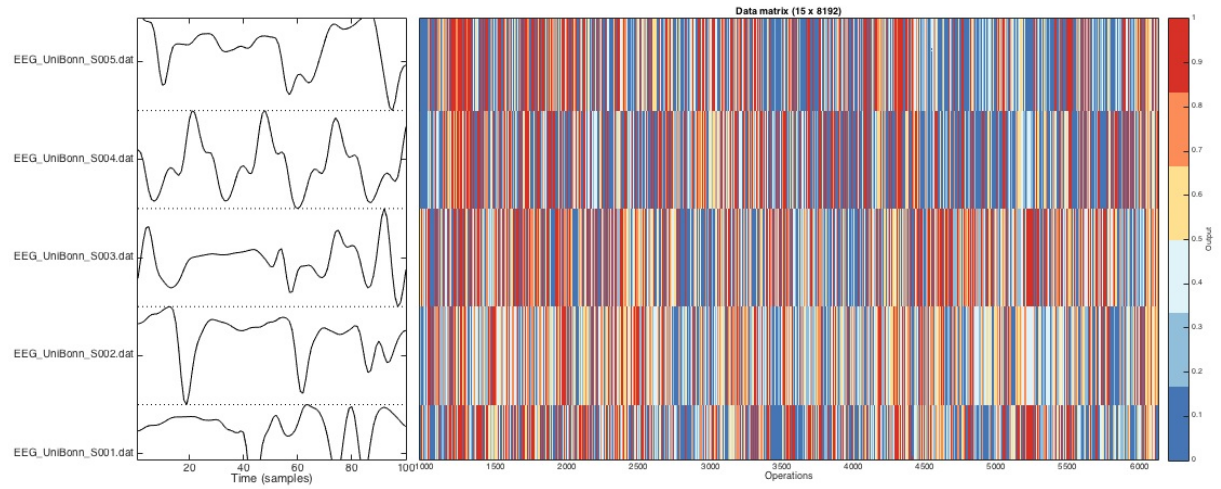
Visualizing the normalized (unclustered) data matrix

Running `TS_plot_DataMatrix('norm')` plots the data contained in the local file `HCTSA_N.mat`, yielding:



where black rectangles label missing values, and other values are shown from low (blue) to high (red) after normalization using the scaled outlier-robust sigmoidal transformation. Due to the size of the matrix, operations are not labeled.

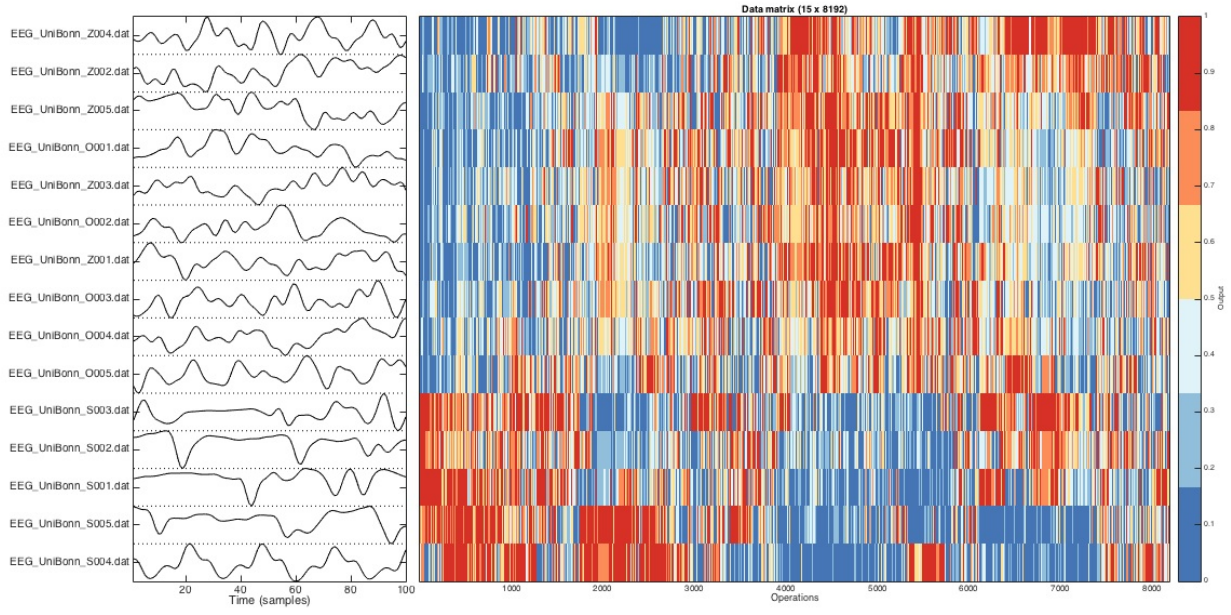
Examples of time series segments are shown to the left of the plot, and when the middle plot is zoomed, the time-series annotations remain matched to the data matrix:



Visualizing the clustered data matrix

It can be useful to display the matrix with the order of time series and operations preserved, but the relationships between rows and columns can be difficult to visualize when ordered randomly.

By running `TS_plot_DataMatrix('cl')`, the clustering information contained in `HCTSA_N.mat` is used to reorder the time series and features by similarity (if this information exists, cf. `TS_cluster`), yielding:



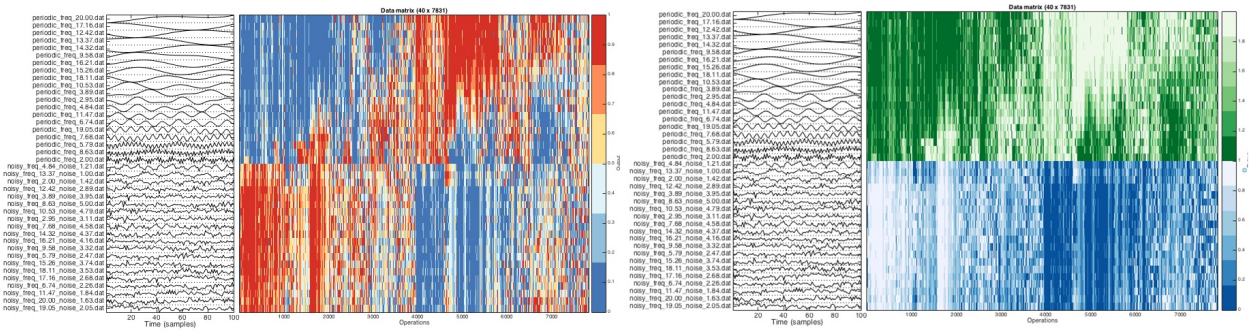
By reordering rows and columns, this representation reveals correlated patterns of outputs across different types of operations, and similar sets of properties between different types of time series.

Example: Incorporating group information

In this example, we consider a set of 20 periodic and 20 noisy periodic signals. We assigned the time series to groups (using `TS_LabelGroups('orig',{'periodic','noisy'},'ts')`), normalized the data matrix (`TS_normalize`), and then clustered it (`TS_cluster`). So now we have a clustered data matrix containing thousands of summaries of each time series, as well as pre-assigned group information as to which time series are periodic and which are noisy. When the time series have been assigned to groups , this can be accessed by setting the second input to 1:

```
    TS_plot_DataMatrix('cl','colorGroups',false); % don't color according to group lab
els
    TS_plot_DataMatrix('cl','colorGroups',true); % color according to group labels
```

producing the following two plots:



When group information is *not used* (the left plot), the data is visualized in the default blue/yellow/red color scheme, but when the assigned groups are colored (right plot), we see that the clustered dataset separates perfectly into the periodic (green) and noisy (blue) time series, and we can visualize the features that contribute to the separation.

Plotting the time series

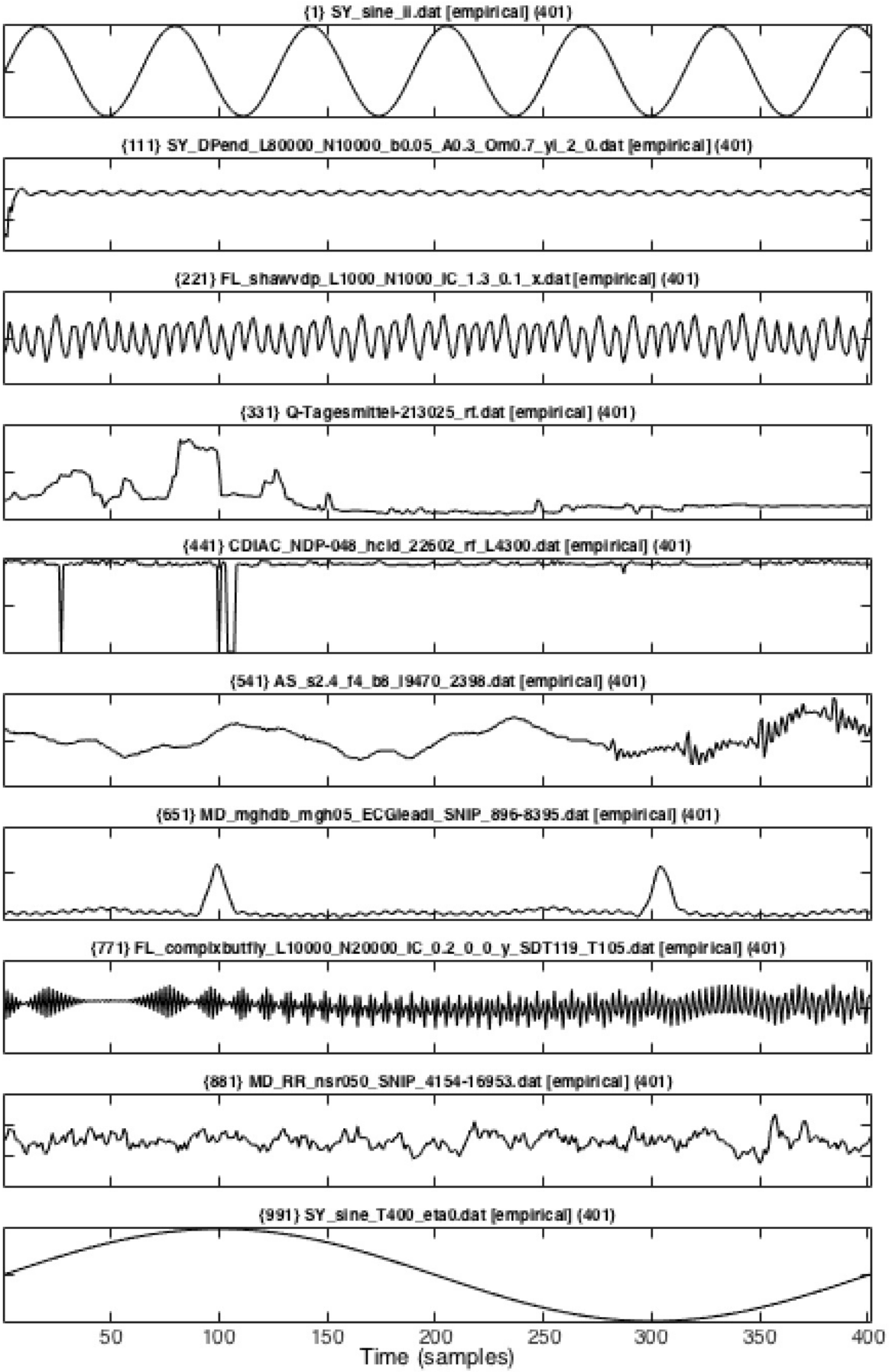
The *hctsa* package provides a simple means of plotting time series, via the `TS_plot_timeseries` function.

Basic plotting

For example, to plot a set of time series that have not been assigned groups, we can run the following:

```
whatData = 'norm'; % Get data from HCTSA_N.mat
plotWhatTimeSeries = 'all'; % plot examples from all time series
plotHowMany = 10; % how many to plot
maxLength = 400; % maximum number of samples to plot for each time series
TS_plot_timeseries(whatData,plotHowMany,plotWhatTimeSeries,maxLength);
```

For our assorted set of time series, this produces the following:

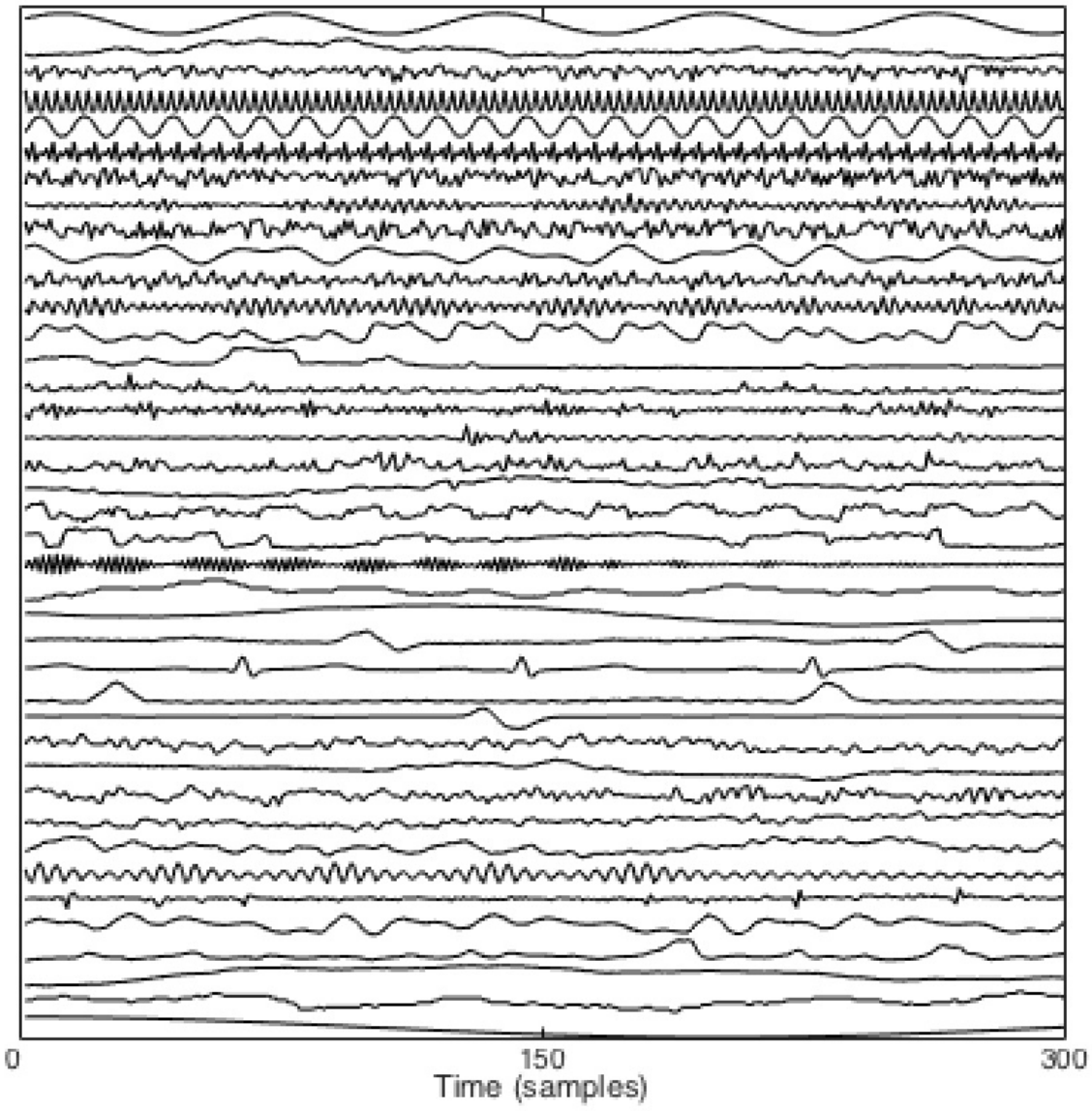

{1} SY_sine_ll.dat [empirical] (401)
{111} SY_DPend_L80000_N10000_b0.05_A0.3_Om0.7_yl_2_0.dat [empirical] (401)
{221} FL_shawvdp_L1000_N1000_IC_1.3_0.1_x.dat [empirical] (401)
{331} Q-Tagesmittel-213025_rt.dat [empirical] (401)
{441} CDIAC_NDP-048_hcld_22602_rf_L4300.dat [empirical] (401)
{541} AS_s2.4_f4_b8_l9470_2398.dat [empirical] (401)
{651} MD_mghdb_mgh05_ECGleadI_SNIP_896-8395.dat [empirical] (401)
{771} FL_complxbutfly_L10000_N20000_IC_0.2_0_0_y_SDT119_T105.dat [empirical] (401)
{881} MD_RR_nsr050_SNIP_4154-16953.dat [empirical] (401)
{991} SY_sine_T400_eta0.dat [empirical] (401)
50
100
150
200
250
300
350
400
Time (samples)

Showing the first 400 samples of 10 selected time series, equally-spaced through the TimeSeries IDs in `HCTSA_N.mat`.

Freeform plotting

Many more custom plotting options are available by passing an options structure to `TS_plot_timeseries`, including the `'plotFreeForm'` option which allows very many time series to be shown in a single plot (without the usual axis borders):

```
% Plot as a freeform plot without labeling time series:
plotOptions = struct('plotFreeForm',true,'displayTitles',false);
TS_plot_timeseries('norm',40,'all',300,plotOptions);
```

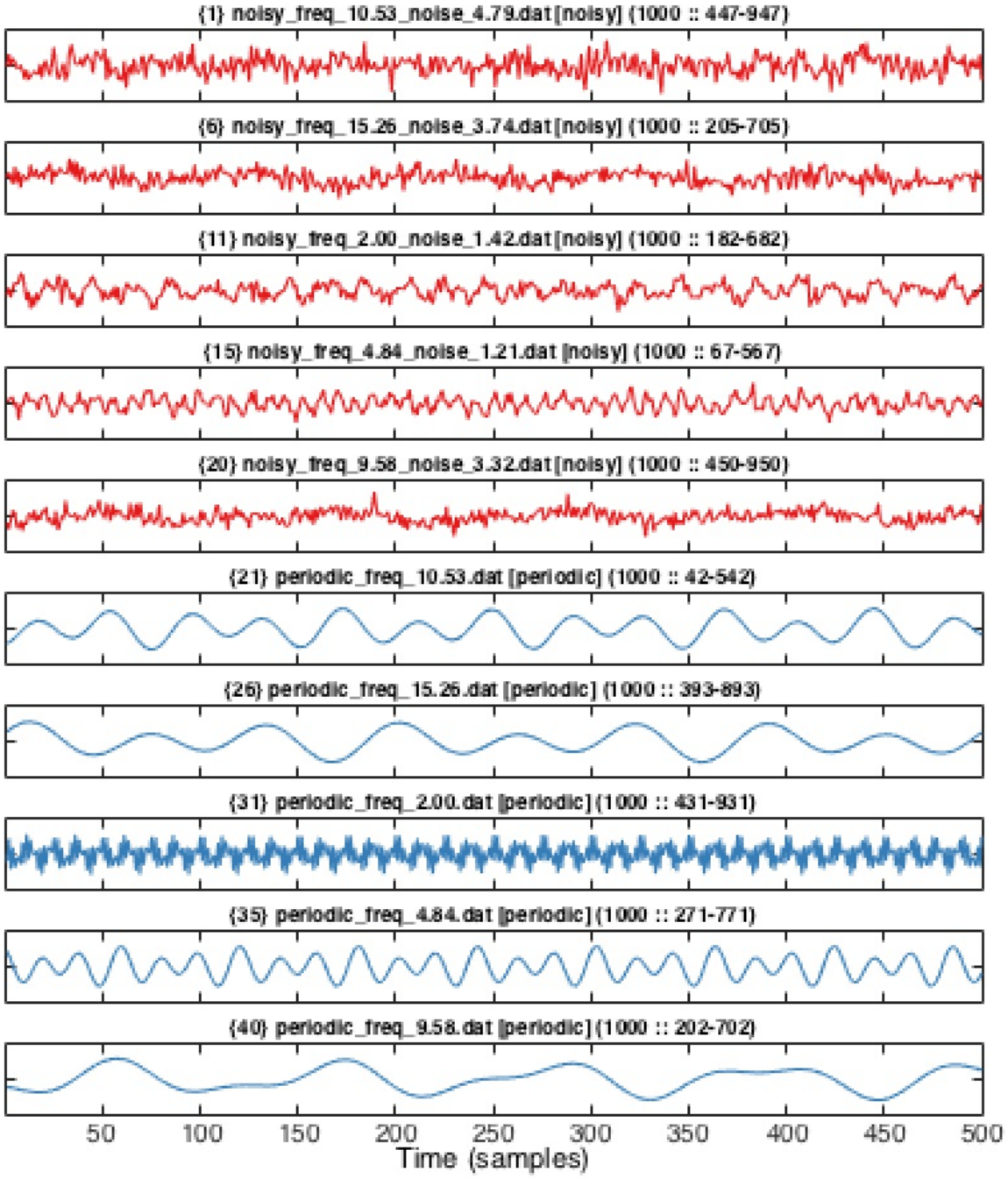


producing an overview picture of the first 300 samples of 40 time series (spaced through the rows of the data matrix).

Dealing with groups of time series

When the time series have been assigned groups (using `TS_LabelGroups`, here), this information is automatically incorporated into `TS_plot_timeseries`, which then plots a given number of each time series group, and colors them accordingly:

```
numPerGroup = 5; % plot this many examples of each group of time series
plotHow = 'grouped'; % plot examples of each assigned group of time series
TS_plot_timeseries('norm',numPerGroup,plotHow,500);
```



In this case the two labeled groups of time series are recognized by the function: red (noisy), blue (no noise), and then 5 time series in each group are plotted, showing the first 500 samples of each time series.

Low dimensional representations

The software also provides a basic means of visualizing low-dimensional representations of the data, using PCA as `TS_plot_pca`.

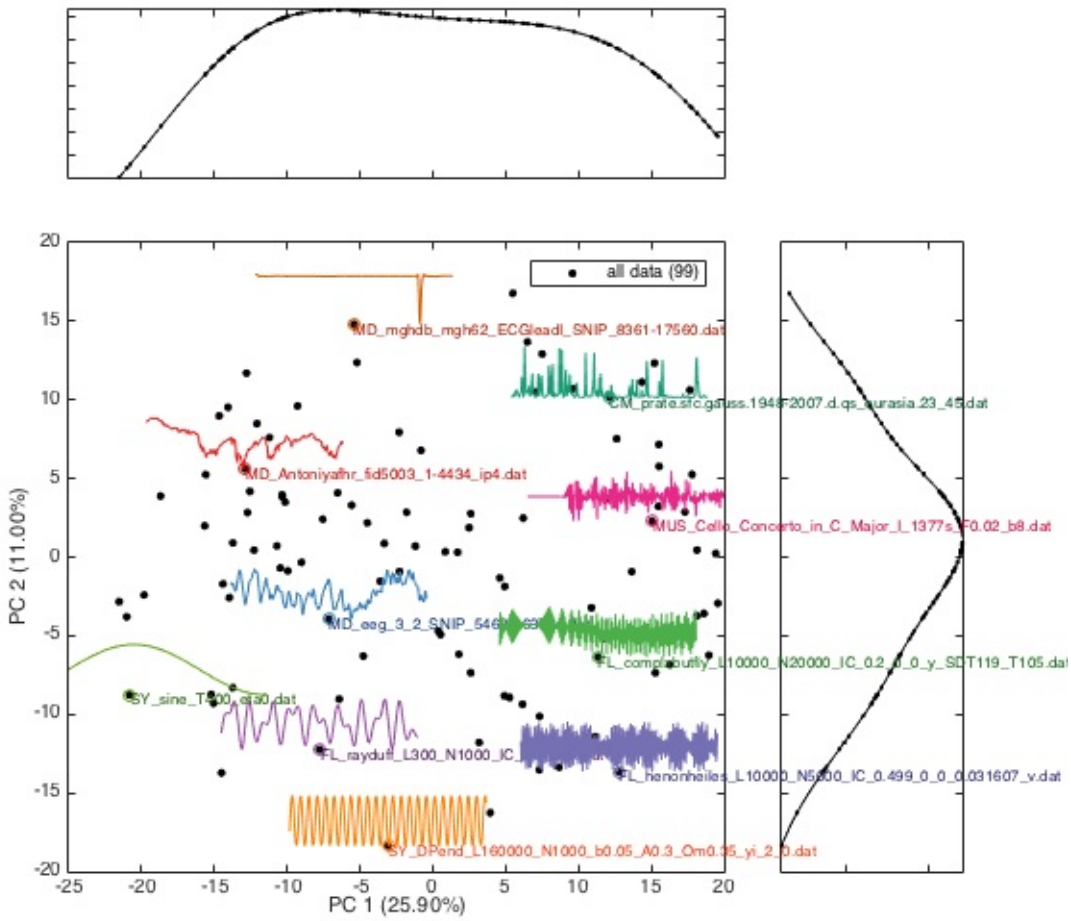
This can be done for a time-series dataset as follows:

```
TS_plot_pca('norm','ts');
```

This uses the normalized data (specifying `'norm'`), plotting time series (specifying `'ts'`) in the reduced, two-dimensional principal components space of operations (the leading two principal components of the data matrix).

By default, the user will be prompted to select 10 points on the plot to annotate with their corresponding time series, which are annotated as the first 300 points of that time series (and their names by default).

After selecting 10 points, we have the following:



The proportion of variance explained by each principal component is provided in parentheses in the axis label.

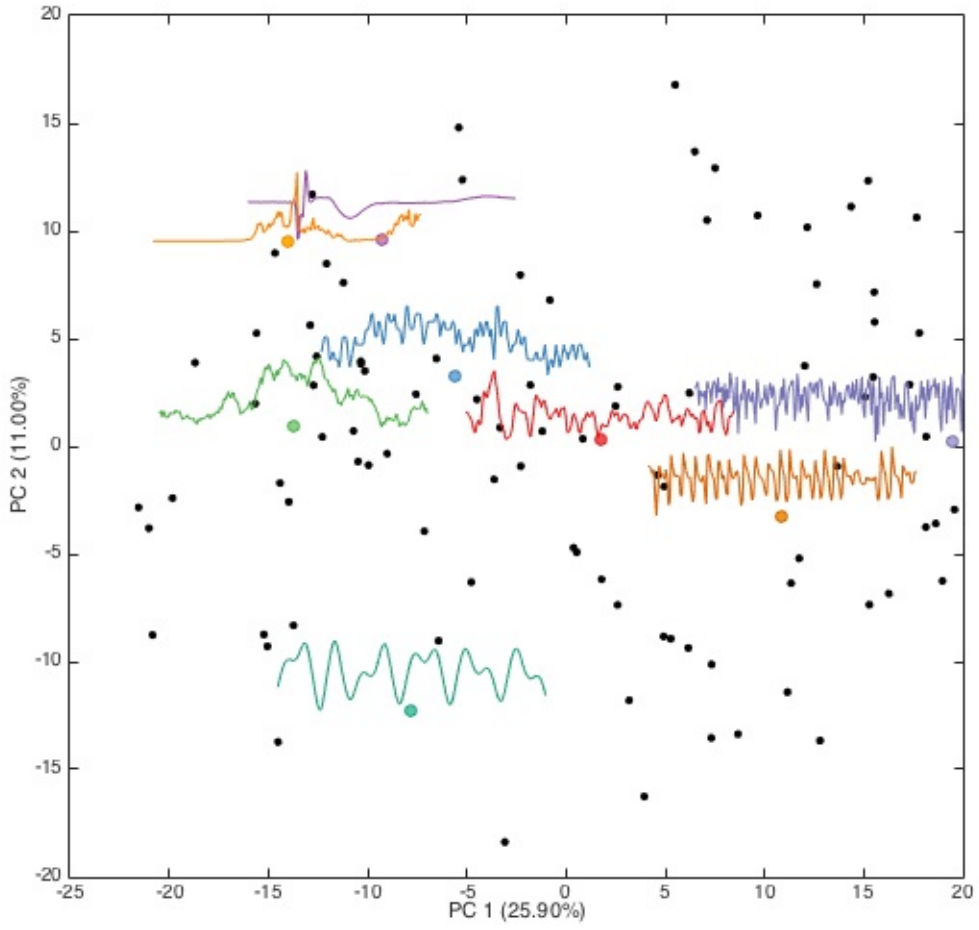
Customizing annotations

Annotation properties can be altered with some detail by specifying properties as the `annotateParams` input variable, for example:

```
% First set up the annotateParams structure:
annotateParams = struct;
annotateParams.n = 8; % annotate 8 time series
annotateParams.maxL = 150; % annotates the first 150 samples of time series
annotateParams.userInput = 0; % points not selected by user but allocated randomly
annotateParams.textAnnotation = 0; % don't display names of annotated time series
showDistributions = 0; % don't plot marginal distributions

% Then generate a plot using these settings:
TS_plot_pca('norm','ts',showDistributions,'',annotateParams)
```

yields:



Plotting grouped time series

If groups of time series have been specified (using `TS_LabelGroups`), then these are automatically recognized by `TS_plot_pca`, which will then distinguish the labeled groups in the resulting 2-dimensional annotated time-series plot.

Consider the sample dataset containing 20 periodic signals with additive noise (given the keyword **noisy** in the database), and 20 purely periodic signals (given the keyword **periodic** in the database). After retrieving and normalizing the data, we store the two groups in the metadata for the normalized dataset **HCTSA_N.mat**:

```
TS_LabelGroups('norm',{'noisy','periodic'},'ts');
```

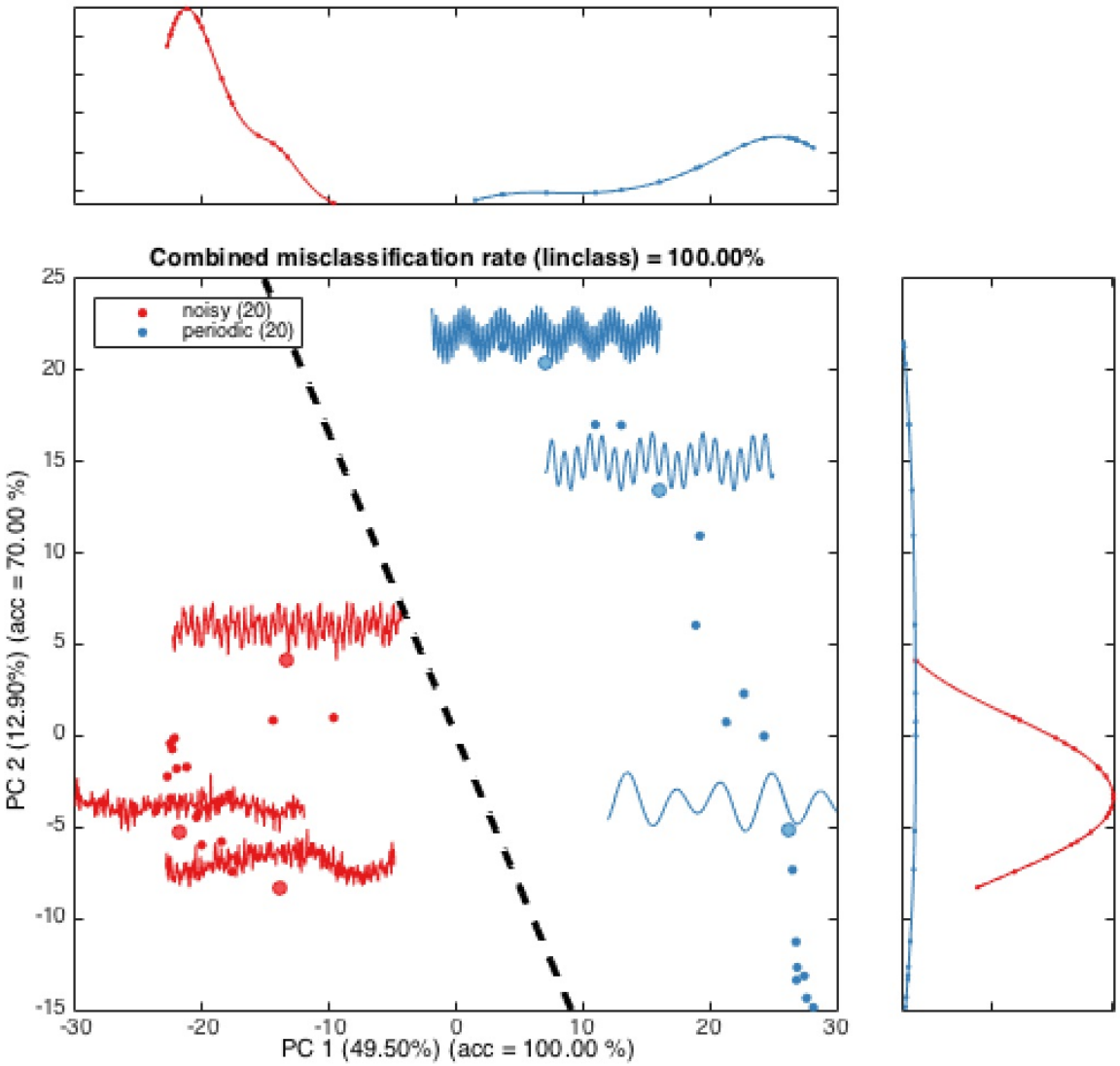
```
We found:
noisy -- 20 matches
periodic -- 20 matches
Saving group labels and information back to HCTSA_N.mat... Saved.
```

Now when we plot the dataset in `TS_plot_pca`, it will automatically distinguish the groups in the plot and attempt to classify the difference in the reduced principal components space.

Running the following:

```
>> annotateParams = struct('n',6); % annotate 6 time series
>> showDistribution = 1; % plot marginal distributions
>> TS_plot_pca('norm','ts',showDistribution,'',annotateParams);
```

The function then directs you to select 6 points to annotate time series to, producing the following:



Notice how the two labeled groups have been distinguished as red and blue points, and a linear classification boundary has been added (with in-sample misclassification rate annotated to the title and to each individual principal component). If marginal distributions are plotted (setting `showDistribution = 1` above), they are labeled according to the same colors.

Visualizing the nearest neighbors of a time series using `TS_SimSearch`

While the global structure of a time-series dataset can be investigated by plotting the data matrix (`TS_plot_DataMatrix`) or a low-dimensional representation of it (`TS_plot_pca`), sometimes it can be more interesting to retrieve and visualize relationships between a set of nearest neighbors to a particular time series of interest.

The *hctsa* framework provides a way to easily compute distances between pairs of time series, e.g., as a Euclidean distance between their normalized feature vectors. This allows very different time series (in terms of their origin, their method of recording and measurement, and their number of samples) to be compared straightforwardly according to their properties, measured by the algorithms in our *hctsa* library.

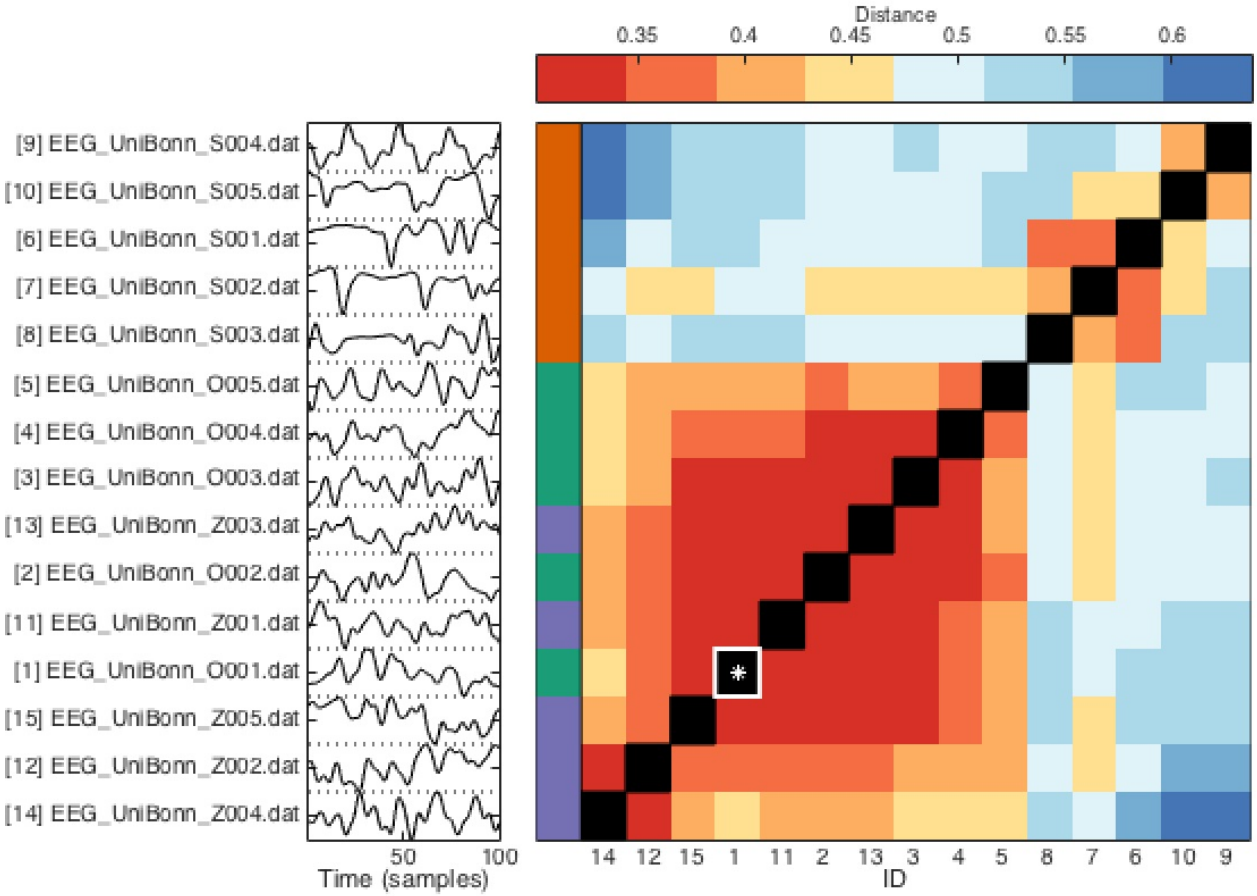
For this, we use the `TS_SimSearch` function, specifying the id of the time series of interest (i.e., the `ID` field of the `TimeSeries` structure) with the first input and the number of neighbors with the 'numNeighbors' input specifier (default: 20). By default, data is loaded from `HCTSA_N.loc` , but a custom source can be specified using the `'whatDataFile'` input specifier (e.g., `TS_SimSearch('whatDataFile','HCTSA_custom.mat')`).

After specifying the target and how many neighbors to retrieve, `TS_SimSearch` outputs the list of neighbors and their distances to screen, and the function also provides a range of plotting options to visualize the neighbors. The plots to produce are specified as a cell using the 'whatPlots' input.

Pairwise similarity matrix of neighbors

```
TS_SimSearch('whatPlots',{'matrix'});
```

To investigate the pairwise relationships between all neighbors retrieved, you specify the `'matrix'` option of the `TS_SimSearch` function. An example output using a publicly-available EEG dataset, retrieving 14 neighbors from the time series with ID = 1, as `TS_SimSearch(1,'whatPlots',{'matrix'},'numNeighbors',14)` , is shown below:



The specified target time series (`ID = 1`) is shown as a white star, and all 14 neighbors are shown, as labeled on the left of the plot with their respective IDs, and a 100-sample subset of their time traces.

Pairwise distances are computed between all pairs of time series (as a Euclidean distance between their feature vectors), and plotted using color, from low (red = more similar pairs of time series) to high (blue = more different pairs of time series).

Because this dataset contains 3 classes that were previously labeled (using `TS_LabelGroups` as: `TS_LabelGroups({'seizure','eyesOpen','eyesClosed'})`), the function shows these class assignments using color labels to the left of the plot (purple, green, and orange in this case).

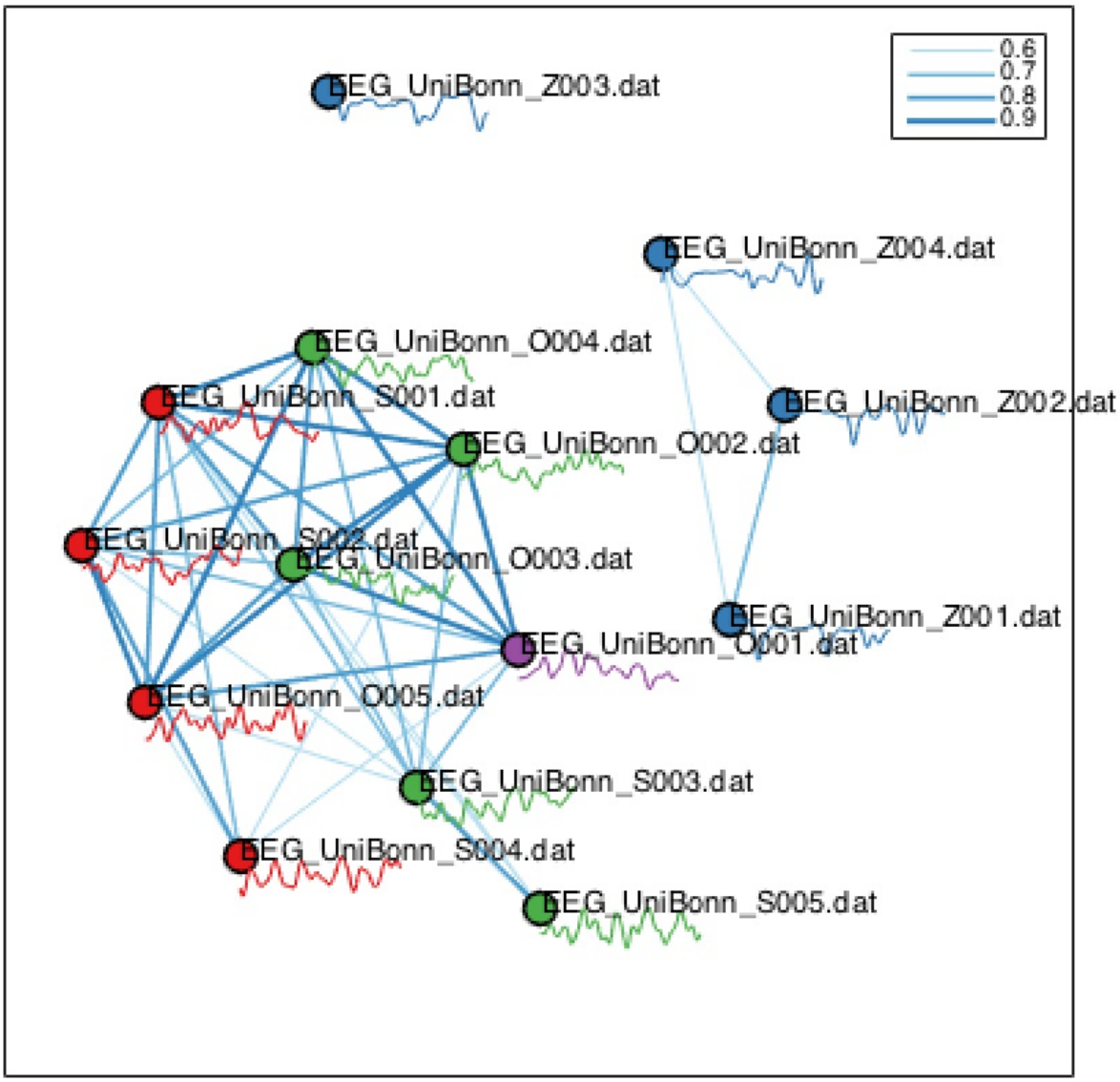
In this case we see that the purple and green classes are relatively similar under this distance metric (eyes open and eyes closed), whereas the orange time series (seizure) are distinguished.

Network of neighbors

Another way to visualize the similarity (under our feature-based distance metric) of all pairs of neighbors is using a network visualization. This is specified as:

```
TS_SimSearch(1,'whatPlots',{'network'});
```

which produces something like the following:



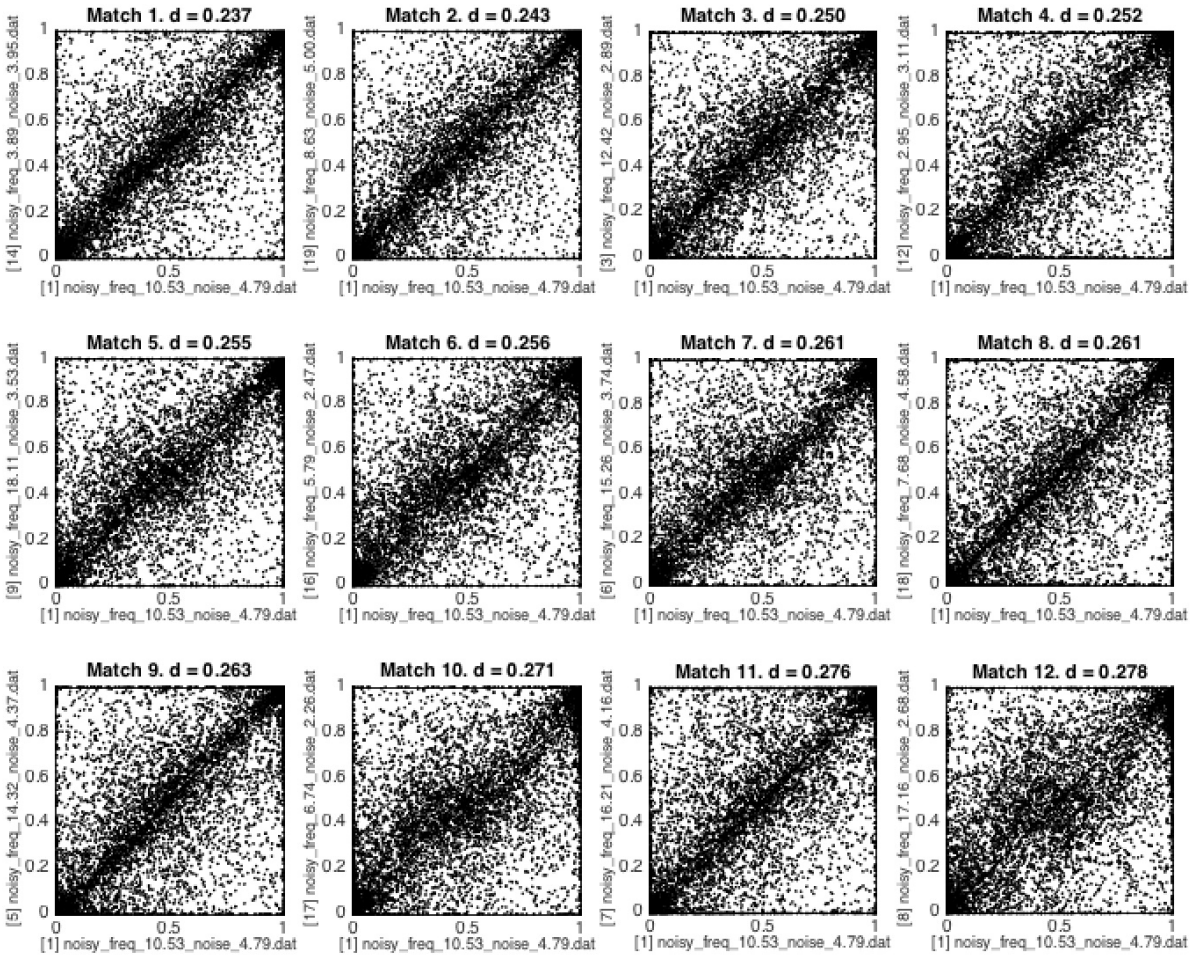
The strongest links are visualized as blue lines (by default, the top 40% of strongest links are plotted, cf. the legend showing 0.9, 0.8, 0.7, and 0.6 for the top 10%, 20%, 30%, and 40% of links, respectivly).

The target is distinguished (as purple in this case), and the other classes of time series are shown using color, with names and time-series segments annotated. Again, you can see that the EEG time series during seizure (blue) are distinguished from eyes open (red) and eyes closed (green).

Pairwise similarity matrix of neighbors

```
TS_SimSearch(1,'whatPlots',{'scatter'});
```

The scatter setting visualizes the relationship between the target and each of 12 time series with the most similar properties to the target. Each subplot is a scatter of the (normalized) outputs of each feature for the specified target (x-axis) and the match (y-axis). An example is shown below.



Other details

Multiple output plots can be produced simultaneously by specifying many types of plots as follows:

```
TS_SimSearch(1,'whatPlots',{'matrix','network','scatter'})
```

This produces a plot of each type.

Note that pairwise distances can be pre-computed and saved in the `HCTSA*.mat` file using `TS_PairwiseDist` for custom distance metrics (which is done by default in `TS_cluster` for datasets containing fewer than 1000 objects). `TS_SimSearch` checks for this information in the specified input data (containing the `ts_clust` or `op_clust` structure), and uses it to retrieve neighbors. If distances have not previously been computed, distances from the target are computed as euclidean distances (time series) or absolute correlation distances (operations) between feature vectors within `TS_SimSearch`.

Investigating specific features using `TS_FeatureSummary`

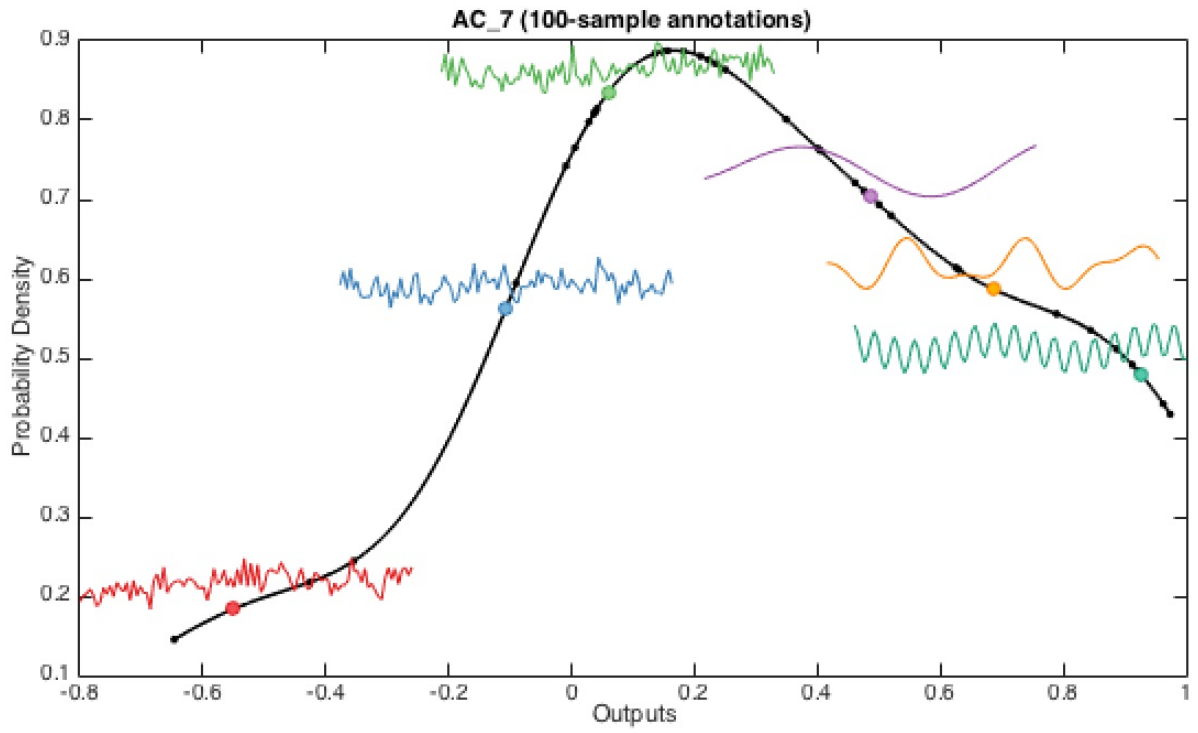
Sometimes it's useful to be able to investigate the behavior of an individual operation (or feature) across a time-series dataset.
What are the distribution of outputs, and what types of time series receive low values, and what receive high values?

These types of simple questions for specific features of interest can be investigated using the `TS_FeatureSummary` function.
The function takes in an operation ID as it's input (and can also take inputs specifying a custom data source, or custom annotation parameters), and produces a distribution of outputs from that operation across the dataset, with the ability to then annotate time series onto that plot.

For example, the following:

```
TS_FeatureSummary(100)
```

Produces the following plot (where 6 points on the distribution have been clicked on to annotate them with short time-series segments):

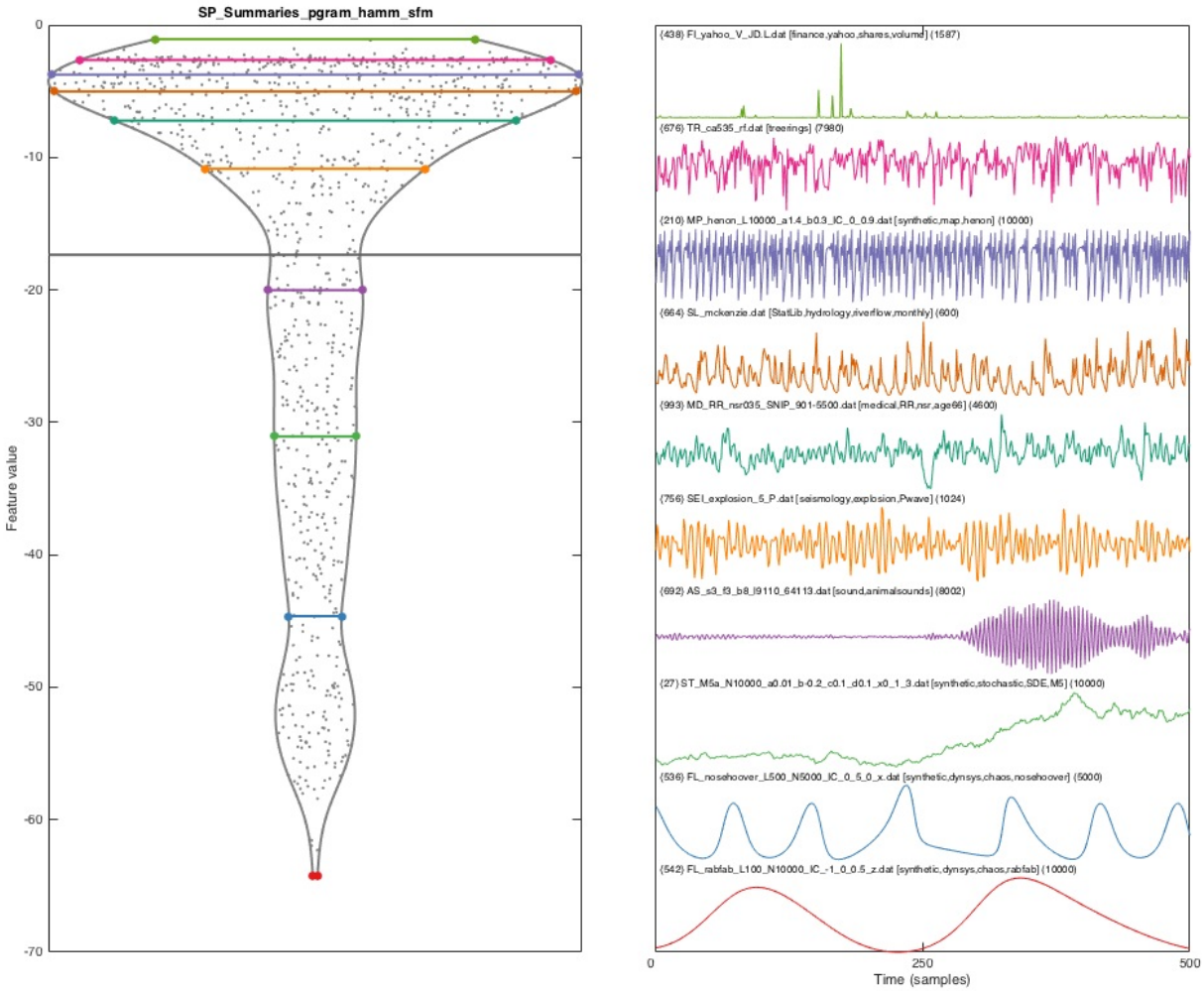


You can visually see that time series with more autocorrelated patterns through time receive higher values from this operation.
Because no group information is present in this dataset, the time series are colored at random.

Running TS_FeatureSummary in violin plot mode provides another representation of the same result:

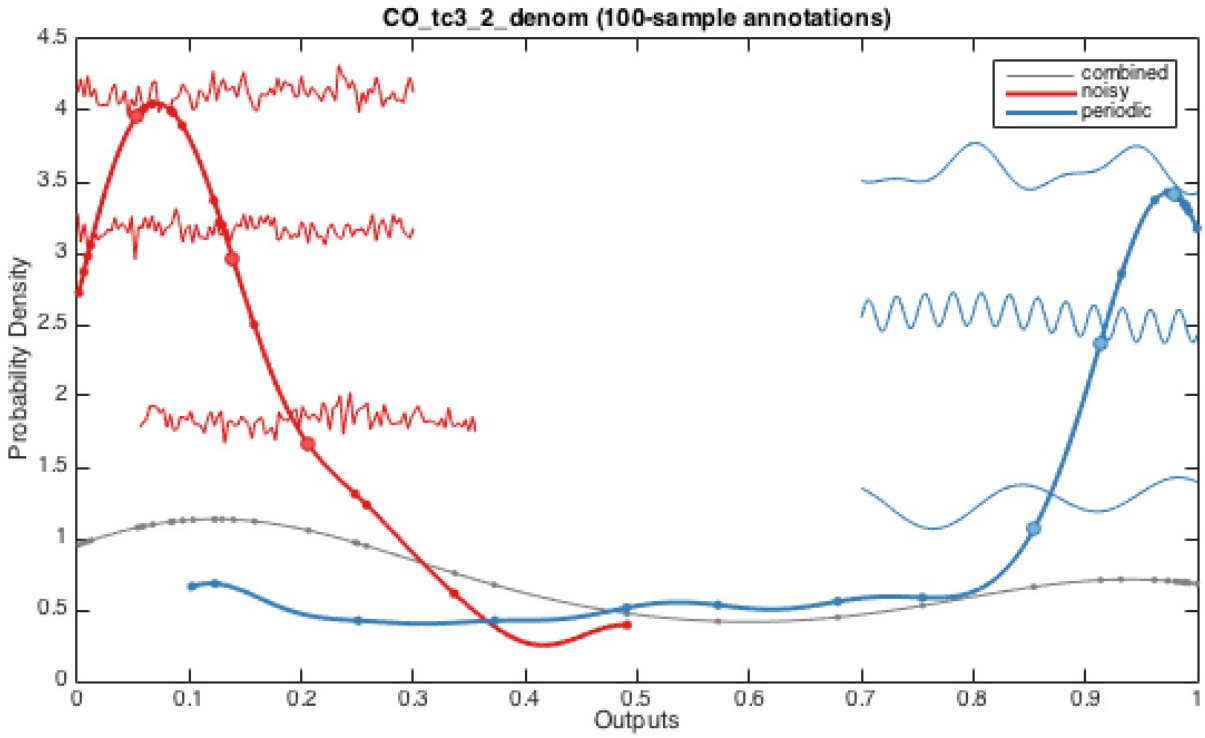
```
annotateParams = struct('maxL',500);
TS_FeatureSummary(4310, 'raw', 1, annotateParams);
```

This plots the distribution of feature 4310 from `HCTSA.mat` as a violin plot, with ten 500-point time series subsegments annotated at different points through the distribution, shown to the right of the plot:



Plotting for labeled groups of time series

When time series groups have been labeled (using `TS_LabelGroups` as: `TS_LabelGroups({'seizure','eyesOpen','eyesClosed'},'raw');`), `TS_FeatureSummary` will plot the distribution for each class separately, as well as an overall distribution.
Annotated points can then be added to each class-specific distributions.
In the example shown below, we can see that the 'noisy' class (red) has low values for this feature (`CO_tc3_2_denom`), whereas the 'periodic' class mostly has high values.

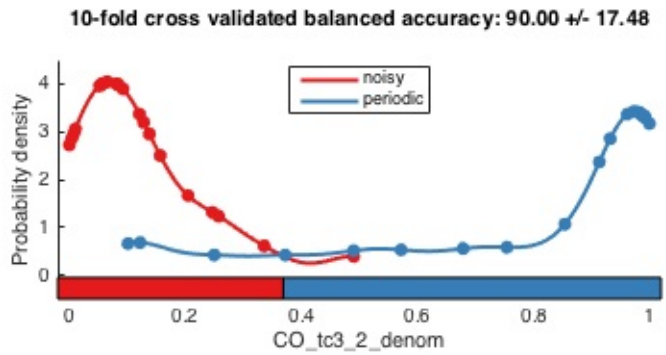


Simpler distributions

`TS_SingleFeature` provides a simpler way of seeing the class distributions without annotations, as either kernel-smoothed distributions, as in `TS_FeatureSummary` , or as violin plots.
See below for example implementations:

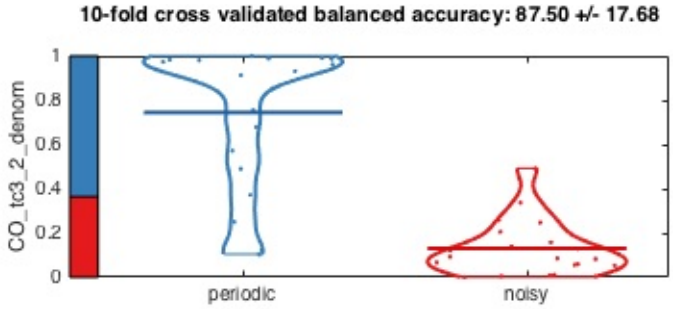
```
opid = 500; makeViolin = false;
TS_SingleFeature('raw',opid,makeViolin);
```

Shows the distributions with classification bar underneath (for where a linear classifier would classify different parts of the space as either noisy or periodic):




```
opid = 500; makeViolin = true;
TS_SingleFeature('raw',opid,makeViolin);
```

Shows the distributions shown as a violin plot, with means annotated and classification bar to the left:



Note that the title, which gives an indication of the 10-fold cross-validated balanced accuracy of a linear classifier in the space is done on the basis of a single 10-fold split and is stochastic.
Thus, as shown above, this can yield slightly different results when repeated.
For a more rigorous analysis than this simple indication, the procedure should be repeated many more times to give a converged estimate of the balanced classification accuracy.

Exploring classification rate using `TS_classify`

When performing a time-series classification task, a basic first exploration of the data is to investigate how accurately a classifier can performing using all of the features computed in your *hctsa* analysis. This can be done by running `TS_classify`. The function requires that group labels be assigned to the time series, using `TS_LabelGroups`. For example, running the function on a dataset with three classes of five time series each, as `TS_classify('norm')`, produces:

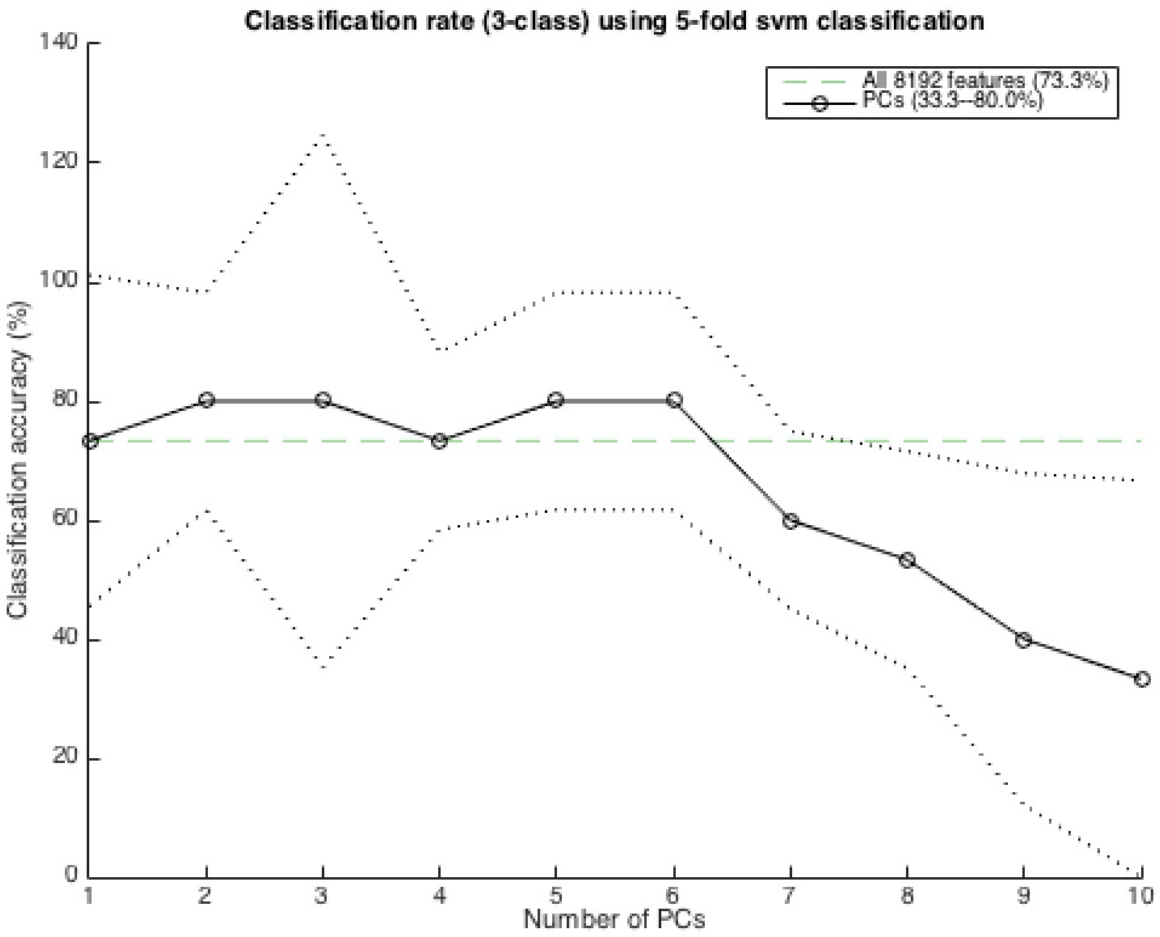
```
Classification rate (3-class) using 5-fold svm classification with 8192 features:
73.333 +/- 14.907%
```

In this case, the function has attempted to learn a linear svm classifier on the features to predict the labels assigned to the data, using 5-fold cross validation (note that the default is 10-fold, but for smaller datasets such as this one, fewer folds are used automatically). The results show that using 8192 features we obtain a mean classification accuracy of 73.3% (with a standard deviation over the 5-folds of 14.9%).

If the 3rd input to the function is set to 1 (or fewer than 3 inputs are provided), the function then computes the top 10 PCs of the data matrix, and uses them to classify the time series, yielding:

```
Computing top 10 PCs... Done.
Computing classification rates keeping top 1--10 PCs...
1 PCs:   73.333 +/- 27.889%
2 PCs:   80.000 +/- 18.257%
3 PCs:   80.000 +/- 44.721%
4 PCs:   73.333 +/- 14.907%
5 PCs:   80.000 +/- 18.257%
6 PCs:   80.000 +/- 18.257%
7 PCs:   60.000 +/- 14.907%
8 PCs:   53.333 +/- 18.257%
9 PCs:   40.000 +/- 27.889%
10 PCs:   33.333 +/- 33.333%
```

The plot shows this information graphically:



where the classification accuracy is shown for all features (green, dashed), and as a function of the number of leading PCs included in the classifier (black circles).

Because we have so few examples of time series in this case (5 time series from each of 3 classes), attempting to learn a classifier for the dataset using thousands of features is overkill -- we have no where near enough data to constrain such a classifier. Indeed, these results show that a classifier using just a single feature (the first PC of the data matrix) reproduces the accuracy of a classifier the full set of 8192 features (both achieving 73.3% on this small dataset).

Determining which features are informative of the class differences using `TS_TopFeatures`

In a time-series classification task, you are often interested in not only determining differences between the labeled classes, but also interpreting them in the context of your application. For example, an entropy estimate may give low values to the RR interval series measured from a healthy population, but higher values to those measured from patients with congestive heart failure. When performing a highly comparative time-series analysis, we have computed a large number of time-series features and can use these results to arrive at these types of interpretable conclusions automatically.

A simple way to determine which individual features are useful for distinguishing the labeled classes of your dataset is to compare each feature individually in terms of its ability to separate the labeled classes of time series. This can be achieved using the `TS_TopFeatures` function.

Example: Classification of seizure from EEG

In this example, we consider 100 EEG time series from seizure, and 100 EEG time series from eyes open (from the publicly-available Bonn University dataset). After running `TS_LabelGroups` (as simply `TS_LabelGroups()`, which automatically detects the two keywords assigned to the two classes of signals), we run simply:

```
TS_TopFeatures()
```

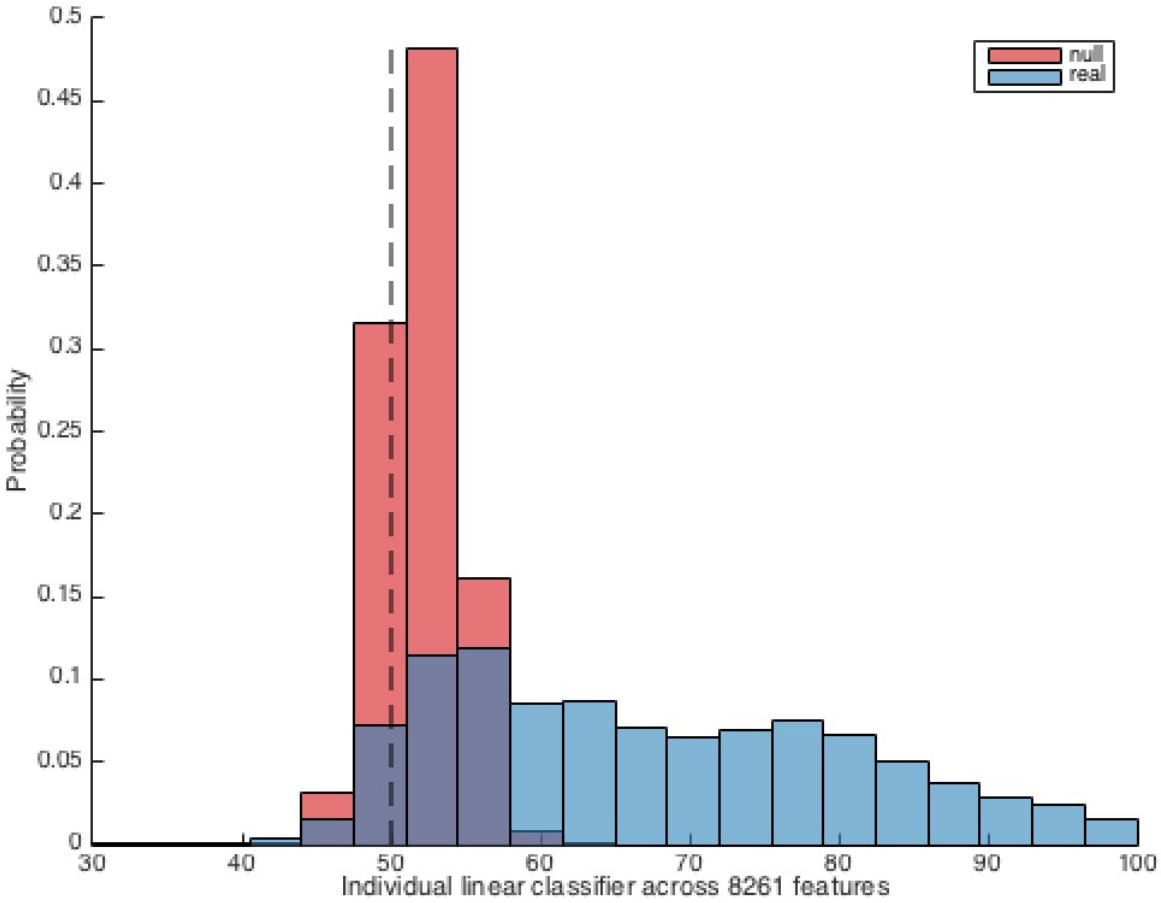
By default this function will compare the in-sample linear classification performance of all operations in separating the labeled classes of the dataset (individually), produce plots summarizing the performance across the full library (compared to an empirical null distribution), characterize the top-performing features, and show their dependencies on the dataset. Inputs to the function control how these tasks are performed (including the statistic used to measure individual performance, what plots to produce, and whether to produce nulls).

First, we get an output to screen, showing the mean linear classification accuracy across all operations, and a list of the operations with top performance (ordered by their test statistic, with their ID shown in square brackets, and keywords in parentheses):

```
Mean linear classifier across 8261 operations = 67.02%
(Random guessing for 2 equiprobable classes = 50.00%)
[908] EN_DistributionEntropy_raw_ks__005 (entropy,raw,spreaddep) -- 100.00%
[982] DN_FitKernelSmoothraw_max (distribution,ksdensity,raw,spreaddep) -- 100.00%
[902] EN_DistributionEntropy_raw_ks_01_0 (entropy,raw,spreaddep) -- 99.50%
[903] EN_DistributionEntropy_raw_ks_02_0 (entropy,raw,spreaddep) -- 99.50%
[904] EN_DistributionEntropy_raw_ks_05_0 (entropy,raw,spreaddep) -- 99.50%
[905] EN_DistributionEntropy_raw_ks_1_0 (entropy,raw,spreaddep) -- 99.50%
[906] EN_DistributionEntropy_raw_ks__001 (entropy,raw,spreaddep) -- 99.50%
[907] EN_DistributionEntropy_raw_ks__002 (entropy,raw,spreaddep) -- 99.50%
[909] EN_DistributionEntropy_raw_ks__01 (entropy,raw,spreaddep) -- 99.50%
[1127] EN_MS_LZcomplexity_8_diff (MichaelSmall,complexity,mex,LempelZiv) -- 99.50%
[1128] EN_MS_LZcomplexity_9_diff (MichaelSmall,complexity,mex,LempelZiv) -- 99.50%
[3412] DN_CompareKSFit_uni_psy (distribution,ksdensity,uni,peaksepy,raw,locdep) -- 99.
50%
[901] EN_DistributionEntropy_raw_ks_005_0 (entropy,raw,spreaddep) -- 99.00%
[1129] EN_MS_LZcomplexity_10_diff (MichaelSmall,complexity,mex,LempelZiv) -- 99.00%
[2958] EN_SampEn_5_01_diff1_sampen4 (entropy,sampen,controlen) -- 99.00%
[910] EN_DistributionEntropy_raw_ks__02 (entropy,raw,spreaddep) -- 98.50%
[980] DN_FitKernelSmoothraw_entropy (distribution,ksdensity,entropy,raw,spreaddep) --
98.50%
[2303] SY_SpreadRandomLocal_ac2_100_meansampen1_015 (stationarity) -- 98.50%
[2616] FC_Surprise_T1_100_4_udq_500_lq (information,symbolic) -- 98.50%
[2624] FC_Surprise_T1_100_5_udq_500_lq (information,symbolic) -- 98.50%
```

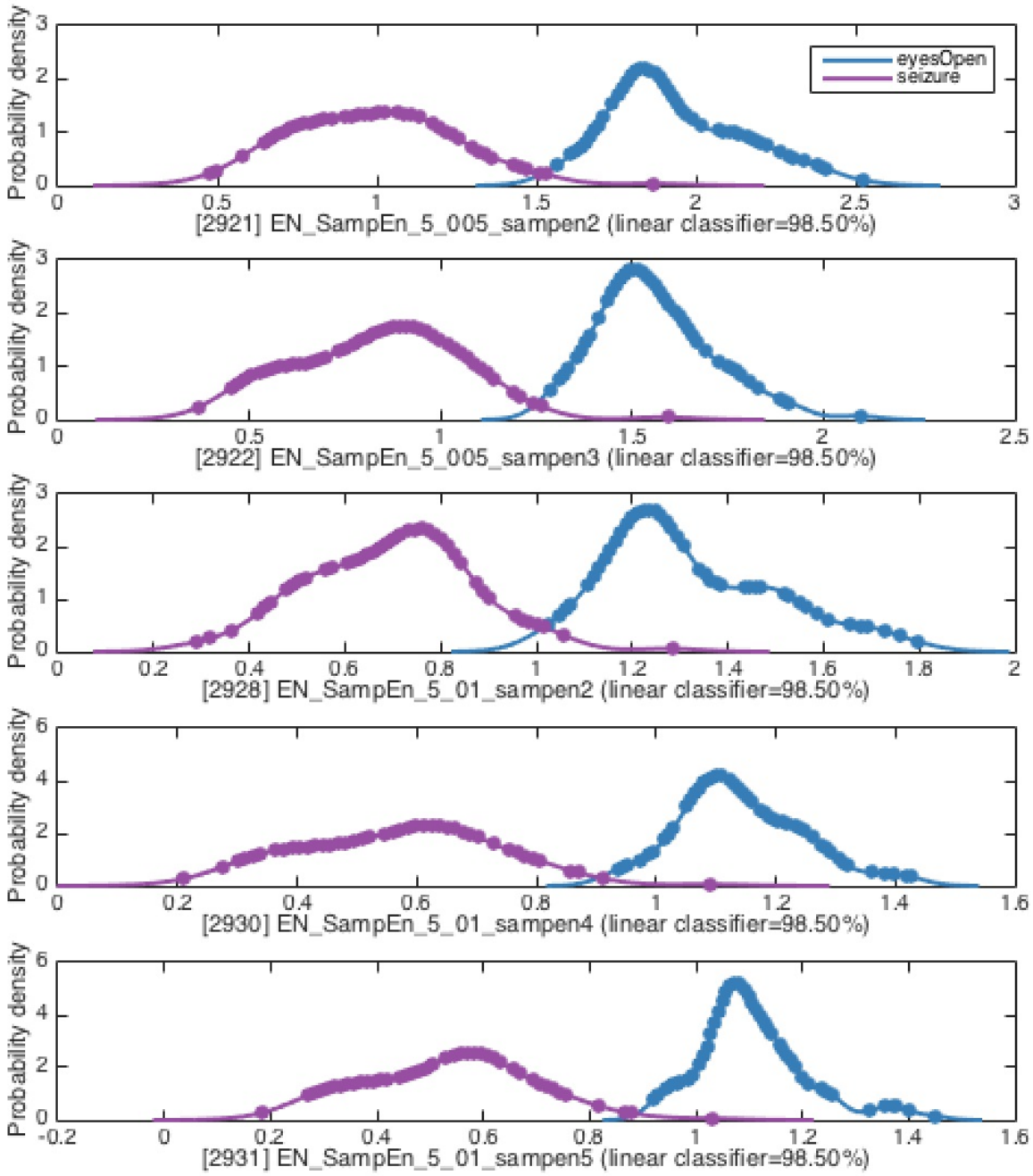
Here we see that measures of entropy are dominating this top list, including measures of the time-series distribution.

An accompanying plot summarizes the performance of the full library on the dataset, compared to a set of nulls (generated by shuffling the class labels randomly):



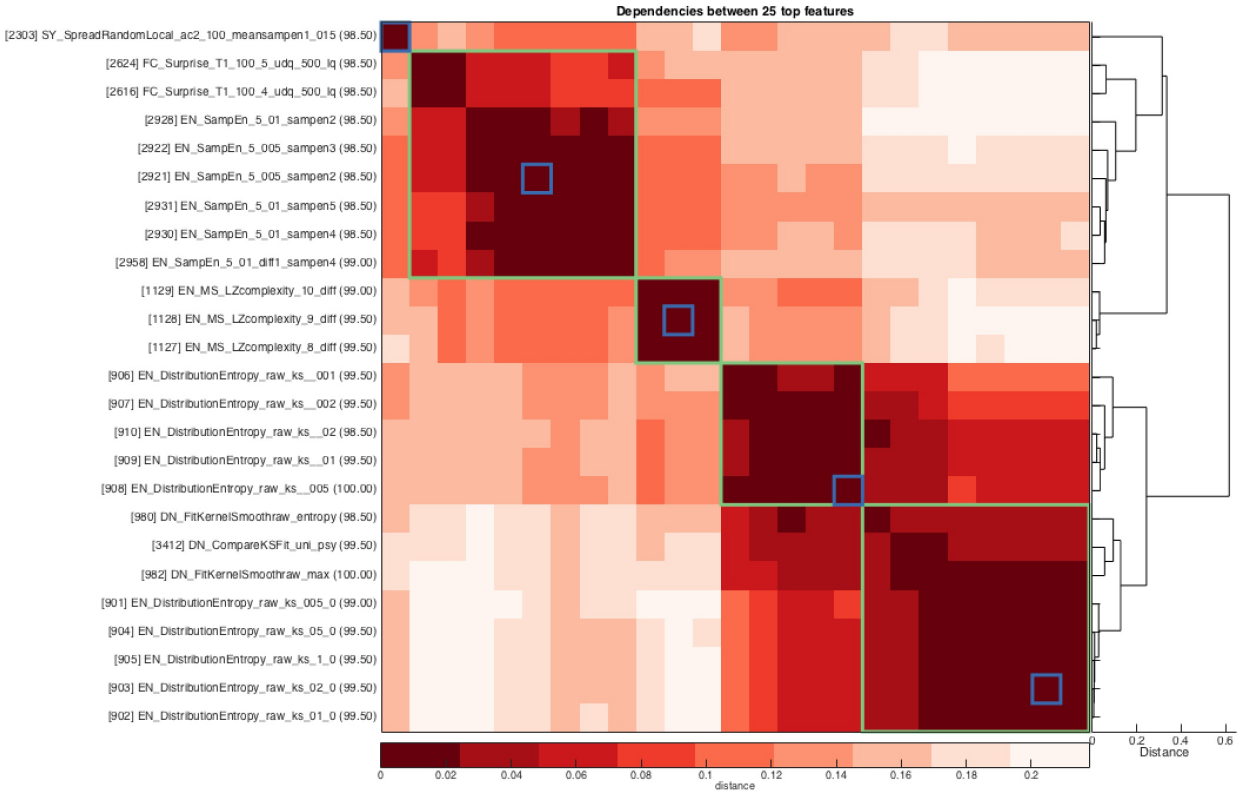
Here we can see that the default feature library (blue: 7275 features remaining after filtering and normalizing) are performing much better than the randomized features (null: red).

Next we get a set of plots showing the class probability distributions for the top features. For example:



This allows us to interpret the values of features in terms of the dataset. After some inspection of the listed operations, we find that various measures of EEG Sample Entropy are higher for healthy individuals with eyes open (blue) than for seizure signals (purple).

Finally, we can visualize how the top operations depend on each other, to try to deduce the main independent behaviors in this set of 25 top features for the given classification task:



In this plot, the magnitude of the linear correlation, `R` (as `1-|R|`) between all pairs of top operations is shown using color, and a visualization of a clustering into 5 groups is shown (with green boxes corresponding to clusters of operations based on the dendrogram shown to the right of the plot). Blue boxes indicate a representative operation of each cluster (closest to the cluster centre).

Here we see that most pairs of operations are quite dependent (at `|R| > ~0.8`), and the main clusters are those measuring distribution entropy (`EN_DistributionEntropy_`), Lempel-Ziv complexity (`EN_MS_LZcomplexity_`), Sample Entropy (`EN_SampEn_`), and a one-step-ahead surprise metric (`FC_Surprise_`).

This function produces all of the basic visualizations required to achieve a basic understanding of the differences between the labeled classes of the dataset, and should be considered a first step in a more detailed analysis. For example, in this case we may investigate the top-performing features in more detail, to understand in detail what they are measuring, and how. This process is described in the Interpreting features section.

Interpreting *hctsa* features

Often, an *hctsa* analysis will yield a list of features that may be particularly relevant to a given time-series analysis task, e.g., features that robustly distinguish time series recorded from individuals with some disease diagnosis compared to that of healthy controls. In cases like this, the next step for the analyst is to bridge the automated feature selection enabled by *hctsa* to domain understanding. Consider a problem in which `TS_TopFeatures` is run to find features that accurately distinguish groups of time series, yielding a list of features like the following:

```
[3016] FC_LocalSimple_mean3_taures (forecasting) -- 59.97%
[3067] FC_LocalSimple_median3_taures (forecasting) -- 58.14%
[2748] EN_mse_1-10_2_015_sampen_s3 (entropy,sampen,mse) -- 54.10%
[7338] MF_armax_2_2_05_1_AR_1 (model) -- 53.71%
[7339] MF_armax_2_2_05_1_AR_2 (model) -- 53.31%
[3185] DN_CompareKSFit_uni_psx (distribution,ksdensity,raw,locdep) -- 52.14%
[6912] MF_steps_ahead_ar_best_6_ac1_3 (model,prediction,arfit) -- 52.11%
[6564] WL_coeffs_db3_4_med_coeff (wavelet) -- 52.01%
[4552] SP_Summaries_fft_logdev_fpoly2csS_p1 (spectral) -- 51.57%
[6634] WL_dwtcoeff_sym2_5_noisestd_l5 (wavelet,dwt) -- 51.48%
[930] DN_FitKernelSmoothraw_entropy (distribution,ksdensity,entropy,raw,spreaddep) --
51.37%
[6574] WL_coeffs_db3_5_med_coeff (wavelet) -- 51.26%
[6630] WL_dwtcoeff_sym2_5_noisestd_l4 (wavelet,dwt) -- 51.04%
[1891] CO_StickAngles_y_ac2_all (correlation) -- 50.85%
[16] rms (distribution,location,raw,locdep,spreaddep) -- 50.83%
[6965] MF_steps_ahead_arma_3_1_6_rmserr_6 (model,prediction) -- 50.83%
[2747] EN_mse_1-10_2_015_sampen_s2 (entropy,sampen,mse) -- 50.35%
[4201] SC_FluctAnal_mag_2_dfa_50_2_logi_ssr (scaling) -- 50.34%
[6946] MF_steps_ahead_ss_best_6_meandiffrms (model,prediction) -- 50.33%
```

Functions like `TS_TopFeatures` are helpful in showing us how these different types of features might cluster into groups that measure similar properties (as shown in the previous section). This helps us to be able to inspect sets of similar, inter-correlated features together as a group, but even when we have isolated such a group, how can we start to interpret and understand what these features are actually measuring? Some features in the list may be easy to interpret directly (e.g., `rms` in the list above is simply the root-mean-square of the distribution of time-series values), and others have clues in the name (e.g., features starting with `WL_coeffs` are to do with measuring wavelet coefficients, features starting with `EN_mse` correspond to measuring the multiscale entropy, mse, and features starting with `FC_LocalSimple_mean` are related to time-series forecasting using local means of the time series). Below we outline a procedure for how a user can go from a time-series feature

selected by *hctsa* towards a deeper understanding of the type of algorithm that feature is derived from, how that algorithm performs across the dataset, and thus how it can provide interpretable information about your specific time-series dataset.

Inspecting keywords

The simplest way of interpreting what sort of property a feature might be measuring is from its keywords, that often label individual features by the class of time-series analysis method from which they were derived. In the list above, we see keywords listed in parentheses, as *'forecasting'* (methods related to predicting future values of a time series), *'entropy'* (methods related to predictability and information content in a time series), and *'wavelet'* (features derived from wavelet transforms of the time series). There are also keywords like *'locdep'* (location dependent: features that change under mean shifts of a time series), *'spreaddep'* (spread dependent: features that change under rescaling about their mean), and *'lengthdep'* (length dependent: features that are highly sensitive to the length of a time series).

Inspecting code

To find more specific detailed information about a feature, beyond just a broad categorical label of the literature from which it was derived, the next step is find and inspect the code file that generates the feature of interest. For example, say we were interested in the top performing feature in the list above:

```
    [3016] FC_LocalSimple_mean3_taures (forecasting) -- 59.97%
```

We know from the keyword that this feature has something to do with forecasting, and the name provides clues about the details (e.g., `FC_` stands for forecasting, the function `FC_LocalSimple` is the one that produces this feature, which, as the name suggests, performs simple local time-series prediction). We can use the feature ID (3016) provided in square brackets to get information from the `Operations` structure array:

```
>> disp(Operations([Operations.ID]==3016));
           ID: 3016
         Name: 'FC_LocalSimple_mean3_taures'
     Keywords: 'forecasting'
   CodeString: 'FC_LocalSimple_mean3.taures'
     MasterID: 836
```

Inspecting the text before the dot, '.', in the `CodeString` field (`FC_LocalSimple_mean3`) tells us the name that *hctsa* uses to describe the Matlab function and its unique set of inputs that produces this feature. Whereas the text following the dot, '.', in the `CodeString` field

(`taures`), tells us the field of the output structure produced by the Matlab function that was run. We can use the `MasterID` to get more information about the code that was run using the `MasterOperations` structure array:

```
>> disp(MasterOperations([MasterOperations.ID]==836));
       ID: 836
    Label: 'FC_LocalSimple_mean3'
     Code: 'FC_LocalSimple(y,'mean',3)'
```

This tells us that the code used to produce our feature was `FC_LocalSimple(y,'mean',3)` . We can get information about this function in the commandline by running a `help` command:

```
>> help FC_LocalSimple
  FC_LocalSimple    Simple local time-series forecasting.

  Simple predictors using the past trainLength values of the time series to
  predict its next value.

 ---INPUTS:
  y, the input time series

  forecastMeth, the forecasting method:
           (i) 'mean': local mean prediction using the past trainLength time-series
                       values,
           (ii) 'median': local median prediction using the past trainLength
                         time-series values
           (iii) 'lfit': local linear prediction using the past trainLength
                         time-series values.

  trainLength, the number of time-series values to use to forecast the next value

 ---OUTPUTS: the mean error, stationarity of residuals, Gaussianity of
  residuals, and their autocorrelation structure.
```

We can also inspect this code `FC_LocalSimple` directly for more information. Like all code files for computing time-series features, `FC_LocalSimple.m` is located in the Operations directory of the *hctsa* repository. Inspecting the code file, we see that running `FC_LocalSimple(y,'mean',3)` does forecasting using local estimates of the time-series mean (since the second input to `FC_LocalSimple` , `forecastMeth` is set to `'mean'`), using the previous three time-series values to make the prediction (since the third input to `FC_LocalSimple` , `trainLength` is set to `3`).

To understand what the specific output quantity from this code is that came up as being highly informative in our `TS_TopFeatures` analysis, we need to look for the output labeled `taures` of the output structure produced by `FC_LocalSimple` . We discover the following relevant lines of code in `FC_LocalSimple.m` :

```
% Autocorrelation structure of the residuals:
out.ac1 = CO_AutoCorr(res,1,'Fourier');
out.ac2 = CO_AutoCorr(res,2,'Fourier');
out.taures = CO_FirstZero(res,'ac');
```

This shows us that, after doing the local mean prediction, `FC_LocalSimple` then outputs some features on whether there is any residual autocorrelation structure in the residuals of the rolling predictions (the outputs labeled `ac1`, `ac2`, and our output of interest: `taures`). The code shows that this `taures` output computes the `CO_FirstZero` of the residuals, which measures the first zero of the autocorrelation function (e.g., cf `help CO_FirstZero`). When the local mean prediction still leaves alot of autocorrelation structure in the residuals, our feature, `FC_LocalSimple_mean3_taures`, will thus take a high value.

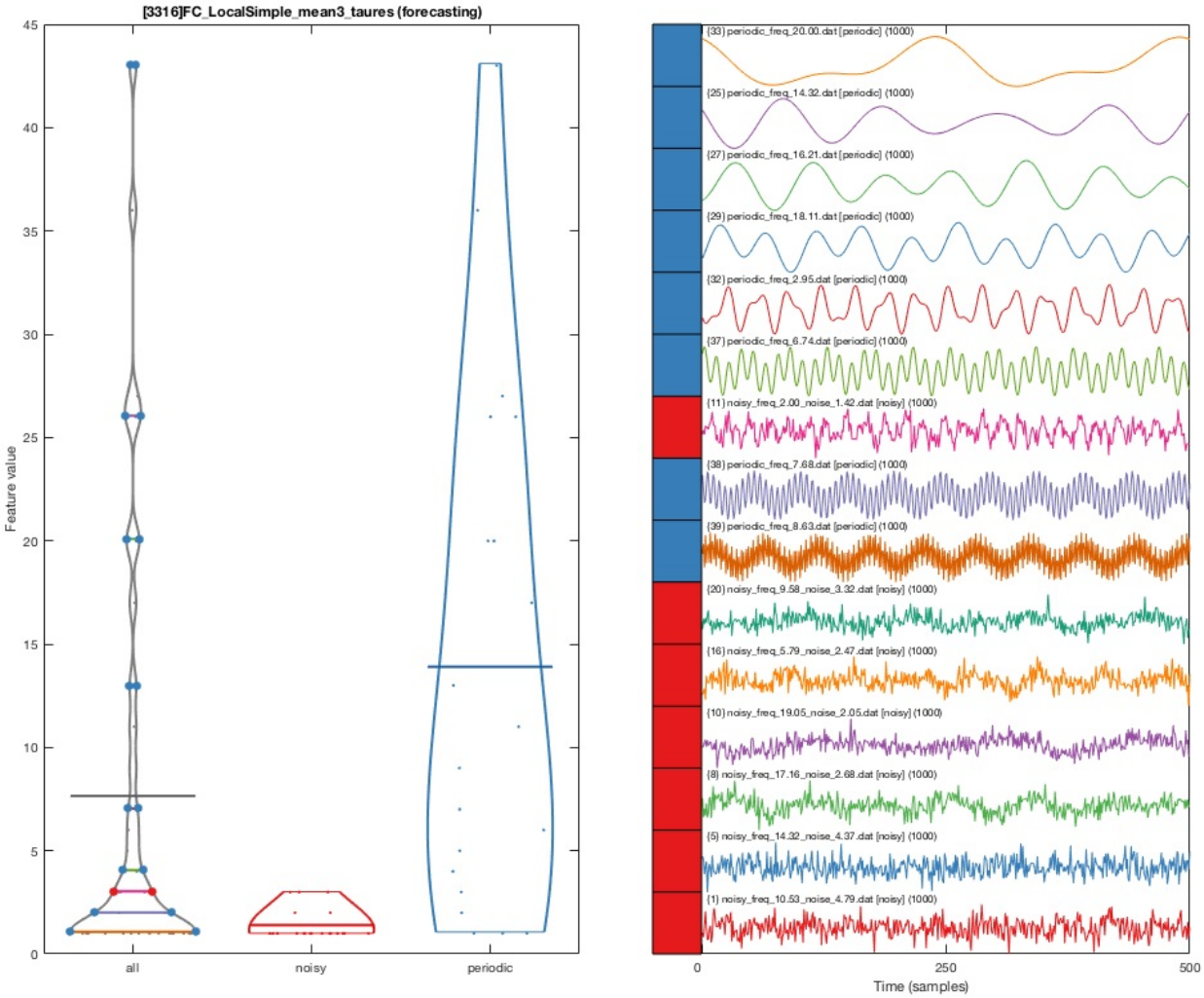
Visualizing outputs

Once we've seen the code that was used to produce a feature, and started to think about how such an algorithm might be measuring useful structure in our time series, we can then check our intuition by inspecting its performance on our dataset (as described in Investigating specific operations).

For example, we can run the following:

```
TS_FeatureSummary(3016,'raw',true);
```

which produces a plot like that shown below. We have run this on a dataset containing noisy sine waves, labeled 'noisy' (red) and periodic signals without noise, labeled 'periodic' (blue):



On the plot on the right, we see how this feature orders time series (with the distribution of values shown on the left, and split between the two groups: 'noisy', and 'periodic'). Our intuition from the code, that time series with longer correlation timescales will have highly autocorrelated residuals after a local mean prediction, appears to hold visually on this dataset. In general, the mechanism provided by `TS_FeatureSummary` to visualize how a given feature orders time series, including across labeled groups, can be a very useful one for feature interpretation.

Summary

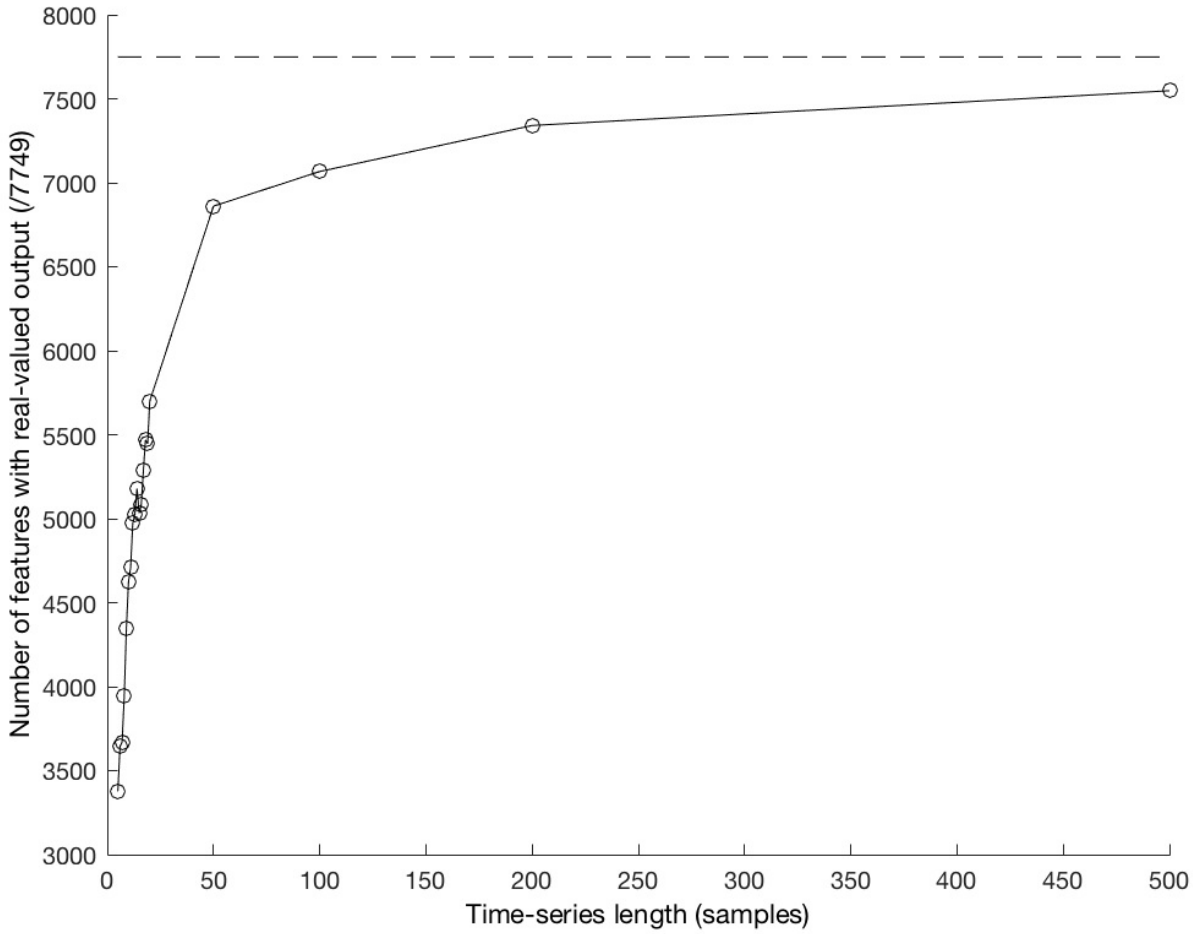
hctsa contains a large number of features, many of which can be expected to be highly inter-correlated on a given time-series dataset. It is thus crucial to explore how a given feature relates to other features in the library, e.g., using the correlation matrix produced by `TS_TopFeatures` (cf. Finding informative features), or by searching for features with similar behavior on the dataset to a given feature of interest (cf. Finding nearest neighbors). In a specific domain context, the analyst typically needs to decide on the trade-off between more complicated features that may have slightly higher in-sample performance on a given task,

and simpler, more interpretable features that may help guide domain understanding. The procedures outlined above are typically the first step to understanding a time-series analysis algorithm, and its relationship to alternatives that have been developed across science.

Working with short time series

Although many sections of the time-series analysis literature has worked to develop methods for quantifying complex temporal structure in long time-series recordings, many time series that are analyzed in practice are relatively short. *hctsa* has been successfully applied to time-series classification problems in the data mining literature, which includes datasets of time series as short as 60 samples, link. However, time-series data are sometimes even shorter, including yearly economic data across perhaps six years, or biological data measured at say 10 points across a lifespan. Although many features in *hctsa* will not give a meaningful output when applied to a short time series, *hctsa* includes methods for filtering such features (cf. `TS_normalize`), after which the remaining features can be used for analysis.

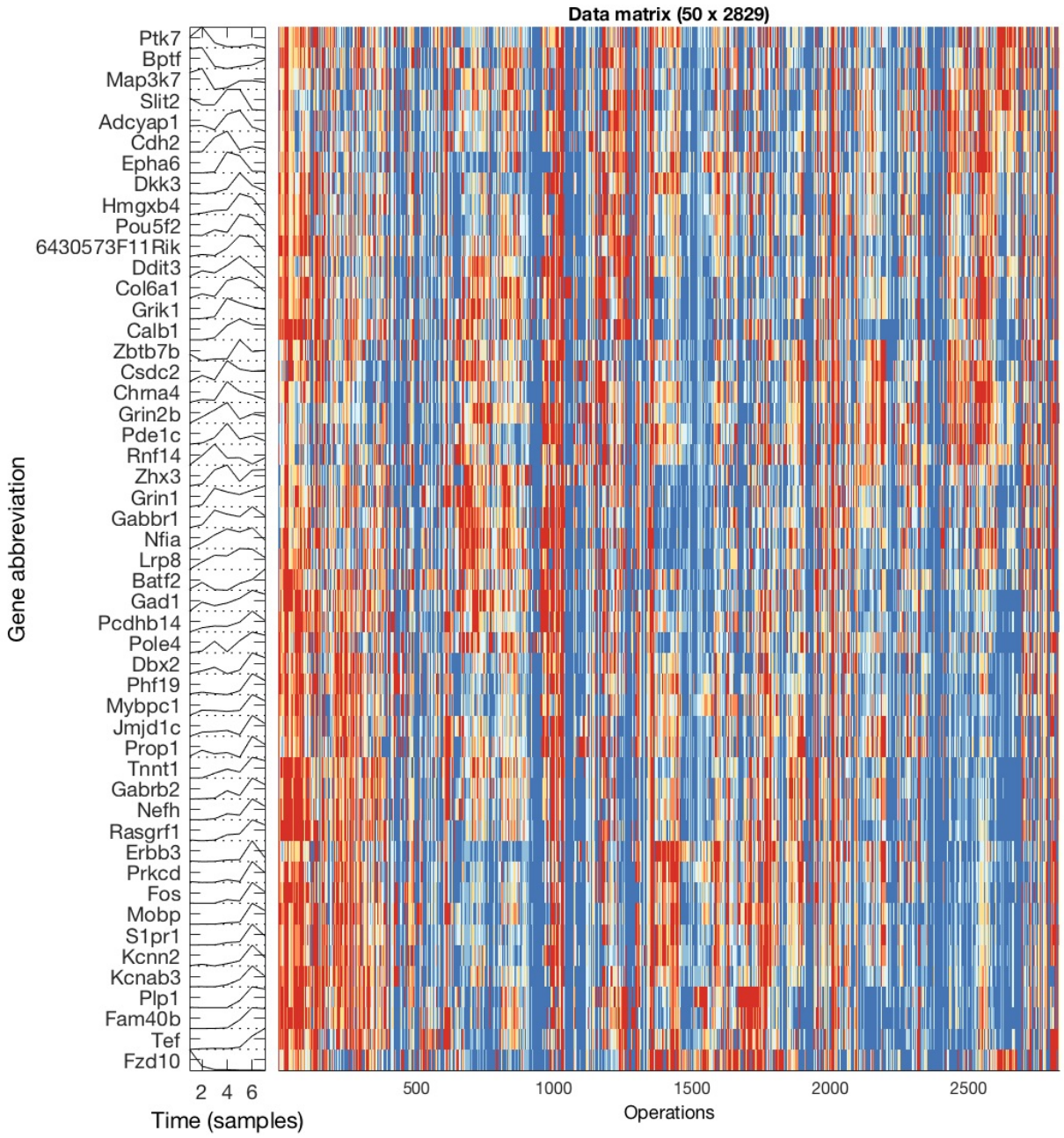
The number of features with a meaningful output, from time series as short as 5 samples, up to those with 500 samples, is shown below:



In each case, over 3000 features can be computed. Note that one must be careful when representing a 5-dimensional object, with thousands of features, the vast majority of which will be highly intercorrelated.

Sample application to developmental gene expression data

To demonstrate the feasibility of running *hctsa* analysis on datasets of short time series, we applied *hctsa* to gene expression data in the cerebellar brain region, r1A, across seven developmental time points (from the Allen Institute's Developing Mouse Brain Atlas), for a subset of 50 genes. After filtering and normalizing (`TS_normalize`), then clustering (`TS_cluster`), we plotted the clustered time-series data matrix (`TS_plot_DataMatrix('cl')`):



Inspecting the time series plots to the left of the colored matrix, we can see that genes with similar temporal expression profiles are clustered together based on their 2829-long feature vector representations. Thus, these feature-based representations are a rich and meaningful representation of these short time-series data. Further, while these 2829-long feature

vectors are shorter than those of longer time series, they still constitute a highly comprehensive representation that can be used as the starting point to obtain interpretable understanding in addressing specific domain questions.

Working with a *mySQL* database

When running large-scale *hctsa* computations, it can be useful to set up a *mySQL* database for time series, operations, and the computation results, and have many Matlab instances (running on different nodes of a compute cluster, for example) communicate directly with the database.

The *hctsa* software comes with this (optional) functionality, allowing a more powerful, distributed way to compute and store results of a large-scale computation.

This chapter outlines the steps involved in setting up, and running *hctsa* computations using a linked *mySQL* database.

Installing the *hctsa* code package to work with a *mySQL* database

The *hctsa* package requires some preliminary set up to work with a *mySQL* database, described here:

1. Installation of *mySQL*, either locally, or on an accessible server.
2. Setting up Matlab with a *mySQL* java connector (done by running the `install_jconnector` script in the **Database** directory, and then restarting Matlab).

After the database is set up, and the packages required by *hctsa* are installed (by running the `install` script), linking to a *mySQL* database can be done by running the `install_database` script, which:

1. Sets up Matlab to be able to communicate with the *mySQL* server and creates a new database to store Matlab calculations in, described here.
2. Populates the database with our default library of master operations and operations, as described here. (NB: a description of the terminology of 'master operations': a set of input arguments to an analysis function, and 'operations': a single time-series feature, is here).

This section contains additional details about each of these steps.

Note that the above steps are one-off installation steps; once the software is installed and compiled, a typical workflow will simply involve opening Matlab, running the `startup` script (which adds all paths required for the *hctsa* software), and then working within Matlab from any desired directory.

Adding a time-series dataset

Once installed using our default library of operations, the typical next step is to add a dataset of time series to the database using the `SQL_add` command. Custom master operations and operations can also be added, if required.

Computation, processing, and analysis

After installing the software and importing a time-series dataset to a *mySQL* database, the process by which data is retrieved from the database to local Matlab files (using `SQL_retrieve`), feature sets computed within Matlab (using `TS_compute`), and computed data stored back in the database (`SQL_store`) is described in detail here.

After the computation is complete for a time-series dataset, a range of processing, analysis, and plotting functions are also provided with the software, as described here.

Setting up the mySQL database

We assume that the user has access to and appropriate read/write privileges for a local or network *mySQL* server database. Instructions on how to install and set up a *mySQL* database on a variety of operating systems can be found here.

Setting Matlab up to talk to a mySQL server using the java connector

Before the structure of the database can be created, Matlab must be set up to be able to talk to the mySQL server, which requires installing a mySQL java connector. The steps required to achieve this are performed by the script `install_jconnector`, which should be run from the main *hctsa* directory. If this script runs successfully and a mySQL server has been installed (either on your local machine or on an external server, see above), you are then ready to run the `install` script.

The following outlines the actions performed by the `install_jconnector` script (including instructions on how to perform the steps manually):

It is necessary to relocate the J connector from the **Database** directory of this code repository (which is also freely available here): the file `mysql-connector-java-5.1.35-bin.jar` (for version 5.1.35). Instructions are here and are summarized below, and described in the Matlab documentation. This .jar file must be added to a static path where it can always be found by Matlab. A good candidate directory is the **java/jarext/** subdirectory of the Matlab root directory (to determine the Matlab root directory, simply type `matlabroot` in an open Matlab command window).

For Matlab to see this file, you need to add a reference to it in the **javaclasspath.txt** file (an alternative is to modify the **classpath.txt** file directly, but this may not be supported by newer versions of Matlab). This file can be found (or if it does not exist, should be created) in Matlab's preferences directory (to determine this location, type `prefdir` in a command window, or navigate to it within Matlab using `cd(prefdir)`).

This **javaclasspath.txt** file must contain a text reference to the location of the java connector on the disk. In the case recommended above, where it has been added to the **java/jarext** directory, we would add the following to the **javaclasspath.txt** file:

```
$matlabroot/java/jarext/mysql-connector-java-5.1.35-bin.jar
```

ensuring that the version number (5.1.35) matches your version of the J connector (if you are using a more recent version, for example).

Note that **javaclasspath.txt** can also be in Matlab's startup directory (for example, to modify this just for an individual user).

After restarting Matlab, Matlab should then have the ability to communicate with *mySQL* servers (we will check whether this works below).

Installing the Matlab/mySQL system

The main tasks involved in installing the Matlab/mySQL interface are achieved by the `install.m` script, which runs the user through the steps below.

Creating the mySQL database

If you have not already done so, creating a mySQL database to use with Matlab can be done using the `SQL_create_db` function. This requires that mySQL is installed on an accessible server, or on the local machine (i.e., using `localhost`). If the database has already been set up, then you do not need to use the `SQL_create_db` function but you must then create a text file, `sql-setting.conf`, in the **Database** directory of the repository. This file contains four comma-delimited entries corresponding to the server name, database name, username, and password, as per the following:

```
hostname,databasename,username,password
```

The settings listed here are those used to connect to the mySQL server. Remember that your password is sitting here in this document in unencrypted plain text, so do not use a secure or important password.

To check that Matlab can connect to external servers using the mySQL J-connector, using correct host name, username, and password settings, we introduce the Matlab routines `SQL_opendatabase` and `SQL_closedatabase`. An example usage is as follows:

```
% Open a connection to the default mySQL database as dbc:
% Connection details are stored in sql-setting.conf
dbc = SQL_opendatabase;

% <Do things with the database connection, dbc>

% Close the connection, dbc:
SQL_closedatabase(dbc);
```

For this to work, the **sql_settings.conf** file must be set up properly. This file specifies (in unencrypted plain text!) the login details for your mySQL database in the form `hostName,databaseName,username,password`.

An example **sql_settings.conf** file:

```
localhost,myTestDatabase,benfulcher,myInsecurePassword
```

Once you have configured your **sql_settings.conf** file, and you can run `dbc = SQL_opendatabase;` and `SQL_closedatabase(dbc)` without errors, then you can smile to yourself and you should at this point be happy because Matlab can communicate successfully with your mySQL server! You should also be excited because you are now ready to set up the database structure!

Note that if your database is not set up on your local machine (i.e., `localhost`), then Matlab can communicate with a mySQL server through an ssh tunnel, which requires some additional setup (described below).

Note also that the `SQL_opendatabase` function uses Matlab's *Database Toolbox* if a license is available, but otherwise will use java commands; both are supported and should give identical operational behavior.

Changing between different databases

To start writing a new dataset to a new database, or start retrieving data from a different database, you will need to change the database that Matlab is configured to connect to. This can be done using the `SQL_ChangeDatabase` script (which walks you through the steps and writes over the existing **sql_settings.conf** file), or by altering the **sql_settings.conf** file directly.

Note that one can swap between multiple databases easily by commenting out lines of the **sql_settings.conf** file (adding `%` to the start of a line to comment it out).

Setting up an ssh tunnel to a mySQL server

In some cases, the mySQL server you wish to connect to requires an ssh tunnel. One solution is to use port forwarding from your local machine to the server. The port forward can be set up in the terminal using a command like:

```
    ssh -L 1234:localhost:3306 myUsername@myServer.edu
```

This command connects port 1234 on your local computer to port 3306 (the default mySQL port) on the server. Now, telling Matlab to connect to `localhost` through port 1234 will connect it, through the established ssh tunnel, to the server. This can be achieved by specifying the server as `localhost` and the port number as 1234 in the **sql_settings.conf** file (or during the `install` process), which can be specified as the (optional) fifth entry, i.e.,:

```
hostname,databasename,username,password,1234
```

The database structure

The *mySQL* database is structured into four main components:

1. A lists of all the filenames and other metadata of time series (the **TimeSeries** table),
2. A list of all the names and other metadata of pieces of time-series analysis operations (the **Operations** table),
3. A list of all the pieces of code that must be evaluated to give each operation a value, which is necessary, for example, when one piece of code produces multiple outputs (the **MasterOperations** table), and
4. A list of the results of applying operations to time series in the database (the **Results** table).

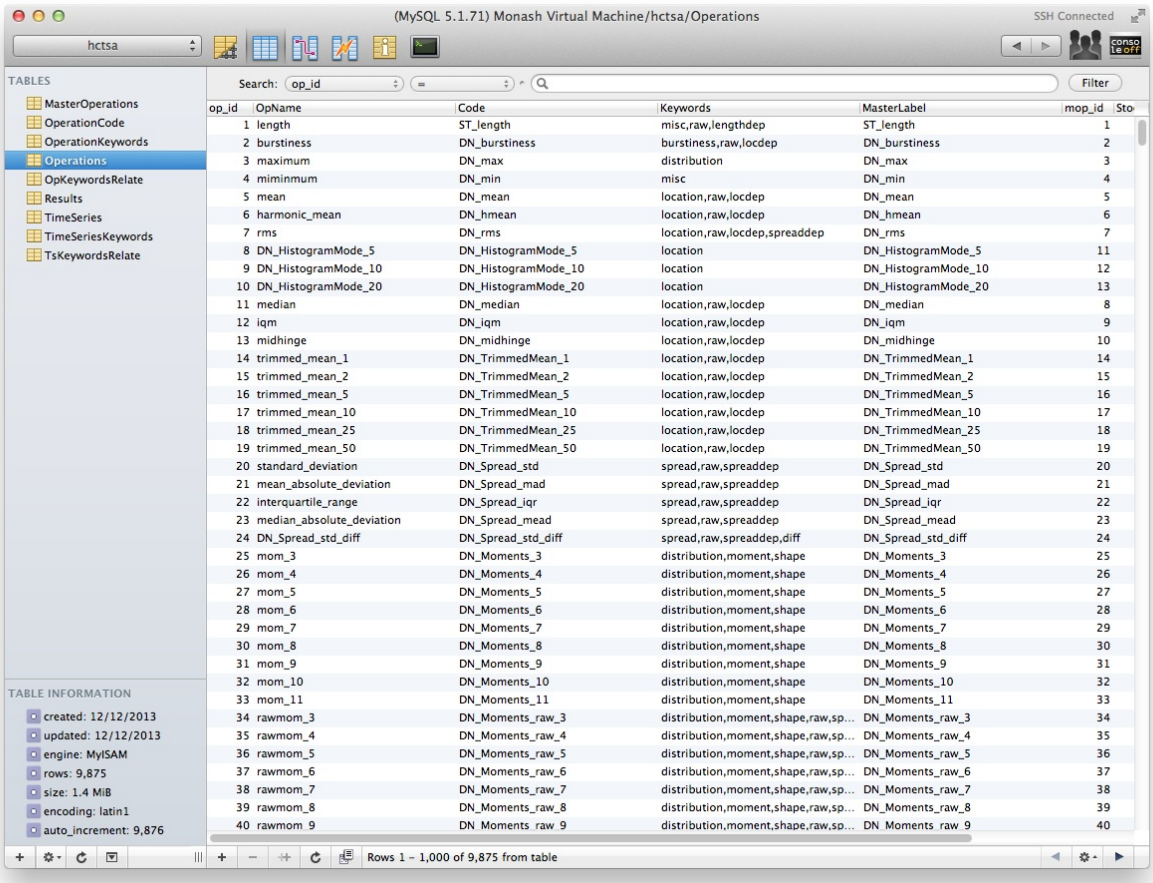
Additional tables are related to indexing and managing efficient keyword labeling, etc.

Time series and operations have their own tables that contain metadata associated with each piece of data, and each operation, and the results of applying each method to each time series is in the **Results** table, that has a row for every combination of time series and operation, where we also record calculation times and the quality of outputs (for cases where there the output of the operation was not a real number, or when some error occurred in the computation). Note that while data for each time series data *is* stored on the database, the executable time-series analysis code files ***are not***, such that all code files must be in Matlab's path (all required paths can be added by running `startup.m`).

Another handy (but dangerous) function to know about is `SQL_reset`, which will ***delete all data in the mySQL database***, create all the new tables, and then fill the database with all the time-series analysis operations. The **TimeSeries**, **Operations**, and **MasterOperations** tables can be generated by running `SQL_create_all_tables`, with master operations and operations added to the database using `SQL_add` commands (described here).

You now you have the basic table structure set up in the database and have done the first bits of *mySQL* manipulation through the Matlab interface.

It is very useful to be able to inspect the database directly, using a graphical interface. A very good example for Mac is the excellent, free application, *Sequel Pro* (a screenshot is shown below, showing the first 40 rows of the Operations table of our default operations library). Applications similar to *Sequal Pro* exist for Windows and Linux platforms. Applications that allow the database to be inspected are extremely useful, however they should ***not*** be used to manipulate the database directly. Instead, Matlab scripts should be used to interact with the database to ensure that the proper relationships between the different tables are maintained (including the indexing of keyword labels).


(MySQL 5.1.71) Monash Virtual Machine/hctsa/Operations
SSH Connected
hctsa
TABLES
MasterOperations
OperationCode
OperationKeywords
Operations
OpKeywordsRelate
Results
TimeSeries
TimeSeriesKeywords
TsKeywordsRelate
TABLE INFORMATION
created: 12/12/2013
updated: 12/12/2013
engine: MyISAM
rows: 9,875
size: 1.4 MiB
encoding: latin1
auto_increment: 9,876
Search: op_id = Filter
op_id | OpName | Code | Keywords | MasterLabel | mop_id
1 | length | ST_length | misc,raw,lengthdep | ST_length | 1
2 | burstiness | DN_burstiness | burstiness,raw,locdep | DN_burstiness | 2
3 | maximum | DN_max | distribution | DN_max | 3
4 | miminmum | DN_min | misc | DN_min | 4
5 | mean | DN_mean | location,raw,locdep | DN_mean | 5
6 | harmonic_mean | DN_hmean | location,raw,locdep | DN_hmean | 6
7 | rms | DN_rms | location,raw,locdep,spreaddep | DN_rms | 7
8 | DN_HistogramMode_5 | DN_HistogramMode_5 | location | DN_HistogramMode_5 | 11
9 | DN_HistogramMode_10 | DN_HistogramMode_10 | location | DN_HistogramMode_10 | 12
10 | DN_HistogramMode_20 | DN_HistogramMode_20 | location | DN_HistogramMode_20 | 13
11 | median | DN_median | location,raw,locdep | DN_median | 8
12 | iqm | DN_iqm | location,raw,locdep | DN_iqm | 9
13 | midhinge | DN_midhinge | location,raw,locdep | DN_midhinge | 10
14 | trimmed_mean_1 | DN_TrimmedMean_1 | location,raw,locdep | DN_TrimmedMean_1 | 14
15 | trimmed_mean_2 | DN_TrimmedMean_2 | location,raw,locdep | DN_TrimmedMean_2 | 15
16 | trimmed_mean_5 | DN_TrimmedMean_5 | location,raw,locdep | DN_TrimmedMean_5 | 16
17 | trimmed_mean_10 | DN_TrimmedMean_10 | location,raw,locdep | DN_TrimmedMean_10 | 17
18 | trimmed_mean_25 | DN_TrimmedMean_25 | location,raw,locdep | DN_TrimmedMean_25 | 18
19 | trimmed_mean_50 | DN_TrimmedMean_50 | location,raw,locdep | DN_TrimmedMean_50 | 19
20 | standard_deviation | DN_Spread_std | spread,raw,spreaddep | DN_Spread_std | 20
21 | mean_absolute_deviation | DN_Spread_mad | spread,raw,spreaddep | DN_Spread_mad | 21
22 | interquartile_range | DN_Spread_iqr | spread,raw,spreaddep | DN_Spread_iqr | 22
23 | median_absolute_deviation | DN_Spread_mead | spread,raw,spreaddep | DN_Spread_mead | 23
24 | DN_Spread_std_diff | DN_Spread_std_diff | spread,raw,spreaddep,diff | DN_Spread_std_diff | 24
25 | mom_3 | DN_Moments_3 | distribution,moment,shape | DN_Moments_3 | 25
26 | mom_4 | DN_Moments_4 | distribution,moment,shape | DN_Moments_4 | 26
27 | mom_5 | DN_Moments_5 | distribution,moment,shape | DN_Moments_5 | 27
28 | mom_6 | DN_Moments_6 | distribution,moment,shape | DN_Moments_6 | 28
29 | mom_7 | DN_Moments_7 | distribution,moment,shape | DN_Moments_7 | 29
30 | mom_8 | DN_Moments_8 | distribution,moment,shape | DN_Moments_8 | 30
31 | mom_9 | DN_Moments_9 | distribution,moment,shape | DN_Moments_9 | 31
32 | mom_10 | DN_Moments_10 | distribution,moment,shape | DN_Moments_10 | 32
33 | mom_11 | DN_Moments_11 | distribution,moment,shape | DN_Moments_11 | 33
34 | rawmom_3 | DN_Moments_raw_3 | distribution,moment,shape,raw,sp... | DN_Moments_raw_3 | 34
35 | rawmom_4 | DN_Moments_raw_4 | distribution,moment,shape,raw,sp... | DN_Moments_raw_4 | 35
36 | rawmom_5 | DN_Moments_raw_5 | distribution,moment,shape,raw,sp... | DN_Moments_raw_5 | 36
37 | rawmom_6 | DN_Moments_raw_6 | distribution,moment,shape,raw,sp... | DN_Moments_raw_6 | 37
38 | rawmom_7 | DN_Moments_raw_7 | distribution,moment,shape,raw,sp... | DN_Moments_raw_7 | 38
39 | rawmom_8 | DN_Moments_raw_8 | distribution,moment,shape,raw,sp... | DN_Moments_raw_8 | 39
40 | rawmom_9 | DN_Moments_raw_9 | distribution,moment,shape,raw,sp... | DN_Moments_raw_9 | 40
Rows 1 – 1,000 of 9,875 from table

Populating the database with time series and operations using `SQL_add`

When linking Matlab to a *mySQL* database, metadata associated with time series, operations, and master operations, as well as the results of computations are all stored in an indexed database. Adding master operations, operations, and time series to the database can be achieved using the `SQL_add` function, as described below.

The following table summarizes the terminology used for each type of object in *hctsa* land:

	Master Operation	Operation	Time Series
Database identifier:	mop_id	op_id	ts_id
Input to `SQL_add` :	'mops'	'ops'	'ts'

Using `SQL_add`

`SQL_add` has two key inputs that specify:

1. Whether to import a set of time series (specify `'ts'`), a set of operations (specify `'ops'`), or a set of master operations (specify `'mops'`),
2. The name of the input file that contains appropriately-formatted information about the time series, master operations, or operations to be imported.

In this section, we describe how to use `SQL_add` to add master operations, operations, and time series to the database.

Users wishing to run the default *hctsa* code library their own time-series dataset will only need to add time series to the database, as the full operation library is added by default by the `install.m` script. Users wishing to add additional features using custom time-series code or different types of inputs to existing code files, can either edit the default *INP_ops.txt* and *INP_mops.txt* files provided with the repository, or create new input files for their custom analysis methods (as explained for operations and master operations).

REMINDER: Manually editing the database, including adding or deleting rows, is very dangerous, as it can create inconsistencies and errors in the database structure. Adding time series and operations to the database should only be done using `SQL_add` which sets up the **Results** table of the database and ensures that the indexing relationships in the database are properly maintained.

Example: Adding our library of master operations to the database

By default, the `install` script populates the database with the default library of highly comparative time-series analysis code. The formatted input file specifying these pieces of code and their input parameters is **INP_mops.txt** in the **Database** directory of the repository. This step can be reproduced using the following command:

```
SQL_add('mops','INP_mops.txt');
```

Once added, each master operation is assigned a unique integer, **mop_id**, that can be used to identify it. For example, when adding individual operations, the **mop_id** is used to map each individual operation to a corresponding master operation.

Adding new pieces of executable code to the database

New functions and their input parameters to execute can be added to the database using `SQL_add` in the same way as described above. For example, lines corresponding to the new code can be added to the current **INP_mops.txt** file, or by generating a new input file and running `SQL_add` on the new input file. Once in the database, the software will then run the new pieces of code. Note that `SQL_add` checks for repeats that already exist in the database, so that duplicate database entries cannot be added with `SQL_add`.

New code added to the database should be checked for the following:

1. Output is a real number or structure (and uses an output of NaN to assign all outputs to a NaN).
2. The function is accessible in the Matlab path.
3. Output(s) from the function have matching operations (or features), which also need to be added to the database.

Corresponding operations (or features) will then need to added separately, to link to the structured outputs of master operations.

Example: Adding our library of operations to the database

Operations can be added to the *mySQL* database using an appropriately-formatted input file, such as `INP_ops.txt`, as follows:

```
SQL_add('ops','INP_ops.txt');
```

Every operation added to the database will be given a unique integer identifier, **op_id**, which provides a common way of retrieving specific operations from the database.

Note that after (hopefully successfully) adding the operations to the database, the `SQL_add` function indexes the operation keywords to an **OperationKeywords** table that produces a unique identifier for each keyword, and another linking table that allows operations with each keyword to be retrieved efficiently.

Adding time series to the database

After setting up a database with a library of time-series features, the next task is to add a dataset of time series to the database. It is up to the user whether to keep all time-series data in a single database, or have a different database for each dataset.

Time series are added using the same function used to add master operations and operations to the database, `SQL_add`, which imports time series data (stored in time-series data files) and associated keyword metadata (assigned to each time series) to the database.

Time series can be indexed by assigning keywords to them (which are stored in the **TimeSeriesKeywords** table and associated index table, **TsKeywordsRelate** of the database).

When added to the *mySQL* database, every time series added to the database is assigned a unique integer identifier, **ts_id**, which can be used to retrieve specific time series from the database.

Adding a set of time series to the database requires an appropriately formatted input file, following either of the following:

```
% Add time series (embedded in a .mat file):
SQL_add('ts','INP_ts.mat');

% Add time series (stored in data files) using an input text file:
SQL_add('ts','INP_ts.txt');
```

We provide an example input file in the **Database** directory as **INP_test_ts.txt**, which can be added to the database, following the syntax above, using `SQL_add('ts','INP_test_ts.txt')`, as well as a sample .mat file input as **INP_test_ts.mat**, which can be added as `SQL_add('ts','INP_test_ts.mat')`.

Adding time series to the database

After setting up a database with a library of time-series features, the next task is to add a dataset of time series to the database. It is up to personal preference of the user whether to keep all time-series data in a single database, or have a different database for each dataset.

Time series are added using the same function used to add master operations and operations to the database, `SQL_add`, which imports time series data (stored in time-series data files) and associated keyword metadata (assigned to each time series) to the database. The time-series data files to import, and the keywords to assign to each time series are specified in either: (i) an appropriately formatted matlab (`.mat`) file, or (ii) a structured input text file, as explained below.

Time series can be indexed by assigning keywords to them (which are stored in the **TimeSeriesKeywords** table and associated index table, **TsKeywordsRelate** of the database). Assigning keywords to time series makes it easier to retrieve a set of time series with a given set of keywords for analysis, and to group time series annotated with different keywords for classification tasks.

When added to the *mySQL* database, every time series added to the database is assigned a unique integer identifier, **ts_id**, which can be used to retrieve specific time series from the database.

`SQL_add` syntax

Adding a set of time series to the database requires an appropriately formatted input file, **INP_ts.txt**, for example, the appropriate code is:

```
% Add time series (embedded in a .mat file):
SQL_add('ts','INP_ts.mat');

% Add time series (stored in data files) using an input text file:
SQL_add('ts','INP_ts.txt');
```

We provide an example input file in the **Database** directory as **INP_test_ts.txt**, which can be added to the database, following the syntax above, using `SQL_add('ts','INP_test_ts.txt')`, as well as a sample .mat file input as **INP_test_ts.mat**, which can be added as `SQL_add('ts','INP_test_ts.mat')`.

Retrieving data from the database using `SQL_retrieve`

Retrieving data from the **Results** table of the database is typically done for one of two purposes:

1. To calculate as-yet uncalculated entries to be stored back into the database, and
2. To analyze already-computed data stored in the database in Matlab.

The function `SQL_retrieve` performs both of these functions, using different inputs. Here we describe the use of the `SQL_retrieve` function for the purposes of populating uncalculated (NULL) entries in the **Results** table of the database in Matlab.

For calculating missing entries in the database, `SQL_retrieve` can be run as follows:

```
SQL_retrieve(ts_ids, op_ids, 'null');
```

The third input, `'null'`, retrieves **ts_id**s and **op_id**s from the sets provided that contain (as-yet) uncalculated (i.e., NULL) elements in the database; these can then be calculated and stored back in the database. An example usage is given below:

```
SQL_retrieve(1:5, 'all', 'null');
```

Running this code will retrieve null (uncalculated) data from the database for time series with **ts_id**s between 1 and 5 (inclusive) and all operations in the database, keeping only the rows and columns of the resulting time series x operations matrix that contain NULLs.

When calculations are complete and one wishes to analyze all of the data stored in the database (not just NULL entries requiring computation), the third input should be set to ‘all’ to retrieve all entries in the **Results** table of the database, as described later.

`SQL_retrieve` writes to a local Matlab file, **HCTSA.mat**, that contains the data retrieved from the database.

Computing operations on time series and writing back to the database

After retrieving data from the *mySQL* database, missing entries (NULL in the database, and NaN in the local Matlab file) can be computed using `TS_compute`, and stored back to the database using `SQL_store`. These functions are described below.

Performing calculations using `TS_compute`

Values retrieved using `SQL_retrieve` (to the local `HCTSA.mat` file) that have not previously been calculated are evaluated using `TS_compute`, as described here. These results can then be inspected directly (if needed), or simply written back to the database using `SQL_store`, as described below.

Writing calculations back to the database using `SQL_store`

Once calculations have been performed using Matlab on local files, the results must be written back to the database. This task is performed by `SQL_store`, which reads the data in `HCTSA.mat`, checks that the metadata still matches the database, and then begins updating the **Output**, **Quality**, and **CalculationTime** columns of the **Results** table in the *mySQL* database. This can be done by simply running:

```
SQL_store;
```

Depending on database latencies, this can be a relatively slow process, up to 20-25 s per time series, updating each row in the **Results** table individually using *mySQL* **UPDATE** statements. However, the delay in this step means that the computation can be distributed across multiple compute nodes, and that stored data can be indexed and retrieved systematically. Keeping results in local Matlab files can be extremely inefficient, and can indeed be untenable for large datasets.

Cycling through computations using runscripts

As described above, computation involves three main steps:

The procedure involves three main steps:

1. Retrieve a set of time series and operations from (the **Results** table) of the database to a local Matlab file, **HCTSA.mat** (using `SQL_retrieve`).
2. Compute the operations on the retrieved time series in Matlab and store the results locally (using `TS_compute`).
3. Write the results back to the **Results** table of the database (using `SQL_store`).

It is usually the most efficient practice to retrieve a small number of time series at each iteration of the `SQL_retrieve` – `TS_compute` – `SQL_store` loop, and distribute this computation across multiple machines if possible. An example runscript is given in the code that accompanies this document, as `sample_runscript_sql`, which retrieves a single time series at a time, computes it, and then writes the results back to the database in a loop. This can be viewed as a template for runscripts that one may wish to use when performing time-series calculations across the database.

This workflow is well suited to distributed computing for large datasets, whereby each node can iterate over a small set of time series, with all the results being written back to a central location (the *mySQL* database).

By designating different sections of the database to cycle through, this procedure can also be used to (manually) distribute the computation across different machines. Retrieving a large section of the database at once can be problematic because it requires large disk reads and writes, uses a lot of memory, and if problems occur in the reading or writing to/from files, one may have to abandon a large number of existing computations.

Clearing or removing data from the database using `SQL_clear_remove`

Sometimes you might wish to remove a problematic set of time series from the database (that might have been input incorrectly, for example), including their metadata, and all records of computed data. Other times you might find a problem with the implementation of one of the operations. In this case, you would like to retain that operation in the database, but flush all of the data currently computed for it (so that you can recompute new values). Both of these types of tasks (both removing and clearing time series or operations) can be achieved with the function `SQL_clear_remove`.

This function takes in information about whether to clear (clear any calculated data) or remove (completely delete the given time series or operations from the database).

Example usages are given below:

```
% Clear time series with ts_ids in the range 10-15
SQL_clear_remove('ts',10:15,0);

% Remove time series with ts_ids in the range 10-15
SQL_clear_remove('ts',10:15,1);

% Clear operations with op_ids in the range 100-200
SQL_clear_remove('ops',100:200,0);
```

Retrieving results from the database using `SQL_retrieve`

The first step of any analysis is to retrieve a relevant portion of data from the *mySQL* database to local Matlab files for analysis. This is done using the `SQL_retrieve` function described earlier, except we use the `'all'` input to retrieve all data, rather than the `'null'` input used to retrieve just missing data (requiring calculation).

Example usage is as follows:

```
>> SQL_retrieve(ts_ids, op_ids,'all');
```

for vectors `ts_ids` and `op_ids`, specifying the **ts_id**s and **op_id**s to be retrieved from the database.

Sets of **ts_id**s and **op_id**s to retrieve can be selected by inspecting the database, or by retrieving relevant sets of keywords using the `SQL_getids` function. Running the code in this way, using the 'all' tag, ensures that the full range of **ts_id**s and **op_id**s specified are retrieved from the database and stored in the local file, **HCTSA.mat**, which can then form the basis of subsequent analysis.

The database structure provides much flexibility in storing and indexing the large datasets that can be analyzed using the *hctsa* approach, however the process of retrieving and storing large amounts of data from a database can take a considerable amount of time, depending on database latencies.

Note that missing, or NULL, entries in the database are converted to NaN entries in the local Matlab matrices.

Error handling and maintenance

In this section we describe how keywords and other types of metadata stored in the database can be manipulated, and how results of whole sets of operations and time series can be cleared or completely deleted from the database. These tasks are implemented as Matlab functions, and ensure that key database structures are maintained. Instead performing such tasks by acting directly on the database can cause inconsistencies and should be avoided.

Java heap space

Running out of java heap space throws the error `java.lang.OutOfMemoryError`, and usually happens when trying to retrieve a large set of time series/operations from the database. Matlab needs to keep the whole retrieval in memory, and has a hard limit on this. The java heap size can be increased in the Matlab preferences, under *General* → *Java Heap Memory*.