LTPDA Toolbox

Welcome to the LTPDA Toolbox for MATLAB. This toolbox provides an object-oriented data analysis environment and is designed to carry out the analysis for the LISA PathFinder mission.

Functions - Alphabetical List Getting Started with the LTPDA Toolbox

Getting Started with the LTPDA Toolbox

What is the LTPDA Toolbox

This section covers the following topics:

Overview

The LTPDA Toolbox is a MATLAB® Toolbox designed for the analysis of data from the LISA Technology Package (part of the LISA Pathfinder Mission). The Toolbox implements accountable and reproducible data analysis within MATLAB®.

With the LTPDA Toolbox you operate with Analysis Objects. An Analysis Object captures more than just the data from a particular analysis: it also captures the full processing history that led to this particular result, as well as full details of by whom and where the analysis was performed.

Features of the LTPDA Toolbox

The LTPDA Toolbox has the following features:

- Create and process multiple Analysis Objects (AOs).
- Save and load AOs from XML files.
- Plot/view the history of the processing in any particular AO.
- Submit and retrieve AOs to/from an LTPDA Repository.
- Powerful and easy to use Signal Processing capabilities.

> **Note** LTPDA Repositories are external to MATLAB and need to be set up independently.

Expected Background for Users

MATLAB

This documentation assumes you have a basic working understanding of MATLAB. You need to know basic MATLAB syntax for writing your own m-files, and you need to have basic familiarity with the SIMULINK interface to use the LTPDA GUI.

System Requirements

The LTPDA Toolbox works with the systems and applications described here:

Platforms

The LTPDA Toolbox is expected to run on all of the platforms that support MATLAB, but you cannot run MATLAB with the `-nojvm` startup option.

MATLAB and Related Products

The LTPDA Toolbox requires MATLAB. To use the LTPDA GUI you need SIMULINK. In addition, the following MathWorks Toolboxes are required:

- Signal Processing Toolbox
- Database Toolbox (for LTPDA Repository interaction)

The following versions represent the reference platform against which the LTPDA Toolbox is tested.

Component	Version	Comment
MATLAB	7.6 (R2008a)	
Simulink	7.1	Needed for GUI programming
Signal Processing Toolbox	6.9	
Database Toolbox	3.4.1	Needed to interact with an LTPDA repository
Symbolic Math Toolbox	3.2.3	
Optimisation Toolbox	4.0	

Setting-up MATLAB

Setting up MATLAB to work properly with the LTPDA Toolbox requires a few steps:

Add the LTPDA Toolbox to the MATLAB path

After downloading and un-compressing the LTPDA Toolbox, you should add the directory `ltpda_toolbox` to your MATLAB path. To do this:

- File > Set Path... >
- Choose "Add with Subfolders" and browse to the location of `ltpda_toolbox`
- "Save" your new path. MATLAB may require you to save your new `pathdef.m` to a new location in the case that you don't have write access to the default location. For more details read the documentation on "pathdef" (`>> doc pathdef`).

Edit the LTPDA startup script

The LTPDA Toolbox comes with a default `ltpda_startup.m` file. This may need to be edited for your particular system (though most of the defaults should be fine). To edit this file, do `>> edit ltpda_startup` at the MATLAB command prompt. The following table lists the various variables that are set there:

Variable	Values	Description
USE_LTPDA_PRINT_SETTINGS	'Yes' or 'no'	Choose the LTPDA Print settings, or stick with your own (MATLAB) defaults.
USE_LTPDA_PLOT_SETTINGS	'Yes' or 'no''	Choose the LTPDA Plot settings, or stick with your own (MATLAB) defaults.
REPO_GUI_SERVERLIST	{'localhost', '130.75.117.67', '130.75.117.61'}	A list of default LTPDA repositories that will be presented to the user by the LTPDA Repository GUI. Each hostname is one entry in a cell-array.
REPO_GUI_FONTSIZE	10	A default font-size used on the LTPDA Repository GUI.
PASSWD_WIN_FONTSIZE	10	A default font-size used on the password dialog box for connection to LTPDA Repositories.
TIMEZONE	'UTC'	A default time-zone used when dealing with

		LTPDA Time objects. A list of valid time-zones can be retried with the command: `>> java.util.TimeZone.getAvailableIDs`
TIME_FORMAT_STR	'yyyy-mm-dd HH:MM:SS.FFF'	A default time format string.
VERBOSE_LEVEL	0-11	The level of terminal output from the toolbox. Higher numbers result in increased terminal output.

Automatically start LTPDA Toolbox on starting MATLAB

In order to automatically load the LTPDA Toolbox settings when MATLAB starts up, add the command `ltpda_startup` to your own `startup.m` file. See `>> doc startup` for more details on installing and editing your own `startup.m` file.

System Requirements Starting the LTPDA Toolbox

Starting the LTPDA Toolbox

In order to access the functionalty of the LTPDA Toolbox, it is necessary to run the command `ltpda_startup`, either at the MATLAB command prompt, or by placing the command in your own `startup.m` file.

The main Graphical User Interface can be started with the command `ltpdagui`. The LTPDA Launch Bay can be started with the command `ltpdalauncher`.

Setting-up MATLAB Trouble-shooting

Trouble-shooting

A collection of trouble-shooting steps.

1. **Java Heap Problem**

   When loading or saving large XML files, MATLAB sometimes reports problems due to insufficient heap memory for the Java Virtual Machine.
   You can increase the heap space for the Java VM in MATLAB 6.0 and higher by creating a `java.opts` file in the `$MATLAB/bin/$ARCH` (or in the current directory when you start MATLAB) containing the following command:

   ```
   -Xmx$MEMSIZE
   ```

   Recommended:

   ```
   -Xmx536870912
   ```

   which is 512Mb of heap memory.
   An additional workaround reported in case the above doesn't work: It sometimes happens with MATLAB R2007b on WinXP that after you create the `java.opts` file, MATLAB won't start (it crashes after the splash-screen).
   The workaround is to set an environment variable `MATLAB_RESERVE_LO=0`.

   This can be set by performing the following steps:

   1. Select `Start->Settings->Control Panel->System`
   2. Select the "Advanced" tab
   3. On the bottom, center, click on "Environment variables"
   4. Click "New" (choose the one under "User variables for Current User")
   5. Enter
      Variable Name: MATLAB_RESERVE_LO
      Variable Value: 0
   6. Click OK as many times as needed to close the window

   Then edit/create the `java.opts` file as described above. You can also specify the units (for instance -Xmx512m or -Xmx524288k or -Xmx536870912 will all give you 512 Mb).

Examples

General

The directory `examples` in the LTPDA Toolbox contains a large number of example scripts that demonstrate the use of the toolbox for scripting.

You can execute all of these tests by running the command `run_tests`

Constructor examples

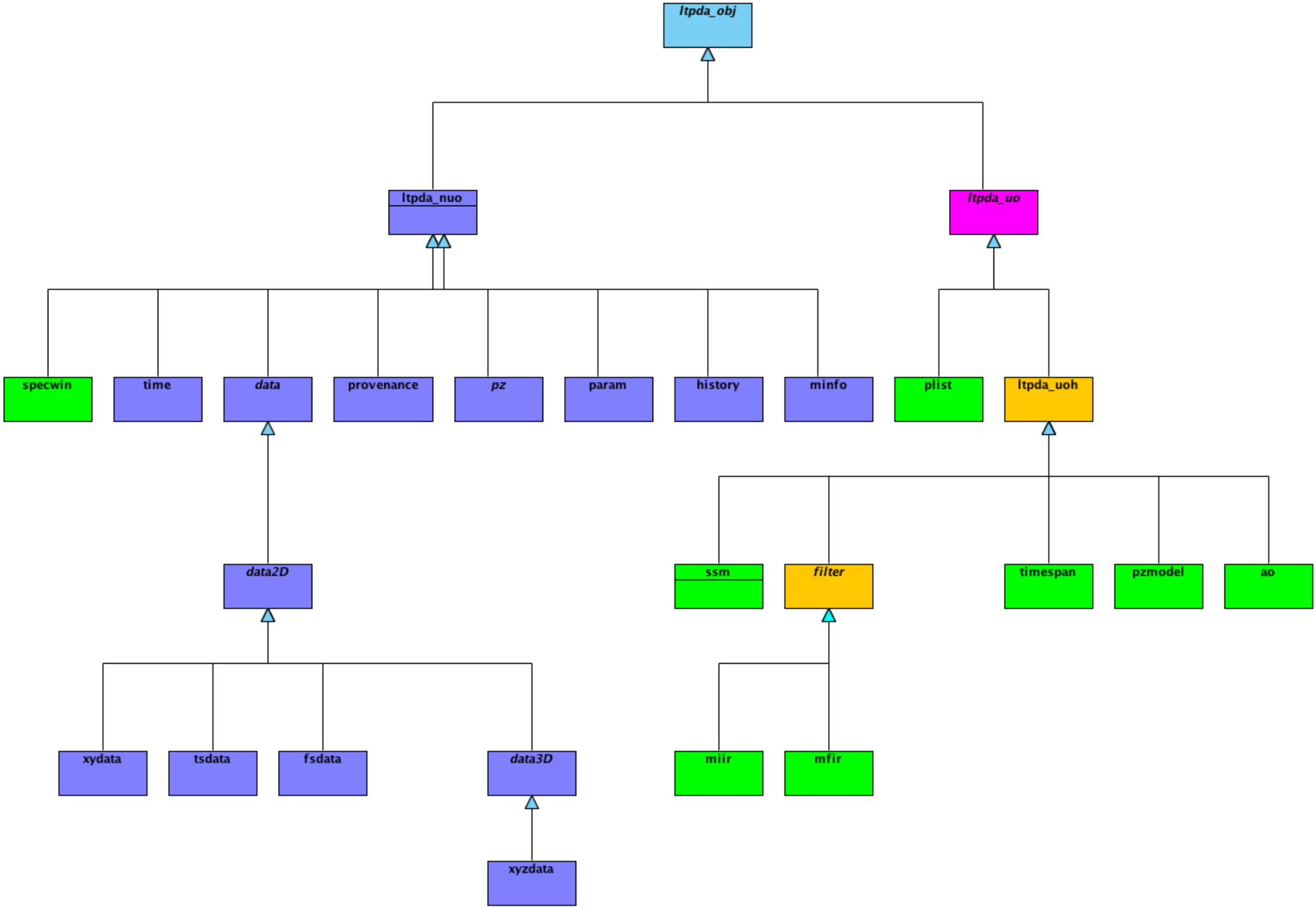
Introducing LTPDA Objects

The LTPDA toolbox is object oriented and as such, extends the MATLAB object types to many others. All data processing is done using objects and methods of those classes.

For full details of objects in MATLAB, refer to MATLAB Classes and Object-Oriented Programming.

LTPDA Classes

Various classes make up the object-oriented infrastructure of LTPDA. The figure below shows all the classes in LTPDA. All classes are derived from the base class, `ltpda_obj`. The classes then fall into two main types deriving from the classes `ltpda_nuo` and `ltpda_uo`.


ltpda_obj
ltpda_nuo
ltpda_uo
specwin
time
data
provenance
pz
param
history
minfo
plist
ltpda_uoh
data2D
ssm
filter
timespan
pzmodel
ao
xydata
tsdata
fsdata
data3D
miir
mfir
xyzdata

The left branch, `ltpda_nuo`, are termed 'non-user objects'. These objects are not typically accessed or created by users. The right branch, `ltpda_uo`, are termed 'user objects'. These objects have a 'name' and a 'history' property which means that their processing history is tracked through all LTPDA algorithms. In addition, these 'user objects' can be saved to disk or to an LTPDA repository.

The objects drawn in green are expected to be created by users in scripts or on the LTPDA GUI.

Examples Creating LTPDA Objects

Creating LTPDA Objects

Content needs written...

Introducing LTPDA Objects Working with LTPDA objects

Working with LTPDA objects

The use of LTPDA objects requires some understanding of the nature of objects as implemented in MATLAB.

For full details of objects in MATLAB, refer to MATLAB Classes and Object-Oriented Programming. For convenience, the most important aspects in the context of LTPDA are reviewed below.

Calling object methods

Setting object properties

Exploiting the handle nature of LTPDA objects

Copying objects

User Objects

Content needs written...

Working with LTPDA objects Analysis Objects

Analysis Objects

Based on the requirement that all results produced by the LTP Data Analysis software must be easily reproducible as well as fully traceable, the idea of implementing analysis objects (AO) as they are described in S2-AEI-TN-3037 arose.

An analysis object contains all information necessary to be able to reproduce a given result. For example

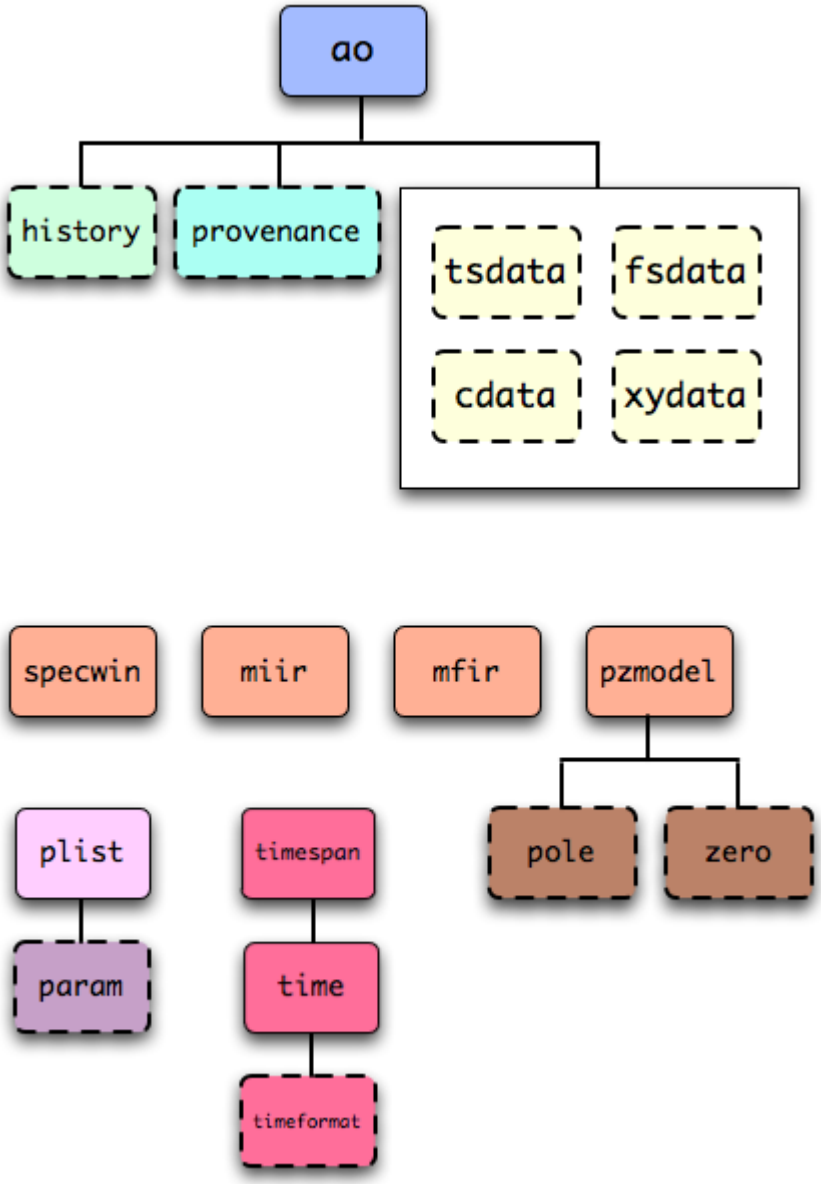
- which raw data was involved (date, channel, time segment, time of retrieval if data can be changed later by new downlinks)
- all operations performed on the data
- the above for all channels of a multi-channel plot

The AO will therefore hold

- the numerical data belonging to the result
- the full processing history needed to reproduce the numerical result

The majority of algorithms in the LTPDA Toolbox will operate on AOs only (in some cases they are methods of the AO class) but there are also utility functions which do not take AOs as inputs, and do not produce AOs as outputs. Functions in the toolbox are designed to be as simple and elementary as possible.

The functionality of Analysis Objects is built upon a set of MATLAB classes. The classes are depicted in the following Figure.



Thos boxes with the dotted outlines are not considered to be user classes; they are typically used internally to algorithms of class methods.

Details of each class are given in:

ao class
History class
Provenance class
tsdata class
fsdata class
xydata class
cdata class
plist class
param class
specwin class
miir class
mfir class

pzmodel class
pole class
zero class
timespan class
time class
timeformat class

User Objects Creating Analysis Objects

Creating Analysis Objects

Analysis objects can be created in MATLAB in many ways. Apart from being created by the many algorithms in the LTPDA Toolbox, AOs can also be created from initial data or descriptions of data. The various *constructors* are listed in the function help: ao help.

Examples of creating AOs

The following examples show some ways to create Analysis Objects.

Creating AOs from text files.

Analysis Objects can be created from text files containing two columns of ASCII numbers. Files ending in '.txt' or '.dat' will be handled as ASCII file inputs. The first column is taken to be the time instances; the second column is taken to be the amplitude samples. The created AO is of type `tsdata` with the sample rate set by the difference between the time-stamps of the first two samples in the file. The name of the resulting AO is set to the filename (without the file extension). The filename is also stored as a parameter in the history parameter list. The following code shows this in action:

```
        >> a = ao('data1.txt')
          + creating AO from text file data1.txt
        ----------- ao: a -----------

               tag:  -00001
              name:  data1
        provenance:  created by unknown@bob.local[192.168.2.100] on MACI/7.4 (R2007a)/0.2a
(R2007a) at 2007-06-23 19:46:05
           comment:
              data:  tsdata / data1
              hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
             mfile:
        ----------------------------
```

As with most constructor calls, an equivalent action can be achieved using an input Parameter List.

```
           >> a = ao(plist('filename', 'data1.txt'))
```

Creating AOs from XML or .mat files

AOs can be saved as both XML and .MAT files. As such, they can also be created from these files.

```
        >> a = ao('a.xml')
        ----------- ao: a -----------

               tag:  -00001
              name:  save(data1,a.xml)
        provenance:  created by unknown@bob.local[192.168.2.100] on MACI/7.4 (R2007a)/0.2a
(R2007a) at 2007-06-23 20:00:21
           comment:
              data:  tsdata / data1
              hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
             mfile:
        ----------------------------
```

Creating AOs from MATLAB functions

AOs can be created from any valid MATLAB function which returns a vector or matrix of values. For such calls, a parameter list is used as input. For example, the following code creates an AO containing 1000 random numbers:

```
        >> a = ao(plist('fcn', 'randn(1000,1)'))
        ----------- ao: a -----------

               tag:  -00001
              name:  randn(1000,1)
        provenance:  created by unknown@bob.local[192.168.2.100] on MACI/7.4 (R2007a)/0.2a
(R2007a) at 2007-06-23 20:04:52
           comment:
              data:  cdata / randn(1000,1)
              hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
             mfile:
        ----------------------------
```

Here you can see that the AO is a `cdata` type and the name is set to be the function that was input.

Creating AOs from functions of time

AOs can be created from any valid MATLAB function which is a function of the variable `t`. For such calls, a parameter list is used as input. For example, the following code creates an AO containing sinusoidal signal at 1Hz with some additional Gaussian noise:

```
        pl = plist();
        pl = append(pl, 'nsecs', 100);
        pl = append(pl, 'fs', 10);
        pl = append(pl, 'tsfcn', 'sin(2*pi*1*t)+randn(size(t))');
        a = ao(pl)
        ----------- ao: a -----------

               tag:  -00001
              name:  TSfcn
        provenance:  created by unknown@bob.local[192.168.2.100] on MACI/7.4 (R2007a)/0.2a
(R2007a) at 2007-06-24 06:47:19
           comment:
              data:  tsdata / sin(2*pi*1*t)+randn(size(t))
              hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
             mfile:
        ----------------------------
```

Here you can see that the AO is a `tsdata` type, as you would expect. Also note that you need to specify the sample rate (`fs`) and the number of seconds of data you would like to have (`nsecs`).

Creating AOs from window functions

The LTPDA Toolbox contains a class for designing spectral windows (see Spectral Windows). A spectral window object can also be used to create an Analysis Object as follows:

```
        >> w = specwin('Hanning', 1000)
        -------- Hanning ------------

        created:
           alpha: 0
            psll: 31.5
             rov: 50
           nenbw: 1.5
            w3db: 1.4382
        flatness: -1.4236
              ws: 500
             ws2: 375
             win: 1000

        -----------------------------

        >> a = ao(w)
        ----------- ao: a -----------

               tag:  -00001
              name:  Hanning
        provenance:  created by unknown@bob.local[192.168.2.100] on MACI/7.4 (R2007a)/0.2a
(R2007a) at 2007-06-24 10:27:18
           comment:
              data:  cdata / Hanning
              hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
             mfile:
        -----------------------------
```

The example code above creates a Hanning window object with 1000 points. The call to the AO constructor then creates a `cdata` type AO with 1000 points. This AO can then be multiplied against other AOs in order to window the data.

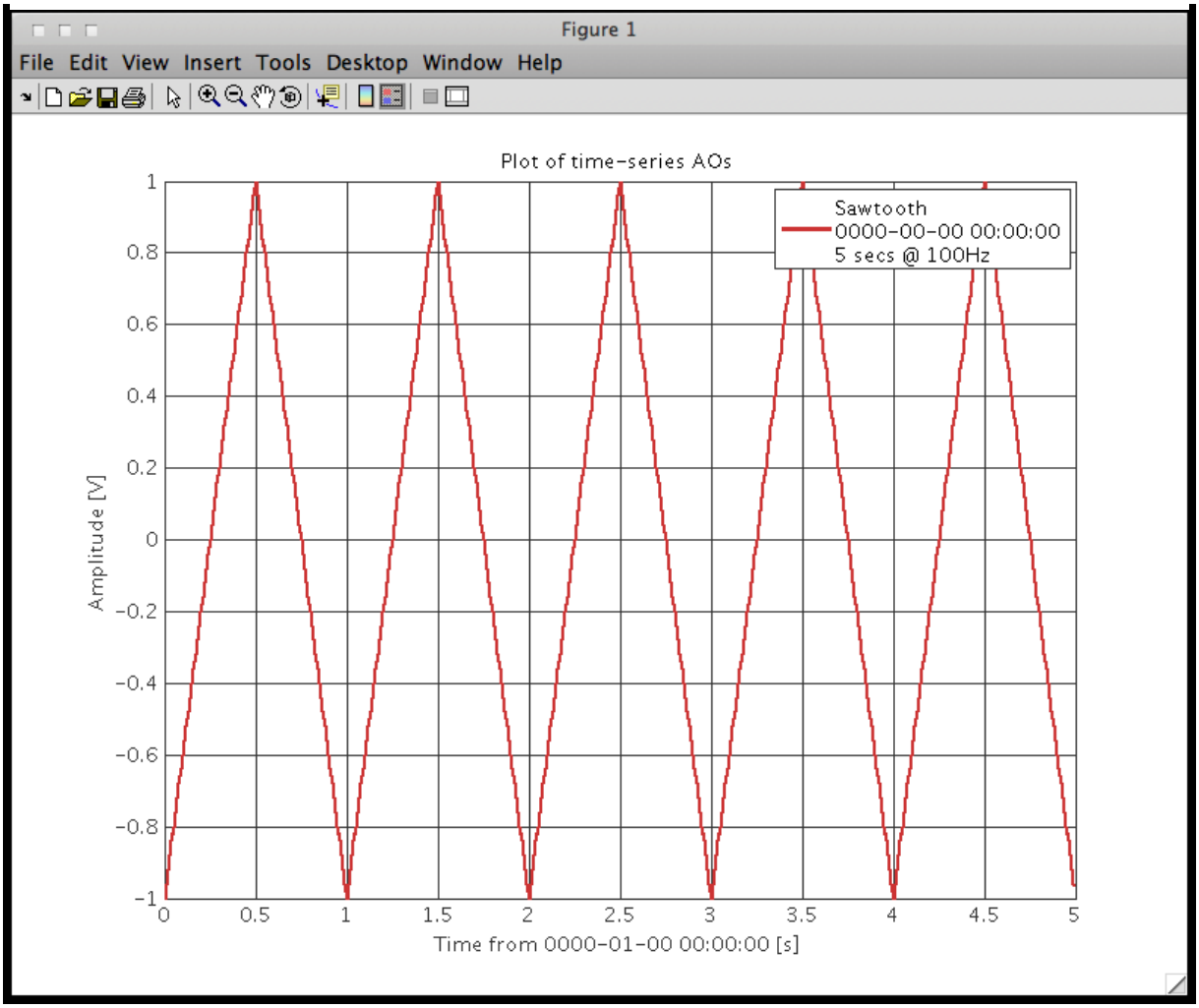
Creating AOs from waveform descriptions

MATLAB contains various functions for creating different waveforms, for example, `square`, `sawtooth`. Some of these functions can be called upon to create Analysis Objects. The following code creates an AO with a sawtooth waveform:

```
        pl = plist();
        pl = append(pl, 'fs', 100);
        pl = append(pl, 'nsecs', 5);
        pl = append(pl, 'waveform', 'Sawtooth');
        pl = append(pl, 'f', 1);
        pl = append(pl, 'width', 0.5);

        asaw = ao(pl)
        ----------- ao: asaw -----------

               tag:  -00001
              name:  Sawtooth
        provenance:  created by unknown@bob.local[192.168.2.100] on MACI/7.4 (R2007a)/0.2a
(R2007a) at 2007-06-24 10:37:51
           comment:
              data:  tsdata / sawtooth(2*pi*1*t,0.5)
              hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
             mfile:
        -------------------------------
```

You can call the `iplot` function to view the resulting waveform:

```
        iplot(asaw);
```



Analysis Objects | Converting existing data into Analysis Objects

Converting existing data into Analysis Objects

In some cases it is desirable to build AOs 'by hand' from existing data files. If the data file doesn't conform to one of the existing AO constructors, then a conversion script can easily be written to create AOs from your data files.

The following example shows how to convert an ASCII data file into Analysis Objects. The data file has 4 columns representing 4 different data channels. All 4 channels are sampled at the same rate of 10Hz. The conversion function returns 4 AOs.

```
 function b = myConverter(filename, fs)

% MYCONVERTER converts a multi-column ASCII data file into multiple AOs.
%
% usage: b = myConverter(filename, fs)
%
% Inputs:
%   filename   - name of input data file
%   fs         - sample rate of input data
%
% Outputs:
%   b    - vector of AOs
%
%

% Load the data from text file
data_in = load(filename);
ncols   = size(data_in, 2);

% Create AOs
b = [];
for j=1:ncols

% Get this column of data
  d = data_in(:,j);

% name for this data
  dataName = ['column' num2str(j)];

% Make tsdata object
  ts = tsdata(d, fs);
% set name of data object
  ts = set(ts, 'name', dataName);
% set xunits to seconds
  ts = set(ts, 'xunits', 's');
% set xunits to Volts
  ts = set(ts, 'yunits', 'V');

% make history object
  h = history('myConverter', '0.1', plist(param('filename', filename)));

% make AO
  a = ao(ts, h);

% set AO name
  a = set(a, 'name', dataName);

% add to output vector
  b = [b a];

end
```

The script works by loading the four columns of data from the ASCII file into a matrix. The script then loops over the number of columns and creates an AO for each column.

First a time-series (`tsdata`) object is made from the data in a column and given the input sample rate of the data. The 'name', 'xunits', and 'yunits' fields of the time-series data object are then set using calls to the `set` method of the `tsdata` class.

Next a history object is made with its 'name' set to 'myConverter', the version set to '0.1', and the input filename is recorded in a parameter list attached to the history object.

Following the creation of the data object and the history object, we can then create an Analysis Object. The name of the AO is then set and the object is added to a vector that is output by the function.

Creating Analysis Objects Saving Analysis Objects

Saving Analysis Objects

Analysis Objects can be saved to disk as either MATLAB binary files (.MAT) or as XML files (.XML). The following code shows how to do this:

```
save(aout, 'a.mat') % save AO aout to a .MAT file
save(aout, 'a.xml') % save AO aout to a .XML file
```

Converting existing data into Analysis Objects Plotting Analysis Objects

Plotting Analysis Objects

The data in an AO can be plotted using two different functions.

1. A simple plot function `plot`
2. An intelligent plotting function `iplot`

AO `plot` method

There are various calls to the simple `plot` method:

```
>> plot(a1)
```

Plots the data contained in the AO (`a1`) into the currently active axes.

As with the standard MATLAB `plot` function, you can pass line specifications to `ao/plot`. For example,

```
>> plot(a1, 'LineStyle', '--')
```

Plots the data contained in the AO (`a1`) into the currently active axes with a dashed line.

See `help plot` for further line specifications.

```
>> plot(axh, a1)
```

Plots the data contained in the AO (`a1`) into the axes specified by the given handle, `axh`.

You can get handles to the various plot elements as follows:

```
>> line_h                      = plot(a1)  % returns a handle to the line
objects
>> [line_h, axes_h]            = plot(a1)  % returns a handle to the line and
axes objects
>> [line_h, axes_h, figure_h] = plot(a1)  % returns a handle to the line,
axes and figure objects
```

AO `iplot` method

The `iplot` method provides a more advanced plotting interface for AOs which tries to make good use of all the information contained within the input AOs. For example, if the `xunits` and `yunits` fields of the input AOs are set, these labels are used on the plot labels.

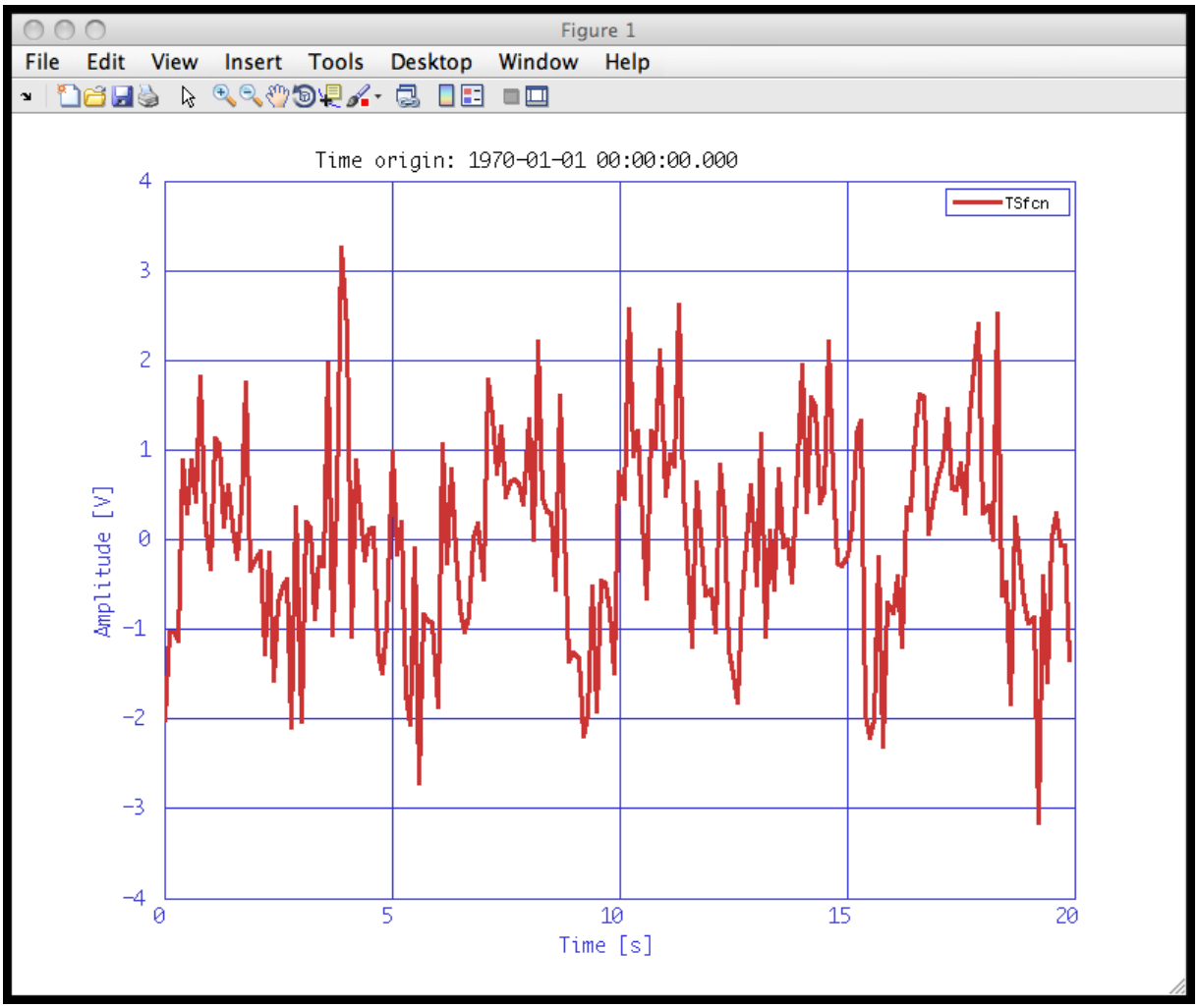
In addition, `iplot` can be configured using a input `plist`. The following examples show some of the possible ways to use `iplot`

```
>> a1 = ao(plist('tsfcn', 'sin(2*pi*0.3*t) + randn(size(t))', 'fs', 10,
'nsecs', 20))
---------- ao: a1 ----------

       name:  TSfcn
 provenance:  created by hewitson@bobmac-2.local[172.16.251.1] on MACI/7.6
(R2008a Prerelease)/0.99 (R2008a Prerelease) at 2008-02-29 18:54:12.127
description:
       data:  tsdata / sin(2*pi*0.3*t) + randn(size(t)) [200x1] | (0,-
2.00888) (0.1,-1.02877) (0.2,-1.02874) (0.3,-1.13014) (0.4,0.883107) ...
       hist:  history / ao / $Id: ao.m,v 1.89 2008/03/07 10:02:29 ingo Exp
  mfilename:
mdlfilename:
------------------------------

>> a1.data
-------- tsdata 01 ------------

  name:  sin(2*pi*0.3*t) + randn(size(t))
    fs:  10
     x:  [200 1], double
     y:  [200 1], double
xunits:  s
yunits:  V
 nsecs:  20
    t0:  1970-01-01 00:00:00.000
-------------------------------
```

Creates a time-series AO. If we look at the data object contained in this AO, we see that the `xunits` and `yunits` are set to the defaults of seconds [s] and Volts [V].

If we plot this object with `iplot` we see these units reflected in the x and y axis labels.

```
>> iplot(a1)
```



We also see that the time-origin of the data (`t0` field of the `tsdata` class) is displayed as the plot

title.

Saving Analysis Objects Parameter Lists

Parameter Lists

Any algorithm that requires input parameters to configure its behaviour should take a Parameter List (`plist`) object as input. A `plist` object contains a vector of Parameter (`param`) objects.

The following sections introduce parameters and parameter lists:

Creating Parameters

Parameter objects are used in the LTPDA Toolbox to configure the behaviour of algorithms. A parameter (`param`) object has two main properties:

- `'key'` — The parameter name
- `'val'` — The parameter value

See param class for further details. The 'key' property is always stored in upper case. The 'value' of a parameter can be any LTPDA object, as well as most standard MATLAB types.

Parameter values can take any form: vectors or matrices of numbers; strings; other objects, for example a `specwin` (spectral window) object.

Parameters are created using the `param` class constructor. The following code shows how to create a parameter 'a' with a value of 1

```
>> p = param('a', 1)
---- param 1 ----
key: a
val: 1
-----------------
```

The contents of a parmeter object can be accessed as follows:

```
>> key = p.key;  % get the parameter key
>> val = p.val;  % get the parameter value
```

Parameter Lists Creating lists of Parameters

Creating lists of Parameters

Parameters can be grouped together into parameter lists (`plist`).

Creating parameter lists from parameters.

The following code shows how to create a parameter list from individual parameters.

```
>> p1 = param('a', 1);  % create first parameter
>> p2 = param('b', specwin('Hanning', 100));  % create second parameter
>> pl = plist([p1 p2])  % create parameter list
----------- plist 01 -----------
n params: 2
---- param 1 ----
key: A
val: 1
-----------------
---- param 2 ----
key: B
val: specwin
-------- Hanning ------------

alpha: 0
psll: 31.5
rov: 50
nenbw: 1.5
w3db: 1.4382
flatness: -1.4236
ws: 50
ws2: 37.5
win: 100

-----------------------------


-----------------
--------------------------------
```

Creating parameter lists directly.

You can also create parameter lists directly using the following constructor format:

```
  >> pl = plist('a', 1, 'b', 'hello')
  ----------- plist 01 -----------
  n params: 2
  ---- param 1 ----
  key: A
  val: 1
  -----------------
  ---- param 2 ----
  key: B
  val: 'hello'
  -----------------
--------------------------------
```

Appending parameters to a parameter list.

Additional parameters can be appended to an existing parameter list using the `append` method:

```
>> pl  = append(pl, param('c', 3))  % append a third parameter
---------- plist 01 ----------
n params: 3
---- param 1 ----
key: A
val: 1
-----------------
---- param 2 ----
key: B
val: 'hello'
-----------------
---- param 3 ----
key: C
val: 3
-----------------
--------------------------------
```

Finding parameters in a parameter list.

Accessing the contents of a `plist` can be achieved in two ways:

```
>> p1  = pl.params(1);   % get the first parameter
>> val = find(pl, 'b');  % get the second parameter
```

If the parameter name ('key') is known, then you can use the `find` method to directly retrieve the value of that parameter.

Removing parameters from a parameter list.

You can also remove parameters from a parameter list:

```
>> pl = remove(pl, 2) % Remove the 2nd parameter in the list
>> pl = remove(pl, 'a') % Remove the parameter with the key 'a'
```

Setting parameters in a parameter list.

You can also set parameters contained in a parameter list:

```
>> pl = plist('a', 1, 'b', 'hello')
>> pl = pset(pl, 'a', 5, 'b', 'ola');  % Change the values of the parameter with the keys 'a' and 'b'
```

Combining multiple parameter lists.

Parameter lists can be combined:

```
>> pl = combine(pl1, pl2)
```

If `pl1` and `pl2` contain a parameter with the same key name, the output `plist` contains a parameter with that name but with the value from the first parameter list input.

Spectral Windows

Spectral windows are an essential part of any spectral analysis. As such, great care has been taken to implement a complete and accurate set of window functions. The window functions are implemented as a class `specwin`. The properties of the class are given in specwin class.

The following pages describe the implementation of spectral windows in the LTPDA framework:

- What are LTPDA spectral windows?
- Creating spectral windows
- Visualising spectral windows
- Using spectral windows

Creating lists of Parameters | What are LTPDA spectral windows?

What are LTPDA spectral windows?

MATLAB already contains a number of window functions suitable for spectral analysis. However, these functions simply return vectors of window samples; no additional information is given. It is also desirable to have more information about a window function, for example, its normalised equivalent noise bandwidth (NENBW), its peak side-lobe level (PSLL), and its recommended overlap (ROV).

The specwin class implements many window functions as class objects that contain many descriptive properties. The following table lists the available window functions and some of their properties:

Window name	NENBW	PSLL [dB]	ROV [%]
Rectangular	1.000	-13.3	0.0
Welch	1.200	-21.3	29.3
Bartlett	1.333	-26.5	50.0
Hanning	1.500	-31.5	50.0
Hamming	1.363	-42.7	50.0
Nuttall3	1.944	-46.7	64.7
Nuttall4	2.310	-60.9	70.5
Nuttall3a	1.772	-64.2	61.2
Nuttall3b	1.704	-71.5	59.8
Nuttall4a	2.125	-82.6	68.0
Nuttall4b	2.021	-93.3	66.3
Nuttall4c	1.976	-98.1	65.6
BH92	2.004	-92.0	66.1
SFT3F	3.168	-31.7	66.7
SFT3M	2.945	-44.2	65.5

FTNI	2.966	-44.4	65.6
SFT4F	3.797	-44.7	75.0
SFT5F	4.341	-57.3	78.5
SFT4M	3.387	-66.5	72.1
FTHP	3.428	-70.4	72.3
HFT70	3.413	-70.4	72.2
FTSRS	3.770	-76.6	75.4
SFT5M	3.885	-89.9	76.0
HFT90D	3.883	-90.2	76.0
HFT95	3.811	-95.0	75.6
HFT116D	4.219	-116.8	78.2
HFT144D	4.539	-114.1	79.9
HFT169D	4.835	-169.5	81.2
HFT196D	5.113	-196.2	82.3
HFT223D	5.389	-223.0	83.3
HFT248D	5.651	-248.0	84.1

In addition to these 'standard' windows, Kaiser windows can be designed to give a chosen PSLL.

Spectral Windows Create spectral windows

Create spectral windows

To create a spectral window object, you call the `specwin` class constructor. The following code fragment creates a 100-point Hanning window:

```
>> w = specwin('Hanning', 100)
-------- Hanning ------------

   alpha: 0
    psll: 31.5
     rov: 50
   nenbw: 1.5
    w3db: 1.4382
flatness: -1.4236
      ws: 50
     ws2: 37.5
     win: 100

-----------------------------
```

List of available window functions

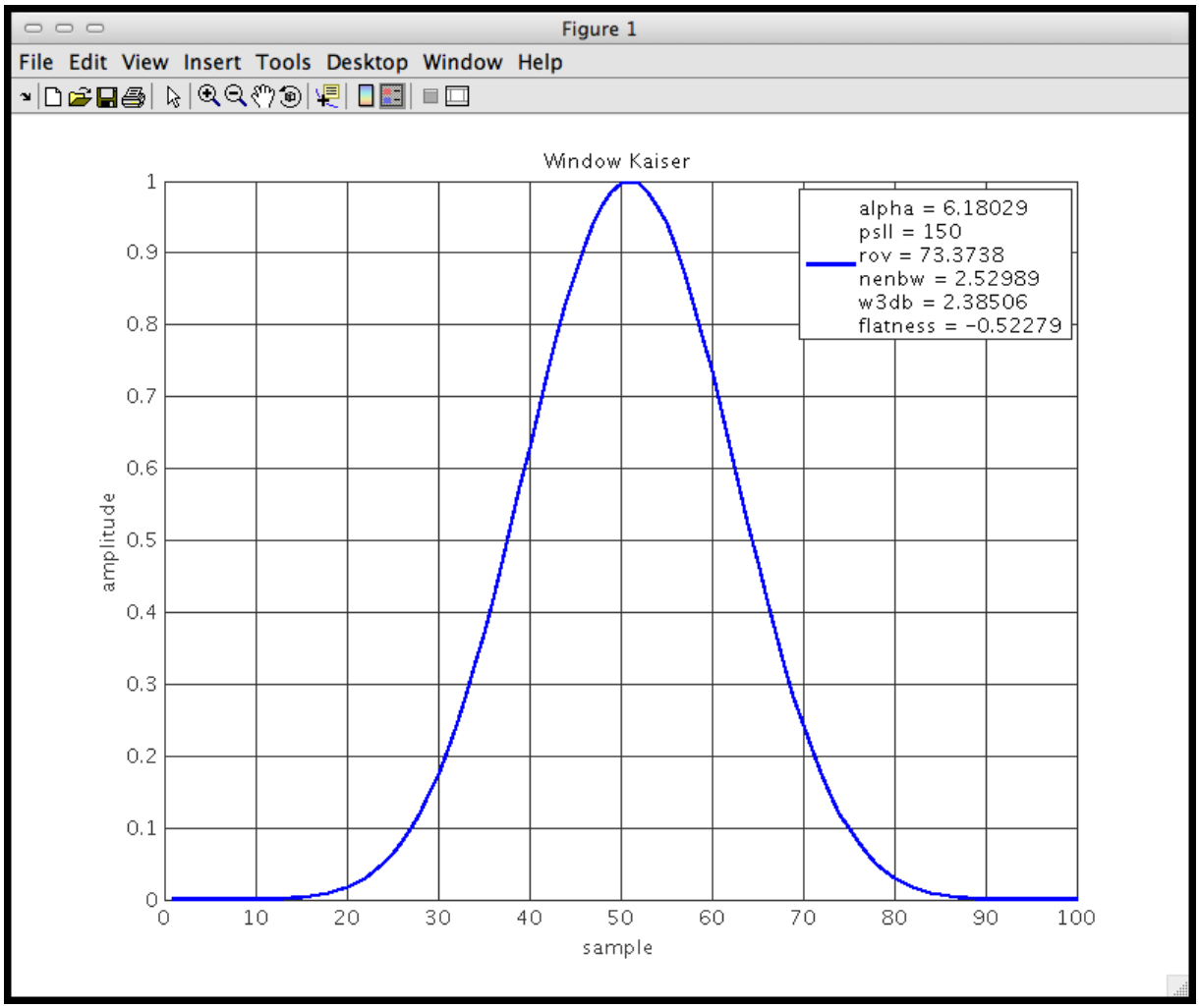
In the special case of creating a Kaiser window, the additional input parameter, PSLL, must be supplied. For example, the following code creates a 100-point Kaiser window with -150dB peak side-lobe level:

```
>> w = specwin('Kaiser', 100, 150)
-------- Kaiser ------------

   alpha: 6.18029
    psll: 150
     rov: 73.3738
   nenbw: 2.52989
    w3db: 2.38506
flatness: -0.52279
      ws: 28.2558
     ws2: 20.1819
     win: 100

-----------------------------
```

Visualising spectral windows

The specwin class has a plot method which will plot the response of the given window function in the current Figure:

```
w = specwin('Kaiser', 100, 150);
figure
plot(w)
```



Windows can also be visualised using the Spectral Window Viewer.

Create spectral windows Using spectral windows

Using spectral windows

Spectral windows are typically used in spectral analysis algorithms. In all LTPDA spectral analysis functions, spectral windows are specified as parameters in an input parameter list. The following code fragment shows the use of `ltpda_pwelch` to estimate an Amplitude Spectral Density of the time-series captured in the input AO, `a_in`. The help for `ltpda_pwelch` reveals that the required parameter for setting the window function is 'Win'.

```
w = specwin('Kaiser', 100, 150);
pl = plist(param('Win', w))
axx = ltpda_pwelch(a_in, pl);
```

In this case, the size of the spectral window (number of samples) may not match the length of the segments in the spectral estimation. The `ltpda_pwelch` algorithm then recomputes the window using the input design but for the correct length of window function.

Spectral windows can also be used more directly by first converting them to Analysis Objects. The following code fragment converts a `specwin` object to an Analysis Object. This AO is then multiplied against another time-series AO to window the data.

```
a = ao('data1.txt')
w  = specwin('Kaiser', len(a), 150);
wa = ao(w);
a_win = a.*wa;
```

Note here that the `len` method of the AO class is used to produce a window function that is of the same length as the time-series contained in `a`.

Visualising spectral windows Simulation/modelling

Simulation/modelling

Content needs written...

Using spectral windows Generating model noise

Generating model noise

Generating non-white random noise means producing arbitrary long time series with a given spectral density. Such time series are needed for example for the following purposes:

- To generate test data sets for programs that compute spectral densities,
- as inputs for various simulations.

One way of doing this is to apply digital filters (FIR or IIR) to white input noise.
This approach is often used because of its simplicity but is has some disadvantages:
For complicated spectra a matching filter is not trivial to find. The filter has to be split into several simple sections. Via nonlinear optimization techniques optimized filters can be found. Those techniques however are non deterministic such that their convergence is not guaranteed. So although the resulting filter transfer function is well known, it may not perfectly match the given spectrum.
Moreover the filtering method requires a 'warm-up period'

A different approach is implemented in LTPDA as Franklin noise-generator.
It produces spectral densities according to a given pole zero model (see Pole/Zero Modeling) and does not require any warm-up period.

Simulation/modelling Franklin noise-generator

Franklin noise-generator

The following sections gives an introduction to the generation of model noise using the noise generator implemented in LTPDA.

Franklin's noise generator

Franklin's noise generator is a method to generate arbitrarily long time series with a prescribed spectral density. The algorithm is based on the paper 'Numerical simulation of stationary and non-stationary gaussian random processes' by Franklin, Joel N. (SIAM review, Volume 7, Issue 1, page 68-80, 1956)
See Generating model noise for more general information on this.

Franklin's method does not require any 'warm up' period. It starts with a transfer function given as ratio of two polynomials.
The generator operates on a real state vector y of length n which is maintained between invocations. It produces samples of the time series in equidistant steps `T = 1/fs`, where `fs` is the sampling frequency.

- y0 = Tinit * r, on initialization
- yi = E * yi-1 + Tprop * r, to propagate
- xi = a * yi , the sampled time series.

r is a vector of independent normal Gaussian random numbers Tinit, E, Tprop which are real matrices and a which is a real vector are determined once by the algorithm.

Description

ltpda_noisegen uses `pzm2ab.m` and `franklin.m` to generate a time-series from a given `pzmodel`

Call

```
>> [b, pl1, pl2] = ltpda_noisegen(pl)
>> [b, pl1] = ltpda_noisegen(pl)
>>  b = ltpda_noisegen(pl1, pl2)
```

Inputs

for the first function call the parameter list has to contain at least:

- nsecs - number of seconds (length of time series)

- fs - sampling frequency
- pzmodel with gain

Outputs

- b - analysis object containing the resulting time series
- pl1 - parameter list containing the last state vector y.
- pl2 - parameter list containing the following parameters:
  - Tinit - matrix to calculate initial state vector
  - Tprop - matrix to calculate propagation vector
  - E - matrix to calculate propagation vector
  - num - numerator coefficients
  - den - denominator coefficients

Usage

ltpda_noisegen consists of the following four main functions.

- ngconv
- ngsetup
- nginit
- ngprop

Depending on the Inputs different functions are called by ltpda_noisegen.
First a parameter list of the input parameters is to be done. For further information on this look at Creating parameter lists from parameters.

Case 1: Starting from a given pole/zero model

The most common case would be to start from a pole zero model `pzm`, the number of seconds the resulting time series should have `nsecs` and the sampling frequency `fs`. In this case all of the four main functions mentioned above are called one after the other.
The function call can then look like this:

```
>> [b, pl1, pl2] = ltpda_noisegen(pl)
```

The output will be an analysis object `b` containing the time series and two parameter lists (pl1, pl2). The first (pl1) contains the state vector `y` that can be used as input to generate a second time series with the same characteristics. The second (pl2) contains:

- the matrices
  - Tinit
  - Tprop
  - E
- the coefficients of transfer function:
  - num
  - den
- the input parameters
  - gain
  - fs
  - nsecs

Case 2: Starting from given matrices

The matrices `Tinit, Tprop and E` can be known from a previous calculation (i.e ltpda_noisegen,

LISO).
These matrices can be stored as parameters of a parameter list and inputted into ltpda_noisegen.
In this case the function starts the operation with calling nginit first and finally ngprop.
The output will be the same as for case 1.

Case 3: Starting from a given state vector and matrices

In this case a state vector y known from a previous run of the function is given as input together with the matrices.
ltpda_noisegen will now only call ngprop.
The output will again be the same as above.

Generating model noise Pole/Zero Modelling

Pole/Zero Modelling

Pole/zero modelling is implemented in the LTPDA Toolbox using three classes: a pole class, a zero class, and a pole/zero model class (pzmodel class).

The following pages introduce how to produce and use pole/zero models in the LTPDA environment.

- Creating poles and zeros
- Building a model
- Model helper GUI
- Converting models to IIR filters

Franklin noise-generator Creating poles and zeros

Creating poles and zeros

Poles and zeros are treated the same with regards creation, so we will look here at poles only. The meaning of a pole and a zero only becomes important when creating a pole/zero model.

Poles are specified by in the LTPDA Toolbox by a frequency, *f*, and (optionally) a quality factor, *Q*.

The following code fragment creates a real pole at 1Hz:

```
>> p1 = pole(1)
---- pole 1 ----
real pole: 1 Hz
----------------
```

To create a complex pole, you can specify a quality factor. For example,

```
>> p1 = pole(10, 4)
---- pole 1 ----
complex pole: 10 Hz, Q=4 [-0.0019894+0.015791i]
----------------
```

creates a complex pole at 10Hz with a *Q* of 4. You can also see that the complex representation is also shown, but only one part of the conjugate pair.

Pole/Zero Modelling Building a model

Building a model

Poles and zeros can be combined together to create a pole/zero model. In addition to a list of poles and zeros, a gain factor can be specified such that the resulting model is of the form:

$$H(s) = G\frac{(s-z_1)(s-z_2)...}{(s-p_1)(s-p_2)...}$$

The following sections introduce how to produce and use pole/zero models in the LTPDA environment.

Direct form

The following code fragment creates a pole/zero model consisting of 2 poles and 2 zeros with a gain factor of 10:

```
>> poles = [pole(1,2) pole(40)];
>> zeros = [zero(10,3) zero(100)];
>> pzm   = pzmodel(10, poles, zeros)
---- pzmodel 1 ----
model: pzmodel
gain: 10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
zero 002: zero(100)
--------------------
```

Creating from a `plist`

You can also create a `pzmodel` by passing a parameter list. The following example shows this

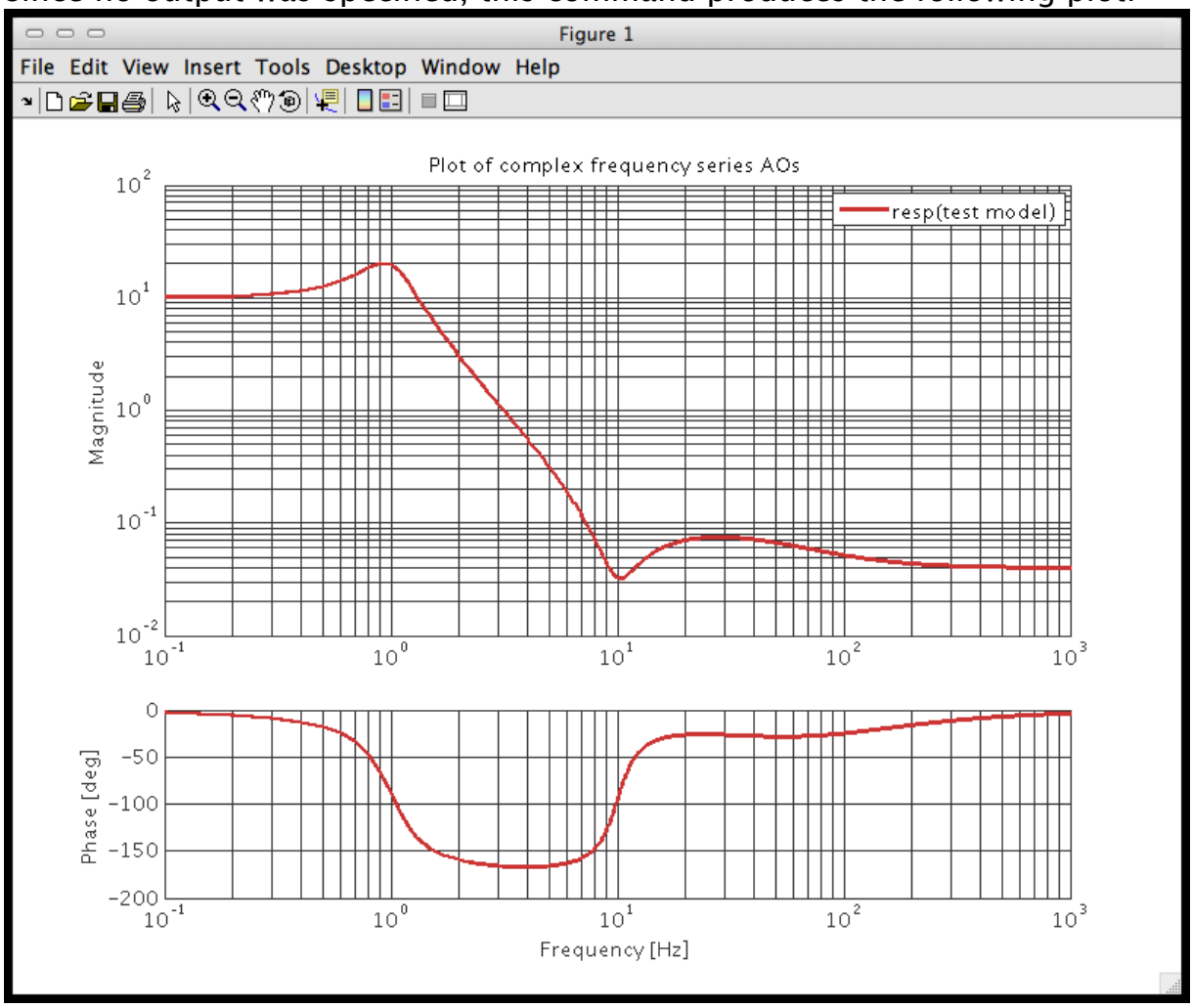
```
>> pl  = plist('name', 'test model', ...
                'gain', 10, ...
            'poles', [pole(1,2) pole(40)], ...
            'zeros', [zero(10,3) zero(100)]);
>> pzm = pzmodel(pl)
---- pzmodel 1 ----
model: test model
gain: 10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
zero 002: zero(100)
--------------------
```

Computing the response of the model

The frequency response of the model can generated using the `resp` method of the `pzmodel` class. To compute the response of the model created above:

```
>> resp(pzm)
```

Since no output was specified, this command produces the following plot:



You can also specify the frequency band over which to compute the response by passing a `plist` to the `resp` method, as follows:

```
        >> rpl = plist('f1', 0.1, ...
                       'f2', 1000, ...
                       'nf', 10000);
        >> a   = resp(pzm, rpl)
        ----------- ao: a -----------

                tag:  -00001
               name:  resp(test model)
         provenance:  created by unknown@42.104.179.202.in-addr.arpa[202.179.104.42] on
MACI/7.4 (R2007a)/0.3 (R2007a) at 2007-07-05 09:56:47
            comment:
               data:  fsdata / resp(test model)
               hist:  history / resp / $Id: resp.m,v 1.16 2008/02/24 21:43:59 mauro Exp
              mfile:
        ----------------------------
```

In this case, the response is returned as an Analysis Object containing `fsdata`. You can now plot the AO using the `iplot` function.

Creating poles and zeros | Model helper GUI

Model helper GUI

A simple GUI exists to help you build pole/zero models. To start the GUI, type

```
>> pzmodel_helper
```

More details on the PZModel Helper GUI.

 Building a model Converting models to IIR filters 

Converting models to IIR filters

Pole/zero models can be converted to IIR filters using the bilinear transform. The result of the conversion is an `miir` object. To convert a model, you call the constructor of the `miir` class with a `plist` input:

```
>> filt = miir(plist('pzmodel', pzm))
```

If no sample rate is specified, then the conversion is done for a sample rate equal to 8 times the highest pole or zero frequency. You can set the sample rate by specifying it in the parameter list:

```
>> filt = miir(plist('pzmodel', pzm, 'fs', 1000))
```

For more information of IIR filters in LTPDA, see IIR Filters.

Model helper GUI Signal Pre-processing in LTPDA

Signal Pre-processing in LTPDA

Signal pre-processing in LTPDA consists on a set of functions intended to pre-process data prior to further analysis. Pre-processing tools are focused on data sampling rates manipulation, data interpolation, spike cleaning and gap filling functions.

The following pages describe the different pre-processing tools available in the LTPDA toolbox:

Converting models to IIR filters Downsampling data

Downsampling data

Downsampling is the process of reducing the sampling rate of a signal. `Downsample` reduces the sampling rate of the input AOs by an integer factor by picking up one out of N samples. Note that no anti-aliasing filter is applied to the original data. Moreover, a `offset` can be specified, i.e., the sample at which the output data starts ---see examples below.

Syntaxis

```
b = downsample(a, pl)
```

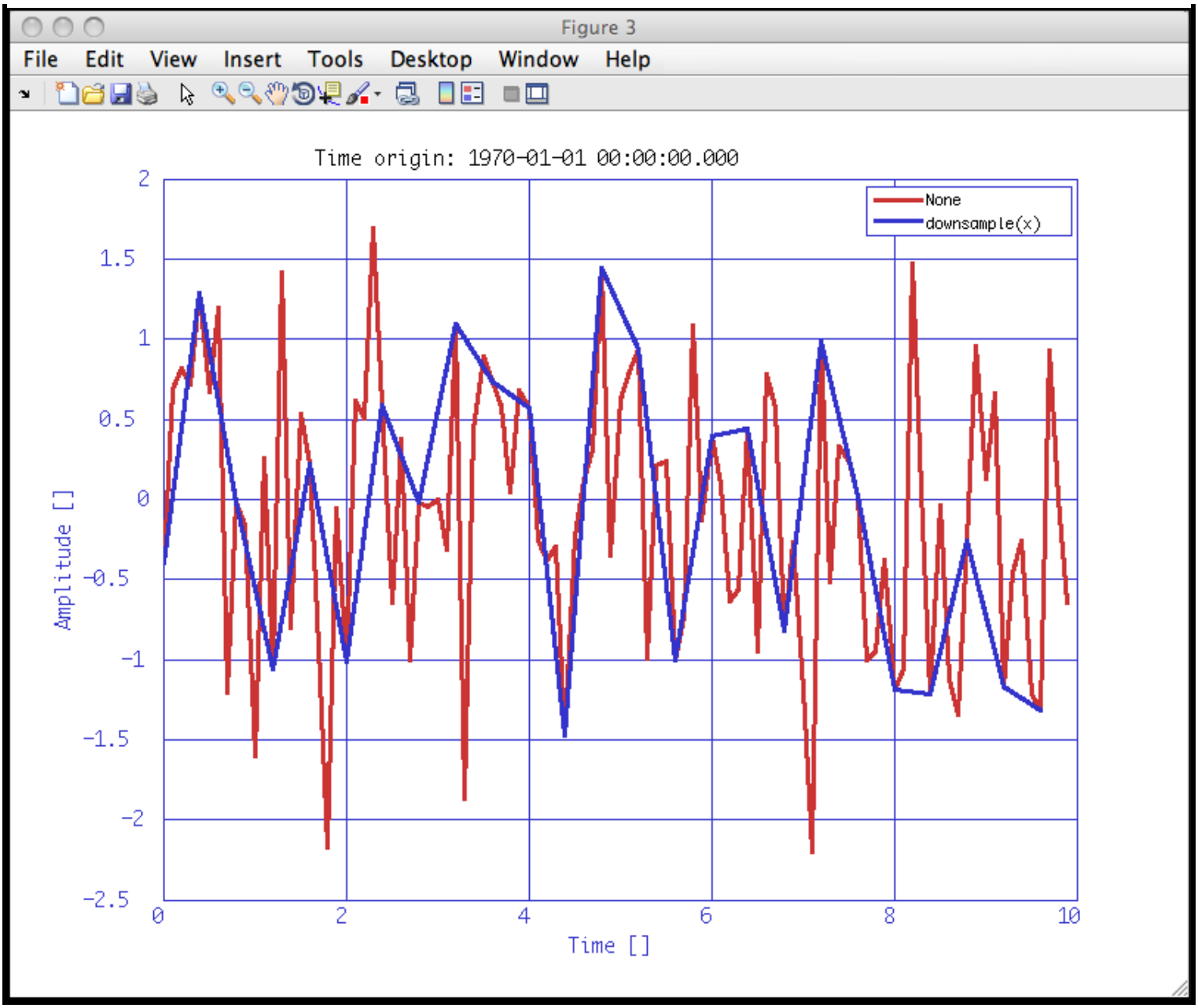
With the following parameters:

- `'factor'` - decimation factor [by default is 1: no downsampling] (must be an integer)
- `'offset'` - sample offset for decimation

Examples

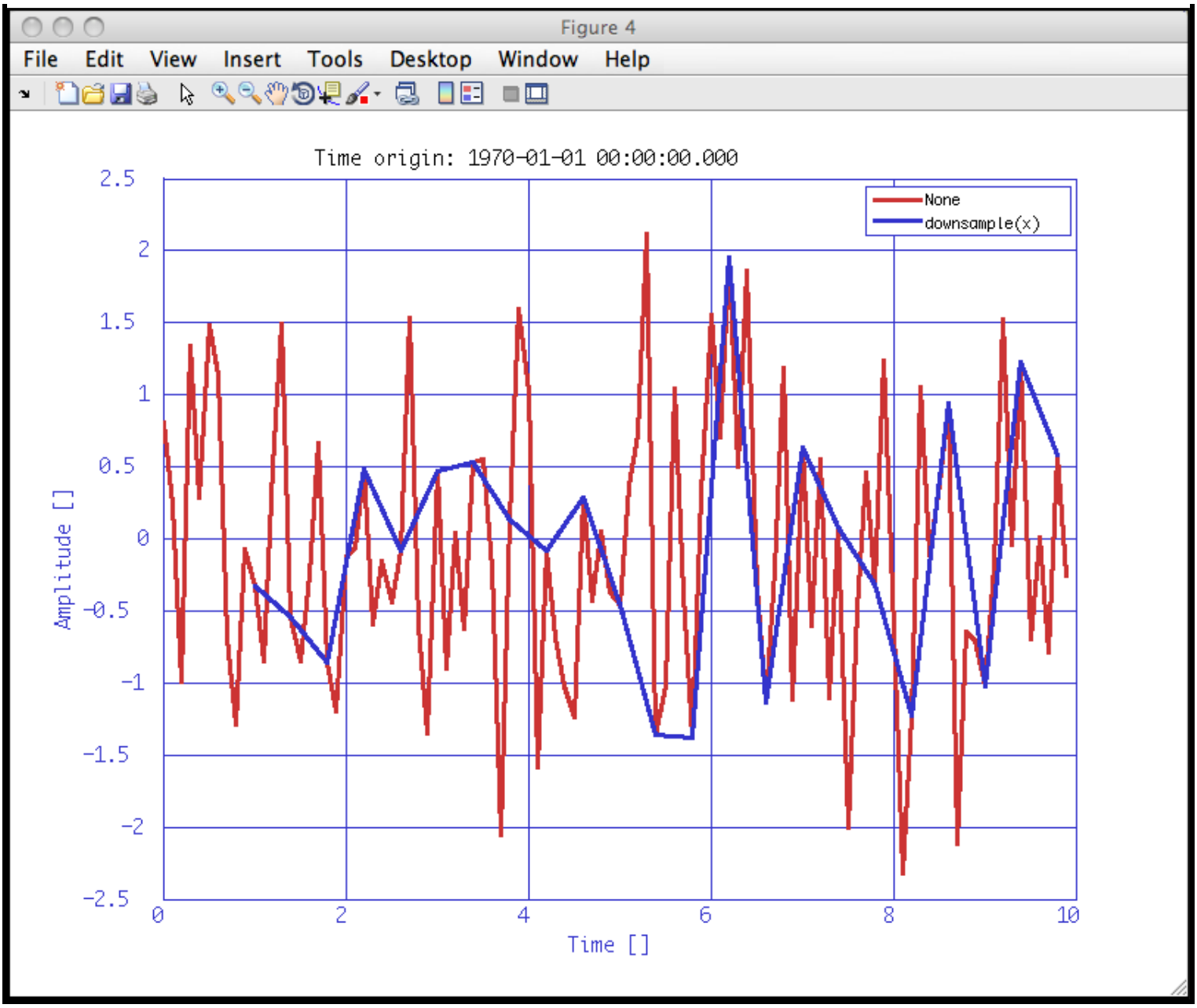
1. Downsampling a sequence of random data at original sampling rate of 10 Hz by a factor of 4 (`fsout` = 2.5 Hz) and no `offset`.

```
x=ao(tsdata(randn(100,1),10)); % create an AO of random data with fs = 10 Hz
pl = plist(); % create an empty parameters list
pl = append(pl, param('factor', 4)); % add the decimation factor
y = downsample(x, pl); % downsample the input AO, x
iplot(x, y) % plot original,x, and decimated,y, AOs
```



2. Downsampling a sequence of random data at original sampling rate of 10 Hz by a factor of 4 (`fsout` = 2.5 Hz) and `offset` = 10.

```
x=ao(tsdata(randn(100,1),10)); % create an AO of random data with fs = 10 Hz.
pl = plist(); % create an empty parameters list
pl = append(pl, param('factor', 4)); % add the decimation factor
pl = append(pl, param('offset', 10)); % add the offset parameter
y = downsample(x, pl); % downsample the input AO, x
iplot(x, y) % plot original,x, and decimated,y, AOs
```



Signal Pre-processing in LTPDA Upsampling data

Upsampling data

Upsampling is the process of increasing the sampling rate of a signal. Upsample increases the sampling rate of the input AOs by an integer factor. LTPDA upsample overloads upsample function from Matlab Signal Processing Toolbox. This function increases the sampling rate of a signal by inserting (n-1) zeros between samples. The upsampled output has (n*input) samples. In addition, an initial phase can be specified and, thus, a delayed output of the input can be obtained by using this option.

Syntaxis

```
b = upsample(a, pl)
```

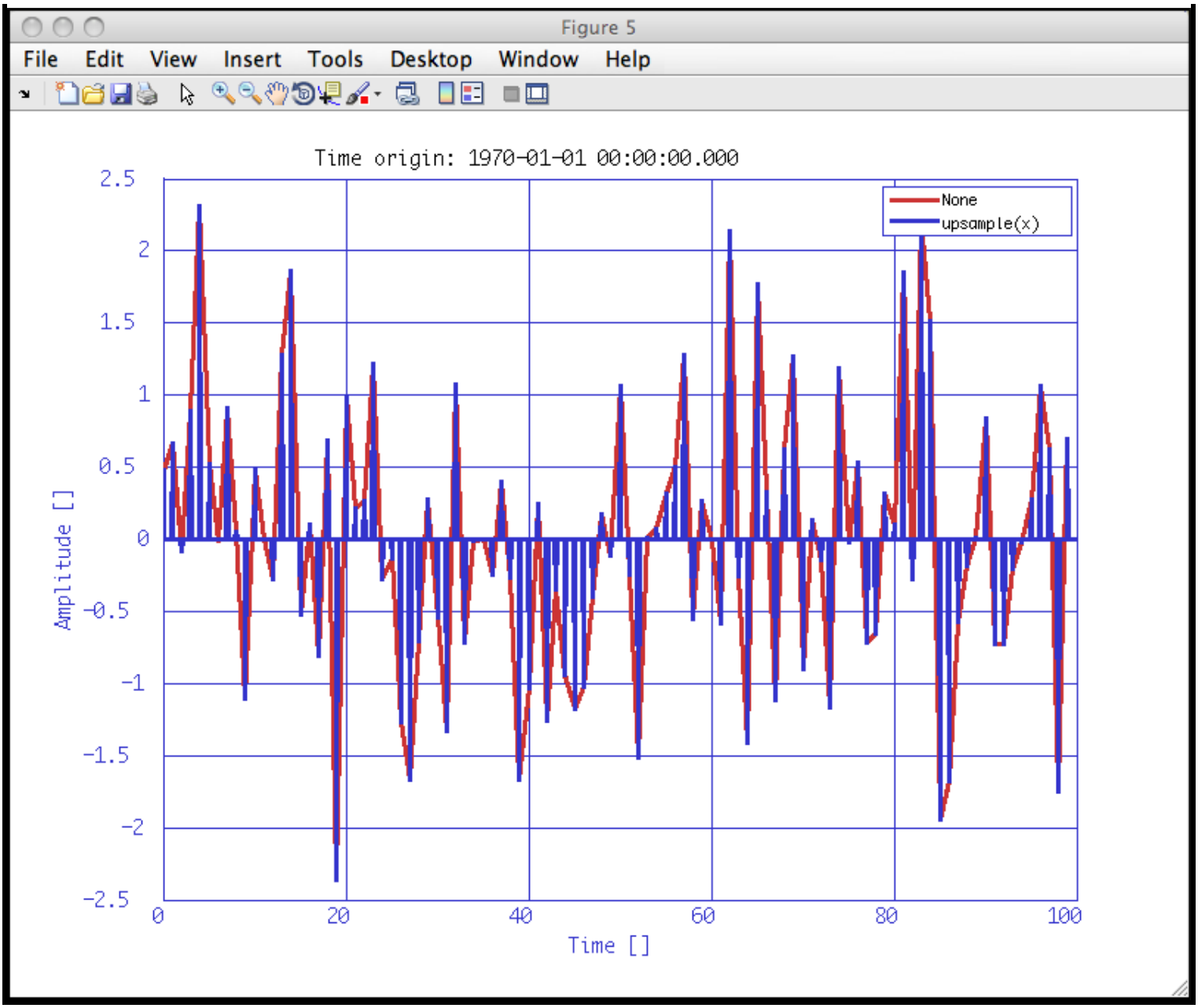
With the following parameters:

- 'N' - specify the desired upsample rate
- 'phase' - specify an initial phase range [0, N-1]

Examples

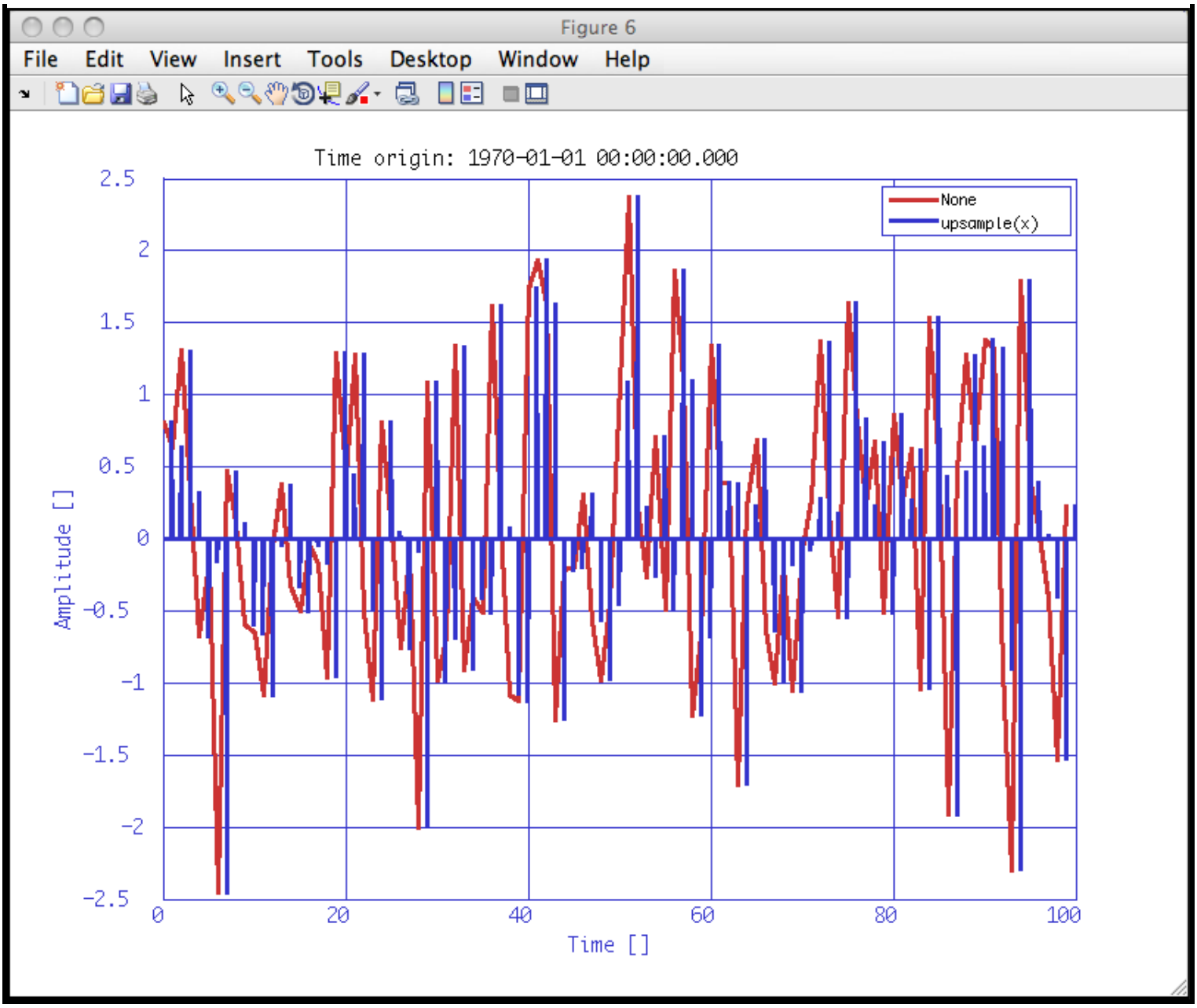
1. Upsampling a sequence of random data at original sampling rate of 1 Hz by a factor of 10 with no initial phase.

```
x=ao(tsdata(randn(100,1),1)); % create an AO of random data sampled at 1 Hz.
pl = plist(); % create an empty parameters list
pl = append(pl, param('N', 10)); % increase the sampling frequency by a factor of
10
y = upsample(x, pl); % resample the input AO (x) to obtain the upsampled AO (y)
iplot(x, y) % plot original and upsampled data
```



2. Upsampling a sequence of random data at original sampling rate of 1 Hz by a factor of 21 with a phase of 20 samples.

```
                x=ao(tsdata(randn(100,1),1)); % create an AO of random data sampled at 1 Hz.
                pl = plist(); % create an empty parameters list
                pl = append(pl, param('N', 21)); % increase the sampling frequency by a factor of
21
                pl = append(pl, param('phase', 20)); % add phase of 20 samples to the upsampled
data
                y = upsample(x, pl); % resample the input AO (x) to obtain the upsampled and
delayed AO (y)
                iplot(x, y) % plot original and upsampled data
```



Downsampling data Resampling data

Resampling data

Resampling is the process of changing the sampling rate of data. `Resample` changes the sampling rate of the input AOs to the desired output sampling frequency. LTPDA `resample` overloads `resample` function of Matlab Signal Processing Toolbox for AOs.

```
b = resample(a, pl)
```

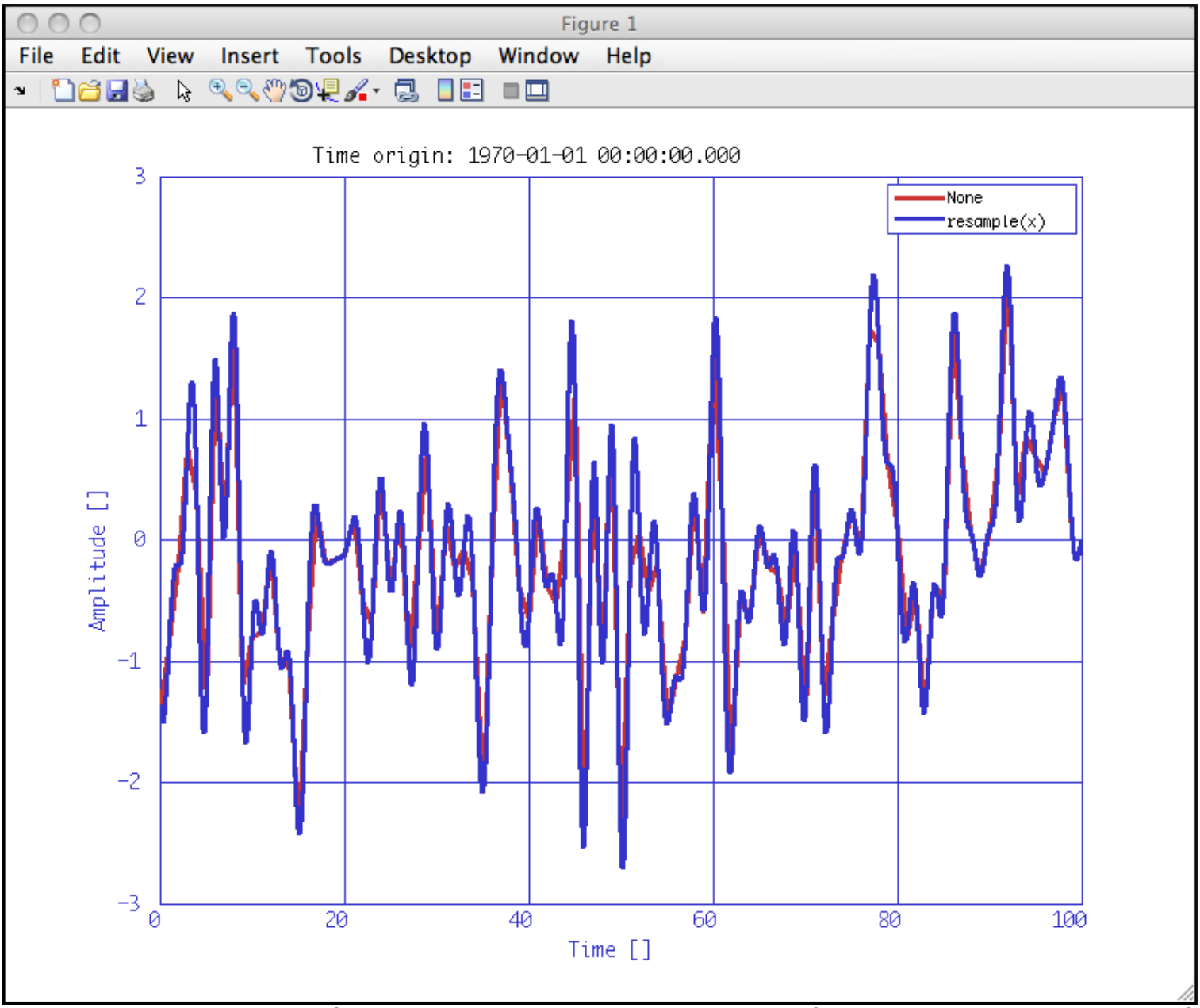
With the following parameters:

- `'fsout'` - specify the desired output frequency (must be positive and integer)
- `'filter'` - specified filter applied to the input, `a`, in the resampling process

Examples

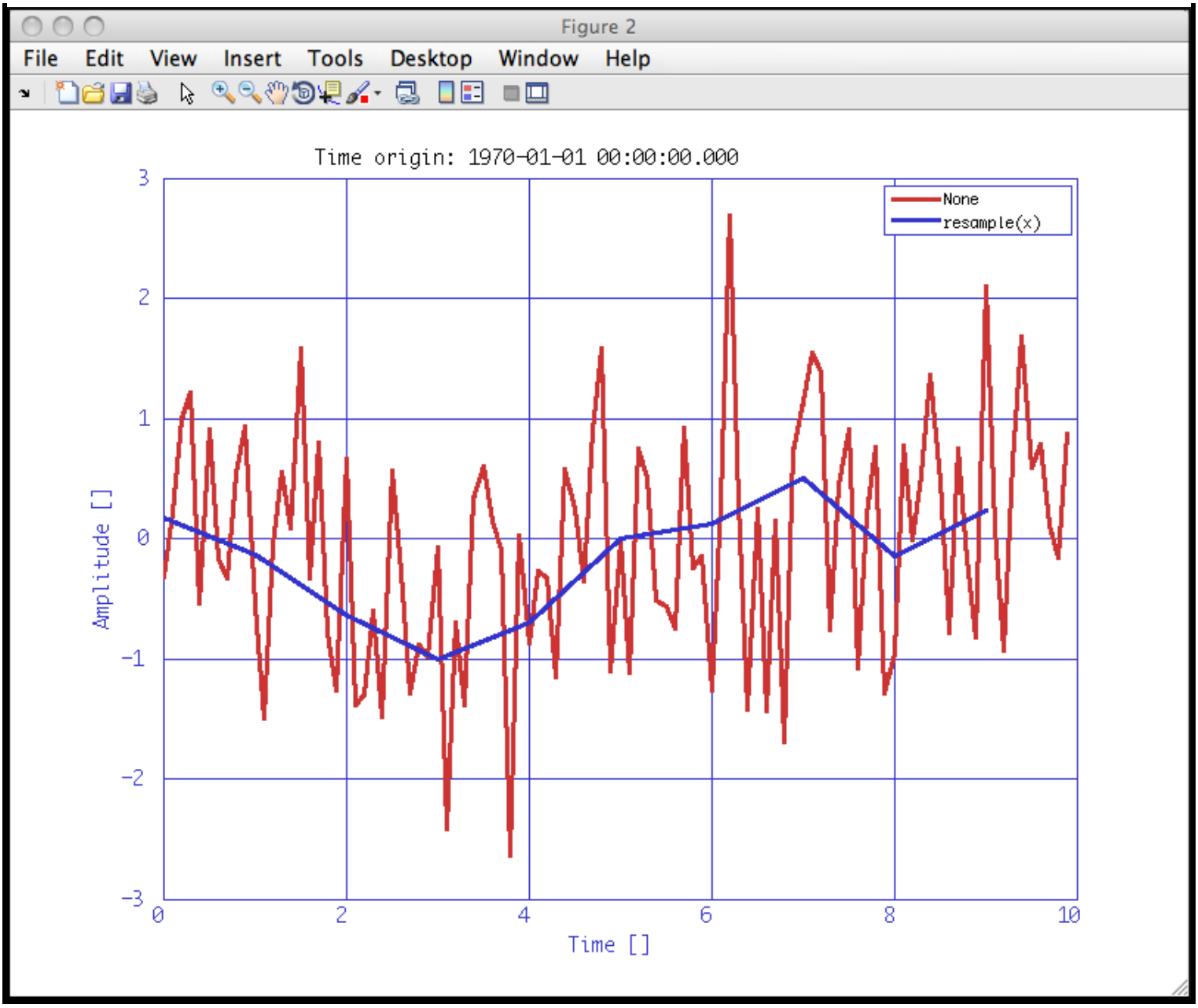
1. Resampling a sequence of random data at original sampling rate of 1 Hz at an output sampling of 50 Hz.

```
x=ao(tsdata(randn(100,1),1)); % create AO of random data with fs = 1 Hz.
pl = plist(); % create an empty parameters list
pl = append(pl, param('fsout', 50)); % add fsout = 50 Hz to parameters list
y = resample(x, pl); % resample the input AO (x) to obtain the resampled output AO
(y)
iplot(x, y) % plot original and resampled data
```



1. Resampling a sequence of random data at original sampling rate of 10 Hz at an output sampling of 1 Hz with a filter defined by the user.

```
x=ao(tsdata(randn(100,1),10)); % create AO of random data with fs = 10 Hz.


% filter definition
plfilter = plist(); % create an empty parameters list for the filter
plfilter = append(plfilter, param('type','lowpass'));
plfilter = append(plfilter, param('Win', specwin('Kaiser', 10, 150)));
plfilter = append(plfilter, param('order', 32));
plfilter = append(plfilter, param('fs', 10));
plfilter = append(plfilter, param('fc', 1));
filter  = mfir(plfilter)


% resampling
pl = plist(); % create an empty parameters list
pl = append(pl, param('fsout', 1)); % add fsout = 50 Hz to parameters list
pl = append(pl, param('filter', filter)); % use the defined filter in the
resampling process
y = resample(x, pl);  % resample the input AO (x) to obtain the resampled output
AO (y)
iplot(x, y) % plot original and resampled data
```



Upsampling data

Interpolating data

Interpolating data

Interpolation of data can be done in the LTPDA Toolbox by means of `interp`. This function interpolates the values in the input AO(s) at new values specified by the input parameter list. `Interp` overloads `interp1` function of Matlab Signal Processing Toolbox for AOs.

Syntaxis

```
b = interpolate(a, pl)
```

With the following parameters:

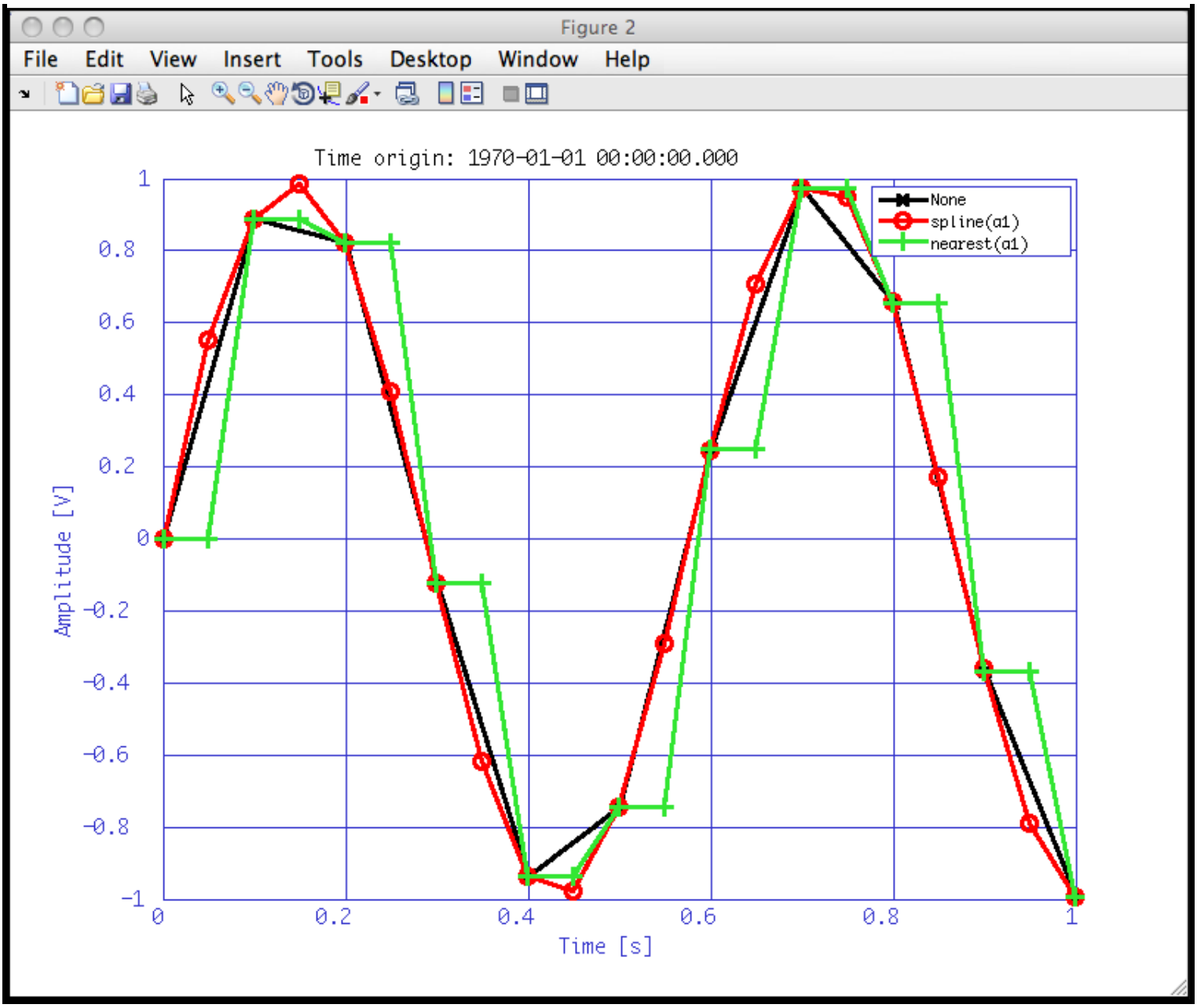
- `'vertices'` – specify the new vertices to interpolate on
- `'method'` – four methods are available for interpolating data
  - 'nearest'– nearest neighbor interpolation
  - 'linear' – linear interpolation
  - 'spline' – spline interpolation (default option)
  - 'cubic' – shape-preserving piecewise cubic interpolation

For details see `interp1` help of Matlab.

Examples

1. Interpolation of a sequence of random data at original sampling rate of 1 Hz by a factor of 10 with no initial phase.

```
% Signal generation
nsecs = 100;
fs    = 10;

pl = plist();
pl = append(pl, param('nsecs', nsecs));
pl = append(pl, param('fs', fs));
pl = append(pl, param('tsfcn', 'sin(2*pi*1.733*t)'));
a1 = ao(pl);

% Interpolate on a new time vector
t = linspace(0, a1.data.nsecs - 1/a1.data.fs, 2*len(a1));

plspline = plist();
plspline = append(plspline, param('vertices', t));

plnearest = plist();
plnearest = append(plnearest, param('vertices', t));
plnearest = append(plnearest, param('method', 'nearest'));

bspline = interp(a1, plspline);
bnearest = interp(a1, plnearest);

iplot([a1 bspline bnearest], plist('Markers', {'x', 'o', '+'}, 'LineColors', {'k', 'r'}));
ltpda_xaxis(0, 1)
```



Resampling data

Spikes reduction in data

Spikes reduction in data

Spikes in data due to different nature can be removed, if desired, from the original data. LTPDA `ltpda_spikecleaning` detects and replaces spikes of the input AOs. A spike in data is defined as a single sample exceeding a certain value (usually, the floor noise of the data) defined by the user:

$$|x_{HPF}[n]| \leq k_{spike}\sigma_{HPF}$$

where $x_{HPF}[n]$ is the input data high-pass filtered, k_{spike} is a value defined by the user (by default is 3.3) and σ_{HPF} is the standard deviation of $x_{HPF}[n]$. In consequence, a spike is defined as the value that exceeds the floor noise of the data by a factor k_{spike}, the higher of this parameter the more difficult to "detect" a spike.

Syntaxis

```
b = ltpda_spikecleaning(a, pl)
```

With the following parameters:

- `'kspike'` - set the k_{spike} value (default is 3.3)
- `'method'` - method used to replace the "spiky" sample. Three methods are available ---see below for details---:
  - `'random'`
  - `'mean'`
  - `'previous'`
- `'fc'` - frequency cut-off of the high-pass IIR filter (default is 0.025)
- `'order'` - specifies the order of the IIR filter (default is 2)
- `'ripple'` - specifies pass/stop-band ripple for bandpass and bandreject filters (default is 0.5)

Methods explained

1. `Random` : this method substitutes the spiky sample by:

$$x[n] = x[n-1] + N(0,1) \cdot \sigma_{HPF}$$

where $N(0,1)$ is a random number of mean zero and standard deviation 1.

2. `Mean` : this method uses the following equation to replace the spike detected in data.

$$x[n] = \frac{x[n-1] + x[n-2]}{2}$$

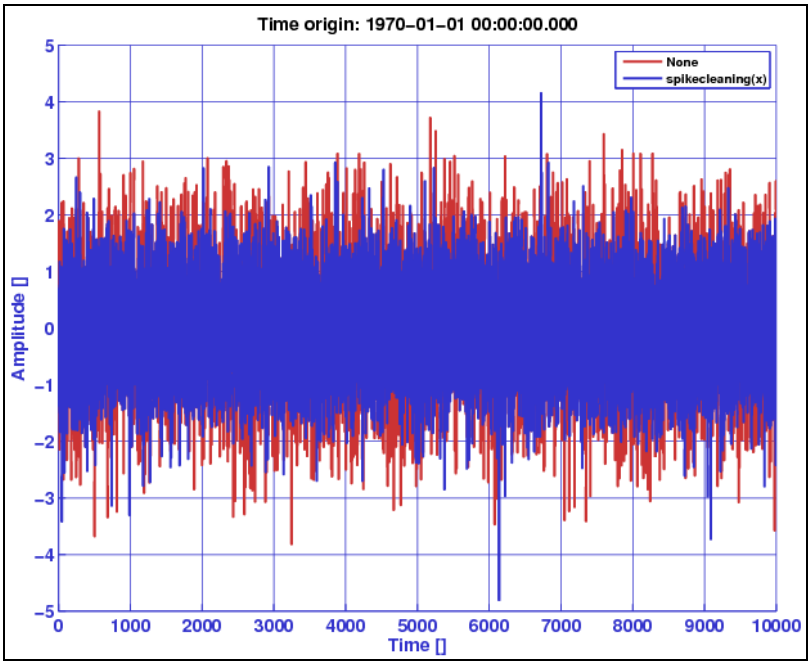
3. `Previous` : the spike is substitued by the previous sample, i.e.:

$$x[n] = x[n-1]$$

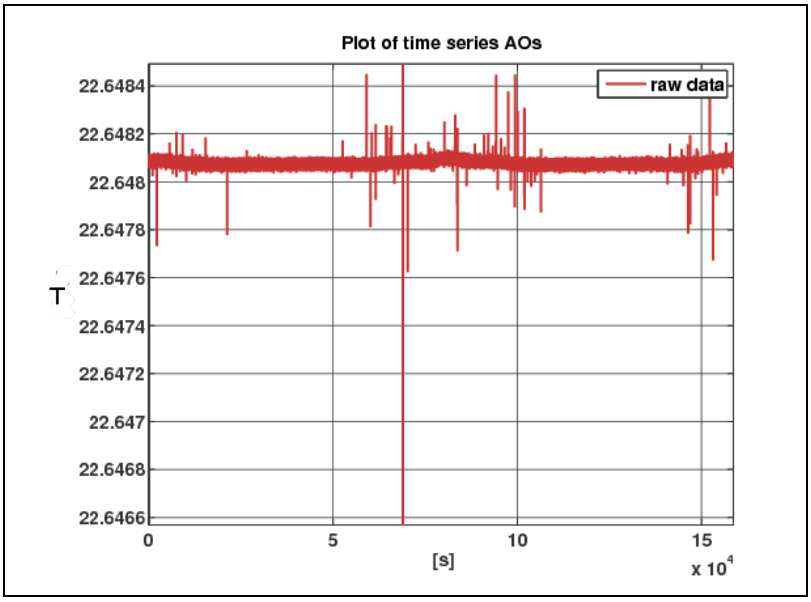
Examples

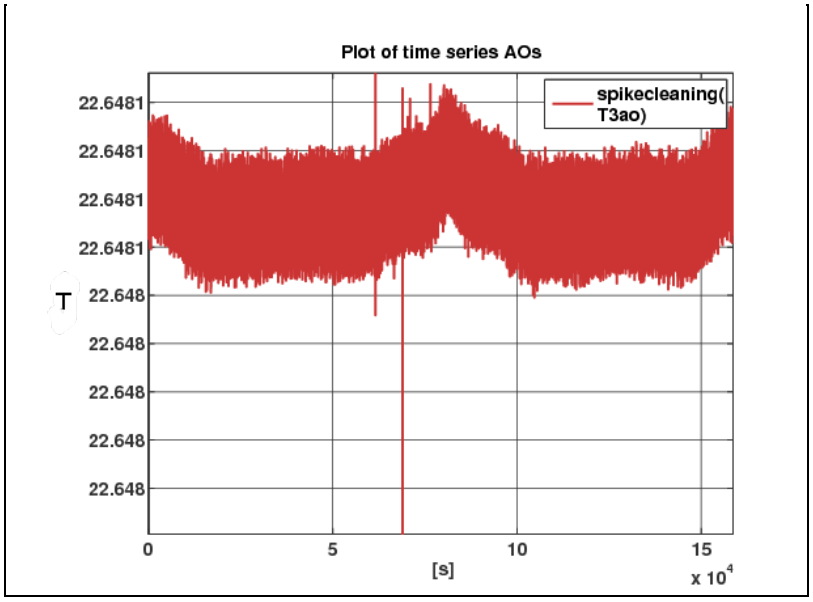
1. Spike cleaning of a sequence of random data with `kspike=2`.

```
                x = ao(tsdata(randn(10000,1),1)); % create an AO of random data sampled at 1
Hz.
                pl = plist();  % create an empty parameters list
                pl = append(pl, param('kspike', 2)); % kspike=2
                y = ltpda_spikecleaning(x, pl); % spike cleaning function applied to the
input AO, x
                iplot(x, y) % plot original and "cleaned" data
```



2. Example of real data: the first image shows data from the real world prior to the application of the spike cleaning tool. It is clear that some spikes are present in data and might be convenient to remove them. The second image shows the same data after the spike samples supression.





Interpolating data

Data gap filling

Data gap filling

Gaps in data can be filled with *interpolated* data if desired. LTPDA `ltpda_gapfilling` joints two AOs by means of a segment of *synthetic* data. This segment is calculated from the two input AOs. Two different methods are possible to fill the gap in the data: `linear` and `spline`. The former fills the data by means of a linear interpolation whereas the latter uses a smoother curve ---see examples below.

Syntaxis

```
b = ltpda_gapfilling(a1, a2, pl)
```

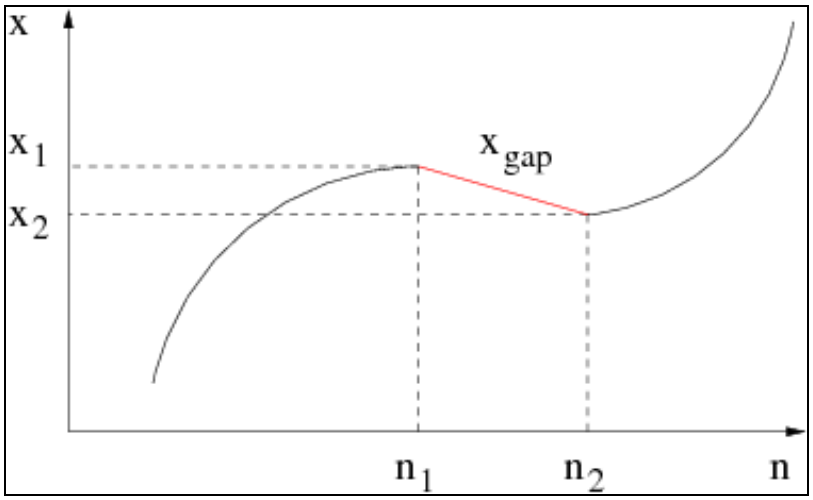
where `a1` and `a2` are the two segments to joint. The parameters are:

- `'method'` - method used to interpolate missing data (see below for details)
  - `'linear'`
    (default option)
  - `'spline'`
- `'addnoise'` - with this option *noise* can be added to the interpolated data. This noise is defined as a
  random variable with zero mean and variance equal to the high-frequency noise of the input AO.

Interpolation methods

1. `Linear` :

$$x_{GAP}[n] = \frac{x_2 - x_1}{n_2 - n_1} n + x + \sigma_{HPF}$$

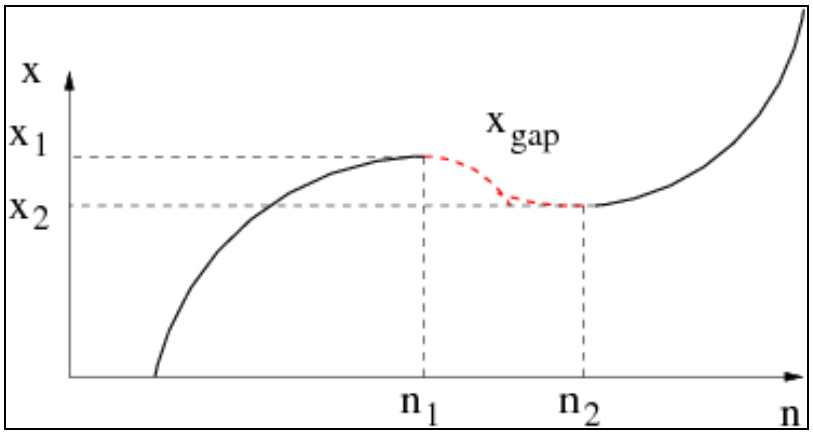


2. `Spline (or third order interpolation)` :

$$x_{GAP}[n] = an^3 + bn^2 + cn + d + \sigma_{HPF}$$

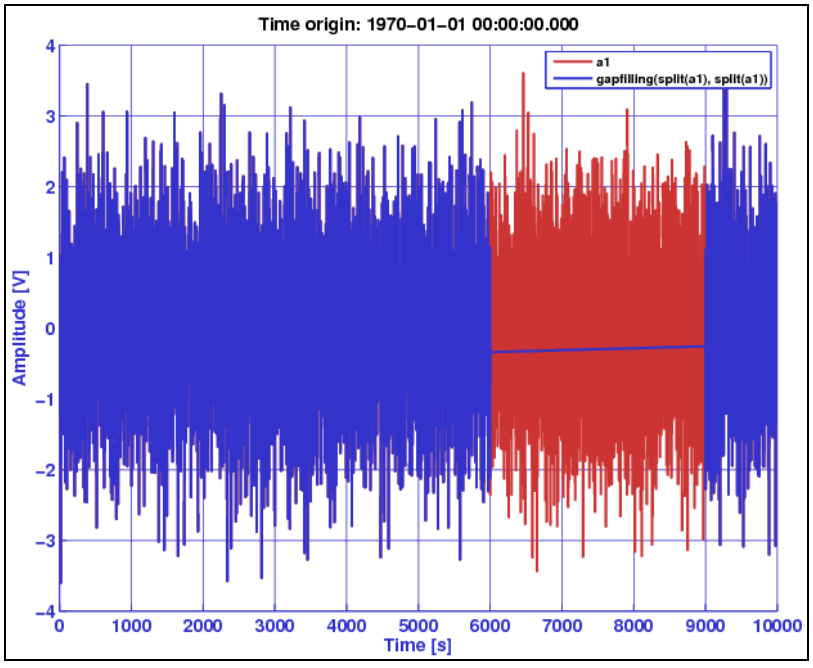
The parameters `a`, `b`, `c` and `d` are calculated by solving next system of equations:

$$
\begin{aligned}
x_{GAP}[n_1] &= x_1 \\
x_{GAP}[n_2] &= x_2 \\
\left.\frac{dx_{GAP}[n]}{dn}\right|_{n=n_1} &= \left.\frac{dx_1[n]}{dn}\right|_{n=n_1} \\
\left.\frac{dx_{GAP}[n]}{dn}\right|_{n=n_2} &= \left.\frac{dx_2[n]}{dn}\right|_{n=n_2}
\end{aligned}
$$

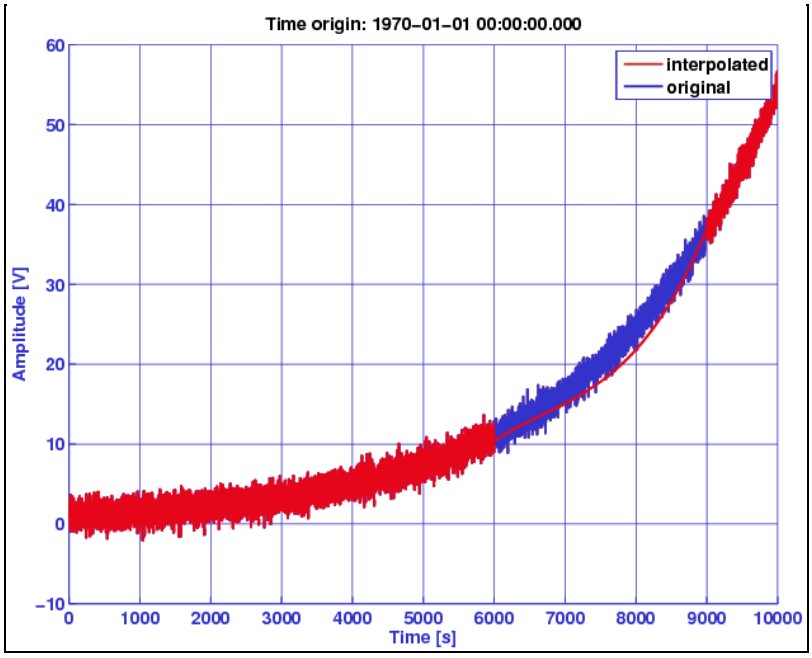


Examples

1. Missing data between two vectors of random data interpolated with the `linear` method.



2. Missing data between two data vectors interpolated with the `spline` method.



Spikes reduction in data

Signal Processing in LTPDA

Signal Processing in LTPDA

The LTPDA Toolbox contains a set of tools to characterise digital data streams within the framework of LTPDA Objects. The current available methods can be grouped in the following categories:

- Digital filtering
- Spectral estimation
- Fitting algorithms

Data gap filling Digital Filtering

Digital Filtering

A digital filter is an operation that associates an input time series x[n] into an output one, y[n]. Methods developed in the LTPDA Toolbox deal with linear digital filters, i.e. those which fulfill that a linear combination of inputs results in a linear combination of outputs with the same coefficients (provided that these are not time dependent). In these conditions, the filter can be expressed as

$$y[n] = \sum_{k=-\infty}^{\infty} h[k]\, x[n-k]$$

described in these terms, the filter is completely described by the impulse response h[k], and can then be subdivided into the following classes:

- Causal: if there is no output before input is fed in.

$$h[k] = 0,\ k < 0$$

- Stable: if finite input results in finite output.

$$\sum_{k=-\infty}^{\infty} h[k] < \infty$$

- Shift invariant: if time shift in the input results in a time shift in the output by the same amount.

$$h[k] = h[-k]$$

Digital filters classification

Digital filters can be described as difference equations. If we consider an input time series x and an output y, three specific cases can then be distinguished:

- Autoregressive (AR) process: the difference equation in this case is given by:

$$y[n] = \sum_{k=1}^{\mathrm{M}} b[k]\, y[n-k]$$

  AR processes can be also classified as IIR Filters.

- Moving Averrage (MA) process:the difference equation in this case is given by:

$$y[n] = \sum_{k=0}^{\mathrm{N}} a[k]\, x[n-k]$$

  MA processes can be also classified as FIR Filters.

- Autoregressive Moving Average (ARMA) process: the difference equation in this case contains both an AR and a MA process:

$$y[n] = \sum_{k=0}^{\mathrm{N}} b[k]\, x[n-k] - \sum_{k=1}^{\mathrm{M}} a[k]\, y[n-k]$$

Signal Processing in LTPDA IIR Filters

IIR Filters

Infinite Impulse Response filters are those filters present a non-zero infinite length response when excited with a very brief (ideally an infinite peak) input signal. A linear causal IIR filter can be described by the following difference equation

$$y[n] = \sum_{k=0}^{N} b[k]\, x[n-k] - \sum_{k=1}^{M} a[k]\, y[n-k]$$

This operation describe a recursive system, i.e. a system that depends on current and past samples of the input x[n], but also on the output data stream y[n].

Creating a IIR filter in the LTPDA

The LTPDA Toolbox allows the implementation of IIR filters by means of the miir class.

Creating from a plist

The following example creates an order 1 highpass filter with high frequency gain 2. Filter is designed for 10 Hz sampled data and has a cut-off frequency of 0.2 Hz.

```
>> pl = plist('type', 'highpass', ...
              'order', 1,         ...
              'gain',  2.0,       ...
              'fs',    10,        ...
              'fc',    0.2);
>> f = miir(pl)
```

Creating from a pzmodel

IIR filters can also be created from a pzmodel .

Creating from a difference equation

Alternatively, the filter can be defined in terms of two vectors specifying the coefficients of the filter and the sampling frequency. The following example creates a IIR filter with sampling frequency 1 Hz and the following recursive equation:

$$y[n] = 0.5\, x[n] - 0.01\, x[n-1] - 0.1\, y[n-1]$$

```
>> a = [0.5 -0.01];
>> b = [1 0.1];
>> fs = 1;
>> f = miir(a,b,fs)
```

Notice that the convetion used in this function is the one described in the Digital filters classification section

Importing an existing model

The miir constructor also accepts as an input existing models in different formats:

-
- LISO files:

```
>> f = miir('foo_iir.fil')
```

- XML files:

```
>> f = miir('foo_iir.xml')
```

- MAT files:

```
>> f = miir('foo_iir.mat')
```

- From repository:

```
>> f = miir(plist('hostname', 'localhost', 'database', 'ltpda', 'ID', []))
```

Digital Filtering FIR Filters

FIR Filters

Finite Impulse Response filters are those filters present a non-zero finite length response when excited with a very brief (ideally an infinite peak) input signal. A linear causal FIR filter can be described by the following difference equation

$$y[n] = \sum_{k=0}^{M} b[k]\, x[n-k]$$

This operation describe a nonrecursive system, i.e. a system that only depends on current and past samples of the input data stream x[n]

Creating a FIR filter in the LTPDA

The LTPDA Toolbox allows the implementation of FIR filters by means of the mfir class.

Creating from a plist

The following example creates an order 64 highpass filter with high frequency gain 2. Filter is designed for 1 Hz sampled data and has a cut-off frequency of 0.2 Hz.

```
>> pl = plist('type', 'highpass', ...
              'order', 64,          ...
              'gain',  2.0,         ...
              'fs',    1,           ...
              'fc',    0.2);
>> f = mfir(pl)
```

Creating from a difference equation

The filter can be defined in terms of two vectors specifying the coefficients of the filter and the sampling frequency. The following example creates a FIR filter with sampling frequency 1 Hz and the following recursive equation:

$$y[n] = -0.8\, x[n] + 10\, x[n-1]$$

```
>> b = [-0.8 10];
>> fs = 1;
>> f = mfir(b,fs)
```

Creating from an Analysis Object

A FIR filter can be generated based on the magnitude of the input Analysis Object or fsdata object. In the following example a fsdata object is first generated and then passed to the mfir constructor to obtain the equivalent FIR filter.

```
>> fs  = 10;                        $ sampling frequency
>> f = linspace(0, fs/2, 1000);
>> y = 1./(1+(0.1*2*pi*f).^2);     $ an arbitrary function
```

```
>> fsd = fsdata(f,y,fs);          $ build the fsdata object
>> f = mfir(ao(fsd));
```

Available methods for this option are: 'frequency-sampling' (uses fir2), 'least-squares' (uses firls) and 'Parks-McClellan' (uses firpm)

Importing an existing model

The mfir constructor also accepts as an input existing models in different formats:

-
- LISO files:

```
>> f = mfir('foo_fir.fil')
```

- XML files:

```
>> f = mfir('foo_fir.xml')
```

- MAT files:

```
>> f = mfir('foo_fir.mat')
```

- From repository:

```
>> f = mfir(plist('hostname', 'localhost', 'database', 'ltpda', 'ID', []))
```

IIR Filters Spectral Estimation

Spectral Estimation

Spectral estimation is a branch of the signal processing, performed on data and based on frequency-domain techniques. Within the LTPDA toolbox many functions of the Matlab Signal Processing Toolbox (which is required) were rewritten to operate on LTPDA Analysis Objects. Univariate and multivariate technique are available, so to estimate for example the linear power spectral density or the cross spectral density of one or more signals. The focus of the tools is on time-series objects, whose spectral content needs to be estimated.

The power spectrum density estimators are based on pwelch from MATLAB, which is an implementation of Welch's averaged, modified periodogram method.

The following pages describe the different Welch-based spectral estimation tools available in the LTPDA toolbox:

- LTPDA_PWELCH power spectral density estimates
- LTPDA_CPSD cross-spectral density estimates
- LTPDA_COHERENCE magnitude squared coherence estimates
- LTPDA_TFE transfer function estimates

As an alternative, the LTPDA toolbox makes available the same set of estimators, based on an implementation of the LPSD algorithm (See "Improved spectrum estimation from digitized time series on a logarithmic frequency axis", M Troebs, G Heinzel, Measurement 39 (2006) 120-129).

The following pages describe the different LPSD-based spectral estimation tools available in the LTPDA toolbox:

- LTPDA_LPSDLog-scale power spectral density estimates
- LTPDA_LCPSDLog-scale cross-spectral density estimates
- LTPDA_LCOHERENCELog-scale magnitude squared coherence estimates
- LTPDA_LTFELog-scale transfer function estimates

More detailed help on spectral estimation can be found in the help associated with the Signal Processing Toolbox (>> doc signal)

Power spectral density estimates

Univariate power spectral density is performed by the Welch's averaged, modified periodogram method. `ltpda_pwelch` estimates the power spectral density of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectrum, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lenghts to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well. The detrending is performed on the individual windows. The user can choose the quantity being given in output among ASD (amplitude spectral density), PSD (power spectral density), AS (amplitude spectrum), and PS (power spectrum).

Syntaxis

```
b = ltpda_pwelch(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AO(s) containing the input time series to be evaluated. `b` includes the output object(s). The parameter list `pl` includes the following parameters:

- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Nfft'` - number of samples in each fft [default: length of input data]
- `'Scale'` - one of
  - 'ASD' - amplitude spectral density
  - 'PSD' - power spectral density [default]
  - 'AS' - amplitude spectrum
  - 'PS' - power spectrum
- `'Order'` - order of segment detrending
  - -1 - no detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

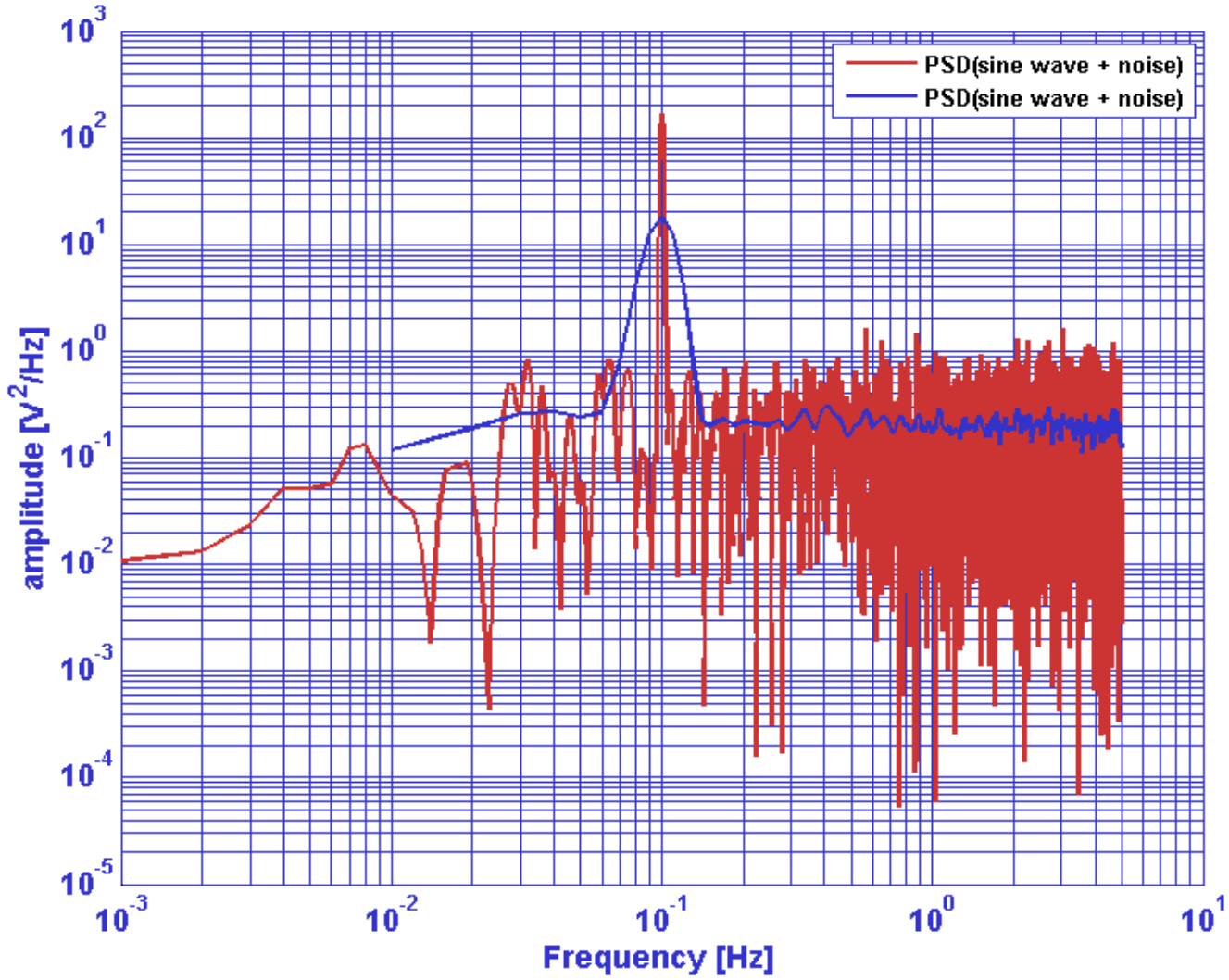
The length of the window is set by the value of the parameter `'Nfft'`, so that the window is actually rebuilt using only the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

```
If the user doesn't specify the value of a given parameter, the default value is used.
```

Examples

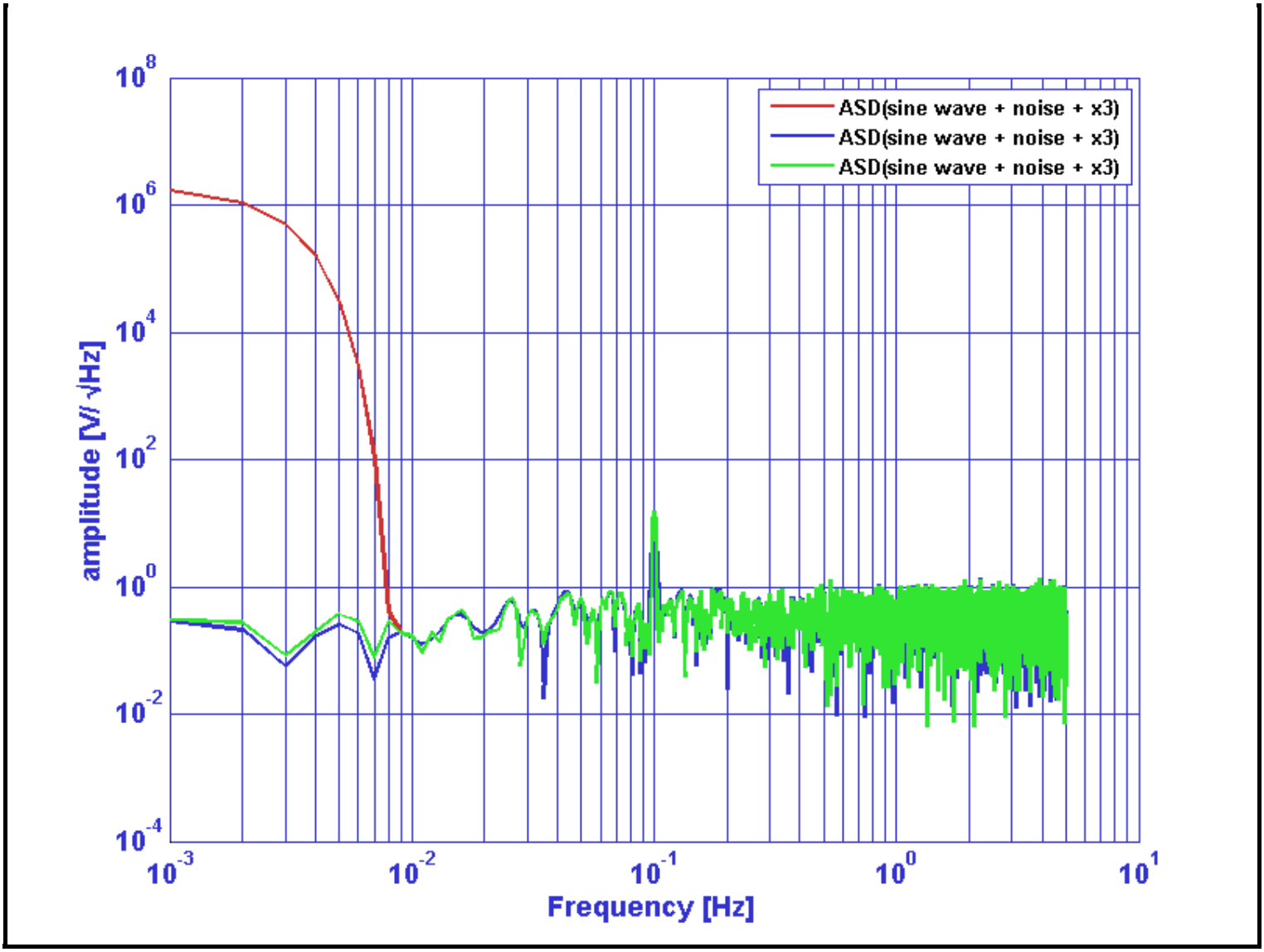
1. Evaluation of the PSD of a time-series represented by a low frequency sinewave signal, superimposed to white noise. Comparison of the effect of windowing on the estimate of the white noise level and on resolving the signal.

```
    x1 = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10));
    x2 = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
x = x1 + x2;
    y_lf = ltpda_pwelch(x);
y_hf = ltpda_pwelch(x,plist('nfft',1000));
    iplot(y_lf, y_hf)
```



2. Evaluation of the PSD of a time-series represented by a low frequency sinewave signal, superimposed to white noise and to a low frequency linear drift. Comparison of the effect of different windows and of detrending.

```
x1 = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10));
x2 = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
x3 = ao(plist('tsfcn', 't.^2 + t','nsecs',1000,'fs',10));
x = x1 + x2 + x3;
y_1 = ltpda_pwelch(x,plist('scale','ASD','order',1));
y_2 = ltpda_pwelch(x,plist('scale','ASD','order',2));
y_3 = ltpda_pwelch(x,plist('scale','ASD','order',2, 'win', specwin(plist('name', 'BH92'))));
    iplot(y_1, y_2, y_3);
```



Spectral Estimation

Cross-spectral density estimates

Cross-spectral density estimates

Multivariate power spectral density is performed by the Welch's averaged, modified periodogram method. `ltpda_cpsd` estimates the cross-spectral density of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lenghts to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well.

Syntaxis

```
b = ltpda_cpsd(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AOs containing the input time series to be evaluated. They need to be in a number N >= 2. `b` includes the NXN output objects. The parameter list `pl` includes the following parameters:

- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Nfft'` - number of samples in each fft [default: length of input data]
- `'Order'` - order of segment detrending
  - -1 - no detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

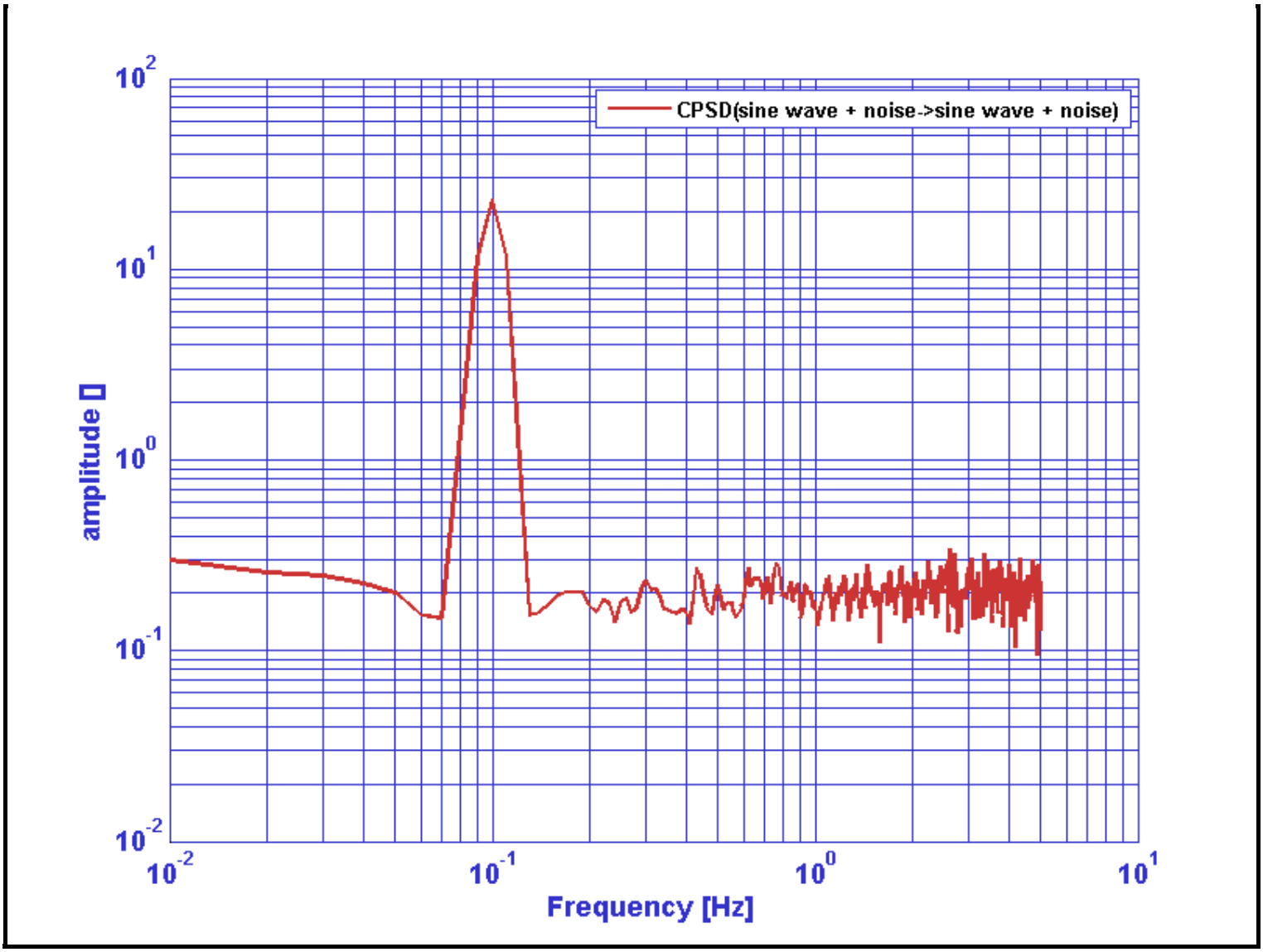
The length of the window is set by the value of the parameter `'Nfft'`, so that the window is actually rebuilt using only the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

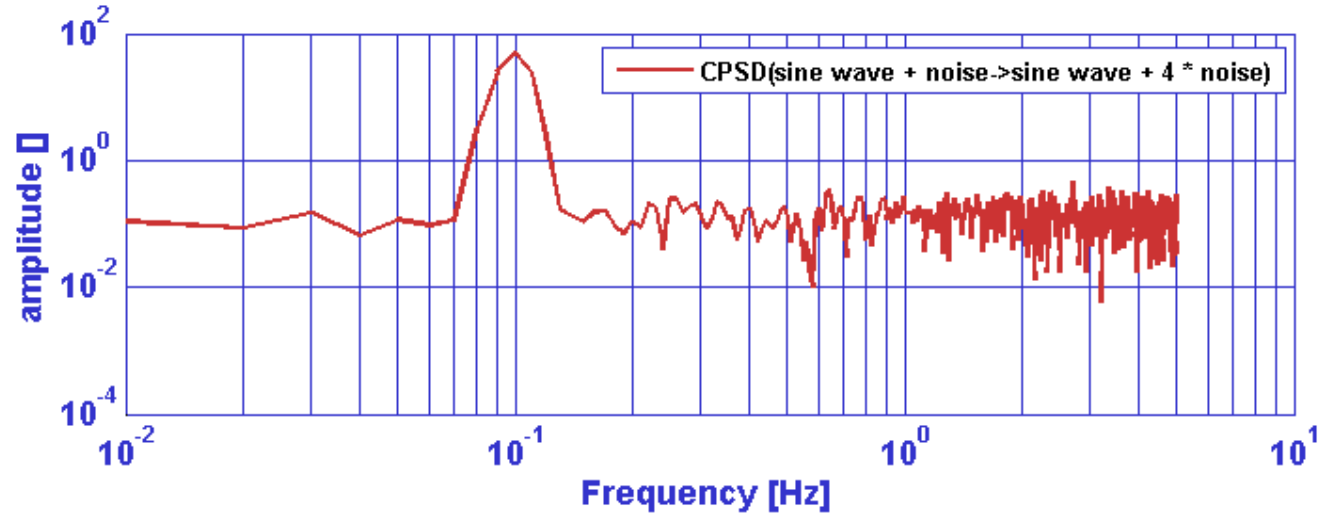
```
  If the user doesn't specify the value of a given parameter, the default value is used.
  The function makes CPSD estimates between all input AOs. Therefore, if the input argument
list contains N analysis objects, the output a will contain NxN CPSD estimates.
The diagonal elements will be S_ai_ai and will be equivalent to the output of ltpda_pwelch(ai).
```

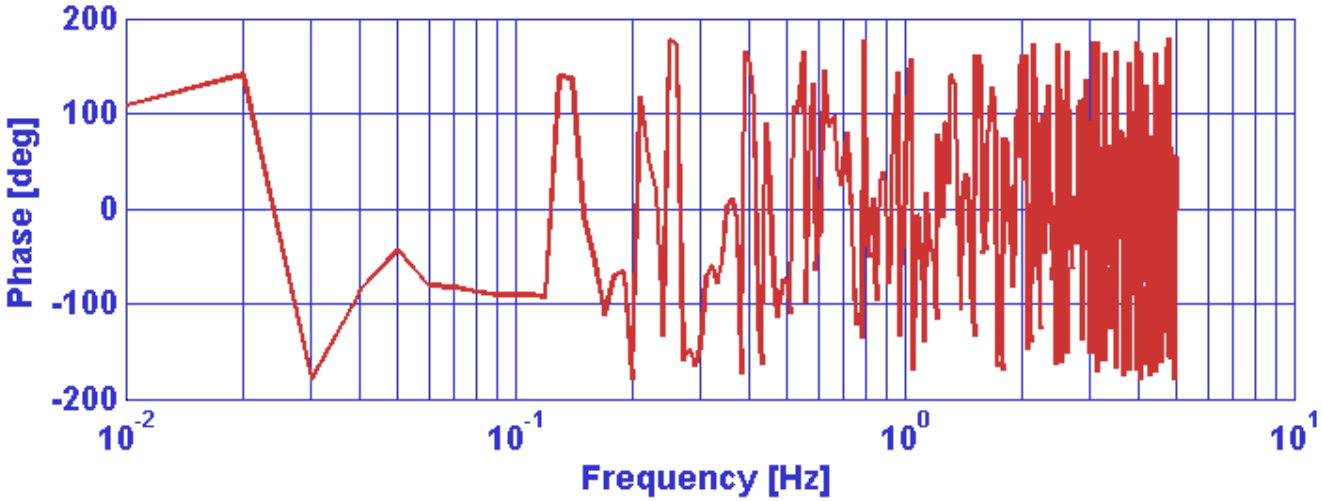
Example

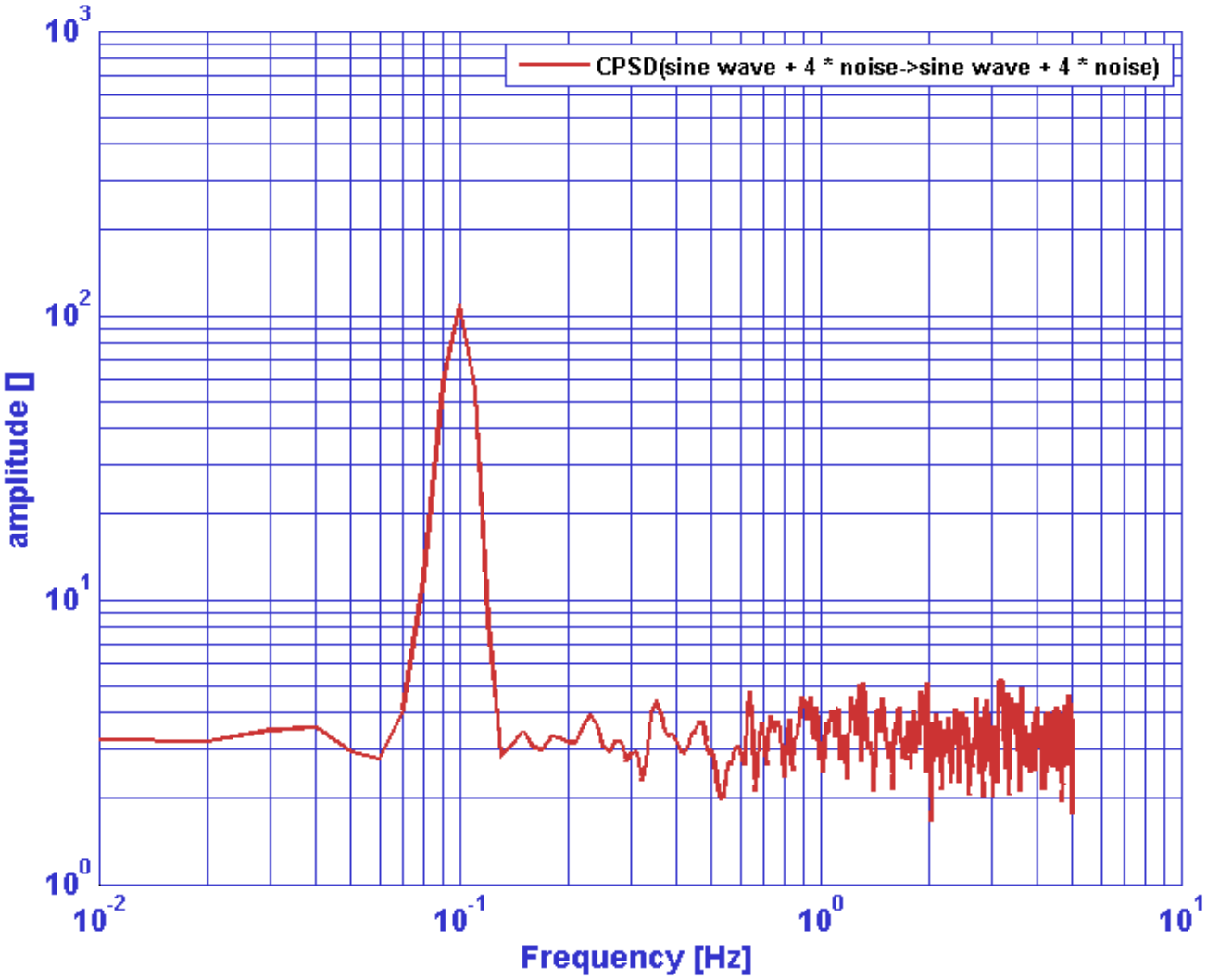
Evaluation of the CPSD of two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
    x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
    y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',1000,'fs',10,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
z = ltpda_cpsd(x,y,plist('nfft',1000));
    iplot(z(1,1));
iplot(z(1,2));
iplot(z(2,2));
```


CPSD(sine wave + noise->sine wave + noise)
amplitude []
Frequency [Hz]
10^2
10^1
10^0
10^-1
10^-2
10^-2
10^-1
10^0
10^1


CPSD(sine wave + noise->sine wave + 4 * noise)
amplitude []
Frequency [Hz]


Phase [deg]
Frequency [Hz]



Power spectral density estimates
Cross coherence estimates

Cross coherence estimates

Multivariate power spectral density is performed by the Welch's averaged, modified periodogram method. `ltpda_cohere` estimates the coherence of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lenghts to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well.

Syntaxis

```
b = ltpda_cohere(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AOs containing the input time series to be evaluated. They need to be in a number N >= 2. `b` includes the NXN output objects. The parameter list `pl` includes the following parameters:

- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Nfft'` - number of samples in each fft [default: length of input data]
- `'Order'` - order of segment detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

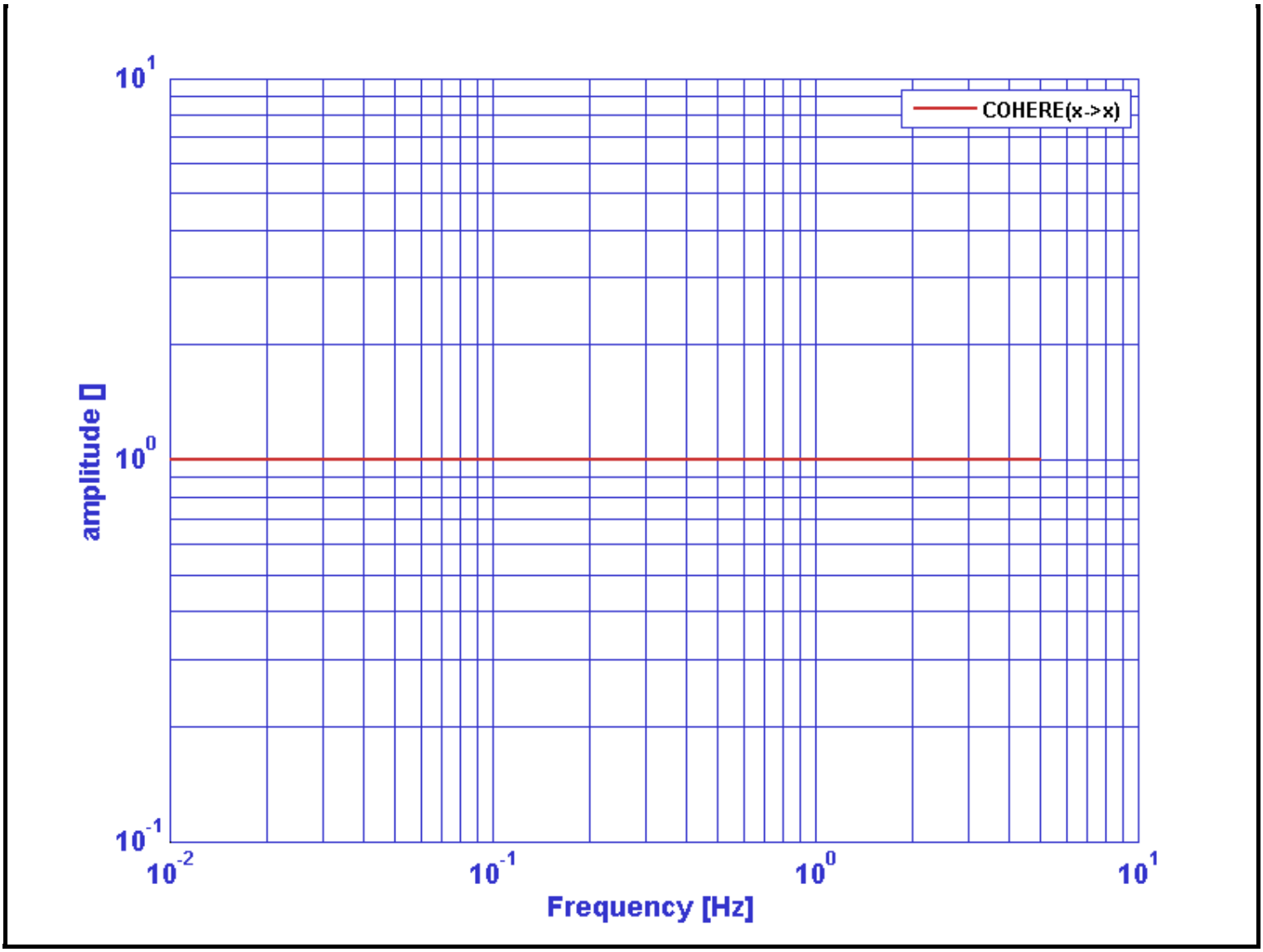
The length of the window is set by the value of the parameter `'Nfft'`, so that the window is actually rebuilt using only the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

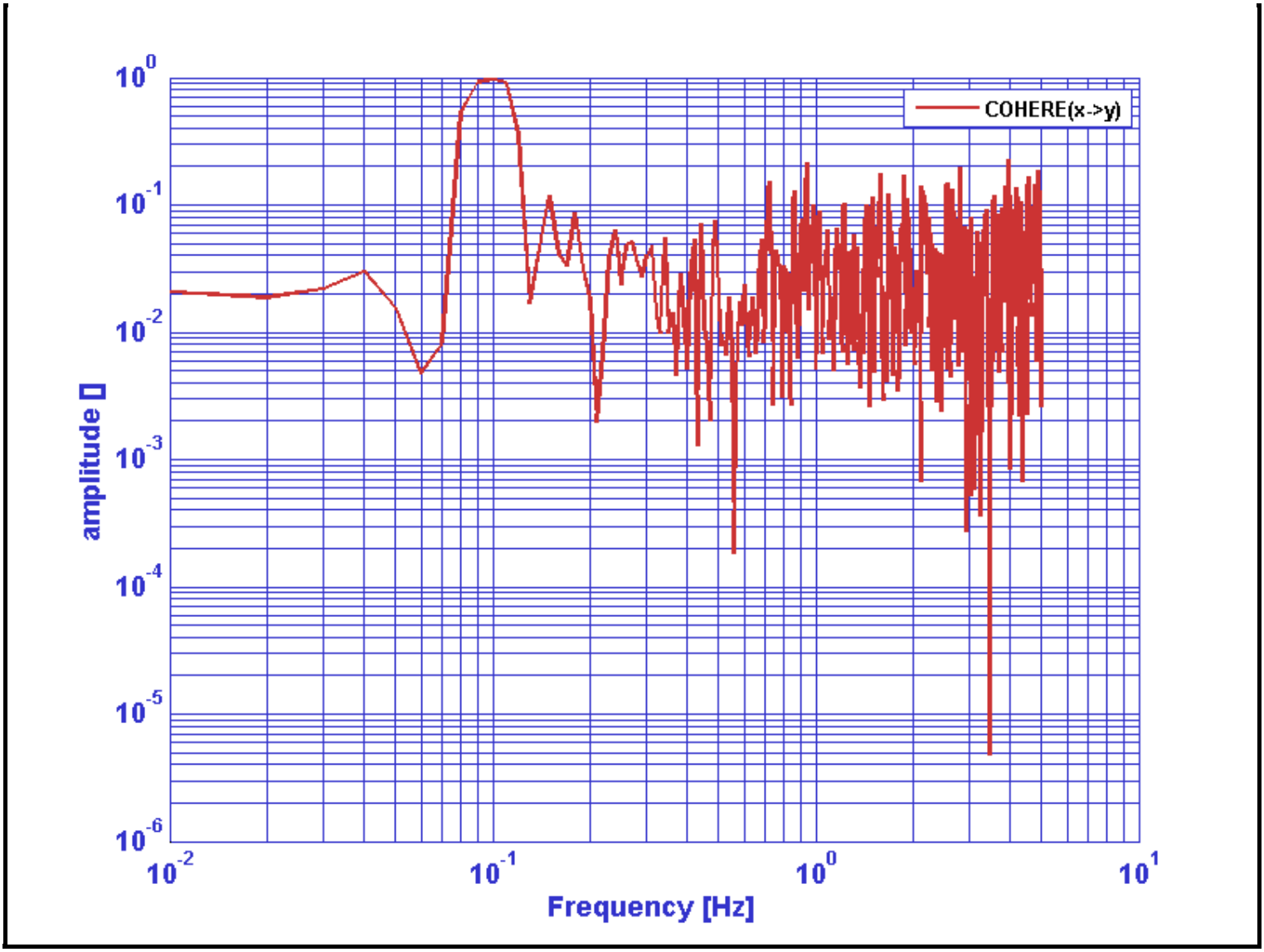
```
  If the user doesn't specify the value of a given parameter, the default value is used.
  The function makes coherence estimates between all input AOs. Therefore, if the input argument
list contains N analysis objects, the output a will contain NxN coherence estimates.
The diagonal elements will be 1.
```

Example

Evaluation of the coherence of two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
    x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10)) + ...
    ao(plist('tsfcn', 't','nsecs',1000,'fs',10));
    y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',1000,'fs',10,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
pl = plist('win',specwin(plist('name','BH92')),'nfft',1000);
z = ltpda_cohere(x,y,pl);
iplot(z(1,1));
iplot(z(1,2));
```


COHERE(x->x)
amplitude []
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-1}
10^{0}
10^{1}
Frequency [Hz]



Cross-spectral density estimates
Transfer function estimates

Transfer function estimates

Multivariate power spectral density is performed by the Welch's averaged, modified periodogram method. `ltpda_tfe` estimates the transfer function of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lenghts to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well.

Syntaxis

```
b = ltpda_tfe(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AOs containing the input time series to be evaluated. They need to be in a number N >= 2. `b` includes the NXN output objects. The parameter list `pl` includes the following parameters:

- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Nfft'` - number of samples in each fft [default: length of input data / 4]
- `'Order'` - order of segment detrending
  - -1 - no detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

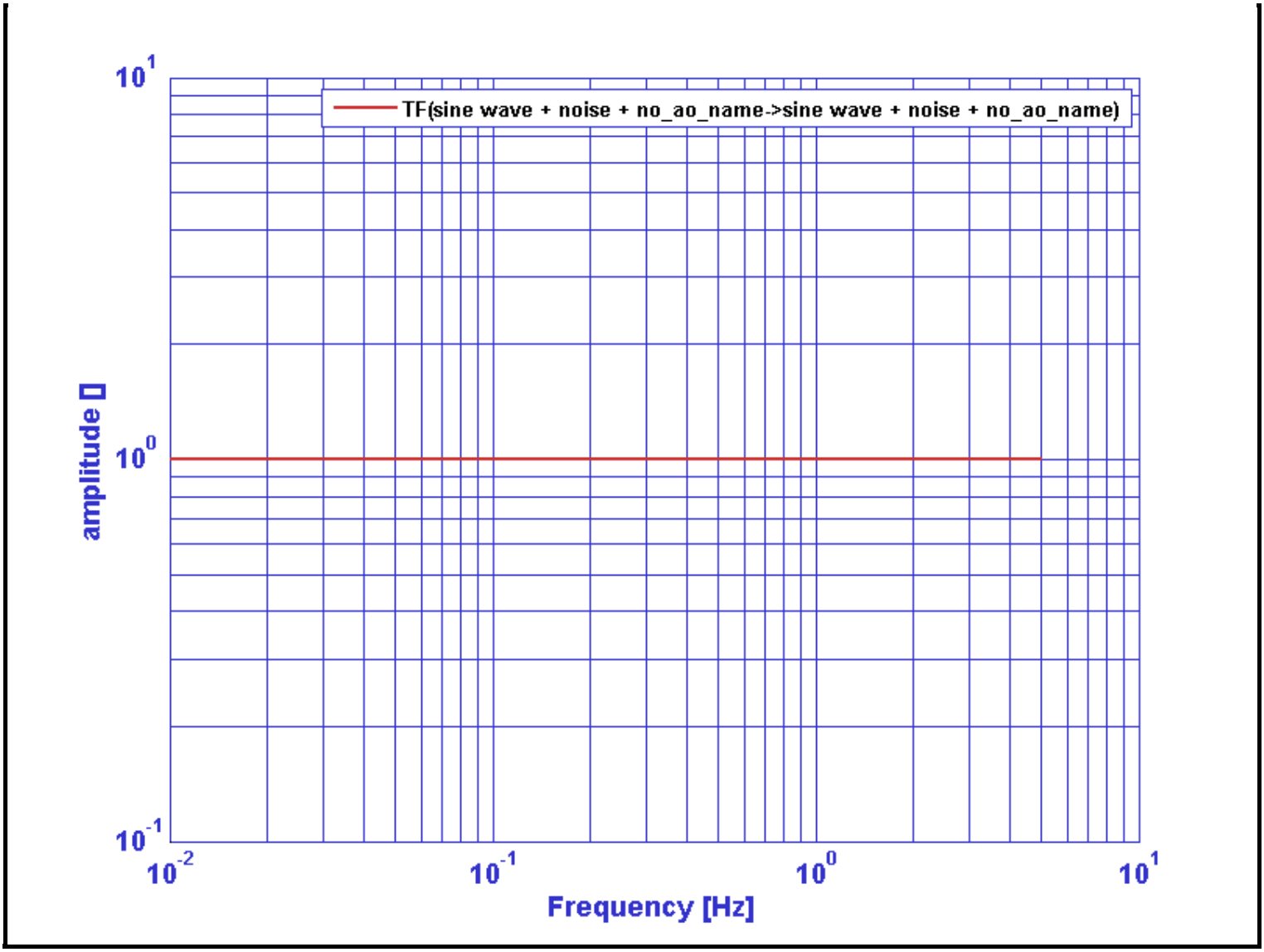
The length of the window is set by the value of the parameter `'Nfft'`, so that the window is actually rebuilt using only the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

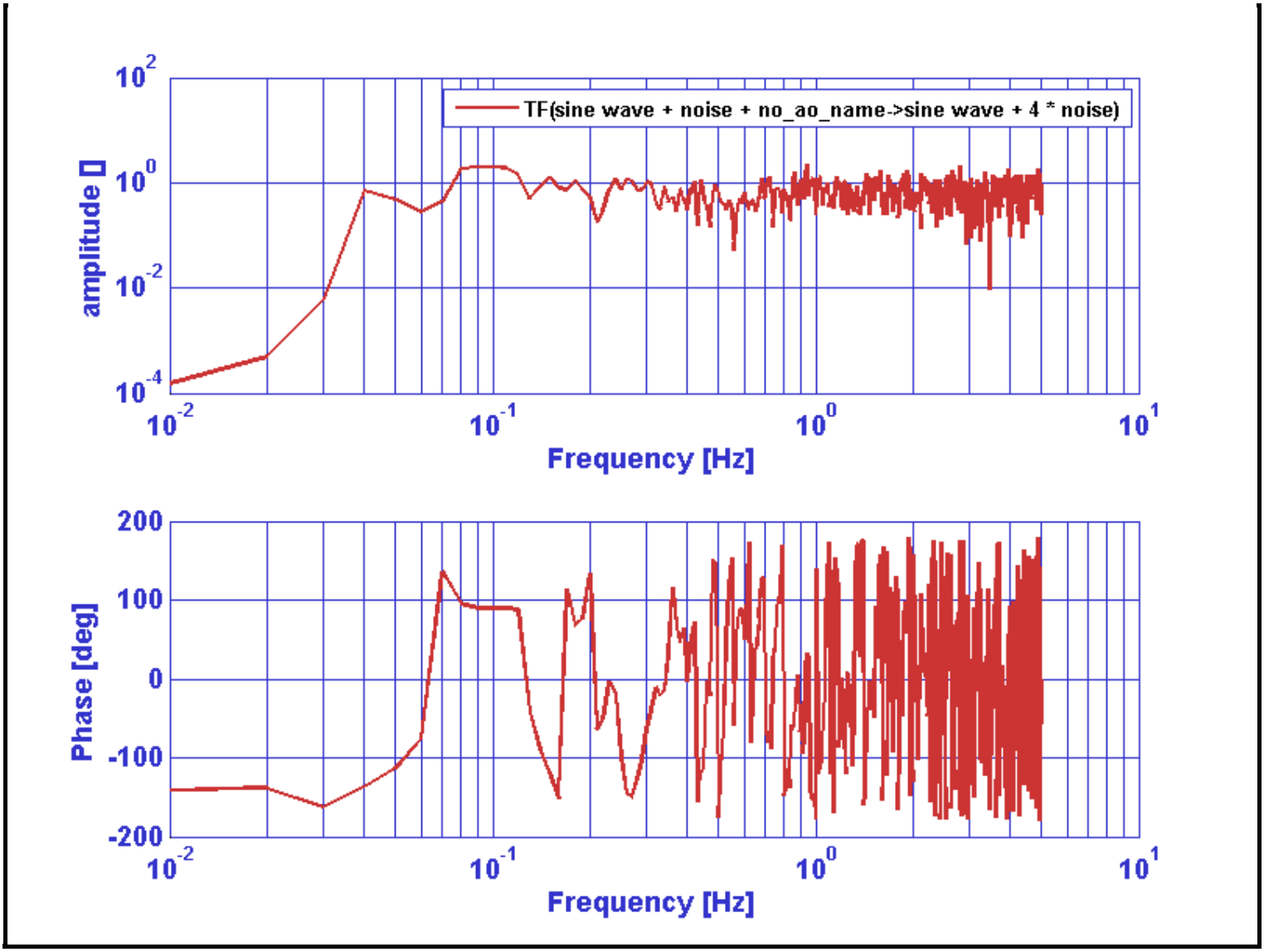
```
  If the user doesn't specify the value of a given parameter, the default value is used.
  The function makes transfer functions estimatesbetween all input AOs. Therefore, if the input
argument
list contains N analysis objects, the output a will contain NxN TFE estimates.
The diagonal elements will be 1.
```

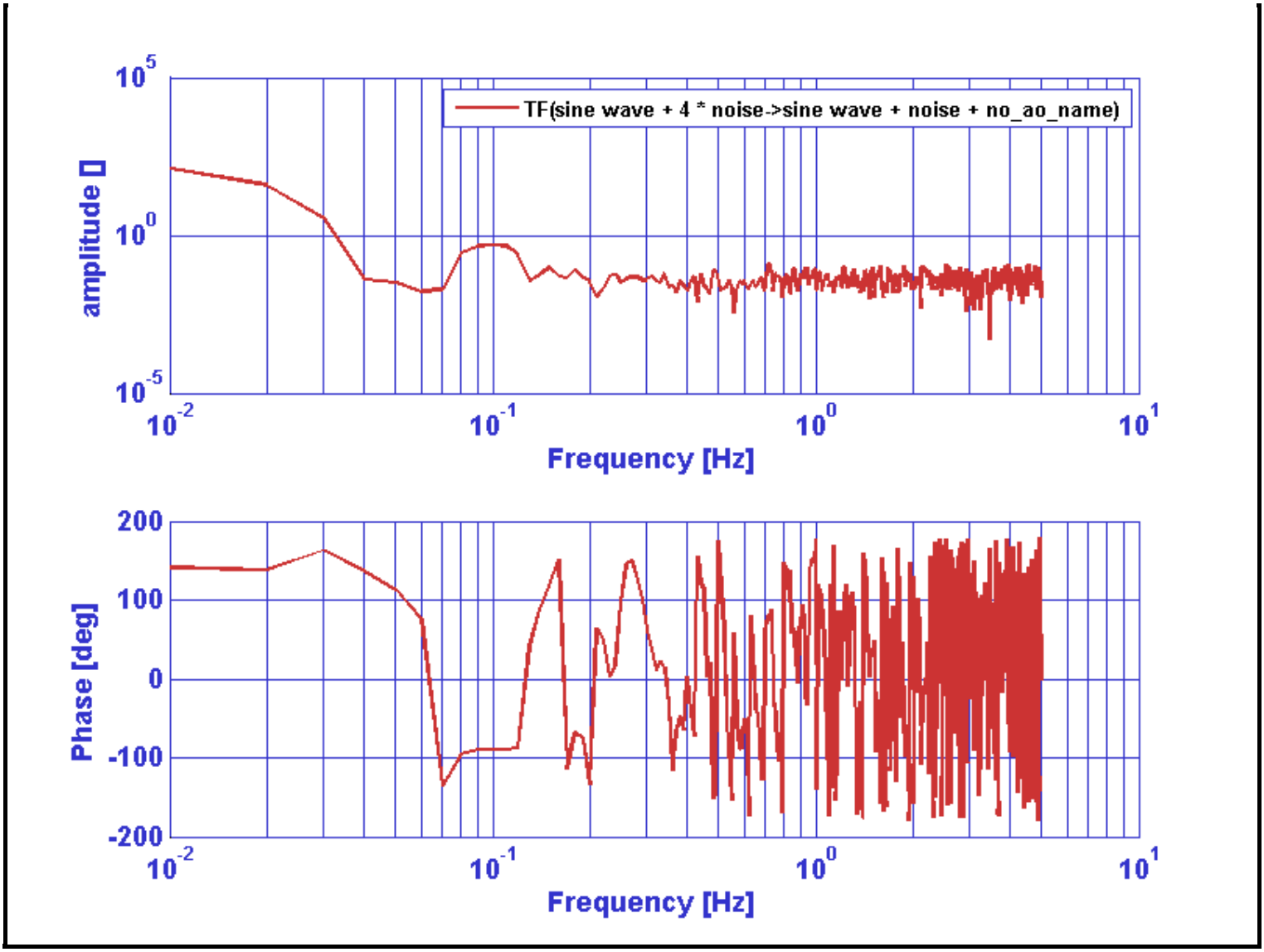
Example

Evaluation of the transfer function between two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
    x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
    y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',1000,'fs',10,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
z = ltpda_tfe(x,y,plist('win',specwin(plist('name','BH92')),'nfft',1000));
    iplot(z(1,1));
iplot(z(1,2));
iplot(z(2,1));
```


TF(sine wave + noise + no_ao_name->sine wave + noise + no_ao_name)
amplitude []
Frequency [Hz]
10^1
10^0
10^-1
10^-2
10^-1
10^0
10^1


TF(sine wave + noise + no_ao_name->sine wave + 4 * noise)
amplitude []
Frequency [Hz]
Phase [deg]
Frequency [Hz]



Cross coherence estimates
Log-scale power spectral density estimates

Log-scale power spectral density estimates

Univariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm, which is an application of Welch's averaged, modified periodogram method, spectral density estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T` where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

`ltpda_lpsd` estimates the power spectral density of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectrum, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. Detrending is performed on each individual window. The user can choose the quantity being given in output among ASD (amplitude spectral density), PSD (power spectral density), AS (amplitude spectrum), and PS (power spectrum).

Syntaxis

```
b = ltpda_lpsd(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AO(s) containing the input time series to be evaluated. `b` includes the output object(s). The parameter list `pl` includes the following parameters:

- `'Kdes'` - the desired number of averages [default: 100]
- `'Jdes'` - the number of spectral frequencies to compute [default: fs/2]
- `'Lmin'` - the minimum segment length [default: 0]
- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Scale'` - one of
  - 'ASD' - amplitude spectral density
  - 'PSD' - power spectral density [default]
  - 'AS' - amplitude spectrum
  - 'PS' - power spectrum
- `'Order'` - order of segment detrending
  - -1 - no detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

The length of the window is recalculated for each freqency bin, so the 'Win' parameter is used only to provide the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

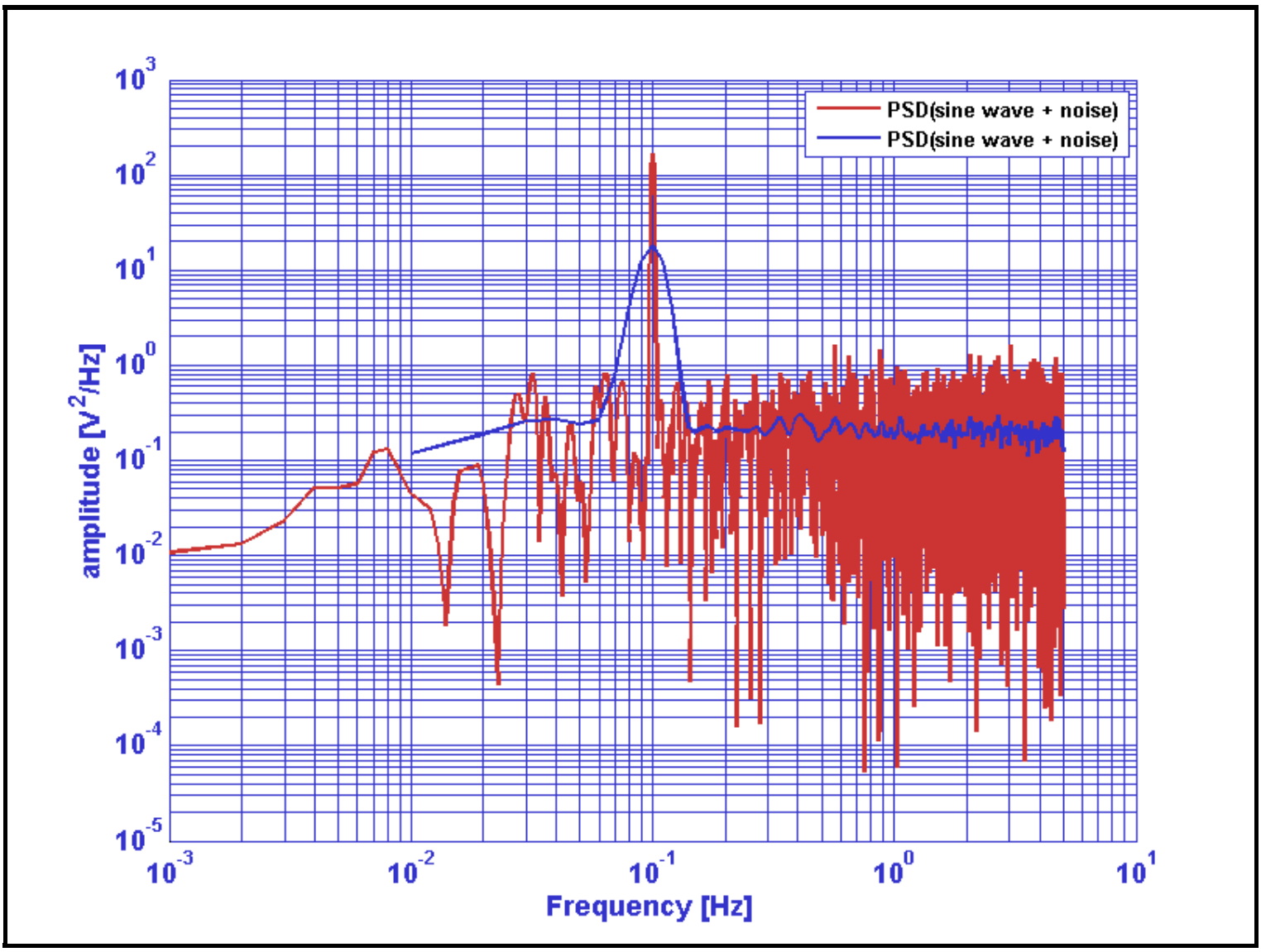
```
If the user doesn't specify the value of a given parameter, the default value is used.
```

Examples

1. Evaluation of the ASD of a time-series represented by a low frequency sinewave signal, superimposed to white noise. Comparison of the effect of using standard Pwelch and LPSD on the estimate of the white noise level and on resolving the signal.

```
x1 = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10));
x2 = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
```

```
x = x1 + x2;
pl = plist('scale','ASD','order',-1,'win',specwin(plist('name','BH92')));
    y1 = ltpda_pwelch(x, pl);
y2 = ltpda_lpsd(x,pl);
    iplot(y1, y2)
```



Transfer function estimates

Log-scale cross-spectral density estimates

Log-scale cross-spectral density estimates

Multivariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm, which is an application of Welch's averaged, modified periodogram method, cross-spectral density estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T` where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

`ltpda_lcpsd` estimates the cross-spectral density of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. Detrending is performed on each individual window.

Syntaxis

```
b = ltpda_lcpsd(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AOs containing the input time series to be evaluated. They need to be in a number N >= 2. `b` includes the NXN output objects. The parameter list `pl` includes the following parameters:

- `'Kdes'` - the desired number of averages [default: 100]
- `'Jdes'` - the number of spectral frequencies to compute [default: fs/4]
- `'Lmin'` - the minimum segment length [default: 0]
- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Order'` - order of segment detrending
  - -1 - no detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

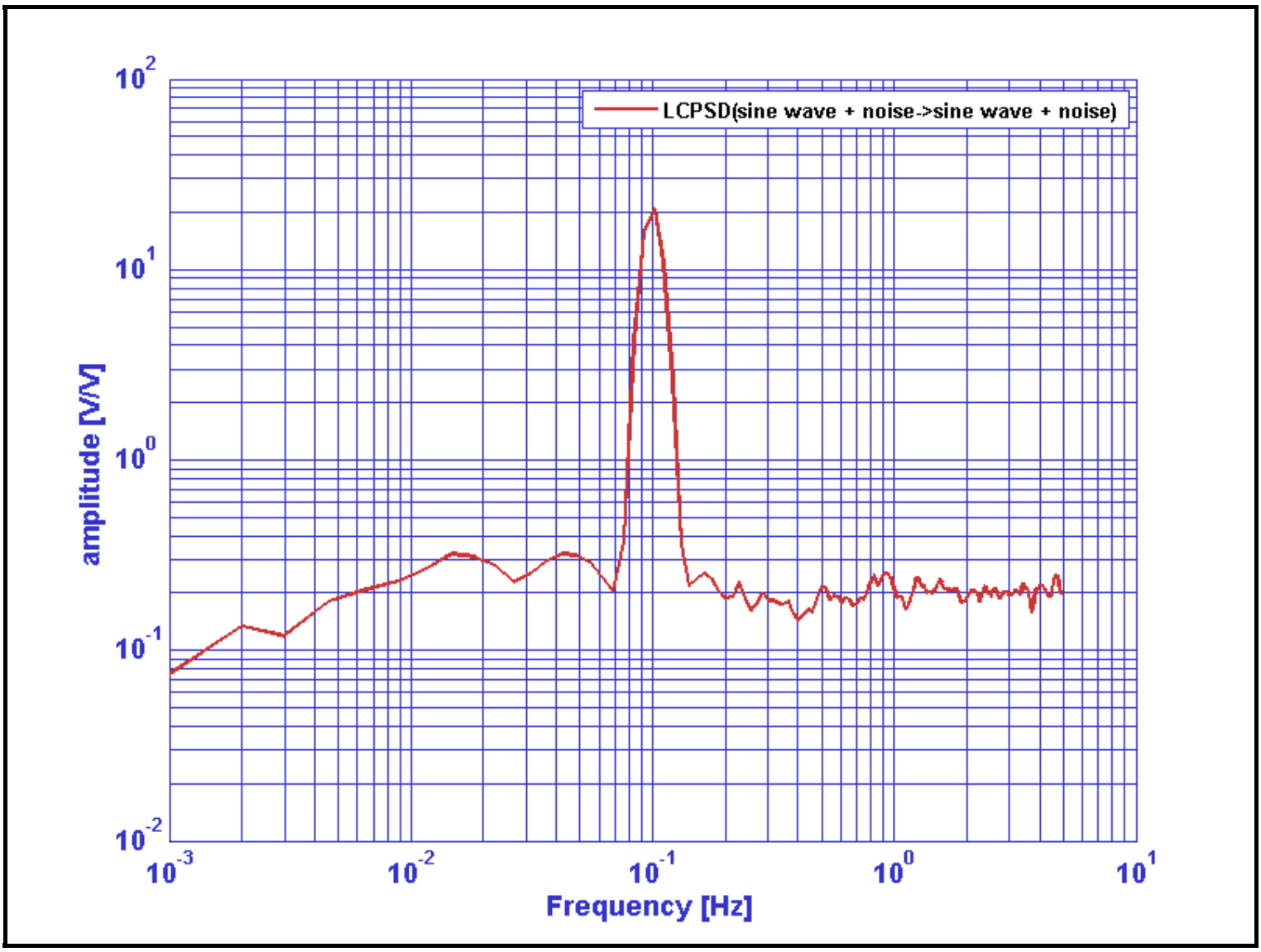
The length of the window is recalculated for each freqency bin, so the 'Win' parameter is used only to provide the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

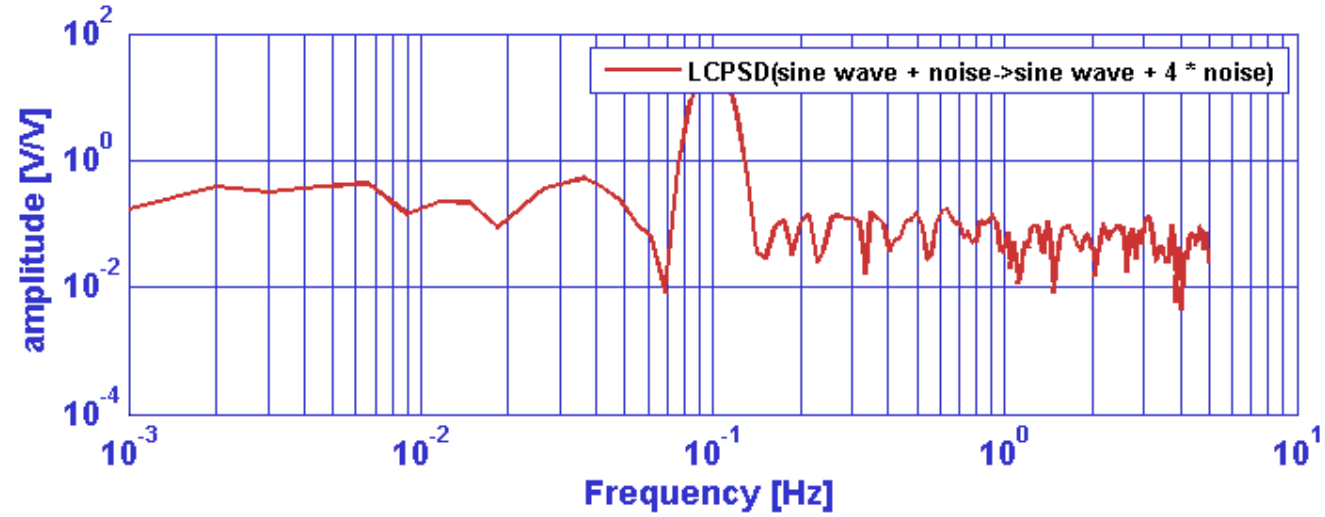
```
  If the user doesn't specify the value of a given parameter, the default value is used.
  The function makes CPSD estimates between all input AOs. Therefore, if the input argument
list contains N analysis objects, the output a will contain NxN CPSD estimates.
The diagonal elements will be S_ai_ai and will be equivalent to the output of ltpda_lpsd(ai).
```

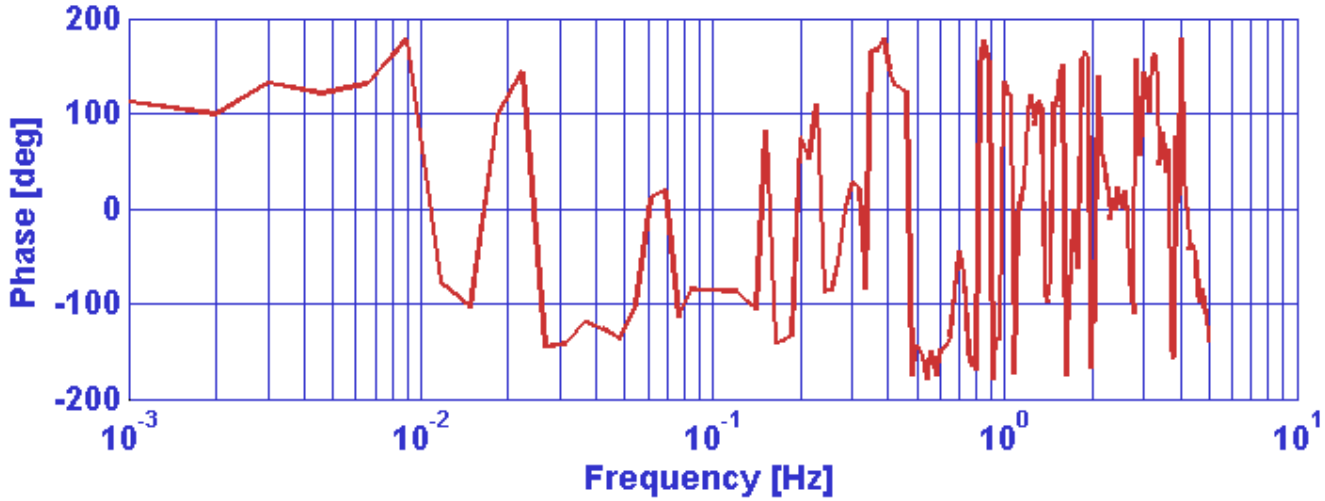
Example

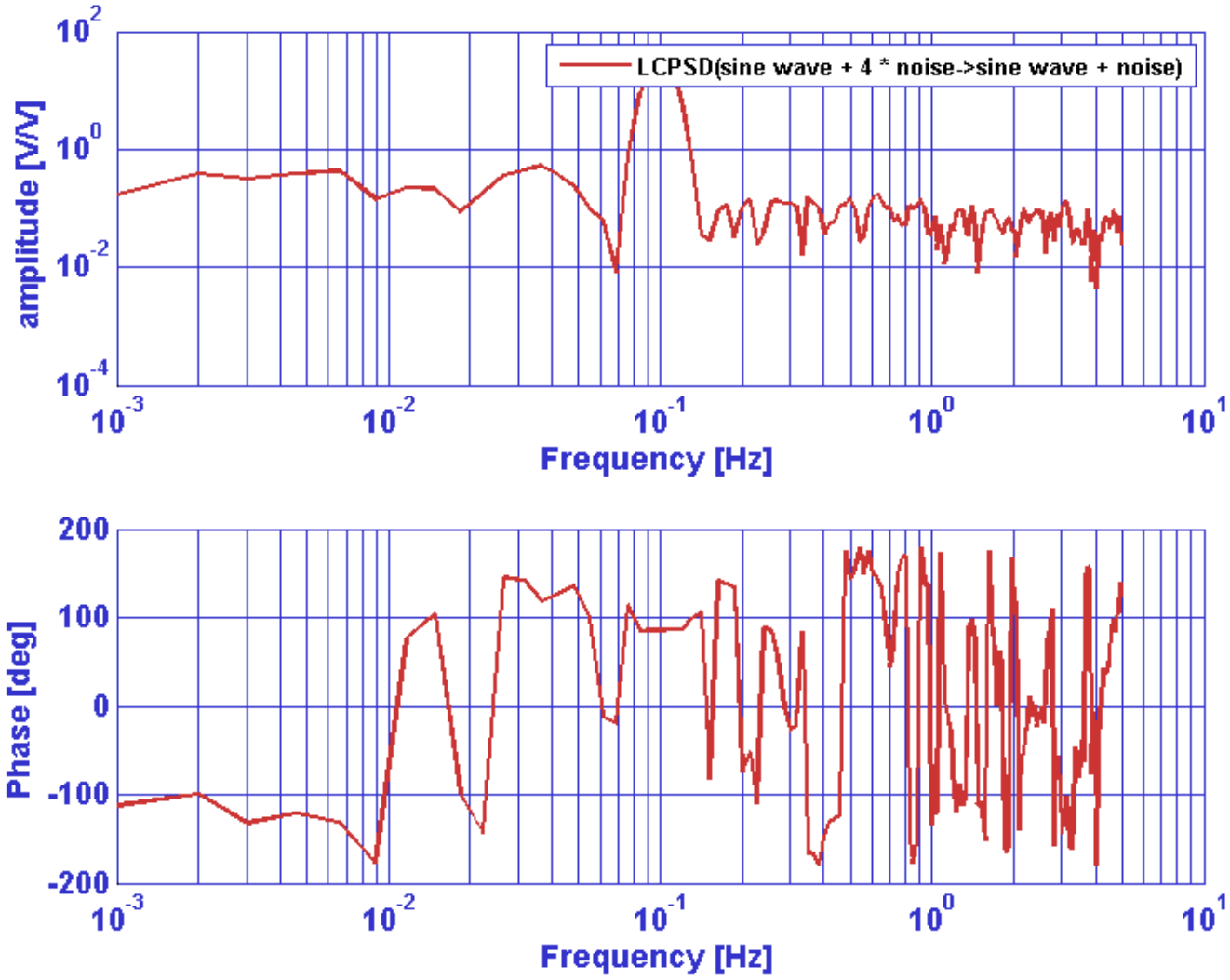
Evaluation of the CPSD of two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
    x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
    y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',1000,'fs',10,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
pl = plist('order',2,'win',specwin(plist('name','BH92')));
z = ltpda_lcpsd(x,y);
    iplot(z(1,1));
iplot(z(1,2));
iplot(z(2,1));
```


LCPSD(sine wave + noise->sine wave + noise)
amplitude [V/V]
Frequency [Hz]
10^2
10^1
10^0
10^-1
10^-2
10^-3
10^-2
10^-1
10^0
10^1


LCPSD(sine wave + noise->sine wave + 4 * noise)
amplitude [V/V]
Frequency [Hz]


Phase [deg]
Frequency [Hz]



Log-scale power spectral density estimates
Log-scale cross coherence density estimates

Log-scale cross coherence density estimates

Multivariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm, which is an application of Welch's averaged, modified periodogram method, spectral density estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T` where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

`ltpda_lcohere` estimates the coherence of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. Detrending is performed on each individual window.

Syntaxis

```
b = ltpda_lcohere(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AOs containing the input time series to be evaluated. They need to be in a number N >= 2. `b` includes the NXN output objects. The parameter list `pl` includes the following parameters:

- `'Kdes'` - the desired number of averages [default: 100]
- `'Jdes'` - the number of spectral frequencies to compute [default: fs/2]
- `'Lmin'` - the minimum segment length [default: 0]
- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Order'` - order of segment detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

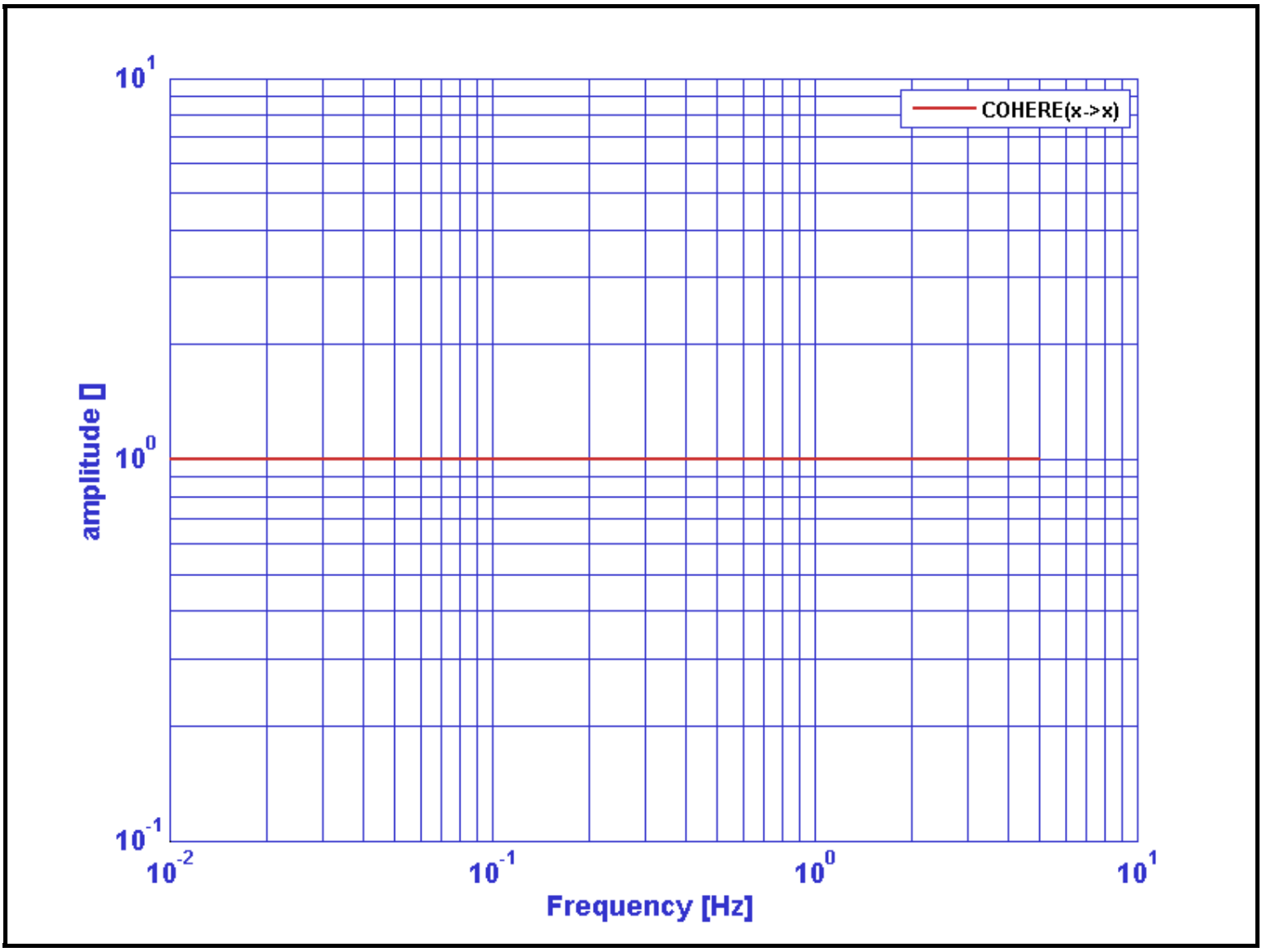
The length of the window is recalculated for each freqency bin, so the 'Win' parameter is used only to provide the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

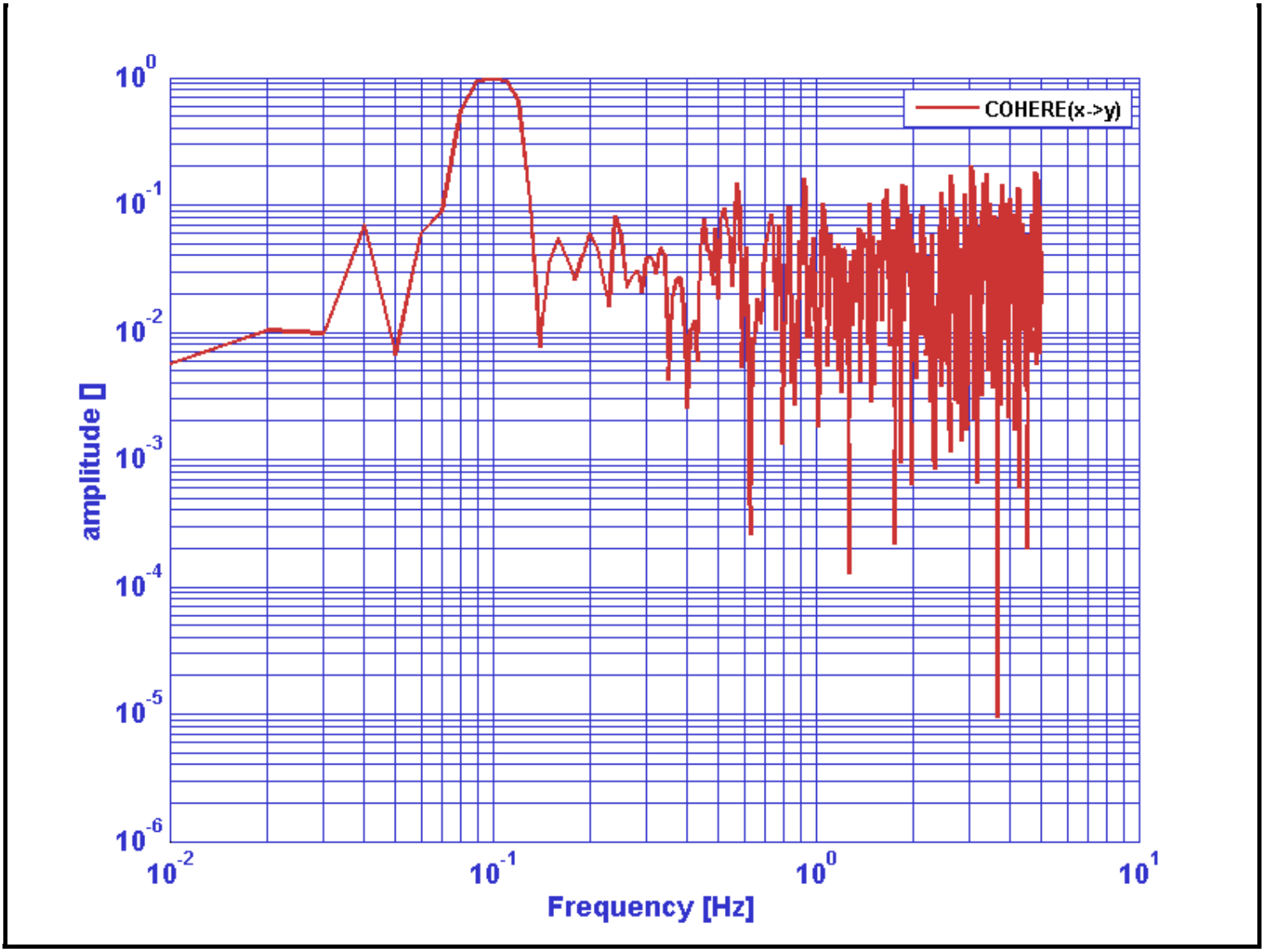
```
  If the user doesn't specify the value of a given parameter, the default value is used.
  The function makes coherence estimates between all input AOs. Therefore, if the input argument
list contains N analysis objects, the output a will contain NxN coherence estimates.
The diagonal elements will be 1.
```

Example

Evaluation of the coherence of two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
      x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10)) + ...
      ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10)) + ...
      ao(plist('tsfcn', 't','nsecs',1000,'fs',10));
      y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',1000,'fs',10,'phi',90)) + ...
      4*ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
pl = plist('win',specwin(plist('name','BH92')));
z = ltpda_lcohere(x,y,pl);
iplot(z(1,1));
iplot(z(1,2));
```


COHERE(x->x)
amplitude []
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-1}
10^{0}
10^{1}
Frequency [Hz]



Log-scale cross-spectral density estimates
Log-scale transfer function estimates

Log-scale transfer function estimates

Multivariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm, which is an application of Welch's averaged, modified periodogram method, cross-spectral density estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T` where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

`ltpda_ltfe` estimates the transfer function of time-series signals, included in the input AOs. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by polinomial of time in order to reduce the impact of the border discontinuities. Detrending is performed on each individual window.

Syntaxis

```
b = ltpda_ltfe(a1,a2,a3,...,pl)
```

`a1`, `a2`, `a3`, ... are AOs containing the input time series to be evaluated. They need to be in a number N >= 2. `b` includes the NXN output objects. The parameter list `pl` includes the following parameters:

- `'Kdes'` - the desired number of averages [default: 100]
- `'Jdes'` - the number of spectral frequencies to compute [default: fs/4]
- `'Lmin'` - the minimum segment length [default: 0]
- `'Win'` - a specwin window object [default: Kaiser -200dB psll]
- `'Olap'` - segment percent overlap [default: taken from window function]
- `'Order'` - order of segment detrending
  - -1 - no detrending
  - 0 - subtract mean [default]
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

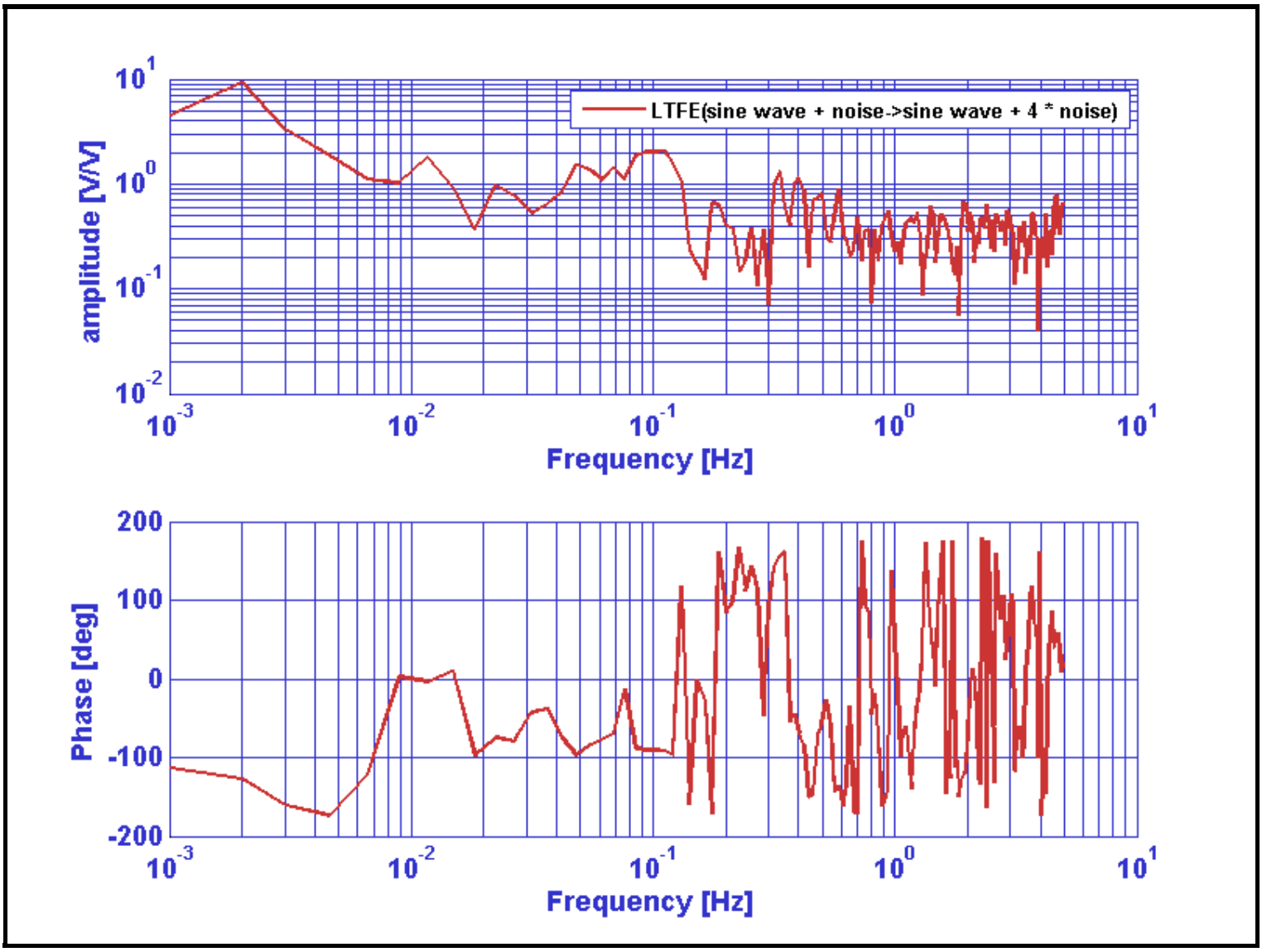
The length of the window is set by the value of the parameter `'Nfft'`, so that the window is actually rebuilt using only the key features of the window, i.e. the name and, for Keiser windows, the PSLL.

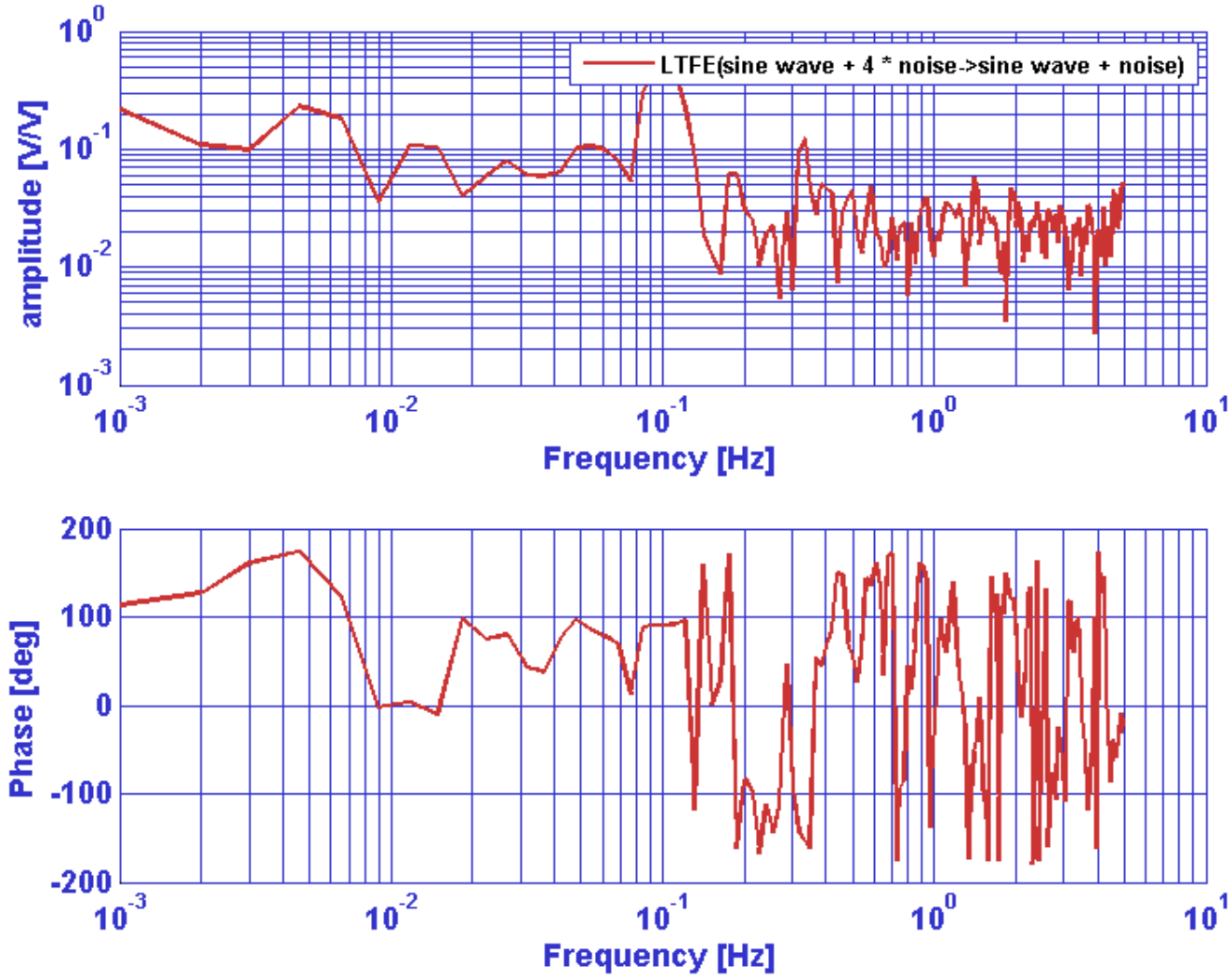
```
  If the user doesn't specify the value of a given parameter, the default value is used.
  The function makes transfer functions estimatesbetween all input AOs. Therefore, if the input
argument
list contains N analysis objects, the output a will contain NxN TFE estimates.
The diagonal elements will be 1.
```

Example

Evaluation of the transfer function between two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
      x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10)) + ...
      ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
      y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',1000,'fs',10,'phi',90)) + ...
      4*ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
  z = ltpda_ltfe(x,y,plist('win',specwin(plist('name','BH92'))));
  iplot(z(1,2));
  iplot(z(2,1));
```


LTFE(sine wave + noise->sine wave + 4 * noise)
amplitude [V/V]
Frequency [Hz]
Phase [deg]
Frequency [Hz]



Log-scale cross coherence density estimates
Fitting Algorithms

Fitting Algorithms

The following sections describe special tools for data fitting implemented in the LTPDA toolbox.

Polynomial Fitting

polyfit.m overloads the polyfit() function of MATLAB for Analysis Objects.
The script calls the following MATLAB functions:

- polyfit.m
- polyval.m

Usage

```
% CALL:        b = polyfit(a, pl)
%
% Parameters:  'N'      - degree of polynomial to fit
%              'coeffs' - (optional) coefficients
%                         formed e.g. by [p,s] = polyfit(x,y,N);
```

The MATLAB function polyfit.m finds the coefficients of the polynomial p(x) of degree N that fits the vector 'x' to the vector 'y', in a least squares sense.
After this in the script polyfit.m the function polyval.m is called, which evaluates the polynomial of order 'N' according to these coefficients.
Using the output of polyval.m the fitted data series is created and outputted as analysis object.

Time domain Fit

ltpda_timedomainfit.m uses the MATLAB function `lscov.m` to fit a set of time-series AOs to a target time-series AO. It gives back a set of fitting coefficients.

One can now subtract the fitted time series from the original one - the target- and produce a new time series by calling ltpda_lincom.m with the calculated coefficients. This function does a linear combination of the inputted coefficients and analysis objects and subtracts the result from the target analysis object, which has to be the first input parameter.

An example:

```
coeffsAO = ltpda_timedomainfit(target, ts1, ts2, ts3, ts4);
%% Make linear combination
x12ns = ltpda_lincom(target, ts1, ts2, ts3, ts4, coeffsAO);
```

`x12ns` represents the noise subtracted target analysis object. With noise here the fitted time series is meant. That is the linear combination of the coefficients `coeffsAO` and the time series objects ts1 to ts4.

The number of time or frequency series analysis objects is variable. The first is always taken as target object.

Graphical User Interfaces in LTPDA

LTPDA has a variety of Graphical User Interfaces:

- The LTPDA Launch Bay
- The LTPDA Analysis GUI
- The LTPDA Repository GUI
- The pole/zero model helper
- The Spectral Window GUI
- The constructor helper
- The LTPDA object explorer
- The quicklook GUI

 Time domain Fit The LTPDA Launch Bay 

The LTPDA Launch Bay

The LTPDA Launch Bay GUI allows quick access to all other GUIs in LTPDA. To start the Launch Bay:

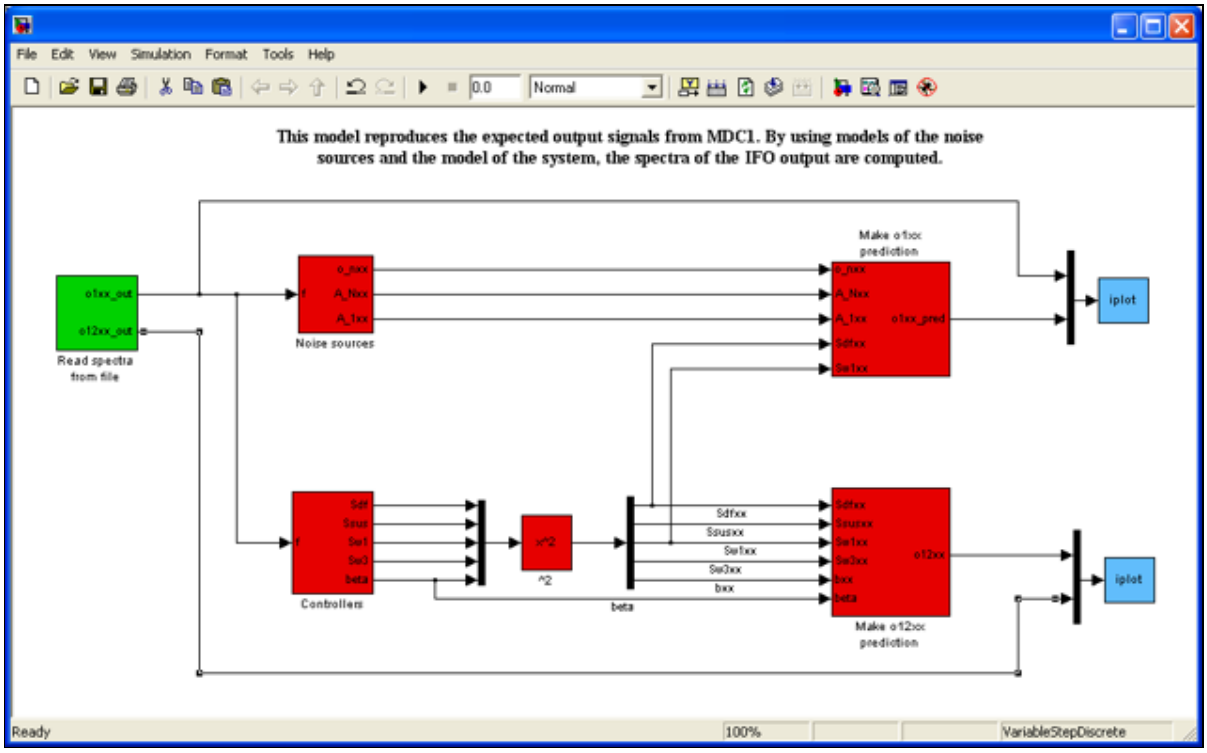
```
>> ltpdalauncher
```

The Launch Bay is automatically started from the `ltpda_startup` script.

The LTPDA Analysis GUI

Introduction

The LTPDA package contains a sophisticated graphical programming interface which allows the user to construct and execute data analysis pipelines entirely graphically.



The Simulink drawing interface is used to construct the analysis pipelines. The blocks that compose the analysis pipelines are drawn from the LTPDA blockset. The LTPDA Analysis GUI then executes the commands represented by each block in the appropriate order in order to achieve the desired analysis pipeline.

To start the LTPDA GUI use the command `ltpdagui` .

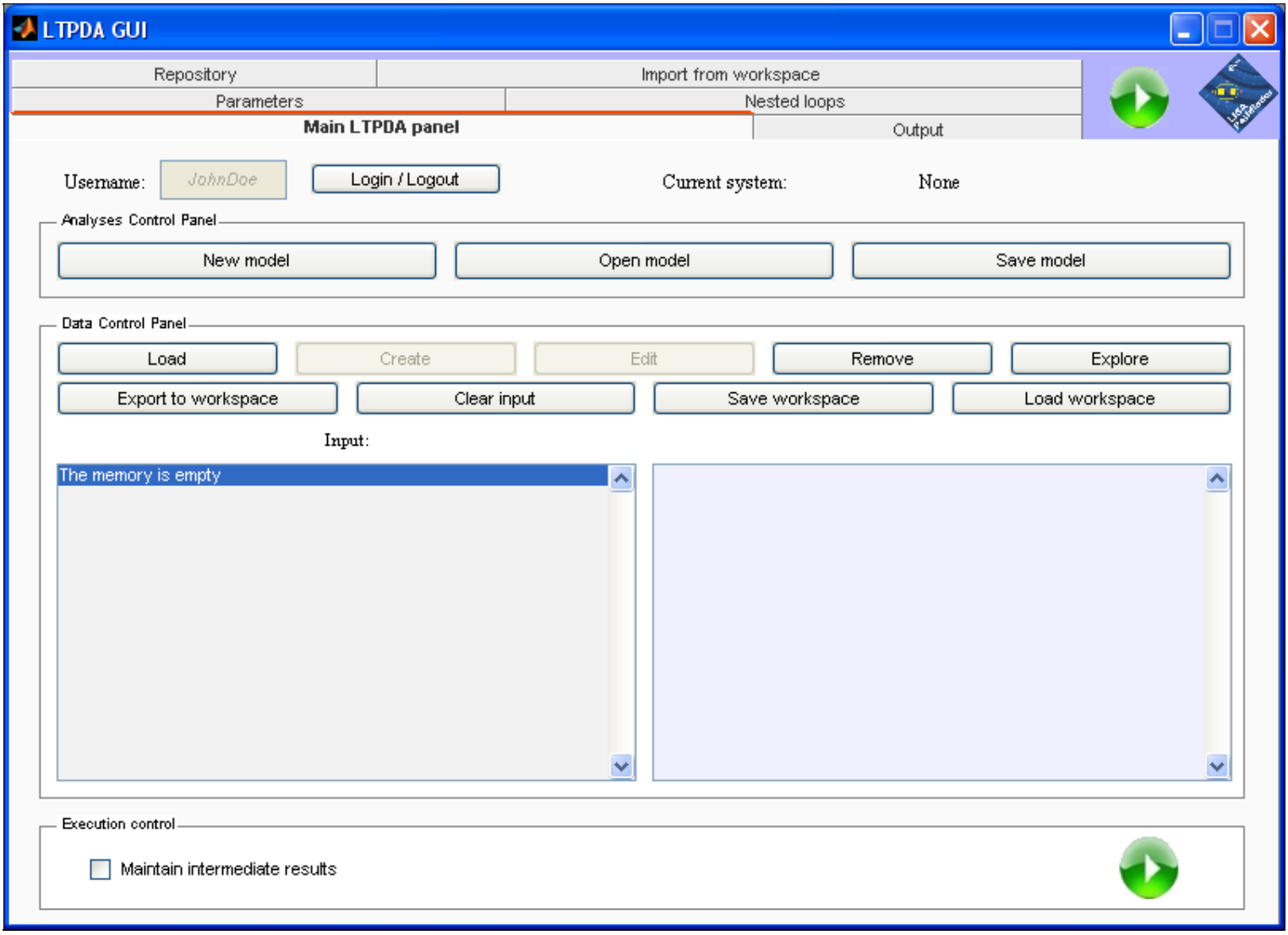
```
>> ltpdagui
```

If there's no open LTPDA GUI window, the user will be asked to log in:

These data are not mandatory, as long as the user do not want to connect to a LTPDA Repository, for example to retrieve previous analyses or to send in his own work. If the user just wants to use the software to compile trials or personal analyses, not meant to be shared with others, he can leave these fields empty: the GUI will identify him as anonymous ('JohnDoe'), and this name will be stored into every produced model.
On the contrary, if the GUI has to produce shared results, i.e. if the user want to submit/retrieve from an LTPDA Repository, these login and password are those necessary to access the repository database. The user must possess a valid account, which can be requested to the database administrator.
The username will be used also to ensure better traceability, storing the name itself into a short description created using the LTPDA GUI.

However, it's possible at anytime to log off and re-log in, using different data, so it will be possible to type in valid credentials to access a repository only when needed.

The first appearance of the GUI is:



The upper part of the window lists all the possible panels: clicking on the corresponding tab the proper panel will be drawn below.

The help associated to each panel will be dealt into different pages.

The LTPDA Launch Bay Basics

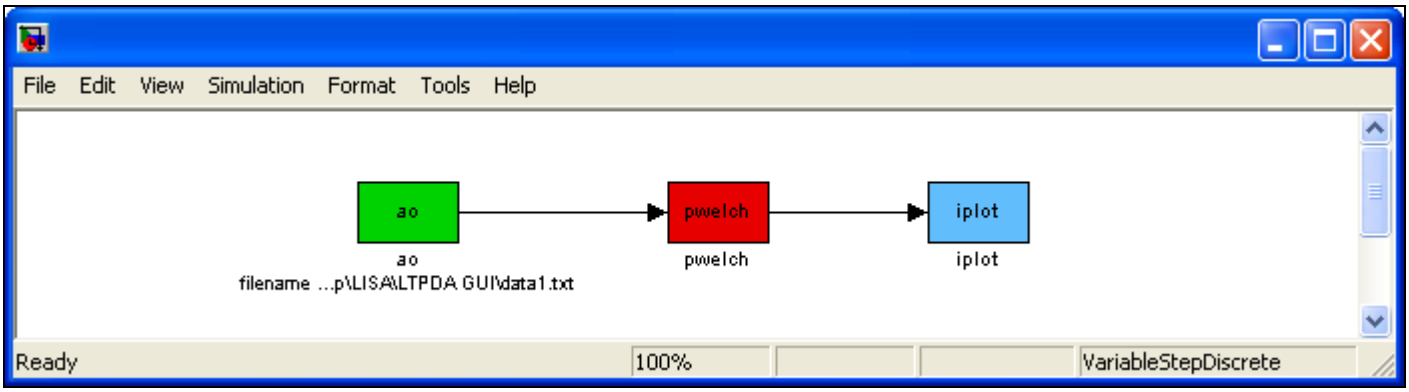
Basics

Data handling in Simulink

Since SIMULINK can only send among blocks a limited set of object types (that is, inside Simulink lines can only travel numbers, vectors or Simulink defined signals), **the main concept about LTPDA GUI and LTPDA Simulink models is that what's passed among blocks is not a true LTPDA object (an AO, a filter, a specwin or so), but rather a pure number, intended as an index inside a given array of objects.**

Every analysis block in a LTPDA model produces the output of its calculation, then stores this output into a global array - which will be accessible by all following functions/blocks - and send back to Simulink just the index of this newly added object inside the array.

For example:



The first block load create an AO from a file on disk. This AO is saved in a global array: assuming the array was empty, this is the 1st object, so '1' is the number sent by the first block into Simulink.
The second block receives this '1': the corresponding (1st) object is retrieved from the array, then the analysis function (`pwelch`) is applied. The output is appended back into the array, as 2nd element. So, '2' is the content of the signal leaving the second block.
The third block receives '2', retrieves the corresponding (2nd) object from the array and produces a plot of it.

Thus, during the analysis cycle the array of objects becomes populated: unless differently set, all the temporary objects created during the analysis will be canceled at the end of the analisys itself.

The LTPDA GUI is based on two global arrays:

- The aforementioned '**Input array**', which encloses all the LTPDA objects created by the user (for example, loading from file) and the objects generated during the analysis.

This is shown into the main panel of the LTPDA GUI.
- The '**Output array**', meant as a way to separate primary objectives of the analysis from the other - working - objects, contained in the Input array.
  This is shown into its own panel, the Output panel.

The only way to save an object into the Output array is using the proper 'Send to Output' block, which will copy the object passed from the Input to the Output array.

Parameters

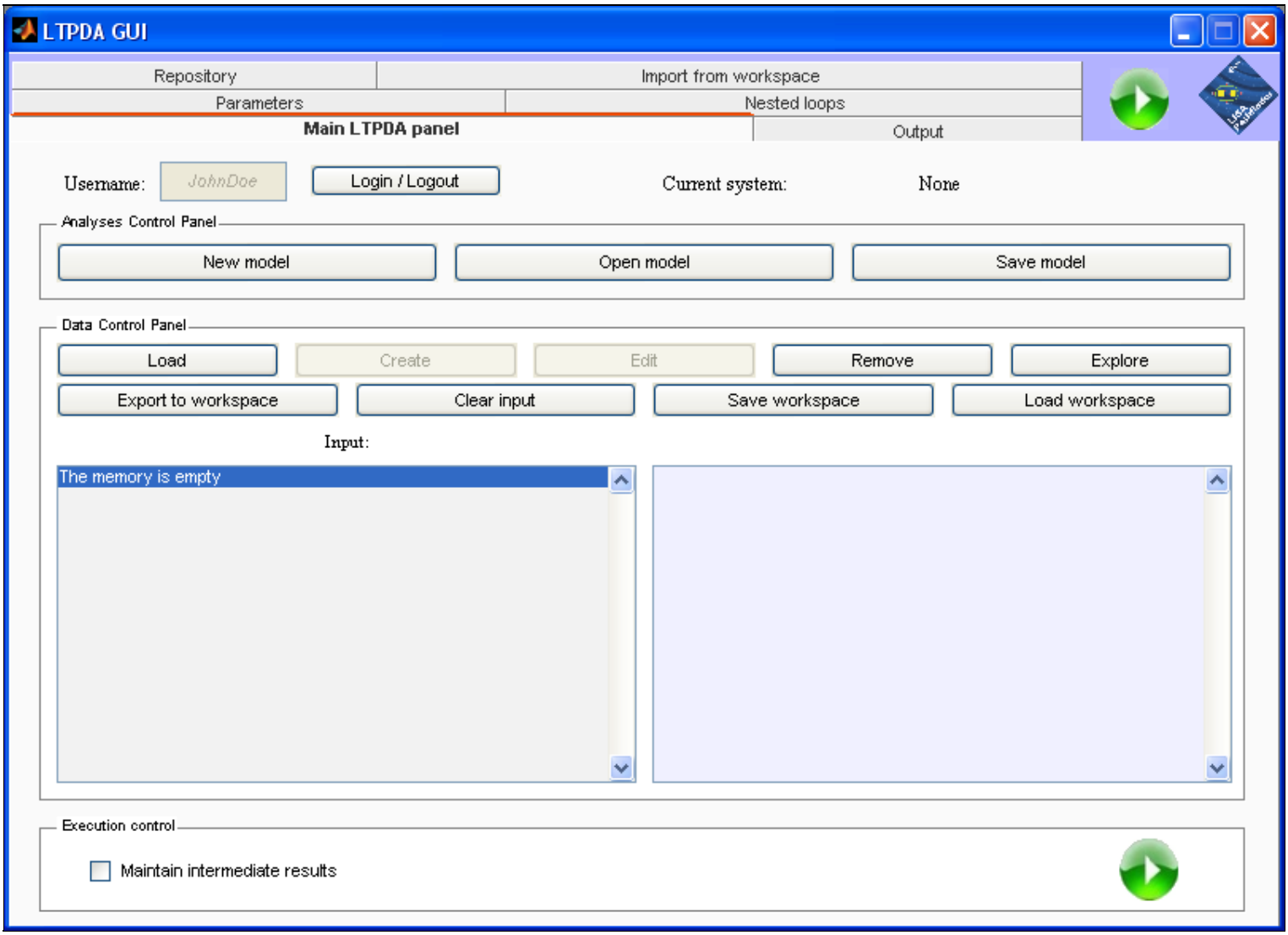
In order to enclose all the parameters of each block together with the Simulink model file, each parameters list is converted into a string and saved into the Block Description, a specific field of every block in Simulink.
Every time the user selects a block inside a valid LTPDA model the LTPDA GUI will automatically read this description to convert it back to a proper plist. 'Valid' LTPDA models are those created by the GUI, thus containing the short annotation with the author's name, time and date of creation and so on.

For more information on parameters handling, see the paragraph regarding the Parameters panel.

The LTPDA Analysis GUI Main panel

Main panel

The appearance of the main (first) panel is:



Access and selection control

In the upper part of the panel is shown the username typed in by the user when the LTPDA GUI was started.
If the login field was left empty the GUI will consider the anonymous '`JohnDoe`' user; the password field in this case has been ignored, was it empty or not.

This username is used to identify the current user of the system, and it's stored into the annotation of every analysis created by the GUI, clicking on the '`New model`' button.

Is possible to change the active user clicking on the '`Login/Logout`' button; this is to be used when:

- The username do not respect the current active user of the system.
- The username and/or password typed in aren't valid credentials to connect a LTPDA Repository, and the user wants to submit/retrieve something.

The current user is leaving the computer, to prevent other users to submit by mistake personal objects to a repository with a wrong account; this would make difficult to keep results traceable.

On the right the GUI shows the currently active model, if the selected model is a valid LTPDA one. 'Valid' LTPDA models are those created by the GUI, thus containing the short annotation with the author's name, time and date of creation and so on. If the selected model isn't recognized by the GUI '`None`' will be shown here, and no selection control will be performed upon what the user clicks inside that model.

Analysis control panel

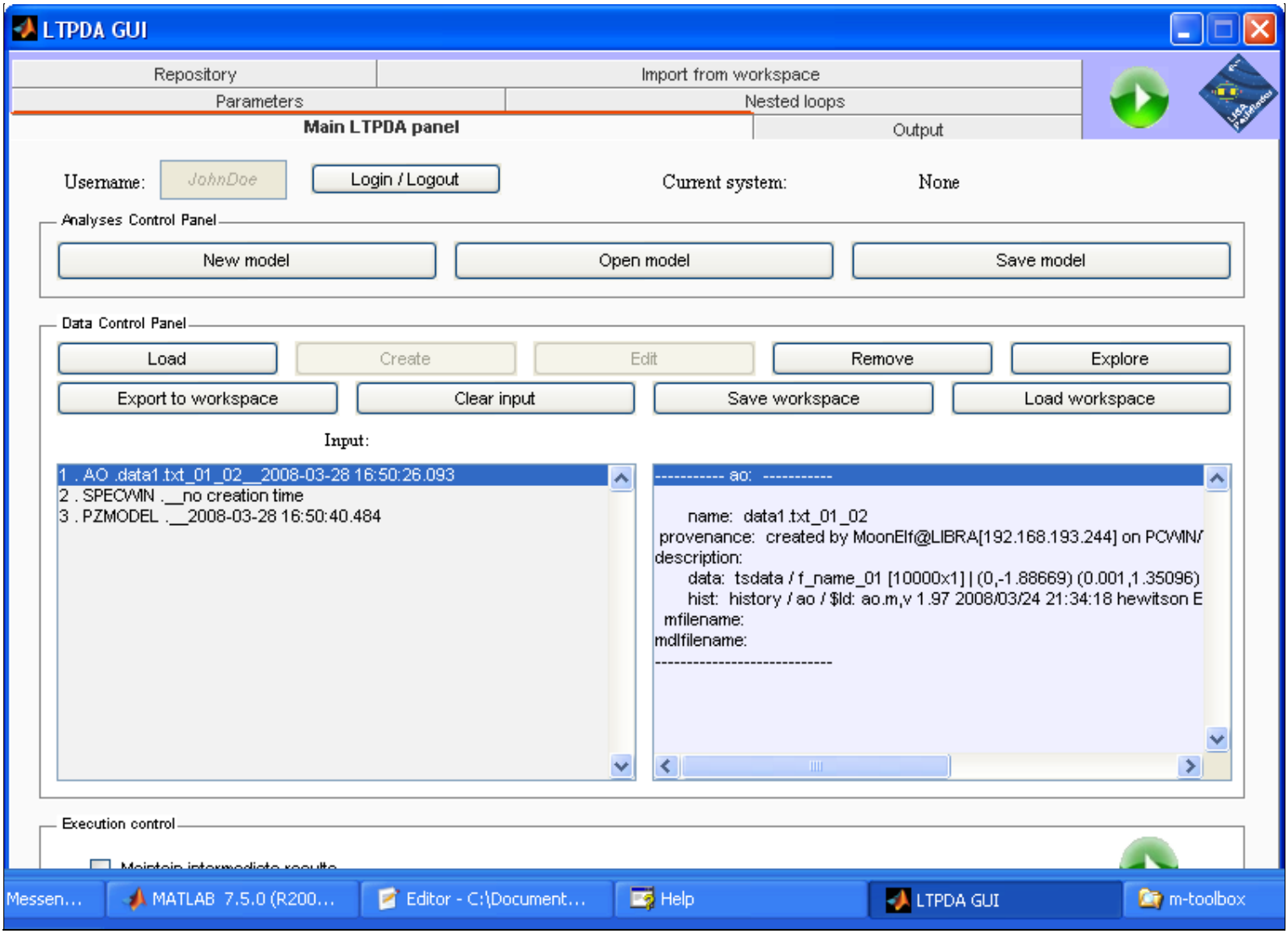
This contains 3 buttons, all devoted to the control of LTPDA Simulink model files.

- '**`New model`**': it creates an empty valid LTPDA Simulink model. The annotation inside will be created automatically to respect the current user, time and date, IP address, operative system running and MATLAB/LTPDA versions.
- '**`Open model`**': similarly to the common '`Open`' command in Simulink, this button loads an existing Simulink file from disk. If the file contains a valid LTPDA model, then the GUI will perform the selection check when it's selected.
- '**`Save model`**': lets the user to save the currently selected model. Unlike the common '`Save`' command in Simulink this will prompt the user with the assigned filename (username + time/date of creation) and the predefined folders for analyses, if any was set in the `ltpda_startup` file. Anyway, the user can freely rename this to any name, withstanding the common Simulink filename limitations.

Data control panel

Containing 9 buttons and the summary of the Input array, this part is devoted to the control of Input objects (i.e., data created by the user or generated inside a previous analysis run).

- '**`Load`**': to load data from ASCII file (to be converted into an AO), or from .MAT or .XML files. If the selected MAT or XML file contains any valid LTPDA object, this is imported into the Input array.
- '**`Create`**': *Reserved for future use.*
- '**`Edit`**': *Reserved for future use.*
- '**`Remove`**': to remove the selected object(s) from the Input array. The numbering of the remaining objects is not altered, to prevent problems in the indexing of objects into the analysis models opened.
- '**`Explore`**': to pass the selected object(s) to the `ltpda_explorer`, to explore them into an expandable and browsable window.
- '**`Export to workspace`**': to save the selected object(s) into the MATLAB workspace. The name of these objects will be automatically set.
- '**`Clear input`**': to clear the entire Input array.
- '**`Save workspace`**': even if this button is situated only inside the main panel, the `Save workspace` function it recalls affects not only the Input array, but the Output array too. It's meant as a solution to capture the entire current status of the LTPDA GUI, saving it into a .MAT file on disk.
- '**`Load workspace`**': as the `Save workspace` button, it lets the user to load from a .MAT file on disk the entire saved status of the Input and Output arrays. The loaded contents are appended to the existing arrays in memory.



The central part of the window contains two boxes: on the left a list representing the contents of the Input array, on the right the details of the object currently selected in the list aside.
Note this will list only LTPDA objects: if any function/block in the analysis produced as output a simple numerical vector or matrix it won't be visible here.

The single click is used to select, to have a look of the details of a single object in the Info panel on the right.
Double clicking on the list will add to the active LTPDA Simulink model (if any is selected) a "Object from list" block, so to import the object double-clicked in the given model.

Execution control panel

At the bottom of the window there are:

- '**`Maintain intermediate results`**': enabling this ALL the outputs produced during the analysis (i.e., the output of every block) will be kept at the end of the analysis itself. This is meant as a way to verify intermediate results without the need to enable the 'Maintain' option for every single block, but of course should be used with care, since it will make the Input array very long and populated in a short time.
- '**`Start button`**': this button starts the execution of the analysis. Till the end of the analysis it will turn to red. Although the common Simulink '`Start simulation`' button will correctly execute the model, the user should anytime use just the button enclosed in the GUI, which fulfills many extra purposes; for example, it will update the Input array list in the main panel, removing all

the intermediate results (whenever not differently set), it will open the progress bar window to represent the analysis proceeding, and it will add terminators in the model wherever necessary.

To make easier the start of the analysis, a direct copy of this button is shown in the upper right corner of the window, outside the first panel. This is visible from every panel, so it will be possible to start the execution without the need to open the main panel first.

Basics Parameters

Parameters

The parameters panel will be empty if:

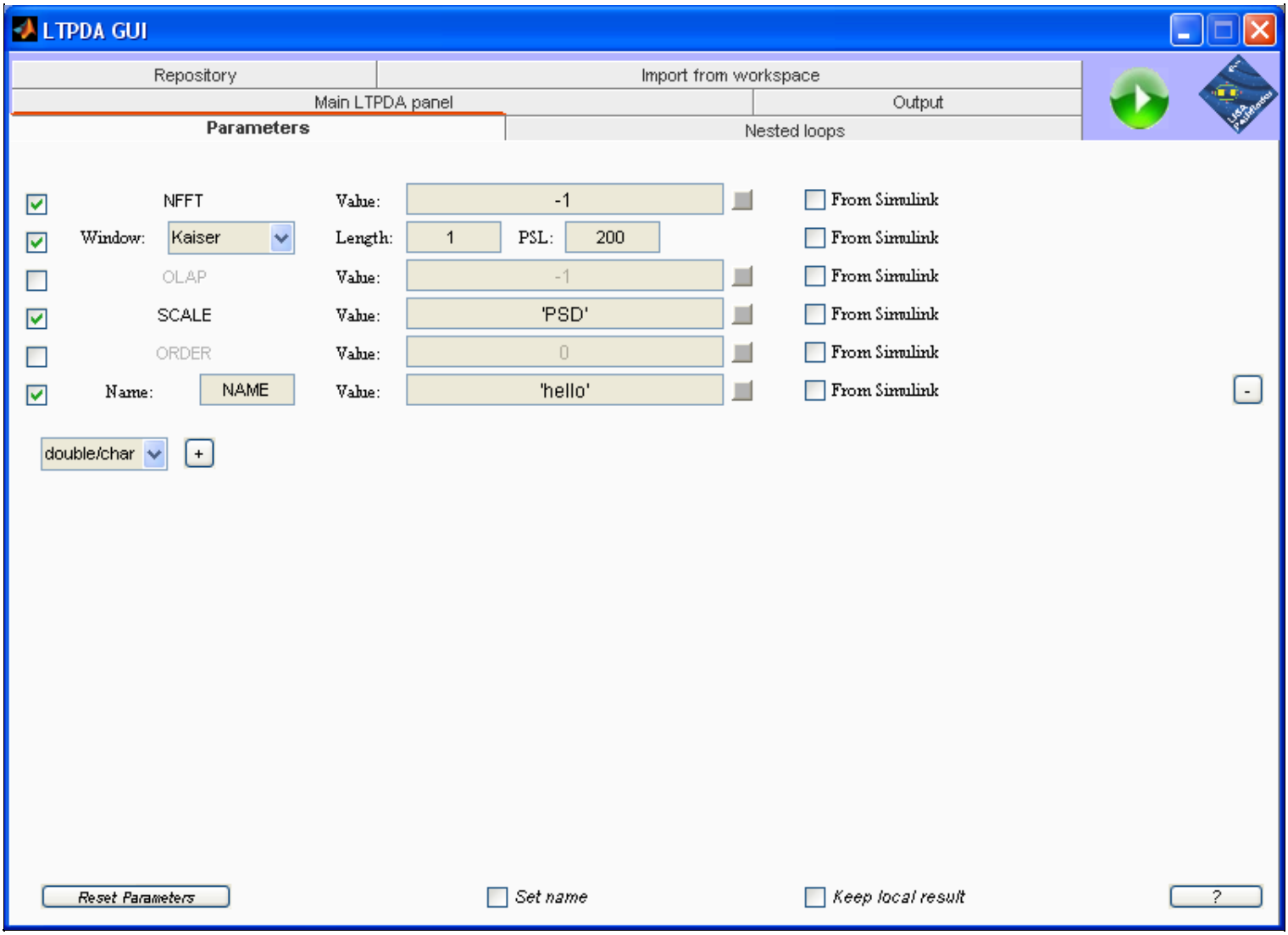
- No block is currently selected.
- The currently selected Simulink model is not a valid LTPDA model.
- The selected block it's not a valid LTPDA block (i.e., it do not contain a proper LTPDA function which answers the call for parameters).

Otherwise, the contents of the parameters panel will change accordingly to the user selection in the active LTPDA Simulink model.

- If a **function/method** block is selected, the panel will show the proper corresponding parameters.
- If a **Object from list** block is selected, the panel will show the referenced index (that is the only parameter accepted).
- If a **Mux/Demux** block is selected, the panel will show the number of inputs/outputs.
- If a **From** block is selected, the panel will present the 'Find origin' button.

Function/method parameters

If a valid LTPDA function or method block is selected (inside a valid LTPDA Simulink model), the GUI will first ask the inner function or method for its required parameters; the panel then will be built accordingly.



The window is built automatically: **every line corresponds to a different parameter**, among those required by the selected function.
Inside every line, then, the structure is obviously similar, so different parameters can have the same fields and boxes.

Required or added parameters

In the construction of the panel the GUI will consider whether the parameter was among those required by the function, or if the parameter was added by the user. In this latter case it will be passed to the function, which can use it (if someway expected, for example if it a possible optional parameter) or ignore it, if just unexpected. The GUI won't change the behavior of the inner function, so it's up to the user to be aware of what can be passed to be used by every function.

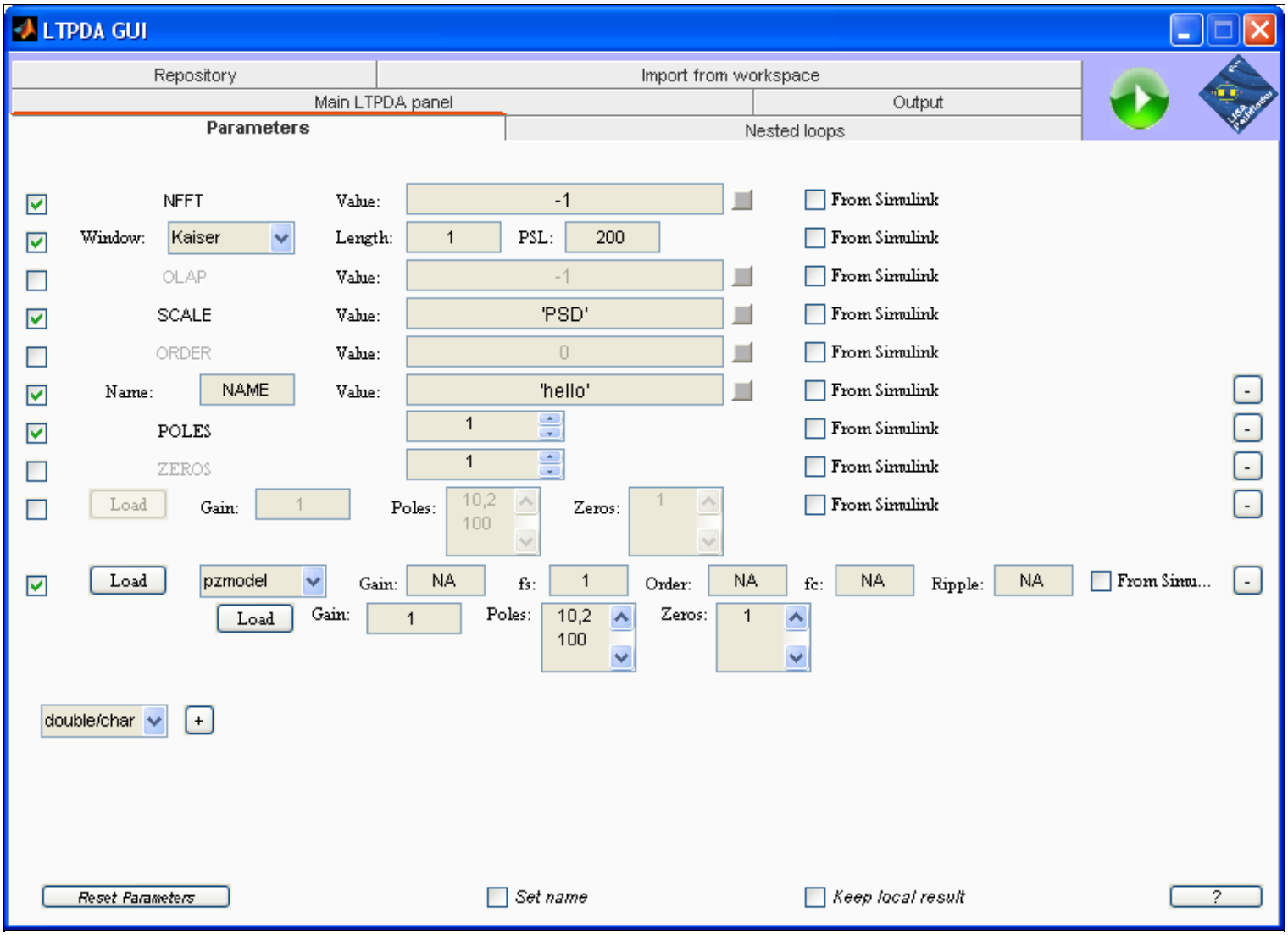
In the parameters panel the difference among required and added parameters is shown on the far right: added parameters have a small '-' button, which lets the user to remove it; required parameters have no such button, since the corresponding parameter was automatically required by the function and not added by the user.

Please note that a parameter can also be disabled (see 'Enable checkbox', below): since a disabled parameter won't be passed to the function, there's no difference in removing an added parameter or just disable it. The only difference of course is a disabled parameter can be re-enabled in a second time, while a removed parameter has to be added and set over again.

Enable checkbox

On the left, at the beginning of every line/parameter, there is a checkbox which lets the user to enable or disable the corresponding parameter. If it's enabled the parameter will be passed to the function, so it's up to the user to set the proper value. If it's disabled, on the contrary, it won't be part of the plist passed to the function, which thus will use the default value (if any is expected).

The central part of the window is built line by line to reflect each parameter, so it can differ a lot.



Parameter name

Wherever expected, the first element on the left, aside the Enable checkbox, is the parameter name. For **double/char** parameters this means the parameter key; note that if the parameter was required by the function, the name is fixed (see parameter 1,3,5,6 in the previous image), while for added parameters it can be changed (see parameter 7).
In case of an added parameter the GUI will show both a text '`Name:`' and an edit field, where the user can type in the parameter key itself.

For **specwin** parameters the key is fixed, so the GUI shows directly the name of the window type ('`Kaiser`', '`Rectangular`',...).

Similarly, for **Poles and Zeros** parameters the key is fixed, so even for added parameters '`Poles`' and '`Zeros`' are shown and cannot be altered.

Since the key is fixed also for **MIIR and PZmodel** parameters, the GUI shows directly the inner parameters to construct these objects; note that this is just a way to build a MIIR object, typing in

directly the plist. To build it in a different way, for example retrieving from the Repository, must be used a proper Constructor block, setting the parameters to be retrieved from Simulink (see 'From Simulink checkbox', below) and connection the Constructor block to the function port having as a name the corresponding parameter's name.

Parameter value

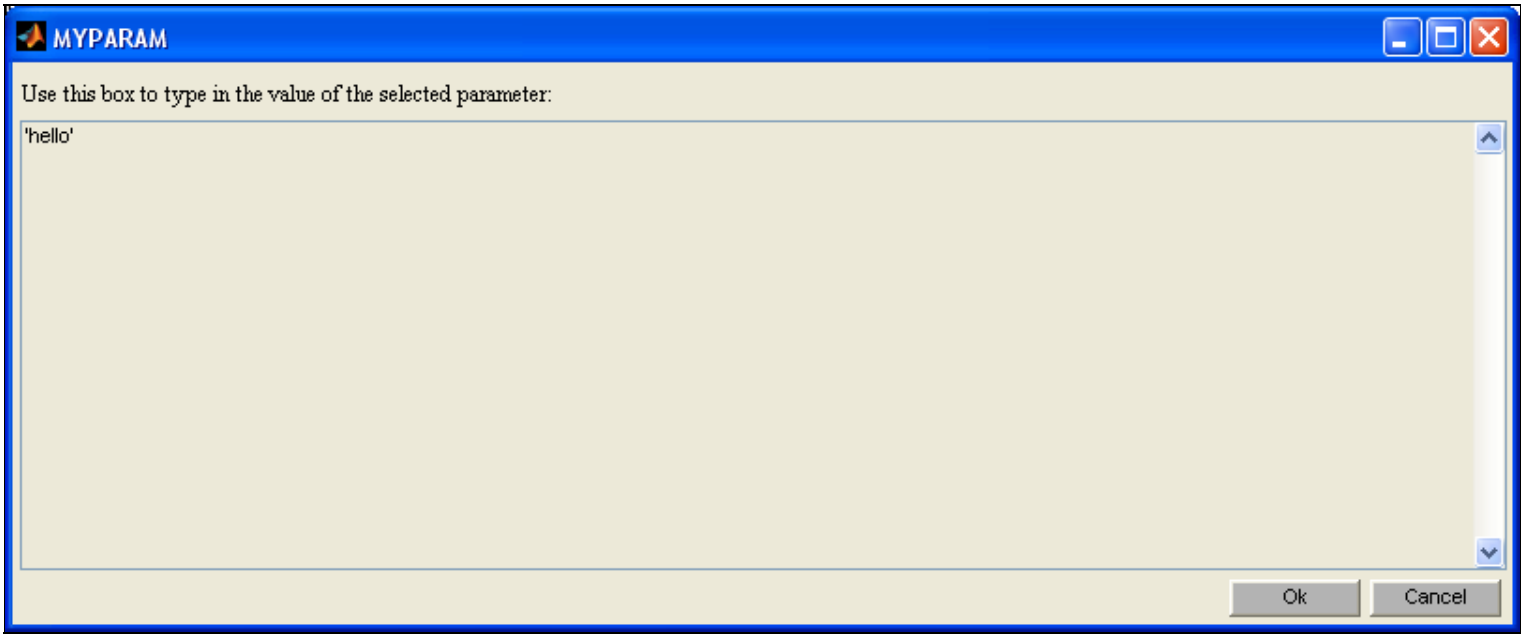
The following field lets the user to type in the parameters value. This is true for every '`Value:`' field in the parameters panel.
Please note that often is possible to have a brief help text in the tooltip string, just moving the mouse pointer over the edit field and waiting a second.
This will explain for example the meaning for the different edit fields in a specwin parameter, or it will explain how to type in poles and zeros into their corresponding edit fields.

Inside an edit field the user can type in using the traditional MATLAB syntax:

- **10**, will be considered a pure number. The parameter's value will be of course the number itself.
- **[1 2 3]**, among square brackets, will be converted into a vector (or matrix). Similarly, using {} will create a cell array. The parameter's value will be a vector, a matrix or a cell array.
- **'Hello'**, among quotes, will be considered a string, and it won't be evaluated. The parameter's value will be a string.
- **rand(1)**, without quotes, will be recognized as a command and it will be evaluated immediately. The parameter's value will the output of the evaluated string.

Expand edit field button

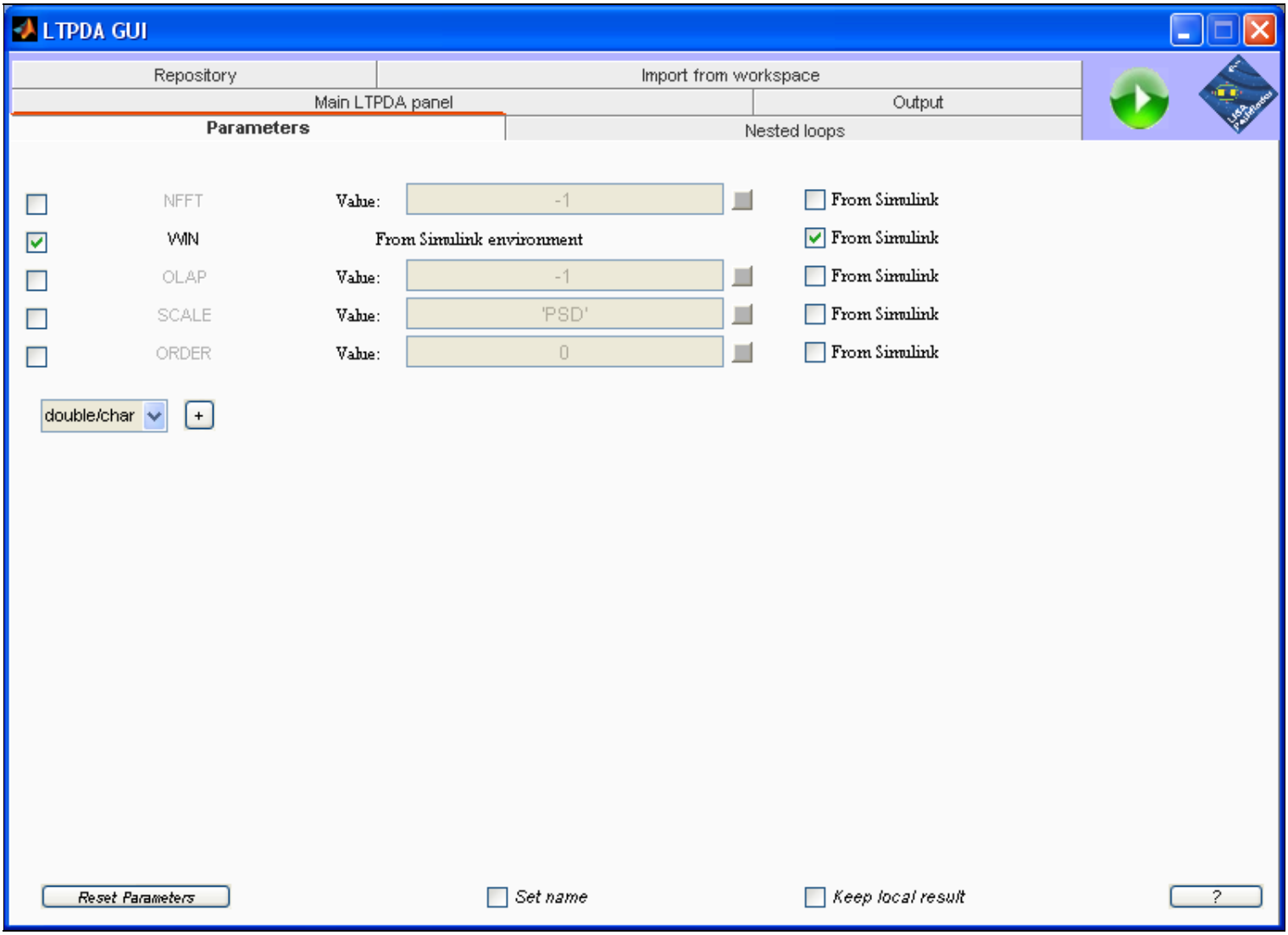
The small gray button aside each parameter's value edit field, for double/char parameters, will open up a window with a bigger edit field, where the user can type in the value more comfortably than in the small one in the panel. This is particularly handy for those case when the parameter's value is a long and complicated command or vector, which can't fit easily in the small space given in the panel.



From Simulink checkbox

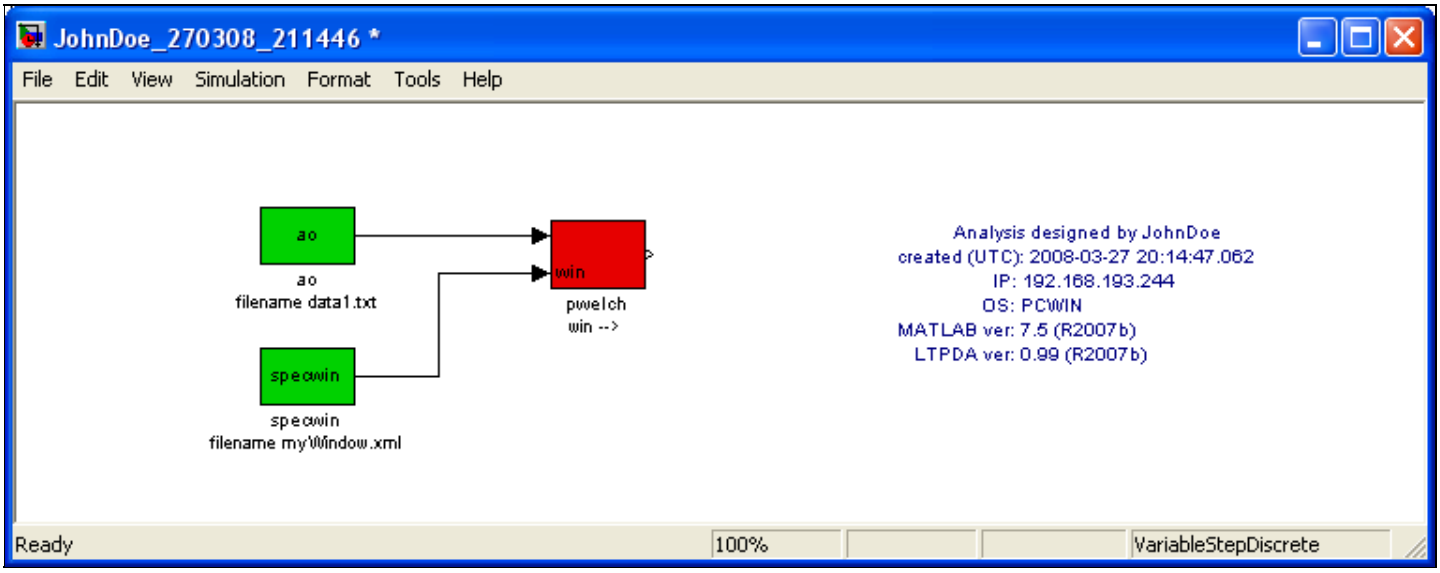
If the parameter is not supposed to be set in the GUI's panel but it must be retrieved from Simulink (for example, using a Constructor block for that particular type of object), the user can click on the

small '`From Simulink`' checkbox.



The line will be updated, showing 'From Simulink environment'. Since this parameter will be retrieved from Simulink, the user cannot alter it anyway in the current parameters panel.

In the LTPDA Simulink model, the block will be modified with a new inport, with the same name as the parameter's key:



While the first inport must be connected to the data coming in (if any is expected), as usual, the second port must be connected to the constructor block providing the object which will become the parameter.
In the example shown, the specwin Constructor block provide the specwin object that, inside the '`pwelch`' block, will be retrieved as a parameter and passed to the function.

If the block had no data input, i.e. no input prior to the 'From Simulink' parameter, it will present a single inport, which must be connected to the Constructor block providing the parameter's contents.

Since it's possible to set multiple parameters coming from Simulink, the user should take care of the connections to the block, to avoid to shuffle the connected Constructor blocks - for example, inverting a specwin object and a MIIR object.

Load button

For MIIR and PZmodel parameters the GUI will show also a '`Load`' button: the purpose of course is to retrieve these objects from a file on disk. Please note that for now this will convert the loaded object into its constructive plist, if possible, and the GUI will show again the parameters of the loaded object into the common parameters panel.
This will be discontinued or modified after R1

Adding parameters

Immediately below the last parameter, on the left, the GUI shows a popup list containing all the parameters types which can be added, and aside a '+' button to add a parameter of the selected type. Since it makes no sense to pass to a function 2 specwin object, PZmodels, MIIR filters, poles and zeros lists, these types won't be available in the popup list if any object of the same type is already shown in the parameters panel.

Reset parameter

In the lower left corner of the window the GUI draws the '`Reset parameters`' button: this will reset the block to its empty initial state. If its a function/method block the GUI will then ask again for its list of required parameters, just like the block was just added to the model from the library.

Set name checkbox

At the bottom of the window, center line, there is the '`Set name`' checkbox.
Enabling this the name of the output of the selected block will be assigned equal to the block's name (which can be altered freely by the user, standing the limitations on the block's name posed by Simulink); this will let the user to set particular and meaningful names instead of the automatically assigned ones, produced by the inner calculation functions.
The plot of an object assigned a particular name, for example, will show the proper name set.

Keep local result

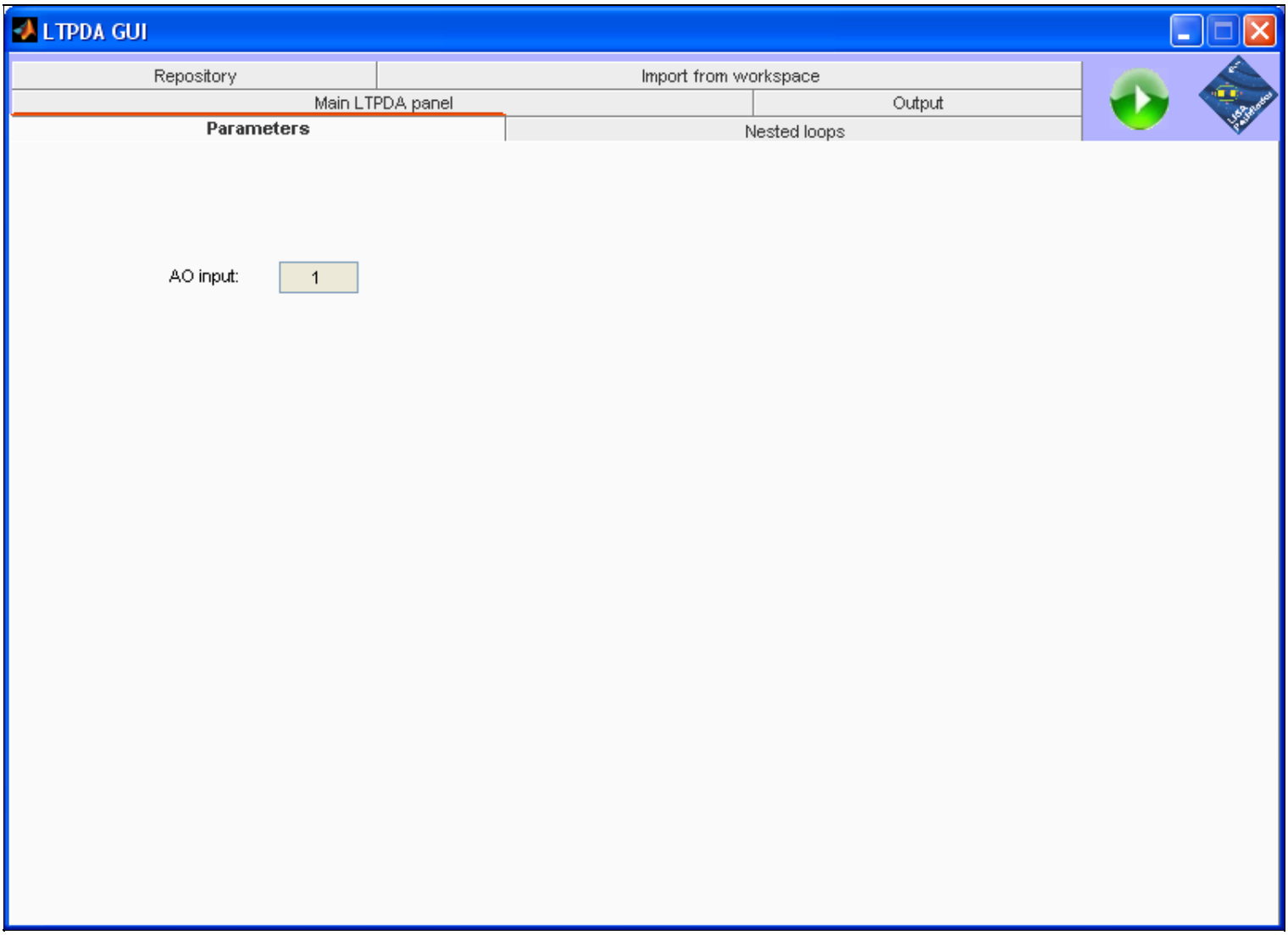
Similarly to the '`Maintain intermediate results` checkbox in the main panel, this checkbox will mark the output of the selected block to be maintained at the end of the analysis, instead of being deleted just like any other intermediate result of the calculation.
In the LTPDA Simulink model the block will be shown with a purple background, to mark it as a 'probe' and to make it immediately recognizable.
Disabling the checkbox the block's background will be back to its common color, and the intermediate result produced by the block itself will be cleared at the end of the analysis.

Help button

In the lower right corner of the window the GUI draws the '?' help button: this will recall the help associated to the function or method contained into the selected block, and it will show it into a new window.
The help shown is exactly the same available by the MATLAB command line, typing `help function_name`.

'Object from list' parameter

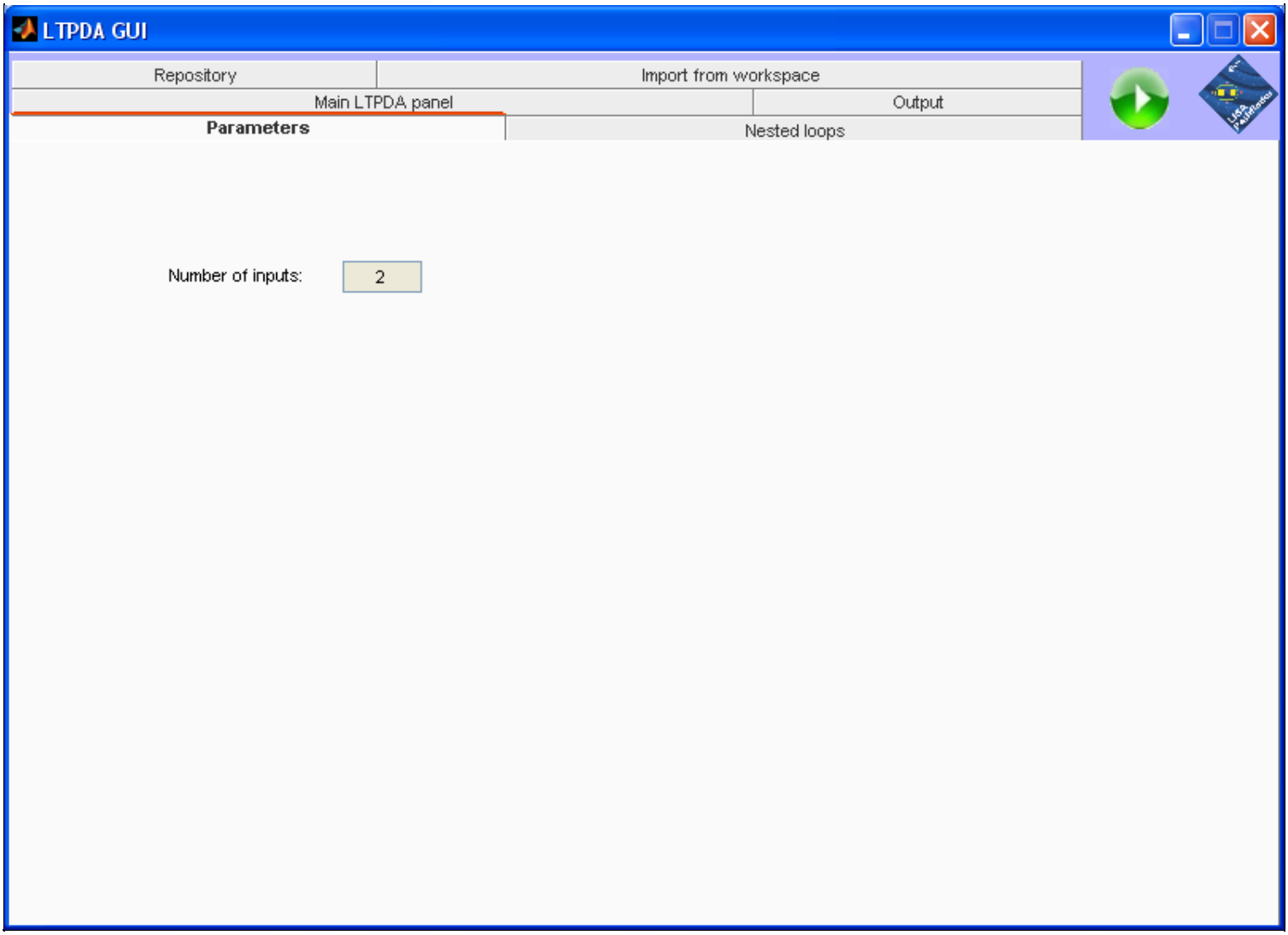
If the user selected a 'Object from list' (input) block into the active LTPDA Simulink model, the GUI will show:



The only parameter which can be set in this case is the reference index contained into the input block selected: the index will be used together with the Input array, so setting '1' will send to the connected block(s) the 1st element of the Input array.

Mux/Demux parameter

Similarly to the 'Input from list' block, the GUI will answer a selected a Mux block into a valid LTPDA Simulink model with:

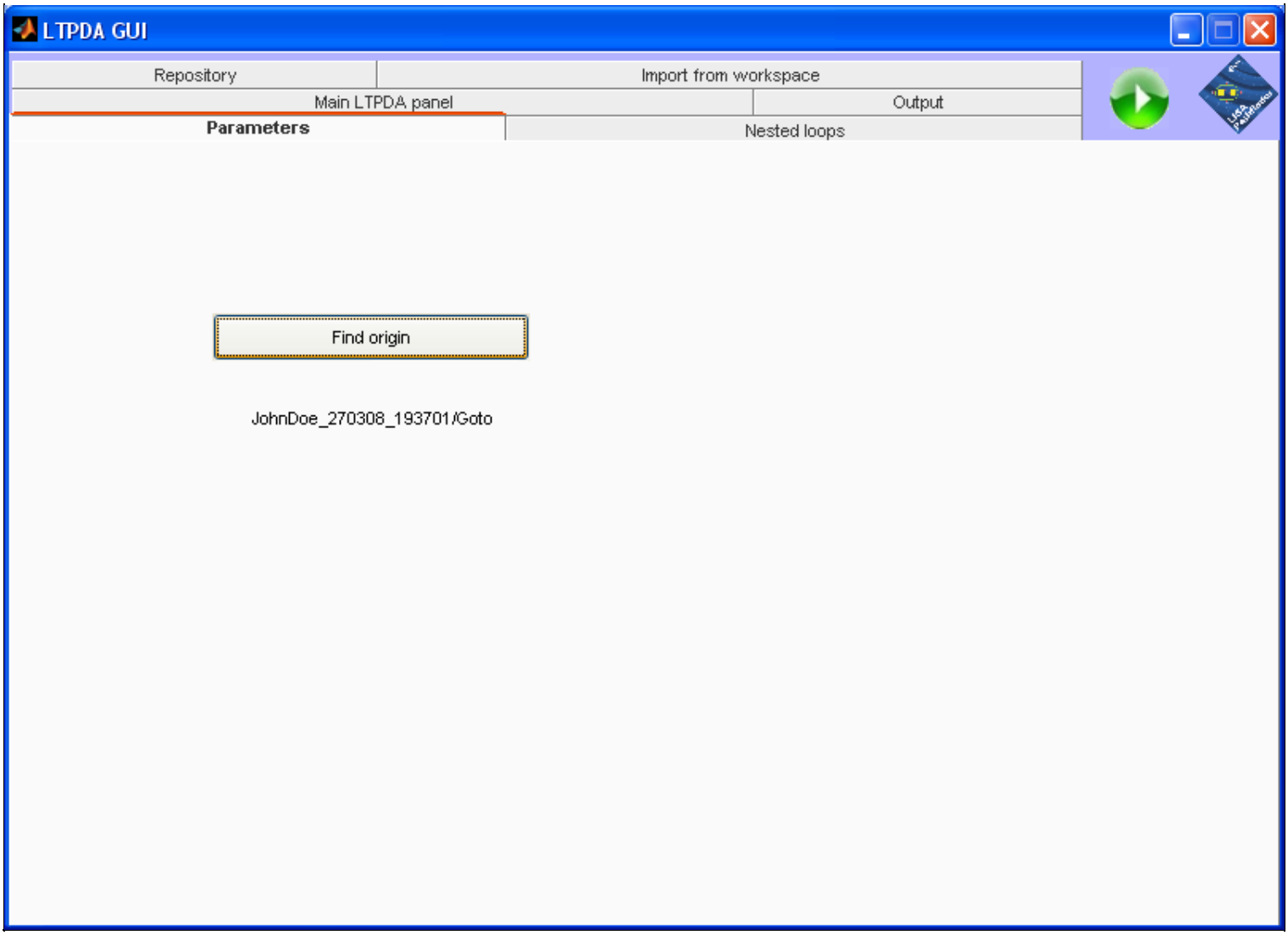


Changing this parameter will immediately alter the mux block in Simulink; unlike the direct setting of the number of inputs from Simulink, the GUI will change also the size of the block, in order to make it easier to connect the blocks and to be read.

The same applies to Demux blocks.

'From' block parameter

When the user selects a 'From' block into a valid LTPDA Simulink model, the GUI answers drawing the button '`Find Origin`', which will provide a direct pointing to the address of the 'Goto' block source of that signal.



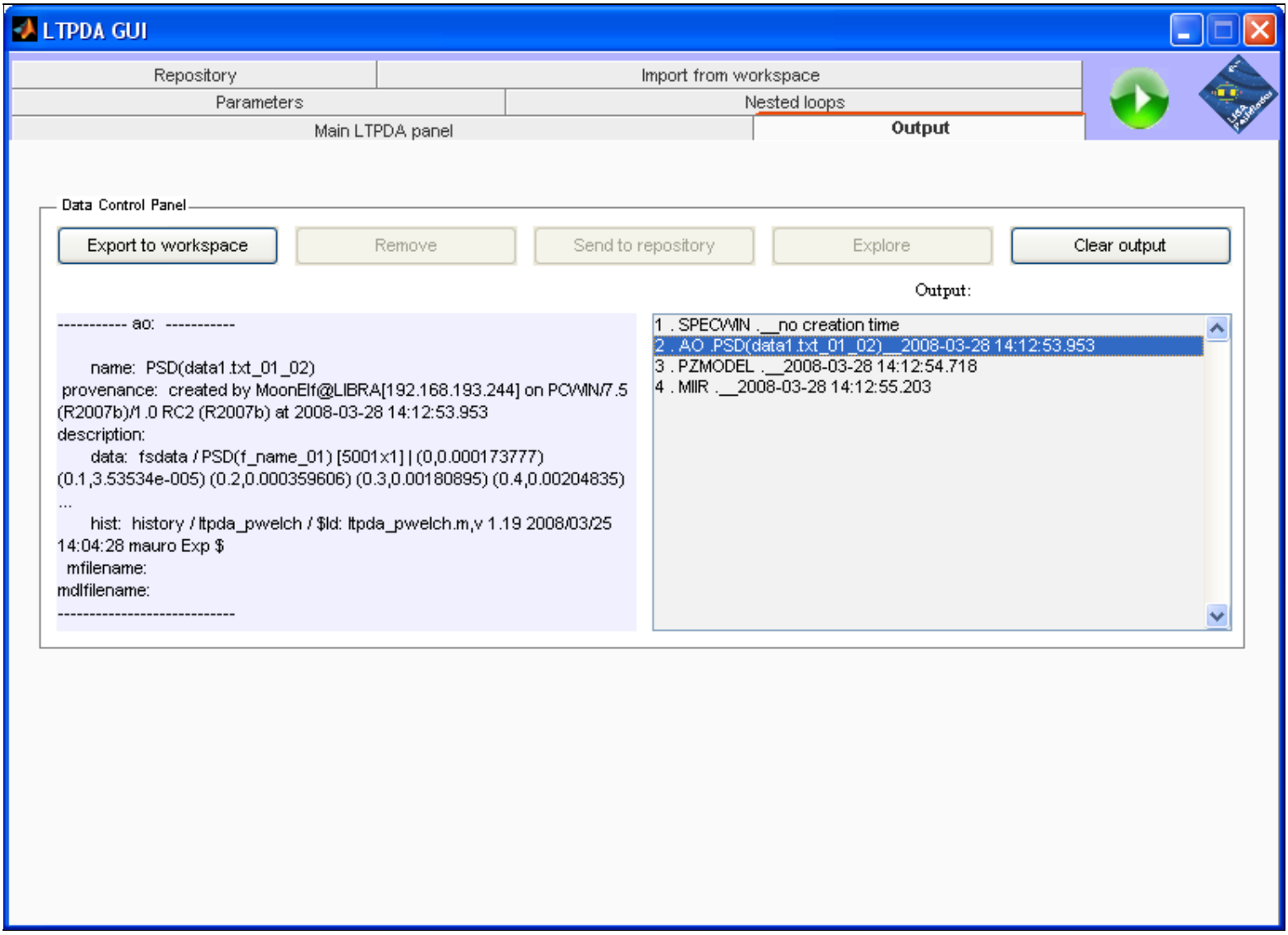
If there's more than a single 'Goto' block, or if there's none, the GUI will show a proper warning.

Main panel Output panel

Output panel

The Output panel summarizes the contents of the Output array.

To save objects into the Output array, separating them from the other input objects created during the analysis, the user must use the proper 'Send to output' block in the Simulink model. The object received from the block will be just copied into the Output array.



Similarly to the Main panel, the central part of the window consists of 2 boxes:

- On the right, the list of objects contained in **the Output array**.
- On the left, **the info panel** showing the details of the selected object in the list aside.

While selecting an object from the list will show its details in the info panel on the left, similarly to what happens in the main panel, double clicking an object here will produce (if possible) a new window with its plot - instead of adding an 'Object from list' block to the model, as double clicking an object in the Input array list of the main panel.

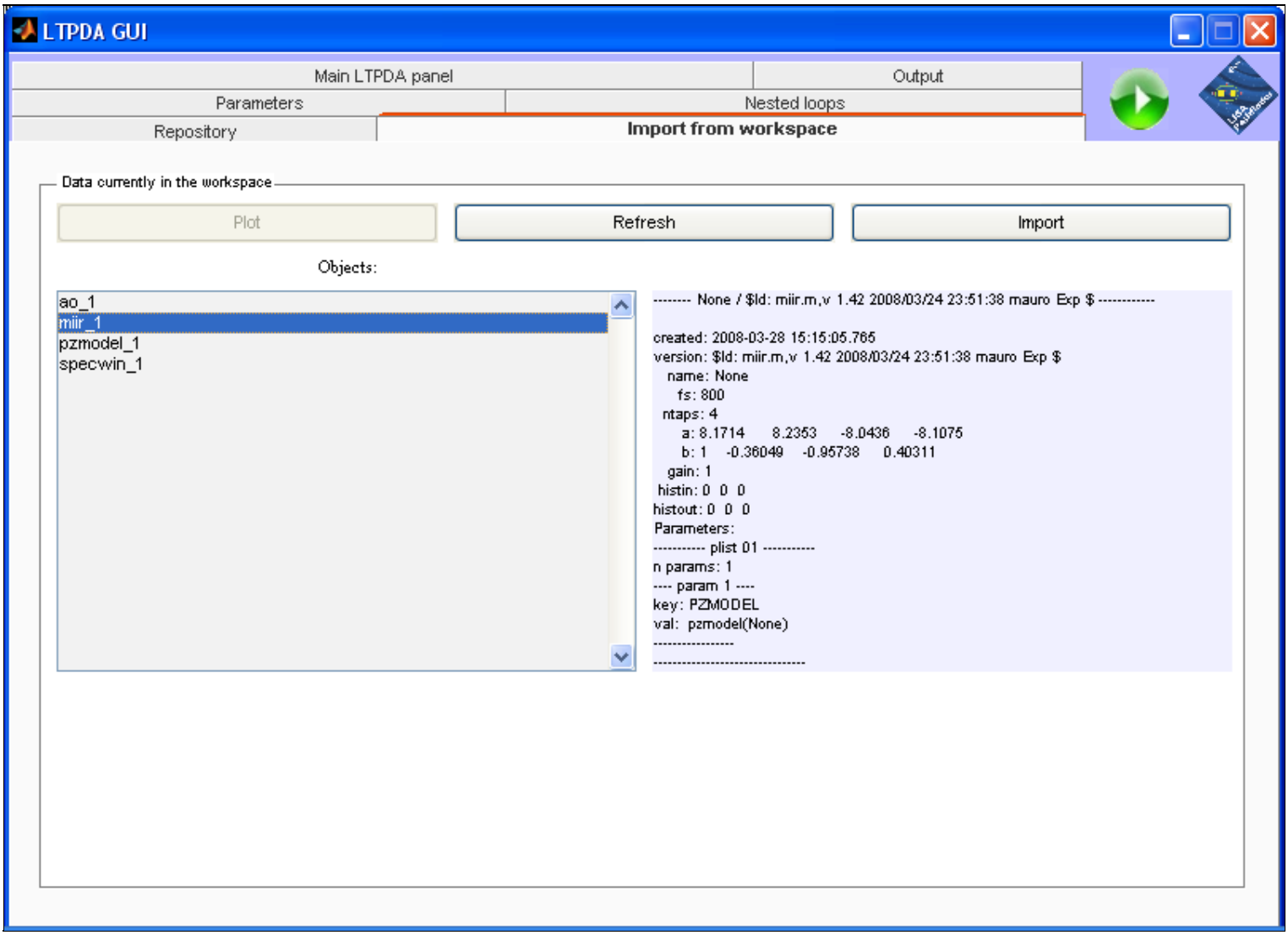
In the upper part of the panel there are 5 buttons:

- **Export to workspace**: just like the same button in the main panel, this will save the selected object(s) into the MATLAB workspace. The name of these objects will be automatically set.
- **Remove**: *not for R1*
- **Send to repository**: *not for R1*
- **Explore**: *not for R1*
- **Clear output**: to clear the entire Output array.

Parameters Import panel

Import panel

The Import panel lets the user to import into the LTPDA GUI workspace LTPDA objects currently residing in the main MATLAB workspace.



Similarly to the Main panel, the central part of the window consists of 2 boxes:

- On the left, the list of LTPDA objects currently stored in **the MATLAB workspace**. Please note that only LTPDA objects will be shown here.
- On the right, **the info panel** showing the details of the selected object in the list aside.

In the upper part of the panel there are 3 buttons:

- **Plot**: *not for R1*
- **Refresh**: to update the list of LTPDA objects in the MATLAB workspace, on the left.
- **Import**: to import the selected object(s) from the MATLAB workspace to the Input array. The object(s) will appear in the Input list of the main panel.

Output panel Repository panel

Repository panel

Not meant for R1

Import panel Nested loops

Nested loops

Not meant for R1

Repository panel The LTPDA Repository GUI

The LTPDA Repository GUI

The LTPDA toolbox contains a client interface that can be used to interact with an LTPDA repository (see Working with an LTPDA Repository). The client interface can be accessed using the LTPDA Repository GUI.

Nested loops The pole/zero model helper

The pole/zero model helper

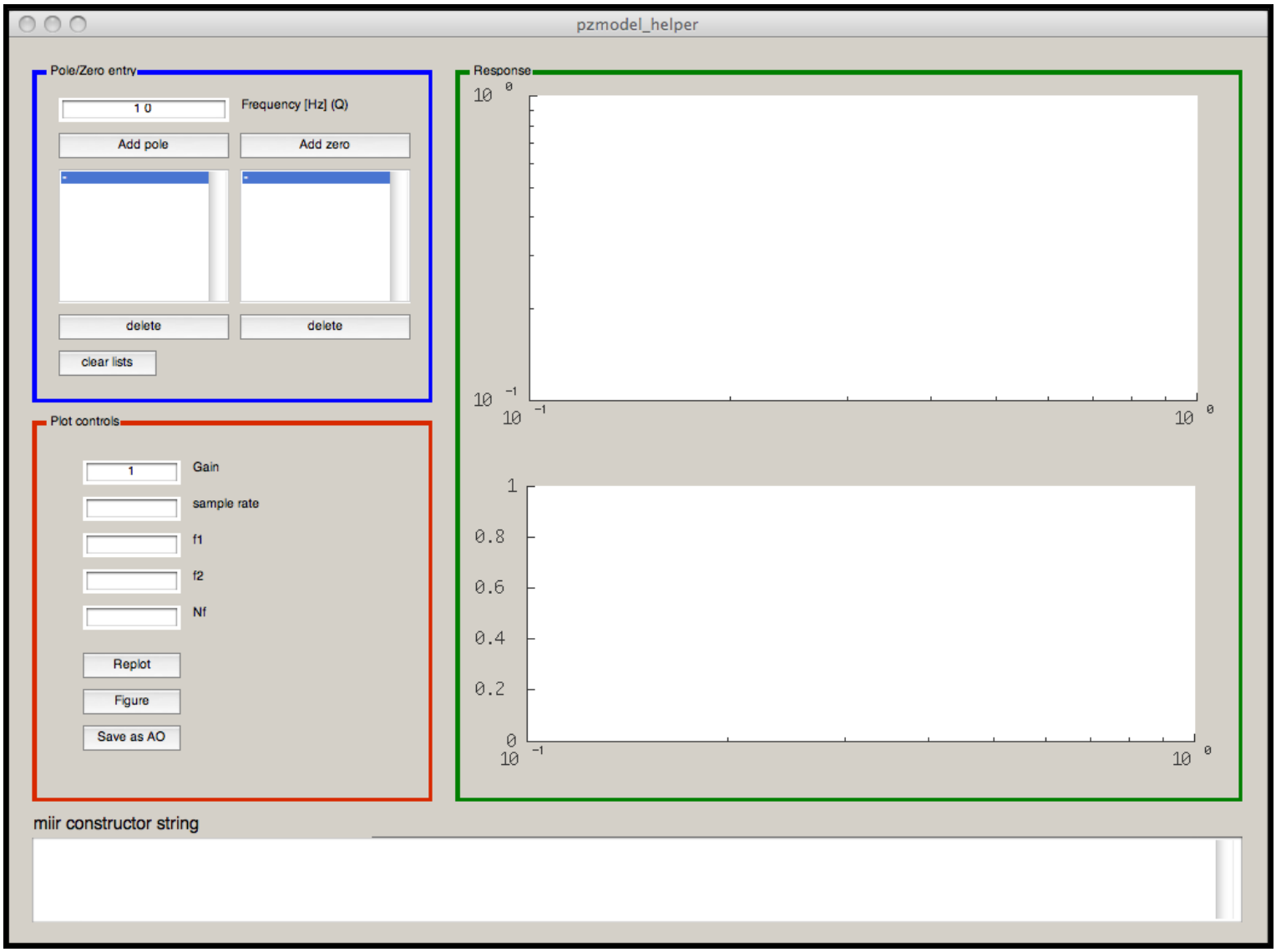
The LTPDA toolbox contains a class (`pzmodel`) for creating and using pole/zero models. The pole/zero model helper GUI allows the user to visualise the pole/zero model as it's being designed. It also allows the user to quickly see how the corresponding IIR filter (`miir` object) will look for different sample rates.

To start the pole/zero model helper:

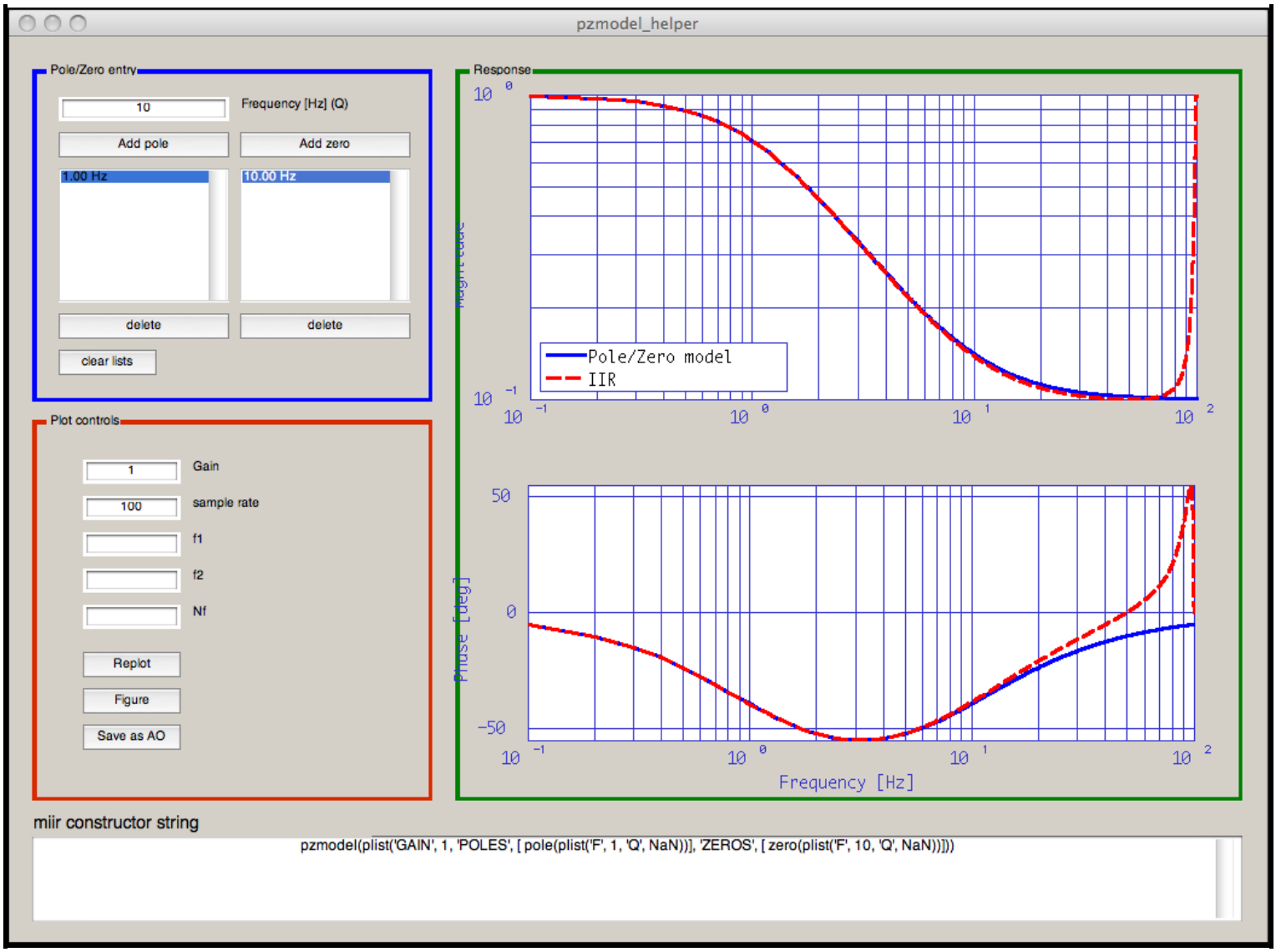
```
>> pzmodel_helper
```

or click the appropriate button on the LTPDA Launch Bay.

Once the GUI is loaded, you will see the following figure:



You can add poles and zeros to the model by entering the frequency (and Q) in the edit boxes, then click `add pole` or `add zero` as appropriate. The response is then updated in the response axes.



The LTPDA Repository GUI

The Spectral Window GUI

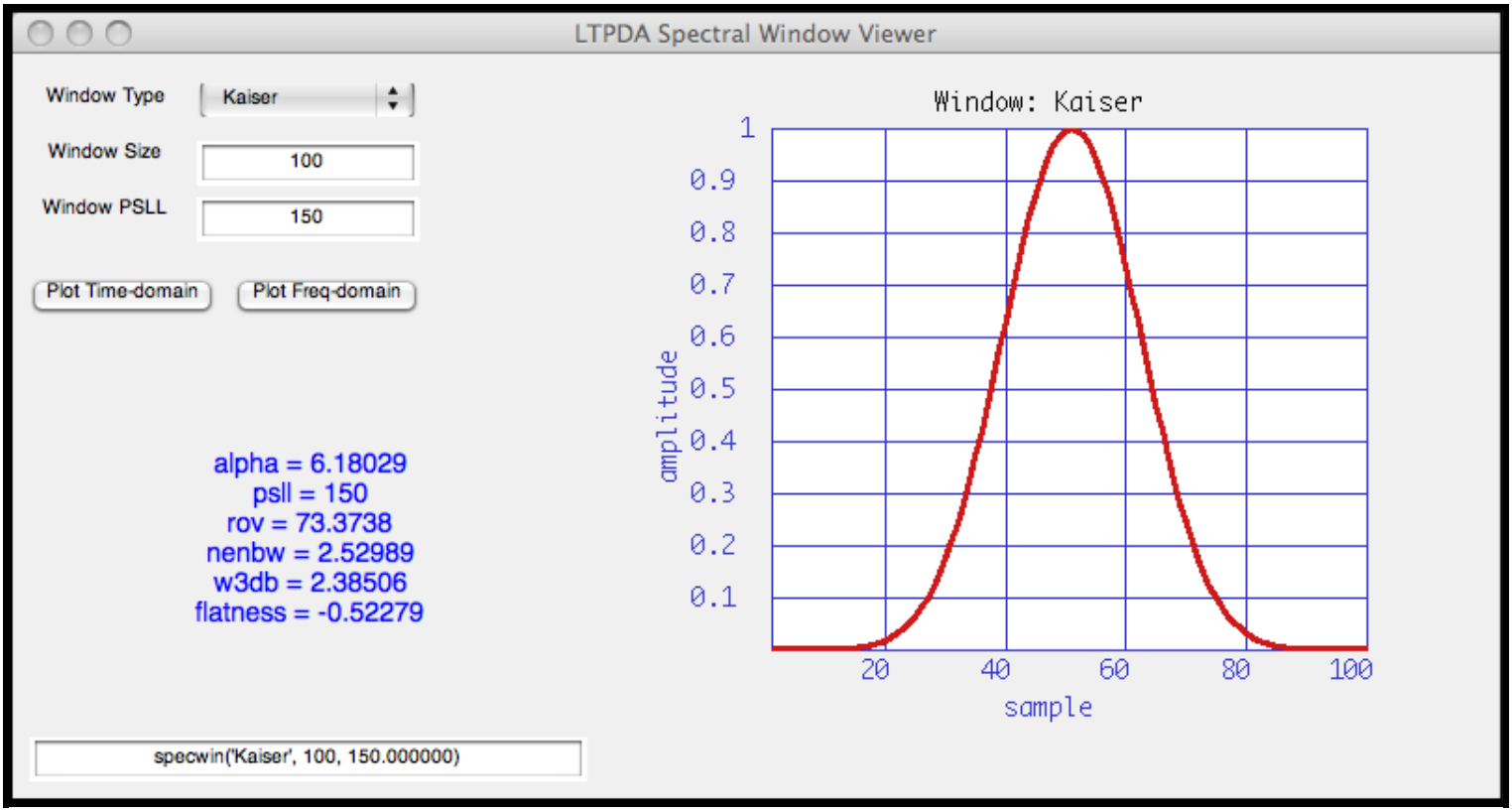
The Spectral Window GUI

The LTPDA Toolbox contains a class for creating spectral window objects (see Spectral Windows). A graphical user interface allows the user to easily explore the time-domain and frequency-domain response of any particular window.

To start the GUI:

```
>> specwin_viewer
```

or click the appropriate button on the Launch Bay.
You should then be presented with the following figure:



The pole/zero model helper

The constructor helper

The constructor helper

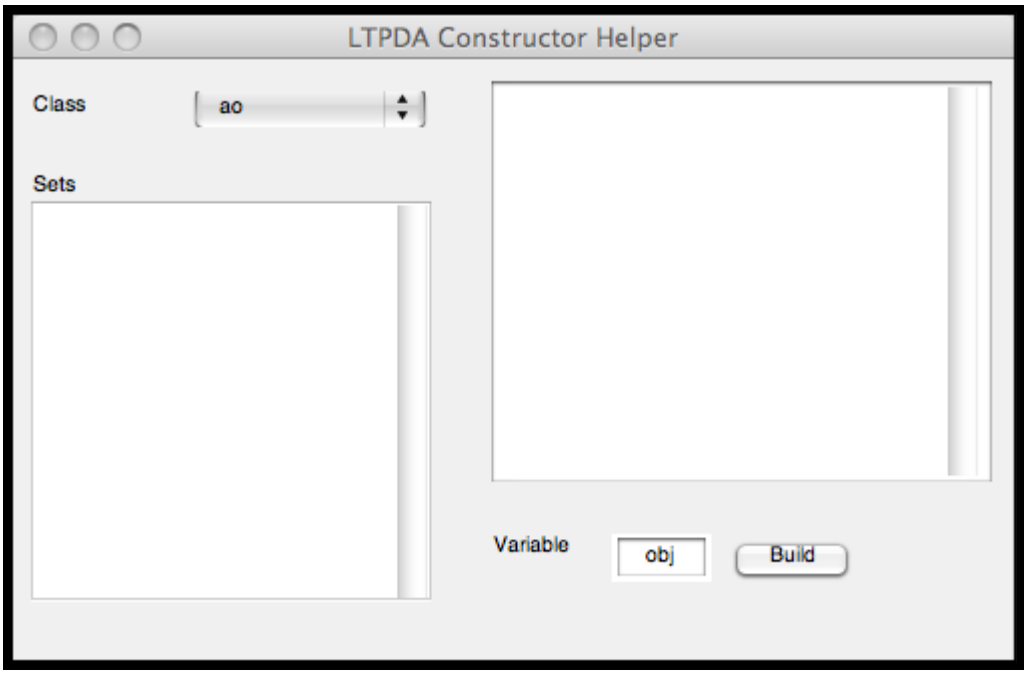
Since LTPDA is an object-oriented system, the user must create objects of different types using the appropriate constructors. The various constructor forms for each different LTPDA class can be explored using the constructor helper.

To start the constructor helper GUI:

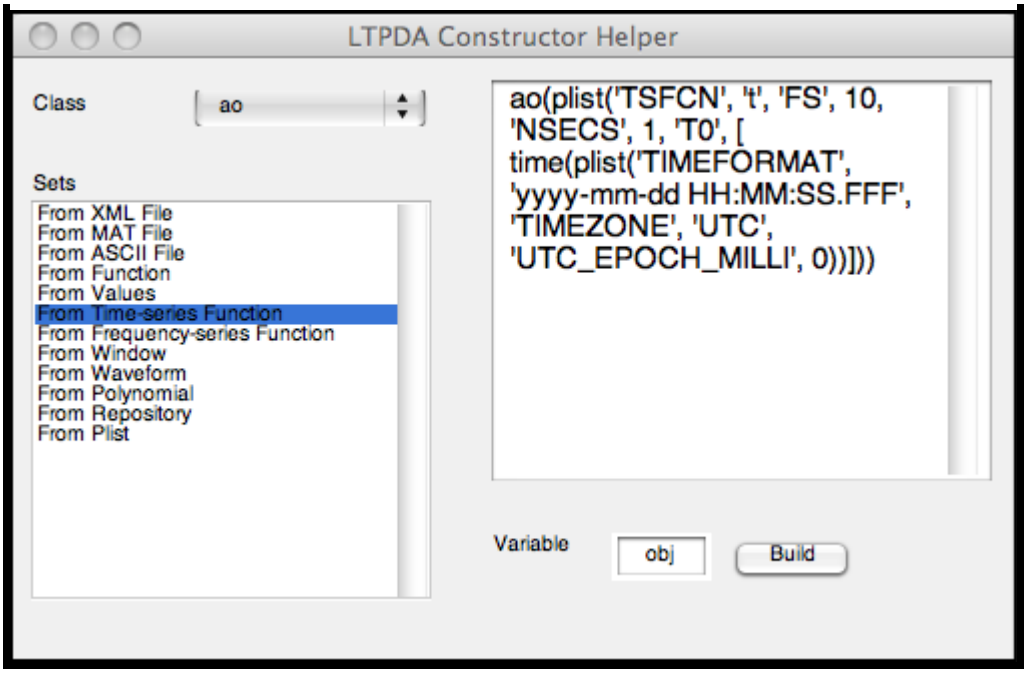
```
>> ltpda_constructor_helper
```

or click the appropriate button on the Launch Bay.

You should then be presented with the following figure:



Selecting a class from the drop-down list reveals the possible parameter sets for that class constructor. Selecting a parameter set reveals the default parameter list constructor string for constructing that class object in this way. For example, if we want to construct and Analysis Object using the time-series constructor, select the AO class then click on "From Time-series Function". You should then see:



You can then edit the parameter list and build the object by clicking on `Build`.

The Spectral Window GUI | The LTPDA object explorer

The LTPDA object explorer

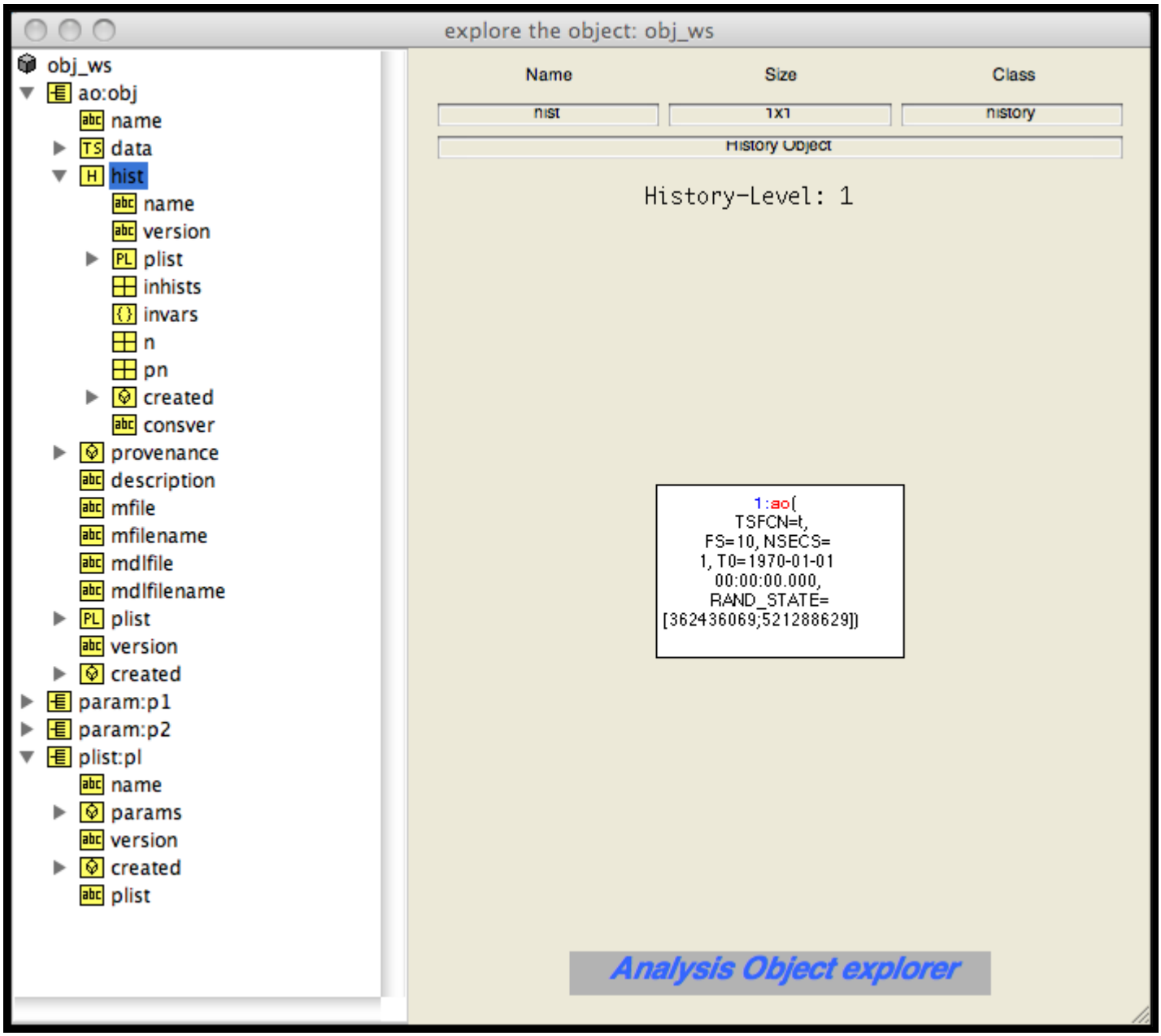
Since LTPDA works mainly with complex object types, it is often useful to explore the content of these objects graphically, particularly for Analysis Objects which may contain deep history trees. To do this, LTPDA offers the object explorer.

To start the object explorer:

```
>> explore_ao
```

or click the appropriate button on the Launch Bay.

The user is then presented with the following figure:



The object list is filled with all LTPDA User Objects currently in the MATLAB workspace.

The constructor helper The quicklook GUI

The quicklook GUI

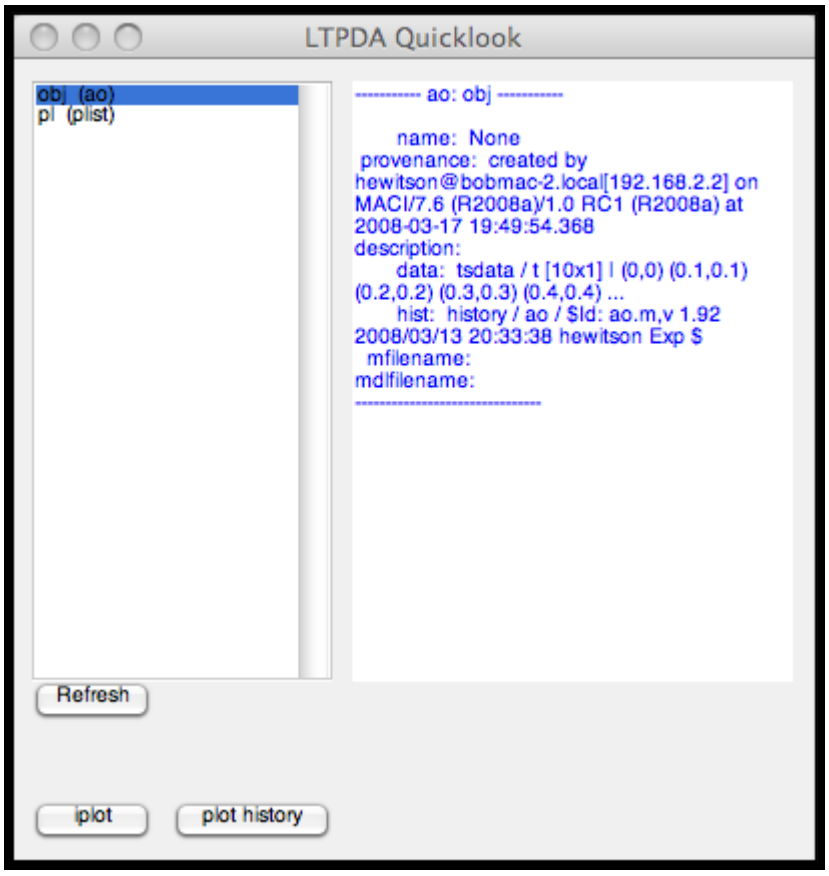
To quickly view all LTPDA objects currently in the MATLAB workspace, you can use the LTPDA Quicklook GUI.

To start the quicklook GUI:

```
>> ltpdaquicklook
```

or click the appropriate button on the Launch Bay.

The user will then be presented with the following figure:



The object list is filled with all LTPDA User Objects currently in the MATLAB workspace.

Working with an LTPDA Repository

An LTPDA repository serves as a de-centralised storage area for LTPDA objects. Objects can be submitted and retrieved from the repository allowing for easy sharing of objects and for easy recovery of previous analysis results. The following sections discuss the use of an LTPDA repository. The final section, for administrators, discusses the setting up of an LTPDA repository.

What is an LTPDA Repository

Introduction

An LTPDA repository has at its core a MySQL server. A single MySQL server can host multiple LTPDA repositories. A single repository comprises a particular set of database tables.

Since the core engine is a MySQL database, in principle any MySQL client can be used to interface with the repository. In order to submit and retrieve objects in the proper way (entering all expected meta-data), it is strongly suggested that you use the LTPDA Toolbox client commands `ltpda_obj_submit` and `ltpda_obj_retrieve` or the MATLAB LTPDA repository GUI (`repogui`).

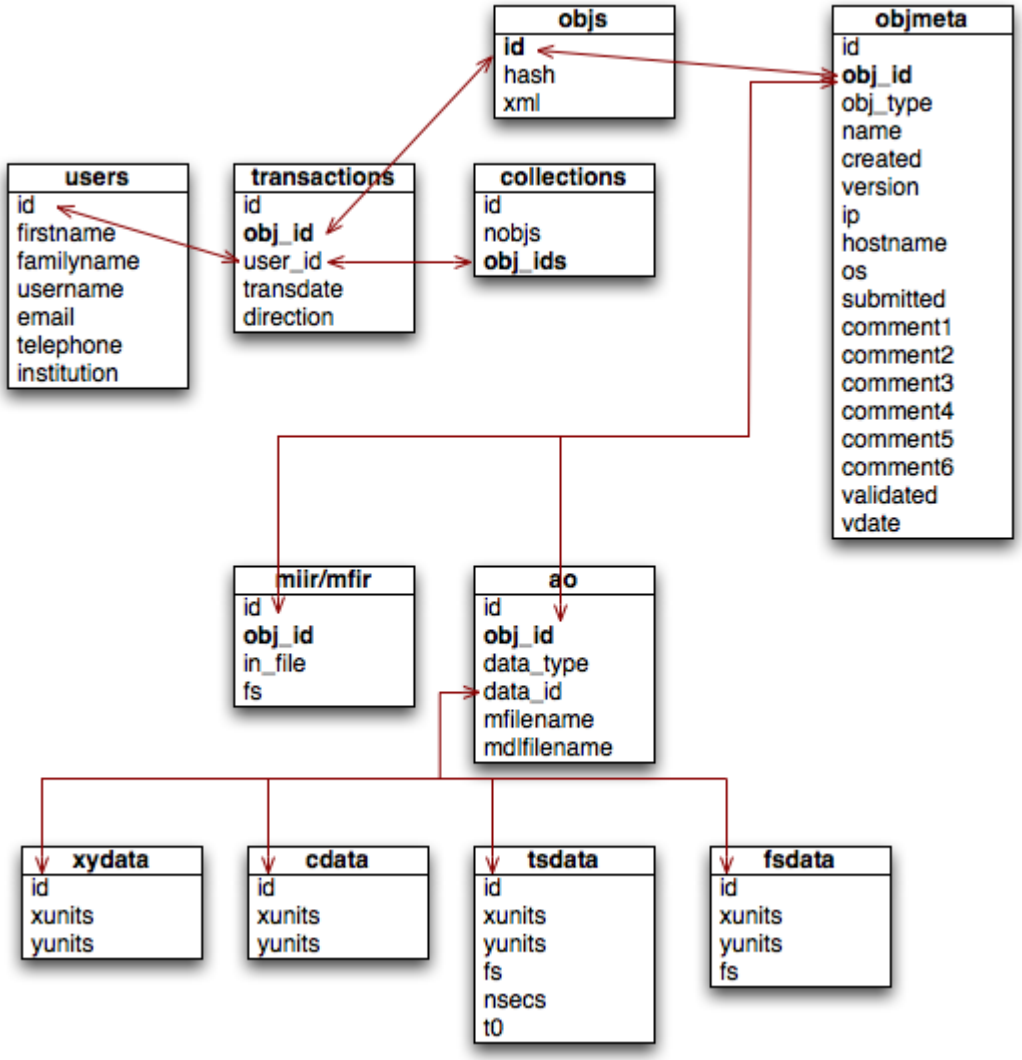
Any standard MySQL client can be used to query and search an LTPDA repository. For example, using a web-client or the standard MySQL command-line interface. In addition, the LTPDA Toolbox provides two ways to search the database: using the command `ltpda_dbquery` or using the LTPDA repository GUI. It is also possible to use the Visual Query Builder provided with the MATLAB Database Toolbox for interacting with a repository.

Database primer

A MySQL database comprises a collection of tables. Each table has a number of fields. Each field describes the type of data stored in that field (numerical, string, date, etc). When an entry is made in a table a new row is created. Interaction with MySQL databases is done using Standard Query Language (SQL) statements. For examples see MySQL Common Queries.

Database design

The database for a single repository uses the tables as shown below:



As you can see, each object that is submitted to a repository receives a unique ID number. This ID number is used to link together the various pieces of meta-data that are collected about each object. In addition, each object that is submitted is check-summed using the MD5 algorithm. That way, the integrity of each object can be checked upon retrieval.

In order to access a particular repository you need:

- The IP address of the MySQL host server
- The name of the repository (the database name)
- An account on the MySQL host server
- Permissions to access the desired database/repository

Working with an LTPDA Repository | Connecting to an LTPDA Repository

Connecting to an LTPDA Repository

Connection to an LTPDA Repository uses the JDBC interface of the Database toolbox. The command `mysql_connect` can be used to connect to a repository. It takes the following input arguments:

`hostname`	A hostname for the repository
`dbname`	A database name to connect to

You will then be prompted for a valid username and password. Here is an example call:

```
>> conn = mysql_connect('localhost', 'ltpda_test')
       ** Connecting to localhost as ltpdaadmin...
       ** Connection status:
                DatabaseProductName: 'MySQL'
             DatabaseProductVersion: '5.0.45'
                     JDBCDriverName: 'MySQL-AB JDBC Driver'
                  JDBCDriverVersion: [1x103 char]
             MaxDatabaseConnections: 0
                    CurrentUserName: 'ltpdaadmin@localhost'
                        DatabaseURL: 'jdbc:mysql://localhost/ltpda_test'
             AutoCommitTransactions: 'True'
```

The result is a `database` object which can be further used to interact with the repository.
To disconnect from the server, use the `close` method of the database class:

```
       >> close(conn)
```

Submitting LTPDA objects to a repository

Any of the following user objects can be submitted to an LTPDA repository:

- ao
- miir
- mfir
- pzmodel
- time
- timespan
- specwin
- plist

The submission process

When an object is submitted, the following steps are taken:

1. The `userid` of the user connecting is retrieved from the Users table of the repository
2. For each object to be submitted:
    1. The object to be submitted is checked to be one of the types listed above
    2. The `name`, `created`, and `version` fields are read from the object
    3. The object is converted to an XML text string
    4. An MD5 hash sum is computed for the XML string
    5. The XML string and the hash code are inserted in to the `objs` table
    6. The automatically assigned ID of the object is retrieved from the `objs` table
    7. Various pieces of meta-data (object name, object type, created time, client IP address, *etc.*) are submitted to the `objmeta` table
    8. Additional meta-data is entered into the table matching the object class (`ao`, tsdata, *etc.*)
    9. An 'in' entry is made in the `transaction` table recording the user ID and the object ID
3. A entry is then made in the `collections` table, even if this is a single object submission
4. The object IDs and the collection ID are returned to the user

Submitting objects

Objects can be submitted using the command `ltpda_obj_submit`. This command takes at least two inputs:

`object`	The LTPDA object to submit
`sinfo`	An information structure (see below)

The information structure should have the following fields:

```
'conn'                  - database connection object
'experiment_title'      - a title for the submission (Mandatory, >4 characters)
'experiment_description' - a description of this submission (Mandatory, >10 characters)
'analysis description    - a description of the analysis performed  (Mandatory, >10
```

```
characters));
'quantity'               - the physical quantity represented by the data);
'keywords'               - a comma-delimited list of keywords);
'reference_ids'          - a string containing any reference object id numbers
'additional_comments'    - any additional comments
'additional_authors'     - any additional author names
```

The following example script connects to a repository and submits an AO:

```
% Connect to a repository
conn = mysql_connect('localhost', 'ltpda_test');

% Load the AO
a = ao('result.xml');

% Build an information structure
sinfo.conn                   = conn;
sinfo.experiment_title       = 'Interferometer noise';
sinfo.experiment_description = 'Spectral estimation of interferometer output signal';
sinfo.analysis_description   = 'Spectrum of the recorded signal';
sinfo.quantity               = 'photodiode output';
sinfo.keywords               = 'interferometer, noise, spectrum';
sinfo.reference_ids          = '';
sinfo.additional_comments    = 'none';
sinfo.additional_authors     = 'no one';

% Submit the AO
[ids, cid] = ltpda_obj_submit(a, sinfo);

% Close the connection
close(conn);
```

Submitting collections

Collections of LTPDA objects can also be submitted. Here a collection is defined as a group of objects submitted at the same time. In this way, a single information structure describing the collection is assigned to all the objects. The collection is just a virtual object; it is defined by a list of object IDs in the database. The following example script connects to a repository and submits three AOs:

```
% Connect to a repository
conn = mysql_connect('localhost', 'ltpda_test');

% Create objects to submit
o1 = ao(plist('waveform', 'sine wave', 'f', 1, 'phi', 0, 'nsecs', 10, 'fs', 100));
o2 = specwin('Hanning', 100);
o3 = plist('b', 2, 'c', 'asd');

% Create an information structure
sinfo.conn                   = conn;
sinfo.experiment_title       = 'submit multiple objects';
sinfo.experiment_description = 'this is just a test of the whole thing';
sinfo.analysis_description   = 'no analysis this time';
sinfo.quantity               = '';
sinfo.keywords               = '';
sinfo.reference_ids          = '';
sinfo.additional_comments    = 'none';
sinfo.additional_authors     = 'no one';

% Submit the objects
[ids, cid] = ltpda_obj_submit(o1, o2, o3, sinfo);

% Close connection
close(conn);

% END
```

Running this script yields the following output:

```
** Connecting to 129.75.117.67 as hewitson...
** Connection status:
       DatabaseProductName: 'MySQL'
    DatabaseProductVersion: '5.0.45'
            JDBCDriverName: 'MySQL-AB JDBC Driver'
         JDBCDriverVersion: [1x103 char]
    MaxDatabaseConnections: 0
           CurrentUserName: 'bob@129.75.117.45'
               DatabaseURL: 'jdbc:mysql://129.75.117.67/ltpda_test'
    AutoCommitTransactions: 'True'

*** sinfo structure is valid.
*** Submitting objects to repository...
 ** got user id 2 for user: bob
 ** submitting object: ao / sine wave

%%%%%   Write property [x]   %%%%%
%%%%%   real data        %%%%%
Write [1000] data samples

%%%%%   Write property [y]   %%%%%
%%%%%   real data        %%%%%
Write [1000] data samples
  + Uploading XML data...done.
  + submitted object ao with id 1254
  + made meta-data entry
  + making meta data entry for ao
  + making meta data entry for tsdata
  + made meta-data entry for tsdata
  + made meta-data entry in tsdata table with id 1116
  + made meta-data entry for ao
  + made meta-data entry in ao table with id 1134
  + updated transactions table
 ** submitting object: specwin / Hanning
  + Uploading XML data...done.
  + submitted object specwin with id 1255
  + made meta-data entry
  + making meta data entry for specwin
   # no special table for objects of type specwin
  + updated transactions table
 ** submitting object: plist / None
  + Uploading XML data...done.
  + submitted object plist with id 1256
  + made meta-data entry
  + making meta data entry for plist
   # no special table for objects of type plist
  + updated transactions table
 ** made collection entry
*** submission complete.
```

Connecting to an LTPDA Repository Exploring an LTPDA Repository

Exploring an LTPDA Repository

Since an LTPDA repository is just a MySQL database, you can query the database using standard SQL commands via any of the popular MySQL clients. In addition, the LTPDA toolbox provides a simplified command that can be used to execute simple queries with only basic SQL knowledge.

The command is called `ltpda_dbquery` and it can be used to perform various queries. It takes the following input arguments:

conn	A database connection object
tablename	The name of a table to search
query	The query string written in MySQL SQL syntax

Examples of usage are:

Searching particular tables

```
>> info  = ltpda_dbquery(conn, 'select * from objmeta where id>1000 and id<2000');
>> info  = ltpda_dbquery(conn, 'ao',           'id>1000 and id<2000');
>> info  = ltpda_dbquery(conn, 'objmeta',      'name like "x12"');
>> info  = ltpda_dbquery(conn, 'users',        'username="aouser"');
>> info  = ltpda_dbquery(conn, 'collections',  'id=3');
>> info  = ltpda_dbquery(conn, 'collections',  'obj_ids="1,2"');
>> info  = ltpda_dbquery(conn, 'transactions', 'user_id=3');
>> info  = ltpda_dbquery(conn, 'transactions', 'obj_id=56');
```

Retrieving a list of tables

You can retrieve a list of the tables in a database with the call:

```
>> info  = ltpda_dbquery(conn)
```

High-level queries

Various standard queries are envisaged which ask typical questions, such as: "Give me data for a particular signal spanning a particular time-span".

Formulating this question as an SQL query requires a good knowledge of the SQL syntax used by MySQL. The query has to search across multiple tables in order to gather the IDs of the objects that fulfill the query. For these standard questions, high-level functions will be built which perform the query given some input information. This avoids the user having to formulate complicated SQL statements.

The following high-level queries currently exist in the toolbox:

ltpda_getAOsInTimeSpan	Retrieve particular AOs in the given time-span

Retrieving LTPDA objects from a repository

Objects can be retrieved from the repository either by specifying an object ID or a collection ID. The LTPDA Toolbox provides the function `ltpda_obj_retrieve` to retrieve objects.

The retrieval process

When an object is retrieved, the following steps are taken:

1. The object type for the requested ID is retrieved from the objmeta table
2. A call is made to the appropriate class constructor
3. The class constructor retrieves the XML string from the objs table
4. The XML string is then converted into an XML Xdoc object
5. The Xdoc object is then parsed to recreate the desired object

Retrieving objects

To retrieve an object, you must know its object ID. The following script shows an example of retrieving a single object:

```
% Connect to a repository
[conn, username] = mysql_connect('130.75.117.67', 'ltpda_test');

% Retrieve the object
q = ltpda_obj_retrieve(conn, 12);

% Close connection
close(conn);
```

Multiple objects can be retrieved simultaneously by giving a list of object IDs. For example

```
q = ltpda_obj_retrieve(conn, 1,2,3);
```

When multiple objects are requested, the results are returned in a cell array.

Retrieving object collections

Collections of objects can be retrieved by specifying the collection ID. The following script retrieves a collection:

```
% Connect to a repository
[conn, username] = mysql_connect('130.75.117.67', 'ltpda_test');

% Retrieve the collection
q = ltpda_obj_retrieve(conn, 'Collection', 1);

% Close connection
close(conn);
```

The output is a cell array containing the objects retrieved.

Exploring an LTPDA Repository Using the LTPDA Repository GUI

Using the LTPDA Repository GUI

The LTPDA Toolbox provides a graphical user interface for interacting with an LTPDA repository.

Starting the LTPDA Repository GUI

The GUI can be started using the command

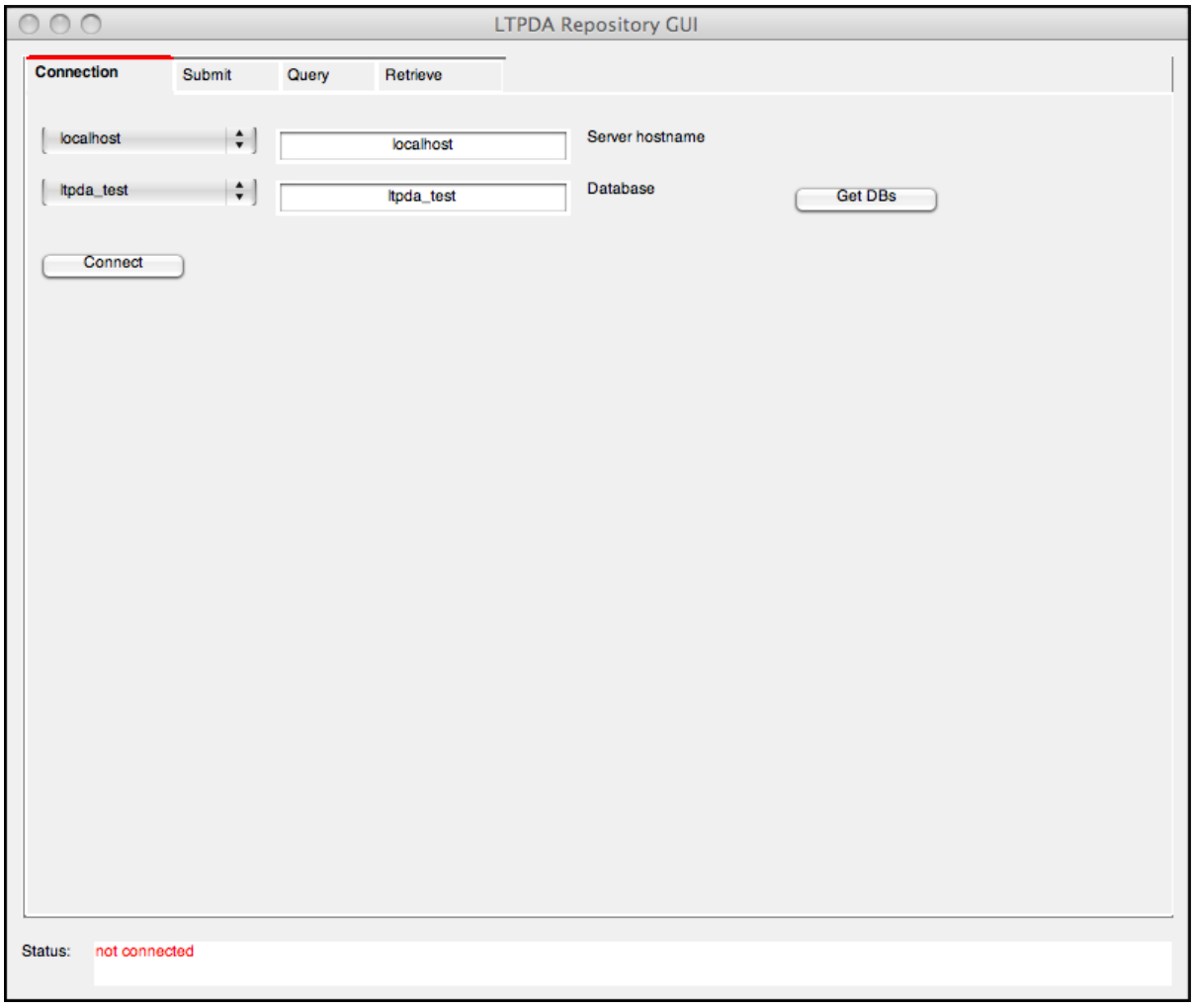
```
>> repogui
```

The interface allows submission and retrieval of objects, as well as querying of a repository.

Retrieving LTPDA objects from a repository — Connecting to a repository

Connecting to a repository

The first tab pane is the connection panel.



The user can select one of the pre-defined hosts or type a new hostname or IP address into the Server hostname field. If the database name is already known, it can be entered directly in the text field. If the database name is not known, a list of LTPDA repositories available on that particular host can be retrieved by clicking on the 'Get DBs' button. If the user has not already authenticated with the host, a login dialog will be presented prompting the user for a username and password. If authentication is successful, the drop-down menu to the left of the database name text entry field will be filled with the names of the available repositories.

The user can then select a repository and connect to it by clicking the 'Connect' button. If the user has permissions to connect to that particular repository, the repository GUI will hold a database connection object for use on the other panels of the GUI.

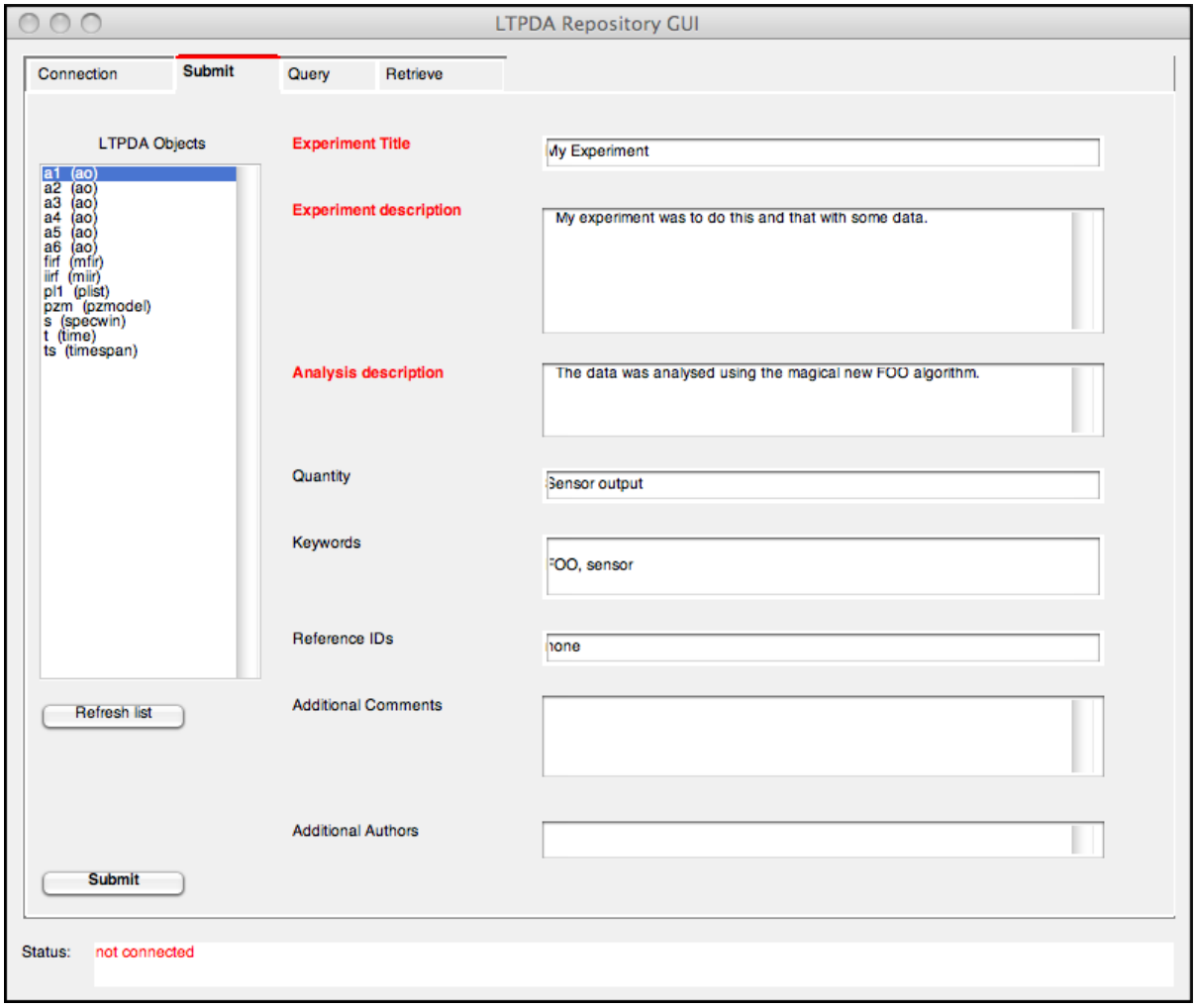
Disconnecting from a repository

To disconnect from the current repository, click the 'Disconnect' button. This must be done before being able to connect to a different repository.

Using the LTPDA Repository GUI Submitting objects to a repository

Submitting objects to a repository

Objects can be submitted to the repository using the 'Submit' panel shown below.



Selecting the objects to submit

The objects available for submission are LTPDA objects currently in the MATLAB workspace. Clicking the 'Refresh list' button will refresh the list of objects. The user `can ctrl-click` to select a subset of objects in the list for submission.

Completing the submission form

In order to ease subsequent usage of data submitted, and to allow for high level queries to be performed, the submission process must be completed with additional and sensitive informations associated with the data included in the objects. The first three fields, marked in red, are mandatory (a submssion without specifying those informations will fail). The availble fields are:

- `Experiment title` A short title for the experiment from which the data originate. Mandatory field, to be filled with more that 4 characters.
- `Experiment description` A description of the experiment from which the data originate.

Mandatory field, to be filled with more that 10 characters.

- `Analysis description` A description of the analysis performed on the data. Mandatory field, to be filled with more that 10 characters.
- `Quantity` The physical quantity that the data represent.
- `Keywords` A comma-delimited list of keywords.
- `Reference IDs` A list od object IDs that are relevant to this/these results.
- `Additional Comments` Anything else the user wants to say about the objects being submitted.
- `Additional Authors` A list of people who helped creating these object(s)

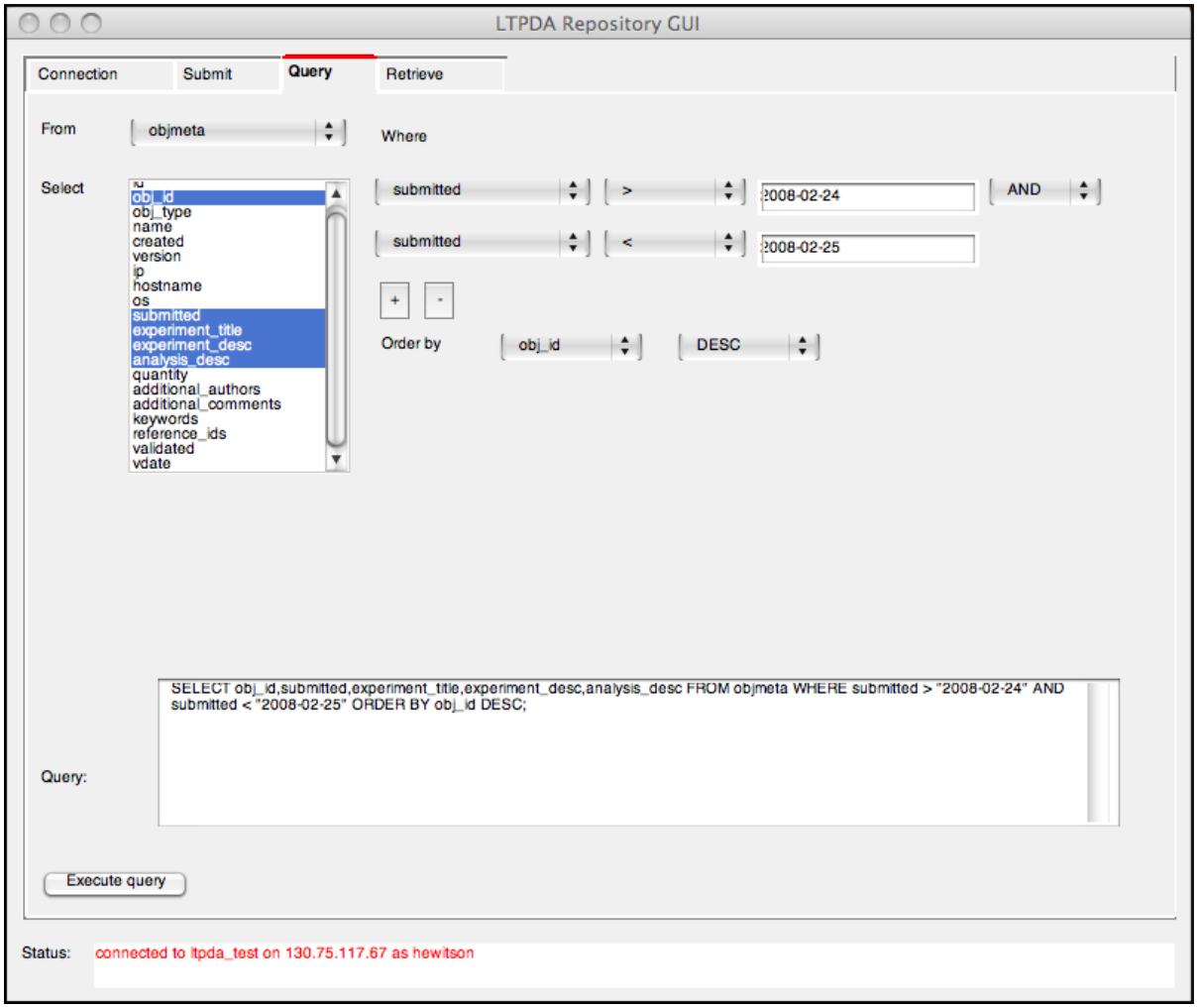
After inserting the useful information by filling the corresponding entries, the user can proceed with the submission by clicking the 'Submit' button. The Matlab window will show the response from the repository, including the IDs assigned to the submitted objects.

Connecting to a repository Querying the contents of a repository

Querying the contents of a repository

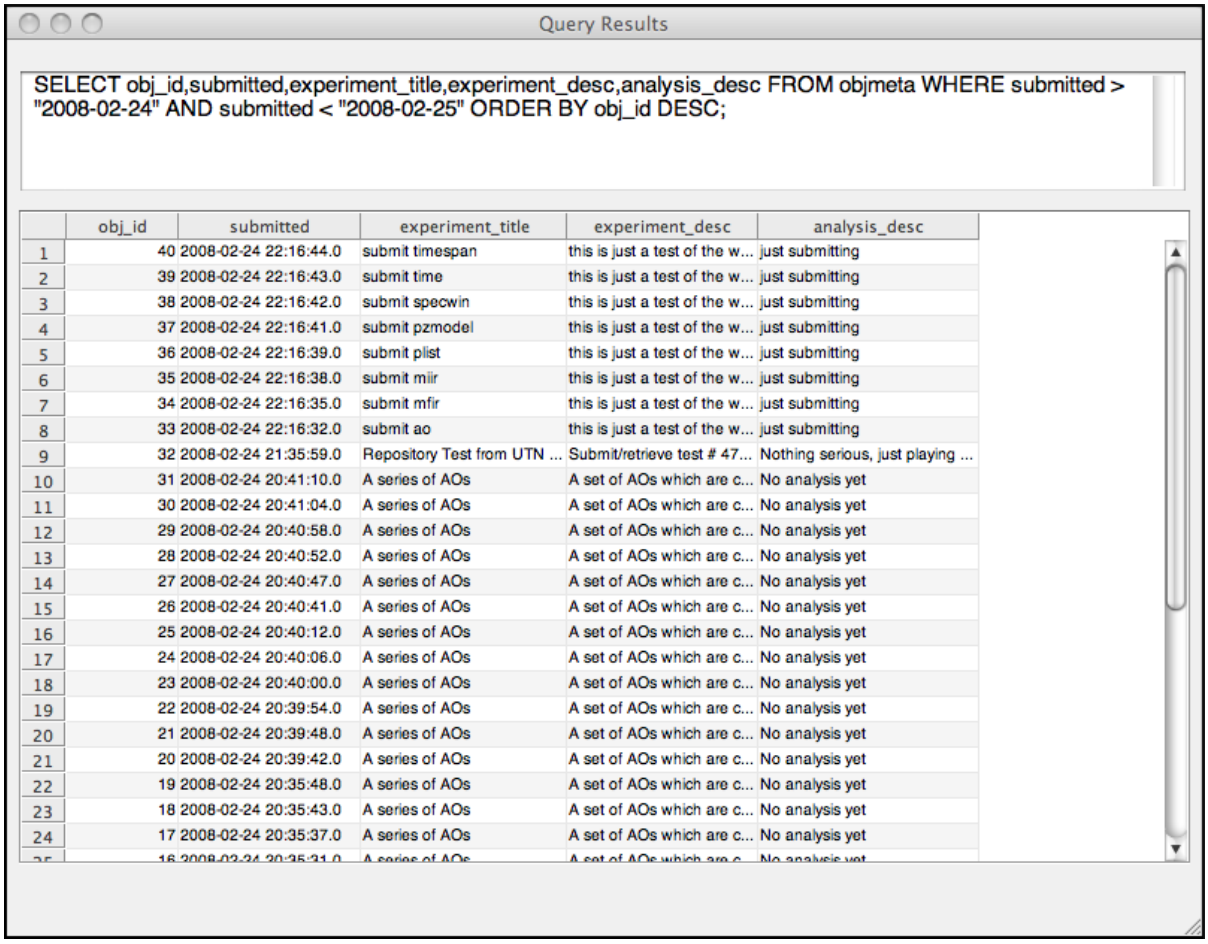
Querying an LTPDA repository is done using standard SQL statements. The repository GUI presents the user with the possibility to graphically build SQL statements which avoids learning SQL syntax. Currently, the SQL statements that can be built in this way are restricted to queries on a single database table. No high-level queries are currently implemented.

The repository GUI has a query panel which looks like the figure below:



In this figure, you see that the user has built a query to select all objects submitted on the 24th February 2008. The results will contain the object id, the submitted date, the experiment title, the experiment description, and the analysis description. The results will be sorted in descending order of the object id.

Executing the query (click the 'Execute query' button) produces the results table shown below.



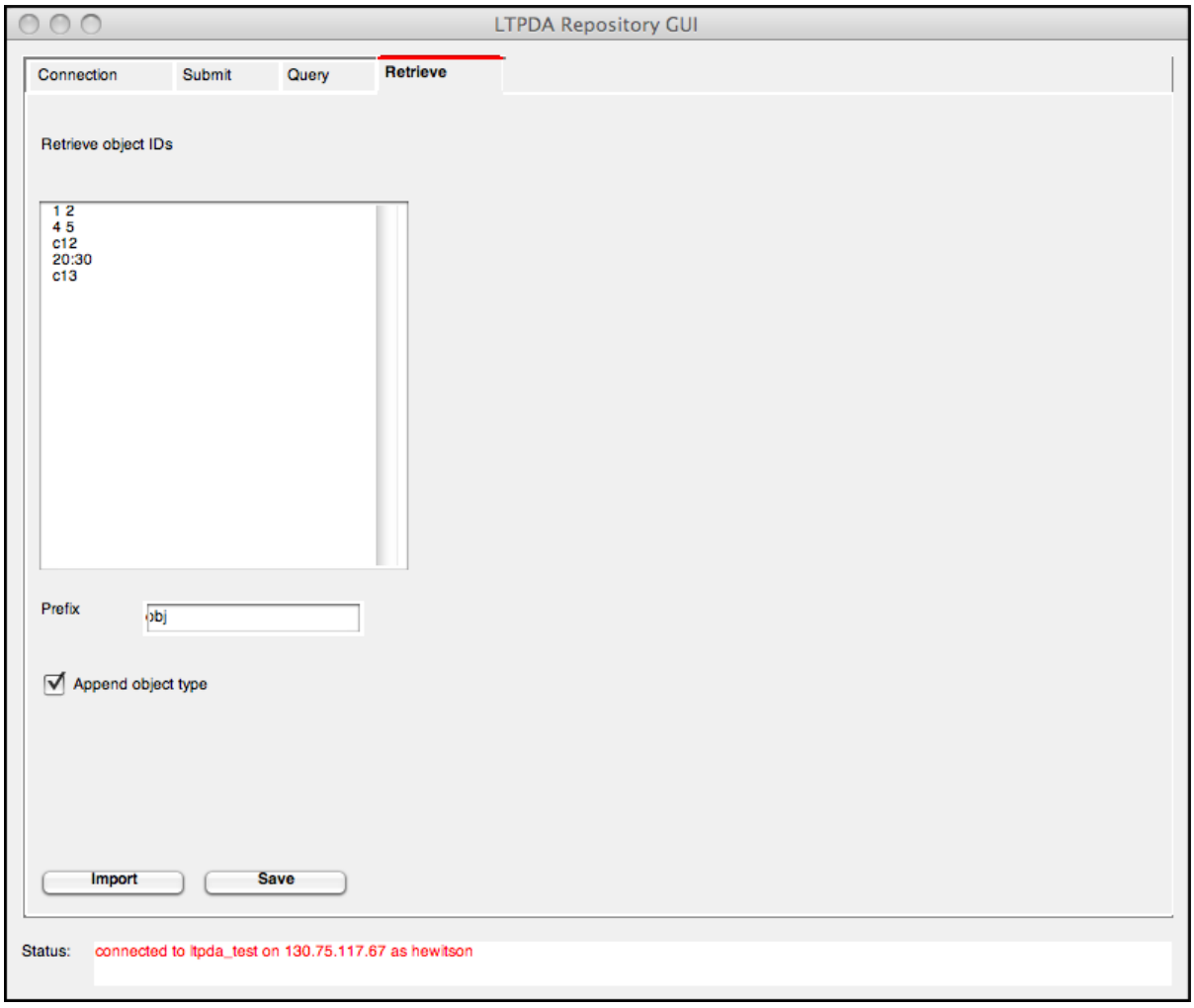
	obj_id	submitted	experiment_title	experiment_desc	analysis_desc
1	40	2008-02-24 22:16:44.0	submit timespan	this is just a test of the w...	just submitting
2	39	2008-02-24 22:16:43.0	submit time	this is just a test of the w...	just submitting
3	38	2008-02-24 22:16:42.0	submit specwin	this is just a test of the w...	just submitting
4	37	2008-02-24 22:16:41.0	submit pzmodel	this is just a test of the w...	just submitting
5	36	2008-02-24 22:16:39.0	submit plist	this is just a test of the w...	just submitting
6	35	2008-02-24 22:16:38.0	submit miir	this is just a test of the w...	just submitting
7	34	2008-02-24 22:16:35.0	submit mfir	this is just a test of the w...	just submitting
8	33	2008-02-24 22:16:32.0	submit ao	this is just a test of the w...	just submitting
9	32	2008-02-24 21:35:59.0	Repository Test from UTN ...	Submit/retrieve test # 47...	Nothing serious, just playing ...
10	31	2008-02-24 20:41:10.0	A series of AOs	A set of AOs which are c...	No analysis yet
11	30	2008-02-24 20:41:04.0	A series of AOs	A set of AOs which are c...	No analysis yet
12	29	2008-02-24 20:40:58.0	A series of AOs	A set of AOs which are c...	No analysis yet
13	28	2008-02-24 20:40:52.0	A series of AOs	A set of AOs which are c...	No analysis yet
14	27	2008-02-24 20:40:47.0	A series of AOs	A set of AOs which are c...	No analysis yet
15	26	2008-02-24 20:40:41.0	A series of AOs	A set of AOs which are c...	No analysis yet
16	25	2008-02-24 20:40:12.0	A series of AOs	A set of AOs which are c...	No analysis yet
17	24	2008-02-24 20:40:06.0	A series of AOs	A set of AOs which are c...	No analysis yet
18	23	2008-02-24 20:40:00.0	A series of AOs	A set of AOs which are c...	No analysis yet
19	22	2008-02-24 20:39:54.0	A series of AOs	A set of AOs which are c...	No analysis yet
20	21	2008-02-24 20:39:48.0	A series of AOs	A set of AOs which are c...	No analysis yet
21	20	2008-02-24 20:39:42.0	A series of AOs	A set of AOs which are c...	No analysis yet
22	19	2008-02-24 20:35:48.0	A series of AOs	A set of AOs which are c...	No analysis yet
23	18	2008-02-24 20:35:43.0	A series of AOs	A set of AOs which are c...	No analysis yet
24	17	2008-02-24 20:35:37.0	A series of AOs	A set of AOs which are c...	No analysis yet

As you can see, the query string that is actually executed is presented in the text edit box above the 'Execute query' button. This query string can be edited to allow for finer control over the query. This is for users who already have a working knowledge of MySQL SQL syntax.

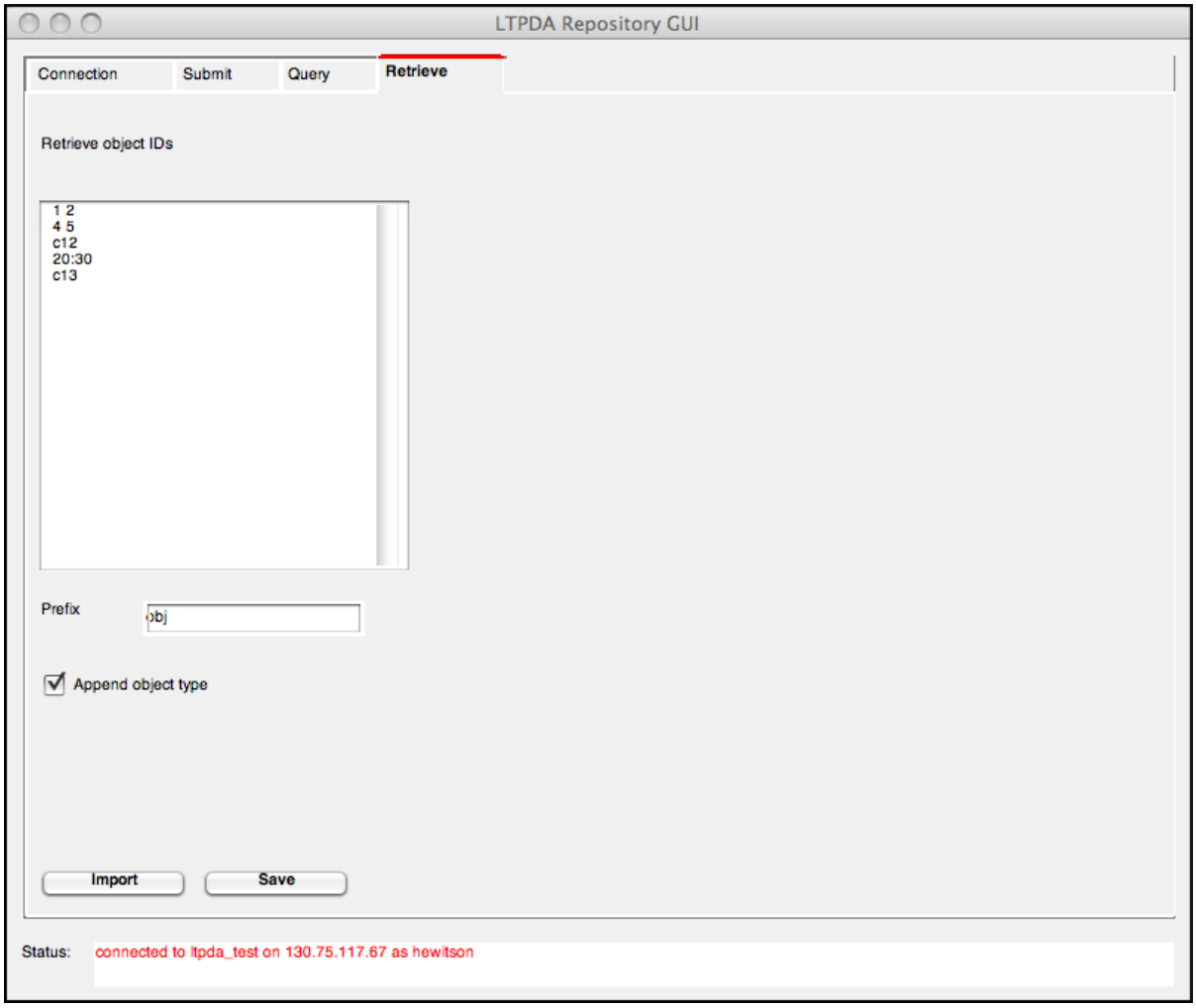
Submitting objects to a repository Retrieving objects and collections from a repository

Retrieving objects and collections from a repository

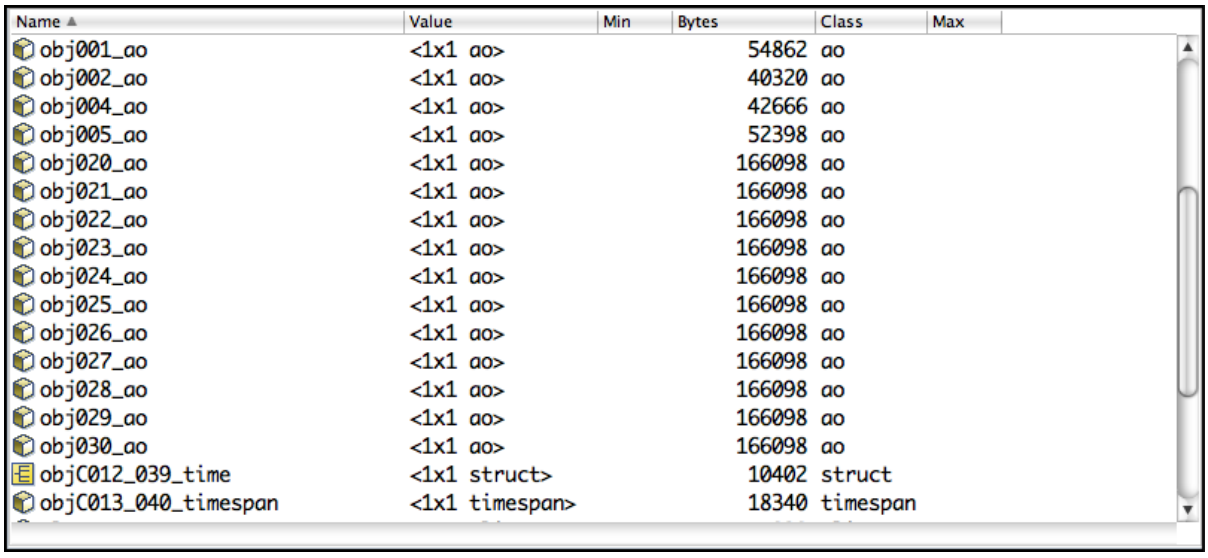
Retrieving objects from an LTPDA repository can be done using the retrieve panel shown below:



The user must enter the IDs of the objects he/she wishes to retrieve. The IDs can be entered using standard MATLAB numerical notation. Collections can be retrieved by prefixing the collection ID with a 'c', for example, 'c12' retrieves collection 12.



Clicking on the 'Import' button retrieves all objects in the list and places them in the MATLAB workspace, as shown below:



The objects can be directly saved to disk in XML format by clicking the 'Save' button.

You can select a prefix for the objects by typing in the 'prefix' edit box.

If the 'Append object type' check-box is checked, each object name (or filename) will have the object type (class) appended.

Querying the contents of a repository Class descriptions

Class descriptions

AO class	Implements analysis objects in the ltpda
CDATA class	Implements constant data objects within ltpda
FSDATA class	Implements frequency-series data objects within ltpda
HISTORY class	Implements history objects within ltpda
MFIR class	Implements finite impulse response filter objects within ltpda
MIIR class	Implements infinite impulse response filter objects within ltpda
PARAM class	Implements parameter objects within ltpda
PLIST class	Implements parameter list objects within ltpda
POLE class	Implements pole objects within ltpda
PROVENANCE class	Implements provenance objects within ltpda
PZMODEL class	Implements pole/zero model objects within ltpda
TIMESPAN class	Implements time span objects within ltpda
TIME class	Implements time objects within ltpda
TSDATA class	Implements time-series data objects within ltpda
XYDATA class	Implements x-y data objects within ltpda
XYZDATA class	Implements x-y-z data objects within ltpda
ZERO class	Implements zero objects within ltpda

ao Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the analysis object (AO)
description	A human-readable/-created description of this AO

Private Properties

Properties	Description
data	Data object associated with this AO
hist	History object associated with this AO
provenance	Provenance object associated with this AO
mfile	Full text representation of the m-file that created this AO
mfilename	The filename of the m-file that created this AO
mdlfile	Full text representation of the mdl-file that created this AO
mdlfilename	The filename of the mdl-file that created this AO
plist	The parameter list object which creates the first version of this AO

version	CVS version string of the constructor
created	Time object which stores the creation time of the AO

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Operator	Operator functions
Output	Output functions
Trigonometry	Trigometry functions

Back to Top

Constructor

Methods	Description
ao	AO analysis object class constructor.

Back to Top of Section

Helper

Methods	Description
attachm	ATTACHM attach an m file to the analysis object.
attachmdl	ATTACHMDL attach an mdl file to the analysis object.
cat	CAT concatenate AOs into a vector.
demux	DEMUX splits the input vector of AOs into a number of output AOs.
find	FIND particular samples that satisfy the input query and return a new AO.
get	GET get ao properties.
index	INDEX index into an AO array or matrix. This properly captures the history.
isfield	No description
isvalid	ISVALID tests if the given ao has all the correct fields of the correct type.

len	LEN overloads the length operator for Analysis objects. Length of the data samples.
md5	MD5 computes an MD5 checksum from an analysis objects.
mux	MUX concatenate AOs into a vector.
set	SET set an analysis object property.
setnh	SET set an analysis object property without recording the history.
string	STRING writes a command string that can be used to recreate the input Analysis object(s).
timeshift	TIMESHIFT for AO/tsdata objects, shifts the time axis such that x(1) = 0.
validate	VALIDATE checks that the input Analysis Object is reproducible and valid.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for analysis object properties.
subsref	SUBSREF Define field name indexing for analysis objects.

Back to Top of Section

Operator

Methods	Description
abs	ABS overloads the Absolute value operator for Analysis objects.
complex	COMPLEX overloads the complex operator for Analysis objects.
conj	CONJ overloads the conjugate operator for Analysis objects.
ctranspose	CTRANSPOSE overloads the ' operator for Analysis Objects.
det	DET overloads the determinant function for Analysis objects.
diag	DIAG overloads the diagonal operator for Analysis Objects.
eig	EIG overloads the determinant function for Analysis objects.

exp	EXP overloads the exp operator for Analysis objects. Exponential.
imag	IMAG overloads the imaginary operator for Analysis objects.
inv	INV overloads the inverse function for Analysis Objects.
ln	LN overloads the log operator for Analysis objects. Natural logarithm.
log	LOG overloads the log operator for Analysis objects. Natural logarithm.
log10	LOG10 overloads the log10 operator for Analysis objects. Common (base 10) logarithm.
mean	MEAN overloads the mean operator for Analysis objects. Compute the mean value.
median	MEDIAN overloads the median operator for Analysis objects. Compute the median value.
norm	NORM overloads the norm operator for Analysis Objects.
phase	PHASE overloads the ltpda_phase operator for Analysis objects.
real	REAL overloads the real operator for Analysis objects.
sqrt	SQRT overloads the sqrt operator for Analysis objects. Square root.
std	STD overloads the std operator for Analysis objects. Compute the standard deviation.
sum	SUM overloads the sum operator for Analysis objects. Compute the sum value.
svd	SVD overloads the determinant function for Analysis objects.
transpose	TRANSPOSE overloads the .' operator for Analysis Objects.
var	VAR overloads the var operator for Analysis objects. Compute the variance.

Back to Top of Section

Output

Methods	Description
ao2m	AO2M converts an analysis object to an '.m' file based on

	the history contained in the analysis object.
char	CHAR overloads char() function for analysis objects.
display	DISPLAY implement terminal display for analysis object.
export	EXPORT export an analysis object to a text file.
extractm	EXTRACTM extracts an m-file from an analysis object and saves it to
extractmdl	EXTRACTMDL extracts an mdl file from an analysis object and saves it to
iplot	IPLOT provides an intelligent plotting tool for LTPDA.
plot	PLOT a simple plot of analysis objects.
save	SAVE overloads save operator for analysis objects.
stairs	STAIRS overloads the stairs function for Analysis Objects.

Back to Top of Section

Trigonometry

Methods	Description
acos	ACOS overloads the acos operator for Analysis objects. Inverse cosine result in radians.
asin	ASIN overloads the asin operator for Analysis objects. Inverse sine result in radians.
atan	ATAN overloads the atan operator for Analysis objects. Inverse tangent result in radians.
cos	COS overloads the cos operator for Analysis objects. Cosine of argument in radians.
sin	SIN overloads the sin operator for Analysis objects. Sine of argument in radians.
tan	TAN overloads the tan operator for Analysis objects. Tangent of argument in radians.

Back to Top of Section

cdata Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the constant data object (CDATA)
xunits	Unit of the x-axes
yunits	Unit of the y-axes

Private Properties

Properties	Description
y	Constant values of the CDATA object
x	Tag of the constant values
version	CVS version string of the constructor
created	Time object which stores the creation time of the CDATA object

Methods

Constructor	Constructor of this class

Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
cdata	CDATA constant data object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get csdata properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given cdata has all the correct fields of the correct type.
set	SET set a cdata property.

Back to Top of Section

Internal

Methods	Description
get_xy_values	GET_XY_VALUES returns the values in vals.
set_xy_axis	SET_XY_AXIS set the x-axis and y-axis values.
single_operation	SINGLE_OPERATION implements specified operator overload for cdata objects.
subsref	SUBSREF Define field name indexing for cdata objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a cdata object into a string.
display	DISPLAY implement terminal display for cdata object.

save	SAVE a cdata object to file.

Back to Top of Section

ao Class fsdata Class

fsdata Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of frequency-series object (FSDATA)
xunits	Units to interpret the frequency samples
yunits	Units to interpret the data samples

Private Properties

Properties	Description
x	Frequency samples vector
y	y samples vector
fs	Sample rate of data
enbw	Equivalent noise bandwidth
navs	Number of averages
version	CVS version string of the constructor
creatd	Time object which stores the creation time of the FSDATA object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
fsdata	FSDATA frequency-series object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get a fsdata property.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given fsdata has all the correct fields of the correct type.
set	SET set a fsdata property.

Back to Top of Section

Internal

Methods	Description
get_xy_values	GET_XY_VALUES returns the x-axis and/or y-axis values.
reshapeF	RESHAPEF reshape the frequency vector to match the y vector.
setFreq	SETFREQ sets the frequency vector of a fsdata object based on the length of the y vector and the sample rate.
set_xy_axis	SET_XY_AXIS set the x-axis and y-axis values.
single_operation	SINGLE_OPERATION implements specified operator overload for fsdata objects.

subsasgn	SUBSASGN define index assignment for fsdata properties.
subsref	SUBSREF Define field name indexing for fsdata objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a param object into a string.
display	DISPLAY implement terminal display for fsdata object.
save	SAVE an fsdata object to file.

Back to Top of Section

cdata Class history Class

history Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description

Private Properties

Properties	Description
name	Mostly contains the history-name the name of the used function
plist	Parameter list object which is used for a AO opertation
inhists	History Objects which collects previous AO-operations
invars	Variable names of the current AO-operation
n	Help variable to build the history tree (see getNodes)
pn	Help variable to build the history tree (see getNodes)
consver	CVS version string of the constructor
version	CVS version string of the AO operation
created	Time object which stores the creation time of the history object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
history	HISTORY History object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get history object properties.
isfield	No description
isvalid	ISVALID tests if the given history has all the correct fields of the correct type.
set	SET set a history property.

Back to Top of Section

Internal

Methods	Description
getNodes	GETNODES converts a history object to a nodes structure suitable for plotting as a tree.
getNodes_plot	GETNODES_PLOT converts a history object to a nodes structure suitable for plotting as a tree.
subsasgn	SUBSASGN define index assignment for history properties.
subsref	SUBSREF Define field name indexing for history objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a param object into a string.
display	DISPLAY implement terminal display for history object.
hist2dot	HIST2DOT converts a history object to a 'DOT' file suitable for processing with graphviz
plot	PLOT plots a history object as a tree diagram.
save	SAVE a history object to file.
string	STRING writes a command string that can be used to recreate the input history object.

Back to Top of Section

fsdata Class mfir Class

mfir Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the finite impulse response filter (MFIR) class

Private Properties

Properties	Description
fs	Frequency of the filter
ntaps	
a	
gd	
gain	
histout	
infile	
plist	The parameter list object which creates the MFIR object
version	CVS version string of the constructor
created	Time object which stores the creation time of the MFIR object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
mfir	MFIR FIR filter object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get mfir properties.
isfield	No description
isvalid	ISVALID tests if the given mfir has all the correct fields of the correct type.
redesign	REDESIGN redesign the input filter to work for the given sample rate.
set	SET set a mfir object property.
string	STRING writes a command string that can be used to recreate the input filter object.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for mfir properties.
subsref	SUBSREF Define field name indexing for mfir objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a mfir object into a string.
display	DISPLAY overloads display functionality for mfir objects.
save	SAVE an mfir object to file.

Back to Top of Section

history Class miir Class

miir Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	name of the finite infinite impulse response filter (MIIR) class

Private Properties

Properties	Description
fs	Frequency of the filter
ntaps	
a	
b	
gain	
histin	
histout	
infile	
plist	The parameter list object which creates the MIIR object
version	CVS version string of the constructor

created	Time object which stores the creation time of the MFIR object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
miir	MIIR IIR filter object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get miir properties.
isfield	No description
isvalid	ISVALID tests if the given miir has all the correct fields of the correct type.
redesign	REDESIGN redesign the input filter to work for the given sample rate.
set	SET set a miir object property.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for miir properties.
subsref	SUBSREF Define field name indexing for miir objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a miir object into a string.
display	DISPLAY overloads display functionality for miir objects.
save	SAVE an miir object to file.
string	STRING writes a command string that can be used to recreate the input filter object.

Back to Top of Section

mfir Class param Class

param Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
`name`	Name of the parameter object (PARAM)
`key`	Key of the key/value pair
`val`	Value of the key/value pair

Private Properties

Properties	Description
`version`	CVS version string of the constructor
`created`	Time object which stores the creation time of the PARAM object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage

Output	Output functions

Back to Top

Constructor

Methods	Description
param	PARAM Parameter object class constructor.

Back to Top of Section

Helper

Methods	Description
cat	CAT concatenate params into a vector.
get	GET get parameter properties.
isfield	No description
isvalid	ISVALID tests if the given param has all the correct fields of the correct type.
mux	MUX concatenate params into a vector.
set	SET set a parameter property.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for parameter properties.
subsref	SUBSREF Define field name indexing for parameter objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a param object into a string.
display	DISPLAY display a parameter
display2	DISPLAY display a parameter
save	SAVE a param object to file.

string	STRING writes a command string that can be used to recreate the input param object.

Back to Top of Section

miir Class plist Class

plist Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the parameter list (PLIST) object
params	Vector of PARAM objects

Private Properties

Properties	Description
plist	The parameter list object which creates the PLIST object
version	CVS version string of the constructor
created	Time object which stores the creation time of the PLIST object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage

Output	Output functions

Back to Top

Constructor

Methods	Description
plist	PLIST Plist class object constructor.

Back to Top of Section

Helper

Methods	Description
append	APPEND append a parameter to the parameter list.
combine	COMBINE multiple parameter lists (plist objects) into a single plist.
find	FIND overloads find routine for a parameter list.
get	GET get parameter list properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isparam	ISPARAM look for a given key in the parameter list.
isvalid	ISVALID tests if the given plist has all the correct fields of the correct type.
nparams	NPARAMS returns the number of param objects in the list.
pset	PSET set parameter list properties.
remove	REMOVE remove a parameter from the parameter list.
set	SET set parameter list properties.
string	STRING converts a plist object to a command string which will recreate the plist object.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for parameter list properties.
subsref	SUBSREF Define field name indexing for parameter list

objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a parameter list into a string.
display	DISPLAY display plist object.
save	SAVE a plist object to file.

Back to Top of Section

param Class pole Class

pole Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of pole (POLE) object

Private Properties

Properties	Description
f	Frequency of the pole
q	Q of the pole
ri	Complex value of the pole
plist	The parameter list object which creates the POLE object
version	CVS version string of the constructor
created	Time object which stores the creation time of the POLE object

Methods

Constructor	Constructor of this class

Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
pole	POLE construct a pole object.

Back to Top of Section

Helper

Methods	Description
cat	CAT concatenate poles into a vector.
char	CHAR convert a pole object into a string.
get	GET get pole properties.
isfield	No description
isvalid	ISVALID tests if the given pole has all the correct fields of the correct type.
set	SET set a pole property.
string	STRING writes a command string that can be used to recreate the input pole object.

Back to Top of Section

Internal

Methods	Description
cp2iir	CP2IIR Return a,b IIR filter coefficients for a complex pole designed using the bilinear transform.
rp2iir	RP2IIR Return a,b coefficients for a real pole designed using the bilinear transform.
subsasgn	SUBSASGN define index assignment for pole properties.
subsref	SUBSREF Define field name indexing for pole objects.

Back to Top of Section

Output

Methods	Description
display	DISPLAY overloads display functionality for pole objects.
save	SAVE a pole object to file.

Back to Top of Section

plist Class provenance Class

provenance Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
creator	Name of the creator

Private Properties

Properties	Description
ip	IP address of the local host
hostname	Host name of the local host
os	String of the operating system
matlab_version	Version of MATLAB
sigproc_version	Version of the Signal Processing Toolbox
ltpda_version	Version of the LTPDA Toolbox
created	Time object which stores the creation time of the PROVENANCE object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
provenance	PROVENANCE constructors for provenance class.

Back to Top of Section

Helper

Methods	Description
get	GET get provenance properties.
isfield	No description
isvalid	ISVALID tests if the given provenance has all the correct fields of the correct type.
set	SET set an provenance property.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for provenance properties.
subsref	SUBSREF Define field name indexing for provenance objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a provenance object into a string.
display	DISPLAY overload terminal display for provenance objects.
save	SAVE a provenance object to file.

string	STRING writes a command string that can be used to recreate the input provenance object.

Back to Top of Section

pole Class pzmodel Class

pzmodel Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the pole zero model (PZMODEL) object
gain	Gain of the model
poles	Vector of POLE objects
zeros	Vector of ZERO objects

Private Properties

Properties	Description
plist	The parameter list object which creates the PZMODEL object
version	CVS version string of the constructor
created	Time object which stores the creation time of the PZMODEL object

Methods

Constructor	Constructor of this class

Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Operator	Operator functions
Output	Output functions

Back to Top

Constructor

Methods	Description
pzmodel	PZMODEL constructor for pzmodel class.

Back to Top of Section

Helper

Methods	Description
get	GET get a pole/zero model property.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given pzmodel has all the correct fields of the correct type.
set	SET sets a pole/zero model property.
string	STRING writes a command string that can be used to recreate the input pzmodel object.

Back to Top of Section

Internal

Methods	Description
getlowerFreq	GETLOWERFREQ gets the frequency of the lowest pole or zero in the model.
getupperFreq	GETUPPERFREQ gets the frequency of the highest pole or zero in the model.
subsasgn	SUBSASGN define index assignment for pzmodel properties.
subsref	SUBSREF Define field name indexing for pole/zero model objects.

Back to Top of Section

Operator

Methods	Description
tomfir	TOMFIR approximates a pole/zero model with an FIR filter.
tomiir	TOMIIR converts a pzmodel to an IIR filter using a bilinear transform.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a pzmodel object into a string.
display	DISPLAY overloads display functionality for pzmodel objects.
save	SAVE a pzmodel object to file.

Back to Top of Section

provenance Class specwin Class

specwin Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the spectral window (SPECWIN) object

Private Properties

Properties	Description
alpha	Alpha parameter for various window functions
psll	Peak sidelobe level
rov	Recommended overlap
nenbw	Normalised equivalent noise bandwidth
w3db	dB bandwidth in bins
flatness	Window flatness
ws	Sum of window values
ws2	Sum of squares of window values
nfft	Window length

win	Window samples (column vector)
plist	The parameter list object which creates the SPECWIN object
version	CVS version string of the constructor
created	Time object which stores the creation time of the SPECWIN object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
specwin	SPECWIN spectral window object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get specwin object properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given specwin has all the correct fields of the correct type.
set	SET sets a specwin property.
string	STRING writes a command string that can be used to recreate the input window object.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for specwin properties.

subsref	SUBSREF Define field name indexing for specwin objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a specwin object into a string.
display	DISPLAY overloads display functionality for specwin objects.
plot	PLOT plots a specwin object.
save	SAVE a specwin object to file.

Back to Top of Section

pzmodel Class time Class

time Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of time (TIME) object
utc_epoch_milli	Unix epoch time in milliseconds. The underlying timezone is UTC
timezone	Timezone of the current time
timeformat	Time format of the current time

Private Properties

Properties	Description
time_str	Time string depending on the unix epoch time and time format
plist	The parameter list object which creates TIME object
version	CVS version string of the constructor
created	Time string which stores the creation time of the TIME object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
time	TIME time object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get time properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given time has all the correct fields of the correct type.
set	SET set time properties.
string	STRING writes a command string that can be used to recreate the input time object.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for time properties.
subsref	SUBSREF Define field name indexing for time objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a time object into a string.

display	DISPLAY overloads display functionality for time objects.
format	FORMAT Returns the time in the handover format.
save	SAVE a time object to file.

Back to Top of Section

specwin Class timespan Class

timespan Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of the time span (TIMESPAN) object
start	TIME object of the start time
end	TIME object of the end time
timeformat	Time format of the start/end time objects
timezone	Timezone of the start/end time objects

Private Properties

Properties	Description
interval	Interval string of the start/end time
plist	The parameter list object which creates the TIMESPAN object
version	CVS version string of the constructor
created	Time object which stores the creation time of the TIMESPAN object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
timespan	TIMESPAN timespan object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get timespan properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given timespan has all the correct fields of the correct type.
set	SET set timespan properties.

Back to Top of Section

Internal

Methods	Description
subsasgn	SUBSASGN define index assignment for timespan properties.
subsref	SUBSREF Define field name indexing for timespan objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a timespan object into a string.
display	DISPLAY overloads display functionality for timespan objects.

save	SAVE a timespan object to file.
string	STRING writes a command string that can be used to recreate the input timespan object.

Back to Top of Section

time Class tsdata Class

tsdata Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of time-series object (TSDATA)
xunits	Units to interpret the time samples (e.g., seconds)
yunits	Units to interpret the data samples (e.g., Volts)
t0	Time object associated with the time-stamp of the first data sample

Private Properties

Properties	Description
y	Time samples vector
x	x samples vector
fs	Sample rate of data
nsecs	The length of this time-series in seconds
version	CVS version string of the constructor
created	Time object which stores the creation time of the TSDATA

object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
tsdata	TSDATA time-series object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET get tsdata properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given tsdata has all the correct fields of the correct type.
set	SET sets a tsdata property.

Back to Top of Section

Internal

Methods	Description
get_xy_values	GET_XY_VALUES returns the x-axis and/or y-axis values.
reshapeT	RESHAPET reshape the time-vector to match the x vector.
setTime	SETTIME sets the time-vector of a tsdata object based on the length of the x vector and the sample rate.
set_xy_axis	SET_XY_AXIS set the x-axis and y-axis values.
single_operation	SINGLE_OPERATION implements specified operator overload for tsdata objects.

subsasgn	SUBSASGN define index assignment for for tsdata properties.
subsref	SUBSREF Define field name indexing for tsdata objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a param object into a string.
display	DISPLAY overloads display functionality for tsdata objects.
save	SAVE a tsdata object to file.

Back to Top of Section

timespan Class xydata Class

xydata Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of x-y data object
xunits	Units to interpret the x-axes samples
yunits	Units to interpret the y-axes samples

Private Properties

Properties	Description
x	X samples vector
y	Y samples vector
version	CVS version string of the constructor
created	Time object which stores the creation time of the XYDATA object

Methods

Constructor	Constructor of this class

Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
xydata	XYDATA X-Y data object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET gets xydata properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given xydata has all the correct fields of the correct type.
set	SET sets a xydata property.

Back to Top of Section

Internal

Methods	Description
get_xy_values	GET_XY_VALUES returns the x-axis and/or y-axis values.
reshapeX	RESHAPEX reshape the X to match the Y vector.
set_xy_axis	SET_XY_AXIS set the x-axis and y-axis values.
single_operation	SINGLE_OPERATION implements specified operator overload for xydata objects.
subsasgn	SUBSASGN define index assignment for xydata properties.
subsref	SUBSREF Define field name indexing for xydata objects.

Back to Top of Section

Output

Methods	Description

char	CHAR convert a param object into a string.
display	DISPLAY overloads display functionality for xydata objects.
save	SAVE an xydata object to file.

Back to Top of Section

tsdata Class xyzdata Class

xyzdata Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of x-y-z data (XYZDATA) object
xunits	Units to interpret the x-axes samples
yunits	Units to interpret the y-axes samples
zunits	Units to interpret the z-axes samples

Private Properties

Properties	Description
x	X samples vector
y	Y samples vector
z	Z samples vector
version	CVS version string of the constructor
created	Time object which stores the creation time of the XYZDATA object

Methods

Constructor	Constructor of this class
Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
xyzdata	XZYDATA X-Y-Z data object class constructor.

Back to Top of Section

Helper

Methods	Description
get	GET gets xyzdata properties.
isfield	ISFIELD tests if the given field is one of the object properties.
isvalid	ISVALID tests if the given xyzdata has all the correct fields of the correct type.
set	SET sets an xyzdata property.

Back to Top of Section

Internal

Methods	Description
get_xyz_values	GET_XYZ_VALUES returns the x-axis and/or y-axis and/or z-axis values.
single_operation	SINGLE_OPERATION implements specified operator overload for xyzdata objects.
subsasgn	SUBSASGN define index assignment for xyzdata properties.
subsref	SUBSREF Define field name indexing for xyzdata objects.

Back to Top of Section

Output

Methods	Description
display	DISPLAY overloads display functionality for xyzdata objects.
save	SAVE an xyzdata object to file.

Back to Top of Section

xydata Class zero Class

zero Class

Properties	Properties of the class
Methods	All Methods of the class ordered by category.
Examples	Some constructor examples

Back to Class descriptions

Properties

Public Properties	Public means that all clients can access the member by its name.
Private Properties	Private restricts the access to the class itself. Only methods that are part of the same class can access private members.

Back to Top

Public Properties

Properties	Description
name	Name of zero (ZERO) object

Private Properties

Properties	Description
f	Frequency of the zero
q	Q of the zero
ri	Complex value of the zero
plist	The parameter list object which creates the ZERO object
version	CVS version string of the constructor
created	Time object which stores the creation time of the ZERO object

Methods

Constructor	Constructor of this class

Helper	Helper functions only for internal usage
Internal	Internal functions only for internal usage
Output	Output functions

Back to Top

Constructor

Methods	Description
zero	ZERO construct a pole object.

Back to Top of Section

Helper

Methods	Description
cat	CAT concatenate zeros into a vector.
get	GET get zero properties.
isfield	No description
isvalid	ISVALID tests if the given zero has all the correct fields of the correct type.
set	SET set a zero property.
string	STRING writes a command string that can be used to recreate the input zero object.

Back to Top of Section

Internal

Methods	Description
cz2iir	CZ2IIR return a,b IIR filter coefficients for a complex zero designed using the bilinear transform.
rz2iir	RZ2IIR Return a,b IIR filter coefficients for a real zero designed using the bilinear transform.
subsasgn	SUBSASGN define index assignment for zero properties.
subsref	SUBSREF Define field name indexing for zero objects.

Back to Top of Section

Output

Methods	Description
char	CHAR convert a zero object into a string.
display	DISPLAY overloads display functionality for zero objects.
save	SAVE a zero object to file.

Back to Top of Section

xyzdata Class Constructor Examples

Constructor Examples

Constructor examples

Constructor examples of the AO class

Construct empty AO

The following example creates an empty analysis object

```
a = ao()
----------- ao: a -----------

name:  None
provenance:  created by diepholz@HWS169[127.0.1.1] on GLNXA64/7.5
description:
data:  None
hist:  history / ao / $Id: ao.m,v 1.97 2008/03/24 21:34:18 hewitson Exp
mfilename:
mdlfilename:
-----------------------------
```

Construct an AO by loading the AO from a file

The following example creates a new analysis object by loading the analysis object from disk.

```
a = ao('a1.mat')
a = ao('a1.xml')
```

Construct an AO from a file

The following example creates a new analysis object by loading the data in 'file.txt'. The ascii file is assumed to be an equally sampled two-column file of time and amplitude.

```
a = ao('file.txt') or
a = ao('file.dat')
```

The following example creates a new analysis object by loading the data in 'file'. The parameter list decide how the analysis object is created. The valid key values of the parameter list are:

'type'	'tsdata','fsdata','xydata' [default: 'tsdata']
'use_fs'	If this value is set, the x-axes is computed by the fs value. [default: empty]

'columns'	[1 2 1 4] Each pair represented the x- and y-axes. (Each column pair creates an analysis object) Is the value 'use_fs' is used then represent each column the y-axes. (Each column creates an analysis object) [default: [1 2]]
'num_columns'	Maximum colums in the file. Are there more columns in the file then set this value. [default: 10]
'comment_char'	The comment character in the file [default: '%']
'description'	To set the description in the analysis object
'...'	Every property of the data object e.g. 'name'

```
% Each pair in col represented the x- and y-axes.
% 'use_fs' isnot used !!!

fname  = 'data.dat';
com    = 'my ao description';
type   = 'xydata';
xunits = 'SEC';
yunits = {'Volt', 'Hz'};  % Each AO get its own yunits
col    = [1 2 1 3];

pl = plist('filename',       fname,  ...
           'description',    com,    ...
           'type',           type,   ...
           'xunits',         xunits, ...
           'yunits',         yunits, ...
           'columns',        col,    ...
           'comment_char', '//');

out = ao('data.dat', pl);
out = ao(pl);
```

```
% 'use_fs is used --> Each column in col creates with the frequency its own AO

fname = 'data.dat';
type = 'tsdata';
fs    = 100;
t0    = time('14:00:00', ...
             'HH:MM:SS');
col   = [1 2 3];
pl = plist('filename', fname,...
           'type', type,      ...
           'use_fs', fs,      ...
           't0', {t0,         ...
                  t0+20,    ...
                  t0+40},   ...
           'columns', col);
out = ao('data.dat', pl);
out = ao(pl);
```

Construct an AO from a data object

creates an analysis object with a data object. Data object can be one of tsdata, fsdata, cdata, xydata, xyzdata.

```
fs = 100;
data1 = cdata([1 2 3; 4 5 6; 7 8 9]);
data2 = fsdata(randn(1000,1), fs);
data3 = tsdata(randn(1000,1), fs);
data4 = xydata(randn(1000,1), randn(1000,1));

a1 = ao(data1);
a2 = ao(data2);
a3 = ao(data3);
a4 = ao(data4);
```

Construct an AO from a parameter list object (PLIST)

Constructs an analysis object from the description given in the parameter list

---

Use the key word 'fcn'

The following example creates an AO from the description of any valid MATLAB function. The data object is from constant data (CDATA) class.

```
pl = plist('fcn', 'randn(100,1)');
a1 = ao(pl);
```

You can pass additional parameters to the fcn as extra parameters in the parameter list:

```
pl = plist('fcn', 'a*b', 'a', 2, 'b', 1:20);
a1 = ao(pl);
```

Use the key word 'tsfcn'

Construct an AO from a function of time, t. The data object is from time-series (TSDATA) class.

'fs'	sampling frequency [default: 10 Hz]
'nsecs'	length in seconds [default: 10 s]
't0'	Time object which is associated with the time-series [default: time(0)]

```
pl = plist('fs', 10, 'nsecs', 10,                                ...
           'tsfcn',  'sin(2*pi*1.4*t) + 0.1*randn(size(t))', ...
           't0',      time('1980-12-01 12:43:12'));

a1 = ao(pl)
```

Use the key word 'fsfcn'

Construct an AO from a function of frequency, f. The data object is from frequency-series data (FSDATA) class. You can also specify optional parameters:

'f1'	the initial frequency [default: 1e-9]
'f2'	the final frequency [default: 5]
'nf'	the number of frequency samples [default: 1000]
'scale'	'log' or 'lin' frequency spacing [default: 'log']

or provide a frequency vector:

'f'	a vector of frequencies on which to evaluate the function

```
f   = logspace(0,3, 1000);
pl1 = plist('fsfcn', '1./f.^2', 'scale', 'lin', 'nf', 100);
pl2 = plist('fsfcn', '1./f.^2', 'f', f);

a1 = ao(pl1)
a2 = ao(pl2)
```

Use the key word 'win'

Construct an AO from a spectral window object.
List of available window functions

```
pl1 = plist('win', specwin('Hannning', 100))
pl2 = plist('win', specwin('Kaiser', 10, 150));

a1 = ao(pl1)
a2 = ao(pl2)
```

Use the key word 'waveform'

Construct an AO from a waveform with the following waveform types

'sine wave'	'A', 'f', 'phi'
'noise'	'type' (can be 'Normal' or 'Uniform')

'chirp'	'f0', 'f1', 't1'
'Gaussian pulse'	'f0', 'bw'
'Square wave'	'f', 'duty'
'Sawtooth'	'f', 'width'

You can also specify additional parameters:

'fs'	sampling frequency [default: 10 Hz]
'nsecs'	length in seconds [default: 10 s]

You can also specify the initial time (t0) associated with the time-series by passing a parameter 't0' with a value that is a time object.

```
pl = plist('nsecs', 10,      ...
           'fs',     1000);

pl_w = append(pl, 'waveform',     ...
                  'sine wave',    ...
                  'phi', 30,      ...
                  'f',    1.23);
out_sin = ao(pl_w)
pl_w = append(pl, 'waveform',     ...
                  'noise',        ...
                  'type',         ...
                  'Normal');
out_noise1 = ao(pl_w)
pl_w = append(pl, 'waveform',     ...
                  'noise',        ...
                  'type',         ...
                  'Uniform');
out_noise2 = ao(pl_w)
pl_w = append(pl, 'waveform',    ...
                  'chirp',       ...
                  'f0', 1,       ...
                  'f1', 50,      ...
                  't1', 100);
out_chirp = ao(pl_w)
pl_w = append(pl, 'waveform',           ...
                  'Gaussian pulse',     ...
                  'f0', 10,             ...
                  'bw', 100);
out_gaus = ao(pl_w)
pl_w =append(pl,'waveform',       ...
                'Square wave',    ...
                'f', 1,           ...
                'duty', 50);
out_square = ao(pl_w)
pl_w = append(pl, 'waveform',     ...
                  'Sawtooth',     ...
                  'width', .5,    ...
                  'f',      1);
out_saw = ao(pl_w)
```

Use the key word 'polyval'

'polyval'	A set of polynomial coefficients. [default: [-0.0001 0.02 -1 -1]]

Additional parameters:

'nsecs'	Number of seconds [default: 10]
'fs'	Sample rate[default: 10 s]

or

't'	vector of time vertices

```
pl = plist('polyval', [1 2 3], 'Nsecs', 10, 'fs', 10);
a1 = ao(pl)
```

Construct from a set of values

The following example creates an AO from a set of values.

```
vals = [1 2 3; 4 5 6; 7 8 9];
pl = plist('vals', vals);

a1 = ao(vals)
a2 = ao(pl)
```

Constructor Examples | Constructor examples of the CDATA class

Constructor examples of the CDATA class

Construct empty CDATA object

The following example creates a empty cdata object

```
c1 = cdata()
-------- cdata 01 ------------

  name:  None
  y:  [0x0], double
  x:
------------------------------
```

Construct a CDATA object by loading the object from a file

The following example creates a new cdata object by loading the cdata object from disk.

```
c1 = cdata('c1.mat')
c1 = cdata('c1.xml')
```

Construct a CDATA object from a parameter list object (PLIST)

Creates a data object with the given parameter list which contains 'fcn' or 'vals'

Use the key word 'fcn'

The following example creates a CDATA object from the description of any valid MATLAB function.

```
pl = plist('fcn', 'randn(100,1)');
c1 = cdata(pl)
```

Use the key word 'vals'

The following example creates a CDATA object with the values specified in 'vals'. Remark: the cdata object stores the values in the property 'y' so it is also possible to use the key-value 'y'

```
vals = [0 1 2 3; 5 6 7 8];
```

```
pl1  = plist('vals', vals);
pl2  = plist('y',    vals);

c1 = cdata(pl1)
c2 = cdata(pl2)
```

Construct a CDATA from a constant data

```
c1 = cdata(1:.1:10)
c2 = cdata([1 2 3; 4 5 6; 7 8 9])
```

Construct a CDATA object with N samples all of the constant value

```
vals = 1:10;
N    = 5;

c1 = cdata(N, vals)
```

Constructor examples of the AO class Constructor examples of the FSDATA class

Constructor examples of the FSDATA class

Construct empty FSDATA object

The following example creates an empty fsdata object

```
f1 = fsdata()
----------- fsdata 01 -----------

  name:  None
    fs:  0
     x:  [0 0], double
     y:  [0 0], double
xunits:  Hz
yunits:
--------------------------------
```

Construct a FSDATA object by loading the object from a file

The following example creates a new fsdata object by loading the fsdata object from disk.

```
f1 = fsdata('f1.mat')
f1 = fsdata('f1.xml')
```

Construct a FSDATA object from a data vector

creates a frequency-series object with the given y-data. Sample rate of the data is assumed to be 1Hz.

```
y  = randn(1000,1);

f1 = fsdata(y)
```

Construct a FSDATA object from a data vector and frequency vector

creates a frequency-series object with the given (x,y)-data. The sample rate is then set as 2*x(end).

```
fs = 10;
x  = linspace(0, fs/2, 1000);
y  = randn(1000,1);

f1 = fsdata(x,y)
```

Construct a FSDATA object from a data vector and the corresponding frequency

creates a frequency-series object with the given y-data and sample rate. The frequency vector is grown assuming the first y sample corresponds to 0Hz and the last sample corresponds to the Nyquist frequency.

```
y = randn(1000,1);
fs = 100;

f1 = fsdata(y,fs)
```

Construct a FSDATA object from a data vector, frequency vector and the corresponding frequency

creates a frequency-series object with the given x,y-data and sample rate.

```
fs = 10;
x  = linspace(0, fs/2, 1000);
y  = randn(1000,1);

f1 = fsdata(x,y,fs);
```

Constructor examples of the CDATA class Constructor examples of the HISTORY class

Constructor examples of the HISTORY class

Construct empty HISTORY object

The following example creates an empty history object.

```
h1 = history()
-------- None / $Id: history.m,v 1.21 2008/03/17 09:16:36 mauro Exp

created: 2008-03-30 15:25:55.503
Inputs:
Parameters
----------
----------- plist 01 -----------
n params: 0
--------------------------------

Histories
---------
n input histories: 0

-------------------------------------------------------------------------
```

Construct a HISTORY object by loading the object from a file

```
h1 = history('h1.mat')
h1 = history('h1.xml')
```

Construct a HISTORY object with a name and version

```
h1 = history('my name', 'my ver')
```

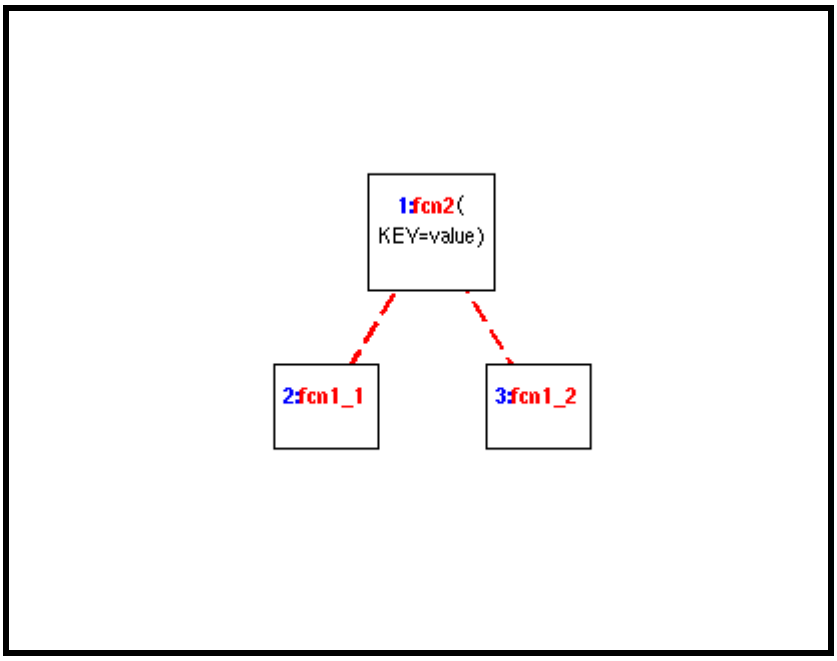
The h1 property 'name' have the content 'my name' and the h1 property 'version have the content 'my version'

Construct a HISTORY object with a name, version and parameter list

```
pl = plist('key', 'value');
h1 = history('my name', 'my ver', pl)
```

Construct a HISTORY object with a with this name, version, parameter list, and vector of input history objects

```
pl     = plist('key', 'value');
histin = [history('fcn1_1', 'Ver') history('fcn1_2', 'ver')];
h1 = history('fcn2', 'ver', pl, histin)
```



This picture illustrates the hierarchy of the history objects.

Constructor examples of the FSDATA class Constructor examples of the MFIR class

Constructor examples of the MFIR class

Construct empty MFIR object

The following example creates an empty mfir object

```
mf = mfir()
-------- None / $Id: mfir.m,v 1.36 2008/03/24 23:51:38 mauro Exp ------------

created: 2008-03-30 16:07:54.311
version: $Id: mfir.m,v 1.36 2008/03/24 23:51:38 mauro Exp
   name: None
     fs: 0
  ntaps: 0
      a:
   gain: 0
histout:
Parameters:
----------- plist 01 -----------
n params: 0
--------------------------------

--------------------------------------------------------------------------------
```

Construct a MFIR object by loading the object from a file

The following example creates a new mfir object by loading the mfir object from disk.

```
mf = mfir('mf.mat')
mf = mfir('mf.xml')
```

Construct a MFIR object from an Analysis Object

FIR filter can be generated based on the magnitude of the input Analysis Object or fsdata object. In the following example a fsdata object is first generated and then passed to the mfir constructor to obtain the equivalent FIR filter.

```
fs = 1000;
f  = linspace(0, fs/2, 1000);
y  = randn(1000,1);
a1 = ao(fsdata(f,y,fs));

mf = mfir(a1);
```

Construct a MFIR object from a parameter list (PLIST)

Use the key word 'type'

Construct an MIIR of a standard type.

'type'	one of the types: 'highpass', 'lowpass', 'bandpass', 'bandreject' [default 'lowpass']

You can also specify optional parameters:

'gain'	The gain of the filter [default: 1]
'fc'	The roll-off frequency [default: 0.1 Hz]
'fs'	The sampling frequency to design for [default: 1 Hz]
'order'	The filter order [default: 64]
'win'	Specify window function used in filter design [default: 'Hamming']

The following example creates an order 64 highpass filter with high frequency gain 2. Filter is designed for 1 Hz sampled data and has a cut-off frequency of 0.2 Hz.

```
pl = plist('type', 'highpass',   ...
                   'order', 64,  ...
                   'gain',  2.0, ...
                   'fs',    1,   ...
                   'fc',    0.2);
f = mfir(pl)
```

Furthermore it is possible to specify a spectral window.

```
win = specwin('Kaiser', 11, 150);
pl = plist('type', 'lowpass', ...
           'Win',   win,      ...
           'fs',    100,      ...
           'fc',    20,       ...
           'order', 10);

f = mfir(pl)
```

Use the key word 'pzmodel'

The following example creates a mfir object from a pole/zero model.

'pzmodel'	A pzmodel object to construct the filter from [default: empty pzmodel]
'fs'	Sample rate [default: 8 * frequency of the

highest pole or zero in the model]

```
ps = [pole(1) pole(200)];
zs = [zero(50)];
pzm = pzmodel(1, ps, zs);
pl = plist('pzmodel', pzm, 'fs', 1000);

mf = mfir(pl)
```

Use the key word 'ao'

Construct an MFIR based on the magnitude of the input AO/fsdata object a

'method'	the design method: 'frequency-sampling' - uses fir2() 'least-squares' - uses firls() 'Parks-McClellan' - uses firpm() [default: 'frequency-sampling']
'win'	Window function for frequency-sampling method [default: 'Hanning']
'N'	Filter order [default: 512]

The following example creates a mfir object from an analysis object.

```
fs = 1000;
f  = linspace(0, fs/2, 1000);
y  = randn(1000,1);
a1 = ao(fsdata(f,y,fs));

pl = plist('ao', a1);

mf = mfir(pl)
```

Construct a MFIR object from an existing model

The mfir constructor also accepts as an input existing models in different formats:

LISO files

```
f = mfir('foo_fir.fil')
```

XML files

```
f = mfir('foo_fir.xml')
```

MAT files

```
f = mfir('foo_fir.mat')
```

From repository

```

```

```
f = mfir(plist('hostname', 'localhost', 'database', 'ltpda', 'ID', []))
```

Construct a MFIR object from a difference equation

The filter can be defined in terms of two vectors specifying the coefficients of the filter and the sampling frequency. The following example creates a FIR filter with sampling frequency 1 Hz and the following recursive equation:

$$y[n] = -0.8\,x[n] + 10\,x[n-1]$$

```
b  = [-0.8 10];
fs = 1;

f = mfir(b,fs)
```

Constructor examples of the HISTORY class Constructor examples of the MIIR class

Constructor examples of the MIIR class

Construct empty MIIR object

The following example creates an empty miir object

```
f = miir()
-------- None / $Id: miir.m,v 1.42 2008/03/24 23:51:38 mauro Exp ------------

created: 2008-03-30 17:46:47.858
version: $Id: miir.m,v 1.42 2008/03/24 23:51:38 mauro Exp
   name: None
     fs: 0
  ntaps: 0
      a:
      b:
   gain: 0
 histin:
histout:
Parameters:
----------- plist 01 -----------
n params: 0
------------------------------

-------------------------------------------------------------------------------
```

Construct a MIIR object by loading the object from a file

The following example creates a new miir object by loading the miir object from disk.

```
f = miir('f.mat')
f = miir('f.xml')
```

Construct a MIIR object from a parameter list (PLIST)

Use the key word 'type'

Construct an MIIR of a standard type.

'type'	one of the types: 'highpass', 'lowpass', 'bandpass', 'bandreject' [default 'lowpass']

You can also specify optional parameters:

'gain'	The gain of the filter [default: 1]
'fc'	The roll-off frequency [default: 0.1 Hz]
'fs'	The sampling frequency to design for [default: 1 Hz]
'order'	The filter order [default: 1]
'ripple'	pass/stop-band ripple for bandpass and bandreject filters [default: 0.5]

The following example creates an order 64 highpass filter with high frequency gain 2. Filter is designed for 1 Hz sampled data and has a cut-off frequency of 0.2 Hz.

```
pl = plist('type', 'highpass',   ...
                   'order', 64,  ...
                   'gain',  2.0, ...
                   'fs',    1,   ...
                   'fc',    0.2);
f = miir(pl)
```

Furthermore it is possible to specify a spectral window.

```
win = specwin('Kaiser', 11, 150);
pl = plist('type', 'lowpass', ...
           'Win',   win,      ...
           'fs',    100,      ...
           'fc',    20,       ...
           'order', 10);

f = miir(pl)
```

Use the key word 'pzm'

The following example creates a miir object from a pole/zero model.

'pzmodel'	A pzmodel object to construct the filter from [default: empty pzmodel]
'fs'	Sample rate [default: 8 * frequency of the highest pole or zero in the model]

```
ps = [pole(1) pole(200)];
zs = [zero(50)];
pzm = pzmodel(1, ps, zs);
pl = plist('pzmodel', pzm, 'fs', 1000);

f = miir(pl)
```

Construct a MIIR object from an existing model

The miir constructor also accepts as an input existing models in different formats:

LISO files

```
f = miir('foo_iir.fil')
```

XML files

```
f = miir('foo_iir.xml')
```

MAT files

```
f = miir('foo_iir.mat')
```

From repository

```
f = miir(plist('hostname', 'localhost', 'database', 'ltpda', 'ID', []))
```

Construct a MIIR object from a difference equation

Alternatively, the filter can be defined in terms of two vectors specifying the coefficients of the filter and the sampling frequency. The following example creates a IIR filter with sampling frequency 1 Hz and the following recursive equation:

$$y[n] = 0.5\,x[n] - 0.01\,x[n-1] - 0.1\,y[n-1]$$

```
a  = [0.5 -0.01];
b  = [1 0.1]
fs = 1;

f = miir(a,b,fs)
```

Constructor examples of the MFIR class Constructor examples of the PARAM class

Constructor examples of the PARAM class

General Information of the PARAM object constructor

Parameter objects are used in the LTPDA Toolbox to configure the behaviour of algorithms. A parameter (`param`) object has two main properties:

- `'key'` — The parameter name
- `'val'` — The parameter value

See param class for further details. The 'key' property is always stored in upper case. The 'value' of a parameter can be any LTPDA object, as well as most standard MATLAB types.

Parameter values can take any form: vectors or matrices of numbers; strings; other objects, for example a `specwin` (spectral window) object.

Parameters are created using the `param` class constructor. The following code shows how to create a parameter 'a' with a value of 1

```
>> p = param('a', 1)
---- param 1 ----
key: a
val: 1
-----------------
```

The contents of a parmeter object can be accessed as follows:

```
>> key = p.key;  % get the parameter key
>> val = p.val;  % get the parameter value
```

Construct empty PARAM object

The following example creates an empty param object

```
p = param()
---- param 1 ----
key:
val: []
-----------------
```

Construct a PARAM object by loading the object from a file

The following example creates a new param object by loading the param object from disk.

```
p = param('param.mat')
p = param('param.xml')
```

Construct a PARAM object from a parameter list (PLIST) object

To construct a PARAM object with a PLIST object it is necessarc to use the key-words 'key' and 'val'

```
pl = plist('key', 'my_key', 'val', 'my_value');
p = param(pl)
```

Constructor examples of the MIIR class Constructor examples of the PLIST class

Constructor examples of the PLIST class

Parameters can be grouped together into parameter lists (`plist`).

Creating parameter lists from parameters.

The following code shows how to create a parameter list from individual parameters.

```
>> p1 = param('a', 1);  % create first parameter
>> p2 = param('b', specwin('Hanning', 100));  % create second parameter
>> pl = plist([p1 p2])  % create parameter list
----------- plist 01 -----------
n params: 2
---- param 1 ----
key: A
val: 1
-----------------
---- param 2 ----
key: B
val: specwin
-------- Hanning ------------

alpha: 0
psll: 31.5
rov: 50
nenbw: 1.5
w3db: 1.4382
flatness: -1.4236
ws: 50
ws2: 37.5
win: 100

------------------------------


-----------------
--------------------------------
```

Creating parameter lists directly.

You can also create parameter lists directly using the following constructor format:

```
>> pl = plist('a', 1, 'b', 'hello')
----------- plist 01 -----------
n params: 2
---- param 1 ----
key: A
val: 1
-----------------
---- param 2 ----
key: B
val: 'hello'
```

```
  -----------------
--------------------------------
```

Appending parameters to a parameter list.

Additional parameters can be appended to an existing parameter list using the `append` method:

```
>> pl  = append(pl, param('c', 3))  % append a third parameter
---------- plist 01 ----------
n params: 3
---- param 1 ----
key: A
val: 1
-----------------
---- param 2 ----
key: B
val: 'hello'
-----------------
---- param 3 ----
key: C
val: 3
-----------------
--------------------------------
```

Finding parameters in a parameter list.

Accessing the contents of a `plist` can be achieved in two ways:

```
>> p1  = pl.params(1);   % get the first parameter
>> val = find(pl, 'b');  % get the second parameter
```

If the parameter name ('key') is known, then you can use the `find` method to directly retrieve the value of that parameter.

Removing parameters from a parameter list.

You can also remove parameters from a parameter list:

```
>> pl = remove(pl, 2) % Remove the 2nd parameter in the list
>> pl = remove(pl, 'a') % Remove the parameter with the key 'a'
```

Setting parameters in a parameter list.

You can also set parameters contained in a parameter list:

```
>> pl = plist('a', 1, 'b', 'hello')
>> pl = pset(pl, 'a', 5, 'b', 'ola');  % Change the values of the parameter with the
keys 'a' and 'b'
```

Combining multiple parameter lists.

Parameter lists can be combined:

```
>> pl = combine(pl1, pl2)
```

If `pl1` and `pl2` contain a parameter with the same key name, the output `plist` contains a parameter with that name but with the value from the first parameter list input.

Constructor examples of the PARAM class Constructor examples of the POLE class

Constructor examples of the POLE class

Poles are specified by in the LTPDA Toolbox by a frequency, f, and (optionally) a quality factor, Q.

Construct empty POLE object

The following example creates an empty pole object

```
p = pole()
---- pole 1 ----
None: NaN Hz
----------------
```

Construct a POLE object by loading the object from a file

The following example creates a new pole object by loading the pole object from disk.

```
p = pole('pole.mat')
p = pole('pole.xml')
```

Construct a POLE object with a real pole

The following code fragment creates a real pole at 1Hz:

```
p = pole(1)
---- pole 1 ----
real pole: 1 Hz
----------------
```

It is also possible to use a parameter list (PLIST) object

```
pl = plist('f', 1);

p = pole(pl)
---- pole 1 ----
real pole: 1 Hz
----------------
```

Construct a POLE object with a complex pole

To create a complex pole, you can specify a quality factor. For example

```
p = pole(10,4)
---- pole 1 ----
complex pole: 10 Hz, Q=4 [-0.0019894+0.015791i]
----------------
```

It is also possible to use a parameter list (PLIST) object

```
pl = plist('f', 10, 'q', 4);

pole(pl)
---- pole 1 ----
complex pole: 10 Hz, Q=4 [-0.0019894+0.015791i]
----------------
```

Constructor examples of the PLIST class Constructor examples of the PROVENANCE class

Constructor examples of the PROVENANCE class

Poles are specified by in the LTPDA Toolbox by a frequency, f, and (optionally) a quality factor, Q.

Construct empty PROVENANCE object

The following example creates an empty provenance object

```
p = provenance()

-------- provenance 01 ------------

     creator:  diepholz
     created:  2008-03-30 17:35:33.943
          ip:  127.0.1.1
    hostname:  HWS169
          os:  GLNXA64
  MATLAB ver:  7.5 (R2007b)
Sig Proc ver:  6.8 (R2007b)
   LTPDA ver:  9.900000e-01 (R2007b)
-----------------------------------
```

Construct a PROVENANCE object by loading the object from a file

The following example creates a new provenance object by loading the provenance object from disk.

```
p = provenance('provenance.mat')
p = provenance('provenance.xml')
```

Construct a PROVENANCE object with a 'creator' name

If a 'creator' name is specified then creates the constructor a PROVENANCE object with this name as the creator.

```
p = provenance('other user')
-------- provenance 01 ------------

     creator:  other user
     created:  2008-03-30 17:36:52.369
          ip:  127.0.1.1
    hostname:  HWS169
          os:  GLNXA64
  MATLAB ver:  7.5 (R2007b)
Sig Proc ver:  6.8 (R2007b)
```

```
      LTPDA ver:  9.900000e-01 (R2007b)
   ----------------------------------
```

Constructor examples of the POLE class Constructor examples of the PZMODEL class

Constructor examples of the PZMODEL class

General information about pole/zero models

Construct empty PZMODEL object

The following example creates an empty pzmodel object

```
pzm = pzmodel()
---- pzmodel 1 ----
model:    None
gain :    0
pole 001: pole(NaN)
zero 001: zero(NaN)
--------------------
```

Construct a PZMODEL object by loading the object from a file

The following example creates a new pzmodel object by loading the pzmodel object from disk.

```
p = pzmodel('pzmodel.mat')
p = pzmodel('pzmodel.xml')
```

Construct a PZMODEL object from gain, poles and zeros

The following code fragment creates a pole/zero model consisting of 2 poles and 2 zeros with a gain factor of 10:

```
gain  = 10;
poles = [pole(1,2) pole(40)];
zeros = [zero(10,3) zero(100)];

pzm = pzmodel(gain, poles, zeros)
---- pzmodel 1 ----
model:    None
gain :    10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
```

```
zero 002: zero(100)
------------------
```

It is possible to give the model direct a name.

```
gain  = 10;
poles = [pole(1,2) pole(40)];
zeros = [zero(10,3) zero(100)];

pzm = pzmodel(gain, poles, zeros, 'my model name')
---- pzmodel 1 ----
model:    my model name
gain :    10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
zero 002: zero(100)
------------------
```

Construct a PZMODEL object from an existing model

The pzmodel constructor also accepts as an input existing models in a LISO file format

```
pzm = pzmodel('foo.fil')
```

Construct a PZMODEL object from a parameter list (PLIST) object

Construct a PZMODEL from its definion.

'gain'	Model gain [default: 1]
'poles'	Vector of pole objects [default: empty pole]
'zeros'	Vector of zero objects [default: empty zero]
'name'	Name of model [default: 'None']

```
poles = [pole(0.1) pole(1,100)];
zeros = [zero(10,3) zero(100)];
pl = plist('name', 'my filter', 'poles', poles, 'zeros', zeros, 'gain', 10);

pzm = pzmodel(pl)
---- pzmodel 1 ----
model:    my filter
gain :    10
pole 001: pole(0.1)
pole 002: pole(1,100)
zero 001: zero(10,3)
zero 002: zero(100)
------------------
```

Constructor examples of the PROVENANCE class

Constructor examples of the SPECWIN class

Constructor examples of the SPECWIN class

Construct empty SPECWIN object

The following example creates an empty specwin object

```
w = specwin()
-------- None ------------

   alpha: -1
    psll: -1
     rov: -1
   nenbw: -1
    w3db: -1
flatness: -1
      ws: -1
     ws2: -1
     win: 1

--------------------------
```

Construct a SPECWIN object by loading the object from a file

The following example creates a new specwin object by loading the specwin object from disk.

```
p = specwin('specwin.mat')
p = specwin('specwin.xml')
```

Construct a SPECWIN of a particular window type an length

To create a spectral window object, you call the `specwin` class constructor. The following code fragment creates a 100-point Hanning window:

```
>> w = specwin('Hanning', 100)
-------- Hanning ------------

   alpha: 0
    psll: 31.5
     rov: 50
   nenbw: 1.5
    w3db: 1.4382
```

```
      flatness: -1.4236
            ws: 50
           ws2: 37.5
           win: 100

      ------------------------------
```

List of available window functions

Construct a SPECWIN Kaiser window with the prescribed psll

In the special case of creating a Kaiser window, the additional input parameter, PSLL, must be supplied. For example, the following code creates a 100-point Kaiser window with -150dB peak side-lobe level:

```
      >> w = specwin('Kaiser', 100, 150)
      -------- Kaiser ------------

         alpha: 6.18029
          psll: 150
           rov: 73.3738
         nenbw: 2.52989
          w3db: 2.38506
      flatness: -0.52279
            ws: 28.2558
           ws2: 20.1819
           win: 100

      ---------------------------
```

Construct a SPECWIN object from a parameter list (PLIST) object

Construct a SPECWIN from its definion.

'Name'	Spectral window name [default: 'Kaiser']
'N'	Spectral window length [default: 10]
'PSLL'	Peak sidelobe length (only for Kaiser type) [default: 150]

```
pl = plist('Name', 'Kaiser', 'N', 1000, 'PSLL', 75);

w = specwin(pl)
-------- Kaiser ------------

   alpha: 3.21394
    psll: 75
     rov: 63.4095
   nenbw: 1.85055
    w3db: 1.75392
flatness: -0.963765
      ws: 389.305
     ws2: 280.892
     win: 1000
```

```
    ----------------------------
```

```
    pl = plist('Name', 'Hanning', 'N', 1000);

    w =specwin(pl)
    -------- Hanning ------------

       alpha: 0
        psll: 31.5
         rov: 50
       nenbw: 1.5
        w3db: 1.4382
    flatness: -1.4236
          ws: 500
         ws2: 375
         win: 1000

    ----------------------------
```

Constructor examples of the PZMODEL class Constructor examples of the TIME class

Constructor examples of the TIME class

Construct empty TIME object

The following example creates an empty time object

```
t = time()
---------- time 01 ----------

name            : None
utc_epoch_milli: 1206904199919
timezone        : UTC
timeformat      : yyyy-mm-dd HH:MM:SS.FFF
time_str        : 2008-03-30 19:09:59.919

created         : 2008-03-30 19:09:59.919
version         : $Id: time.m,v 1.33 2008/03/27 13:59:06 mauro Exp
plist           :
------------------------------
```

Construct a TIME object by loading the object from a file

The following example creates a new time object by loading the time object from disk.

```
t = time('time.mat')
t = time('time.xml')
```

Construct a TIME object from a string

Create a time object with different string formats.

```
t1 = time('14:00:00')              % HH:MM:SS
t1 = time('14:00:00.123')          % HH:MM:SS.FFF
t2 = time('2008.04.01 17:00:00')   % yyyy.mm.dd. HH:MM:SS
t3 = time('17:00:00 2008.04.01')   % HH:MM:SS yyyy.mm.dd
t4 = time('17:00:00 2008-04-01')   % HH:MM:SS yyyy-mm-dd
```

You can combine the following string formats:

- HH:MM:SS
- MM:SS
- dd-mm-yyyy
- dd.mm.yyyy
- yyyy-mm-dd

- yyyy.mm.dd
- mm-dd
- .fff

Construct a TIME object from a string and time format

Create a time object with different string formats and a time format. The time format is either a MATLAB format string or a MATLAB format number.

```
t1 = time('14:00:00', 'HH:MM:SS')
t2 = time('14:00:00', 'yyyy.mm.dd HH:MM:SS')
t3 = time('14:00:00', 31)
t4 = time('14:00:00', 17)
```

Construct a TIME object from the unix epoch time

Create a time objrct from the unix epoch time. The epoch time starts from '1970-01-01 00:00:00.000'
Remark: the epoch time must be in milliseconds

```
t1 = time(0)
t2 = time(1206900000000) % 2008-03-30 18:00:00
```

Construct a TIME object from a parameter list (PLIST) object

Use the key word 'utc_epoch_milli'

Construct a TIME by its properties definition

'utc_epoch_milli' The time in milliseconds
[default: 0]

Additional parameters:

'timezone' Timezone (string or java object)
[default: 'UTC']

'timeformat' Time format (string)
[default: 'yyyy-mm-dd HH:MM:SS.FFF']

```
pl1 = plist('utc_epoch_milli', 1206900000000);
pl2 = plist('utc_epoch_milli', 1206900000000, ...
            'timeformat',      'HH:MM:SS',    ...
            'timezone',        'CET');

t1 = time(pl1)
t2 = time(pl2)
```

Use the key word 'time_str'

Construct a TIME by its properties definition

'time_str' the time string [default: '1970-01-01 00:00:00.000']

Additional parameters:

'timezone' Timezone (string or java object) [default: 'UTC']

'timeformat' Time format (string) [default: 'yyyy-mm-dd HH:MM:SS.FFF']

```
pl1 = plist('time_str', '14:00:00 15.01.2008');
pl2 = plist('time_str', '14:00:00 15.01.2008', ...
            'timeformat',       'HH:MM:SS',         ...
            'timezone',         'CET');

t1 = time(pl1)
t2 = time(pl2)
```

Constructor examples of the SPECWIN class Constructor examples of the TIMESPAN class

Constructor examples of the TIMESPAN class

Construct empty TIMESPAN object

The following example creates an empty timespan object

```
ts = timespan()
---------- timespan 01 ----------

name       : None
start      : 2008-03-30 20:00:00.000
end        : 2008-03-30 20:00:00.000
timeformat: yyyy-mm-dd HH:MM:SS.FFF
interval   :
timezone   : UTC

created    : 2008-03-30 20:00:00.000
version    : $Id: timespan.m,v 1.23 2008/03/25 10:57:49 mauro Exp
plist      : plist class
---------------------------------
```

Construct a TIMESPAN object by loading the object from a file

The following example creates a new timespan object by loading the timespan object from disk.

```
t = timespan('timespan.mat')
t = timespan('timespan.xml')
```

Construct a TIMESPAN object with a start and end time

It is possible to specify the start-/end- time either with a time-object or with a time string.

```
t1_obj = time('14:00:00');
t2_obj = time('15:00:00');
t1_str = '14:00:00';
t2_str = '15:00:00';

ts1 = timespan(t1_obj, t2_obj)    % two time-objects
ts2 = timespan(t1_obj, t2_obj)    % two strings
ts3 = timespan(t1_str, t2_obj)    % combination of time-object and string
ts4 = timespan(t1_str, t2_str)    % combination of time-object and string
```

Construct a TIMESPAN object from a parameter list (PLIST) object

Construct an TIMESPAN by its properties definition

'start' The starting time [default: '1970-01-01 00:30:00.000']

'end' The ending time [default: '1980-01-01 12:00:00.010']

Additional parameters:

'timezone' Timezone (string or java object) [default: 'UTC']

'timeformat' Time format (string) [default: 'yyyy-mm-dd HH:MM:SS.FFF']

```
t1 = time('14:00:00');
t2 = time('15:00:00');

pl1 = plist('start', t1, ...
            'end',   t2);
pl2 = plist('start',        t1, ...
            'end',          t2, ...
            'timeformat', 'yyyy-mm-dd HH:MM:SS', ...
            'timezone',   'CET');

ts1 = timespan(pl1)
ts2 = timespan(pl2)
```

Constructor examples of the TIME class Constructor examples of the TSDATA class

Constructor examples of the TSDATA class

Construct empty TSDATA object

The following example creates an empty tsdata object

```
ts = tsdata()
-------- tsdata 01 ------------

  name:  None
    fs:  0
     x:  [0 0], double
     y:  [0 0], double
xunits:
yunits:
 nsecs:  0
    t0:  1970-01-01 00:00:00.000
-------------------------------
```

Construct a TSDATA object by loading the object from a file

The following example creates a new tsdata object by loading the tsdata object from disk.

```
tsd = tsdata('tsd.mat')
tsd = tsdata('tsd.xml')
```

Construct a TSDATA object from a data vector

Creates a time-series object with the given x-data. Sample rate of the data is assumed to be 1Hz.

```
y  = randn(1000,1);

tsd = tsdata(y)
```

Construct a TSDATA object from a data vector and time vector

Creates a time-series object with the given (t,x)-data. The sample rate is then set as 1/(t(2)-t(1)).

```
fs = 10;
x  = linspace(0, fs/2, 1000);
y  = randn(1000,1);

f1 = tsdata(x,y)
```

Construct a TSDATA object from a data vector and the corresponding frequency

creates a time-series object with the given x-data. The time vector t[] is grown from the sample rate. The first sample is assigned time 0.

```
y = randn(1000,1);
fs = 100;

tsd = tsdata(y,fs)
```

Constructor examples of the TIMESPAN class Constructor examples of the XYDATA class

Constructor examples of the XYDATA class

Construct empty XYDATA object

The following example creates an empty xydata object

```
xy = xydata()
-------- xydata 01 ------------

  name:  None
     x:  [0 0], double
     y:  [0 0], double
xunits:
yunits:
-------------------------------
```

Construct a XYDATA object by loading the object from a file

The following example creates a new xydata object by loading the xydata object from disk.

```
xy = xydata('xy.mat')
xy = xydata('xy.xml')
```

Construct a XYDATA object from a y-data vector

Creates an xy data object with the given y-data.

```
y  = randn(1000,1);
xy = xydata(y)
```

Construct a XYDATA object from a x-data and y-data vector

creates an xy-data object with the given (x,y)-data.

```
y  = randn(1000,1);
x  = 1:length(y);
```

```
xy = xydata(x,y)
```

Constructor examples of the TSDATA class Constructor examples of the XYZDATA class

Constructor examples of the XYZDATA class

Construct empty XYZDATA object

The following example creates an empty xyzdata object

```
xyz = xyzdata()
-------- xyzdata 01 ------------

  name:  None
     x:  [0 0], double
     y:  [0 0], double
     z:  [0 0], double
xunits:
yunits:
zunits:
--------------------------------
```

Construct a XYZDATA object by loading the object from a file

The following example creates a new xyzdata object by loading the xyzdata object from disk.

```
xyz = xyzdata('xyz.mat')
xyz = xyzdata('xyz.xml')
```

Construct a XYZDATA object from z-data

Creates an xyz object with the given z-data.

```
z = randn(1000);

xyz = xyzdata(z)
-------- xyzdata 01 ------------

  name:  None
     x:  [1000 1], double
     y:  [1000 1], double
     z:  [1000 1000], double
xunits:
yunits:
zunits:
--------------------------------
```

Construct a XYZDATA object from a x-data, y-data and z-data

creates an xy-data object with the given (x,y)-data.

```
x = randn(1000,1);
y = randn(1000,1);
z = x*y';

xy = xyzdata(x,y,z)
-------- xyzdata 01 ------------

  name:  None
     x:  [1000 1], double
     y:  [1000 1], double
     z:  [1000 1000], double
xunits:
yunits:
zunits:
--------------------------------
```

Constructor examples of the XYDATA class Constructor examples of the ZERO class

Constructor examples of the ZERO class

Poles are specified by in the LTPDA Toolbox by a frequency, f, and (optionally) a quality factor, Q.

Construct empty ZERO object

The following example creates an empty zero object

```
p = zero()
---- zero 1 ----
None: NaN Hz
----------------
```

Construct a ZERO object by loading the object from a file

The following example creates a new zero object by loading the zero object from disk.

```
p = zero('zero.mat')
p = zero('zero.xml')
```

Construct a ZERO object with a real zero

The following code fragment creates a real zero at 1Hz:

```
p = zero(1)
---- zero 1 ----
real zero: 1 Hz
----------------
```

It is also possible to use a parameter list (PLIST) object

```
pl = plist('f', 1);

p = zero(pl)
---- zero 1 ----
real zero: 1 Hz
----------------
```

Construct a ZERO object with a complex zero

To create a complex zero, you can specify a quality factor. For example

```
p = zero(10,4)
---- zero 1 ----
complex zero: 10 Hz, Q=4 [-0.0019894+0.015791i]
----------------
```

It is also possible to use a parameter list (PLIST) object

```
pl = plist('f', 10, 'q', 4);

zero(pl)
---- zero 1 ----
complex zero: 10 Hz, Q=4 [-0.0019894+0.015791i]
----------------
```

Constructor examples of the XYZDATA class | Functions - By Category

Functions - By Category

Categories

Operator

Function name	Directory	Description
abs	classes/@ao/	% ABS overloads the Absolute value method for Analysis objects.
complex	classes/@ao/	% COMPLEX overloads the complex operator for Analysis objects.
conj	classes/@ao/	% CONJ overloads the conjugate operator for Analysis objects.
ctranspose	classes/@ao/	% CTRANSPOSE overloads the ' operator for Analysis Objects.
det	classes/@ao/	% DET overloads the determinant function for Analysis objects.
diag	classes/@ao/	% DIAG overloads the diagonal operator for Analysis Objects.
eig	classes/@ao/	% EIG overloads the determinant function for Analysis objects.
exp	classes/@ao/	% EXP overloads the exp operator for Analysis objects.

		Exponential.
imag	classes/@ao/	% IMAG overloads the imaginary operator for Analysis objects.
inv	classes/@ao/	% INV overloads the inverse function for Analysis Objects.
ln	classes/@ao/	% LN overloads the log operator for Analysis objects. Natural logarithm.
log	classes/@ao/	% LOG overloads the log operator for Analysis objects. Natural logarithm.
log10	classes/@ao/	% LOG10 overloads the log10 operator for Analysis objects. Common (base 10) logarithm.
max	classes/@ao/	% MAX computes the maximum value of the data in the AO.
mean	classes/@ao/	% MEAN computes the mean value of the data in the AO.
median	classes/@ao/	% MEDIAN computes the median value of the data in the AO.
min	classes/@ao/	% MIN computes the minimum value of the data in the AO.
mode	classes/@ao/	% MODE computes the modal value of the data in the AO.
norm	classes/@ao/	% NORM overloads the norm operator for Analysis Objects.
phase	classes/@ao/	% PHASE overloads the ltpda_phase operator for Analysis objects.
real	classes/@ao/	% REAL overloads the real operator for Analysis objects.
sign	classes/@ao/	% SIGN overloads the sign operator for Analysis objects.%
sort	classes/@ao/	% SORT the values in the AO.
sqrt	classes/@ao/	% SQRT computes the square root of the data in the AO.

std	classes/@ao/	% STD computes the standard deviation of the data in the AO.
sum	classes/@ao/	% SUM computes the sum of the data in the AO.
svd	classes/@ao/	% SVD overloads the determinant function for Analysis objects.
transpose	classes/@ao/	% TRANSPOSE overloads the .' operator for Analysis Objects.
uminus	classes/@ao/	% UMINUS overloads the uminus operator for Analysis objects.
var	classes/@ao/	% VAR computes the variance of the data in the AO.
zeropad	classes/@ao/	% ZEROPAD zero pads the input data series.
tomfir	classes/@pzmodel/	% TOMFIR approximates a pole/zero model with an FIR filter.
tomiir	classes/@pzmodel/	% TOMIIR converts a pzmodel to an IIR filter using a bilinear transform.

Back to top

Trigonometry

Function name	Directory	Description
acos	classes/@ao/	% ACOS overloads the acos method for Analysis objects.
asin	classes/@ao/	% ASIN overloads the asin method for Analysis objects.
atan	classes/@ao/	% ATAN overloads the atan method for Analysis objects.
atan2	classes/@ao/	% ATAN2 overloads the atan2 operator for Analysis objects. Four quadrant inverse tangent.
cos	classes/@ao/	% COS overloads the cos operator for Analysis objects. Cosine of argument in radians.
sin	classes/@ao/	% SIN overloads the sin method for Analysis objects.
tan	classes/@ao/	% TAN overloads the tan method for Analysis objects.

Back to top

Constructor

Function name	Directory	Description
ao	classes/@ao/	% AO analysis object class constructor.
cdata	classes/@cdata/	% CDATA is the constant data class.
data2D	classes/@data2D/	% DATA2D is the abstract base class for 2-dimensional data objects.
data3D	classes/@data3D/	% DATA3D is the abstract base class for 3-dimensional data objects.
fsdata	classes/@fsdata/	% FSDATA frequency-series object class constructor.
history	classes/@history/	% HISTORY History object class constructor.
ltpda_data	classes/@ltpda_data/	% LTPDA_DATA is the abstract base class for ltpda data objects.
ltpda_filter	classes/@ltpda_filter/	% LTPDA_FILTER is the abstract base class for ltpda filter objects.
ltpda_nuo	classes/@ltpda_nuo/	% LTPDA_NUO is the abstract ltpda base class for ltpda non user object classes.
ltpda_obj	classes/@ltpda_obj/	% LTPDA_OBJ is the abstract ltpda base class.
ltpda_uo	classes/@ltpda_uo/	% LTPDA_UO is the abstract ltpda base class for ltpda user object classes.
ltpda_uoh	classes/@ltpda_uoh/	% LTPDA_UOH is the abstract ltpda base class for ltpda user object classes with history
mfir	classes/@mfir/	% MFIR FIR filter object class constructor.
miir	classes/@miir/	% MIIR IIR filter object class constructor.
minfo	classes/@minfo/	% MINFO a helper class for LTPDA methods.
param	classes/@param/	% PARAM Parameter object class constructor.

plist	classes/@plist/	% PLIST Plist class object constructor.
provenance	classes/@provenance/	% PROVENANCE constructors for provenance class.
pz	classes/@pz/	% PZ is the ltpda class that provides a common definition of poles and zeros.
pzmodel	classes/@pzmodel/	% PZMODEL constructor for pzmodel class.
specwin	classes/@specwin/	% % SPECWIN spectral window object class constructor.
ssm	classes/@ssm/	% SSM statespace model class constructor.
time	classes/@time/	% TIME time object class constructor.
timespan	classes/@timespan/	% TIMESPAN timespan object class constructor.
tsdata	classes/@tsdata/	% TSDATA time-series object class constructor.
xydata	classes/@xydata/	% XYDATA X-Y data object class constructor.
xyzdata	classes/@xyzdata/	% XZYDATA X-Y-Z data object class constructor.

Back to top

Output

Function name	Directory	Description
ao2m	classes/@ao/	% AO2M converts an analysis object to an '.m' file based on the history.
char	classes/@ao/	% CHAR overloads char() function for analysis objects.
display	classes/@ao/	% DISPLAY implement terminal display for analysis object.
export	classes/@ao/	% EXPORT export an analysis object to a text file.
extractm	classes/@ao/	% EXTRACTM extracts an m-file from an analysis object and saves it to disk.
extractmdl	classes/@ao/	% EXTRACTMDL extracts an mdl file from an analysis object and saves it to disk.

iplot	classes/@ao/	% IPLOT provides an intelligent plotting tool for LTPDA.
plot	classes/@ao/	% PLOT a simple plot of analysis objects.
display	classes/@cdata/	% DISPLAY implement terminal display for cdata object.
display	classes/@fsdata/	% DISPLAY implement terminal display for fsdata object.
char	classes/@history/	% CHAR convert a param object into a string.
display	classes/@history/	% DISPLAY implement terminal display for history object.
hist2dot	classes/@history/	% HIST2DOT converts a history object to a 'DOT' file suitable for processing with graphviz
hist2m	classes/@history/	% HIST2M writes a new m-file that reproduces the analysis described in the history object.
plot	classes/@history/	% PLOT plots a history object as a tree diagram.
string	classes/@history/	% STRING writes a command string that can be used to recreate the input history object.
char	classes/@mfir/	% CHAR convert a mfir object into a string.
display	classes/@mfir/	% DISPLAY overloads display functionality for mfir objects.
string	classes/@mfir/	% STRING writes a command string that can be used to recreate the input filter object.
char	classes/@miir/	% CHAR convert a miir object into a string.
display	classes/@miir/	% DISPLAY overloads display functionality for miir objects.
string	classes/@miir/	% STRING writes a command string that can be used to recreate the input filter object.
char	classes/@minfo/	% CHAR convert an minfo object into a string.
display	classes/@minfo/	% DISPLAY display an minfo object.
char	classes/@param/	% CHAR convert a param object into a string.

display	classes/@param/	% DISPLAY display a parameter
display2	classes/@param/	% DISPLAY display a parameter
string	classes/@param/	% STRING writes a command string that can be used to recreate the input param object.
char	classes/@plist/	% CHAR convert a parameter list into a string.
display	classes/@plist/	% DISPLAY display plist object.
char	classes/@provenance/	% CHAR convert a provenance object into a string.
display	classes/@provenance/	% DISPLAY overload terminal display for provenance objects.
string	classes/@provenance/	% STRING writes a command string that can be used to recreate the input provenance object.
char	classes/@pz/	% CHAR convert a pz object into a string.
display	classes/@pz/	% DISPLAY display a pz object.
string	classes/@pz/	% STRING writes a command string that can be used to recreate the input pz object.
char	classes/@pzmodel/	% CHAR convert a pzmodel object into a string.
display	classes/@pzmodel/	% DISPLAY overloads display functionality for pzmodel objects.
char	classes/@specwin/	% CHAR convert a specwin object into a string.
display	classes/@specwin/	% DISPLAY overloads display functionality for specwin objects.
plot	classes/@specwin/	% PLOT plots a specwin object.
string	classes/@specwin/	% STRING writes a command string that can be used to recreate the input window object.
char	classes/@ssm/	% CHAR convert a ssm object into a string.
display	classes/@ssm/	% DISPLAY display ssm object.
string	classes/@ssm/	% STRING writes a command string that can be used

		to recreate the input statespace model object.
char	classes/@time/	% CHAR convert a time object into a string.
display	classes/@time/	% DISPLAY overloads display functionality for time objects.
format	classes/@time/	% FORMAT Returns the time in specified format.
char	classes/@timespan/	% CHAR convert a timespan object into a string.
display	classes/@timespan/	% DISPLAY overloads display functionality for timespan objects.
string	classes/@timespan/	% STRING writes a command string that can be used to recreate the input timespan object.
display	classes/@tsdata/	% DISPLAY overloads display functionality for tsdata objects.
display	classes/@xydata/	% DISPLAY overloads display functionality for xydata objects.
char	classes/@xyzdata/	% CHAR convert a ltpda_data-object into a string.
display	classes/@xyzdata/	% DISPLAY overloads display functionality for xyzdata objects.

Back to top

Helper

Function name	Directory	Description
attachm	classes/@ao/	% ATTACHM attach an m file to the analysis object.
attachmdl	classes/@ao/	% ATTACHMDL attach an mdl file to the analysis object.
cat	classes/@ao/	% CAT concatenate AOs into a vector.
copy	classes/@ao/	% COPY makes a (deep) copy of the input AOs.
demux	classes/@ao/	% DEMUX splits the input vector of AOs into a number of output AOs.

find	classes/@ao/	% FIND particular samples that satisfy the input query and return a new AO.
index	classes/@ao/	% INDEX index into an AO array or matrix. This properly captures the history.
join	classes/@ao/	% JOIN multiple AOs into a single AO.
len	classes/@ao/	% LEN overloads the length operator for Analysis objects. Length of the data samples.
md5	classes/@ao/	% MD5 computes an MD5 checksum from an analysis objects.
mdc1_ifo2acc_inloop	classes/@ao/	% MDC1_IFO2ACC_INLOOP calculates the inloop acceleration in the time-domain.
search	classes/@ao/	% SEARCH selects AOs that match the given name.
setDescription	classes/@ao/	% SETDESCRIPTION sets the 'description' property of the ao.
setFs	classes/@ao/	% SETFS sets the 'fs' property of the ao.
setT0	classes/@ao/	% SETT0 sets the 't0' property of the ao.
setX	classes/@ao/	% SETX sets the 'x' property of the ao.
setXY	classes/@ao/	% SETXY sets the 'xy' property of the ao.
setXunits	classes/@ao/	% SETXUNITS sets the 'xunits' property of the ao.
setY	classes/@ao/	% SETY sets the 'y' property of the ao.
setYunits	classes/@ao/	% SETYUNITS sets the 'yunits' property of the ao.
setZ	classes/@ao/	% SETZ sets the 'z' property of the ao.
string	classes/@ao/	% STRING writes a command string that can be used to recreate the input Analysis object(s).
timeshift	classes/@ao/	% TIMESHIFT for AO/tsdata objects, shifts the time axis such that x(1) = 0.
validate	classes/@ao/	% VALIDATE checks that the input Analysis Object is reproducible and valid.

redesign	classes/@mfir/	% REDESIGN redesign the input filter to work for the given sample rate.
redesign	classes/@miir/	% REDESIGN redesign the input filter to work for the given sample rate.
mux	classes/@param/	% MUX concatenate params into a vector.
append	classes/@plist/	% APPEND append a param-object, plist-object or a key/value pair to the parameter list.
combine	classes/@plist/	% COMBINE multiple parameter lists (plist objects) into a single plist.
find	classes/@plist/	% FIND overloads find routine for a parameter list.
isparam	classes/@plist/	% ISPARAM look for a given key in the parameter lists.
nparams	classes/@plist/	% NPARAMS returns the number of param objects in the list.
pset	classes/@plist/	% PSET set or add a key/value pairor a param-object into the parameter list.
remove	classes/@plist/	% REMOVE remove a parameter from the parameter list.
string	classes/@plist/	% STRING converts a plist object to a command string which will recreate the plist object.
string	classes/@pzmodel/	% STRING writes a command string that can be used to recreate the input pzmodel object.
getTimezones	classes/@time/	% GETTIMEZONES Get all possible timezones.
string	classes/@time/	% STRING writes a command string that can be used to recreate the input time object.
setEndT	classes/@timespan/	% SETENDT Set the property 'endT'.
setStartT	classes/@timespan/	% SETSTARTT Set the property 'startT'.
setTimeformat	classes/@timespan/	% SETTIMEFORMAT Set the property 'timeformat'.
setTimezone	classes/@timespan/	% SETTIMEZONE Set the property 'timezone'.

Back to top

Signal Processing

Function name	Directory	Description
cohere	classes/@ao/	% COHERE makes coherence estimates of the time-series objects
compute	classes/@ao/	% COMPUTE performs the given operations on the input AOs.
consolidate	classes/@ao/	% CONSOLIDATE resamples all input AOs onto the same time grid.
cpsd	classes/@ao/	% CPSD makes cross-spectral density estimates of the time-series objects.
delay	classes/@ao/	% DELAY delays a time-series using various methods.
detrend	classes/@ao/	% DETREND detrends the input analysis object using a polynomial of degree N.
dft	classes/@ao/	% DFT computes the DFT of the input time-series at the requested frequencies.
diff	classes/@ao/	% DIFF differentiates the data in AO.
downsample	classes/@ao/	% DOWNSAMPLE AOs containing time-series data.
dropduplicates	classes/@ao/	% DROPDUPLICATES drops all duplicate samples in time-series AOs.
dsmean	classes/@ao/	% DSMEAN performs a simple downsampling by taking the mean of every N samples.
fft	classes/@ao/	% FFT overloads the fft method for Analysis objects.
filter	classes/@ao/	% FILTER overrides the filter function for analysis objects.
filtfilt	classes/@ao/	% FILTFILT overrides the filtfilt function for analysis objects.
firwhiten	classes/@ao/	% FIRWHITEN whitens the input time-series by building an FIR whitening filter.
fixfs	classes/@ao/	% FIXFS resamples the input time-series to have a

		fixed sample rate.
fngen	classes/@ao/	% FNGEN creates an arbitrarily long time-series based on the input PSD.
gapfilling	classes/@ao/	% GAPFILLING fills possible gaps in data.
hist	classes/@ao/	% HIST overloads the histogram function (hist) of MATLAB for Analysis Objects.
ifft	classes/@ao/	% IFFT overloads the ifft operator for Analysis objects.
interp	classes/@ao/	% INTERP interpolate the values in the input AO(s) at new values.
interpmissing	classes/@ao/	% INTERPMISSING interpolate missing samples in a time-series.
lcohere	classes/@ao/	% LCOHERE implement coherence estimation computed on a log frequency axis.
lcpsd	classes/@ao/	% LCPSD implement cross-power-spectral density estimation computed on a log frequency axis.
linedetect	classes/@ao/	% LINEDETECT find spectral lines in the ao/fsdata objects.
lpsd	classes/@ao/	% LPSD implement the LPSD algorithm for analysis objects.
ltfe	classes/@ao/	% LTFE implement transfer-function estimation computed on a log frequency axis.
polyfit	classes/@ao/	% POLYFIT overloads polyfit() function of MATLAB for Analysis Objects.
psd	classes/@ao/	% PSD makes power spectral density estimates of the time-series objects
pwelch	classes/@ao/	% PWELCH makes power spectral density estimates of the time-series objects
resample	classes/@ao/	% RESAMPLE overloads resample function for AOs.
rms	classes/@ao/	% RMS Calculate RMS deviation from spectrum
select	classes/@ao/	% SELECT select particular samples from the input AOs and return new AOs with only those samples.

smoother	classes/@ao/	% SMOOTHER smooths a given series of data points using the specified method.
spectrogram	classes/@ao/	% SPECTROGRAM computes a spectrogram of the given ao/tsdata.
spikecleaning	classes/@ao/	% spikecleaning detects and corrects possible spikes in analysis objects
split	classes/@ao/	% SPLIT split an analysis object into the specified segments.
tfe	classes/@ao/	% TFE makes transfer function estimates of the time-series objects.
upsample	classes/@ao/	% UPSAMPLE overloads upsample function for AOs.
xcorr	classes/@ao/	% XCORR makes cross-correlation estimates of the time-series
resp	classes/@mfir/	% RESP Make a frequency response of the filter.
resp	classes/@miir/	% RESP Make a frequency response of the filter.
resp	classes/@plist/	% RESP shadows miir/iirResp and pzmodel/resp.
resp	classes/@pz/	% RESP returns the complex response of the pz object.
fngen	classes/@pzmodel/	% FNGEN creates an arbitrarily long time-series based on the input pzmodel.
resp	classes/@pzmodel/	% RESP returns the complex response of a pzmodel as an Analysis Object.

Back to top

CATEGORY

Function name	Directory	Description
dopplercorr	classes/@ao/	% FOO description for function 'foo' in one line. Necessary for lookfor functionality.
lincom	classes/@ao/	% LINCOM
timedomainfit	classes/@ao/	% TIMEDOMAINFIT uses lscov to fit a set of time-series

		AOs to a target time-series AO.

Back to top

Relational Operator

Function name	Directory	Description
ge	classes/@ao/	% GE overloads >= operator for analysis objects. Compare the y-axis values.
gt	classes/@ao/	% GT overloads > operator for analysis objects. Compare the y-axis values.
le	classes/@ao/	% LE overloads <= operator for analysis objects. Compare the y-axis values.
lt	classes/@ao/	% LT overloads < operator for analysis objects. Compare the y-axis values.

Back to top

MDC1

Function name	Directory	Description
mdc1_ifo2control	classes/@ao/	% MDC1_IFO2CONTROL converts the input time-series to control forces.
mdc1_x2acc	classes/@ao/	% MDC1_X2ACC converts the input time-series to acceleration with a time-domain filter

Back to top

Arithmetic Operator

Function name	Directory	Description
minus	classes/@ao/	% MINUS implements minus operator for analysis objects.
mpower	classes/@ao/	% MPOWER implements mpower operator for analysis objects.
mrdivide	classes/@ao/	% MRDIVIDE implements mrdivide operator for analysis objects.

mtimes	classes/@ao/	% MTIMES implements mtimes operator for analysis objects.
plus	classes/@ao/	% PLUS implements addition operator for analysis objects.
power	classes/@ao/	% POWER implements power operator for analysis objects.
rdivide	classes/@ao/	% RDIVIDE implements rdivide operator for analysis objects.
times	classes/@ao/	% TIMES implements times operator for analysis objects.
minus	classes/@time/	% MINUS overloads - operator for time objects.
plus	classes/@time/	% PLUS overloads + operator for time objects.

Back to top

Internal

Function name	Directory	Description
applymethod	classes/@cdata/	% APPLYMETHOD applys the given method to the input 2D data.
applyoperator	classes/@cdata/	% APPLYOPERATOR applys the given operator to the two input data objects.
char	classes/@cdata/	% CHAR convert a cdata-object into a string.
copy	classes/@cdata/	% COPY Make copy of cdata objects depending of the second input
getX	classes/@cdata/	% GETX Get the property 'x'.
getY	classes/@cdata/	% GETY Get the property 'y'.
setX	classes/@cdata/	% SETX Set the property 'x'.
setXY	classes/@cdata/	% SETXY Set the property 'xy'.
setXunits	classes/@cdata/	% SETXUNITS Set the property 'xunits'.
setY	classes/@cdata/	% SETY Set the property 'y'.

setYunits	classes/@cdata/	% SETYUNITS Set the property 'yunits'.
copy	classes/@fsdata/	% COPY Make copy of fsdata objects depending of the second input
setEnbw	classes/@fsdata/	% SETENBW Set the property 'enbw'.
setFs	classes/@fsdata/	% SETFS Set the property 'fs'.
setNavs	classes/@fsdata/	% SETNAVS Set the property 'navs'.
setT0	classes/@fsdata/	% SETT0 Set the property 't0'.
copy	classes/@history/	% COPY Make copy of history objects depending of the second input
getNodes	classes/@history/	% GETNODES converts a history object to a nodes structure suitable for plotting as a tree.
getNodes_plot	classes/@history/	% GETNODES_PLOT converts a history object to a nodes structure suitable for plotting as a tree.
copy	classes/@mfir/	% COPY Make copy of mfir objects depending of the second input
copy	classes/@miir/	% COPY Make copy of miir objects depending of the second input
setHistin	classes/@miir/	% SETHISTIN Set the property 'histin'
copy	classes/@minfo/	% COPY Make copy of minfo objects depending of the second input
setMversion	classes/@minfo/	% SETMVERSION Set the property 'mversion'.
copy	classes/@param/	% COPY Make copy of param objects depending of the second input
setKey	classes/@param/	% SETKEY Set the property 'key'.
setKeyVal	classes/@param/	% SETKEYVAL Set the properties 'key' and 'val'
setVal	classes/@param/	% SETVAL Set the property 'val'.
copy	classes/@plist/	% COPY Make copy of plist objects depending of the second input

copy	classes/@provenance/	% COPY Make copy of provenance objects depending of the second input
copy	classes/@pz/	% COPY Make copy of pz objects depending of the second input
cp2iir	classes/@pz/	% CP2IIR Return a,b IIR filter coefficients for a complex pole designed using the bilinear transform.
cz2iir	classes/@pz/	% CZ2IIR return a,b IIR filter coefficients for a complex zero designed using the bilinear transform.
rp2iir	classes/@pz/	% RP2IIR Return a,b coefficients for a real pole designed using the bilinear transform.
rz2iir	classes/@pz/	% RZ2IIR Return a,b IIR filter coefficients for a real zero designed using the bilinear transform.
copy	classes/@pzmodel/	% COPY Make copy of pzmodel objects depending of the second input
getlowerFreq	classes/@pzmodel/	% GETLOWERFREQ gets the frequency of the lowest pole or zero in the model.
getupperFreq	classes/@pzmodel/	% GETUPPERFREQ gets the frequency of the highest pole or zero in the model.
pzm2ab	classes/@pzmodel/	% PZM2AB convert pzmodel to IIR filter coefficients using bilinear transform.
copy	classes/@specwin/	% COPY Make copy of specwin objects depending of the second input
copy	classes/@ssm/	% COPY Make copy of ssm objects depending of the second input
copy	classes/@time/	% COPY Make copy of time objects depending of the second input
setEpochtime	classes/@time/	% SETEPOCHTIME Set the property 'utc_epoch_milli'.
setTime_str	classes/@time/	% SETTIME_STR Set the property 'time_str'.
setTimeformat	classes/@time/	% SETTIMEFORMAT Set the property 'timeformat'.
setTimezone	classes/@time/	% SETTIMEZONE Set the property 'timezone'.

copy	classes/@timespan/	% COPY Make copy of time objects depending of the second input
collapseX	classes/@tsdata/	% COLLAPSEX Checks whether the x vector is evenly sampled and then removes it
copy	classes/@tsdata/	% COPY Make copy of tsdata objects depending of the second input
fixNsecs	classes/@tsdata/	% FIXNSECS fixes the numer of seconds.
getX	classes/@tsdata/	% GETX Get the property 'x'.
growT	classes/@tsdata/	% GROWT grows the time (x) vector if it is empty.
setFs	classes/@tsdata/	% SETFS Set the property 'fs'.
setNsecs	classes/@tsdata/	% SETNSECS Set the property 'nsecs'.
setT0	classes/@tsdata/	% SETT0 Set the property 't0'.
copy	classes/@xydata/	% COPY Make copy of xydata objects depending of the second input
copy	classes/@xyzdata/	% COPY Make copy of xyzdata objects depending of the second input

Back to top

STATESPACE

Function name	Directory	Description
assemble	classes/@ssm/	% assembles embedded subsytems, with exogenous inputs
double	classes/@ssm/	% Convert a statespace model object to double arrays for given i/o
kalman	classes/@ssm/	%kalman applies Kalman filtering to a discrete ssm with given i/o
modifparams	classes/@ssm/	% modifparams enables to options(i_options)y and substitute parameters
modiftimestep	classes/@ssm/	% modiftime modifies the timestep of a ssm object

modify	classes/@ssm/	% modify allows to exectue a string to modify a ssm object
reduce	classes/@ssm/	% reduce enables to do model simplification
simulate	classes/@ssm/	%simulate simulates a discrete ssm with given inputs
ssm2iirpz	classes/@ssm/	% ssm2iirpz converts a statespace model object to a miir or a pz
ssm2ss	classes/@ssm/	% SSM2SS converts a statespace model object to a MATLAB statespace object.

Back to top

Constructor examples of the ZERO class

Functions - Alphabetical List