



- [LTPDA Toolbox](#)
 - [Getting Started with the LTPDA Toolbox](#)
 - [What is the LTPDA Toolbox](#)
 - [System Requirements](#)
 - [Setting-up MATLAB](#)
 - [Additional 3rd-party software](#)
 - [Trouble-shooting](#)
 - [Examples](#)
 - [Introducing LTPDA Objects](#)
 - [Types of User Objects](#)
 - [Creating LTPDA Objects](#)
 - [Working with LTPDA objects](#)
 - [Analysis Objects](#)
 - [Creating Analysis Objects](#)
 - [Saving Analysis Objects](#)
 - [Plotting Analysis Objects](#)
 - [Collection objects](#)
 - [Units in LTPDA](#)
 - [Parameter Lists](#)
 - [Creating Parameters](#)
 - [Creating lists of Parameters](#)
 - [Simulation/modelling](#)
 - [Built-in models of LTPDA](#)
 - [Built-in Analysis Object Models](#)
 - [Built-in Statespace Models](#)
 - [Generating model noise](#)
 - [Franklin noise-generator](#)
 - [Noise generation with given cross-spectral density](#)
 - [Parametric models](#)
 - [Introduction to parametric models in LTPDA](#)
 - [Statespace models](#)
 - [Introduction to Statespace Models with LTPDA](#)
 - [Building Statespace models](#)
 - [Building from scratch](#)
 - [Building from built-in models](#)
 - [Modifying systems](#)
 - [Assembling systems](#)
 - [Simulations](#)
 - [Transfer Function Modelling](#)
 - [Pole/Zero representation](#)
 - [Creating poles and zeros](#)
 - [Building a model](#)
 - [Model helper GUI](#)
 - [Sum of partial fractions representation](#)
 - [Rational representation](#)
 - [Converting models between different representations](#)
 - [Converting models to digital filters](#)
 - [Signal Pre-processing in LTPDA](#)
 - [Downsampling data](#)

- [Upsampling data](#)
- [Resampling data](#)
- [Interpolating data](#)
- [Spikes reduction in data](#)
- [Data gap filling](#)
- [Noise whitening](#)
- [Signal Processing in LTPDA](#)
 - [Digital Filtering](#)
 - [IIR Filters](#)
 - [FIR Filters](#)
 - [Filter banks](#)
 - [Applying digital filters to data](#)
 - [Discrete Derivative](#)
 - [Spectral Estimation](#)
 - [Introduction](#)
 - [Spectral Windows](#)
 - [What are LTPDA spectral windows?](#)
 - [Using spectral windows](#)
 - [Spectral Estimation Methods](#)
 - [Power spectral density estimates](#)
 - [Cross-spectral density estimates](#)
 - [Cross coherence estimates](#)
 - [Transfer function estimates](#)
 - [Log-scale power spectral density estimates](#)
 - [Log-scale cross-spectral density estimates](#)
 - [Log-scale cross coherence density estimates](#)
 - [Log-scale transfer function estimates](#)
 - [Fitting Algorithms](#)
 - [Polynomial Fitting](#)
 - [Z-Domain Fit](#)
 - [S-Domain Fit](#)
- [Graphical User Interfaces in LTPDA](#)
 - [The LTPDA Launch Bay](#)
 - [The LTPDA Workbench](#)
 - [Loading the LTPDA Workbench](#)
 - [Mouse and keyboard actions](#)
 - [The canvas](#)
 - [Building pipelines by hand](#)
 - [Block types](#)
 - [Adding blocks to the canvas](#)
 - [Setting block properties and parameters](#)
 - [Connecting blocks](#)
 - [Creating subsystems](#)
 - [Using the Workbench Shelf](#)
 - [Accessing the shelf](#)
 - [Creating shelf categories](#)
 - [Adding and using shelf subsystems](#)
 - [Importing and exporting shelf categories](#)
 - [Execution plans](#)
 - [Editing the plan](#)
 - [Linking pipelines](#)
 - [Executing pipelines](#)
 - [The LTPDA Workspace Browser](#)
 - [The pole/zero model helper](#)

- [The Spectral Window GUI](#)
 - [The constructor helper](#)
 - [The LTPDA object explorer](#)
- [Working with an LTPDA Repository](#)
 - [What is an LTPDA Repository](#)
 - [Connecting to an LTPDA Repository](#)
 - [Submitting LTPDA objects to a repository](#)
 - [Exploring an LTPDA Repository](#)
 - [Retrieving LTPDA objects from a repository](#)
- [Class descriptions](#)
 - [ao Class](#)
 - [ssm Class](#)
 - [smodel Class](#)
 - [filterbank Class](#)
 - [mfir Class](#)
 - [miir Class](#)
 - [pzmodel Class](#)
 - [parfrac Class](#)
 - [rational Class](#)
 - [timespan Class](#)
 - [plist Class](#)
 - [matrix Class](#)
 - [collection Class](#)
 - [Constructor Examples](#)
 - [General constructor examples](#)
 - [Constructor examples of the AO class](#)
 - [Constructor examples of the SMODEL class](#)
 - [Constructor examples of the MFIR class](#)
 - [Constructor examples of the MIIR class](#)
 - [Constructor examples of the PZMODEL class](#)
 - [Constructor examples of the PARFRAC class](#)
 - [Constructor examples of the RATIONAL class](#)
 - [Constructor examples of the TIMESPAN class](#)
 - [Constructor examples of the PLIST class](#)
 - [Constructor examples of the SPECWIN class](#)
- [Functions - By Category](#)
- [LTPDA Training Session 1](#)
 - [Topic 1 - The basics of LTPDA](#)
 - [Introducing Analysis Objects](#)
 - [Making AOs](#)
 - [Making a time-series AO](#)
 - [Basic math with AOs](#)
 - [Saving and loading AOs](#)
 - [Constructing AOs from data files](#)
 - [Writing LTPDA scripts](#)
 - [IFO/Temperature Example - Introduction](#)
 - [Topic 2 - Pre-processing of data](#)
 - [Downsampling a time-series AO](#)
 - [Upsampling a time-series AO](#)
 - [Resampling a time-series AO](#)
 - [Interpolation of a time-series AO](#)
 - [Remove trends from a time-series AO](#)
 - [Whitening noise](#)
 - [Select and find data from an AO](#)

[Split and join AOs](#)

- [IFO/Temperature Example - Pre-processing](#)

- [Topic 3 - Spectral Analysis](#)

- [Power Spectral Density estimation](#)
 - [Example 1: Simply PSD](#)
 - [Example 2: Windowing data](#)
 - [Example 3: Log-scale PSD on MDC1 data](#)
- [Empirical Transfer Function estimation](#)
- [IFO/Temperature Example - Spectral Analysis](#)

- [Topic 4 - Transfer function models and digital filtering](#)

- [Create transfer function models in s domain](#)
 - [Pole zero model representation](#)
 - [Partial fraction representation](#)
 - [Rational representation](#)
- [Transforming models between representations](#)
- [Modelling a system](#)
- [How to filter data](#)
 - [By discretizing transfer function models](#)
 - [By defining filter properties](#)
- [IFO/Temperature Example - Simulation](#)

- [Topic 5 - Model fitting](#)

- [System identification in z-domain](#)
- [Generation of noise with given psd](#)
- [Fitting time series with polynomials](#)
- [Non-linear least square fitting of time series](#)
- [IFO/Temperature Example - signal subtraction](#)

- [LTPDA Web Site](#)

LTPDA Toolbox

Welcome to LTPDA!

LTPDA provides a framework for doing object-oriented data analysis. LTPDA objects can represent typical data structures needed for data analysis, for example, time-series data or digital filters. These objects can then flow through a data analysis pipeline where at each stage they record the actions. The output objects of a data analysis pipeline therefore contain a full history of the processing steps that have been performed in reaching this point. This history can be view to allow others to understand how results were arrived at, but it can also be used to recreate the result.

 IFO/Temperature Example – signal subtraction Getting Started with the LTPDA Toolbox 

©LTP Team

Getting Started with the LTPDA Toolbox

What is the LTPDA Toolbox?	Overview of the main functionality of the Toolbox
System Requirements	Supported platforms, MATLAB versions, and required toolboxes.
Setting up MATLAB to work with LTPDA	Installing and editing the LTPDA Startup file.
Starting the LTPDA Toolbox	

What is the LTPDA Toolbox

This section covers the following topics:

- [Overview](#)
- [Features of the LTPDA Toolbox](#)
- [Expected Background for Users](#)

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). However, the toolbox can be used for any general purpose signal processing, and is particularly useful for analysis of typical lab-based experiments.

The Toolbox implements accountable and reproducible data analysis within MATLAB[®].

With the LTPDA Toolbox you operate with LTPDA Objects. An LTPDA 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 LTPDA Objects.
- Save and load objects from XML files.
- Plot/view the history of the processing in any particular object.
- Submit and retrieve objects 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.

System Requirements

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

- [Platforms](#)
- [MATLAB and Related Products](#)
- [Additional Programs](#)

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. In addition, the following MathWorks Toolboxes are required for some features:

Component	Version	Comment
MATLAB	7.6 (R2009a)	
Signal Processing Toolbox	6.11	
Symbolic Math Toolbox	5.2	
Optimization Toolbox*	4.2	The optimisation toolbox is only used for a small number of methods.

The toolbox should also work with MATLAB 2008b. The toolbox may also work, to some extent, with earlier versions of MATLAB, but the symbolic math toolbox changed at 2008b, so you're on your own.

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](#)
- [Starting LTPDA Toolbox](#)
- [Edit the LTPDA Preferences](#)

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`).

Starting LTPDA Toolbox

To start using the LTPDA Toolbox, execute the following command on the MATLAB terminal:

```
ltpda_startup
```

This should launch the LTPDA Launchbay, and you should see the LTPDA logo on the MATLAB terminal. When you run this for the first time, you will also be presented with the LTPDA Preferences GUI from where you can edit the preferences for the toolbox (see below).

If everything has gone well, you should be able to run a set of built-in tests by doing:

```
run_tests
```

This will run about 100 test scripts. These test scripts can be found in `$ltpda_toolbox/examples` and serve as useful example scripts.

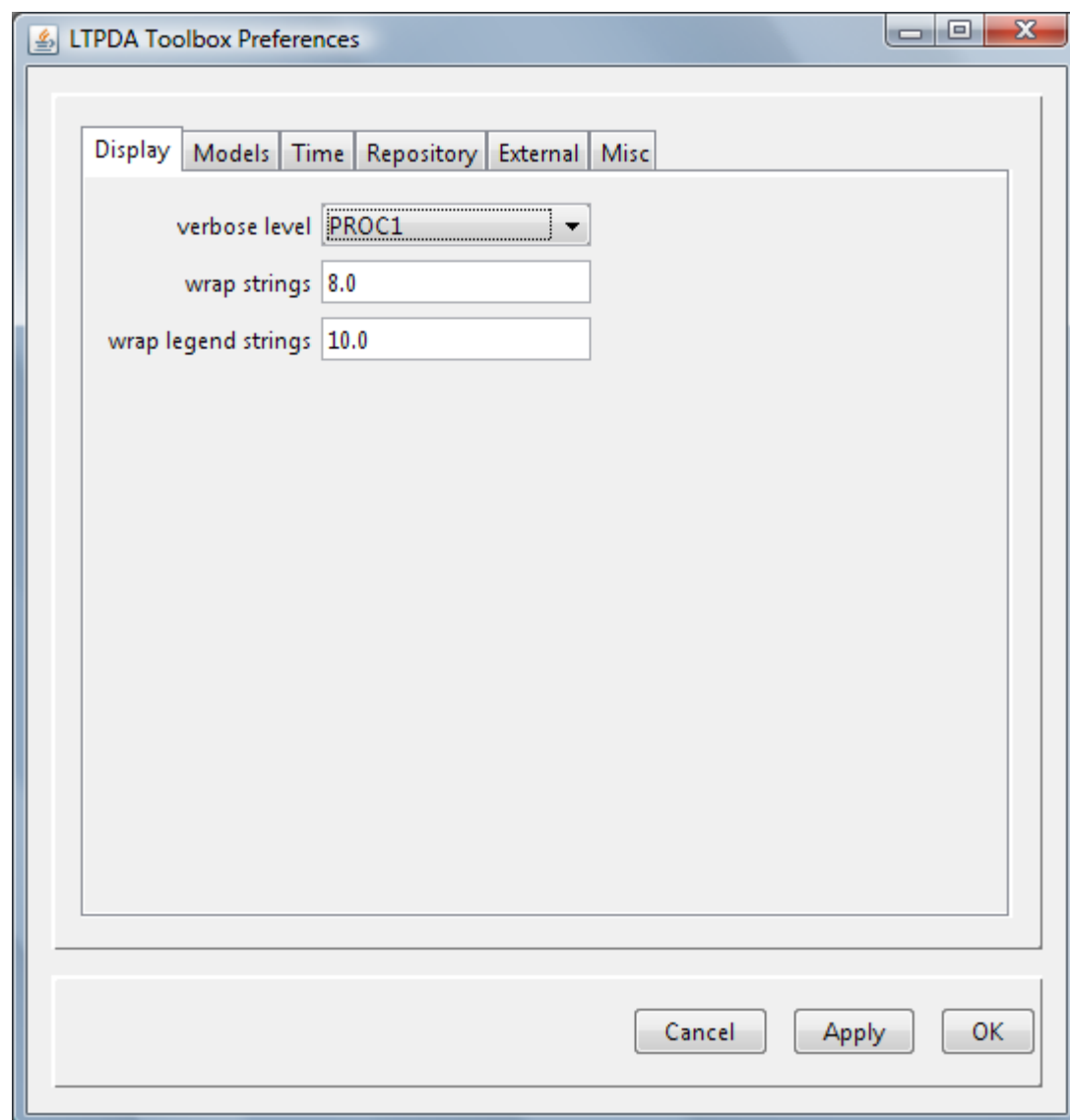
In order to automatically start the LTPDA Toolbox when MATLAB starts up, add the command `ltpda_startup` to your own `startup.m` file. See `>> doc startup` for more details on installing and editing your own `startup.m` file.

Edit the LTPDA Preferences

The LTPDA Toolbox comes with a default set of starting preferences. These may need to be edited for your particular system (though most of the defaults should be fine). To edit the preferences, you first need to have the LTPDA toolbox installed as described above, then run the command

```
LTPDAprefs
```

or click on the "LTPDA Preferences" button on the launchbay. You should see the following GUI:



Edit all the preferences you want and then click apply to save the preferences. These new preferences will be used each time you start LTPDA.

◀ System Requirements

Additional 3rd-party software ▶

©LTP Team

Additional 3rd-party software

Some features of the LTPDA Toolbox require additional 3rd-party software to be installed. These are listed below.

Graphviz

In order to use the commands listed below, the [Graphviz](#) package must be installed.

Method	Description
history/dotview	Convert a history object to a tree-diagram using the DOT interpreter.
ssm/dotview	Convert the statespace model object to a block-diagram using the DOT interpreter.
report	Generates a HTML report about the input objects which includes a DOT block-diagram of the history.

The following installation guidelines can be used for different platforms

- [Installation Guide for Windows](#)
- [Installation Guide for Mac OS X](#)
- [Installation Guide for Linux](#)

Windows

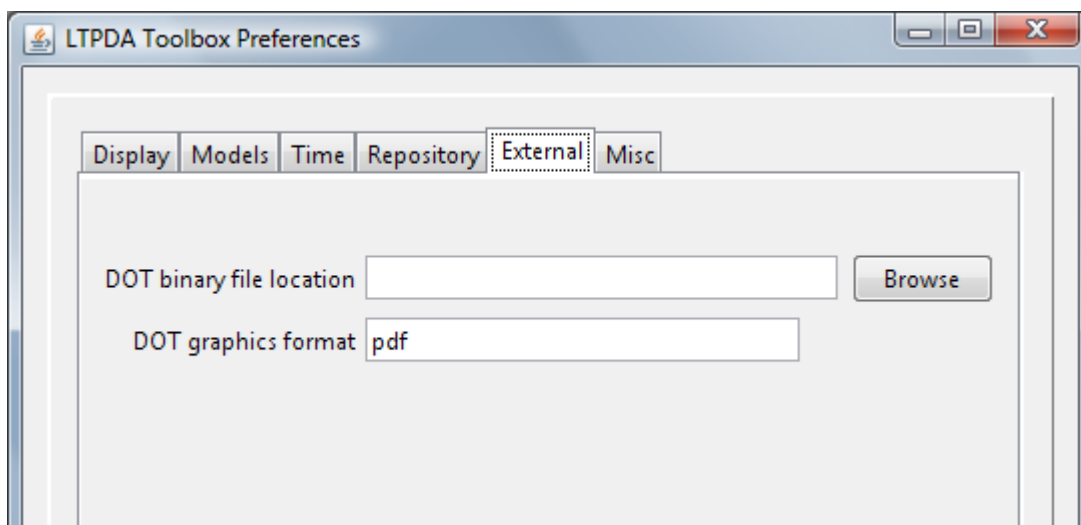
- Download the relevant package from Downloads section of www.graphviz.org.
- Install the package by following the relevant instructions.
- Set the two relevant preferences with your LTPDA Preferences Panel
 - For this start the LTPDApres



- or press the "LTPDA Preferences" button on the LTPDA Launch Bay



3. Select on your LTPDA Preferences Panel the category "External Programs"



4. Set the path to the 'dot.exe' binary in the editable text field "DOT binary". If you perform the default installation, this should be something like:
'c:\Program Files\Graphviz2.20\bin\dot.exe';
5. Define in the editable text field "DOT format" the graphics format to output. See [formats](#) for available formats. To view the final graphics file you must have a suitable viewer for that graphics format installed on the system. For example, to output as PDF

Mac OS X

1. Choose from:
 1. From graphviz:
 - Download the relevant package from Downloads section of www.graphviz.org.

Install the package by following the relevant instructions.

2. From Fink:
 - If you use the fink package manager, in a terminal: `> fink install graphviz`
2. Set the two relevant preferences with your LTPDA Preferences Panel.
 1. Start the LTPDA Preferences Panel. For this follow the step 3.1 or 3.2 of the window installation.
 2. Set the path to the 'dot' binary in the editable text field "DOT binary". If you perform the default installation from fink, this should be something like:
'/sw/bin/dot'
 3. Define in the editable text field "DOT format" the graphics format to output. See [formats](#) for available formats. To view the final graphics file you must have a suitable viewer for that graphics format installed on the system. For example, to output as PDF

Linux

1. Choose from:
 1. From graphviz:
 - Download the relevant package from Downloads section of www.graphviz.org.
 - Install the package by following the relevant instructions.
 2. From terminal (Ubuntu):
 - Please type in a terminal: `>sudo apt-get install graphviz`
 3. From graphical package manager like **YaSt, Synaptic, Adept, ...**
 - Start your graphical package manager
 - Search for the `>graphviz` package
 - Select the package and all depending packages and install these packages.
2. Set the two relevant preferences with your LTPDA Preferences Panel.
 1. Start the LTPDA Preferences Panel. For this follow the step 3.1 or 3.2 of the window installation.
 2. Set the path to the 'dot' binary in the editable text field "DOT binary". If you perform the default installation from the terminal, this should be something like:
'/usr/bin/dot';
even 'dot' without the path should work
 3. Define in the editable text field "DOT format" the graphics format to output. See [formats](#) for available formats. To view the final graphics file you must have a suitable viewer for that graphics format installed on the system. For example, to output as PDF
3. Define a program in MATLAB which opens the file.
 1. The default program to open a pdf file is the Acrobat Reader
 2. Define another program under File -> Preferences -> Help -> PDF Reader

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).

2. LTPDA Directory Name

Problems have been seen on Windows machines if the LTPDA toolbox directory name contains `'.'`. In this case, just rename the base LTPDA directory before adding it to the MATLAB path.

3. Problems to execute MEX files

User with the operating system "Windows XP" will get trouble if you want to execute MEX files. This happens if you are using for example the methods `'detrend'`, `'smoother'`, `'lpsd'`, ...

You will get the following error:

```
??? Invalid MEX-file 'C:\ltpda_toolbox\ltpda\src\ltpda_dft\ltpda_dft.mexw32':  
This application has failed to start because the application configuration is incorrect.  
Reinstalling the application may fix this problem.
```

You can solve this problem by installing the Microsoft Visual C++ 2008 Redistributable Package. This is necessary because all the MEX files are compiled with Microsoft Visual C++ 2008. You can install this package from the LTPDA toolbox or from the official Microsoft web page.

- LTPDA toolbox

- [Windows 32 bit](#)
- [Windows 64 bit](#)

You will find the files in the directory: \$LTPDA_TOOLBOX_DIR/ltppda/src

- Official Microsoft web page

- [Windows 32 bit](#)
- [Windows 64 bit](#)

◀ Additional 3rd-party software

Examples ▶

©LTP Team

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

[Constructor examples of the AO class](#)

[Constructor examples of the MFIR class](#)

[Constructor examples of the MIIR class](#)

[Constructor examples of the PZMODEL class](#)

[Constructor examples of the PARFRAC class](#)

[Constructor examples of the RATIONAL class](#)

[Constructor examples of the TIMESPAN class](#)

[Constructor examples of the PLIST class](#)

[Constructor examples of the SPECWIN class](#)

For help in constructing LTPDA objects, you may also launch the Constructor Helper GUI.



[See the Constructor Helper GUI documentation for more information.](#)

◀ Troubleshooting

Introducing LTPDA Objects ▶

©LTP Team

Introducing LTPDA Objects

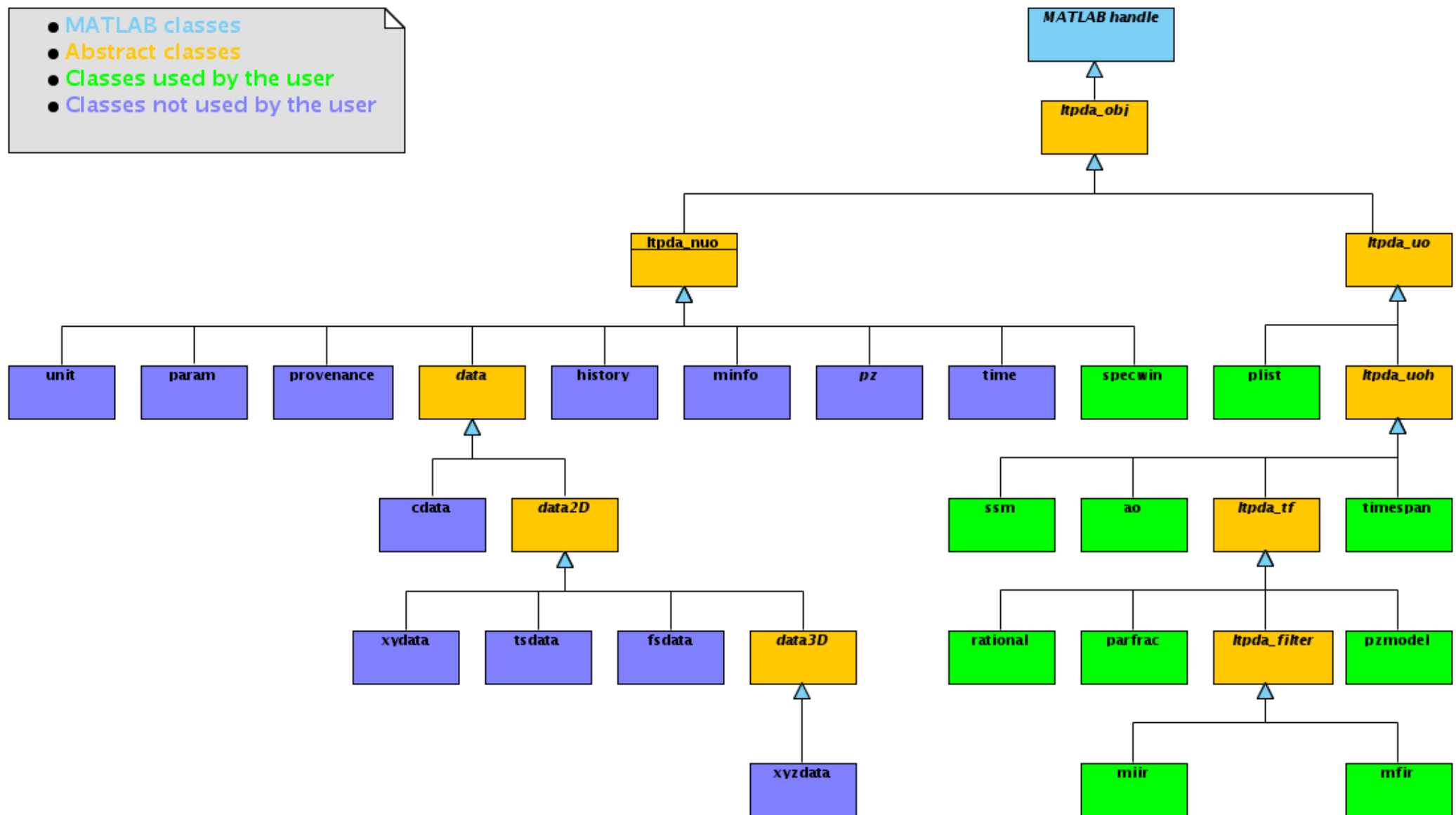
The LTPDA toolbox is object oriented and as such, extends the MATLAB object types to many others. All data processing is done using objects and methods of those classes.

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`.

- MATLAB classes
- Abstract classes
- Classes used by the user
- Classes not used by the user



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.

Details of each class are given in:

analysis object class
statespace model class
rational class
partial fraction class
pole/zero model class
iir filter class
fir filter class
timespan class
parameter list class
spectral window class
time class
pole/zero class
method info class
history class
provenance class
parameter class
unit class
constant data class
xy data class
time-series data class
frequency-series data class
xyz data class

◀ Examples

Types of User Objects ▶

©LTP Team

Types of User Objects

LTPDA has various classes of objects which the user can create and manipulate. All of these classes of object (termed 'user objects'), with the exception of the `plist` class, track their history through all manipulations. The following table summarizes these types.

Class	Description
<code>plist</code>	A class of object that allows storing a set of key/value pairs. This is the equivalent to a 'dictionary' in other languages, like python or Objective-C. Since a history step must contain a <code>plist</code> object, this class cannot itself track history, since it would be recursive.
<code>ao</code>	A class of object that implements Analysis Objects. Such objects can store various types of numeric data: time-series, frequency-series, x-y data series, x-y-z data series, and arrays of numbers.
<code>mfir</code>	A class of object that implements Finite Impulse Response (FIR) digital filters.
<code>miir</code>	A class of object that implements Infinite Impulse Response (IIR) digital filters.
<code>pzmodel</code>	A class of object that represents transfer functions given in a pole/zero format.
<code>rational</code>	A class of object that represents transfer functions given in rational form.
<code>parfrac</code>	A class of object that represents transfer functions given as a series of partial fractions.
<code>timespan</code>	A class of object that represents a span of time. A <code>timespan</code> object has a start time and an end time.
<code>ssm</code>	A class of object that represents a statespace model.
<code>smodel</code>	A class of object that represents a parametric model of a chosen x variable. Such objects can be combined, manipulated symbolically, and then evaluated numerically.
<code>filterbank</code>	A class of object that represents a bank of digital filters. The filter bank can be of type 'parallel' or 'serial'.
<code>matrix</code>	A class of object that allows storing other User Objects in a matrix-like array. There are various methods which act on the

	object array as if it were a matrix of the underlying data. For example, you can form a matrix of Analysis Objects, then compute the determinant of this matrix, which will yield another matrix object containing a single AO.
collection	A class of object that allows storing other User Objects in a cell-array. This is purely a convenience class, aimed at simplifying keeping a collection of different classes of user objects together. This is useful, for example, for submitting or retrieving a collection of user objects from/to an LTPDA Repository.

Creating LTPDA Objects

Creating LTPDA objects within MATLAB is achieved by calling the constructor of the class of object you want to create. Typically, each class within LTPDA has many possible constructor calls which can produce the objects using different methods and inputs.

For example, if we want to create a parameter list object (`plist`), the help documentation of the `plist` class describes the various constructor methods. Type `help plist` to see the documentation.

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](#)
- [Copying objects](#)
- [Exploring objects](#)

Calling object methods

Each class in LTPDA has a set of methods (functions) which can operate/act on instances of the class (objects). For example, the AO class has a method `psd` which can compute the Power Spectral Density estimate of a time-series AO.

To see which methods a particular class has, use the `methods` command. For example,

```
>> methods('ao')
```

To call a method on an object, `obj.method`, or, `method(obj)`. For example,

```
>> b = a.psd
```

or

```
>> b = psd(a)
```

Additional arguments can be passed to the method (a `plist`, for example), as follows:

```
>> b = a.psd(pl)
```

or

```
>> b = psd(a, pl)
```

In order to pass multiple objects to a method, you must use the form

```
>> b = psd(a1, a2, pl)
```

Some methods can behave as modifiers which means that the object which the method acts on is modified. To modify an object, just give no output. If we start with a time-series AO then modify it with the `psd` method,

```
>> a = ao(1:100, randn(100,1), 10)
>> a
M:    running ao/display
----- ao 01: a -----

      name:    None
      data: (0,0.840375529753905) (0.1,-0.88803208232901) (0.2,0.100092833139322)
(0.3,-0.544528929990548) (0.4,0.303520794649354) ...
      ----- tsdata 01 -----

          fs:    10
          x:    [100 1], double
          y:    [100 1], double
          dx:    [0 0], double
          dy:    [0 0], double
      xunits:    [s]
      yunits:    []
          nsecs: 10
          t0:    1970-01-01 00:00:01.000
      -----

      hist:    ao / ao / SId: fromVals ... -->$Id: ao .... S
      mdlfile: empty
      description:
      UUID:    8cffab46-61f0-494a-af03-eb310aa76114
      -----
```

Then call the `psd` method:

```
>> a.psd
M:    running ao/psd
M:    running ao/len
M:    running ao/len
M:    running ao/display
----- ao 01: PSD(a) -----

      name:    PSD(a)
      data: (0,0.117412356146407) (0.1,0.179893990497347) (0.2,0.173957816470448)
(0.3,0.245076068355785) (0.4,0.213036543621994) ...
      ----- fsdata 01 -----

          fs:    10
          x:    [51 1], double
          y:    [51 1], double
          dx:    [0 0], double
          dy:    [0 0], double
      xunits:    [Hz]
      yunits:    [Hz^(-1)]
          t0:    1970-01-01 00:00:01.000
          navs:    1
      -----

      hist:    ao / psd / SId: psd.m,v 1.52 2009/09/05 05:57:32 mauro Exp S
      mdlfile: empty
      description:
      UUID:    0d2395cd-22af-4645-a94c-69fa32c15982
      -----
```

then the object `a` is converted to a frequency-series AO.

This modifier behaviour only works with certain methods, in particular, methods requiring more than one input object will not behave as modifiers.

Setting object properties

All object properties must be set using the appropriate setter method. For example, to set the name of a IIR filter object,

```
>> ii = miir();
>> ii.setName('My Filter');
```

Reading the value of a property is achieved by:

```
>> ii.name
ans =
My Filter
```

Copying objects

Since all objects in LTPDA are handle objects, creating copies of objects needs to be done differently than in standard MATLAB. For example,

```
>> a = ao();
>> b = a;
```

in this case, the variable `b` is a copy of the handle `a`, not a copy of the object pointed too by the handle `a`. To see how this behaves,

```
>> a = ao();
>> b = a;
>> b.setName('My Name');
>> a.name

ans =
My Name
```

Copying the object can be achieved using the copy constructor:

```
>> a = ao();
>> b = ao(a);
>> b.setName('My Name');
>> a.name

ans =
none
```

In this case, the variable `b` points to a new distinct copy of the object pointed to by `a`.

Exploring objects

A browsing tool is provided on purpose to enable LTPDA objects exploring.

[See the LTPDA Objects Explorer GUI documentation.](http://www.lisa.aei-hannover.de/ltpda/usermanual/ug/objects_working.html)

©LTP Team

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 (these are always methods of the AO class) but there are also utility functions which do not take AOs as inputs, as well as methods of other classes. Functions in the toolbox are designed to be as simple and elementary as possible.

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](#)
- [Creating AOs from XML or MAT files](#)
- [Creating AOs from MATLAB functions](#)
- [Creating AOs from functions of time](#)
- [Creating AOs from window functions](#)
- [Creating AOs from waveform descriptions](#)
- [Creating AOs from pole zero models](#)

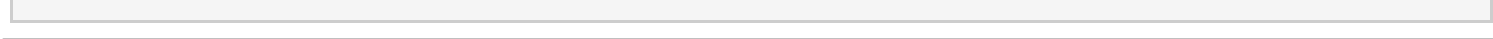
Creating AOs from text files.

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('data.txt')
----- ao 01: data.txt_01_02 -----
      name: data.txt_01_02
      data: (0, -1.06421341288933) (0.1, 1.60345729812004) (0.2, 1.23467914689078) ...
      ----- tsdata 01 -----
              fs: 10
              x: [100 1], double
              y: [100 1], double
              dx: [0 0], double
              dy: [0 0], double
      xunits: [s]
      yunits: []
      nsecs: 10
      t0: 1970-01-01 00:00:00.000
      -----
      hist: ao / ao / SId: fromDatafile ... $-->$Id: ao ... S
      mdlfile: empty
      description:
      UUID: e6ccfcb6-da49-4f4c-8c2c-3054fe5d2762
      -----
```

As with most constructor calls, an equivalent action can be achieved using an input [Parameter List](#).

```
>> a = ao(plist('filename', 'data.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 01: a -----
      name:  None
      data:  (0,-0.493009815316451) (0.1,-0.180739356415037) (0.2,0.045841105713705)
...
      ----- tsdata 01 -----
              fs:  10
              x:  [100 1], double
              y:  [100 1], double
              dx:  [0 0], double
              dy:  [0 0], double
      xunits:  [s]
      yunits:  []
      nsecs:  10
      t0:  1970-01-01 00:00:01.000
      -----
      hist:  ao / ao / SId: fromVals ... $-->$Id: ao ... S
      mdlfile:  empty
      description:
      UUID:  2fed6155-6468-4533-88f6-e4b27bc6e1aa
      -----
```

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 01: a -----
      name:  None
      data:  -1.28325610460477 -2.32895451628334 0.901931466951714 -1.83563868373519
0.06675 ...
      ----- cdata 01 -----
              y:  [1000x1], double
              dy:  [0x0], double
      yunits:  []
      -----
      hist:  ao / ao / SId: fromFcn ... $-->$Id ... $
      mdlfile:  empty
      description:
      UUID:  0072f8d0-f804-472b-a4aa-e9ec6a8de803
      -----
```

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 01: a -----
name: None
data: (0,1.37694916561229) (0.1, -0.820427237640771) (0.2,1.09228819960292) ...
----- tsdata 01 -----
fs: 10
x: [1000 1], double
y: [1000 1], double
dx: [0 0], double
dy: [0 0], double
xunits: [s]
yunits: []
nsecs: 100
t0: 1970-01-01 00:00:00.000
-----
hist: ao / ao / SId: fromTSfcn ... $-->$Id: ao ... S
mdlfile: empty
description:
UUID: c0f481cf-4bdd-4a91-bc78-6d34f8222313
-----
```

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)
----- specwin/1 -----
type: Hanning
alpha: 0
psll: 31.5
rov: 50
nenbw: 1.5
w3db: 1.4382
flatness: -1.4236
ws: 500
ws2: 375.0000000000001
win: [0 9.86957193144233e-06 3.94778980919441e-05 8.88238095955174e-05 0.0001579
...
version: SId: specwin.m,v 1.67 2009/09/01 09:25:24 ingo Exp S
-----

>> a = ao(w)
----- ao 01: ao(Hanning) -----
name: ao(Hanning)
data: 0 9.86957193144233e-06 3.94778980919441e-05 8.88238095955174e-05
0.0001579 ...
----- cdata 01 -----
y: [1x1000], double
dy: [0x0], double
yunits: []
-----
hist: ao / ao / SId: fromSpecWin ... $-->$Id: ao ... S
mdlfile: empty
description:
UUID: ea1a9036-b9f5-4bdb-b3a3-211e9d697060
-----
```

It is also possible to pass the information about the window as a plist to the `ao` constructor.

```
>> ao(plist('win', 'Hanning', 'length', 1000))
----- ao 01: ao(Hanning) -----
      name:  ao(Hanning)
      data:  0 9.86957193144233e-06 3.94778980919441e-05 8.88238095955174e-05
0.0001579 ...
      ----- cdata 01 -----
              y:  [1x1000], double
              dy:  [0x0], double
            yunits:  []
      -----
      hist:  ao / ao / SId: fromSpecWin ... -->$Id: ao ... S
      mdlfile:  empty
      description:
      UUID:  5b81f67a-45b9-43f8-a74a-bc5161fd718f
      -----
```

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.

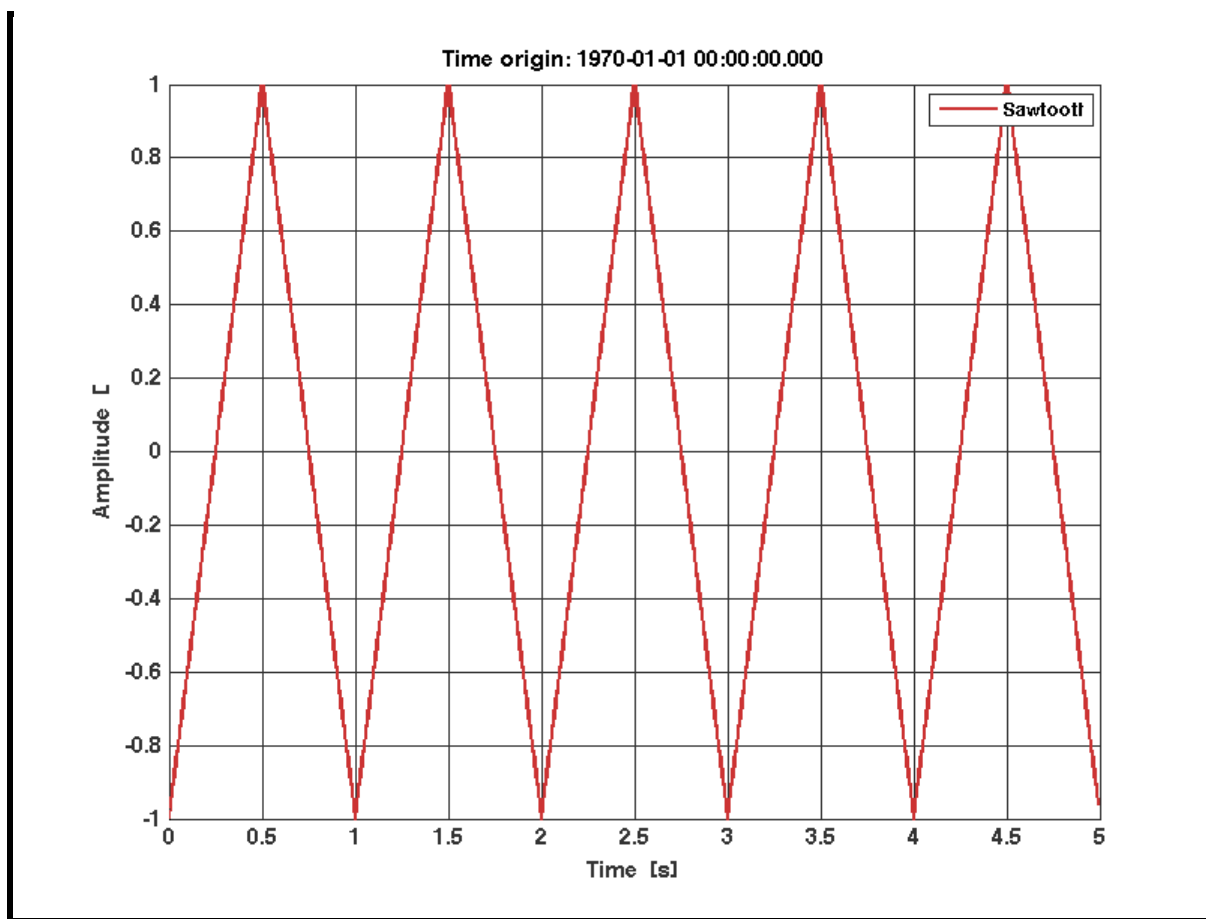
Creating AOs from waveform descriptions

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 01: Sawtooth -----
      name:  Sawtooth
      data:  (0, -1) (0.01, -0.96) (0.02, -0.92) (0.03, -0.88) (0.04, -0.84) ...
      ----- tsdata 01 -----
              fs:  100
              x:  [500 1], double
              y:  [500 1], double
              dx:  [0 0], double
              dy:  [0 0], double
            xunits:  [s]
            yunits:  []
              nsecs:  5
              t0:  1970-01-01 00:00:00.000
      -----
      hist:  ao / ao / SId: fromWaveform ... $-->$Id: ao ... S
      mdlfile:  empty
      description:
      UUID:  cb76c866-ee3f-47e6-bb29-290074666e43
      -----
```

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

```
ipLOT(asaw);
```



Creating AOs from pole zero models

When generating an AO from a pole zero model, the noise generator function is called. This a method to generate arbitrarily long time series with a prescribed spectral density. The algorithm is based on the following paper:

Franklin, Joel N.: *Numerical simulation of stationary and non-stationary gaussian random processes*, SIAM review, Volume { 7}, Issue 1, page 68--80, 1965.

The Document *Generation of Random time series with prescribed spectra* by Gerhard Heinzel (S2-AEI-TN-3034)

corrects a mistake in the aforesaid paper and describes the practical implementation. The following code creates an AO with a time series having a prescribed spectral density, defined by the input pole zero model:

```
f1 = 5;
f2 = 10;
f3 = 1;
gain = 1;
fs = 10; %sampling frequency
nsecs = 100; %number of seconds to be generated

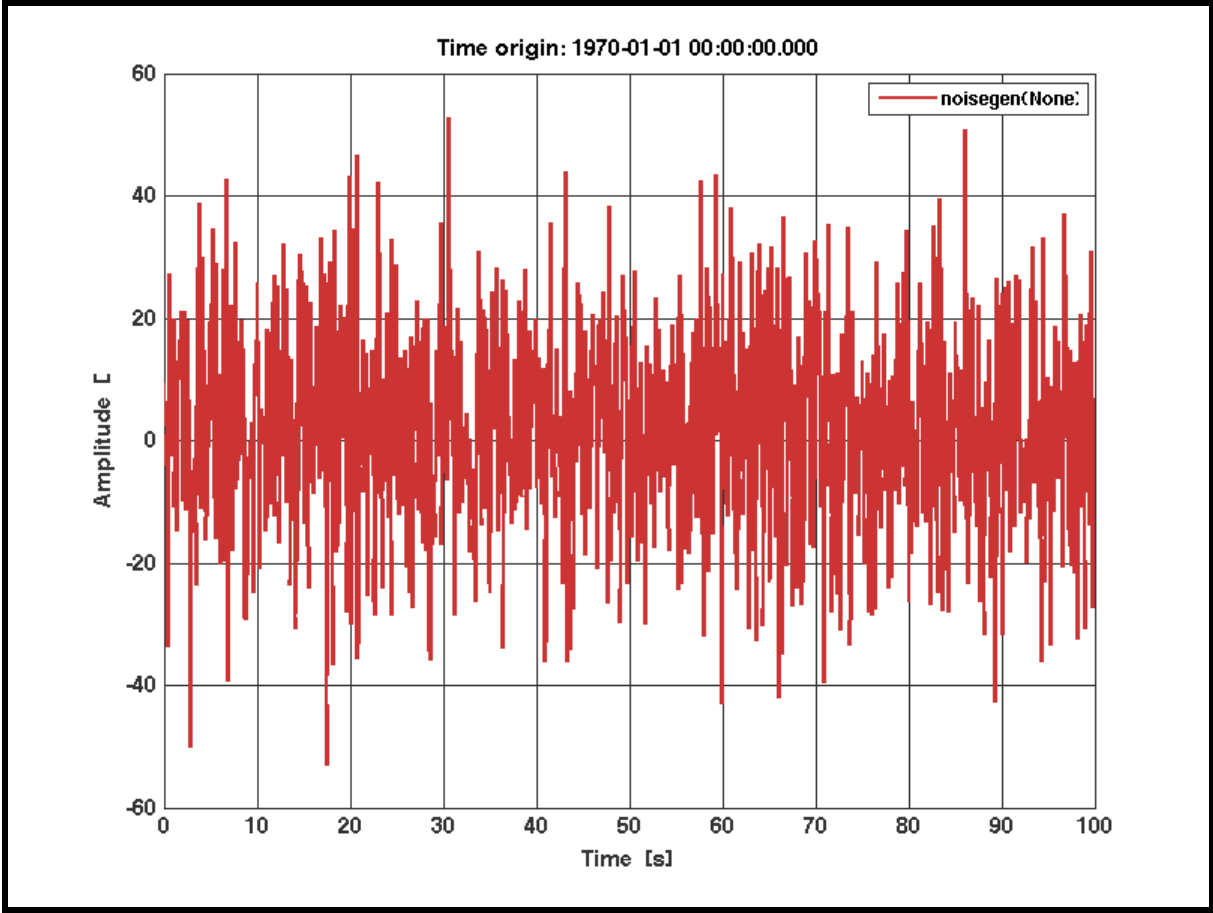
p = [pz(f1) pz(f2)];
z = [pz(f3)];
pzm = pzmodel(gain, p, z);
a = ao(pzm, nsecs, fs)
----- ao 01: noisegen(None) -----
name: noisegen(None)
data: (0,9.20287001568168) (0.1, -3.88425345108961) (0.2,6.31042718242658) ...
----- tsdata 01 -----

fs: 10
x: [1000 1], double
y: [1000 1], double
dx: [0 0], double
```

```
dy: [0 0], double
xunits: [s]
yunits: []
nsecs: 100
t0: 1970-01-01 00:00:00.000
-----
hist: ao / ao / SId: fromPzmodel ... $-->$Id: ao ... S
mdlfile: empty
description:
  UUID: 4a89d910-8672-475f-91cd-4fcc4b52a6b4
-----
```

You can call the `ipLOT` function to view the resulting noise.

```
ipLOT(a);
```



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
```


Plotting Analysis Objects

The data in an AO can be plotted using the `ipplot` method.

The `ipplot` method provides an 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.

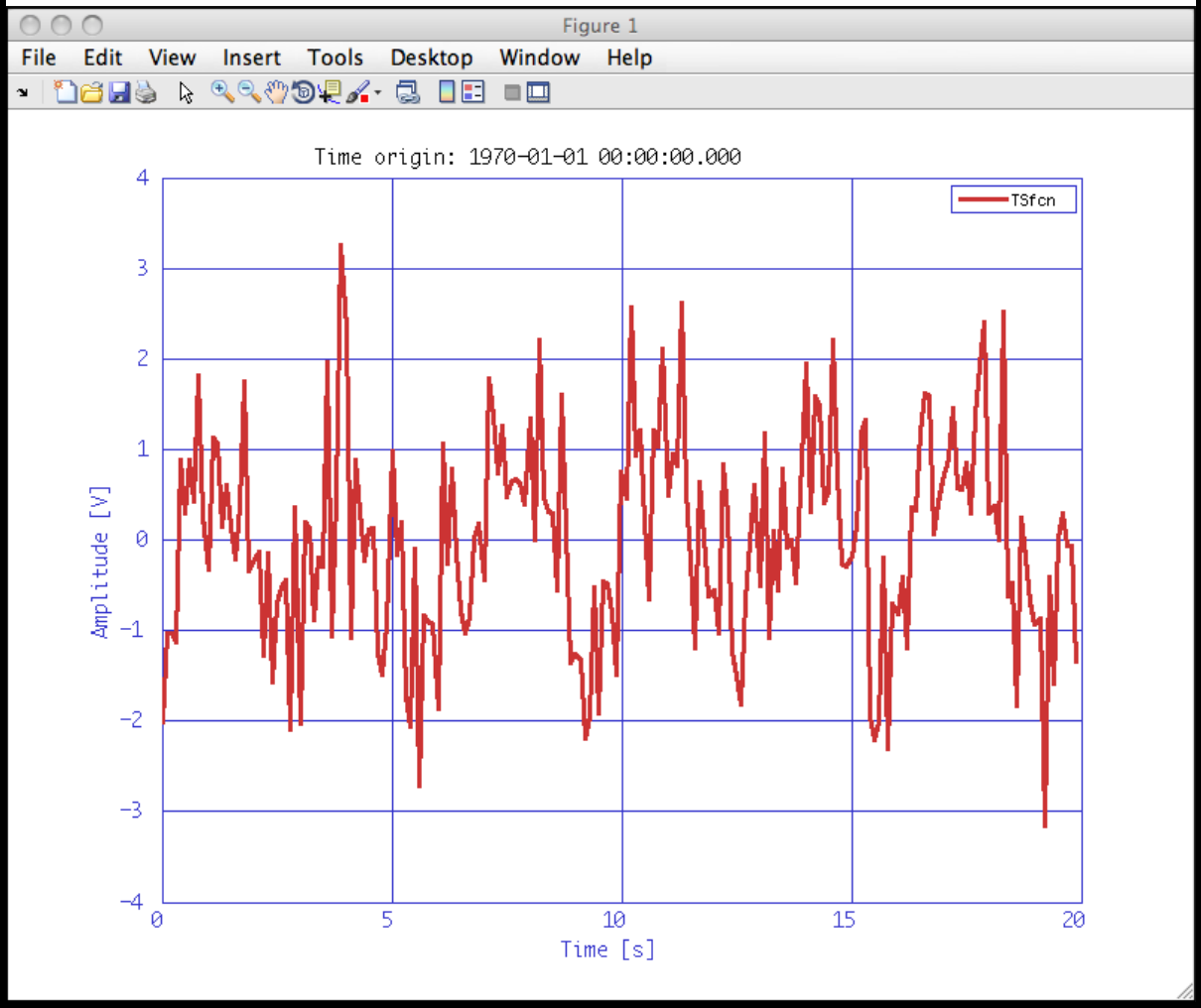
In addition, `ipplot` can be configured using a input `plist`. The following examples show some of the possible ways to use `ipplot`

```
>> a1 = ao(plist('tsfcn', 'sin(2*pi*0.3*t) + randn(size(t))', 'fs', 10, 'nsecs', 20))
----- ao 01: a1 -----
name:      None
data:      (0,0.887479404587351) (0.1, -0.449797172615395) (0.2,0.479022102390939) ...
----- tsdata 01 -----
          fs:      10
          x:      [200 1], double
          y:      [200 1], double
          dx:      [0 0], double
          dy:      [0 0], double
xunits:     [s]
yunits:     []
nsecs:      20
t0:         1970-01-01 00:00:00.000
-----
hist:       ao / ao / SId: fromTSfcn ... $-->$Id: ao ... S
mdlfile:    empty
description:
UUID:       67d31c2f-8ee5-42ba-a96b-cfb288ecc9ea
-----
>> a1.data
----- tsdata 01 -----
          fs:      10
          x:      [200 1], double
          y:      [200 1], double
          dx:      [0 0], double
          dy:      [0 0], double
xunits:     [s]
yunits:     []
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` are set to the defaults of seconds [s].

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

```
>> ipplot(a1)
```



We also see that the time–origin of the data (t_0 field of the `tsdata` class) is displayed as the plot title.

Collection objects

Collection objects serve as merely a wrapper for a cell-array of LTPDA User Objects. The point of this is to provide a way to put together user objects of different classes in to a collection which can be saved/loaded/submitted/retrieved, etc.

You can create a collection object like:

```
>> c = collection
---- collection 1 ----
name: none
num objs: 0
description:
UUID: a339a156-956b-4657-96ef-9ea4feef8101
-----
```

You can then add objects to the collection by doing:

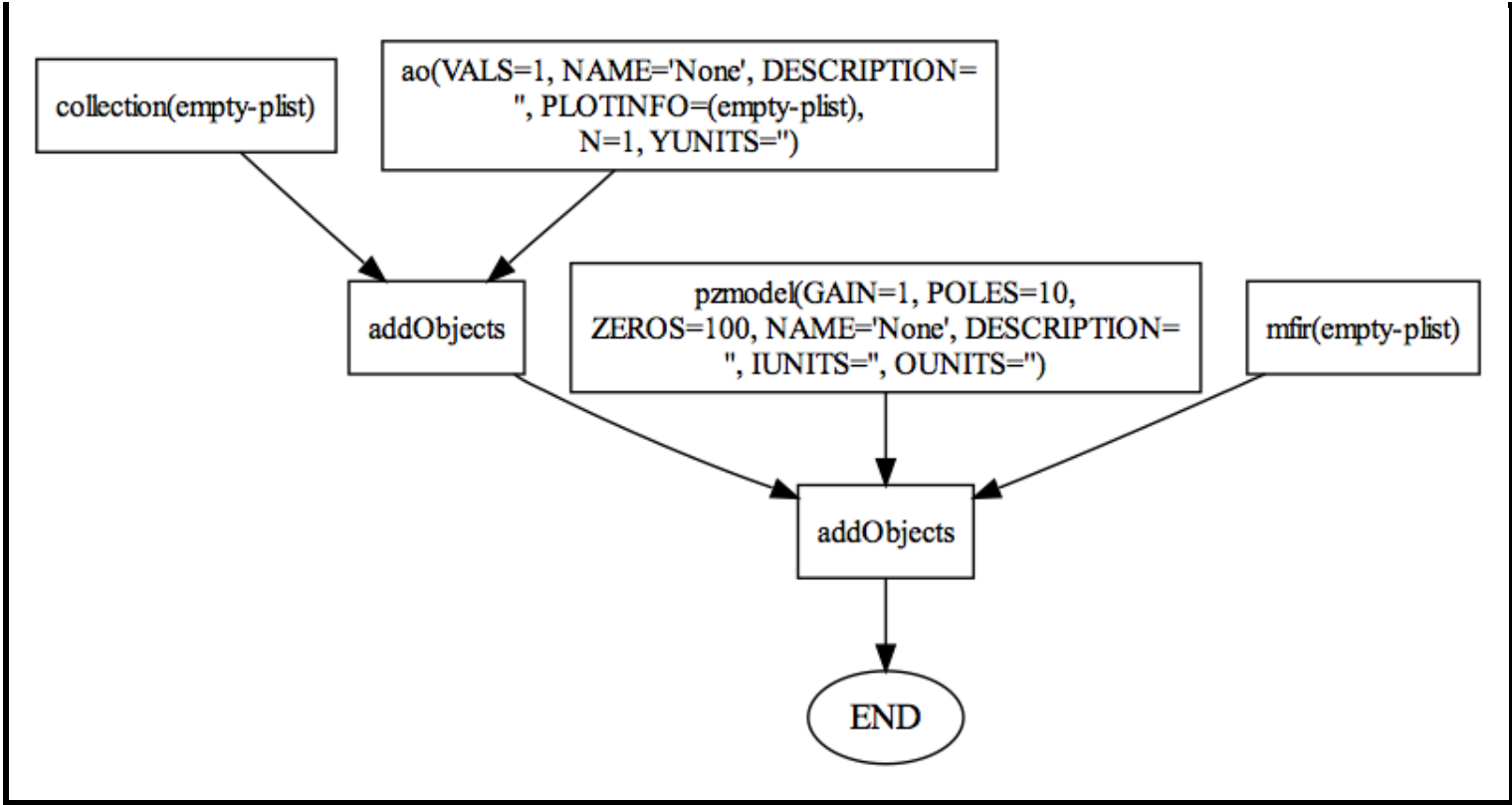
```
>> c.addObjects(ao(1))
M: running ao/ao
M: constructing from values
M: running collection/addObjects
M: running ao/char
---- collection 1 ----
name: none
num objs: 1
01: ao | None/cdata(Ndata=[1x1])
description:
UUID: c1f0200f-ad98-4fd6-beae-2dcdabb33515
-----
```

or

```
>> c.addObjects(pzmodel(1, 10, 100), mfir())
M: running collection/addObjects
M: running ao/char
---- collection 1 ----
name: none
num objs: 3
01: ao | None/cdata(Ndata=[1x1])
02: pzmodel | pzmodel(None)
03: mfir | none(fs=, ntaps=0.00, a=[])
description:
UUID: e7f02380-5650-4788-b819-f7fa7ee50a7d
-----
```

You can then extract objects from the collection using `getObjectAtIndex` or get an array of all objects of a particular class using `getObjectsOfClass`. Objects can be removed from the collection using `removeObjectAtIndex`.

Since a collection object is an LTPDA User Object, it has history tracking capabilities. That means that everytime you add or remove an object to/from the collection, a history step is added. For example, the history of our collection above looks like:



Units in LTPDA

Content needs written...

 Collection objects

Parameter Lists 

©LTP Team

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](#)
- [Creating lists of Parameters](#)
- [Working with 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 parameter object can be accessed as follows:

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


Creating lists of Parameters

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

- [Creating parameter lists from parameters](#)
- [Creating parameter lists directly](#)
- [Appending parameters to a parameter list](#)
- [Finding parameters in a parameter list](#)
- [Removing parameters from a parameter list](#)
- [Setting parameters in a parameter list](#)
- [Combining multiple parameter lists](#)

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:  (Hanning, alpha=0, psll=31.5, rov=50, nenbw=1.5, w3db=1.4382, flatness=-1.4236)
-----
description:
UUID:      8048fb7a-c32a-4679-9540-c1287be4afdc
-----
```

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'
-----
description:
UUID:      9cc7d1f1-6ca3-477b-a6cc-2629d52579e6
-----
```

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.

Simulation/modelling

Content needs written...

 Creating lists of Parameters	Built-in models of LTPDA 

©LTP Team

Built-in models of LTPDA

LTPDA provides a mechanism for storing typical object constructions as built-in models. Currently, this feature is only supported for the `ao` and `ssm` classes. The toolbox comes already supplied with some built-in models, though many of these are particular to LTP data analysis activities. It is, however, relatively straightforward to add your own built-in models.

- 1. [Built-in Analysis Object Models](#)
- 2. [Built-in Statespace Models](#)

◀ Simulation/modelling	Built-in Analysis Object Models ▶

©LTP Team

Built-in Analysis Object Models

- [Introduction](#)
- [Available models](#)
- [Adding new models](#)

Introduction

Built-in Analysis Object models provide a convenient way to add parametric constructors to the AO class. This is best explained with an example.

One of the supplied built-in models is called 'whitenoise'. To see how to build this model, do

```
>> help ao_model_whitenoise
```

All AO model files are called `ao_model_<model_name>`.

In this case, the help shows:

```
AO_MODEL_WHITENOISE constructs a known white-noise time-series
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DESCRIPTION: AO_MODEL_WHITENOISE constructs a known white-noise time-series.

CALL:      a = ao(plist('built-in', 'whitenoise'), pl);

INPUTS:    pl - a parameter list of additional parameters (see below)

PARAMETERS:
    'sigma' - standard deviation of the noise. [default: 1]
    'nsecs' - number of seconds [s] of data. [default: 1]
    'fs'     - sample rate [Hz] for the white noise. [default: 10]

VERSION:    $Id: builtin_models_ao_content.html,v 1.2 2009/02/18 13:14:23 hewitson Exp $

HISTORY:    29-10-08 M Hewitson
             Creation

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

To build this model, use the following constructor:

```
a = ao(plist('built-in', 'whitenoise', 'sigma', 2, 'nsecs', 100, 'fs', 10))
----- ao 01: WN -----
name:    WN
description:
data:    (0,0.260207192213954) (0.1,-1.01369469442225) (0.2,-2.1763634062959)
(0.3,1.00632778971068) (0.4,0.523897003913847) ...
----- tsdata 01 -----
fs:      10
x:        [1000 1], double
y:        [1000 1], double
xunits:   [s]
yunits:   [V]
nsecs:    100
t0:       1970-01-01 00:00:00.000
-----
hist:     ao / ao / $Id: builtin_models_ao_content.html,v 1.2 2009/02/18 13:14:23
hewitson Exp $-->$Id: builtin_models_ao_content.html,v 1.2 2009/02/18 13:14:23 hewitson Exp $
mfilename:
```

```
mdlfilename:
-----
```

The special thing about this model, is that it always generates noise from the same seed, thus providing a reproducible data series.

Available models

To see a list of the currently available built-in models, you can use the `ao` class static method, `getBuiltInModels`:

```
>> ao.getBuiltInModels
```

This returns a cell-array with two columns: the first columns contains the model names; the second column descriptions of the models.

You can also do

```
>> ao(plist('built-in', ''))
```

Adding new models

The available AO models are determined by looking through a set of directories for all M-files with names like `ao_model_<model_name>`. The directories to be searched are specified in the LTPDA Preferences under the "models" section.

To add a new model of your own, do the following steps:

1. Create a directory to store your AO model(s) in
2. Add this directory to the list of AO model directories in the preferences
3. Create a new M-file called `ao_model_myModel` (using an appropriate replacement for 'myModel')
4. Edit the contents of the new model file using the supplied model files as an example

It is recommended to use the above 'whitenoise' model as an example when building your own models. The source code of that model is heavily commented. In particular, pay attention to the blocks of code between the EDIT tags. For example, the following code extract from `ao_model_whitenoise.m` is concerned with retrieving and processing the additional parameters for this model:

```
%----- <EDIT>
% Here we deal with the additional parameters of the model. In this case,
% the sample rate of the resulting time-series, the number of seconds,
% and the amplitude of the white-noise data series. When writing your own
% model, you should include the necessary parameters here, together with
% any checks on the possible parameter values.
fs = find(pl, 'fs');
nsecs = find(pl, 'nsecs');
sigma = find(pl, 'sigma');

if isempty(fs)
    fs = 10;
end
if isempty(nsecs)
    nsecs = 1;
end
if isempty(sigma)
    sigma = 1;
end
%----- </EDIT>
```

To inspect the rest of the code for this model, just edit it:

```
>> edit ao_model_whitenoise
```

 Built-in models of LTPDA	Built-in Statespace Models 

©LTP Team

Built-in Statespace Models

The handling of built-in SSM models is done pretty much the same as for the AO models. To see a list of available models, use

```
>> ssm.getBuiltInModels
```

For creating your own SSM models, follow similar rules as in the AO case. A good example to follow is

```
>> edit ssm_model_standard_system_params
```


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 effectively implemented for the generation of [multichannel noise](#) with a given cross spectral density. Multichannel transfer functions are identified by an automatic fit procedure based on a modified version of the vector-fitting algorithm (see [Z-Domain Fit](#) for further details on the algorithm). Partial fraction expansion of multichannel transfer functions and the implementation of [filter](#) state initialization avoid the presence of unwanted '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.

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](#)
- [Description](#)
- [Inputs](#)
- [Outputs](#)
- [Usage](#)

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 following paper:

Franklin, Joel N.: *Numerical simulation of stationary and non-stationary gaussian random processes* , SIAM review, Volume { 7}, Issue 1, page 68--80, 1965.

The Document *Generation of Random time series with prescribed spectra* by Gerhard Heinzel (S2-AEI-TN-3034) corrects a mistake in the aforesaid paper and describes the practical implementation.

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 $\tau = 1/f_s$, where f_s is the sampling frequency.

- $y_0 = T_{init} * r$, on initialization
- $y_i = E * y_{i-1} + T_{prop} * r$, to propagate
- $x_i = a * y_i$, the sampled time series.

r is a vector of independent normal Gaussian random numbers T_{init} , E , T_{prop} which are real matrices and a which is a real vector are determined once by the algorithm.

Description

When an analysis object is constructed from a pole zero model Franklin's noise generator is called (compare [Creating AOs from pole zero models](#)).

Inputs

for the 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

Usage

The analysis object constructor [ao](#) calls the following four functions when the input is a pzmodel.

- ngconv
- ngsetup
- nginit
- ngprop

First a parameter list of the input parameters is to be done. For further information on this look at [Creating parameter lists from parameters](#).

Starting from a given pole/zero model

The parameter list should contain the number of seconds the resulting time series should have *nsecs* and the sampling frequency *fs*.

The constructor call should look like this:

```
f1 = 5;
f2 = 10;
f3 = 1;
gain = 1;
fs = 10; %sampling frequency
nsecs = 100; %number of seconds to be generated
p = [pz(f1) pz(f2)];
z = [pz(f3)];
pzm = pzmodel(gain, p, z);
a = ao(pzm, plist('nsecs', nsecs, 'fs', fs))
```

The output will be an analysis object *a* containing the time series with the spectrum described by the input pole-zero model.

◀ Generating model noise

Noise generation with given cross-spectral density ▶

©LTP Team

Noise generation with given cross-spectral density

[Multichannel Spectra](#)

Theoretical background on multichannel spectra.

[Multichannel noise generation](#)

Theoretical introduction to multichannel noise generation.

[Noisegen 1D](#)

Generation of one-dimensional noise with given spectral density.

[Noisegen 2D](#)

Generation of two-dimensional noise with given cross-spectral density.

The following sections gives an introduction to the generation of model noise with a given cross spectral density.

Theoretical background on multichannel spectra

We define the autocorrelation function (ACF) of a stationary multichannel process as:

$$\vec{\vec{R}}_{xx}[k] = \mathbb{E}\left(\vec{x}[n] \vec{x}^H[n+k]\right)$$

If the multichannel process is L dimensional then the kth element of the ACF is a LxL matrix:

$$\vec{\vec{R}}_{xx}[k] = \begin{pmatrix} r_{11}[k] & \cdots & r_{1L}[k] \\ \vdots & \ddots & \vdots \\ r_{L1}[k] & \cdots & r_{LL}[k] \end{pmatrix}$$

The ACF matrix is not hermitian but have the property that:

$$\vec{\vec{R}}_{xx}^H[k] = \vec{\vec{R}}_{xx}[-k]$$

The cross-spectral density matrix (CSD) is defined as the fourier transform of the ACF:

$$P_{xx}(f) = \begin{pmatrix} P_{11}(f) & \cdots & P_{1L}(f) \\ \vdots & \ddots & \vdots \\ P_{L1}(f) & \cdots & P_{LL}(f) \end{pmatrix}$$

the CSD matrix is hermitian.

A multichannel white noise process is defined as the process whose ACF satisfies:

$$\vec{\vec{R}}_{xx}[k] = \vec{\vec{\Sigma}} \delta[k]$$

therefore the cross-spectral matrix has constant terms as a function of the frequency:

$$\vec{P}_{xx}(f) = \vec{\Sigma}$$

The individual processes are each white noise processes with power spectral density (PSD) given by Σ_{ii} . The cross-correlation between the processes is zero except at the same time instant where they are correlated with a cross-correlation given by the off-diagonal elements of $\vec{\Sigma}$. A common assumption is $\vec{\Sigma} = \vec{I}$ (identity matrix) that is equivalent to assume the white processes having unitary variance and are completely uncorrelated being zero the off diagonal terms of the CSD matrix. Further details can be found in [1, 2].

Theoretical introduction to multichannel noise generation

The problem of multichannel noise generation with a given cross-spectrum is formulated in frequency domain as follows:

$$\vec{P}_{xx}(\Omega) = \vec{H}'(e^{j\Omega}) \vec{H} \vec{H}'^H(e^{j\Omega})$$

$\vec{H}'(z)$ is a multichannel digital filter that generating colored noise data with given cross-spectrum $\vec{P}_{xx}(\Omega)$ starting from a set of mutually independent unitary variance with noise processes.

After some mathematics it can be showed that the desired multichannel coloring filter can be written as:

$$\vec{H}'(e^{j\Omega}) = \vec{V}(\Omega) \vec{\Lambda}^{1/2}(\Omega)$$

where $\vec{V}(\Omega)$ and $\vec{\Lambda}(\Omega)$ are the eigenvectors and eigenvalues matrices of $\vec{P}_{xx}(\Omega)$ matrix.

Generation of one-dimensional noise with given spectral density

`noisegen1D` is a coloring tool allowing the generation of colored noise from white noise with a given spectrum. The function constructs a coloring filter through a fitting procedure to the model provided. If no model is provided an error is prompted. The colored noise provided has one-sided psd corresponding to the input model. The function needs a model for the one-sided power spectral density of the given process. Details on accepted parameters can be found on the [noisegen1D](#) documentation page.

1. The square root of the model for the power spectral density is fit in z-domain in order to determine a coloring filter.
2. Unstable poles are removed by an all-pass stabilization procedure.
3. White input data are filtered with the identified filter in order to be colored.

Generation of two-dimensional noise with given cross-spectral density

`noisegen2D` is a noise coloring tool allowing the generation two data series with the given cross-spectral density from two starting white and mutually uncorrelated data series. Coloring filters are constructed by a fitting procedure to a model for the cross-spectral density matrix provided. In order to work with `noisegen2D` you must provide a model (frequency series analysis

objects) for the cross-spectral density matrix of the process. Details on accepted parameters can be found on the [noise2D](#) documentation page.

1. Coloring filters frequency response is calculated by the eigendecomposition of the model cross-spectral matrix.
2. Calculated responses are fit in z-domain in order to identify corresponding autoregressive moving average filters.
3. Input time-series are filtered. The filtering process corresponds to:
$$o(1) = \text{Filt11}(a(1)) + \text{Filt12}(a(2))$$
$$o(2) = \text{Filt21}(a(1)) + \text{Filt22}(a(2))$$

References

1. S. M. Kay, Modern Spectral Estimation, Prentice-Hall, 1999
2. G. M. Jenkins and D. G. Watts, Spectral Analysis and Its Applications, Holden-Day 1968.

◀ Franklin noise-generator

Parameteric models ▶

©LTP Team

Parameteric models

Content needs written...

 Noise generation with given cross-spectral density

Introduction to parametric models in  LTPDA

©LTP Team

Introduction to parametric models in LTPDA

Content needs written...

◀ Parameteric models

Statespace models ▶

©LTP Team

Statespace models

Why use state space modeling?

State space modeling is efficient to simulate systems with large dimensionality, be it in terms of inputs, outputs, or pole/zeros. Adding nonlinearities to a model is also easier as in the frequency domain, however there is no such capability in the toolbox yet. Another reason to use them is to build complex parametric models, where intricate parameters make impossible the symbolic calculation of the transfer function coefficients – as a matter of fact the transfer function can be computed out of a determinant involving the A matrix, explaining the complexity of the calculation.

For tasks such as identification, state space modeling is a computationally rather heavy, especially if colored noise is involved in the process.

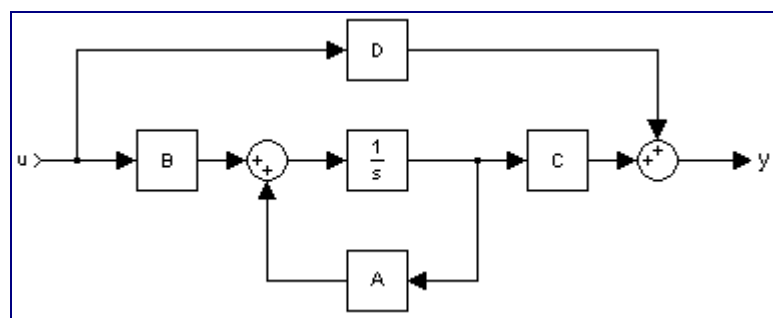
State space models can be converted into a matrix of transfer function in the s- or the z-domain. The functions in the toolbox that enable this are `ss2pzmodel`, `ss2miir`, `ss2rational`.

Generalities on State Space Modeling

In order to familiarize with state space modeling, this help page is a mere copy of the wiki page.

In control engineering, a **state space representation** is a mathematical model of a physical system as a set of input, output and state variables related by first-order [differential equations](#). To abstract from the number of inputs, outputs and states, the variables are expressed as vectors and the differential and algebraic equations are written in matrix form (the last one can be done when the [dynamical system](#) is linear and time invariant). The state space representation (also known as the "time-domain approach") provides a convenient and compact way to model and analyze systems with multiple inputs and outputs. With p inputs and q outputs, we would otherwise have to write down [Laplace transforms](#) to encode all the information about a system. Unlike the frequency domain approach, the use of the state space representation is not limited to systems with linear components and zero initial conditions. "State space" refers to the space whose axes are the state variables. The state of the system can be represented as a vector within that space.

State variables



Typical state space model

The internal [state variables](#) are the smallest possible subset of system variables that can represent the entire state of the system at any given time. State variables must be linearly

independent; a state variable cannot be a linear combination of other state variables. The minimum number of state variables required to represent a given system, n , is usually equal to the order of the system's defining differential equation. If the system is represented in transfer function form, the minimum number of state variables is equal to the order of the transfer function's denominator after it has been reduced to a proper fraction. It is important to understand that converting a state space realization to a transfer function form may lose some internal information about the system, and may provide a description of a system which is stable, when the state-space realization is unstable at certain points. In electric circuits, the number of state variables is often, though not always, the same as the number of energy storage elements in the circuit such as [capacitors](#) and [inductors](#).

Linear systems

The most general state-space representation of a linear system with p inputs, q outputs and n state variables is written in the following form:

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)$$

$$\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t)$$

where:

- $\mathbf{x}(\cdot)$ is called the "state vector", $\mathbf{x}(t) \in \mathbb{R}^n$;
- $\mathbf{y}(\cdot)$ is called the "output vector", $\mathbf{y}(t) \in \mathbb{R}^q$;
- $\mathbf{u}(\cdot)$ is called the "input (or control) vector", $\mathbf{u}(t) \in \mathbb{R}^p$;
- $\mathbf{A}(\cdot)$ is the "state matrix", $\dim[\mathbf{A}(\cdot)] = n \times n$,
- $\mathbf{B}(\cdot)$ is the "input matrix", $\dim[\mathbf{B}(\cdot)] = n \times p$,
- $\mathbf{C}(\cdot)$ is the "output matrix", $\dim[\mathbf{C}(\cdot)] = q \times n$,
- $\mathbf{D}(\cdot)$ is the "feedthrough (or feedforward) matrix" (in cases where the system model does not have a direct feedthrough, $\mathbf{D}(\cdot)$ is the zero matrix), $\dim[\mathbf{D}(\cdot)] = q \times p$,
- $\dot{\mathbf{x}}(t) := \frac{d}{dt}\mathbf{x}(t)$.

In this general formulation, all matrices are allowed to be time-variant (i.e., their elements can depend on time); however, in the common [LTI](#) case, matrices will be time invariant. The time variable t can be a "continuous" (e.g., $t \in \mathbb{R}$) or discrete (e.g., $t \in \mathbb{Z}$). In the latter case, the time variable is usually indicated as k . [Hybrid systems](#) allow for time domains that have both continuous and discrete parts. Depending on the assumptions taken, the state-space model representation can assume the following forms:

System type

State-space model

Continuous time-invariant $\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t) + \mathbf{D}\mathbf{u}(t) \end{aligned}$

Continuous time-variant $\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \\ \mathbf{y}(t) &= \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t) \end{aligned}$

Discrete time-invariant $\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k)$

$$\mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) + \mathbf{D}\mathbf{u}(k)$$

Discrete time-variant

$$\begin{aligned}\mathbf{x}(k+1) &= \mathbf{A}(k)\mathbf{x}(k) + \mathbf{B}(k)\mathbf{u}(k) \\ \mathbf{y}(k) &= \mathbf{C}(k)\mathbf{x}(k) + \mathbf{D}(k)\mathbf{u}(k)\end{aligned}$$

Laplace domain of continuous time-invariant

$$\begin{aligned}s\mathbf{X}(s) &= \mathbf{A}\mathbf{X}(s) + \mathbf{B}\mathbf{U}(s) \\ \mathbf{Y}(s) &= \mathbf{C}\mathbf{X}(s) + \mathbf{D}\mathbf{U}(s)\end{aligned}$$

Z-domain of discrete time-invariant

$$\begin{aligned}z\mathbf{X}(z) &= \mathbf{A}\mathbf{X}(z) + \mathbf{B}\mathbf{U}(z) \\ \mathbf{Y}(z) &= \mathbf{C}\mathbf{X}(z) + \mathbf{D}\mathbf{U}(z)\end{aligned}$$

Example: Continuous-time LTI case

Stability and natural response characteristics of a continuous-time [LTI system](#) (i.e., linear with matrices that are constant with respect to time) can be studied from the [eigenvalues](#) of the matrix **A**. The stability of a time-invariant state-space model can be determined by looking at the system's [transfer function](#) in factored form. It will then look something like this:

$$\mathbf{G}(s) = k \frac{(s - z_1)(s - z_2)(s - z_3)}{(s - p_1)(s - p_2)(s - p_3)(s - p_4)}$$

The denominator of the transfer function is equal to the [characteristic polynomial](#) found by taking the [determinant](#) of $sI - A$,

$$\lambda(s) = |sI - A|.$$

The roots of this polynomial (the [eigenvalues](#)) are the system transfer function's [poles](#) (i.e., the [singularities](#) where the transfer function's magnitude is unbounded). These poles can be used to analyze whether the system is [asymptotically stable](#) or [marginally stable](#). An alternative approach to determining stability, which does not involve calculating eigenvalues, is to analyze the system's [Lyapunov stability](#).

The zeros found in the numerator of $\mathbf{G}(s)$ can similarly be used to determine whether the system is [minimum phase](#).

The system may still be **input-output stable** (see [BIBO stable](#)) even though it is not internally stable. This may be the case if unstable poles are canceled out by zeros (i.e., if those singularities in the transfer function are [removable](#)).

Controllability

Main article: [Controllability](#)

Thus, state controllability condition implies that it is possible – by admissible inputs – to steer the states from any initial value to any final value within some finite time window. A continuous time-invariant linear state-space model is **controllable** [if and only if](#)

$$\text{rank} \begin{bmatrix} B & AB & A^2B & \dots & A^{n-1}B \end{bmatrix} = n$$

Observability

Main article: [Observability](#)

Observability is a measure for how well internal states of a system can be inferred by

knowledge of its external outputs. The observability and controllability of a system are mathematical duals (i.e., as controllability provides that an input is available that brings any initial state to any desired final state, observability provides that knowing an output trajectory provides enough information to predict the initial state of the system).

A continuous time-invariant linear state-space model is **observable** if and only if

$$\text{rank} \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix} = n$$

([Rank](#) is the number of linearly independent rows in a matrix.)

Transfer function

The "[transfer function](#)" of a continuous time-invariant linear state-space model can be derived in the following way:

First, taking the [Laplace transform](#) of

$$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$$

yields

$$s\mathbf{X}(s) = A\mathbf{X}(s) + B\mathbf{U}(s)$$

Next, we simplify for $\mathbf{X}(s)$, giving

$$(s\mathbf{I} - A)\mathbf{X}(s) = B\mathbf{U}(s) \\ \mathbf{X}(s) = (s\mathbf{I} - A)^{-1}B\mathbf{U}(s)$$

this is substituted for $\mathbf{X}(s)$ in the output equation

$$\mathbf{Y}(s) = C\mathbf{X}(s) + D\mathbf{U}(s), \text{ giving} \\ \mathbf{Y}(s) = C((s\mathbf{I} - A)^{-1}B\mathbf{U}(s)) + D\mathbf{U}(s)$$

Because the [transfer function](#) $\mathbf{G}(s)$ is defined as the ratio of the output to the input of a system, we take

$$\mathbf{G}(s) = \mathbf{Y}(s)/\mathbf{U}(s)$$

and substitute the previous expression for $\mathbf{Y}(s)$ with respect to $\mathbf{U}(s)$, giving

$$\mathbf{G}(s) = C(s\mathbf{I} - A)^{-1}B + D$$

Clearly $\mathbf{G}(s)$ must have q by p dimensionality, and thus has a total of qp elements. So for every input there are q transfer functions with one for each output. This is why the state-space representation can easily be the preferred choice for multiple-input, multiple-output (MIMO) systems.

Canonical realizations

Any given transfer function which is [strictly proper](#) can easily be transferred into state-space

by the following approach (this example is for a 4-dimensional, single-input, single-output system)):

Given a transfer function, expand it to reveal all coefficients in both the numerator and denominator. This should result in the following form:

$$\mathbf{G}(s) = \frac{n_1 s^3 + n_2 s^2 + n_3 s + n_4}{s^4 + d_1 s^3 + d_2 s^2 + d_3 s + d_4}.$$

The coefficients can now be inserted directly into the state-space model by the following approach:

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} -d_1 & -d_2 & -d_3 & -d_4 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \mathbf{u}(t)$$

$$\mathbf{y}(t) = [n_1 \quad n_2 \quad n_3 \quad n_4] \mathbf{x}(t).$$

This state-space realization is called **controllable canonical form** because the resulting model is guaranteed to be controllable (i.e., because the control enters a chain of integrators, it has the ability to move every state).

The transfer function coefficients can also be used to construct another type of canonical form

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} -d_1 & 1 & 0 & 0 \\ -d_2 & 0 & 1 & 0 \\ -d_3 & 0 & 0 & 1 \\ -d_4 & 0 & 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_4 \end{bmatrix} \mathbf{u}(t)$$

$$\mathbf{y}(t) = [1 \quad 0 \quad 0 \quad 0] \mathbf{x}(t).$$

This state-space realization is called **observable canonical form** because the resulting model is guaranteed to be observable (i.e., because the output exits from a chain of integrators, every state has an effect on the output).

Proper transfer functions

Transfer functions which are only [proper](#) (and not [strictly proper](#)) can also be realised quite easily. The trick here is to separate the transfer function into two parts: a strictly proper part and a constant.

$$\mathbf{G}(s) = \mathbf{G}_{SP}(s) + \mathbf{G}(\infty)$$

The strictly proper transfer function can then be transformed into a canonical state space realization using techniques shown above. The state space realization of the constant is trivially $\mathbf{y}(t) = \mathbf{G}(\infty)\mathbf{u}(t)$. Together we then get a state space realization with matrices A, B and C determined by the strictly proper part, and matrix D determined by the constant.

Here is an example to clear things up a bit:

$$\mathbf{G}(s) = \frac{s^2 + 3s + 3}{s^2 + 2s + 1} = \frac{s + 2}{s^2 + 2s + 1} + 1$$

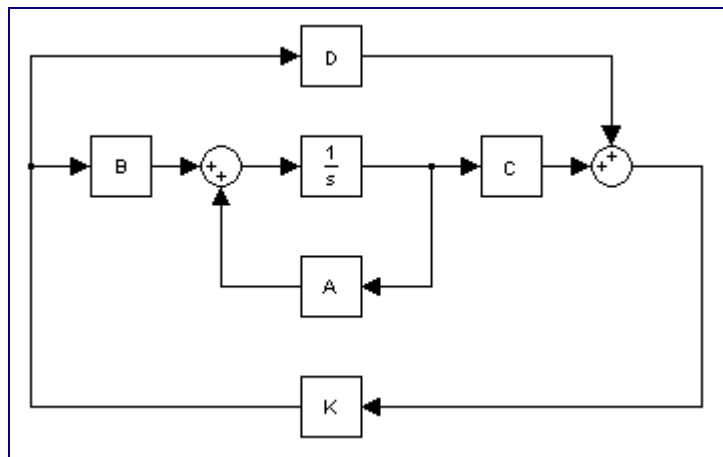
which yields the following controllable realization

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} -2 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathbf{u}(t)$$

$$\mathbf{y}(t) = \begin{bmatrix} 1 & 2 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 1 \end{bmatrix} \mathbf{u}(t)$$

Notice how the output also depends directly on the input. This is due to the $\mathbf{G}(\infty)$ constant in the transfer function.

Feedback



Typical state space model with feedback

A common method for feedback is to multiply the output by a matrix K and setting this as the input to the system: $\mathbf{u}(t) = K\mathbf{y}(t)$. Since the values of K are unrestricted the values can easily be negated for [negative feedback](#). The presence of a negative sign (the common notation) is merely a notational one and its absence has no impact on the end results.

$$\begin{aligned} \dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + B\mathbf{u}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) + D\mathbf{u}(t) \end{aligned}$$

becomes

$$\begin{aligned} \dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + BK\mathbf{y}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) + DK\mathbf{y}(t) \end{aligned}$$

solving the output equation for $\mathbf{y}(t)$ and substituting in the state equation results in

$$\begin{aligned} \dot{\mathbf{x}}(t) &= (A + BK(I - DK)^{-1}C) \mathbf{x}(t) \\ \mathbf{y}(t) &= (I - DK)^{-1}C\mathbf{x}(t) \end{aligned}$$

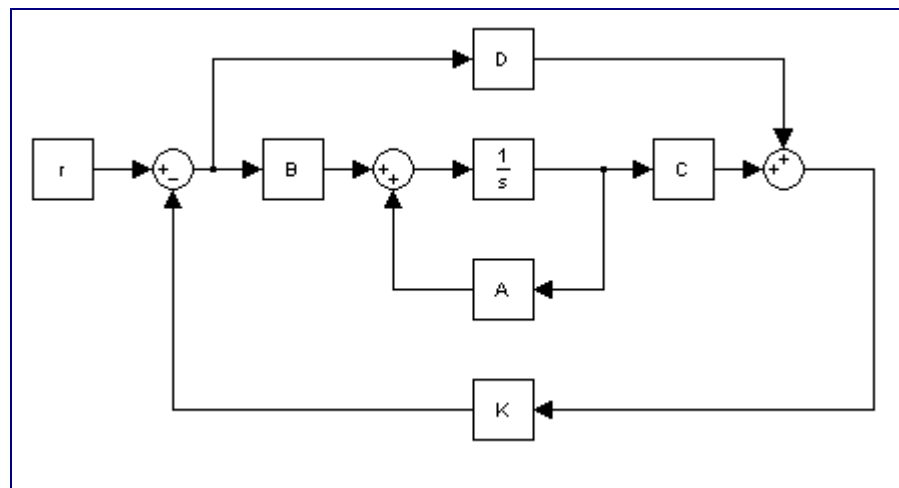
The advantage of this is that the [eigenvalues](#) of A can be controlled by setting K appropriately through eigendecomposition of $(A + BK(I - DK)^{-1}C)$. This assumes that the open-loop system is [controllable](#) or that the unstable eigenvalues of A can be made stable through appropriate choice of K .

One fairly common simplification to this system is removing D and setting C to identity, which reduces the equations to

$$\begin{aligned}\dot{\mathbf{x}}(t) &= (A + BK) \mathbf{x}(t) \\ \mathbf{y}(t) &= \mathbf{x}(t)\end{aligned}$$

This reduces the necessary eigendecomposition to just $A + BK$.

Feedback with setpoint (reference) input



Output feedback with set point

In addition to feedback, an input, $r(t)$, can be added such that $\mathbf{u}(t) = -K\mathbf{y}(t) + \mathbf{r}(t)$.

$$\begin{aligned}\dot{\mathbf{x}}(t) &= A\mathbf{x}(t) + B\mathbf{u}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) + D\mathbf{u}(t)\end{aligned}$$

becomes

$$\begin{aligned}\dot{\mathbf{x}}(t) &= A\mathbf{x}(t) - BK\mathbf{y}(t) + B\mathbf{r}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t) - DK\mathbf{y}(t) + D\mathbf{r}(t)\end{aligned}$$

solving the output equation for $\mathbf{y}(t)$ and substituting in the state equation results in

$$\begin{aligned}\dot{\mathbf{x}}(t) &= (A - BK(I + DK)^{-1}C) \mathbf{x}(t) + B(I - K(I + DK)^{-1}D) \mathbf{r}(t) \\ \mathbf{y}(t) &= (I + DK)^{-1}C\mathbf{x}(t) + (I + DK)^{-1}D\mathbf{r}(t)\end{aligned}$$

One fairly common simplification to this system is removing D , which reduces the equations to

$$\begin{aligned}\dot{\mathbf{x}}(t) &= (A - BKC) \mathbf{x}(t) + B\mathbf{r}(t) \\ \mathbf{y}(t) &= C\mathbf{x}(t)\end{aligned}$$

Moving object example

A classical linear system is that of one-dimensional movement of an object. The [Newton's laws of motion](http://www.lisa.aei-hannover.de/ltpda/usermanual/ug/ssm.html) for an object moving horizontally on a plane and attached to a wall with a spring

$$m\ddot{y}(t) = u(t) - k_1\dot{y}(t) - k_2y(t)$$

where

- $y(t)$ is position; $\dot{y}(t)$ is velocity; $\ddot{y}(t)$ is acceleration

- $u(t)$ is an applied force
- k_1 is the viscous friction coefficient
- k_2 is the spring constant
- m is the mass of the object

The state equation would then become

$$\begin{bmatrix} \dot{\mathbf{x}}_1(t) \\ \dot{\mathbf{x}}_2(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix} \begin{bmatrix} \mathbf{x}_1(t) \\ \mathbf{x}_2(t) \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \mathbf{u}(t)$$

$$\mathbf{y}(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1(t) \\ \mathbf{x}_2(t) \end{bmatrix}$$

where

- $x_1(t)$ represents the position of the object
- $x_2(t) = \dot{x}_1(t)$ is the velocity of the object
- $\dot{x}_2(t) = \ddot{x}_1(t)$ is the acceleration of the object
- the output $\mathbf{y}(t)$ is the position of the object

The [controllability](#) test is then

$$\begin{bmatrix} B & AB \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} & \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{m} \\ \frac{1}{m} & \frac{k_1}{m^2} \end{bmatrix}$$

which has full rank for all k_1 and m .

The [observability](#) test is then

$$\begin{bmatrix} C \\ CA \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -\frac{k_2}{m} & -\frac{k_1}{m} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

which also has full rank. Therefore, this system is both controllable and observable.

Nonlinear systems

The more general form of a state space model can be written as two functions.

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{f}(t, \mathbf{x}(t), \mathbf{u}(t)) \\ \mathbf{y}(t) &= \mathbf{h}(t, \mathbf{x}(t), \mathbf{u}(t)) \end{aligned}$$

The first is the state equation and the latter is the output equation. If the function $\mathbf{f}(\cdot, \cdot, \cdot)$ is a linear combination of states and inputs then the equations can be written in matrix notation like above. The $\mathbf{u}(t)$ argument to the functions can be dropped if the system is unforced (i.e., it has no inputs).

Pendulum example

A classic nonlinear system is a simple unforced [pendulum](#)

$$ml\ddot{\theta}(t) = -mg \sin \theta(t) - kl\dot{\theta}(t)$$

where

- $\theta(t)$ is the angle of the pendulum with respect to the direction of gravity
- m is the mass of the pendulum (pendulum rod's mass is assumed to be zero)
- g is the gravitational acceleration
- k is coefficient of friction at the pivot point
- l is the radius of the pendulum (to the center of gravity of the mass m)

The state equations are then

$$\begin{aligned}\dot{x}_1(t) &= x_2(t) \\ \dot{x}_2(t) &= -\frac{g}{l} \sin x_1(t) - \frac{k}{m} x_2(t)\end{aligned}$$

where

- $x_1(t) := \theta(t)$ is the angle of the pendulum
- $x_2(t) := \dot{x}_1(t)$ is the rotational velocity of the pendulum
- $\ddot{x}_1 = \ddot{x}_2$ is the rotational acceleration of the pendulum

Instead, the state equation can be written in the general form

$$\dot{x}(t) = \begin{pmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \end{pmatrix} = \mathbf{f}(t, x(t)) = \begin{pmatrix} x_2(t) \\ -\frac{g}{l} \sin x_1(t) - \frac{k}{m} x_2(t) \end{pmatrix}.$$

The [equilibrium/stationary points](#) of a system are when $\dot{x} = 0$ and so the equilibrium points of a pendulum are those that satisfy

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} n\pi \\ 0 \end{pmatrix}$$

for integers n .

Introduction to Statespace Models with LTPDA

What is a ssm object?

The ssm class is a class to use “state space models” for simulation, identification and modeling in the toolbox. A state space model is a mathematical object constituted of

- A time model represented by the field “timestep”, either discrete or continuous (“timestep” is then 0)
- A linear differential equation, which turns into a difference equation if the model is time discrete. This equation is written using a state space x in the form $x' = Ax + Bu$ where A is the state transition matrix and B the state input matrix, u being the exogenous input signal. In this setup, x and u are time series.
- An observation equation which allows for linear observation of the input and the state : $y = Cx + Du$. The variable y is the output of the system.

Input and output variables (lines in u and y) may be named and assigned a unit. However for the state x there is more of a choice since it is free of choice via a basis change. The most usual choice is that x contains the same units as y and its derivatives, or the same units as u and its integrals. In general the size of the state space is equal to the number of poles in the system.

In the LTPDA toolbox, lines in u, y, x are grouped in blocks (input blocks, state blocks, output blocks) to ease the understanding of a large system. This means the matrices A, B, C , and D are also split into blocks of the corresponding size. The diagonal blocks of A may represent the dynamics of a LTP subsystem (the controller, the propulsion ...) or the coupling between two systems (from the propulsion to the equations of motion)... In addition the dynamic equation also contains an inertia matrix M so that the parametric equation is made simple. The equation becomes $M x' = Ax + Bu$. If not user set M is supposed to be the identity.

Below we build a standard system to show the contents of the object.

```
>> system = ssm(plist('built-in', 'standard_system_params', 'withparams', 'ALL'))
M: running ssm/ssm
M: running ssmFromBuiltinSystem
M: looking for models in C:\Documents and Settings\Adrien.IFR-NB01\My
Documents\MATLAB2008\LTPDA_SSM_MODELS\ltp_ssm_models
M: looking for models in C:\Users\Adrien.IFR-NB01\My
Documents\MATLAB2008\ltpda_toolbox\ltpda\classes\@ssm\...\m\built_in_models
M: running buildParamList
M: running ssm/ssm
M: running ssm/ssm
M: running fromStruct
M: running ssm/ssm
M: running validate
M: running validate
M: running display
----- ssm/1 -----
    amats: { [2x2] } [1x1]
    mmats: { [2x2] } [1x1]
    bmats: { [2x1] [2x2] } [1x2]
    cmats: { [1x2] } [1x1]
    dmats: { [1x1] [1x2] } [1x2]
timestep: 0
inputs: [1x2 ssmblock]
    1 : U | Fu [kg m s^(-2)]
    2 : N | Fn [kg m s^(-2)], On [m]
states: [1x1 ssmblock]
```

```

1 : standard test system | x [m], xdot [m s^(-1)]
outputs: [1x1 ssmblock]
1 : Y | y [m]
params: (W=0.2, C=0.5, C1=0, C2=0, B=1, D=0) [1x1 plist]
version: $Id: ssm_introduction_content.html,v 1.3 2009/08/28 14:20:07 adrien Exp $-->$Id:
ssm_introduction_content.html,v 1.3 2009/08/28 14:20:07 adrien Exp $
Ninputs: 2
inputsizes: [1 2]
Noutputs: 1
outputsizes: 1
Nstates: 1
statesizes: 2
Nparams: 6
isnumerical: false
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: standard_system_params
description: standard spring-mass-dashpot test system
mdlfile:
UUID: 8eafd65e-9052-4800-a694-9482e2bd7b70
-----

```

The fields “amats”, “mmats”, “bmats”, “cmats”, “dmats” contain the matrices of the differential and observation equation of a mass spring system with dampening.

The systems inputs/states/outputs are listed by blocks. There are two input blocks, which correspond to the commanded signal (U) and the noise (N). There is one state block and one output block. Each input has its name displayed followed with individual input variable names and units. Note that the empty ssm object does not have an input block, a state block and an output block of size 0, it has no such blocks, and the size fields are empty.

The “params” field is the 1x1 default plist with no parameter. This is the case when the system is fully numerical, parameters are deleted from the field “params” as they get substituted .

Inputs, states and output sizes are summed up in the fields N* and *sizes.

◀ Statespace models

Building Statespace models ▶

©LTP Team

Building Statespace models

How to build a ssm object.

- The empty constructor creates an empty statespace model

```
>> s = ssm()
```

- Models can be built out of built-in models (mfiles stored in a folder), out of a description with a plist, from a repository or a xml file, but also out of a pzmodel, a rational or a miir model. Conversion out of a parFrac object is not implemented yet.

```
>> s = ssm(<pzmodel>)
>> s = ssm(<miir>)
>> s = ssm(<rational>)
```

- This creates a new statespace model by loading the object from disk, either out of an xml file or a .mat file. The latter is not recommended as Matlab data format poses some new retro-compatibility issues at each new release.

```
>> s = ssm('a1.xml')
>> s = ssm('a1.mat')
```

- For internal use (see built-in models), a structure constructor is available. This is constructor should not be used – except inside a built-in model – as it does not increment history, making it impossible to rebuild the object.

```
>> s = ssm(<struct>)
```

- There are dedicated help pages on how to build a model out of a built-in object or a plist description.

```
>> system = ssm(plist('built-in', '<model name>'))
>> system = ssm(plist('built-in', <model number>))
>> system = ssm(plist('amats', <a matrices> ... ))
```


Building from scratch

The models can be built from a plist constructor describing each field. It is possible to give incomplete information on the model and let the constructor auto-complete some fields.

It is still rather lengthy to build a ssm object, and it is advisable to write built-in models to limit the time spent on modeling.

Incomplete description without parameters (params field) and input/state/outputs names

The most retrained set of inputs is

```
sys = ssm(plist( 'amats',amats, 'bmats',bmats, 'cmats',cmats, 'dmats',dmats,
'timestep',timestep, 'name',name )
```

Then a shortcut for the scripts is (note that the input order must be respected)

```
sys = ssm( amats, bmats, cmats, dmats, timestep, name )
```

example :

```
>> name = 'sys';
>> timestep = 0;
>> amats = cell(3,3);
>> bmats = cell(3,3);
>> cmats = cell(3,3);
>> dmats = cell(3,3);
>> amats{1,1} = -(sym('OMEGA'));
>> amats{2,2} = -2;
>> amats{3,3} = [0 1 ; -0.05 -0.01];
>> amats{3,1} = [-1;-3];
>> bmats{1,1} = 1;
>> bmats{2,2} = 2;
>> bmats{3,3} = 3*eye(2);
>> cmats{1,1} = 1;
>> cmats{2,2} = 1;
>> cmats{3,3} = eye(2);
>> dmats{1,3} = [6 6];
>> dmats{2,1} = 6;
>> dmats{3,2} = [6;6];
>> sys = ssm(plist( ...
'amats',amats,'bmats',bmats,'cmats',cmats,'dmats',dmats, ...
'timestep',timestep,'name',name))
----- ssm/1 -----
amats: { [1x1] [] []
          [] [1x1] []
          [2x1] [] [2x2] } [3x3]
mmats: { [1x1] [] []
          [] [1x1] []
          [1x1] [] [2x2] } [3x3]
bmats: { [1x1] [] []
          [] [1x1] []
          [1x1] [] [2x2] } [3x3]
cmats: { [1x1] [] []
          [] [1x1] []
          [1x1] [] [2x2] } [3x3]
dmats: { [1x1] [] [2x2]
          [1x1] [1x2]
          [1x1] [2x1] [] } [3x3]
timestep: 0
inputs: [1x3 ssmblock]
        1 : input 1 | input 1 > 1 []
        2 : input 2 | input 2 > 1 []
```



```

3 : input 3 | input 3 > 1 [], input 3 > 2 []
states: [1x3 ssmblock]
1 : state 1 | state 1 > 1 []
2 : state 2 | state 2 > 1 []
3 : state 3 | state 3 > 1 [], state 3 > 2 []
outputs: [1x3 ssmblock]
1 : output 1 | output 1 > 1 []
2 : output 2 | output 2 > 1 []
3 : output 3 | output 3 > 1 [], output 3 > 2 []
params: (empty-plist) [1x1 plist]
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
Ninputs: 3
inputsizes: [1 1 2]
Noutputs: 3
outputsizes: [1 1 2]
Nstates: 3
statesizes: [1 1 2]
Nparams: 0
isnumerical: false
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: sys
description:
mdlfile:
UUID: 61f33fcc-f06a-4d71-944f-3ea094c80458
-----

```

Then the field “params” must be user set using the syntax “sys.setParams(<parameter plist>)”. Otherwise the toolbox will simply assume there are no parameters in the system and the matrices should be exclusively numerical in this case..

The content of „params“ is a plist with parameters whose:

- KEY is the name of the parameter in the matrices
- VALUE is a property which must be set before proceeding numerical substitutions
- MIN, MAX, and SIGMA which are useful for an optimizer if system identification is proceeded.

```

>>sys.setParams(plist({'OMEGA', 'system frequency'}, 2))
----- ssm/1 -----
amats: { [1x1] [] []
          [1x1] [1x1]
          [2x1] [1x1] [2x2] } [3x3]
mmats: { [1x1] [1x1] [2x2] } [3x3]
bmats: { [1x1] [1x1] [2x2] } [3x3]
cmats: { [1x1] [1x1] [2x2] } [3x3]
dmats: { [1x1] [1x1] [2x2] } [3x3]
          [1x1] [1x1] [1x2]
          [1x1] [2x1] [] } [3x3]
timestep: 0
inputs: [1x3 ssmblock]
1 : input 1 | input 1 > 1 []
2 : input 2 | input 2 > 1 []
3 : input 3 | input 3 > 1 [], input 3 > 2 []
states: [1x3 ssmblock]
1 : state 1 | state 1 > 1 []
2 : state 2 | state 2 > 1 []
3 : state 3 | state 3 > 1 [], state 3 > 2 []
outputs: [1x3 ssmblock]
1 : output 1 | output 1 > 1 []
2 : output 2 | output 2 > 1 []
3 : output 3 | output 3 > 1 [], output 3 > 2 []
params: (OMEGA=2) [1x1 plist]
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
Ninputs: 3
inputsizes: [1 1 2]
Noutputs: 3
outputsizes: [1 1 2]
Nstates: 3
statesizes: [1 1 2]
Nparams: 1

```

```

isnumerical: false
  hist: ssm.hist [1x1 history]
  procinfo: (empty-plist) [1x1 plist]
  plotinfo: (empty-plist) [1x1 plist]
  name: sys
description:
  mdlfile:
  UUID: 7bccc440-fcc3-4d71-bf9c-057bbc08d318
-----

```

The method “setParams” should not be confused with “setParameters” which only allows to modify the property VALUE in the “params” field.

Input blocks and input variables will be automatically named after their index number in this case, and the description field will be empty.

Incomplete description without input/state/outputs names

In case the user has a plist describing the parameters, he may use the following either of the syntaxes:

```

>> sys = ssm(plist( 'amats',amats, 'bmats',bmats, 'cmats',cmats, 'dmats',dmats, 'timestep',timestep,
'name',name, 'params',params ));
>> sys = ssm( amats, bmats, cmats, dmats, timestep, name, params )

>> name = 'sys';
>> timestep = 0;
>> params = plist({'OMEGA', 'system frequency'}, 2);
>> amats = cell(3,3);
>> bmats = cell(3,3);
>> cmats = cell(3,3);
>> dmats = cell(3,3);
>> amats{1,1} = -(sym('OMEGA'));
>> amats{2,2} = -2;
>> amats{3,3} = [0 1 ; -0.05 -0.01];
>> amats{3,1} = [-1;-3];
>> bmats{1,1} = 1;
>> bmats{2,2} = 2;
>> bmats{3,3} = 3*eye(2);
>> cmats{1,1} = 1;
>> cmats{2,2} = 1;
>> cmats{3,3} = eye(2);
>> dmats{1,3} = [6 6];
>> dmats{2,1} = 6;
>> dmats{3,2} = [6;6];
>> sys = ssm(plist( ...
'amats',amats, 'bmats',bmats, 'cmats',cmats, 'dmats',dmats, ...
'timestep',timestep, 'name',name, 'params',params ));

```

Then the inputs, states and outputs name fields are still automatically set.

They can be modified by using the “setBlock*” and “setPort*” setter methods. Blocks are vectors of ports which will be matched when assembling two system. Setting their name correctly is important as the names are used to identify automatically which output will be fed to which input. Ports are independent variables and have properties like the name, the description and the unit which the user may want to set to keep track of the systems data. Note that there is no unit check implemented when assembling two systems.

The setter functions are respectively “setBlockNames”, “setBlockDescription”, “setPortNames”, “setPortDescriptions”, “setPortUnits”.

```

>> sys.setBlockNames(plist('field', 'inputs', 'blocks', [1 2 3], 'names',
{'myinput1','myinput2','myinput3'}));
>> sys.setBlockDescriptions(plist('field', 'inputs', 'blocks', {'myinput1'},
'descriptions',{'myinput1 description'}));
>> sys.setPortNames(plist('field', 'inputs', 'block', {'myinput3'}, 'ports', [1 2],
'names',{'my port 3-1','my port 3-2'}));
>> sys.setPortDescriptions(plist('field', 'inputs', 'block', {'myinput3'}, 'ports', [1 2],
'descriptions',{'first description','second description'}));
>> sys.setPortUnits(plist('field', 'inputs', 'block', {'myinput3'}, 'ports', [1 2], 'units',
[unit('kg m^-2') unit('m')] ));

```

```
>> sys.inputs

----- ssmblock/1 -----
name: myinput1
ports: input 1 > 1 [] [1x1 ssmpport]
description: myinput1 description
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmblock/2 -----
name: myinput2
ports: input 2 > 1 [] [1x1 ssmpport]
description:
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmblock/3 -----
name: myinput3
ports: my port 3-1 [kg m^(-2)], my port 3-2 [m] [1x2 ssmpport]
description:
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
-----

>>sys.inputs(3).ports

----- ssmpport/1 -----
name: my port 3-1
units: [kg m^(-2)] [1x1 unit]
description: first description
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmpport/2 -----
name: my port 3-2
units: [m] [1x1 unit]
description: second description
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
-----
```

So far there is no special display function for ssmblocks so the description of the ports is not displayed. This may change in the future

Description with input/state/outputs names

```
Then a more extensive set of inputs is :
>>sys = ssm(plist('amats',amats,'bmats',bmats,'cmats',cmats,'dmats',dmats,
'timestep',timestep,'name',name,'params',params,'statenames',statenames,
'inputnames',inputnames,'outputnames',outputnames));
>> sys = ssm( amats, bmats, cmats, dmats, timestep, name, params, statenames, inputnames,
outputnames );
```

Still here, the port names are set automatically without any description or unit.

Complete description

example :

```
>> name = 'sys';
>> statenames = {'ss1' 'ss2' 'ss3'};
>> inputnames = {'input1' 'input2' 'input3'};
>> outputnames = {'output1' 'output2' 'output3'};
>> timestep = 0;
>> params = plist({'OMEGA', 'system frequency'}, 2);
>> amats = cell(3,3);
>> bmats = cell(3,3);
>> cmats = cell(3,3);
>> dmats = cell(3,3);
>> amats{1,1} = -(sym('OMEGA'));
>> amats{2,2} = -2;
>> amats{3,3} = [0 1 -0.05 -0.01];
>> amats{3,1} = [-1;-3];
>> bmats{1,1} = 1;
>> bmats{2,2} = 2;
>> bmats{3,3} = 3*eye(2);
>> cmats{1,1} = 1;
>> cmats{2,2} = 1;
>> cmats{3,3} = eye(2);
>> dmats{1,3} = [6 6];
>> dmats{2,1} = 6;
>> dmats{3,2} = [6;6];
>> sys = ssm(plist( ...
```

```
'amats',amats, 'bmats',bmats, 'cmats',cmats, 'dmats',dmats, ...
'timestep',timestep, 'name',name, 'params',params, ...
'statenames',statenames, 'inputnames',inputnames, 'outputnames',outputnames ));
>> sys
----- ssm/1 -----
amats: { [1x1] [] []
          [] [1x1] []
          [2x1] [] [2x2] } [3x3]
mmats: { [1x1] [] []
          [] [1x1] []
          [1x1] [] [2x2] } [3x3]
bmats: { [1x1] [] []
          [] [1x1] []
          [1x1] [] [2x2] } [3x3]
cmats: { [1x1] [] []
          [] [1x1] []
          [1x1] [] [2x2] } [3x3]
dmats: { [1x1] [] [2x2] } [3x3]
          [1x1] [] [1x2]
          [] [2x1] [] } [3x3]
timestep: 0
inputs: [1x3 ssmblock]
1 : input1 | input1 > 1 []
2 : input2 | input2 > 1 []
3 : input3 | input3 > 1 [], input3 > 2 []
states: [1x3 ssmblock]
1 : ss1 | ss1 > 1 []
2 : ss2 | ss2 > 1 []
3 : ss3 | ss3 > 1 [], ss3 > 2 []
outputs: [1x3 ssmblock]
1 : output1 | output1 > 1 []
2 : output2 | output2 > 1 []
3 : output3 | output3 > 1 [], output3 > 2 []
params: (OMEGA=2) [1x1 plist]
version: $Id: ssm_build_description_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
Ninputs: 3
inputsizes: [1 1 2]
Noutputs: 3
outputsizes: [1 1 2]
Nstates: 3
statesizes: [1 1 2]
Nparams: 1
isnumerical: false
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: sys
description:
mdlfile: UUID: b30c0d1d-9d09-4fd5-8dc1-8185f8b1c165
```

This constructor still does not set ports properties (except automatic name) nor the block description.

These may be set by the user later on using the setter functions above.

◀ Building Statespace models

Building from built-in models ▶

©LTP Team

Building from built-in models

Built-in models enable to build fast predefined models to use later in simulations.

Using existing built-in models

The constructor can be called using either of the following syntax:

```
>> system = ssm(plist('built-in', '<model name>'))
>> system = ssm(plist('built-in', '<model number>'))
```

If the user does not know the model name/number, then he may use the following call to see the list of all models :

```
>> system = ssm(plist('built-in', ''))
```

This way the numbers and names are displayed in an error message.

Note that the numbers may change if a new model is added, and using the name is highly recommended.

The LTPDA includes one mass-spring model which can be generated :

```
>> system = ssm(plist('built-in', 'standard_system_params'))
M: running ssm/ssm
M: running ssmFromBuiltinSystem
M: looking for models in C:\Documents and Settings\Adrien.IFR-NB01\My
Documents\MATLAB2008\LTPDA_SSM_MODELS\ltp_ssm_models
M: looking for models in C:\Users\Adrien.IFR-NB01\My
Documents\MATLAB2008\ltpda_toolbox\ltpda\classes\@ssm\...\m\built_in_models
M: running buildParamList
M: running ssm/ssm
M: running ssm/ssm
M: running fromStruct
M: running ssm/ssm
M: running validate
M: running validate
M: running display
----- ssm/1 -----
    amats: { [2x2] } [1x1]
    mmats: { [2x2] } [1x1]
    bmats: { [2x1] } [2x2] } [1x2]
    cmats: { [1x2] } [1x1]
    dmats: { [] } [1x2] } [1x2]
timestep: 0
inputs: [1x2 ssmblock]
    1 : U | Fu [kg m s^(-2)]
    2 : N | Fn [kg m s^(-2)], On [m]
states: [1x1 ssmblock]
    1 : standard test system | x [m], xdot [m s^(-1)]
outputs: [1x1 ssmblock]
    1 : Y | y [m]
params: (empty-plist) [1x1 plist]
version: $Id: ssm_build_built_in_content.html,v 1.3 2009/08/28 14:20:07 adrien Exp $--
>$Id: ssm_build_built_in_content.html,v 1.3 2009/08/28 14:20:07 adrien Exp $
Ninputs: 2
inputsizes: [1 2]
Noutputs: 1
outputsizes: 1
Nstates: 1
statesizes: 2
Nparams: 0
isnumerical: true
    hist: ssm.hist [1x1 history]
    procinfo: (empty-plist) [1x1 plist]
```

```
plotinfo: (empty-plist) [1x1 plist]
name: standard_system_params
description: standard spring-mass-dashpot test system
mdlfile:
UUID: 4415784d-79d9-408d-9b82-3f39ace0d518
```

Options for the built-in constructor

There are two options:

- NAME and DESCRIPTION allow to set the object's name and description in the constructor
- DIM allows to reduce the model to 3, 2 or 1 dimension. Lower dimension models are built using the 3D model and deleting inputs/states/outputs. The consequence is that parameters that affect primarily the 3D model may still remain in the 1D model in the „params“ field, but also in the matrices if they have a small effect on the 1D dynamics.
- WITHPARAMS allows to keep selected parameters symbolic, to modify their value later. However current issues with MuPad make the use of multiple symbolic systems complicated, as the symbolic parameters are global variable with problematic side effects. The keyword 'ALL' returns the system with all its parameters symbolic.

```
>> system = ssm(plist('built-in', 'standard_system_params', 'withparams', {'D'}))
>> system = ssm(plist('built-in', 'standard_system_params', 'withparams', 'ALL'))
>> sys = ssm(plist('built-in', 'standard_system_params', 'setnames', {'W'}, 'setvalues', 0.2*i));
```

Trying sys.isStable with the first and the third system shows a different result, the negative stiffness making the latter unstable.

- SETNAMES and SETVALUES allow to set the value for parameters, including those which are not kept symbolic. The first field is a cell array of strings, the second is a double array of the corresponding values.

How to make your own built-in model?

The model must exist as a function mfile in the built-in folders defined in the LTPDA preferences panel. The file name must be “ssm_model_<model name>.m” .

A good template to start writing a file should be “ssm_model_standard_system_params.m”. The model should support all the options above, and the use of the ssm “structure“ constructor is recommended because it does not increment the history.

◀ Building from scratch

Modifying systems ▶

©LTP Team

Modifying systems

Built-in models enable to build fast predefined models to use later in simulations.

Modifying using the setter functions

It is possible to set the content of the fields “input”, “output”, “states”, as well as the field “params”.

Here is a simple model

```
>> name = 'sys';
>> timestep = 0;
>> amats = cell(3,3);
>> bmats = cell(3,3);
>> cmats = cell(3,3);
>> dmats = cell(3,3);
>> amats{1,1} = -(sym('OMEGA'));
>> amats{2,2} = -2;
>> amats{3,3} = [0 1 ; -0.05 -0.01];
>> amats{3,1} = [-1;-3];
>> bmats{1,1} = 1;
>> bmats{2,2} = 2;
>> bmats{3,3} = 3*eye(2);
>> cmats{1,1} = 1;
>> cmats{2,2} = 1;
>> cmats{3,3} = eye(2);
>> dmats{1,3} = [6 6];
>> dmats{2,1} = 6;
>> dmats{3,2} = [6;6];
>> sys = ssm(plist( ...
'amats',amats, 'bmats',bmats, 'cmats',cmats, 'dmats',dmats, ...
'timestep',timestep, 'name',name));
>> sys.setParams(plist({'OMEGA', 'system frequency'}, 2))
M: running ssm/ssm
M: running ssmFromDescription
M: running validate
M: running ssm/setParams
M: running display
----- ssm/1 -----
amats: { [1x1] [] []
[] [1x1] []
[2x1] [] [2x2] } [3x3]
mmats: { [1x1] [] []
[] [1x1] []
[] [] [2x2] } [3x3]
bmats: { [1x1] [] []
[] [1x1] []
[] [] [2x2] } [3x3]
cmats: { [1x1] [] []
[] [1x1] []
[] [] [2x2] } [3x3]
dmats: { [1] [] [1x2]
[1x1] [] []
[] [2x1] [] } [3x3]
timestep: 0
inputs: [1x3 ssmblock]
1 : input 1 | input 1 > 1 []
2 : input 2 | input 2 > 1 []
3 : input 3 | input 3 > 1 [], input 3 > 2 []
states: [1x3 ssmblock]
1 : state 1 | state 1 > 1 []
2 : state 2 | state 2 > 1 []
3 : state 3 | state 3 > 1 [], state 3 > 2 []
outputs: [1x3 ssmblock]
1 : output 1 | output 1 > 1 []
2 : output 2 | output 2 > 1 []
3 : output 3 | output 3 > 1 [], output 3 > 2 []
params: (OMEGA=2) [1x1 plist]
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
Ninputs: 3
inputsizes: [1 1 2]
```

```
Noutputs: 3
outputsizes: [1 1 2]
Nstates: 3
statesizes: [1 1 2]
Nparams: 1
isnumerical: false
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: sys
description:
mdlfile:
UUID: 227fd0a2-1de0-4e01-855f-d0548e1eb1ff
-----
```

In then we set the names and descriptions of the input blocks, the names, descriptions and units of the port variables:

```
>> sys.setBlockNames(plist('field', 'inputs', 'blocks', [1 2 3], 'names',
{'myinput1', 'myinput2', 'myinput3'}));
>> sys.setBlockDescriptions(plist('field', 'inputs', 'blocks', {'myinput1'},
'descriptions', {'myinput1 description'}));
>> sys.setPortNames(plist('field', 'inputs', 'block', {'myinput3'}, 'ports', [1 2],
'names', {'my port 3-1', 'my port 3-2'}));
>> sys.setPortDescriptions(plist('field', 'inputs', 'block', {'myinput3'}, 'ports', [1 2],
'descriptions', {'first description', 'second description'}));
>> sys.setPortUnits(plist('field', 'inputs', 'block', {'myinput3'}, 'ports', [1 2], 'units',
[unit('kg m^-2') unit('m')]) );
>> sys.inputs
M: running ssm/setBlockNames
M: running ssm/setBlockDescriptions
M: running ssm/setPortNames
----- ssmport/1 -----
name: my port 3-1
units: [] [1x1 unit]
description:
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmport/2 -----
name: my port 3-2
units: [] [1x1 unit]
description:
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
M: running ssm/setPortDescriptions
----- ssmport/1 -----
name: my port 3-1
units: [] [1x1 unit]
description: first description
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmport/2 -----
name: my port 3-2
units: [] [1x1 unit]
description: second description
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
M: running ssm/setPortUnits
----- ssmblock/1 -----
name: myinput1
ports: input 1 > 1 [] [1x1 ssmport]
description: myinput1 description
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmblock/2 -----
name: myinput2
ports: input 2 > 1 [] [1x1 ssmport]
description:
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
----- ssmblock/3 -----
name: myinput3
ports: my port 3-1 [kg m^(-2)], my port 3-2 [m] [1x2 ssmport]
description:
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
-----
```


Modifying the parameter values

Once the parameter field “params” is set, one may want to set or modify (like in an optimizer) the parameter values. This is done by the functions “*Parameters.m”. The function `setParameters` takes two inputs ('setnames' and 'setvalues') which are the names of the parameters to modify and the values assigned to them.

```
>> sys.params
>> sys.setParameters('OMEGA', 0.002);
>> sys.params
----- plist 01 -----
n params: 1
---- param 1 ----
key: OMEGA
val: 2
desc: system frequency
-----
description:
UUID: 84ff1ed8-eee2-406c-89d8-7e6e703d36dd
-----
M: running setParameters
M: running ssm/ssm
----- plist 01 -----
n params: 1
---- param 1 ----
key: OMEGA
val: 0.002
desc: system frequency
-----
description:
UUID: 84ff1ed8-eee2-406c-89d8-7e6e703d36dd
-----
```

The function `subsParameterskeep` and `Parameters` substitute numerical values to the symbolic expressions, using a list of the parameters to substitute (or its complementary). All parameters substituted are removed from the “params” field. The call `<system>.keepParameters` substitutes all the parameters. The field `isnumerical` is then set to 1, unless a parameter in the matrices was forgotten in the params field.

```
>> sys.keepParameters;
>> sys.params
>> sys.isNumeric

M: running keepParameters
----- plist 01 -----
n params: 0
description:
UUID: 84ff1ed8-eee2-406c-89d8-7e6e703d36dd
-----
ans =
1
```

Modifying the inputs/states/outputs

Let us first build a simple system constituted of four parallel 1st order systems.

```
>> name = 'sys';
>> timestep = 0;
>> amats = {diag([-1 -2 -3 -4])};
>> bmats = {diag([1 2 0 0])};
>> cmats = {[diag([1 0 3 0])]; [0 0 0 0]};
>> dmats = {[diag([0 0 0 0])]; [0 0 0 0]};
>> sys = ssm(plist(...
'amats',amats,'bmats',bmats,'cmats',cmats,'dmats',dmats,...
'timestep',timestep,'name',name));
```

It is possible to obtain the structural realization of the system using `sMinReal`. This function deletes states which are either not observable or controllable. However, there may still exist such linear combinations of states.

The process only modify the states, here the 2nd and 4th states disappear since they are not observable, and the 3rd and 4th disappear since they are not controllable.

```
>> sys.sMinReal
M: running ssm/ssm
```

```

M: running ssmFromDescription
M: running validate
M: running ssm/ssmReal
M: running display
----- ssm/1 -----
amats: { [1x1] } [1x1]
mmats: { [1x1] } [1x1]
bmats: { [1x4] } [1x1]
cmats: { [5x1] } [1x1]
dmats: { [ ] } [1x1]
timestep: 0
inputs: [1x1 ssmblock]
1 : input 1 | input 1 > 1 [], input 1 > 2 [], input 1 > 3 [], input 1 > 4 []
states: [1x1 ssmblock]
1 : state 1 | state 1 > 1 []
outputs: [1x1 ssmblock]
1 : output 1 | output 1 > 1 [], output 1 > 2 [], output 1 > 3 [], output 1 > 4 [], output 1
> 5 []
params: (empty-plist) [1x1 plist]
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
Ninputs: 1
inputsizes: 4
Noutputs: 1
outputsizes: 5
Nstates: 1
statesizes: 1
Nparams: 0
isnumerical: true
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: sys
description:
mdlfile:
UUID: 349cd9c4-ad91-4151-b542-63d34b01bed8
-----

```

Then we can work on the input/outputs to simplify the model.

The user must specify which block variable he want to keep, and three syntaxes are allowed.

- 'ALL' meaning all ports in all blocks are kept
- {<varname1> <varname2> ...} giving the name of each port to keep, in this case one must ensure ports of different blocks have different names.
- {<logical index for block1> <double index for block2> <varnames cellstr for block3> "ALL" "NONE"} using a different index for each individual block.

This syntax is also used for any converter to select the inputs and outputs in the ssm.

It is important to note that this removes lines and columns in the system's matrices, but no parameters are removed. So second order parameters that should be remove (by setting them to a neutral value) must be taken care of separately.

```

>> sys.simplify(plist('inputs', {[1 2]}, 'states', 'ALL', 'outputs', { 'output 1 > 1'
'output 1 > 4'})))
M: running ssm/simplify
M: running ssm/ssm
M: running ssm/simplify
M: running display
----- ssm/1 -----
amats: { [1x1] } [1x1]
mmats: { [1x1] } [1x1]
bmats: { [1x2] } [1x1]
cmats: { [2x1] } [1x1]
dmats: { [ ] } [1x1]
timestep: 0
inputs: [1x1 ssmblock]
1 : input 1 | input 1 > 1 [], input 1 > 2 []
states: [1x1 ssmblock]
1 : state 1 | state 1 > 1 []
outputs: [1x1 ssmblock]
1 : output 1 | output 1 > 1 [], output 1 > 4 []
params: (empty-plist) [1x1 plist]
version: $Id: ssm_modify_content.html,v 1.2 2009/08/28 15:11:53 adrien Exp $
Ninputs: 1
inputsizes: 2

```

```
Noutputs: 1
outputsizes: 2
Nstates: 1
statesizes: 1
Nparams: 0
isnumerical: true
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: sys
description:
mdlfile:
UUID: 8e55110a - eb07 - 44d2 - aaee - 4f839a5fb155
-----
```

Assembling systems

A collection of ssm arrays can be assembled into one ssm using the assemble function.

Managing inputs/outputs when assembling

The order of the systems does not modify the output of . The function assemble work by matching inputs and outputs of the same name. Of course, they must have the same dimensionality, same time-step. However, the check for units is not implemented yet.

In terms of time-step it is important to assemble all continuously linked models together when the system is continuous, and discretize it later on. Time discrete models (typically digital systems, and the dynamical branch from the actuator input to the sensor output) should be assembled together when they are time-discrete. The discretization includes a zero hold which models correctly the A/D filter behavior.

Moreover, a system can be assembled in multiple steps. In this case there is a risk of “closing a loop” multiple times. To avoid this, the method “assemble” suppresses the inputs once they are assembled. May the user need the input for a simulation later on, he must duplicate it using the function “inputDuplicate”. Built-in models should also have built-in duplicated inputs to insert signals.

Example using an existing built-in models

```
>> sys = ssm(plist('built-in', 'standard_system_params', 'setnames', {'W'}, 'setvalues', 0.2*i));
```

Trying sys.isStable with the first and the third system shows a different result, the negative stiffness making the latter unstable. The system can be made time discrete using the function “modifTimeStep”. The input is then duplicated to be used for a controller feedback.

```
>> sys = ssm(plist('built-in', 'standard_system_params', 'setnames', {'W' 'C'}, 'setvalues', [-0.2 -0.5]));
>> sys.modifTimeStep(0.01);
>> sys.duplicateInput('U', 'Negative Bias')
----- ssm/1 -----
amats: { [2x2] } [1x1]
mmats: { [2x2] } [1x1]
bmats: { [2x1] [2x2] [2x1] } [1x3]
cmats: { [1x2] } [1x1]
dmats: { [] [1x2] [] } [1x3]
timestep: 0.01
inputs: [1x3 ssmblock]
1 : U | Fu [kg m s^(-2)]
2 : N | Fn [kg m s^(-2)] On [m]
3 : Negative Bias | Fu [kg m s^(-2)]
states: [1x1 ssmblock]
1 : standard test system | x [m], xdot [m s^(-1)]
outputs: [1x1 ssmblock]
1 : Y | y [m]
params: (empty-plist) [1x1 plist]
version: $Id: ssm_assemble_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $-->$Id:
ssm_assemble_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
Ninputs: 3
inputsizes: [1 2 1]
Noutputs: 1
outputsizes: 1
Nstates: 1
statesizes: 2
Nparams: 0
isnumerical: true
```

```

    hist: ssm.hist [1x1 history]
    procinfo: (empty-plist) [1x1 plist]
    plotinfo: (empty-plist) [1x1 plist]
    name: standard_system_params
description: standard spring-mass-dashpot test system
    mdlfile:
        UUID: 284b0ae3-947d-430a-802e-d9c3738ebb14
-----
M: running isStable
ans =
    2.05114794494015
warning, system named "standard_system_params" is not stable
ans =
    0

```

Then a controller can be created to make the system stable.

```

>> controller = ssm(plist( ...
    'amats',cell(0,0), 'bmats',cell(0,1), 'cmats',cell(1,0), 'dmats',{ -1}, ...
    'timestep',0.01, 'name','controller', 'params',plist, ...
    'statenames',{ }, 'inputnames',{'Y'}, 'outputnames',{'U'} ));
>> sysCL = assemble(sys, controller);
>> sysCL.isStable

----- ssm/1 -----
    amats: { [2x2] } [1x1]
    mmats: { [2x2] } [1x1]
    bmats: { [2x2] } [2x1] [1x2]
    cmats: { [1x2] } [2x1]
    dmats: { [1x2] } [2x1]
    timestep: 0.01
    inputs: [1x2 ssmblock]
        1 : N | Fn [kg m s^(-2)], On [m]
        2 : Negative Bias | Fu [kg m s^(-2)]
    states: [1x1 ssmblock]
        1 : standard test system | x [m], xdot [m s^(-1)]
    outputs: [1x2 ssmblock]
        1 : Y | y [m]
        2 : U | U > 1 []
    params: (empty-plist) [1x1 plist]
    version: $Id: ssm_assemble_content.html,v 1.4 2009/08/28 15:11:53 adrien Exp $
    Ninputs: 2
    inputsizes: [2 1]
    Noutputs: 2
    outputsizes: [1 1]
    Nstates: 1
    statesizes: 2
    Nparams: 0
isnumerical: true
    hist: ssm.hist [1x1 history]
    procinfo: (empty-plist) [1x1 plist]
    plotinfo: (empty-plist) [1x1 plist]
    name: assembled( standard_system_params + controller))
description:
    mdlfile:
        UUID: c6f7397e-add7-4f4c-8eb0-fd0a4841e3cf
-----
M: running isStable
System named "assembled( standard_system_params + controller))" is stable
ans =
    1

```

We can then use the system to produce a simulation.

Simulations

The function `simulate` can use `ssm` object to produce simulations.

Simulation example.

The following closed loop system is built.

```
>> sys = ssm(plist('built-in', 'standard_system_params', 'setnames', {'W' 'C'}, 'setvalues',
[-0.2 -0.5]));
>> sys.modifTimeStep(0.01);
>> sys.duplicateInput('U', 'Negative Bias');
>> controller = ssm(plist( ...
'amats', cell(0,0), 'bmats', cell(0,1), 'cmats', cell(1,0), 'dmats', {-1}, ...
'timestep', 0.01, 'name', 'controller', 'params', plist, ...
'statenames', {}, 'inputnames', {'Y'}, 'outputnames', {'U'} ));
----- ssm/1 -----
      amats: { [2x2] } [1x1]
      mmats: { [2x2] } [1x1]
      bmats: { [2x2] [2x1] } [1x2]
      cmats: { [1x2] } [2x1]
      dmats: { [1x2] [ ] } [2x2]
timestep: 0.01
inputs: [1x2 ssmblock]
  1 : N | Fn [kg m s^(-2)], On [m]
  2 : Negative Bias | Fu [kg m s^(-2)]
states: [1x1 ssmblock]
  1 : standard test system | x [m], xdot [m s^(-1)]
outputs: [1x2 ssmblock]
  1 : Y | y [m]
  2 : U | U > 1 [ ]
params: (empty-plist) [1x1 plist]
version: $Id: ssm_simulation_content.html,v 1.3 2009/08/28 14:20:07 adrien Exp $
Ninputs: 2
inputsizes: [2 1]
Noutputs: 2
outputsizes: [1 1]
Nstates: 1
statesizes: 2
Nparams: 0
isnumerical: true
hist: ssm.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: assembled( standard_system_params + controller)
description:
  mdlfile:
    UUID: 163d7103-063b-4a57-af7e-b08d22fe42c1
-----
```

Then we wish to use the inputs of `N` for a correlated force noise and measurement noise, “Negative Bias” for a sinewave, and there will be an observation DC offset.

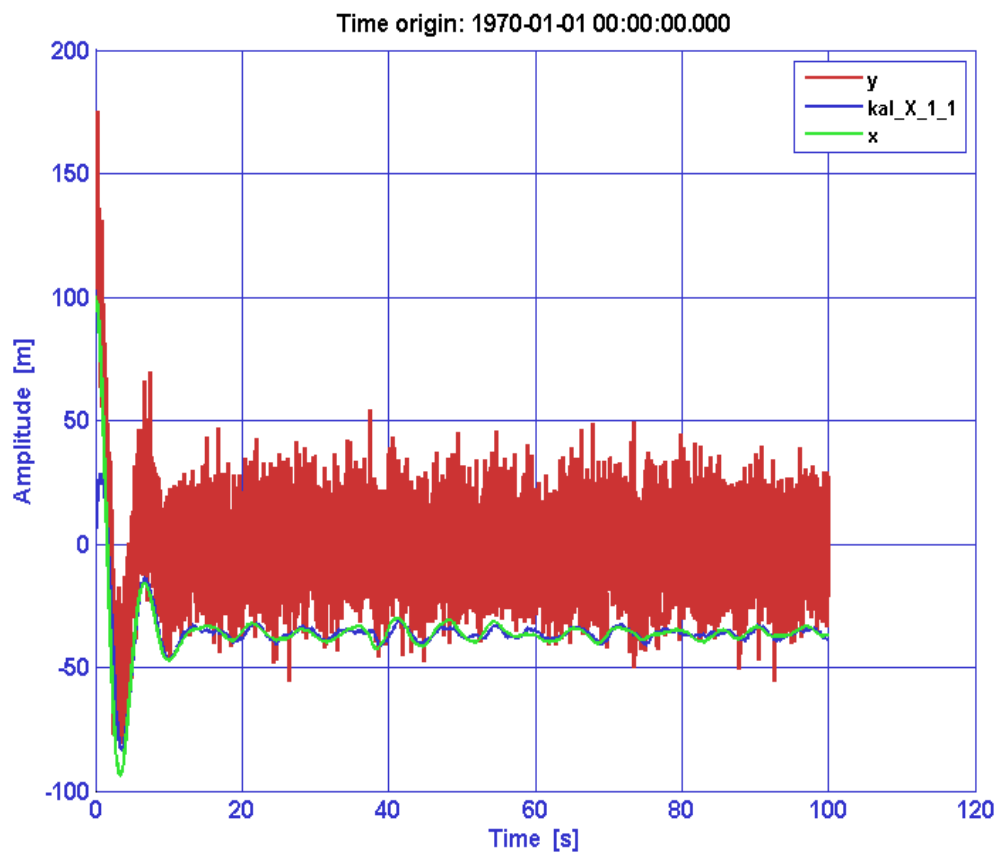
We want as an output the controller output “`U`” and the sensor output “`y`”.

```
>> ao1 = ao(plist('FCN', 'sin(0:0.01:100)'));
>> ao_out = sysCL.simulate( plist( ...
'NOISE VARIABLE NAMES', {'Fn' 'On'}, 'COVARIANCE', [1 0.1 ; 0.1 2] , ...
'AOS VARIABLE NAMES', {'Fu'}, 'AOS', ao1, ...
'CONSTANT VARIABLE NAMES', {'On'}, 'CONSTANTS', 35, ...
'RETURN STATES', {'x'}, 'RETURN OUTPUTS', {'y' 'U > 1'}, ...
'SSINI', {[100;3]}, 'TINI', 0));
>> iplot(ao_out([1, 2, 3]));
```



It turns out the system output (blue) is not much like the state (red), causing the control (green) to waste a lot of energy. The state is not experimentally available, but might be obtained through filtering. The kalman method is so far the only filtering method implemented in the toolbox.

```
>> ao_est = sysCL.kalman( plist( ...
    'NOISE VARIABLE NAMES', {'Fn', 'On'}, 'COVARIANCE', [1 0.1 ; 0.1 2] , ...
    'AOS VARIABLE NAMES', {'Fu'} , 'AOS', ao1 , ...
    'CONSTANT VARIABLE NAMES', {'On'}, 'CONSTANTS', 35, ...
    'OUTPUT VARIABLE NAMES', {'y'}, 'OUTPUTS', ao_out(2), ...
    'RETURN STATES', 1, 'RETURN OUTPUTS', 1 ));
>> iplot(ao_out(2), ao_est(1), ao_out(1))
```



In this example the estimate (blue) of the state (green) is satisfactory. It leads us to think that such a filter should be used to provide with the input of the controller.

However, the DC offset correction by the kalman filter is one information that is not available under usual circumstances.

Transfer Function Modelling

In the LTPDA toolbox you have several way to define transfer functions depending on the mathematical representation you want to use

- [Pole/zero representation](#)
- [Sum of partial fractions representation](#)
- [Rational representation](#)

Pole/Zero representation

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

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](#)

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 , or by a complex number.

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

```
>> pz(1)
----- pz/1 -----
f: 1
q: NaN
ri: -6.28318530717959
version: $Id: pzmodel_pz_content.html,v 1.3 2009/02/18 12:15:16 hewitson Exp $
-----
```

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

```
>> pz(1,4)
----- pz/1 -----
f: 1
q: 4
ri: [0.785398163397448+i*6.23390466154956;0.785398163397448-i*6.23390466154956]
version: $Id: pzmodel_pz_content.html,v 1.3 2009/02/18 12:15:16 hewitson Exp $
-----
```

This creates a complex pole at 1Hz with a Q of 4. You can also see that the complex representation is also shown, but only one part of the conjugate pair.

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 and a delay can be specified such that the resulting model is of the form:

$$H(s) = G \frac{(s - z_1)(s - z_2) \dots (s - z_n)}{(s - p_1)(s - p_2) \dots (s - p_m)} e^{-i\omega\tau}$$

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

- [Direct form](#)
- [Creating from a plist](#)
- [Computing the response of the model](#)

Direct form

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

```
>> pzm = pzmodel(10, {[1 2], 3}, {5, 10}, 0.01)
---- pzmodel 1 ----
name: None
gain: 10
delay: 0.01
iunits: []
ounits: []
pole 001: (f=1 Hz, Q=2)
pole 002: (f=3 Hz, Q=NaN)
zero 001: (f=5 Hz, Q=NaN)
zero 002: (f=10 Hz, Q=NaN)
-----
```

Notice, you can also pass arrays of pz objects to the `pzmodel` constructor, but this should rarely be necessary.

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', {[1 2], 3}, ...
              'zeros', {5, 10}, ...
              'delay', 0.01, ...
              'iunits', 'm', ...
              'ounits', 'V^2');
>> pzm = pzmodel(pl)
---- pzmodel 1 ----
name: test model
gain: 10
delay: 0.01
iunits: [m]
ounits: [V^2]
pole 001: (f=1 Hz, Q=2)
pole 002: (f=3 Hz, Q=NaN)
zero 001: (f=5 Hz, Q=NaN)
zero 002: (f=10 Hz, Q=NaN)
```

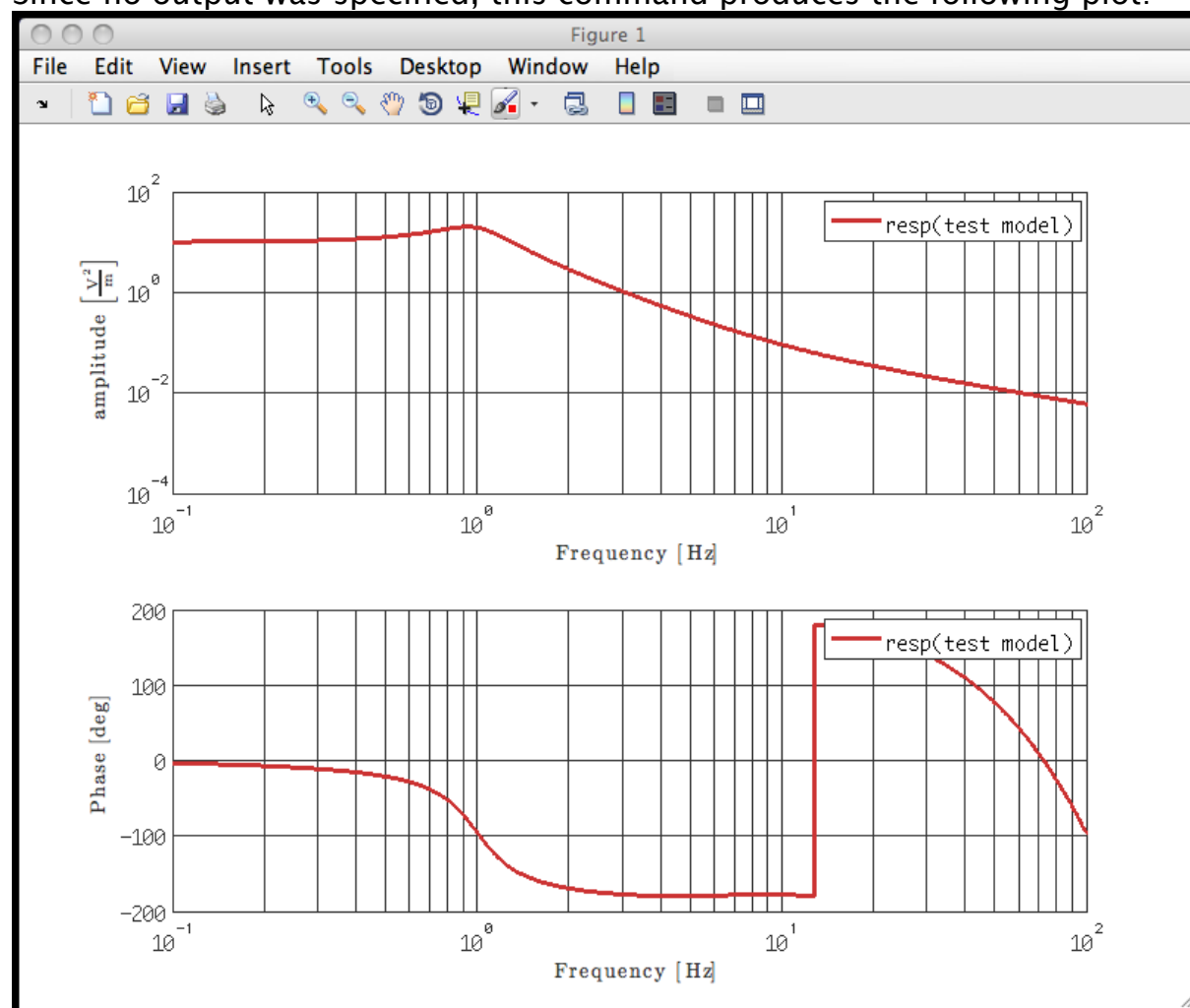
Here we also specified the input units of the transfer function ('iunits') and the output units, ('ounits'). In this case, the model represents a transfer function from metres to Volts squared.

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 01: resp(test model) -----
name:    resp(test model)
description:
data: (0.1,10.0668830776529-i*0.605439551995965) (0.100092155051679,10.067006787497-
i*0.606014805088671) (0.100184395028894,10.0671307268392-i*0.606590636924472)
(0.100276720009908,10.0672548961078-i*0.607167048174596) (0.100369130073055,10.0673792957318-
i*0.607744039511284) ...
----- fsdata 01 -----
fs:      NaN
```



```

        x:  [1 10000], double
        y:  [1 10000], double
    xunits:  [Hz]
    yunits:  [V^(2)][m^(-1)]
        t0: 1970-01-01 00:00:00.000
        navs: NaN
    -----
    hist:  pzmodel / resp / $Id: pzmodel_model_content.html,v 1.5 2009/02/24 09:44:39
miquel Exp $
    mfilename:
    mdlfilename:
    -----
```

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 ▶

©LTP Team



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](#).

Sum of partial fractions representation

Transfer functions can be expressed as a quotient of polynomials

$$H(s) = K(s) + \sum_{i=1}^N \frac{R_i}{s - p_i}$$

The constructor can be used in different ways

From poles and residues

The standard way is to input the coefficients of your filter. The constructor accepts as a optional properties the name

```
>> par = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], [])
---- parfrac 1 ----
model:      None
res:        [1;2+i*1;2-i*1]
poles:      [6;1+i*3;1-i*3]
dir:        0
pmul:       [1;1;1]
iunits:     []
ounits:     []
-----
```

From partial XML file

You can input a XML file containing a transfer function model into the constructor

```
>> par = parfrac('datafile.xml')
```

From mat file

You can input a mat file containing a transfer function model into the constructor

```
>> rat = parfrac('datafile.mat')
```

From plist

All the properties of the filter can be specified in a plist and then passed to the constructor:

```
>> pl = plist('iunits','m','ounits','V','res',[1 2+1i 2-1i],'poles',[6 1+3i 1-3i],...
'name','filter_mame');
>> par = parfrac(pl)
---- parfrac 1 ----
model:      filter_mame
res:        [1;2+i*1;2-i*1]
poles:      [6;1+i*3;1-i*3]
dir:        0
pmul:       [1;1;1]
iunits:     [m]
```

```
ounits:  [V]
-----
```

From repository

Rational transfer function can be obtained from the [repository](#) with the following syntax.

```
>> rat    = rational('Hostname','localhost','Database','ltpda',...
'ID',[], 'CID',[], 'Binary',yes)
```

← Model helper GUI	Rational representation →

©LTP Team

Rational representation

Transfer functions can be expressed as a quocient of polynomials as in the following expression

$$H(s) = \frac{a_1 s^m + a_2 s^{m-1} + \dots + a_{m+1}}{b_1 s^n + b_2 s^{n-1} + \dots + b_{n+1}}$$

The constructor can be used in different ways

From coefficients

The standard way is to input the coefficients of your filter. The constructor accepts as a optional properties the name

```
>> rat = rational([1 3 5 7],[5 10 0.01],'filter_name')
---- rational 1 ----
model:    filter_name
num:      [1 3 5 7]
den:      [5 10 0.01]
iunits:   []
ounits:   []
-----
```

From partial XML file

You can input a XML file containing a transfer function model into the constructor

```
>> rat = rational('datafile.xml')
```

From mat file

You can input a mat file containing a transfer function model into the constructor

```
>> rat = rational('datafile.mat')
```

From plist

All the properties of the filter can be specified in a plist and then passed to the constructor:

```
>> pl = plist('iunits','m','ounits','V','num',[1 3 10],'den',[4 6],...
'name','filter_name');
>> par = parfrac(pl)
---- rational 1 ----
model:    filter_name
num:      [1 3 10]
den:      [4 6]
iunits:   [m]
ounits:   [V]
-----
```

From repository

Rational transfer function can be obtained from the [repository](#) with the following syntax.

```
>> rat = rational('Hostname','localhost','Database','ltpda',...  
'ID',[],'CID',[],'Binary',yes)
```

◀ Sum of partial fractions representation	Converting models between different representations ▶

©LTP Team

Converting models between different representations

The different constructors from each transfer function representations accept as an input a model from a another representation so that they can all be converted between the different representations. In the current LTPDA version, this applies for pole/zero model and rational representation. Following versions will cover the partial fraction representation. This is shown in the following transformation table:

	Pole/Zero	Rational	Partial Fraction
Pole/Zero		✓	✗
Rational	✓		✗
Partial Fraction	✗	✗	

From pzmodel to rational

You can transform a `pzmodel` into a `rational` by typing:

```
>> rat = rational(pzmodel)
```

From rational to pzmodel

You can transform a `rational` into a `pzmodel` by typing:

```
>> rat = pzmodel(rational)
```

Algorithm

To translate from `rational` to `pzmodel` representation we need to compute the roots of a polynomial and the inverse operation is performed going from `pzmodel` to `rational`. More information about the algorithm used can be found in MATLAB's functions [poly](#) and [roots](#).

Converting models to digital filters

Transfer functions models can be converted to IIR/FIR filters using the bilinear transform. The result of the conversion is an `miir/mfit` object. To convert a model, you need simply to input your model into the `miir/mfir` constructor. In the current LTPDA version, only the following discretization from transfer function models are allowed:

	Digital (IIR/FIR)
Pole/Zero	✓
Rational	✗
Partial Fraction	✓

From pzmodel

To get an IIR filter from a given `pzmodel` we need to write down

```
>> filt = miir(pzmodel)
```

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(pzmodel,plist('fs', 1000))
```

For more information of IIR filters in LTPDA, see [IIR Filters](#).

From partial fraction

Analogously, the same rules apply to a partial fraction model `parfrac` constructor

```
>> filt = miir(parfrac)
```


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:

- [Downsampling data](#)
- [Upsampling data](#)
- [Resampling data](#)
- [Interpolating data](#)
- [Spikes reduction in data](#)
- [Data gap filling](#)
- [Noise Whitening](#)

Downsampling data

Description

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.

Syntax

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

Parameters

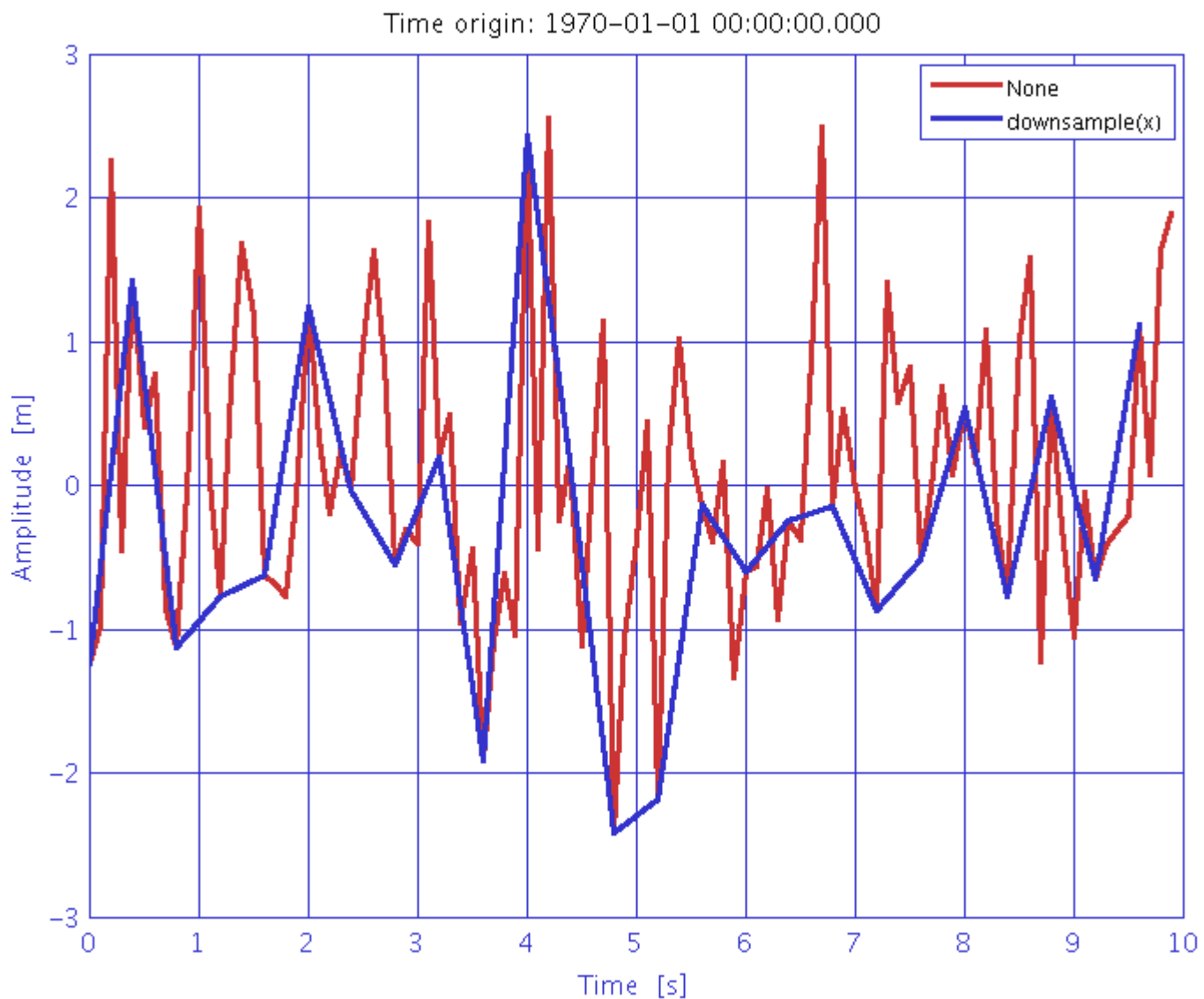
The following parameters can be set in this method:

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

Examples

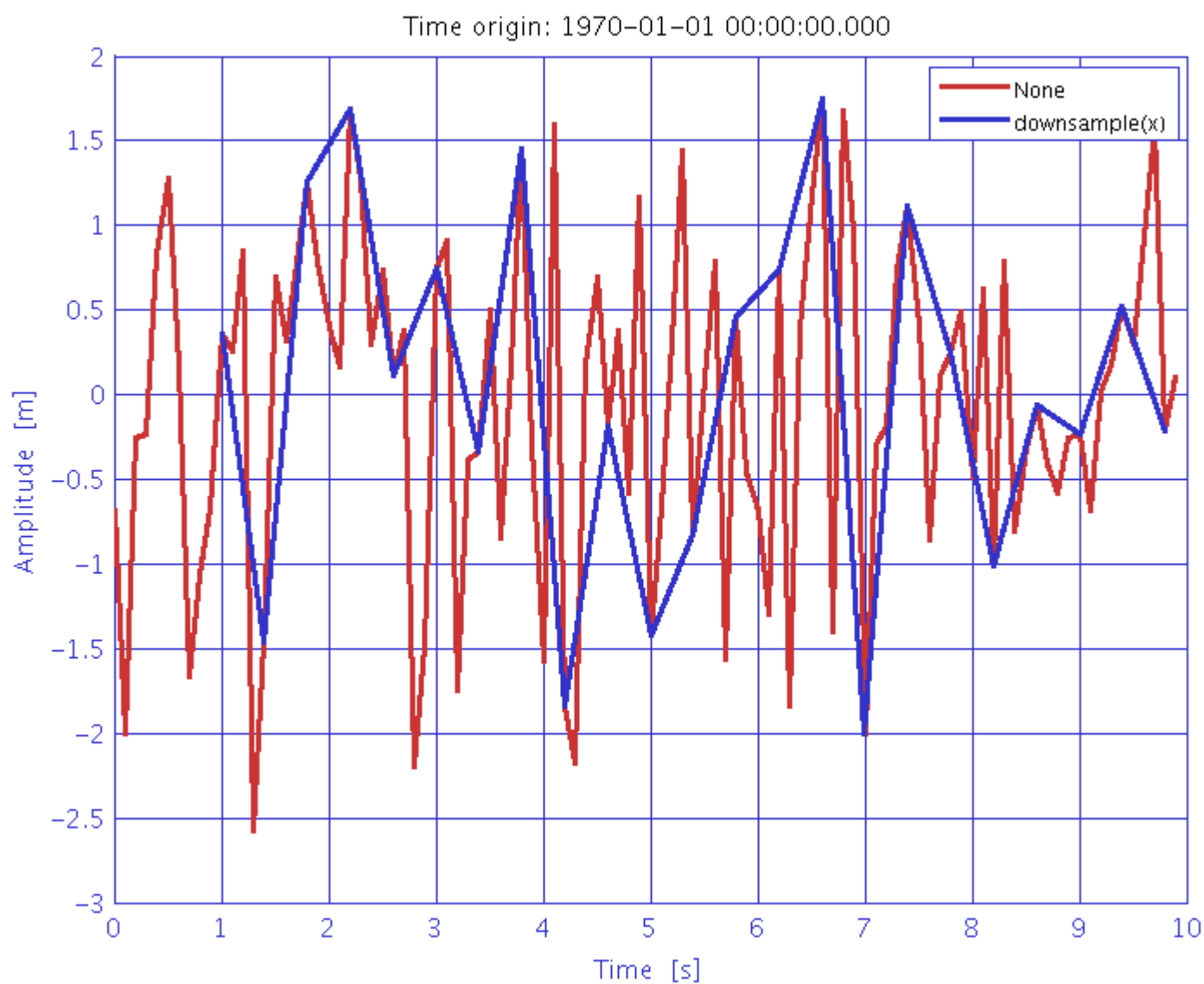
1. Downsampling a sequence of random data at original sampling rate of 10 Hz by a factor of 4 (`fsout` = 2.5 Hz) and no offset.

```
% create an AO of random data with fs = 10 Hz
pl      = plist('tsfcn', 'randn(size(t))','fs',10,'units','m');
x       = ao(pl)
pl_down = plist('factor', 4);      % add the decimation factor
y       = downsample(x, pl_down); % downsample the input AO, x
ipplot(x, y)
```



2. Downsampling a sequence of random data at original sampling rate of 10 Hz by a factor of 4 ($f_{\text{sout}} = 2.5$ Hz) and offset = 10.

```
% create an AO of random data with fs = 10 Hz
pl      = plist('tsfcn', 'randn(size(t))', 'fs', 10, 'yunits', 'm');
x       = ao(pl)
pl_downoff = plist('factor', 4, 'offset', 10); % add the decimation factor and the offset
parameter
y       = downsample(x, pl_downoff);          % downsample the input AO, x
ipplot(x, y)
```

[◀ Signal Pre-processing in LTPDA](#)[Upsampling data ▶](#)

©LTP Team

Upsampling data

Description

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 $(n \times \text{input})$ samples. In addition, an initial phase can be specified and, thus, a delayed output of the input can be obtained by using this option.

Syntax

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

Parameters

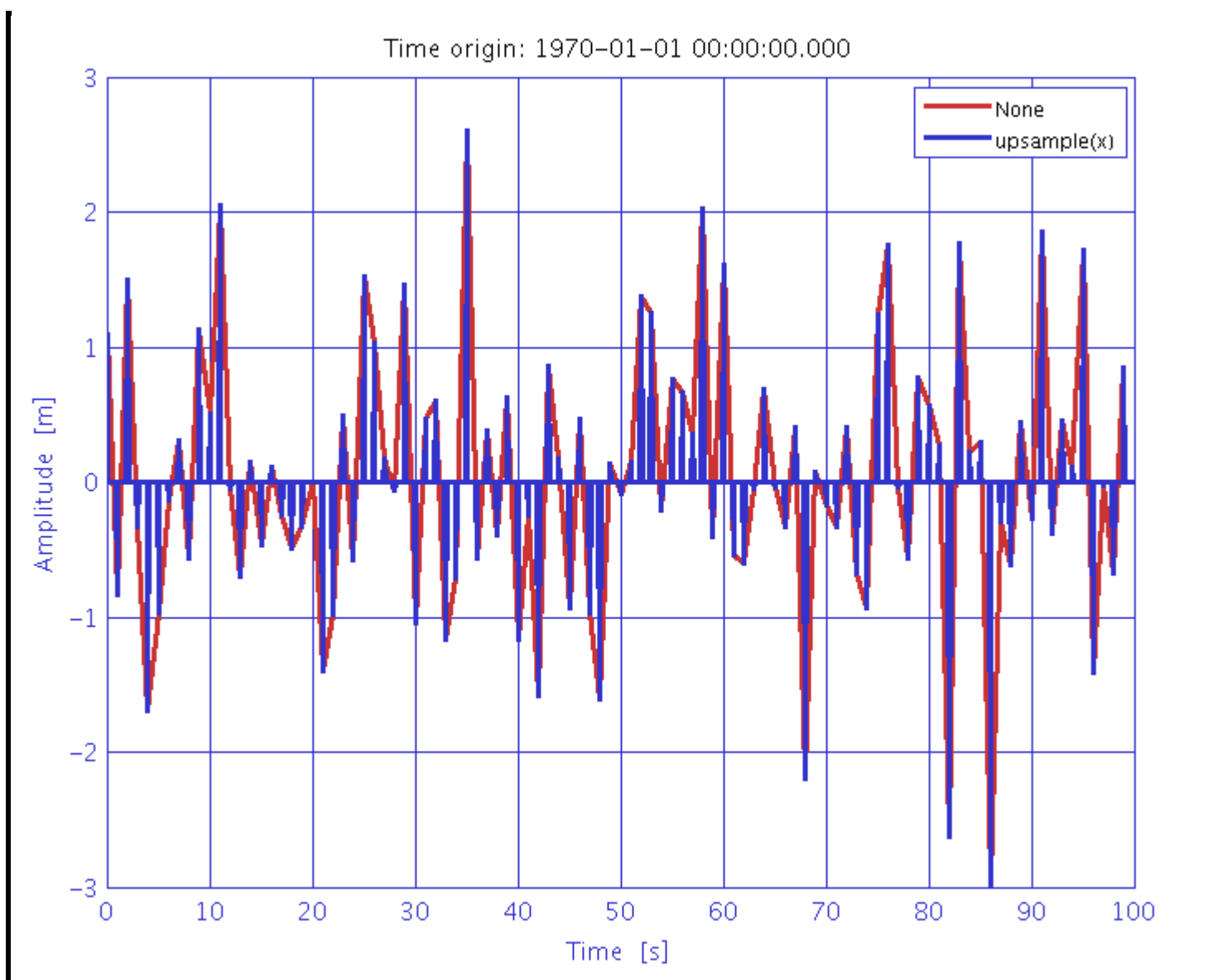
The following parameters can be set in this method:

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

Examples

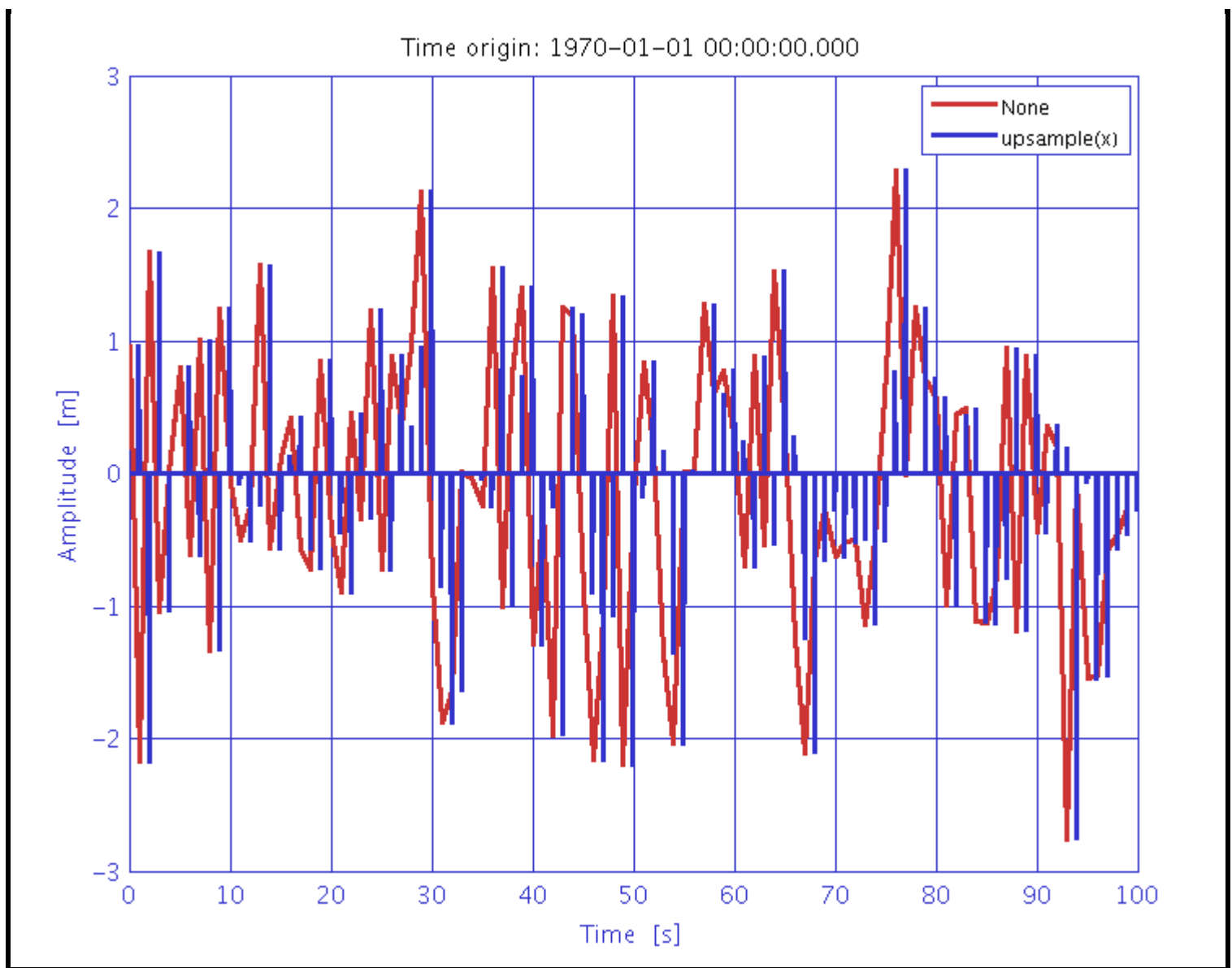
1. Upsampling a sequence of random data at original sampling rate of 1 Hz by a factor of 10 with no initial phase.

```
pl    = plist('tsfcn', 'randn(size(t))', 'fs', 1, 'yunits', 'm', 'nsecs', 100);  
x     = ao(pl);  
pl_up = plist('N', 10);           % increase the sampling frequency by a factor of 10  
y     = upsample(x, pl_up);       % resample the input AO (x) to obtain the upsampled AO (y)  
iplot(x, y)
```



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.

```
pl      = plist('tsfcn', 'randn(size(t))', 'fs', 1, 'yunits', 'm', 'nsecs', 100);
x       = ao(pl);
pl_upphase = plist('N', 21, 'phase', 20); % increase the sampling frequency and add phase
of 20 samples to upsampled data
y       = upsample(x, pl_upphase);      % resample the input A0 (x) to obtain the
upsampled and delayed A0 (y)
iplot(x, y)
```

☐ Downsampling dataResampling data ☐

©LTP Team

Resampling data

Description

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.

Syntax

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

Parameters

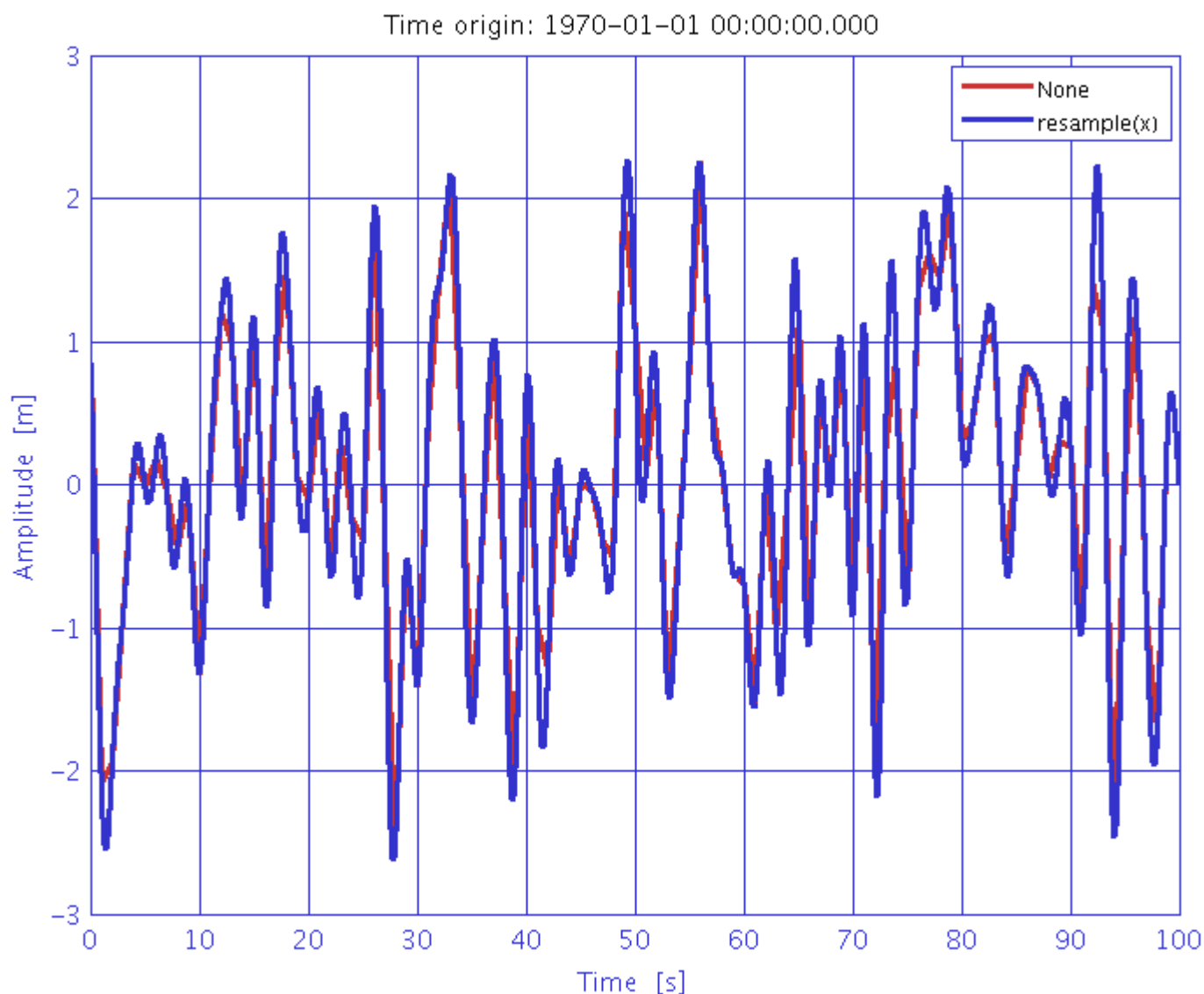
The following parameters can be set:

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

Examples

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

```
% create an AO of random data with fs = 10 Hz
pl = plist('tsfcn', 'randn(size(t))', 'fs', 1, 'yunits', 'm', 'nsecs', 100);
x = ao(pl);
y = resample(x, plist('fsout', 50)); % resample the input AO (x) to obtain the
resampled output AO (y)
ipplot(x, y) % plot original and resampled data
```

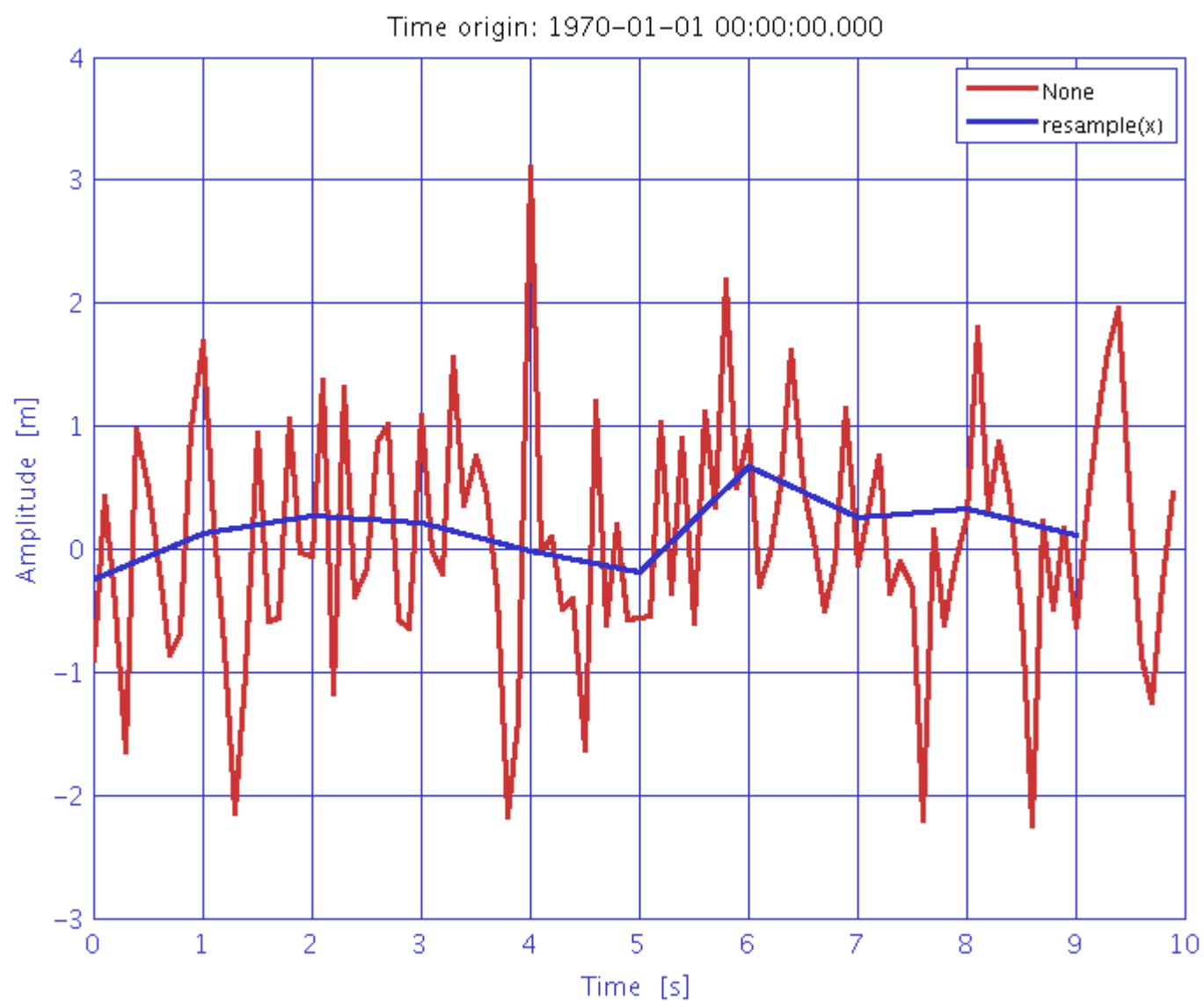


1. Resampling a sequence of random data at original sampling rate of 10 Hz at an output sampling of 1 Hz with a filter defined by the user.

```
% create an AO of random data with fs = 10 Hz
pl = plist('tsfcn', 'randn(size(t))', 'fs', 10, 'yunits', 'm');
x = ao(pl)

% filter definition
plfilter = plist('type', 'lowpass', 'Win', specwin('Kaiser', 10,
150), 'order', 32, 'fs', 10, 'fc', 1);
f = mfir(plfilter)

% resampling
pl = plist('fsout', 1, 'filter', f)); define parameters list with fsout = 1 Hz and the
defined filter
y = resample(x, pl); % resample the input AO (x) to obtain the resampled output AO (y)
ipplot(x, y) % plot original and resampled data
```



◀ Upsampling data

Interpolating data ▶

©LTP Team

Interpolating data

Description

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.

Syntax

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

Parameters

The following parameters can be set in the method:

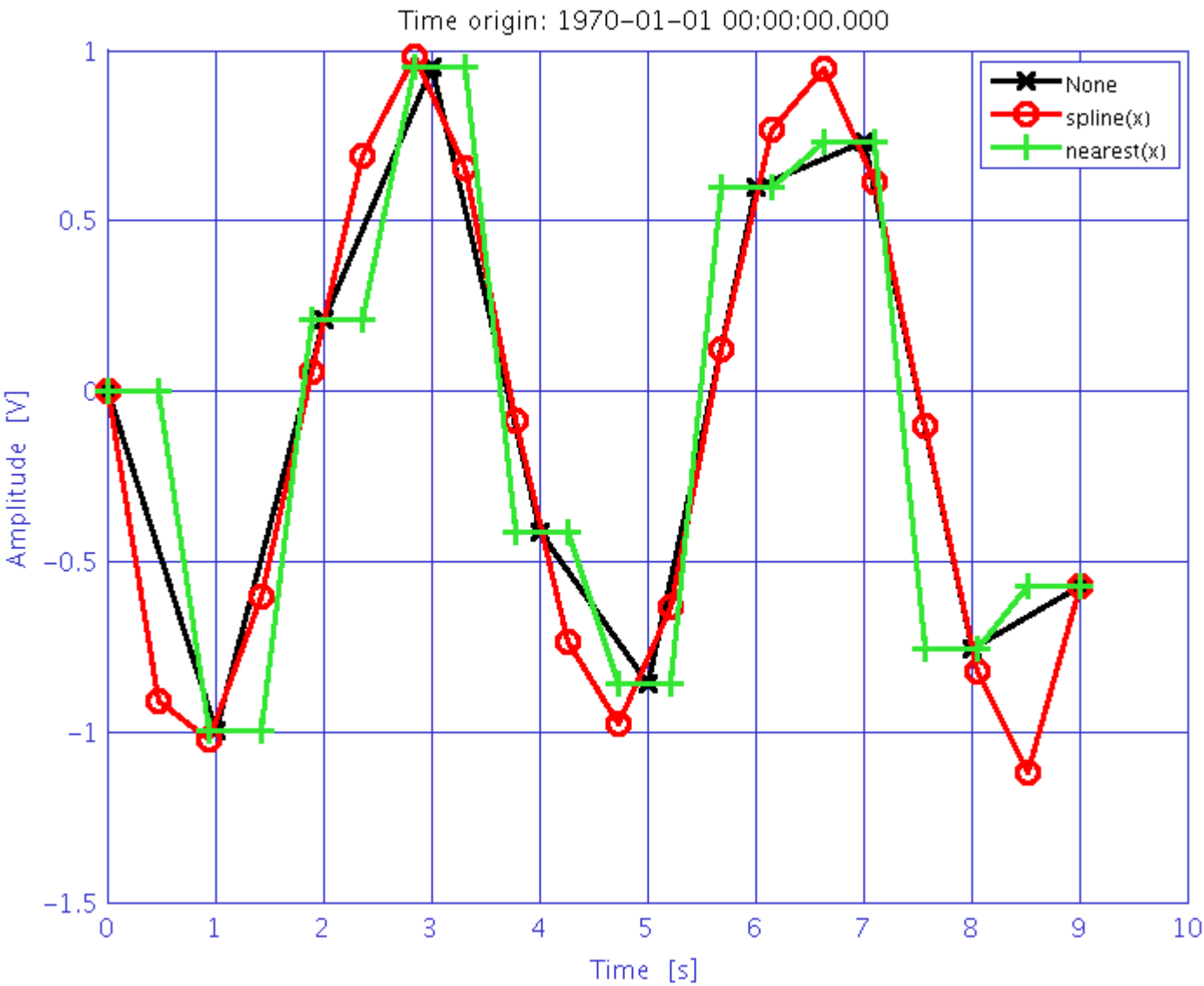
- `vertices` – specify the new vertices to interpolate on
- `method` – four methods are available for interpolating data
 - `'nearest'` – nearest neighbor interpolation
 - `'linear'` – linear interpolation
 - `'spline'` – spline interpolation (default option)
 - `'cubic'` – shape-preserving piecewise 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
pl = plist('tsfcn','sin(2*pi*1.733*t)',...
'fs',1,'nsecs',10,...
'yunits','V');
x = ao(pl);
% Interpolate on a new time vector
t = linspace(0, x.data.nsecs - 1/x.data.fs, 2*len(x));
pl_spline = plist('vertices',t);
pl_nearest = plist('vertices',t,'method','nearest');
x_spline = interp(x,pl_spline);
x_nearest = interp(x,pl_nearest);
ipplot([x x_spline x_nearest], plist('Markers', {'x', 'o', '+'}, ...
'LineColors', {'k', 'r'}));
```



◀ Resampling data

Spikes reduction in data ▶

Spikes reduction in data

Description

Spikes in data due to different nature can be removed, if desired, from the original data. 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.

Syntax

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

Parameters

The following parameters can be set in this method:

- '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)

Algorithm

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.

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

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

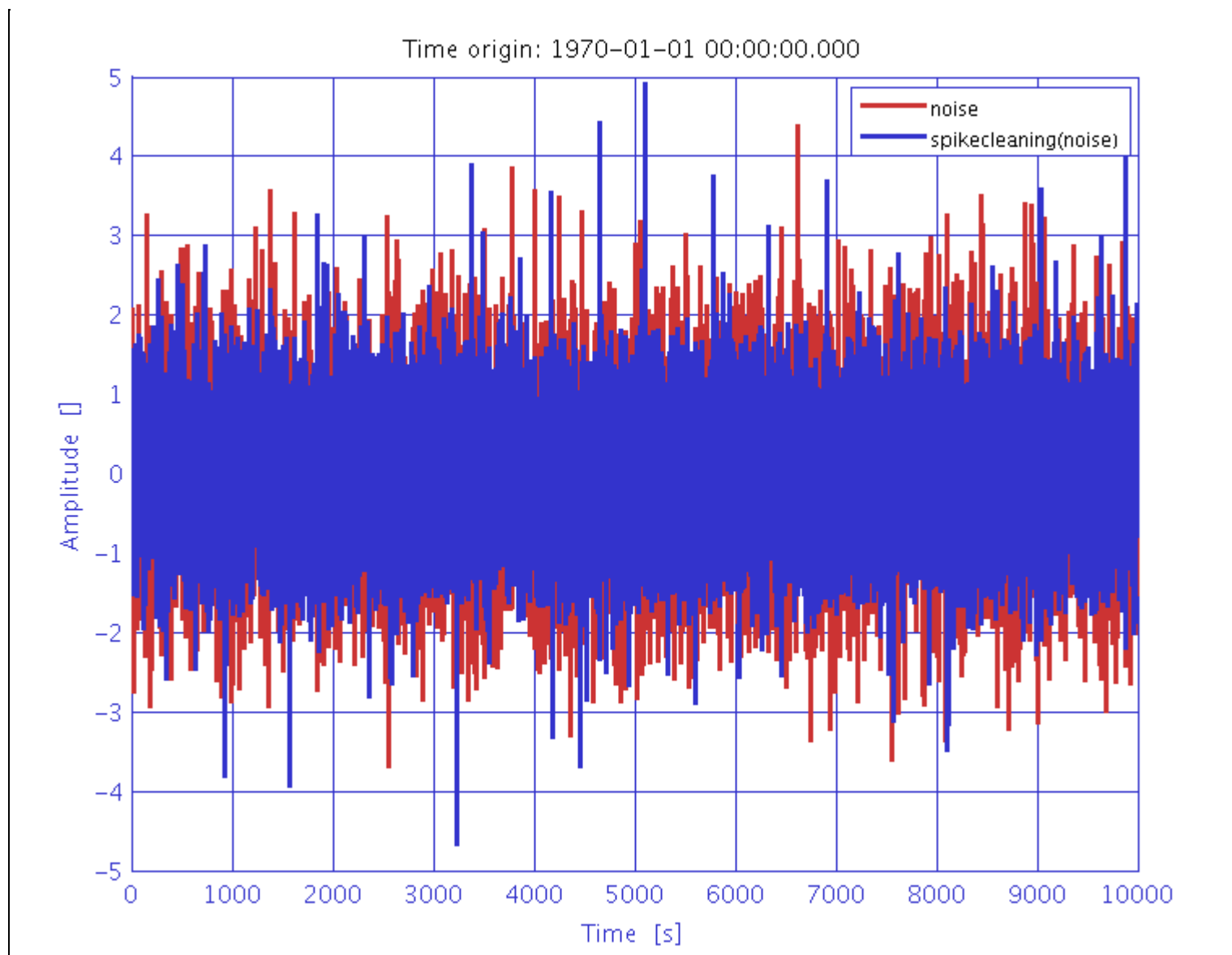
Previous: the spike is substituted by the previous sample, i.e.:

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

Examples

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

```
x = ao(plist( 'waveform', 'noise', 'nsecs',1e4)); % create an AO of random data
sampled at 1 Hz.
pl = plist( 'kspike', 2); % kspike = 2
y = spikecleaning(x, pl); % spike cleaning function applied to
the input AO, x
ipplot(x, y) % plot original and "cleaned" data
```



◀ Interpolating data

Data gap filling ▶

©LTP Team

Data gap filling

Description

Gaps in data can be filled with *interpolated* data if desired. LTPDA [gapfilling](#) joins 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.

Syntax

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

where `a1` and `a2` are the two segments to join.

Parameters

The following parameters apply for this method:

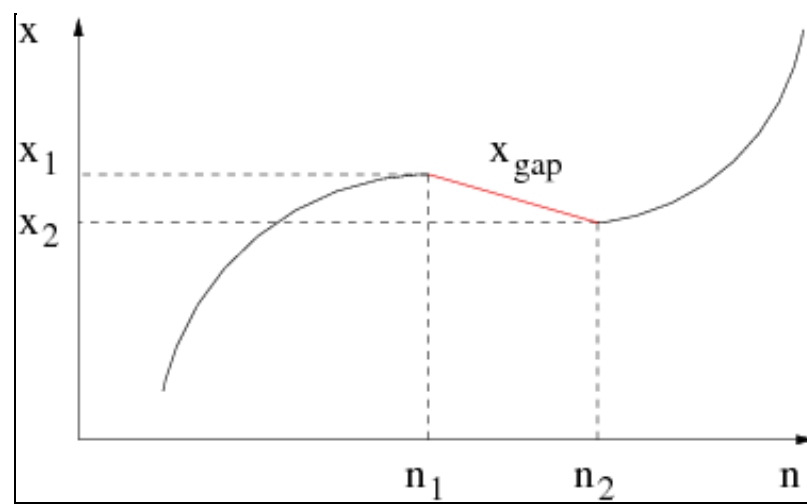
- 'method' – method used to interpolate missing data (see below for details)
 - 'linear' (default option)
 - 'spline'
- 'addnoise' – with this option *noise* can be added to the interpolated data. This noise is defined as a random variable with zero mean and variance equal to the high-frequency noise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

Algorithm

Linear : The gap is filled using a linear approximation.

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

the notation used in the previous expression is schematically explained in the figure below



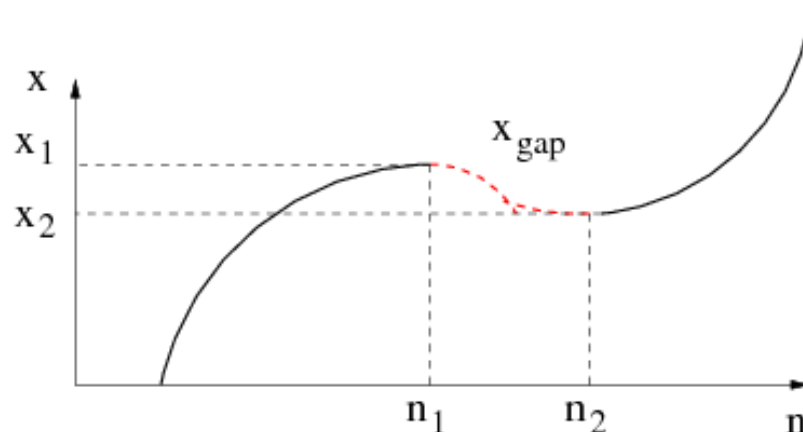
Spline : a third order interpolation is used in this case

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

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

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

The result is schematically shown in the figure below



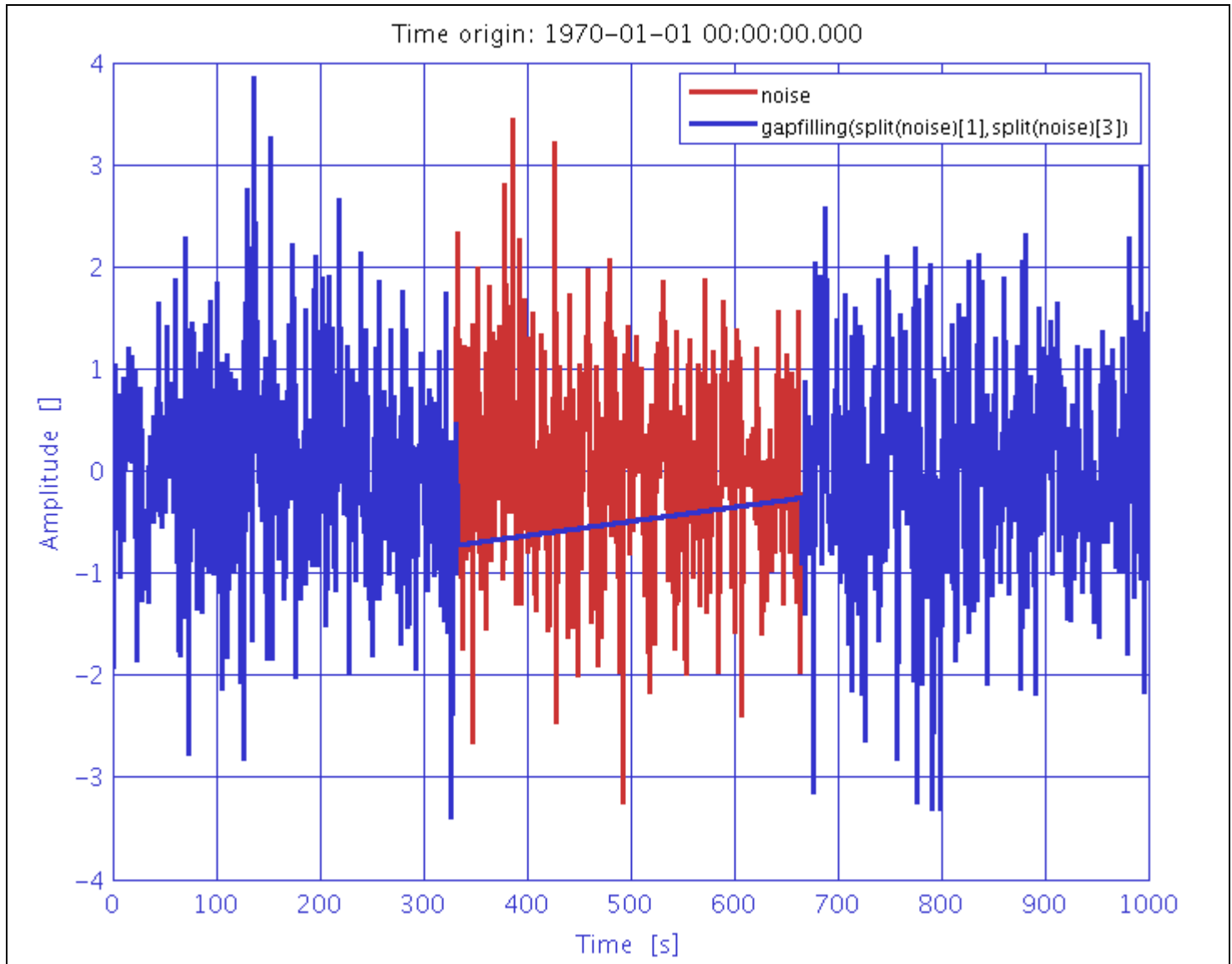
Examples

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

```

x1 = ao(plist('waveform','noise','nsecs',1e3,'name','noise')); % create an AO of random
data
xs = split(x1,plist('chunks',3)); % split in three segments
pl = plist('method','linear','addnoise','no'); % linear gapfilling
y = gapfilling(xs(1),xs(3), pl); % fill between 2nd and 3rd
segment
iplot(x1,y)

```

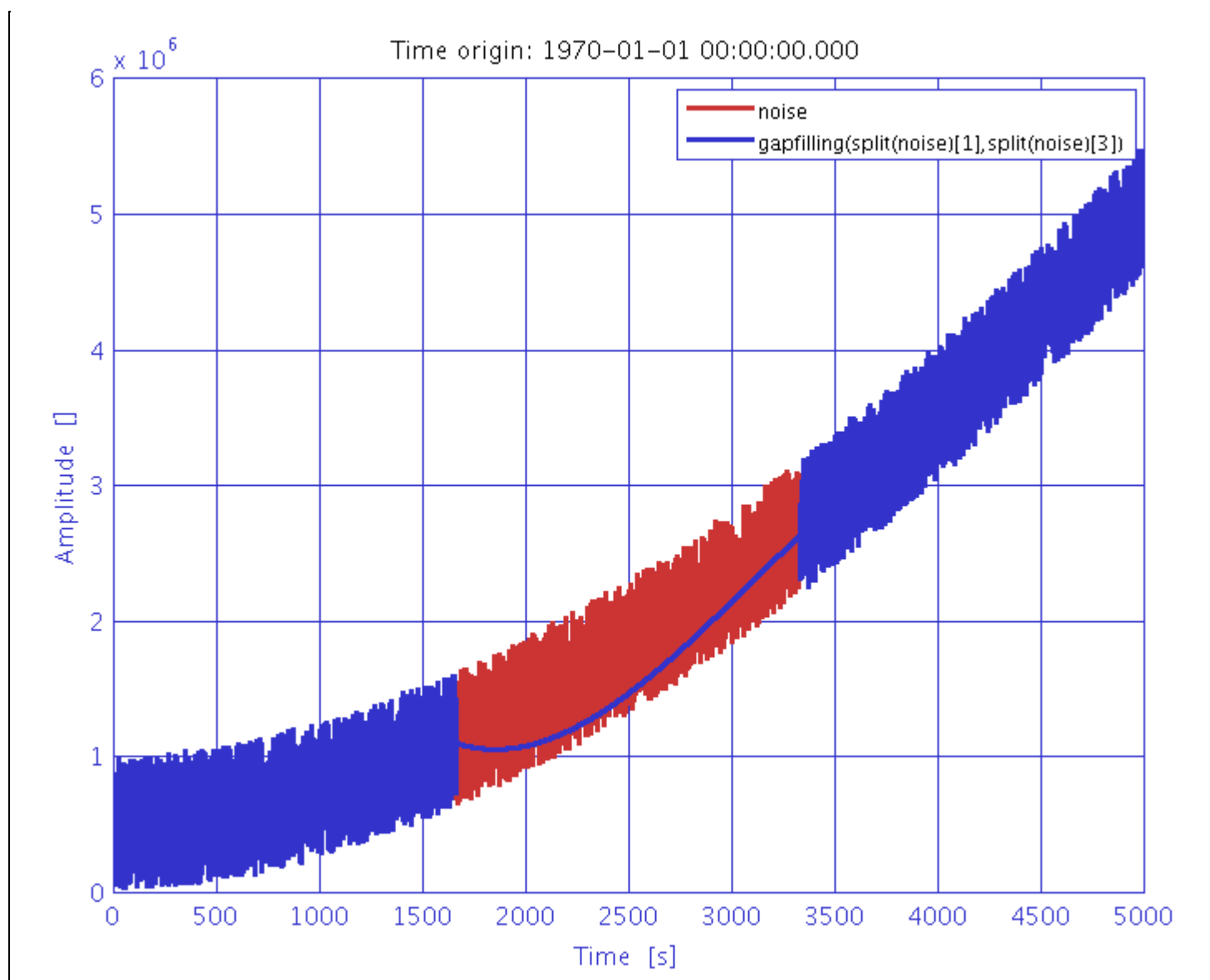


2. Missing data between two data vectors interpolated with the spline method.

```

x1 = ao(plist('tsfcn','t.^1.8 + 1e6*rand(size(t))','nsecs',5e3,'name','noise')); % create
an AO
xs = split(x1,plist('chunks',3)); % split in three segments
pl = plist('method','spline','addnoise','no'); % cubic gapfilling
y = gapfilling(xs(1),xs(3), pl); % fill between 2nd and 3rd
segment
iplot(x1,y)

```



◀ Spikes reduction in data

Noise whitening ▶

©LTP Team

Noise whitening

Introduction	Noise whitening in LTPDA.
Algorithm	Whitening Algorithms.
1D data	Whitening noise in one-dimensional data.
2D data	Whitening noise in two-dimensional data.

Noise whitening in LTPDA

A random process $w(t)$ is considered white if it is zero mean and uncorrelated:

$$\begin{aligned}\mu_w &= \mathbb{E}[w(t)] = 0 \\ C_w(t_i, t_j) &= C_0 \delta(t_i - t_j)\end{aligned}$$

As a consequence, the power spectral density of a white process is a constant at every frequency:

$$S(\omega) = C_0$$

In other words, The power per unit of frequency associated to a white noise process is uniformly distributed on the whole available frequency range. An example is reported in figure 1.

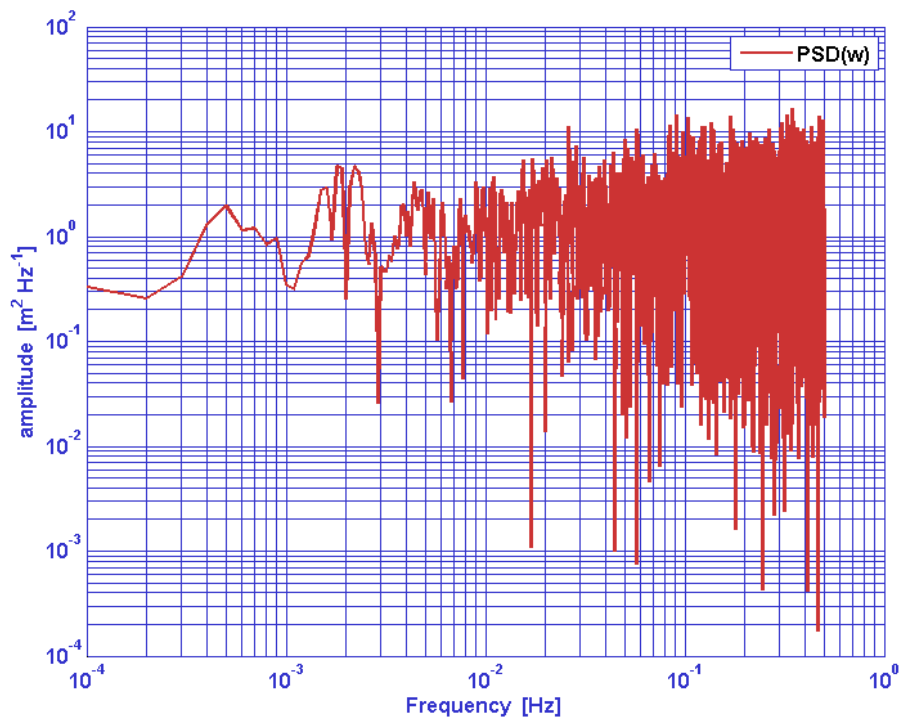


Figure 1: Power spectral density (estimated with the welch method) of a gaussian unitary variance zero mean random process. The process $w(t)$ is assumed to have physical units of m therefore its power spectral density has physical units of m^2/Hz . Note that the power spectral density average value is 2 instead of the expected 1 (unitary variance process) since we calculated one-sided power spectral density.

A non-white (colored) noise process is instead characterized by a given distribution of the power per unit of frequency along the available frequency bandwidth. Whitening operation on a given non-white process corresponds to force such a process to satisfy the conditions described above for a white process.

In LTPDA there are three different methods for noise whitening:

- [whiten1D.m](#)
- [firwhiten.m](#)
- [whiten2D.m](#)

They accept time series analysis objects as an input and they output noise whitened time series analysis objects.

Whitening Algorithms

Whiten1D

`whiten1D` performs a frequency domain identification of the system in order to extract the proper whitening filter. The function needs a model for the one-sided power spectral density of the given process. If no model is provided, the power spectral density of the process is calculated with the [lpsd](#) algorithm.

1. The inverse of the square root of the model for the power spectral density is fit in z-domain in order to determine a whitening filter.
2. Unstable poles are removed by an all-pass stabilization procedure.
3. Input data are filtered with the identified filter in order to be whitened.

Firwhiten

`firwhiten` whitens the input time-series by building an FIR whitening filter.

1. Make ASD of time-series.
2. Perform running median to get noise-floor estimate [ao/smoothier](#).
3. Invert noise-floor estimate.
4. Call [mfir0](#) on noise-floor estimate to produce whitening filter.
5. Filter data.

Whiten2D

`whiten2D` whitens cross-correlated time-series. Whitening filters are constructed by a fitting procedure to the models for the corss-spectral matrix provided. In order to work with `whiten2D` you must provide a model (frequency series analysis objects) for the cross-spectral density matrix of the process.

1. Whitening filters frequency response is calculated by the eigendecomposition of the cross-spectral matrix.
2. Calculated responses are fit in z-domain in order to identify corresponding autoregressive moving average filters.
3. Input time-series is filtered. The filtering process corresponds to:
 $w(1) = \text{Filt11}(a(1)) + \text{Filt12}(a(2))$
 $w(2) = \text{Filt21}(a(1)) + \text{Filt22}(a(2))$

Whitening noise in one-dimensional data

We can now test an example of the one-dimensional whitening filters capabilities. With the following commands we can generate a colored noise data series for parameters description please refer to the [ao](#), [miir](#) and [filter](#) documentation pages.

```
>> fs = 1; % sampling frequency
>> % Generate gaussian white noise
>> pl = plist('tsfcn', 'randn(size(t))', ...
'fs', fs, ...
'nsecs', 1e5, ...
'yunits', 'm');
>> a = ao(pl);

>> % Get a coloring filter
>> pl = plist('type', 'bandpass', ...
'fs', fs, ...
'order', 3, ...
'gain', 1, ...
'fc', [0.03 0.1]);
```

```
>> ft = miir(pl);
>> % Coloring noise
>> af = filter(a, ft);
```

Now we can try to white colored noise.

Whiten1D

If you want to try `whiten1D` to whiten the present colored noise you can try the following code. Please refer to the [whiten1D](#) documentation page for the meaning of any parameter. The result of the whitening procedure is reported in figure 2.

```
>> pl = plist(...
    'model', [], ...
    'MaxIter', 30, ...
    'PoleType', 2, ...
    'MinOrder', 9, ...
    'MaxOrder', 15, ...
    'Weights', 2, ...
    'Plot', false, ...
    'Disp', false, ...
    'MSEVARTOL', 1e-1, ...
    'FITTOL', 5e-2);
>> aw = whiten1D(af,pl);
```

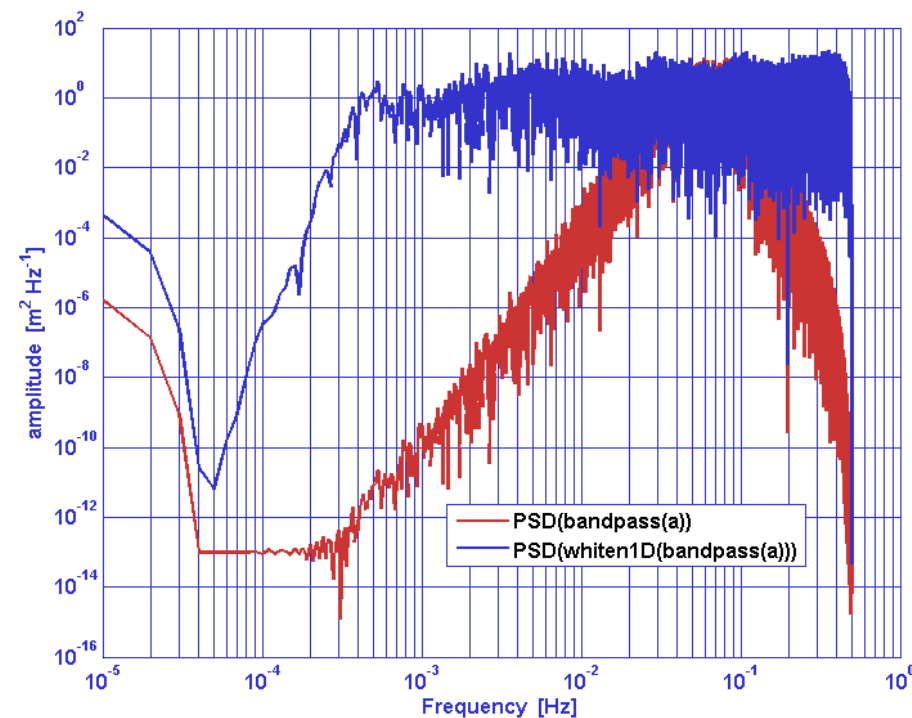


Figure 2: Power spectral density (estimated with the welch method) of colored and whitened processes.

Firwhiten

As an alternative you can try `firwhiten` to whiten the present colored noise. Please refer to the [firwhiten](#) documentation page for the meaning of any parameter. The result of the whitening procedure is reported in figure 3.

```
>> pl = plist(...
'Ntaps', 5000, ...
'Nfft', 1e5, ...
'BW', 5);

>> aw = firwhiten(af, pl);
```

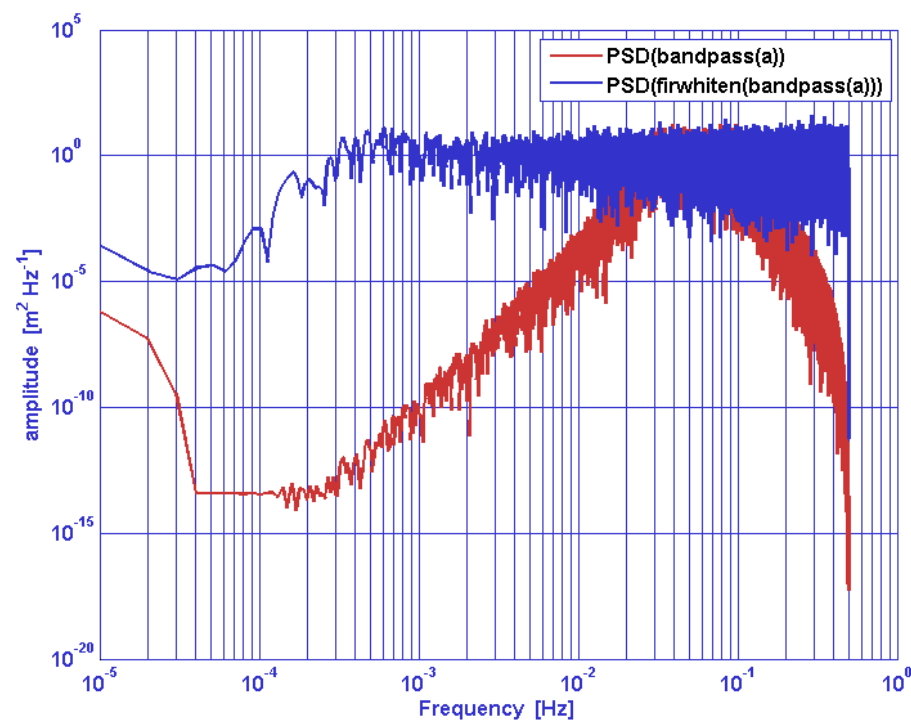


Figure 3: Power spectral density (estimated with the welch method) of colored and whitened processes.

Whitening noise in two-dimensional data

We consider now the problem of whitening cross correlated data series. As a example we consider a typical couple of x-dynamics LTP data series. a_1 and a_2 are interferometer output noise data series. In order to whiten data we must input a frequency response model of the cross spectral matrix of the cross-correlated process.

$$CSD(f) = \begin{pmatrix} csd_{11}(f) & csd_{12}(f) \\ csd_{21}(f) & csd_{22}(f) \end{pmatrix}$$

Refer to [whiten2D](#) documentation page for the meaning of any parameter.

```
>> pl = plist(...
'csd11', mod11, ...
'csd12', mod12, ...
'csd21', mod21, ...
'csd22', mod22, ...
'MaxIter', 75, ...
'PoleType', 3, ...
'MinOrder', 20, ...
'MaxOrder', 40, ...
'Weights', 2, ...
'Plot', false, ...
'Disp', false, ...)
```

```

'MSEVARTOL', 1e-2, ...
'FITTOLE', 1e-3);

>> [aw1,aw2] = whiten2D(a1,a2,p1);

```

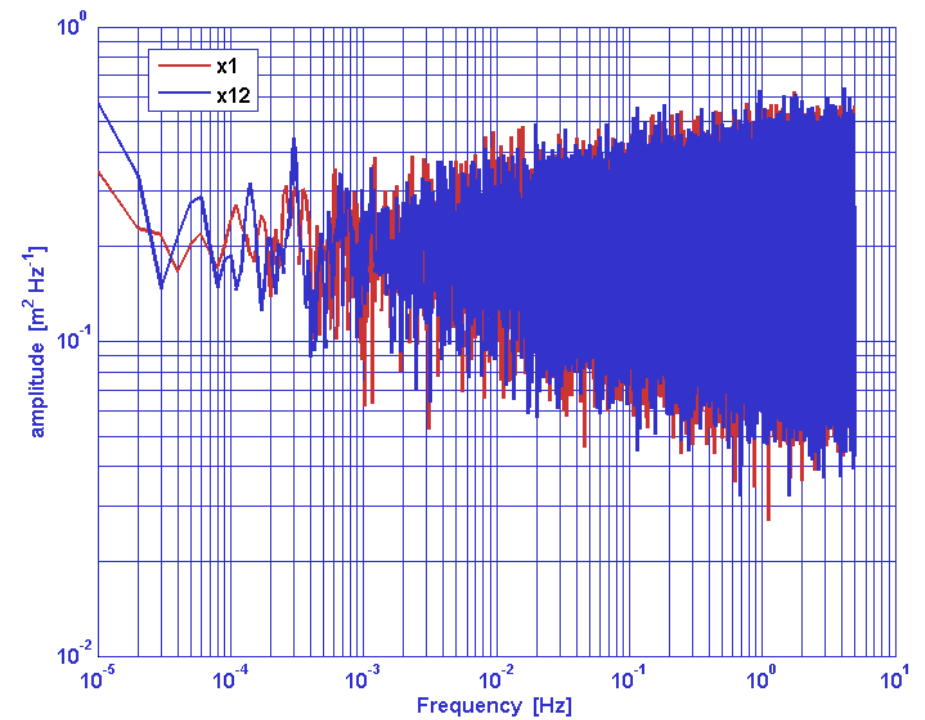
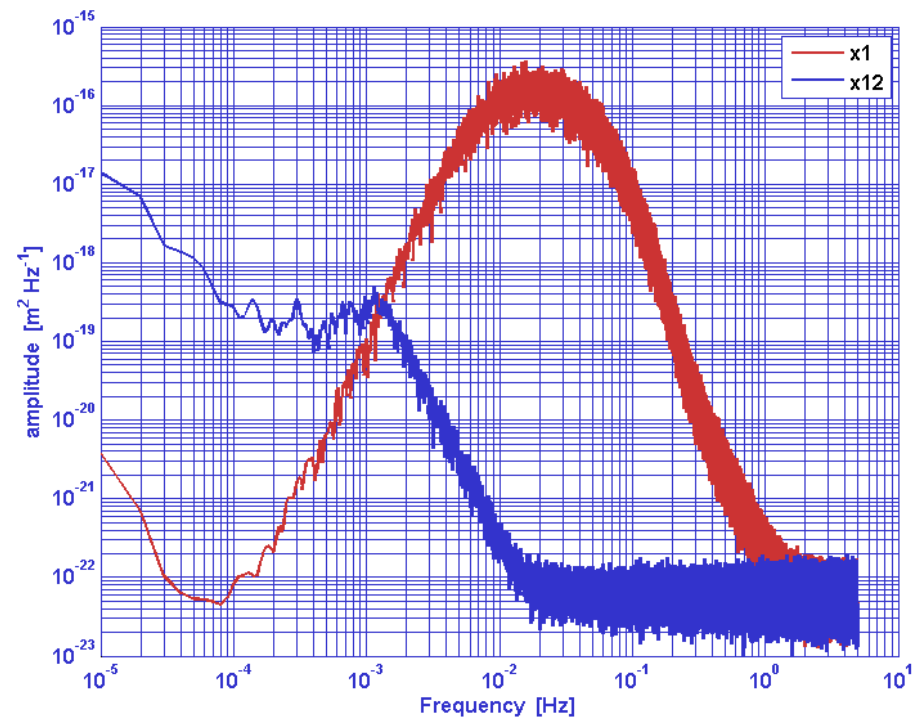


Figure 4: Power spectral density of the noisy data series before (left) and after (right) the whitening.

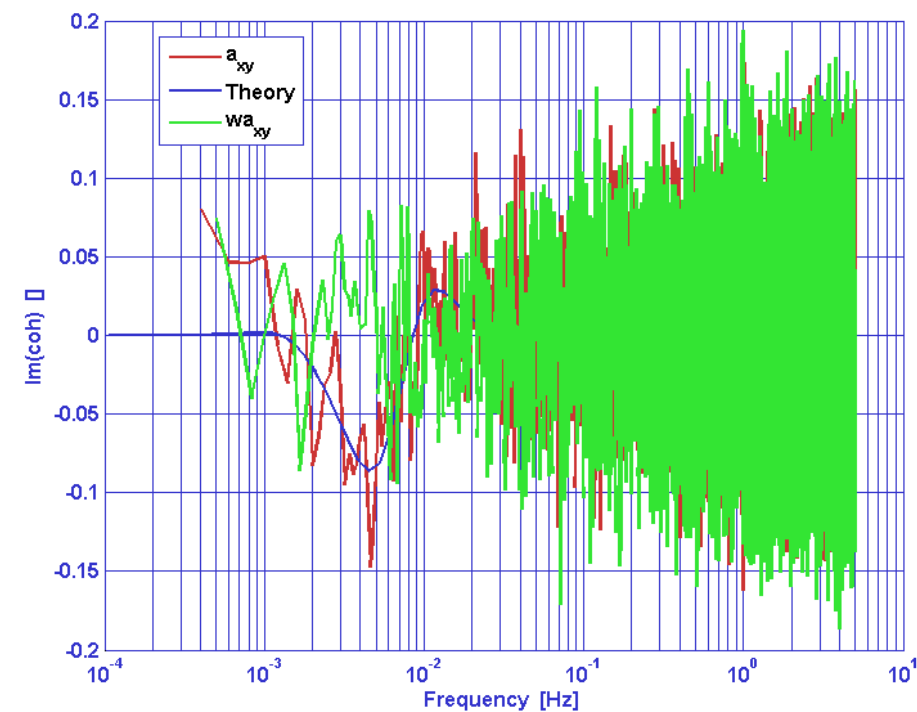
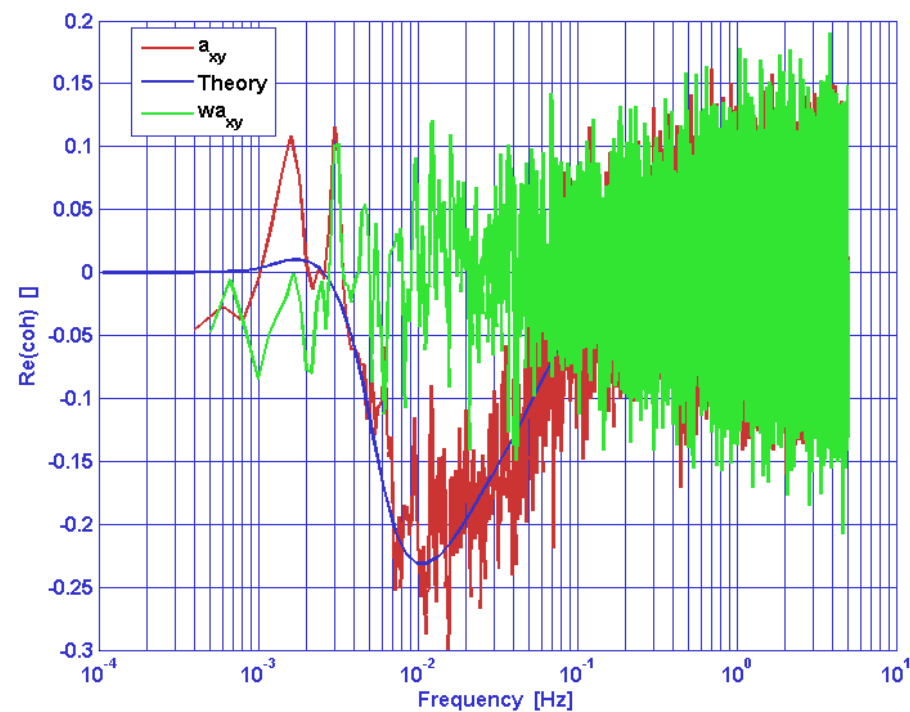


Figure 5: Real (left) and Imaginary (right) part of the [coherence](#) function. Blue line refers to theoretical expectation for colored noise data. Red line refers to calculated values for colored noise data. Green line refers to calculated values for whitened noise data.

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](#)
- [Discrete Derivative](#)
- [Spectral estimation](#)
- [Fitting algorithms](#)

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:

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

AR processes can be also classified as [IIR Filters](#).

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

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

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:

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

◀ Signal Processing in LTPDA

IIR Filters ▶

©LTP Team

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 convention 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 ▶](#)

©LTP Team

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. The 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:

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

```
>> 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

Filter banks ▶

©LTP Team

Filter banks

Description	Constructor description.
Create	Creating a filter bank in the LTPDA.
From plist	Creating from a plist.
From filters	Creating from filter objects.
Import	Importing an existing filterbank.

Description

A filter bank is a container object for collections of [MIIR](#) and [MFIR](#) filters. The filter bank object can be used to filter a discrete data series in parallel or in series, in accordance to the schematic representations of figures 1 and 2.

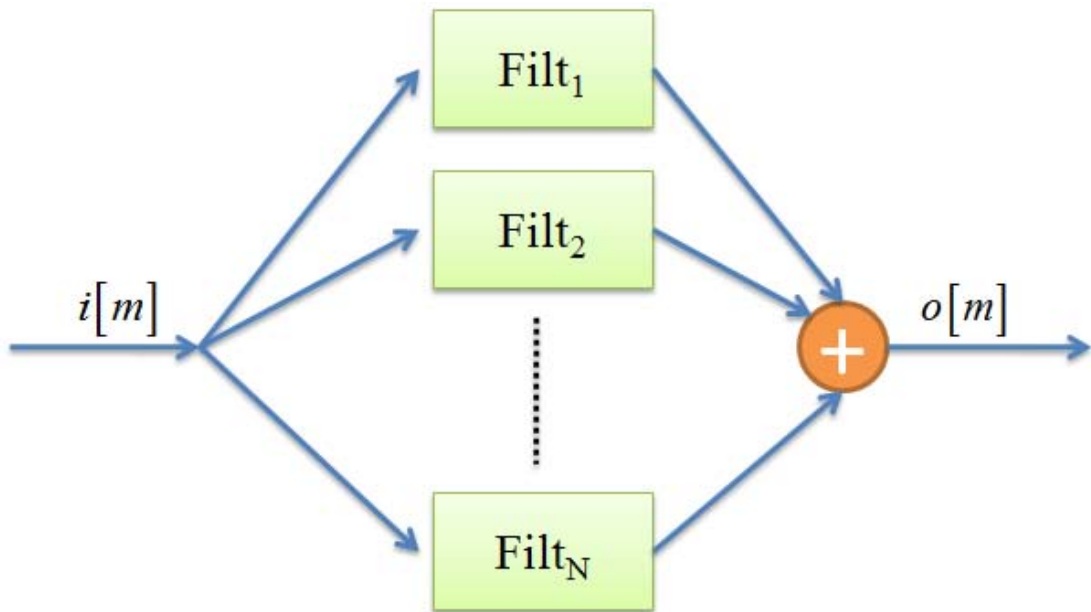


Figure 1: Scheme for parallel filtering. Data is input to the different filters of the bank. Filters output are summed each other in order to obtain the complete output. The different filters of the bank must have the same input and output units.



Figure 2: Scheme for serial filtering. The output of each filter is input to the following filter till the end of the series.

Creating a filter bank in the LTPDA

The LTPDA Toolbox allows the implementation of filter banks by means of the [filterbank class](#).

Creating from a plist

The following example creates a parallel filter bank of [MIIR](#) filters.

```
>> iirhp = miir(plist('type', 'highpass'));
>> iirlp = miir(plist('type', 'lowpass'));
>> pl = plist('filters', [iirhp iirlp], ...
'type', 'parallel');
>> f = filterbank(pl)
```

The following example creates a serial filter bank of [MIIR](#) filters.

```
>> iirhp = miir(plist('type', 'highpass'));
>> iirlp = miir(plist('type', 'lowpass'));
>> pl = plist('filters', [iirhp iirlp], ...
'type', 'series');
>> f = filterbank(pl)
```

Creating from filter objects

The following example creates a parallel filter bank of [MIIR](#) filters.

```
>> iirhp = miir(plist('type', 'highpass'));
>> iirlp = miir(plist('type', 'lowpass'));
>> pl = plist('type', 'parallel');
>> f = filterbank(iirhp, iirlp, pl)
```

The following example creates a serial filter bank of [MIIR](#) filters.

```
>> iirhp = miir(plist('type', 'highpass'));
>> iirlp = miir(plist('type', 'lowpass'));
>> pl = plist('type', 'series');
>> f = filterbank(iirhp, iirlp, pl)
```

Importing an existing model

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

-
- XML files:

```
>> f = filterbank('foo_filterbank.xml')
```

- MAT files:

```
>> f = filterbank('foo_filterbank.mat')
```

- From repository:

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

Applying digital filters to data

Description	Digital filtering with LTPDA filters objects.
Examples	How to filter a data series in LTPDA.

Description

Digital filtering in LTPDA can be performed by a call to the `ao/filter` method. `ao/filter` method is a wrapper of the standard MATLAB filter method. Details of the core algorithm can be found in the proper [documentation page](#).

Examples

Given an input analysis object (can be a time series or a frequency series object) `a_in`, digital filtering operation can be easily performed:

```
>> a_out = filter(a_in,fobj)
```

Where `fobj` is a filter. It can be a [MIIR](#), a [MEIR](#) or a [FILTERBANK](#) object.

The filter object `fobj` can be also embedded in a `plist`

```
>> pl = plist('filter', fobj);
>> a_out = filter(a_in,pl)
```


Discrete Derivative

Description	Discrete derivative estimation in LTPDA.
Algorithm	Derivatives Algorithms.
Examples	Usage examples of discrete derivative estimation tools.
References	Bibliographic references.

Derivative calculation for discrete data series

Derivative estimation on discrete data series is implemented by the function [ao/diff](#). This function embeds several algorithms for the calculation of zero, first and second order derivative. Where with zero order derivative we intend a particular category of data smoothers [1].

Algorithm

Method	Description
'2POINT'	Compute first derivative with two point equation according to: $\frac{dy[k]}{dx} \approx \frac{y[k+1] - y[k]}{x[k+1] - x[k]}$
'3POINT'	Compute first derivative with three point equation according to: $\frac{dy[k]}{dx} \approx \frac{y[k+1] - y[k-1]}{x[k+1] - x[k-1]}$
'5POINT'	Compute first derivative with five point equation according to: $\frac{dy[k]}{dx} \approx \frac{-y[k+2] + 8y[k+1] - 8y[k-1] + y[k-2]}{3(x[k+2] - x[k-2])}$
'FPS'	Five Point Stencil is a generalized method to calculate zero, first and second order discrete derivative of a given time series. Derivative approximation, at a given time $t = kT$ (k being an integer and T being the sampling time), is calculated by means of finite differences between the element at t with its four neighbors: $y[k] \approx ay[k-2] + by[k-1] + cy[k] + dy[k+1] + gy[k+2]$ $\frac{dy[k]}{dt} \approx \frac{a'y[k-2] + b'y[k-1] + c'y[k] + d'y[k+1] + g'y[k+2]}{T}$ $\frac{d^2y[k]}{dt^2} \approx \frac{a''y[k-2] + b''y[k-1] + c''y[k] + d''y[k+1] + g''y[k+2]}{T^2}$

It can be demonstrated that the coefficients of the expansion can be expressed as a function of one of them [1]. This allows the construction of a family of discrete derivative estimators characterized by a good low frequency accuracy and a smoothing behavior at high frequencies (near the nyquist frequency). Non-trivial values for the '**COEFF**' parameter are:

- Parabolic fit approximation
These coefficients can be obtained by a parabolic fit procedure on a generic set of data [1].
 - Zeroth order $-3/35$
 - First order $-1/5$
 - Second order $2/7$
- Taylor series expansion
These coefficients can be obtained by a series expansion of a generic set of data [1 – 3].
 - First order $1/12$
 - Second order $-1/12$
- PI
This coefficient allows to define a second derivative estimator with a notch feature at the nyquist frequency [1].
 - Second order $1/4$

Examples

Consider *a* as a time series analysis object. First and second derivative of *a* can be easily obtained with a call to [diff](#). Please refer to [ao/diff](#) documentation page for the meaning of any parameter. Frequency response of first and second order estimators is reported in figures 1 and 2 respectively.

First derivative

```
>> pl = plist(...
'method', '2POINT');
>> b = diff(a, pl);

>> pl = plist(...
'method', 'ORDER2SMOOTH');
>> c = diff(a, pl);

>> pl = plist(...
'method', '3POINT');
>> d = diff(a, pl);

>> pl = plist(...
'method', '5POINT');
>> e = diff(a, pl);

>> pl = plist(...
'method', 'FPS', ...
'ORDER', 'FIRST', ...
'COEFF', -1/5);
>> f = diff(a, pl);
```

Second derivative

```
>> pl = plist(...
'method', 'FPS', ...
'ORDER', 'SECOND', ...
'COEFF', 2/7);
>> b = diff(a, pl);

>> pl = plist(...
'method', 'FPS', ...
'ORDER', 'SECOND', ...
'COEFF', -1/12);
>> c = diff(a, pl);
```

```
>> pl = plist(...
'method', 'FPS', ...
'ORDER', 'SECOND', ...
'COEFF', 1/4);
>> d = diff(a, pl);
```

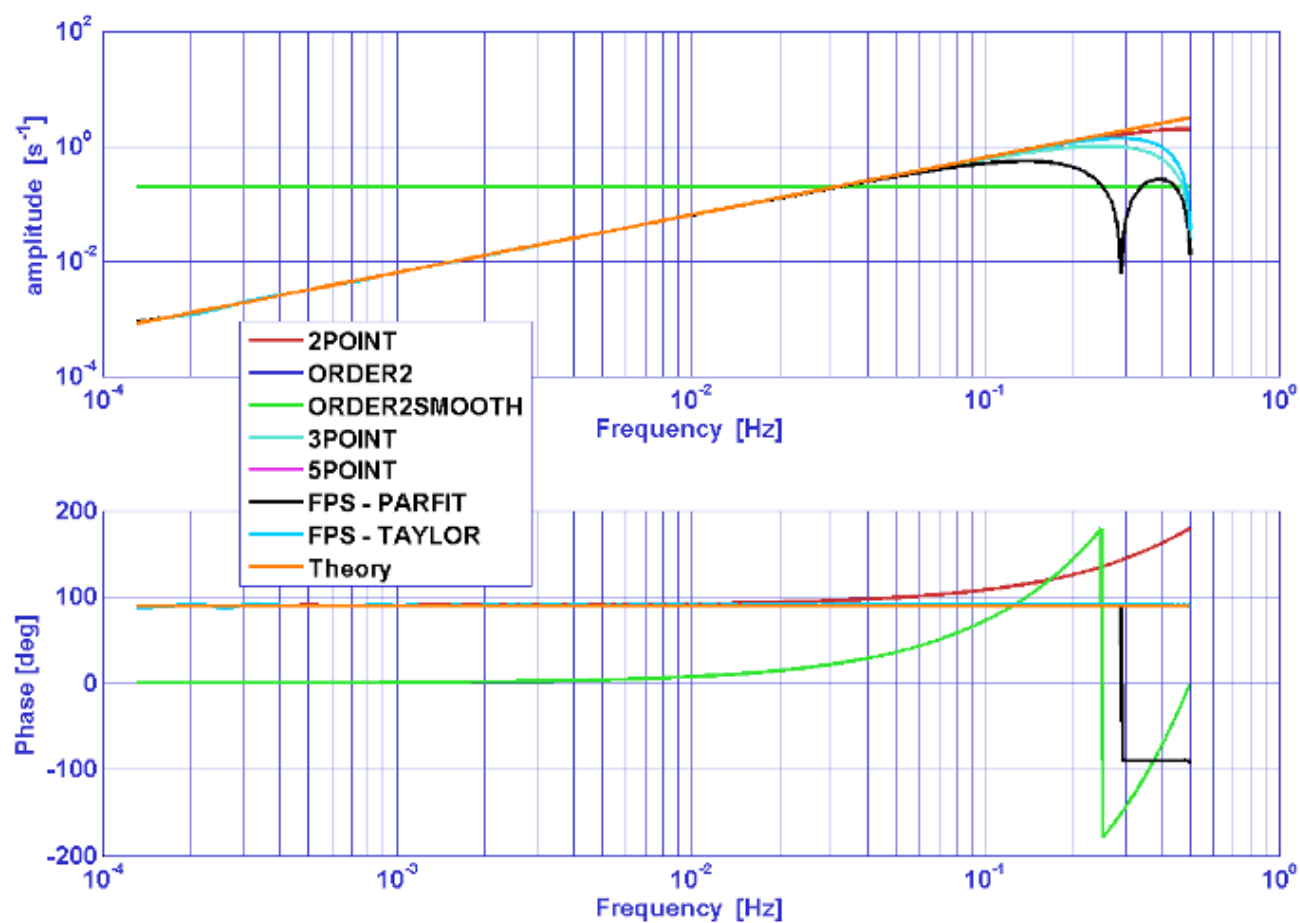


Figure 1: Frequency response of first derivative estimators.

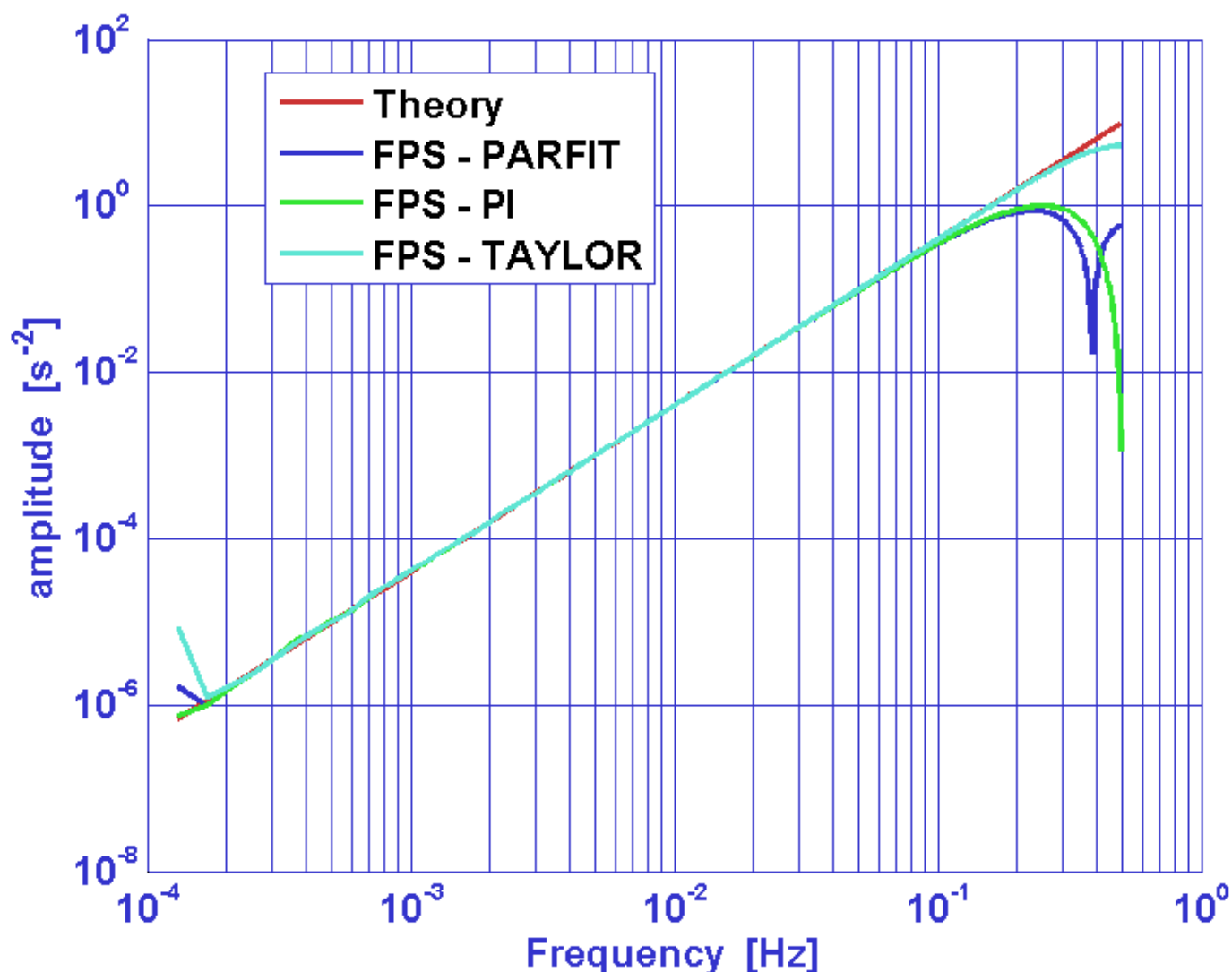


Figure 2: Frequency response of second derivative estimators.

References

1. L. Ferraioli, M. Hueller and S. Vitale, Discrete derivative estimation in LISA Pathfinder data reduction, [Class. Quantum Grav. 26 \(2009\) 094013.](#)
L. Ferraioli, M. Hueller and S. Vitale, Discrete derivative estimation in LISA Pathfinder data reduction [arXiv:0903.0324v1](#)
2. Steven E. Koonin and Dawn C. Meredith, Computational Physics, Westview Press (1990).
3. John H. Mathews, Computer derivations of numerical differentiation formulae, *Int. J. Math. Educ. Sci. Technol.*, 34:2, 280 – 287.

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.

More detailed help on spectral estimation can also be found in the help associated with the Signal Processing Toolbox:

>> doc signal

 Discrete Derivative

Introduction 

©LTP Team

Introduction

An introduction to spectral estimators can be found directly inside Matlab documentation regarding the Signal Processing Toolbox. More detailed description can be found typing:

```
>> doc signal
```

in the Matlab terminal.

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](#)

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 PSL.

 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 example shows the use of `ao/psd` to estimate an Amplitude Spectral Density of the time-series captured in the input AO, `a`. The help for [ao/lpsd](#) reveals that the required parameter for setting the window function is `'win'`.

```
% Parameters
nsecs = 1000;
fs = 10;

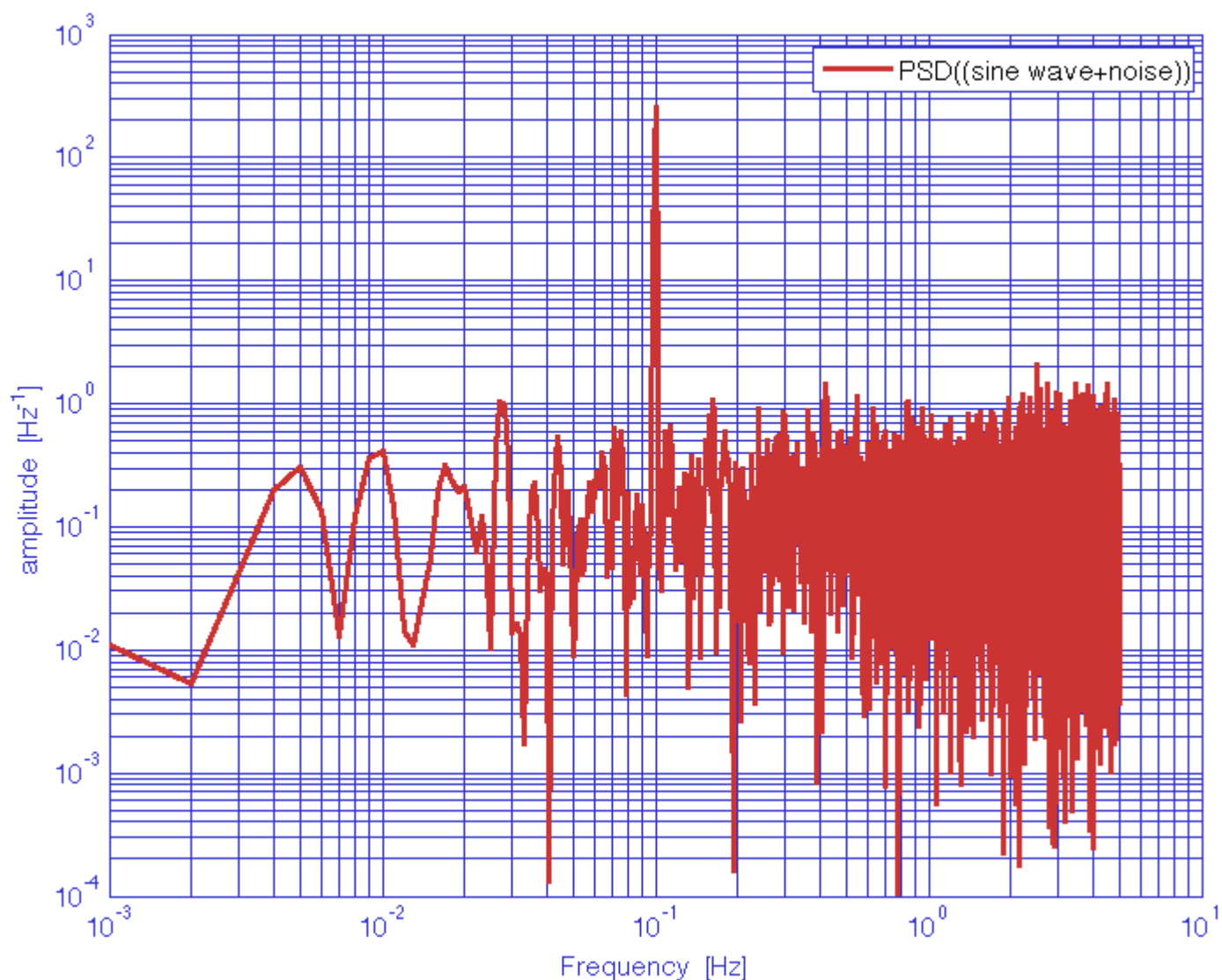
% Create input AOs
x1 = ao(plist( 'waveform', 'sine wave', 'f', 0.1, 'A', 1, 'nsecs', nsecs, 'fs', fs));
x2 = ao(plist( 'waveform', 'noise', 'type', 'normal', 'nsecs', nsecs, 'fs', fs));

% Add both
x = x1 + x2;

% Compute psd with Blackman-Harris window
z = psd(x,plist('win', 'BH92'));

% Plot
ipplot(z);
```

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 `psd` algorithm then recomputes the window using the input design but for the correct length of window function.



Selecting the Peak Side-Lobe level (psll) with Kaiser's window

The [table](#) in the previous section shows how each standard spectral window is defined by the Peak Side-Lobe level (psll). However, Kaiser's window allows the user to define the psll of the window.

The following example shows the importance of selecting a suitable psll according to each application. The example creates 1/f noise (in fact, noise generated by a pole-zero model with a pole at low frequencies) and computes the Amplitude Spectrum Density (ASD) with three different psll values. The ASD with the lowest value shows a bias at high frequencies compared with the response of the pzmodel used to generate the data (in black). This effect is due to the power added by the high order lobes of the window. The ASD with the highest value of the psll adds a feature at low frequencies because the main lobe of the window is too wide. Only the middle value gives an estimation of the ASD without adding window related features.

```
% Parameters
nsecs = 10000;
fs = 1;

% Create pzmodel with a low frequency pole
pzm = pzmodel(1e5,[1e-7,0.1],[]);

% Build (nearly) 1/f noise
x = ao(plist('pzmodel',pzm, 'nsecs',nsecs, 'fs',fs));

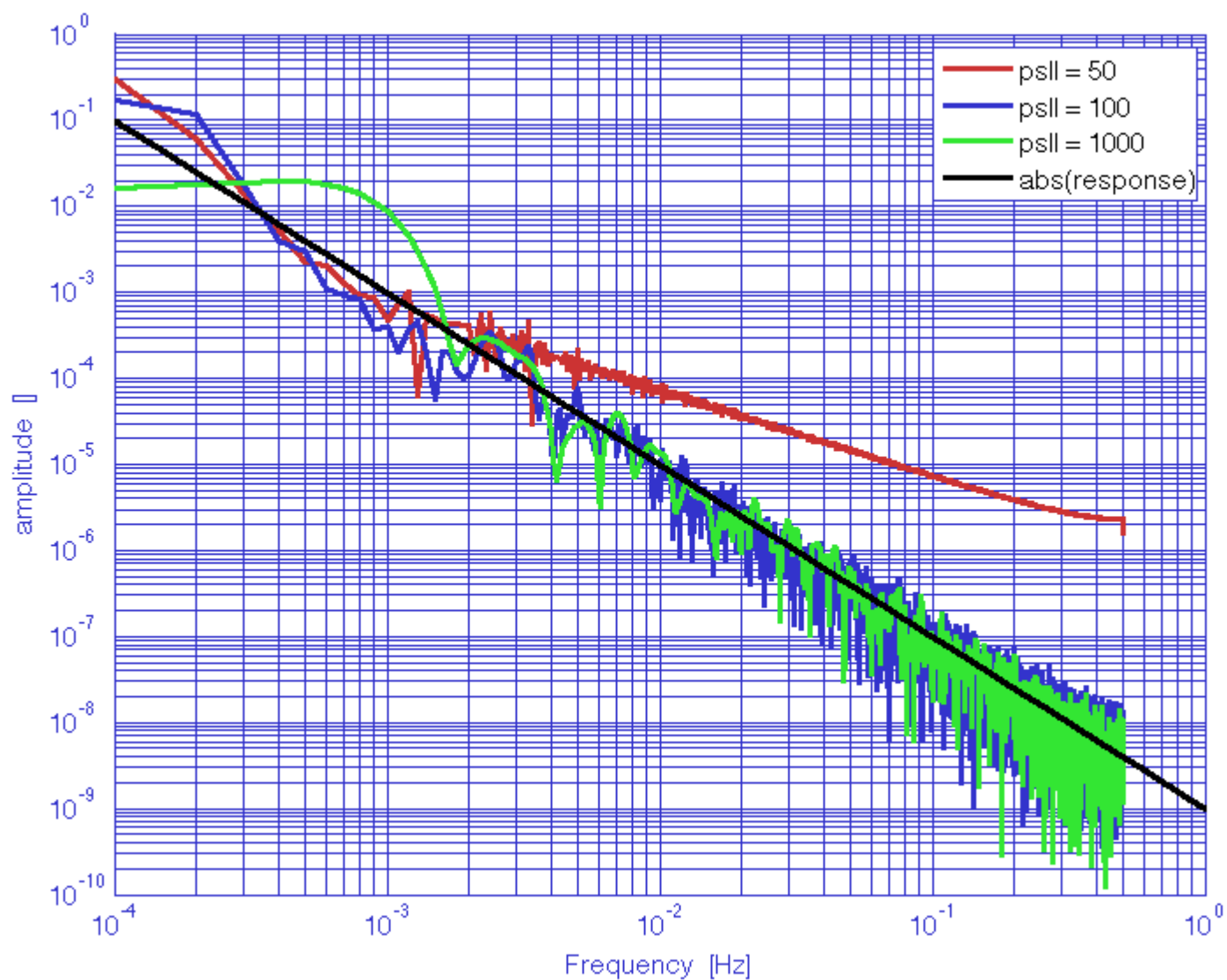
% Compute psd with Blackman-Harris window
z1 = psd(x,plist('scale','ASD','win','Kaiser','psll',50));
```

```

z1.setName('psll = 50');
z2 = psd(x,plist('scale','ASD','win','Kaiser','psll',100));
z2.setName('psll = 100');
z3 = psd(x,plist('scale','ASD','win','Kaiser','psll',1000));
z3.setName('psll = 1000');

% Plot
r = resp(pzm,plist('f1',1e-4,'f2',1));
r.setName('response');
r.setPlotinfo(plist('color','k'));
ipplot(z1,z2,z3,abs(r));

```



◀ What are LTPDA spectral windows?

Spectral Estimation Methods ▶

©LTP Team

Spectral Estimation Methods

Linear and Log-scale Methods

The LTPDA Toolbox offers two kind of spectral estimators. The first ones are based on `pwelch` from MATLAB, which is an implementation of Welch's averaged, modified periodogram method [1]. More details about spectral estimation techniques can be found [here](#).

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

- [power spectral density estimates](#)
- [cross-spectral density estimates](#)
- [cross-coherence estimates](#)
- [transfer function estimates](#)

As an alternative, the LTPDA toolbox makes available the same set of estimators, based on an implementation of the LPSD algorithm [2]).

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

- [log-scale power spectral density estimates](#)
- [log-scale cross-spectral density estimates](#)
- [log-scale cross-coherence estimates](#)
- [log-scale transfer function estimates](#)

More detailed help on spectral estimation can also be found in the help associated with the [Signal Processing Toolbox](#).

Computing the sample variance

The spectral estimators previously described usually return the average of the spectral estimator applied to different segments. This is a standard technique used in spectral analysis to reduce the variance of the estimator.

When using one of the previous methods in the LTPDA Toolbox, the value of this average over different segments is stored in the `ao.y` field of the output analysis object, but the user obtains also information about the spectral estimator variance in the `ao.dy` field.

The methods listed above store in the `ao.dy` field the **sample variance**, defined as

$$\sigma = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (X_i - \langle X \rangle)^2}$$

For more details on how the sample variance is computed, please refer to the the help page of each method.

Note that when we only have one segment we can not evaluate the variance. This will happen in

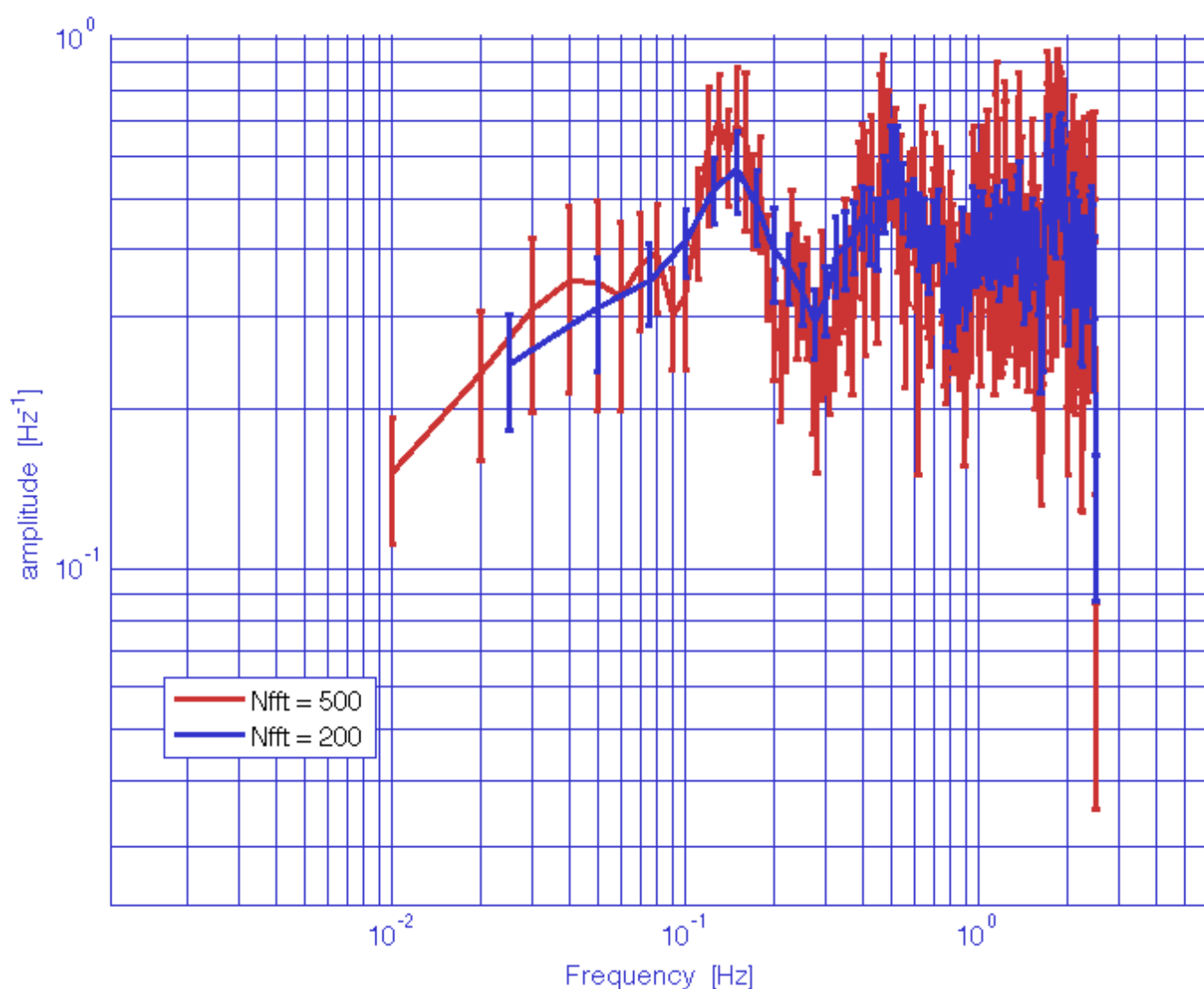
- linear estimators: when the number of averages is equal to one.
- log-scale estimators: in the lowest frequency bins.

The following example compares the sample variance computed by `ao/psd` with two different segment length.

```
% create white noise A0
pl = plist('nsecs', 500, 'fs', 5, 'tsfcn', 'randn(size(t))');
a = ao(pl);

% compute psd with different Nfft
b1 = psd(a, plist('Nfft', 500));
b1.setName('Nfft = 500');
b2 = psd(a, plist('Nfft', 200));
b2.setName('Nfft = 200');

% plot with errorbars
ipplot(b1,b2,plist('YErrU',{b1.dy,b2.dy}))
```



References

1. P.D. Welch, The Use of Fast Fourier Transform for the Estimation of Power Spectra: A

Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Trans. on Audio and Electroacoustics*, Vol. 15, No. 2 (1967), pp. 70 – 73

2. M. Troebs, G. Heinzel, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, [*Measurement*, Vol. 39 \(2006\), pp. 120 – 129.](#)

◀ Using spectral windows

Power spectral density estimates ▶

©LTP Team

Power spectral density estimates

Description

The LTPDA method [ao/psd](#) estimates the power spectral density of time-series signals, included in the input `aos` following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectrum, by multiplying it with a [spectral window object](#), and can be detrended by a polynomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lengths 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).

Syntax

```
bs = psd(a1, a2, a3, ..., pl)
bs = psd(as, pl)
bs = as.psd(pl)
```

`a1`, `a2`, `a3`, ... are `ao(s)` containing the input time series to be evaluated. `bs` includes the output object(s) and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] A string value containing the variable `'fs'` can also be used, e.g., `plist('Nfft', '2*fs')`
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- `'Olap'` – segment percent overlap [default: –1, (taken from window function)]
- `'Scale'` – scaling of output. Choose from:
 - `'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
- `'Navs'` – number of averages. If set, and if `Nfft` was set to 0 or –1, the number of points for each window will be calculated to match the request. [default: –1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.

The length of the window is set by the value of the parameter 'Nfft', so that the window is actually built using only the key features of the window: the name and, for Kaiser windows, the psll.

As an alternative to setting the number of points 'Nfft' in each window, it's possible to ask for a given number of PSD estimates by setting the 'Navs' parameter, and the algorithm takes care of calculating the correct window length, according to the amount of overlap between subsequent segments.

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

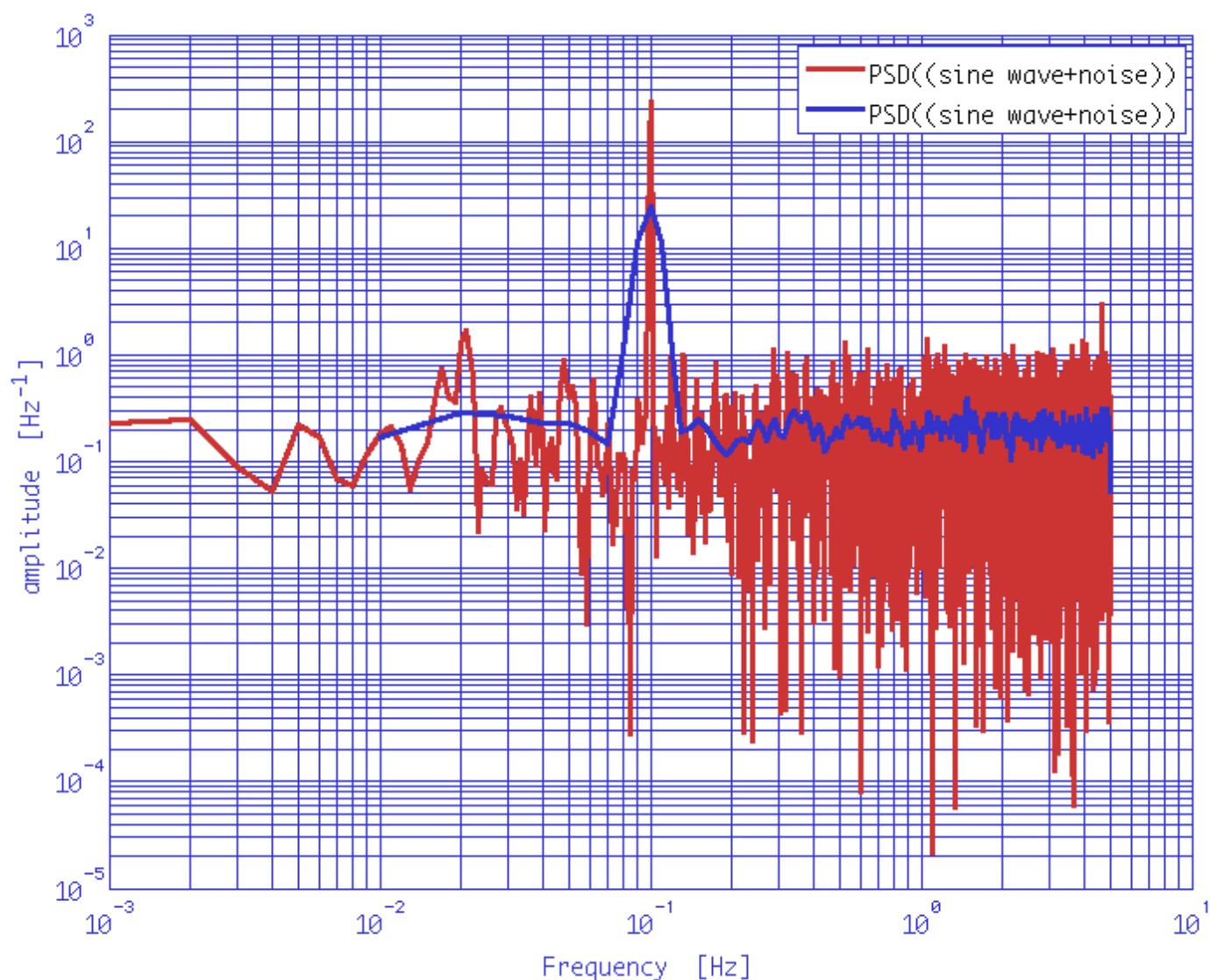
Algorithm

The algorithm is based in standard MATLAB's tools, as the ones used by [pwelch](#). However, in order to compute the sample variance of the PSD for each frequency bin, the averaging of the different segments is performed using Welford's algorithm [\[2\]](#) which allows to compute mean and variance in one loop.

Examples

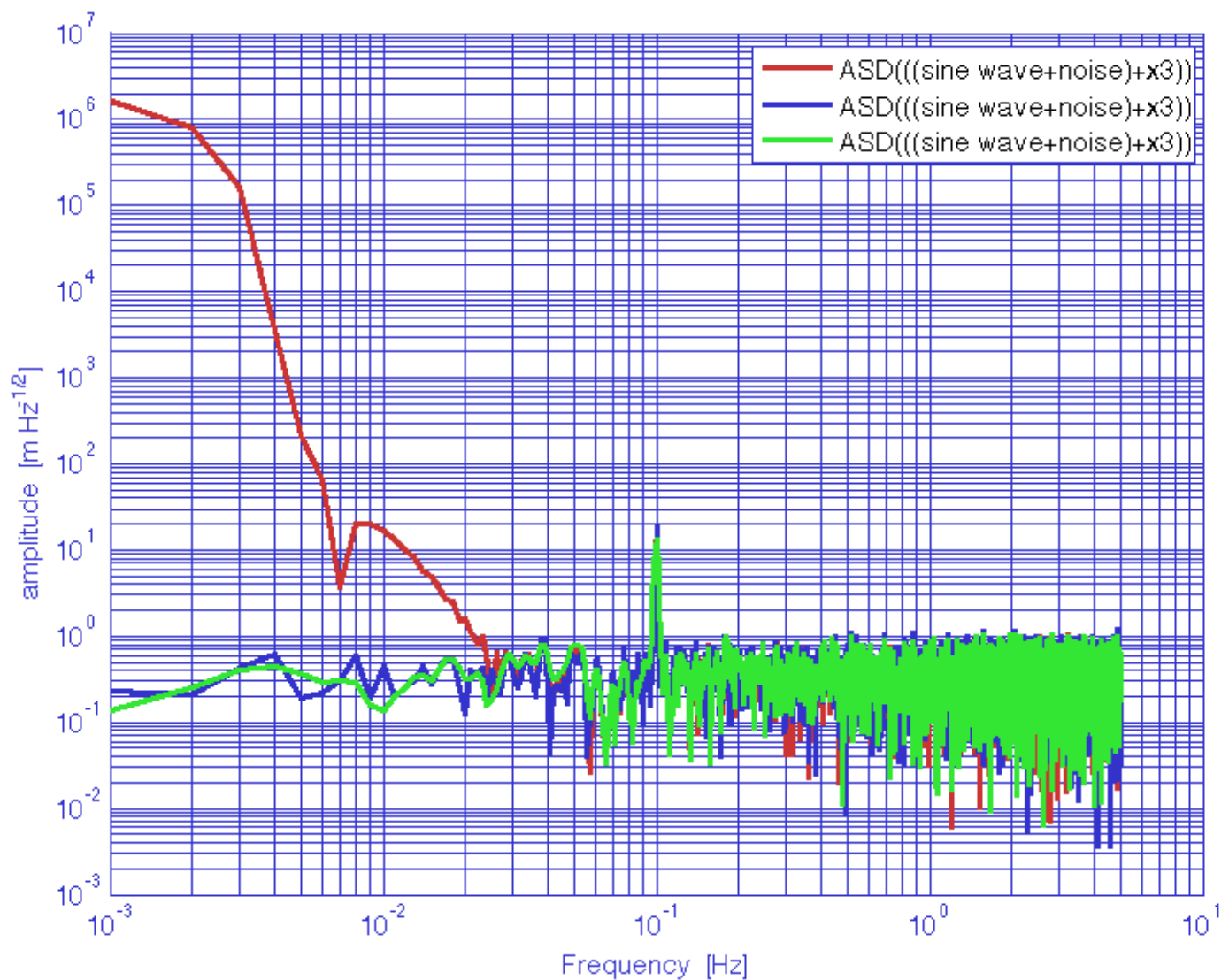
1. Evaluation of the PSD of a time-series represented by a low frequency sinewave signal, superimposed to white noise. Comparison of the effect of windowing on the estimate of the white noise level and on resolving the signal.

```
% create two AOs
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));
% add both AOs
x = x1 + x2;
% compute the psd changing the 'nfft'
y_lf = psd(x);
y_hf = psd(x,plist('nfft',1000));
% compare
ipplot(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. In the example, the same spectrum is computed with different spectral windows.

```
% create three AOs
x1 = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10,'yunits','m'));
x2 = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10,'yunits','m'));
x3 = ao(plist('tsfcn','t.^2 + t','nsecs',1000,'fs',10,'yunits','m'));
% add them
x = x1 + x2 + x3;
% compute psd with different windows
y_1 = psd(x,plist('scale','ASD','order',1,'win','BH92'));
y_2 = psd(x,plist('scale','ASD','order',2,'win','Hamming'));
y_3 = psd(x,plist('scale','ASD','order',2,'win','Kaiser','psll',200));
% compare
ipplot(y_1, y_2, y_3);
```



References

1. P.D. Welch, The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Trans. on Audio and Electroacoustics*, Vol. 15, No. 2 (1967), pp. 70 – 73.
2. B. P. Weldford, Note on a Method for Calculating Corrected Sums of Squares and Products, *Technometrics*, Vol. 4, No. 3 (1962), pp 419 – 420.

◀ Spectral Estimation Methods

Cross-spectral density estimates ▶

©LTP Team

Cross-spectral density estimates

Description

Cross-power spectral density is performed by the Welch's averaged, modified periodogram method. The LTPDA method [ao/cpsd](#) estimates the cross-spectral density of time-series signals, included in the input `aos` following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectra, by multiplying it with a [spectral window object](#), and can be detrended by polynomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lengths to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well.

Syntax

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

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated; `b` is the output objectm, and `pl` and optional `plist`

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] A string value containing the variable `'fs'` can also be used, e.g., `plist('Nfft', '2*fs')`
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- `'Olap'` – segment percent overlap [default: -1, (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
- `'Navs'` – number of averages. If set, and if `Nfft` was set to 0 or -1, the number of points for each window will be calculated to match the request. [default: -1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.

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 Kaiser windows, the `PSLL`.

As an alternative to setting the number of points `'Nfft'` in each window, it's possible to ask for a given number of CPSD estimates by setting the `'Navs'` parameter, and the algorithm takes care of calculating the correct window length, according to the amount of overlap between

subsequent segments.

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

The function makes CPSD estimates between the 2 input `aos`. The input argument list must contain 2 analysis objects, and the output will contain the CPSD estimate. If passing two identical objects `ai`, the output will be equivalent to the output of `psd(ai)`.

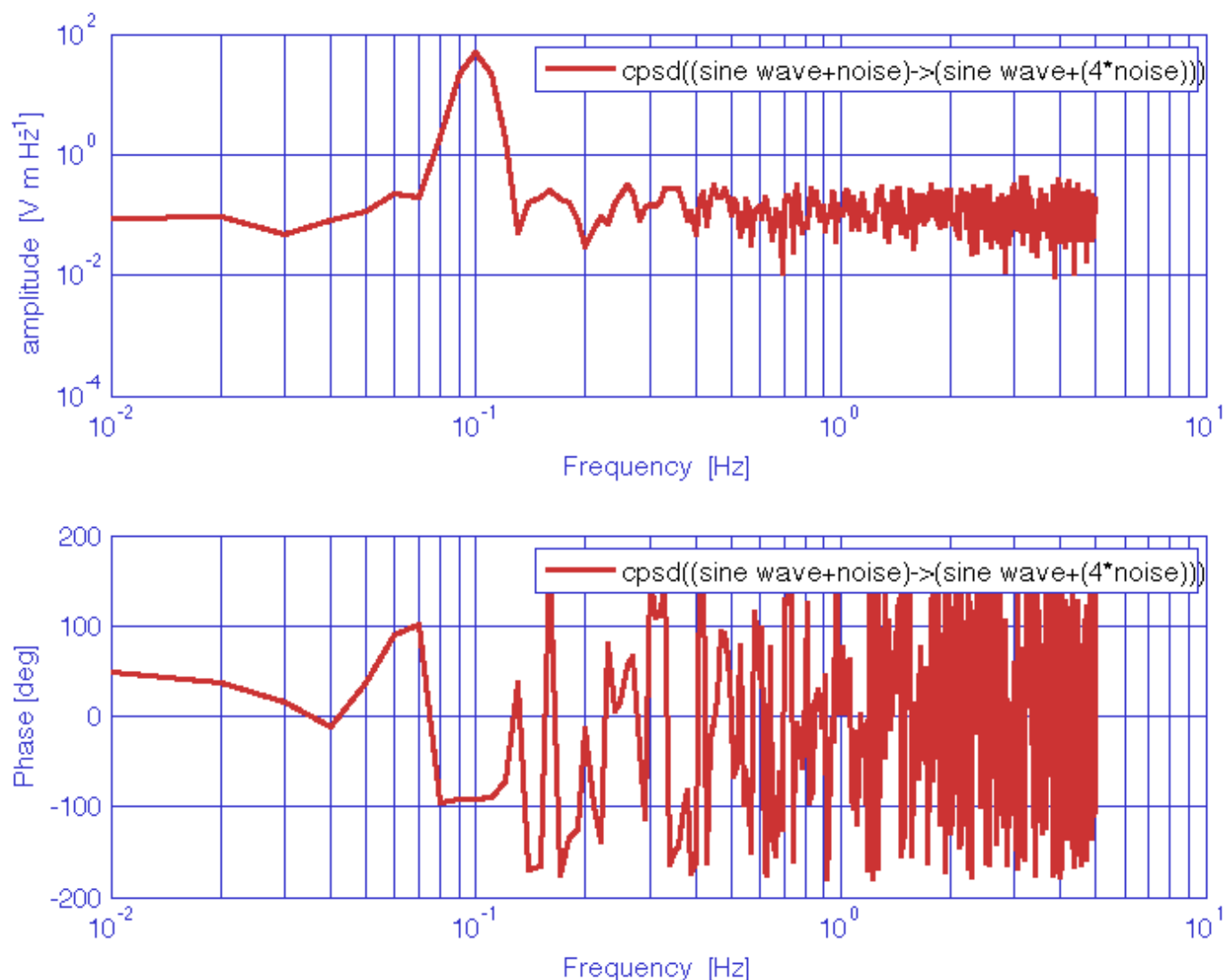
Algorithm

The algorithm is based in standard MATLAB's tools, as the ones used by [pwelch](#). However, in order to compute the sample variance of the PSD for each frequency bin, the averaging of the different segments is performed using Welford's algorithm [\[2\]](#) which allows to compute mean and variance in one loop.

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.

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



References

1. P.D. Welch, The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Trans. on Audio and Electroacoustics*, Vol. 15, No. 2 (1967), pp. 70 – 73
2. B. P. Weldford, Note on a Method for Calculating Corrected Sums of Squares and Products, *Technometrics*, Vol. 4, No. 3 (1962), pp 419 – 420.

◀ Power spectral density estimates

Cross coherence estimates ▶

©LTP Team

Cross coherence estimates

Description

The LTPDA method [ao/cohere](#) estimates the cross-coherence of time-series signals, included in the input `aos` following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectra, by multiplying it with a [spectral window object](#), and can be detrended by a polynomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lengths to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well.

Syntax

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

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated `m` `b` is the output object and `pl` is an optional parameters list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] Notice: analyzing a single segment produces as a result an object full of 1! A string value containing the variable `'fs'` can also be used, e.g., `plist('Nfft', '2*fs')`
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- `'olap'` – segment percent overlap [default: -1, (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
- `'Navs'` – number of averages. If set, and if `Nfft` was set to 0 or -1, the number of points for each window will be calculated to match the request. [default: -1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.
- `'Type'` – type of scaling of the coherence function. Choose between:
 - `'c'` – Complex Coherence $S_{xy} / \sqrt{S_{xx} * S_{yy}}$ [default]
 - `'MS'` – Magnitude-Squared Coherence $(\text{abs}(S_{xy}))^2 / (S_{xx} * S_{yy})$

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 Kaiser windows, the `PSLL`.

As an alternative to setting the number of points 'Nfft' in each window, it's possible to ask for a given number of coherence estimates by setting the 'Navs' parameter, and the algorithm takes care of calculating the correct window length, according to the amount of overlap between subsequent segments.

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

The function makes cross-coherence estimates between the 2 input aos. If passing two identical objects or linearly combined signals, the output will be 1 at all frequencies. The same will happen if analyzing only a single window.

Algorithm

The algorithm is based in standard MATLAB's tools, as the ones used by [pwelch](#). The sample variance is computed as [\[2\]](#)

$$\sigma(f) = \sqrt{\frac{2}{N|\gamma(f)|^2}(1 - |\gamma(f)|^2)^2}$$

where

$$|\gamma(f)|^2 = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}$$

is the coherence function.

Example

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

```
% parameters
nsecs = 5000;
fs = 10;
nfft = 1000;

% build first signal components
x1 = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',nsecs,'fs',fs,'yunits','m'))
x2 = ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs,'yunits','m'))
x3 = ao(plist('tsfcn','t','nsecs',nsecs,'fs',fs,'yunits','m'));

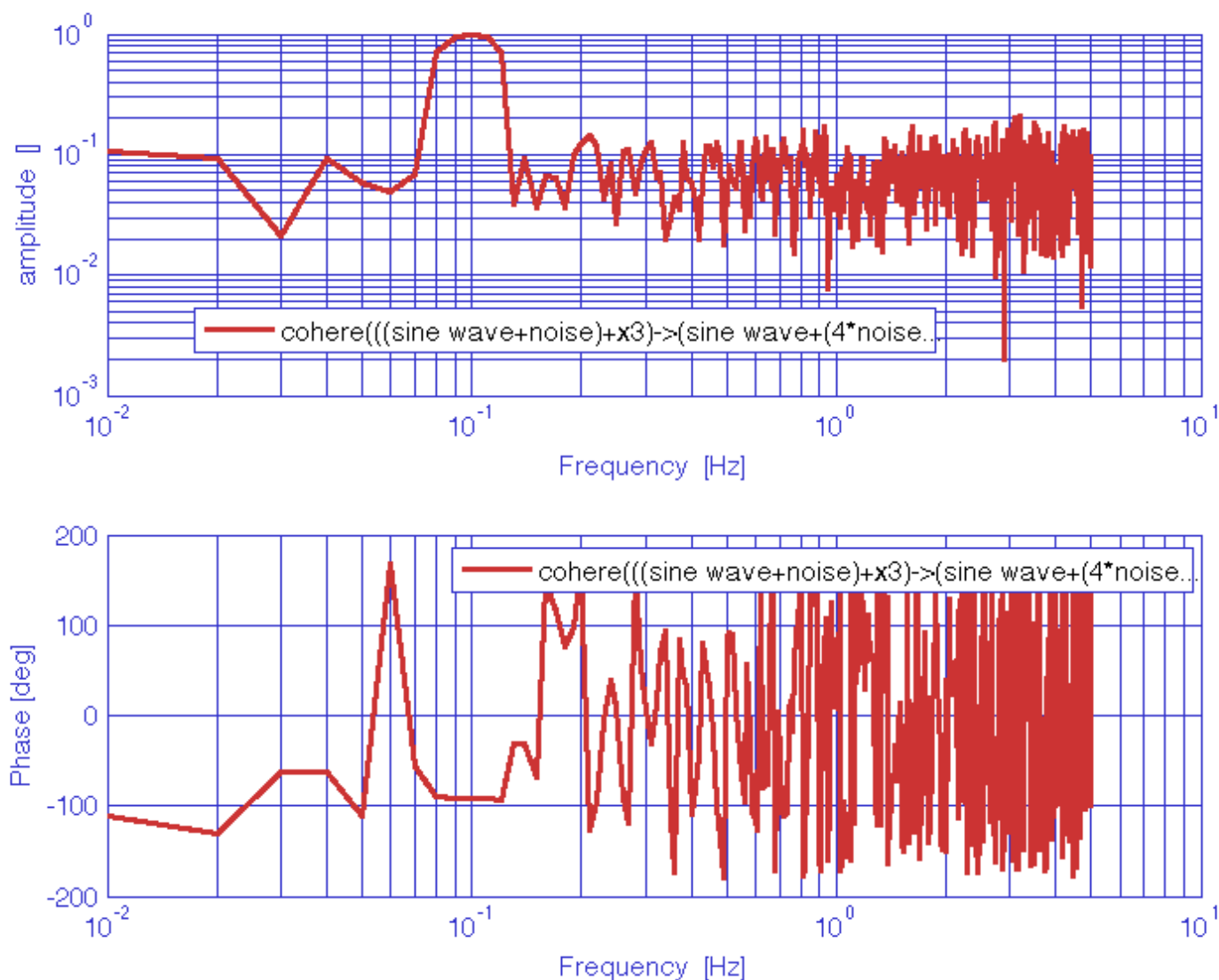
% add components
x = x1 + x2 + x3;

% build second signal components
y1 = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',nsecs,'fs',fs,'phi',90));
y2 = 4*ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));

% add components and set units
y = y1 + y2;
y.setYunits('V');

% compute coherence
pl = plist('win','BH92','nfft',nfft, 'order',1);
Cxy = cohere(x,y,pl);

%plot
iplot(Cxy);
```



References

1. P.D. Welch, The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Trans. on Audio and Electroacoustics*, Vol. 15, No. 2 (1967), pp. 70 – 73.
2. G.C. Carter, C.H. Knapp, A.H. Nuttall, Estimation of the Magnitude-Squared Coherence Function Via Overlapped Fast Fourier Transform Processing , *IEEE Trans. on Audio and Electroacoustics*, Vol. 21, No. 4 (1973), pp. 337 – 344.

◀ Cross-spectral density estimates

Transfer function estimates ▶

©LTP Team

Transfer function estimates

Description

The LTPDA method [ao/tfe](#) estimates the transfer function of time-series signals, included in the input `aos` following the Welch's averaged, modified periodogram method [\[1\]](#). Data are windowed prior to the estimation of the spectra, by multiplying it with a [spectral window object](#), and can be detrended by polynomial of time in order to reduce the impact of the border discontinuities. The window length is adjustable to shorter lengths to reduce the spectral density uncertainties, and the percentage of subsequent window overlap can be adjusted as well.

Syntax

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

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated, `b` is the output object and `pl` an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] A string value containing the variable `'fs'` can also be used, e.g., `plist('Nfft', '2*fs')`
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- `'Olap'` – segment percent overlap [default: -1, (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
- `'Navs'` – number of averages. If set, and if `Nfft` was set to 0 or -1, the number of points for each window will be calculated to match the request. [default: -1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.

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 Kaiser windows, the `PSLL`.

As an alternative to setting the number of points `'Nfft'` in each window, it's possible to ask for a given number of TFE estimates by setting the `'Navs'` parameter, and the algorithm takes care of calculating the correct window length, according to the amount of overlap between

subsequent segments.

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

The function makes transfer functions estimates between the 2 input aos, and the output will contain the transfer function estimate from the first ao to the second.

Algorithm

The algorithm is based in standard MATLAB's tools, as the ones used by [pwelch](#). The sample variance is computed as

$$\sigma(f) = \sqrt{\frac{P_{yy}(f)}{P_{xx}(f)} \frac{1 - |\gamma(f)|^2}{N}}$$

where

$$|\gamma(f)|^2 = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}$$

is the coherence function.

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.

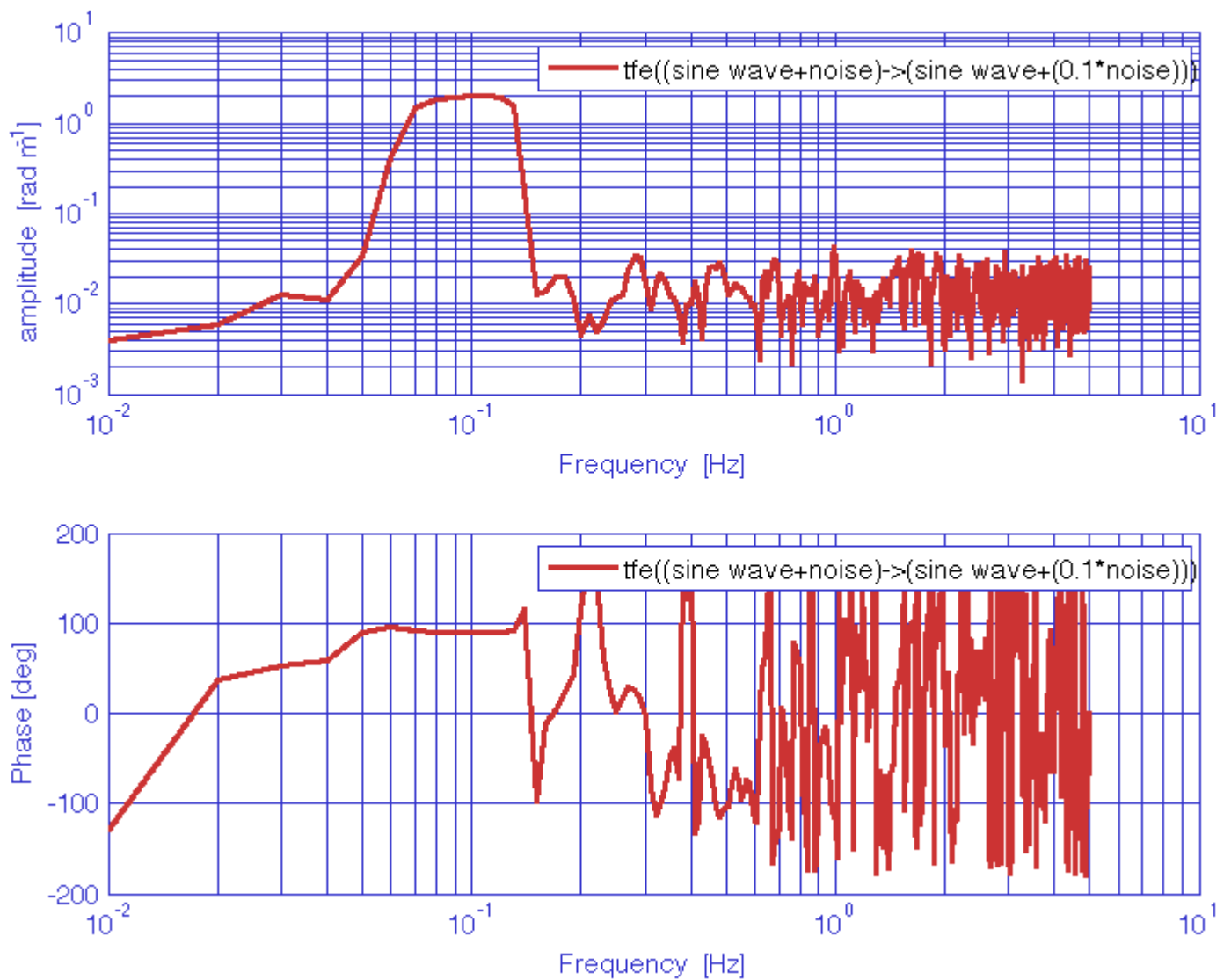
```
% parameters
nsecs = 1000;
fs = 10;

% create first signal AO
x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',nsecs,'fs',fs)) + ...
ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
x.setYunits('m');

% create second signal AO
y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',nsecs,'fs',fs,'phi',90)) + ...
0.1*ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
y.setYunits('rad');

% compute transfer function
nfft = 1000;
psll = 200;
Txy = tfe(x,y,plist('win','Kaiser','psll',psll,'nfft',nfft));

% plot
iplot(Txy)
```



References

1. P.D. Welch, The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Trans. on Audio and Electroacoustics*, Vol. 15, No. 2 (1967), pp. 70 – 73.

◀ Cross coherence estimates

Log-scale power spectral density estimates ▶

©LTP Team

Log-scale power spectral density estimates

Description

The LTPDA method [ao/lpsd](#) estimates the power spectral density of time-series signals, included in the input `aos` following the LPSD algorithm [1]. Spectral density estimates are not evaluated at frequencies which are linear multiples of the minimum frequency resolution $1/\tau$, where τ is the window length, 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.

Data are windowed prior to the estimation of the spectrum, by multiplying it with a [spectral window object](#), and can be detrended by polynomial 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).

Syntax

```
bs = lpsd(a1,a2,a3,...,pl)
bs = lpsd(as,pl)
bs = as.lpsd(pl)
```

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated, `b` is the output object and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- 'Kdes' – desired number of averages [default: 100]
- 'Jdes' – number of spectral frequencies to compute [default: 1000]
- 'Lmin' – minimum segment length [default: 0]
- 'Win' – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter 'psll'.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- 'olap' – segment percent overlap [default: -1, (taken from window function)]
- 'Scale' – scaling of output. Choose from:
 - '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 Kaiser windows, the PSLL.

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

Algorithm

The algorithm is implemented according to [1]. In order to compute the sample variance of the LPSD for each frequency bin, the averaging of the different segments is performed using Welford's algorithm [2] which allows to compute mean and variance in one loop. In the LPSD algorithm, the first frequencies bins are usually computed using a single segment containing all the data. For these bins, the sample variance is set to Inf.

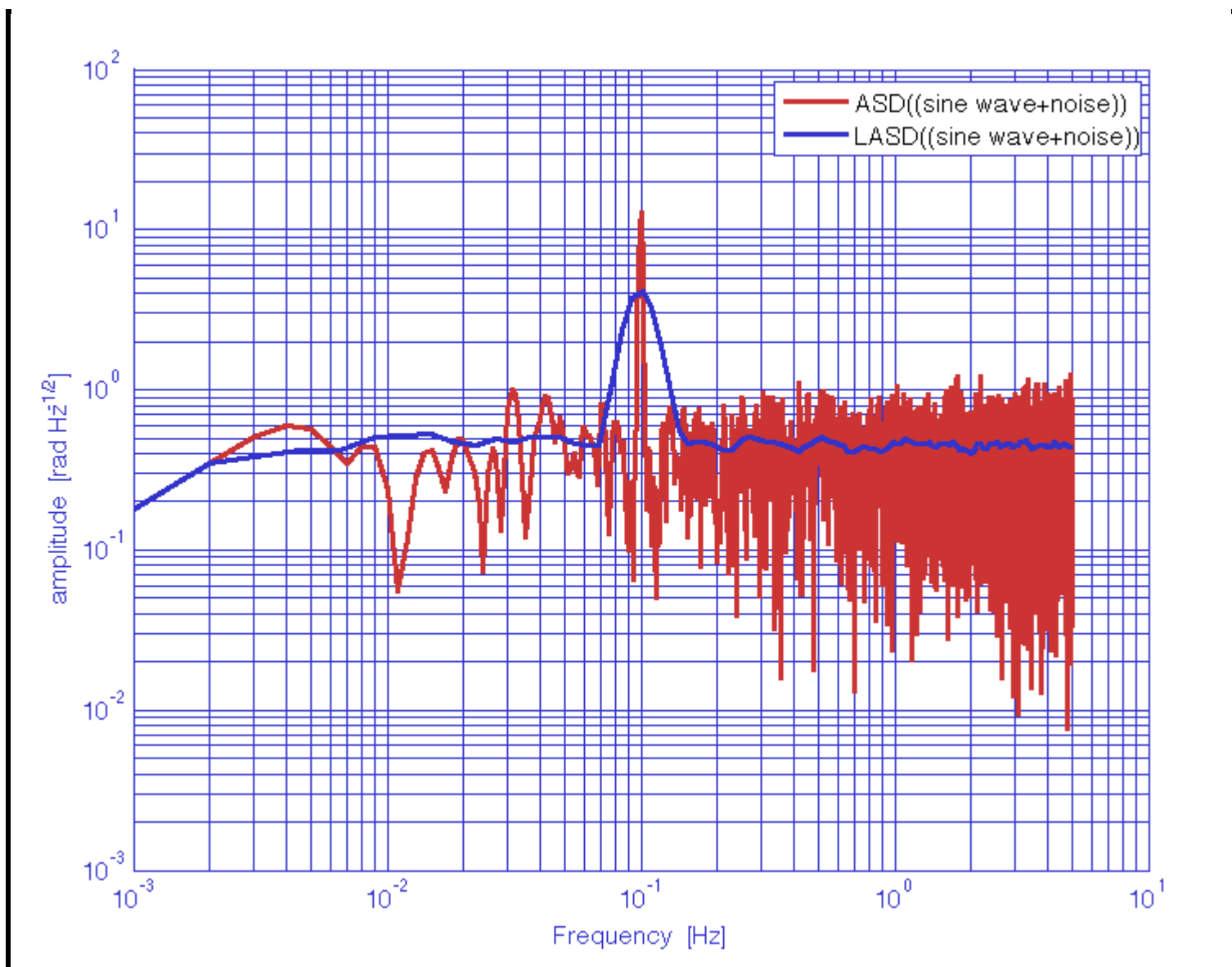
Examples

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.

```
% Create input AO
x1 = ao(plist('waveform','sine
wave','f',0.1,'A',1,'nsecs',1000,'fs',10,'yunits','rad')));
x2 = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10,'yunits','rad')));
x = x1 + x2;

% Compute psd and lpsd
pl = plist('scale','ASD','order',-1,'win','Kaiser','psll',200);
y1 = psd(x, pl);
y2 = lpsd(x, pl);

% Compare
iplot(y1, y2)
```



References

1. M. Troebs, G. Heinzel, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, [Measurement, Vol. 39 \(2006\), pp. 120 – 129.](#)
2. B. P. Weldford, Note on a Method for Calculating Corrected Sums of Squares and Products, *Technometrics*, Vol. 4, No. 3 (1962), pp 419 – 420.

◀ Transfer function estimates

Log-scale cross-spectral density estimates ▶

©LTP Team

Log-scale cross-spectral density estimates

Description

The LTPDA method [ao/lcpsd](#) estimates the cross-power spectral density of time-series signals, included in the input `aos` following the LPSD algorithm [1]. Spectral density estimates are not evaluated at frequencies which are linear multiples of the minimum frequency resolution $1/T$, where T is the window length, 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.

Data are windowed prior to the estimation of the spectrum, by multiplying it with a [spectral window object](#), and can be detrended by polynomial 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).

Syntax

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

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated, `b` is the output object and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- 'Kdes' – desired number of averages [default: 100]
- 'Jdes' – number of spectral frequencies to compute [default: 1000]
- 'Lmin' – minimum segment length [default: 0]
- 'Win' – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter 'psll'.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- 'Olap' – segment percent overlap [default: -1, (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 Kaiser windows, the PSLL.

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

The function makes log-scale CPSD estimates between the 2 input aos. The input argument list must contain 2 analysis objects, and the output will contain the LCPSD estimate. If passing two identical objects a_i , the output will be equivalent to the output of `lpsd(ai)`.

Algorithm

The algorithm is implemented according to [1]. In order to compute the sample variance of the LCPSD for each frequency bin, the averaging of the different segments is performed using Welford's algorithm [2] which allows to compute mean and variance in one loop. In the LPSD algorithm, the first frequencies bins are usually computed using a single segment containing all the data. For these bins, the sample variance is set to `Inf`.

Example

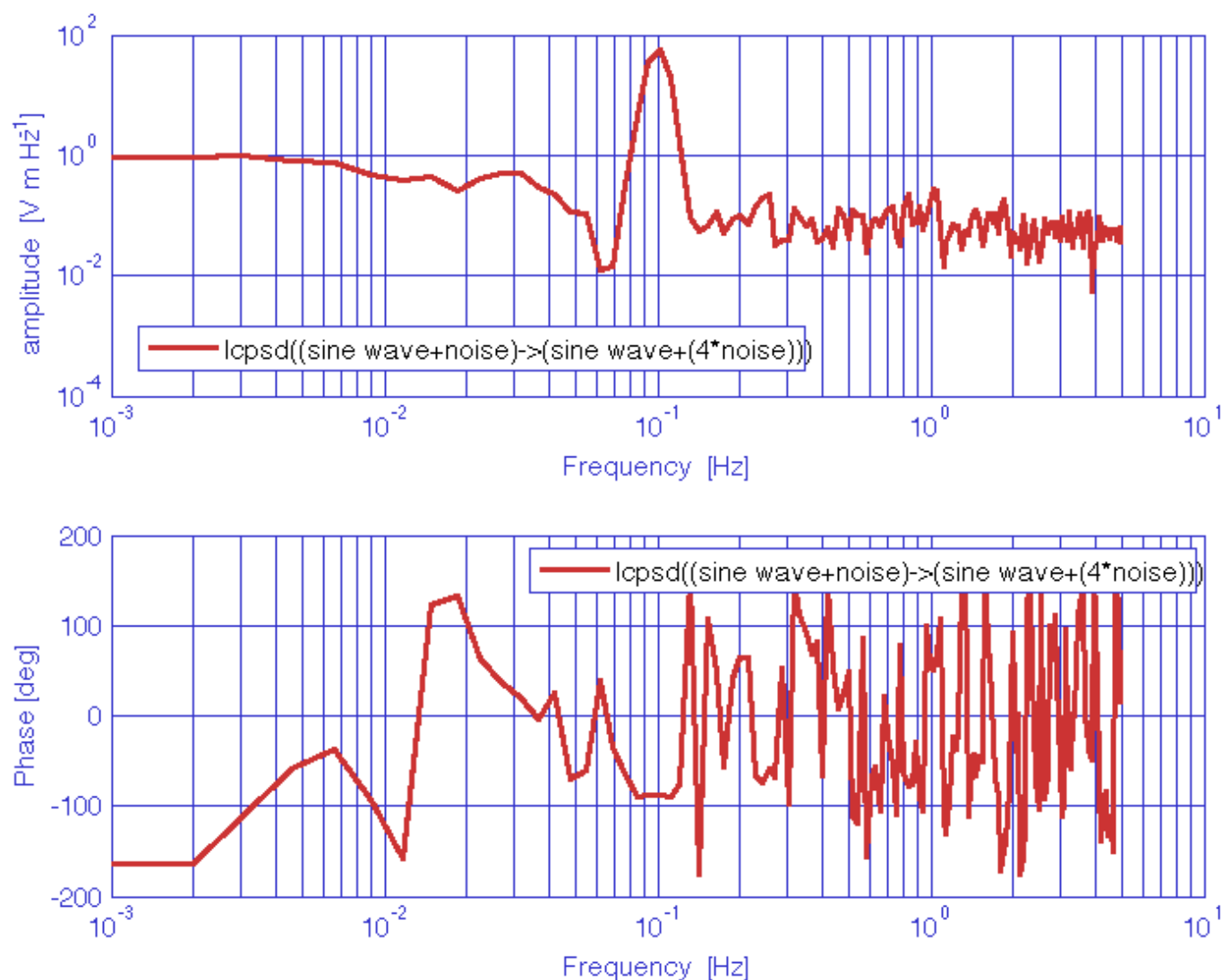
Evaluation of the log-scale CPSD of two time-series represented by: a low frequency sine wave signal superimposed to white noise, and a low frequency sine wave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
% Parameters
nsecs = 1000;
fs = 10;

% Create input AOs
x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',nsecs,'fs',fs)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
x.setYunits('m');
y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',nsecs,'fs',fs,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
y.setYunits('V');

% Compute log cpsd
z = lcpsd(x,y,plist('nfft',1000));

% Plot
iplot(z);
```



References

1. M. Troebs, G. Heinzl, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, [Measurement, Vol. 39 \(2006\), pp. 120 – 129.](#)
2. B. P. Weldford, Note on a Method for Calculating Corrected Sums of Squares and Products, *Technometrics*, Vol. 4, No. 3 (1962), pp 419 – 420.

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

©LTP Team

Log-scale cross coherence density estimates

Description

The LTPDA method [ao/lcohere](#) estimates the coherence function of time-series signals, included in the input `aos` following the LPSD algorithm [1]. Spectral density estimates are not evaluated at frequencies which are linear multiples of the minimum frequency resolution $1/\tau$, where τ is the window length, 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.

Data are windowed prior to the estimation of the spectrum, by multiplying it with a [spectral window object](#), and can be detrended by polynomial 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).

Syntax

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

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated, `b` is the output object and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- 'Kdes' – desired number of averages [default: 100]
- 'Jdes' – number of spectral frequencies to compute [default: 1000]
- 'Lmin' – minimum segment length [default: 0]
- 'Win' – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter 'psll'.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- 'Olap' – segment percent overlap [default: -1, (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
- 'Type' – type of scaling of the coherence function. Choose between:
 - 'c' – Complex Coherence $S_{xy} / \sqrt{S_{xx} * S_{yy}}$ [default]
 - 'MS' – Magnitude-Squared Coherence $(\text{abs}(S_{xy}))^2 / (S_{xx} * S_{yy})$

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 Kaiser windows, the PSLL.

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

The function makes magnitude-squared coherence estimates between the 2 input `aos`, on a logarithmic frequency scale. If passing two identical objects `ai` or linearly combined signals, the output will be 1 at all frequencies.

Algorithm

The algorithm is implemented according to [1]. The sample variance is computed according to [2]:

$$\sigma(f) = \sqrt{\frac{2}{N|\gamma(f)|^2}(1 - |\gamma(f)|^2)^2}$$

where

$$|\gamma(f)|^2 = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}$$

is the coherence function. In the LPSD algorithm, the first frequencies bins are usually computed using a single segment containing all the data. For these bins, the sample variance is set to `Inf`.

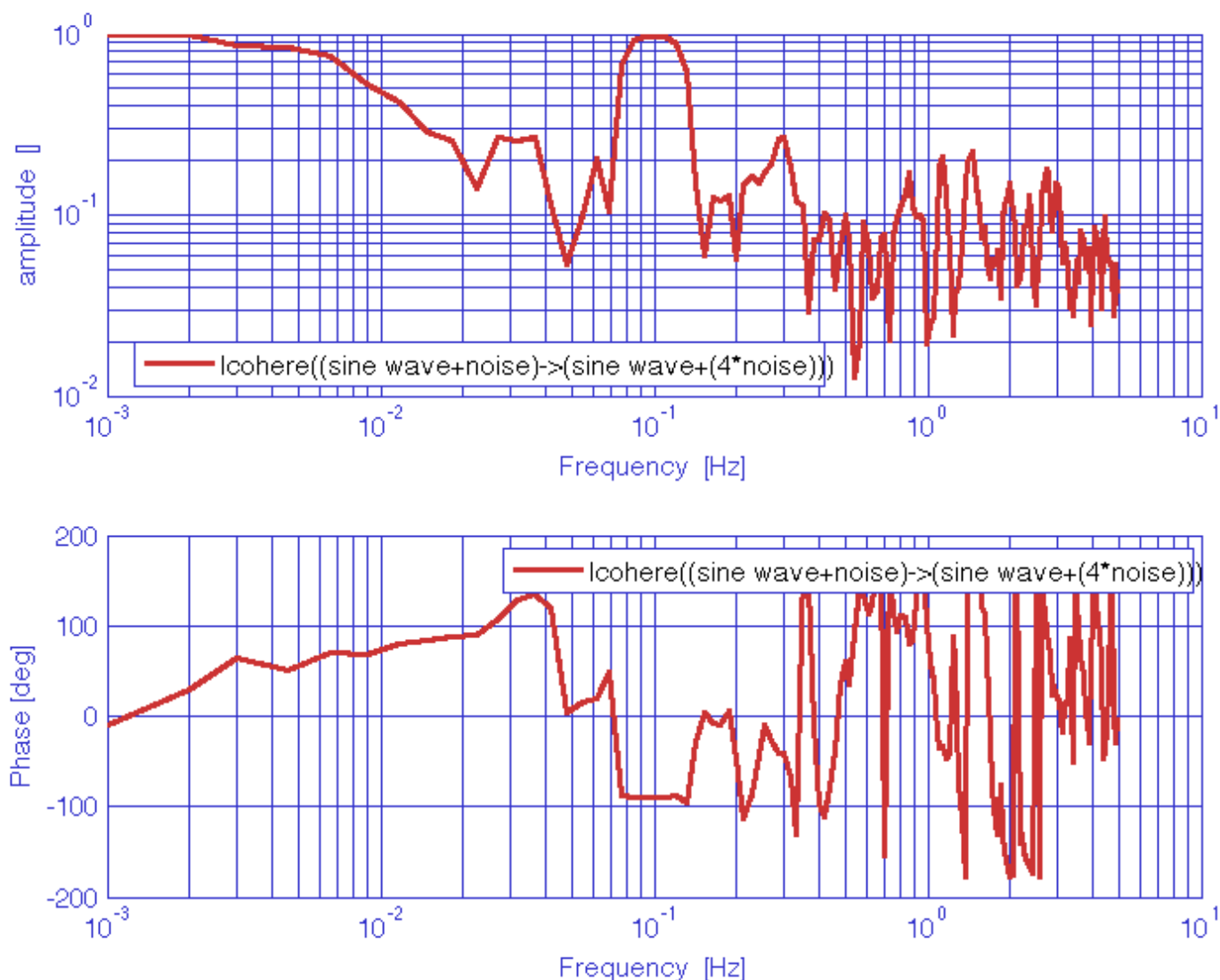
Example

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.

```
% Parameters
nsecs = 1000;
fs = 10;
x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',nsecs,'fs',fs)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
x.setYunits('m');
y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',nsecs,'fs',fs,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
y.setYunits('V');

% Compute log coherence
Cxy = lcohere(x,y,plist('win','Kaiser','psll',200));

% Plot
iplot(Cxy);
```

References

1. M. Troebs, G. Heinzl, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, [Measurement, Vol. 39 \(2006\), pp. 120 – 129.](#)
2. G.C. Carter, C.H. Knapp, A.H. Nuttall, Estimation of the Magnitude-Squared Coherence Function Via Overlapped Fast Fourier Transform Processing , *IEEE Trans. on Audio and Electroacoustics*, Vol. 21, No. 4 (1973), pp. 337 – 344.

◀ Log-scale cross-spectral density estimates

Log-scale transfer function estimates ▶

©LTP Team

Log-scale transfer function estimates

Description

The LTPDA method [ao/ltfe](#) estimates the transfer function of time-series signals, included in the input `aos` following the LPSD algorithm [1]. Spectral density estimates are not evaluated at frequencies which are linear multiples of the minimum frequency resolution $1/\tau$, where τ is the window length, 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.

Data are windowed prior to the estimation of the spectrum, by multiplying it with a [spectral window object](#), and can be detrended by polynomial 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).

Syntax

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

`a1` and `a2` are the 2 `aos` containing the input time series to be evaluated, `b` is the output object and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- 'Kdes' – desired number of averages [default: 100]
- 'Jdes' – number of spectral frequencies to compute [default: 1000]
- 'Lmin' – minimum segment length [default: 0]
- 'Win' – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter 'psll'.
For instance: `plist('Win', 'Kaiser', 'psll', 200)`.
- 'olap' – segment percent overlap [default: -1, (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 Kaiser windows, the PSLL.

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

The function makes logarithmic frequency scale transfer functions estimates between the 2 input aos, and the output will contain the transfer function estimate from the first ao to the second.

Algorithm

The algorithm is implemented according to [\[1\]](#). The sample variance is computed according to:

$$\sigma(f) = \sqrt{\frac{P_{yy}(f)}{P_{xx}(f)} \frac{1 - |\gamma(f)|^2}{N}}$$

where

$$|\gamma(f)|^2 = \frac{|P_{xy}(f)|^2}{P_{xx}(f)P_{yy}(f)}$$

is the coherence function. In the LPSD algorithm, the first frequencies bins are usually computed using a single segment containing all the data. For these bins, the sample variance is set to Inf.

Example

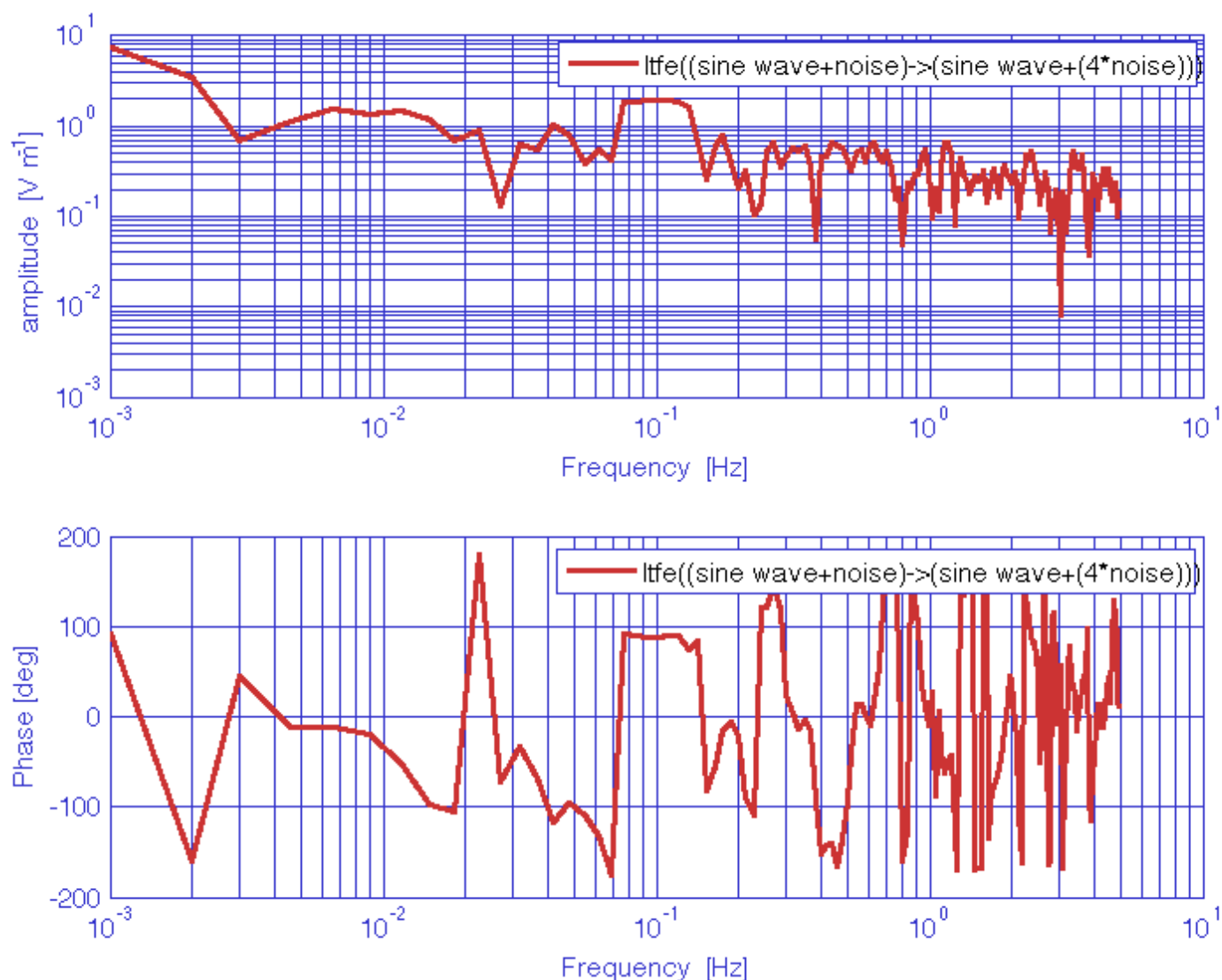
Evaluation of the transfer function between two time-series represented by: a low frequency sinewave signal superimposed to white noise, and a low frequency sinewave signal at the same frequency, phase shifted and with different amplitude, superimposed to white noise.

```
% Parameters
nsecs = 1000;
fs = 10;
nfft = 1000;

% Create input AOs
x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',nsecs,'fs',fs)) + ...
    ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
x.setYunits('m');
y = ao(plist('waveform','sine wave','f',0.1,'A',2,'nsecs',nsecs,'fs',fs,'phi',90)) + ...
    4*ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
y.setYunits('V');

% Compute transfer function
Txy = ltfe(x,y,plist('win',specwin('Kaiser',1,200),'nfft',nfft));

% Plot
iplot(Txy);
```



References

1. M. Troebbs, G. Heinzl, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, [Measurement, Vol. 39 \(2006\), pp. 120 - 129.](#)

◀ Log-scale cross coherence density estimates

Fitting Algorithms ▶

©LTP Team

Fitting Algorithms

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

- [Polynomial Fitting](#)
- [Time Domain Fit](#)
- [Z-Domain Fit](#)
- [S-Domain Fit](#)

 Log-scale transfer function estimates

Polynomial Fitting 

©LTP Team

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.

Z-Domain Fit

Description	Z-domain system identification in LTPDA.
Algorithm	Fit Algorithm.
Examples	Usage example of z-domain system identification tool.
References	Bibliographic references.

Z-domain system identification in LTPDA

System identification in z-domain is performed with the function [zDomainFit](#). It is based on a modeified version of the vector fitting algorithm that was adapted to fit in z-domain. Details on the core algorithm can be found in [1 – 3].

If you provide more than one AO as input, they will be fitted together with a common set of poles. Only frequency domain ([fsdata](#)) data can be fitted. Each non fsdata object is ignored. Input objects must have the same number of elements.

Fit algorithm

The function performs a fitting loop to automatically identify model order and parameters in z-domain. Output is a z-domain model expanded in partial fractions:

$$F(z) = \frac{r_1}{1 - p_1 z^{-1}} + \dots + \frac{r_n}{1 - p_n z^{-1}}$$

Each element of the partial fraction expansion can be seen as a [miir](#) filter. Therefore the complete expansion is simply a parallel [filterbank](#) of [miir](#) filters. Since the function can fit more than one input analysis object at a time with a common set of poles, output filterbank are embedded in a [matrix](#) (note that this characteristic will be probably changed because of the introduction of the [collection](#) class).

Identification loop stops when the stop condition is reached. Stop criterion is based on three different approaches:

1. Mean Squared Error and variation
Check if the normalized mean squared error is lower than the value specified in `FITTOL` and if the relative variation of the mean squared error is lower than the value specified in `MSEVARTOL`. E.g. `FITTOL = 1e-3`, `MSEVARTOL = 1e-2` search for a fit with normalized meam square error lower than `1e-3` and MSE relative variation lower than `1e-2`.
2. Log residuals difference and root mean squared error
 - Log Residuals difference
Check if the minimum of the logarithmic difference between data and residuals is larger than a specified value. ie. if the conditioning value is 2, the function ensures that the difference between data and residuals is at lest two order of magnitude lower than data itsleves.
 - Root Mean Squared Error

Check that the variation of the root mean squared error is lower than $10^{(-1 \cdot \text{value})}$.

3. Residuals spectral flatness and root mean squared error

◦ Residuals Spectral Flatness

In case of a fit on noisy data, the residuals from a good fit are expected to be as much as possible similar to a white noise. This property can be used to test the accuracy of a fit procedure. In particular it can be tested that the spectral flatness coefficient of the residuals is larger than a certain quantity sf such that $0 < sf < 1$.

◦ Root Mean Squared Error

Check that the variation of the root mean squared error is lower than $10^{(-1 \cdot \text{value})}$.

Fitting loop stops when the two stopping conditions are satisfied, in both cases.

The function can also perform a single loop without taking care of the stop conditions. This happens when '**AUTOSEARCH**' parameter is set to '**OFF**'.

Usage example of z-domain system identification tool

In this example we fit a given frequency response to get a stable `miir` filter. For the meaning of any parameter please refer to [ao](#) and [zDomainFit](#) documentation pages.

```
pl = plist(...
    'fsfcn', '(1e-3./(2.*pi.*1i.*f).^2 + 1e3./(0.001+2.*pi.*1i.*f) +
1e5.*(2.*pi.*1i.*f).^2).*1e-10',...
    'f1', 1e-6,...
    'f2', 5,...
    'nf', 100);

a = ao(pl);
a.setName;

% Fit parameter list
pl_fit = plist('FS',10,...
    'AutoSearch','on',...
    'StartPolesOpt','clog',...
    'maxiter',50,...
    'minorder',15,...
    'maxorder',30,...
    'weightparam','abs',...
    'CONDTYPE','MSE',...
    'FITTOL',1e-2,...
    'MSEVARTOL',1e-1,...
    'Plot','on',...
    'ForceStability','on');

% Do fit
mod = zDomainFit(a, pl_fit);
```

`mod` is a matrix object containing a filterbank object.

```
>> mod
---- matrix 1 ----
name: fit(a)
size: 1x1
01: filterbank | filterbank(fit(a)(fs=10.00, ntaps=2.00, a=[-1.19e+005 0], b=[1 0.0223]),
fit(a)(fs=10.00, ntaps=2.00, a=[1.67e+005 0], b=[1 0.137]), fit(a)(fs=10.00, ntaps=2.00, a=[-
5.41e+004 0], b=[1 0.348]), fit(a)(fs=10.00, ntaps=2.00, a=[1.15e+004 0], b=[1 0.603]),
fit(a)(fs=10.00, ntaps=2.00, a=[-1.69e+005 0], b=[1 0.639]), fit(a)(fs=10.00, ntaps=2.00,
a=[1.6e+005 0], b=[1 0.64]), fit(a)(fs=10.00, ntaps=2.00, a=[9.99e-009 0], b=[1 -1]),
fit(a)(fs=10.00, ntaps=2.00, a=[-4.95e-010 0], b=[1 1]), fit(a)(fs=10.00, ntaps=2.00,
a=[9.4e+003-i*3.7e+003 0], b=[1 -0.0528-i*0.0424]), fit(a)(fs=10.00, ntaps=2.00,
a=[9.4e+003+i*3.7e+003 0], b=[1 -0.0528+i*0.0424]), fit(a)(fs=10.00, ntaps=2.00,
a=[1.66e+003-i*1.45e+004 0], b=[1 0.0233-i*0.112]), fit(a)(fs=10.00, ntaps=2.00,
a=[1.66e+003+i*1.45e+004 0], b=[1 0.0233+i*0.112]), fit(a)(fs=10.00, ntaps=2.00, a=[-
1.67e+004+i*432 0], b=[1 0.171-i*0.14]), fit(a)(fs=10.00, ntaps=2.00, a=[-1.67e+004-i*432 0],
b=[1 0.171+i*0.14]), fit(a)(fs=10.00, ntaps=2.00, a=[7.61e+003+i*7.36e+003 0], b=[1 0.378-
i*0.112]), fit(a)(fs=10.00, ntaps=2.00, a=[7.61e+003-i*7.36e+003 0], b=[1 0.378+i*0.112]),
fit(a)(fs=10.00, ntaps=2.00, a=[3.67e-015-i*4.61e-006 0], b=[1 -1-i*1.08e-010]),
fit(a)(fs=10.00, ntaps=2.00, a=[3.67e-015+i*4.61e-006 0], b=[1 -1+i*1.08e-010]))
description:
UUID: 9274455a-68e8-4bf1-b1ad-db81551f3cd6
-----
```

The filterbank object contains a parallel bank of 18 filters.

```
>> mod.objs
---- filterbank 1 ----
name: fit(a)
type: parallel
01: fit(a)(fs=10.00, ntaps=2.00, a=[-1.19e+005 0], b=[1 0.0223])
02: fit(a)(fs=10.00, ntaps=2.00, a=[1.67e+005 0], b=[1 0.137])
03: fit(a)(fs=10.00, ntaps=2.00, a=[-5.41e+004 0], b=[1 0.348])
04: fit(a)(fs=10.00, ntaps=2.00, a=[1.15e+004 0], b=[1 0.603])
05: fit(a)(fs=10.00, ntaps=2.00, a=[-1.69e+005 0], b=[1 0.639])
06: fit(a)(fs=10.00, ntaps=2.00, a=[1.6e+005 0], b=[1 0.64])
07: fit(a)(fs=10.00, ntaps=2.00, a=[9.99e-009 0], b=[1 -1])
08: fit(a)(fs=10.00, ntaps=2.00, a=[-4.95e-010 0], b=[1 1])
09: fit(a)(fs=10.00, ntaps=2.00, a=[9.4e+003-i*3.7e+003 0], b=[1 -0.0528-i*0.0424])
10: fit(a)(fs=10.00, ntaps=2.00, a=[9.4e+003+i*3.7e+003 0], b=[1 -0.0528+i*0.0424])
11: fit(a)(fs=10.00, ntaps=2.00, a=[1.66e+003-i*1.45e+004 0], b=[1 0.0233-i*0.112])
12: fit(a)(fs=10.00, ntaps=2.00, a=[1.66e+003+i*1.45e+004 0], b=[1 0.0233+i*0.112])
13: fit(a)(fs=10.00, ntaps=2.00, a=[-1.67e+004+i*432 0], b=[1 0.171-i*0.14])
14: fit(a)(fs=10.00, ntaps=2.00, a=[-1.67e+004-i*432 0], b=[1 0.171+i*0.14])
15: fit(a)(fs=10.00, ntaps=2.00, a=[7.61e+003+i*7.36e+003 0], b=[1 0.378-i*0.112])
16: fit(a)(fs=10.00, ntaps=2.00, a=[7.61e+003-i*7.36e+003 0], b=[1 0.378+i*0.112])
17: fit(a)(fs=10.00, ntaps=2.00, a=[3.67e-015-i*4.61e-006 0], b=[1 -1-i*1.08e-010])
18: fit(a)(fs=10.00, ntaps=2.00, a=[3.67e-015+i*4.61e-006 0], b=[1 -1+i*1.08e-010])
description:
UUID: 21af6960-61a8-4351-b504-e6f2b5e55b06
-----
```

Each object of the filterbank is a miir filter.

```
filt = mod.objs.filters.index(3)
----- miir/1 -----
b: [1 0.348484501572296]
histin: 0
version: $Id: zdomainfit_content.html,v 1.6 2009/08/27 11:38:58 luigi Exp $
ntaps: 2
fs: 10
infile:
a: [-54055.7700068032 0]
histout: 0
iunits: [] [1x1 unit]
ounits: [] [1x1 unit]
hist: miir.hist [1x1 history]
procinfo: (empty-plist) [1x1 plist]
plotinfo: (empty-plist) [1x1 plist]
name: (fit(a)(3,1))(3)
description:
mdlfile:
UUID: 6e2a1cd8-f17d-4c9d-aea9-4d9a96e41e68
-----
```

References

1. B. Gustavsen and A. Semlyen, "Rational approximation of frequency domain responses by Vector Fitting", IEEE Trans. Power Delivery vol. 14, no. 3, pp. 1052–1061, July 1999.
2. B. Gustavsen, "Improving the Pole Relocating Properties of Vector Fitting", IEEE Trans. Power Delivery vol. 21, no. 3, pp. 1587–1592, July 2006.
3. Y. S. Mekonnen and J. E. Schutt-Aine, "Fast broadband macromodeling technique of sampled time/frequency data using z-domain vector-fitting method", Electronic Components and Technology Conference, 2008. ECTC 2008. 58th 27–30 May 2008 pp. 1231 – 1235.

S-Domain Fit

Description	S-domain system identification in LTPDA.
Algorithm	Fit Algorithm.
Examples	Usage example of s-domain system identification tool.
References	Bibliographic references.

S-domain system identification in LTPDA

System identification in s-domain is performed with the function [sDomainFit](#). It is based on a modeified version of the vector fitting algorithm. Details on the core agorithm can be found in [1 – 2].

Fit Algorithm

The function performs a fitting loop to automatically identify model order and parameters in s-domain. Output is a s-domain model expanded in partial fractions:

$$f(s) = \frac{r_1}{s - p_1} + \dots + \frac{r_N}{s - p_N} + d$$

Since the function can fit more than one input analysis object at a time with a common set of poles, output [parfrac](#) are embedded in a [matrix](#) (note that this characteristic will be probably changed because of the introduction of the [collection](#) class).

Identification loop stops when the stop condition is reached. Stop criterion is based on three different approaches:

1. Mean Squared Error and variation
Check if the normalized mean squared error is lower than the value specified in `FITTOL` and if the relative variation of the mean squared error is lower than the value specified in `MSEVARTOL`. E.g. `FITTOL = 1e-3`, `MSEVARTOL = 1e-2` search for a fit with normalized mean square error lower than `1e-3` and MSE relative variation lower than `1e-2`.
2. Log residuals difference and root mean squared error
 - Log Residuals difference Check if the minimum of the logarithmic difference between data and residuals is larger than a specified value. ie. if the conditioning value is 2, the function ensures that the difference between data and residuals is at least two order of magnitude lower than data itsleves.
 - Root Mean Squared Error Check that the variation of the root mean squared error is lower than $10^{(-1 \cdot \text{value})}$.
3. Residuals spectral flatness and root mean squared error
 - Residuals Spectral Flatness In case of a fit on noisy data, the residuals from a good fit are expected to be as much as possible similar to a white noise. This property can be used to test the accuracy of a fit procedure. In particular it can be tested that the spectral flatness coefficient of the residuals is larger than a certain qiantity `sf` such that $0 < sf < 1$.

- Root Mean Squared Error Check that the variation of the root mean squared error is lower than $10^{(-1 \cdot \text{value})}$.

The function can also perform a single loop without taking care of the stop conditions. This happens when '**AutoSearch**' parameter is set to '**off**'.

Usage example of s-domain system identification tool

In this example we fit a given frequency response to get a partial fraction model. For the meaning of any parameter please refer to [ao](#) and [sDomainFit](#) documentation pages.

```
pl = plist(...
    'fsfcn', '(1e-3./(f).^2 + 1e3./(0.001+f) + 1e5.*f.^2).*1e-10',...
    'f1', 1e-6,...
    'f2', 5,...
    'nf', 100);

a = ao(pl);
a.setName;

% Fit parameter list
pl_fit = plist(...
    'AutoSearch', 'on',...
    'StartPolesOpt', 'clog',...
    'maxiter', 50,...
    'minorder', 7,...
    'maxorder', 15,...
    'weightparam', 'abs',...
    'CONDTYPE', 'MSE',...
    'FITTOL', 1e-3,...
    'MSEVARTOL', 1e-2,...
    'Plot', 'on',...
    'ForceStability', 'off');

% Do fit
mod = sDomainFit(a, pl_fit);
```

mod is a matrix object containing a parfrac object.

```
>> mod
---- matrix 1 ----
      name: fit(a)
      size: 1x1
      01: parfrac | parfrac(fit(a))
description:
      UUID: 2dc1ac28-4199-42d2-9b1a-b420252b3f8c
-----
```

```
>> mod.objs
---- parfrac 1 ----
model:      fit(a)
res:        [1.69531090137847e-006; -1.69531095674486e-006; 1.39082537801437e-007; -
1.39094453401266e-007; 3.9451875151135e-007; -3.94524993613367e-007; 4.53671387948961e-007; -
4.53664974359603e-007; 1124.81020427899; 0.000140057852149302-
i*0.201412268649905; 0.000140057852149302+i*0.201412268649905]
poles:      [-1.18514026248382e-006; 1.18514354570495e-006; -
0.00457311582050939; 0.0045734088943545; -0.0316764149343339; 0.0316791653277322; -
0.276256442292693; 0.27627799022013; 330754.550617933; -0.0199840558095427+i*118.439896186467; -
0.0199840558095427-i*118.439896186467]
dir:        0
pmul:       [1;1;1;1;1;1;1;1;1;1;1]
iunits:     []
ounits:     []
description:
UUID:       2afc4c82-7c2a-4fe3-8910-d8590884d58c
-----
```

References

1. B. Gustavsen and A. Semlyen, "Rational approximation of frequency domain responses by Vector Fitting", IEEE Trans. Power Delivery vol. 14, no. 3, pp. 1052–1061, July 1999.

2. B. Gustavsen, "Improving the Pole Relocating Properties of Vector Fitting", IEEE Trans. Power Delivery vol. 21, no. 3, pp. 1587–1592, July 2006.

◀ Z-Domain Fit

Graphical User Interfaces in LTPDA ▶

©LTP Team

Graphical User Interfaces in LTPDA

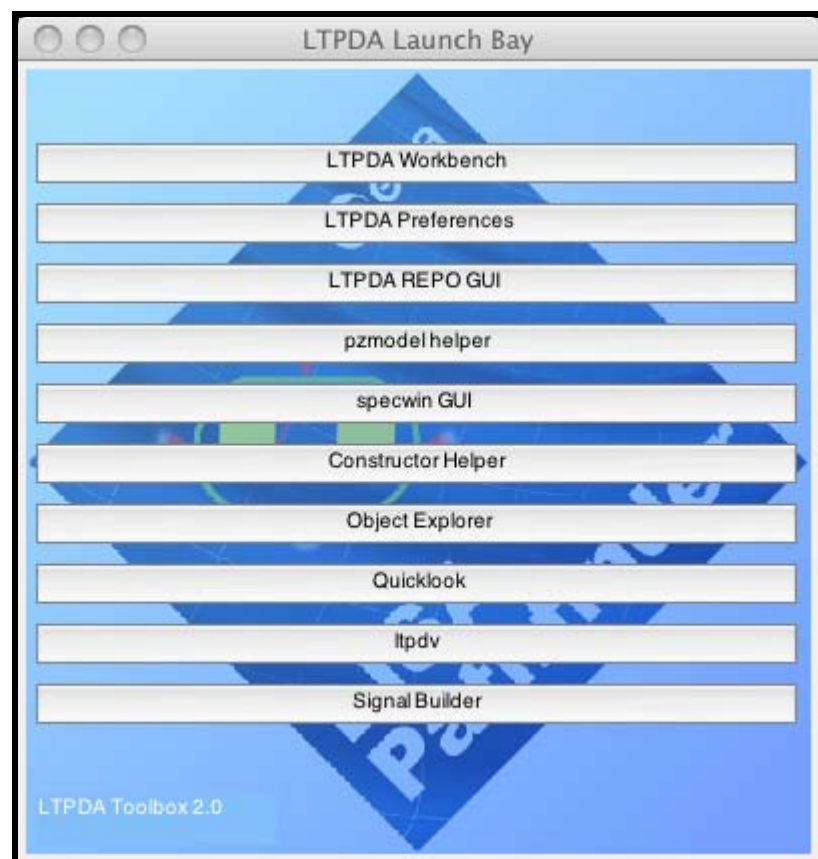
LTPDA has a variety of Graphical User Interfaces:

- [The LTPDA Launch Bay](#)
- [The LTPDA Workbench](#)
- [The LTPDA Workspace Browser](#)
- [The pole/zero model helper](#)
- [The Spectral Window GUI](#)
- [The constructor helper](#)
- [The LTPDA object explorer](#)

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 Workbench

The LTPDA Workbench offers a graphical interface for creating signal processing pipelines. By dragging and dropping blocks which represent LTPDA algorithms, users can build up a signal processing pipeline and then execute it at the press of a button. The progress of the execution can be followed graphically on the screen.

The following sections describe the use of the LTPDA Workbench.

- [Introduction](#)
- [Mouse and keyboard actions](#)
- [The canvas](#)
- [Building pipelines by hand](#)
- [Using the Workbench Shelf](#)
- [Building pipelines programatically](#)
- [Executing pipelines](#)

Loading the LTPDA Workbench

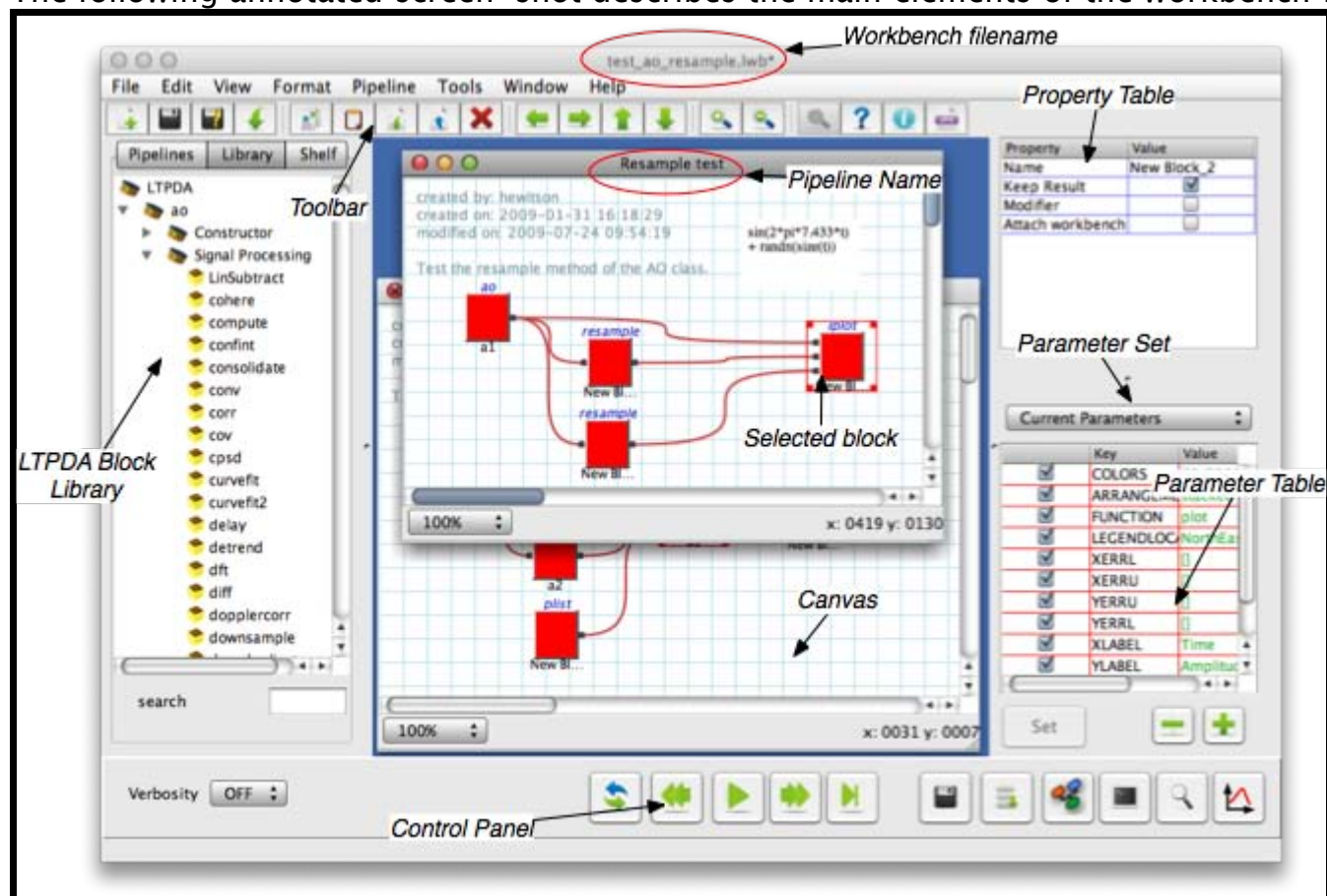
Overview

An LTPDA Workbench is a collection of pipelines. Each pipeline can have sub-pipelines which are represented as *subsystem blocks* on the parent canvas. Nested subsystems are supported to any depth.

Only one LTPDA Workbench can be open at any one time, but a collection of pipelines in a workbench saved on disk can be imported to the current workbench.

Each block/element must have a unique name on a particular canvas.

The following annotated screen-shot describes the main elements of the workbench interface:



Starting the Workbench

To start the LTPDA Workbench, click on the launcher on the LTPDA Launch Bay. Alternatively, the workbench can be started from the command window by typing:

```
>> LTPDAworkbench
```

You can also get a handle to the workbench so that you can use the programmatic interface. To do that

```
>> wb = LTPDAworkbench
```

If you loose the variable `wb`, for example, by using the `clear` command, then you can retrieve a handle to the workbench by doing

```
>> wb = getappdata(0, 'LTPDAworkbench');
```

More advanced uses of the workbench command interface (such as creating pipelines from LTPDA objects), are described in [Building pipelines programatically](#).

◀ The LTPDA Workbench

Mouse and keyboard actions ▶

©LTP Team

Mouse and keyboard actions

This section describe the various mouse and key actions that are possible on the LTPDA Workbench.

- [Keyboard actions on the main workbench](#)
- [Keyboard actions on the Canvas](#)
- [Mouse actions on the Canvas](#)

Keyboard actions on the main workbench

Action (on Windows/Linux)	Action (on Mac OS X)	Description
enter on a selected element in the block library	same	Add the block to the active canvas
enter in property or parameter value (or key)	same	Set the new value to the property or parameter

Keyboard actions on the Canvas

Action (on Windows/Linux)	Action (on Mac OS X)	Description
Arrow keys	same	Scroll canvas
ctrl-c	cmd-c	Copy selected elements
ctrl-v	cmd-v	Paste selected elements
shift-arrow keys	same	Move selected elements
shift-alt right-arrow	same	Jump to next pipeline
shift-alt left-arrow	same	Jump to previous pipeline
ctrl-i	cmd-i	Open the canvas info dialog panel.
ctrl-b	cmd-b	Open the "Quick Block" dialog panel.
ctrl-f	cmd-f	Open the "Block Search" dialog panel.

escape	same	De-select all blocks.
delete	same	Delete selected blocks (or pipes).

Mouse actions on the Canvas

Action (on Windows/Linux)	Action (on Mac OS X)	Description
Mouse-wheel scroll	same	Zoom in and out on the canvas
drag with left-mouse-button	same	Draw rubber-band box to select elements
alt-left-mouse-button	same	Move the canvas around
right-click	right-click (or ctrl-left-click)	Bring up canvas context menu
left-click on canvas	same	De-select all selected blocks or pipes
left-click on a block	same	Select the block and bring up its property and parameter tables
shift-left-click on a block	same	Add the block to the selected blocks
left mouse button down on a block	same	Move this and all other selected blocks
click-drag on selected block handles	same	Resize the block
ctrl-left-click on block	cmd-left-click on block	If a single block is selected before this action then the result of this action is to connect the two blocks.
mouse-drag on a port	mouse-drag on a port	Start drawing a pipe originating from the port.
release left mouse button on a port	same	If a pipe was being dragged, then the source port and destination port are connected by a pipe.
release left mouse button on a block	same	If a pipe was being dragged, then the source port is connected to the first free input of the destination block.



◀ Loading the LTPDA Workbench

The canvas ▶

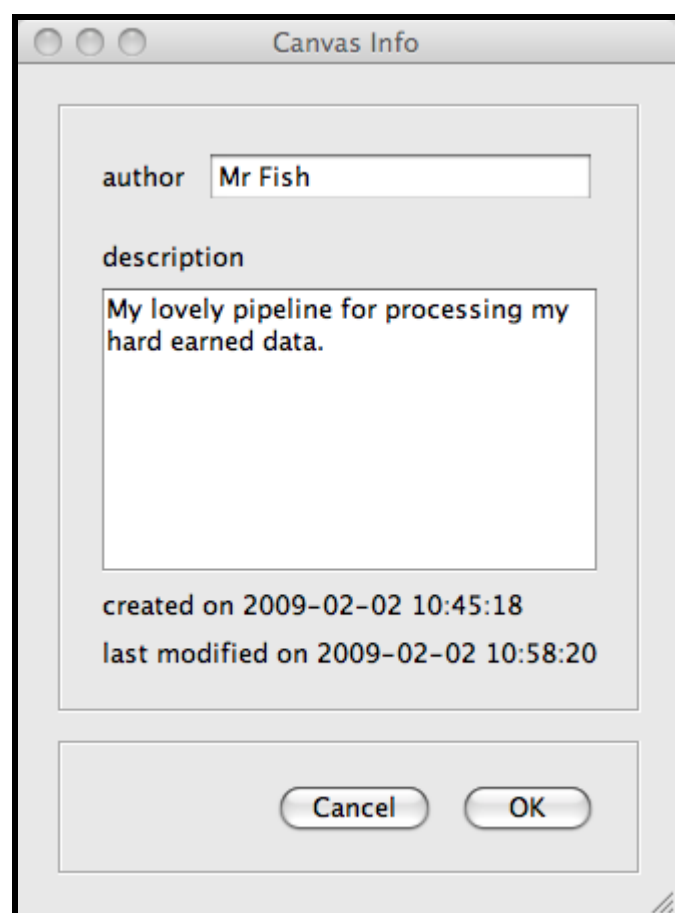
©LTP Team

The canvas

Composing LTPDA pipelines is done on a "Canvas". Each top-level pipeline, and each subsystem, is represented on a canvas. A subsystem block is also a view of pipeline.

Canvas properties

You can set properties of a canvas via the canvas inspector. To open the inspector, hit `ctrl-i` (`cmd-i` on Mac OS X) on the canvas.

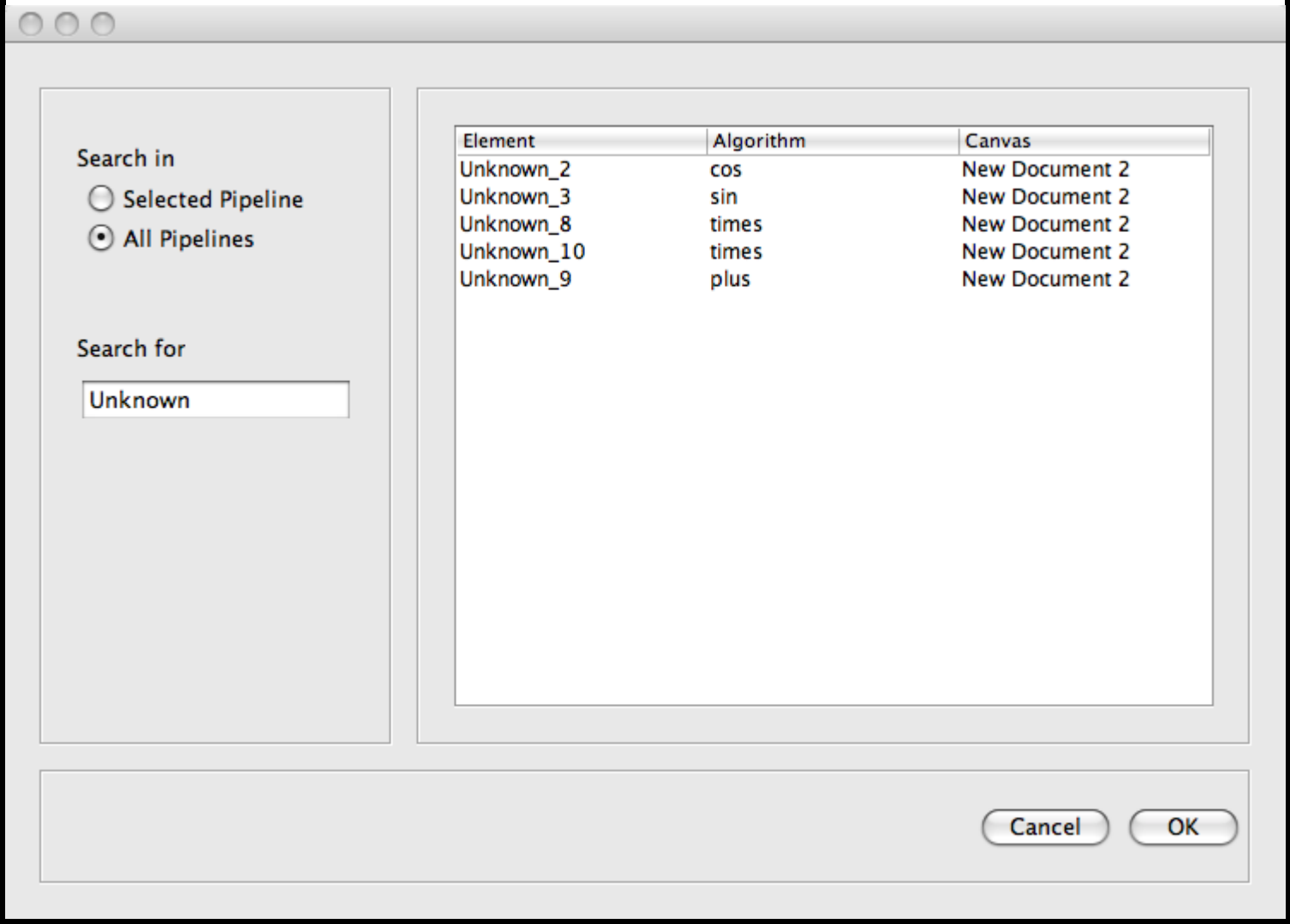


With the canvas inspector you can edit

- The author of the canvas
- A description of the canvas

Searching on the Canvas

You can search for blocks on the current canvas, or across all canvases in the workbench. To open the block search dialog, hit `ctrl-f` (`cmd-f` on Mac OS X) on the canvas.



Building pipelines by hand

This section describes how to compose pipelines by hand.

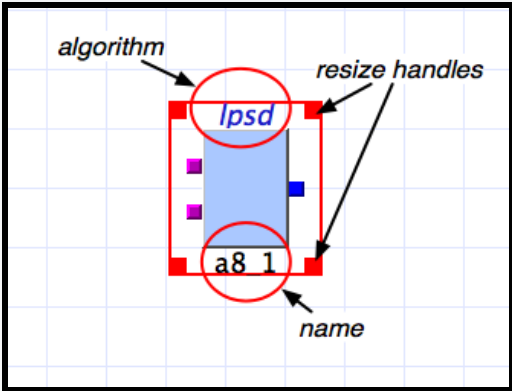
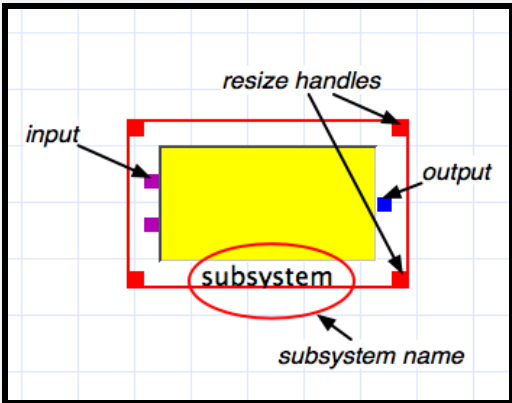
- [Block types](#)
- [Adding blocks to the Canvas](#)
- [Setting block properties and parameters](#)
- [Connecting blocks](#)
- [Creating subsystems](#)

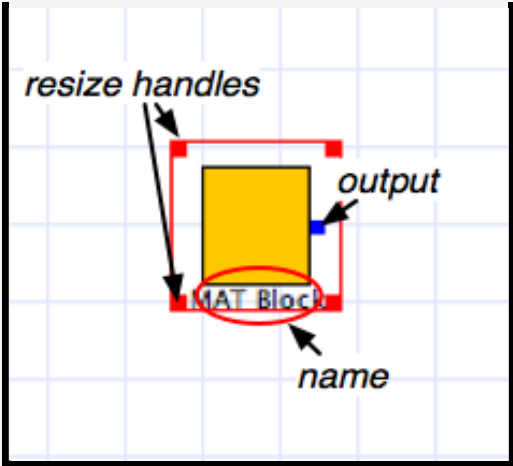
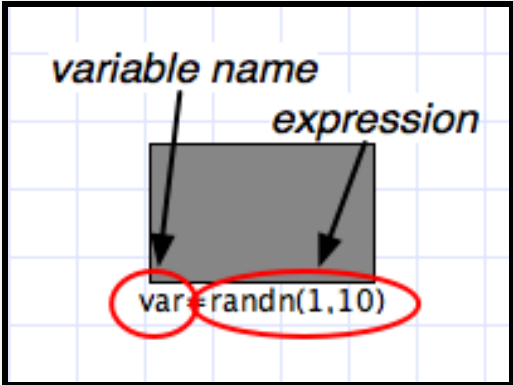
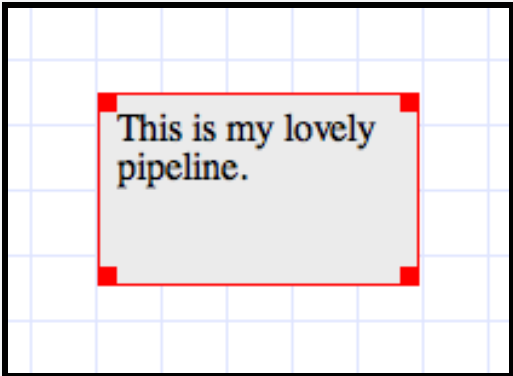
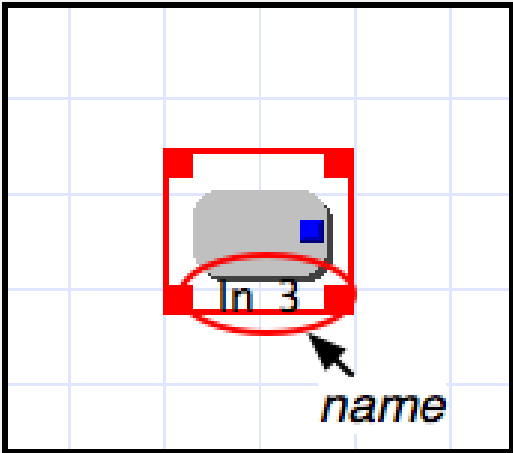
Block types

Various types of elements can be present on a pipeline. Blocks can have different states: idle, ready, or executed. These are color coded as

idle	ready	executed

The following table describes these elements:

Block	Name	Java Class	Description
	LTPDA Block	MBlock	A block that represents a method of one of the LTPDA user classes. These blocks hold a parameter list (plist) which can be set in the parameter table. They can have any number of input and output ports that the underlying algorithm supports.
	Subsystem Block	MSubsystem	A block that represents a subsystem. This is a view of another pipeline that can be placed on a canvas.
	MATLAB Expression Block	MATBlock	A block evaluates a MATLAB expression. The result is stored in the variable and can be

			passed to subsequent blocks.
	MATLAB Constant Block	MConstant	A block evaluates a MATLAB expression and stores the result in the MATLAB workspace with the given variable name.
	Annotation Block	MAnnotation	A block containing editable text to allow for annotating pipelines.
	Input Terminal	MTerminal	A block which represents an input terminal to a subsystem. These blocks can only be placed on a subsystem canvas.
	Output Terminal	MTerminal	A block which represents an output terminal of a subsystem. These blocks

			can only be placed on a subsystem canvas.

◀ Building pipelines by hand

Adding blocks to the canvas ▶

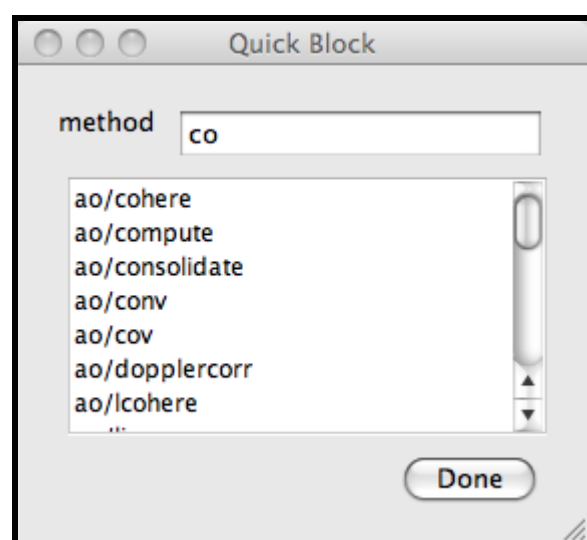
Adding blocks to the canvas

LTPDA Algorithm Blocks

To add an LTPDA Algorithm block to the canvas, select the block in the LTPDA library, and either

- drag the block to the canvas
- hit return to add the block to the canvas
- right-click on the library entry and select 'add block'

You can also use the "Quick Block" dialog. This is especially useful if you know the name of the block you are looking for. To open the Quick Block dialog, hit `ctrl-b` (`cmd-b` on Mac OS X) on the Canvas.



To get the block you want, just start typing in the "method" edit field. Once the block you want is top of the list, just hit enter to add it to the canvas. You can also double-click on the block list to add any of the blocks in the list.

MATLAB Expression Blocks

To add a MATLAB Expression block to the canvas, right-click on the canvas and select 'Additional Blocks -> MATBlock' from the context menu.

MATLAB Constant Blocks

To add a MATLAB Constant block to the canvas, right-click on the canvas and select 'Additional Blocks -> Constant' from the context menu.

Annotation Blocks

To add an annotation block to the canvas, right-click on the canvas and select 'Additional Blocks -> Annotate' from the context menu.



©LTP Team

Setting block properties and parameters

The different block types have different properties that the user can set.

LTPDA Algorithm Blocks

LTPDA Algorithm blocks (MBlocks) have both *properties* and *parameters*. Properties of an MBlock are

Property	Description
Name	The name of the block as it appears on the canvas Block names are unique on a canvas. This is also the string that will be converted to a valid MATLAB variable name when the pipeline is executed.
Modifier	Set this block to be a modifier or not. For more details on modifier blocks in LTPDA see Calling object methods . The accepted values are "true" or "false".

To set the properties of a block, select one or more MBlocks, then double click in the value column entry for the property you want to change. Enter the new value and press return/enter.

Setting the parameter list

LTPDA Algorithm Blocks also have parameters which translate as a parameter list upon execution. To set the parameters of a block, click on a block (or multiple MBlocks which represent the same LTPDA algorithm). You will then see the 'current parameters' that the block holds. To edit the 'key' or 'value' of a parameter, double click the table entry you want to edit, enter the new value, and hit enter or click OK.

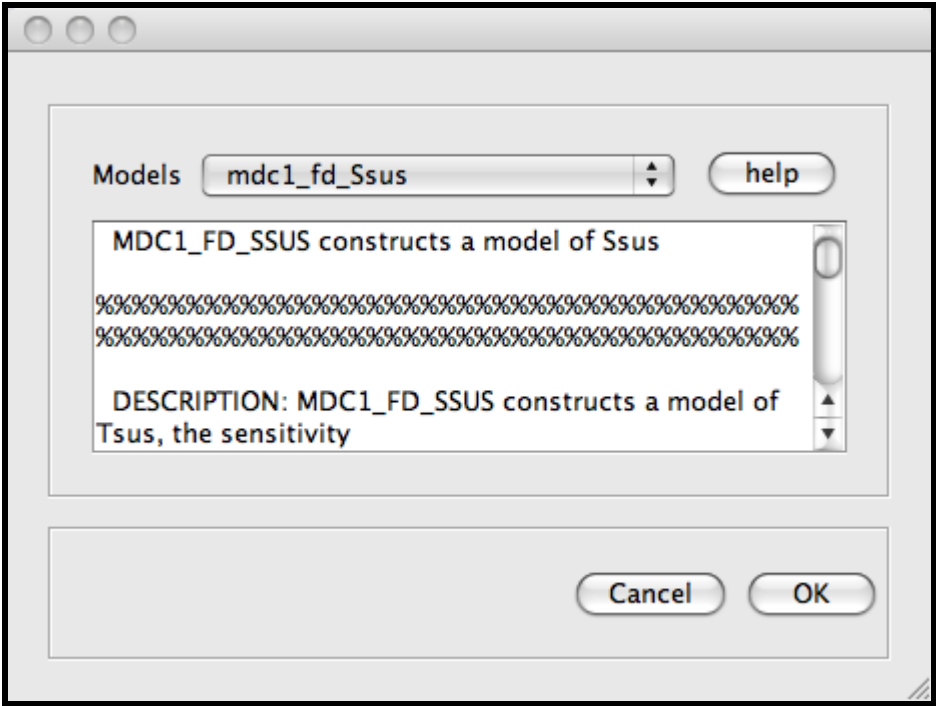
To add or remove parameters from this list use the 'plus' and 'minus' buttons.

You can also select a set of predefined parameter sets from the drop-down menu above the parameter table. Having selected a parameter set, you need to click the 'set' button to push these parameters to the block. You can then go ahead and add or remove parameters from the 'current parameters' on the block.

Editing of most parameter keys and values is done in a simple editor dialog box. However, there are some key/value pairs which are edited using special dialog boxes:

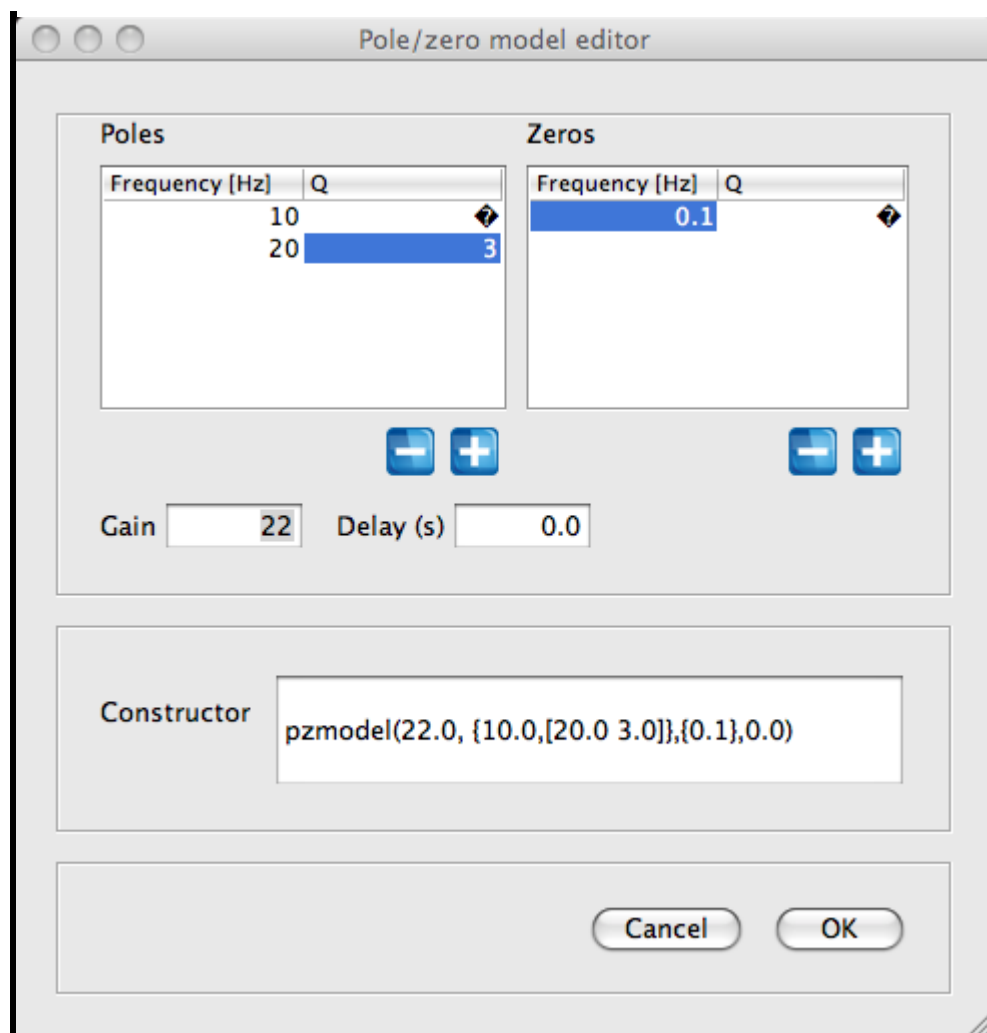
Built-in models of AO and SSM classes

Both the AO and the SSM classes can be built from pre-defined, built-in models. These are typically created with a plist containing the key `BUILT-IN`. If you try to edit the value for this key for one of these constructors, you will be presented with a dialog box that allows you to choose from the built-in models. For all other classes, editing the value for the key `BUILT-IN` is done via a standard input dialog.



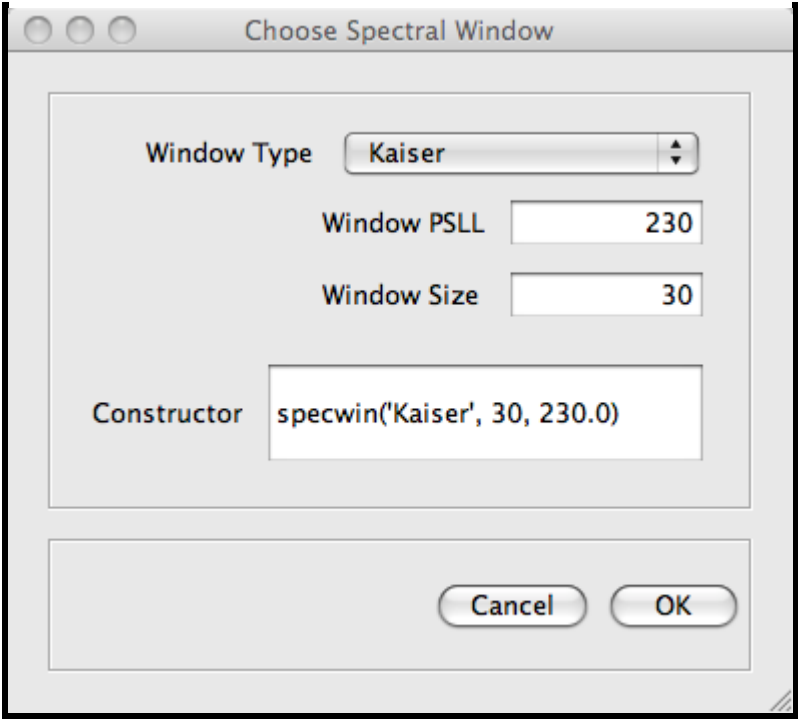
Pole/zero model editor

If any block has a parameter with the key PZMODEL then the corresponding value will be edited via the Pole/zero model editor. Here you can type directly in the constructor edit box, or you can add/remove poles and zeros from the lists. To edit the frequency or Q of a pole or zero, double-click on the table entry. To enter a real pole or zero (no Q), set the Q to 'NaN'.



Spectral window selector

Many algorithms in LTPDA accept a parameter with the key `WIN` for a spectral window parameter. Editing the value for such a parameter presents the user with a dialog where the spectral window can be selected from the list of supported windows. You can also type the constructor directly in the edit box.



Repository hostname selector

Editing parameters with the key `hostname` will give the user a dialog containing the pop-up menu of possible hostnames. This list of hostnames is taken from the LTPDA Preferences. If the preferences are changed, the workbench needs to be closed and reopened for the changes to propagate.

Filenames

If the parameter list contains a parameter with the key `FILENAME`, this will be edited using standard file dialog boxes. If the block algorithm is `save` a save dialog is presented. In all other cases, a load dialog is presented.

MATLAB Expression Blocks

MATLAB Expression blocks have two properties:

Property	Description
Name	The name of the block as it appears on the canvas Block names are unique on a canvas. This is also the string that will be converted to a valid MATLAB variable name when the pipeline is executed.
Expression	This is the (valid) MATLAB expression which, when evaluated, will be set to the variable name.

To set the properties of a block, select one or more MATBlocks then double click in the value column entry for the property you want to change. Enter the new value and press return/enter. Alternatively, you can double-click on a MATBlock to get a dialog box where you can enter the expression.

MATLAB Constant Blocks

Setting of properties on a MATLAB Constant block is just the same as MATBlocks; these blocks only differ in the way they are handled at the time of execution.

Annotation Blocks

To set the text of an Annotation block, double click on the text area to start editing. Click off the block to end editing.

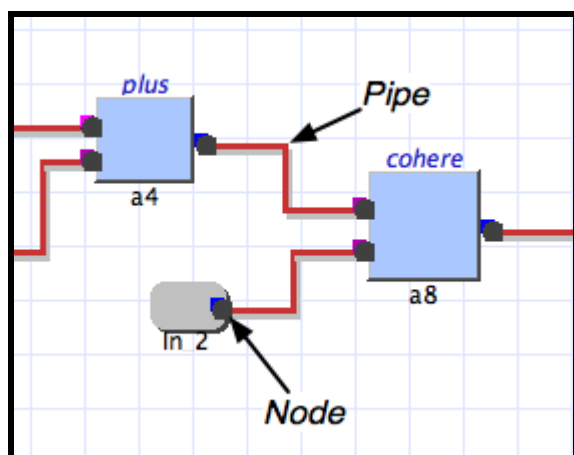
◀ Adding blocks to the canvas Connecting blocks ▶

©LTP Team

Connecting blocks

Blocks of type "LTPDA Block" (MBlock), "Subsystem Block" (MSubsystem), "MATLAB Expression Block" (MATBlock), "Terminal Block" (MTerminal) all have input ports or output ports, or both.

These ports are connected together with "pipes" (MPipe is the underlying java class). Output ports can have more than one pipe connected; input ports can have only one pipe at a time. The binding object between a port and pipe is a "node" (MNode is the underlying java class). Nodes are displayed as small black circles.



To connect these blocks together, do one of the following:

- Click and drag from one port to another.
- Click and drag from one output node to an input port.
- Click and drag from one output port to a block. Connection is made to the first free input (if there is one).
- Click and drag from one output node to a block. Connection is made to the first free input (if there is one).
- Select a source block, then ctrl-left-click a destination block to join the two. There must be at least one free input on the destination block. On the source block, the next free output is used, or the first output if no free outputs are available.

Creating subsystems

To create a subsystem on a canvas, right-click on the canvas and select "create subsystem" from the context menu.

All selected blocks will be placed in the subsystem and all connections will be updated accordingly.

To edit a subsystem canvas, double-click on the subsystem block to open the corresponding canvas.

To add new inputs or outputs to a subsystem do one of:

- right-click on the subsystem canvas and select "Additional Blocks -> Input (or Output)"
- right-click on the subsystem block and select "Add Input" or "Add Output"; the corresponding terminals are placed on the subsystem canvas.

◀ Connecting blocks	Using the Workbench Shelf ▶

Using the Workbench Shelf

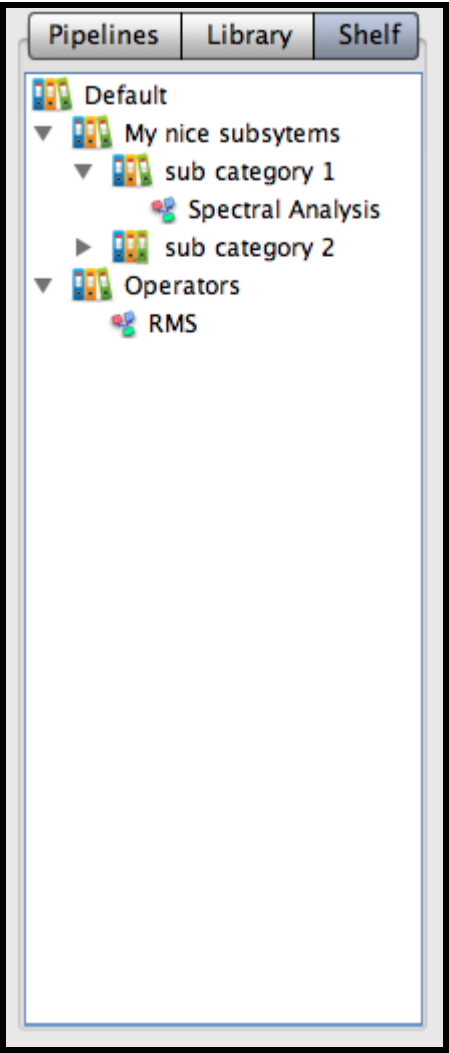
The Workbench 'Shelf' offers the user the possibility for storing subsystems for later use. Different categories and sub-categories can be created and subsystems can be added from the current pipeline to the shelf. The subsystems in the shelf can then be added to other pipelines at a later time. The shelf is persistent across restarts of the workbench.

It is also possible to export shelf categories to disk and then import them again. This offers the possibility to exchange shelf categories containing pre-built subsystems.

- [Accessing the shelf](#)
- [Creating shelf categories](#)
- [Adding and using shelf subsystems](#)
- [Importing and exporting shelf categories](#)

Accessing the shelf

Access to the workbench shelf is through the tabbed pannel on the left of the workbench. The following figure shows the shelf.



Here you can see the shelf has various sub-categories denoted by the book icons. In addition we see some subsystems in the sub-categories, denoted by the small pipeline icon.

Creating shelf categories

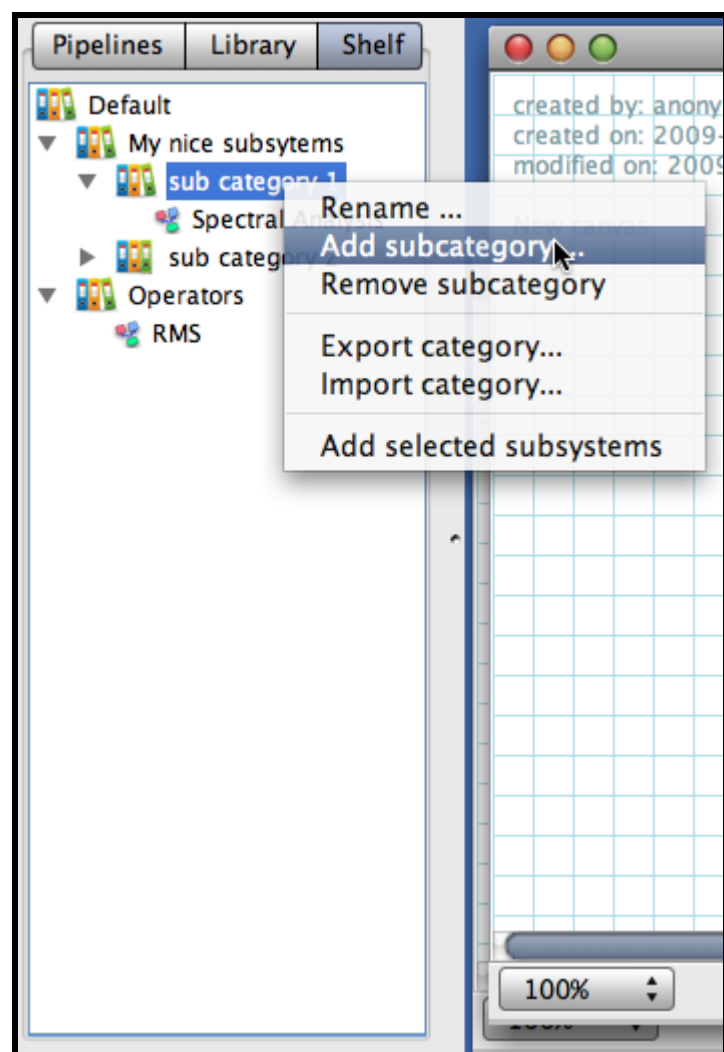
Adding and removing categories is done via the shelf context menu.

Add/remove a category

To add a category or sub-category, right-click on an existing category and choose 'Add sub-category...'.
 Removing a category or sub-category is similar: right-click on an existing category and select 'Remove sub-category'.

The root node of the shelf is special: this cannot be removed and only categories can be added by right-clicking and selecting 'Add category...'.
 Removing a category or sub-category is similar: right-click on an existing category and select 'Remove sub-category'.

The root node of the shelf is special: this cannot be removed and only categories can be added by right-clicking and selecting 'Add category...'.
 Removing a category or sub-category is similar: right-click on an existing category and select 'Remove sub-category'.



©LTP Team

Adding and using shelf subsystems

Adding subsystems to the shelf can be done in two ways:

1. via the context menu on the shelf
2. via the context menu on a subsystem

Adding subsystems via the shelf

To add one or more subsystems to the shelf from the current pipeline, do the following:

1. Select one or more subsystems on the current pipeline
2. Right-click on a category in the shelf and choose 'Add selected subsystems'

Adding subsystems via the subsystem context menu

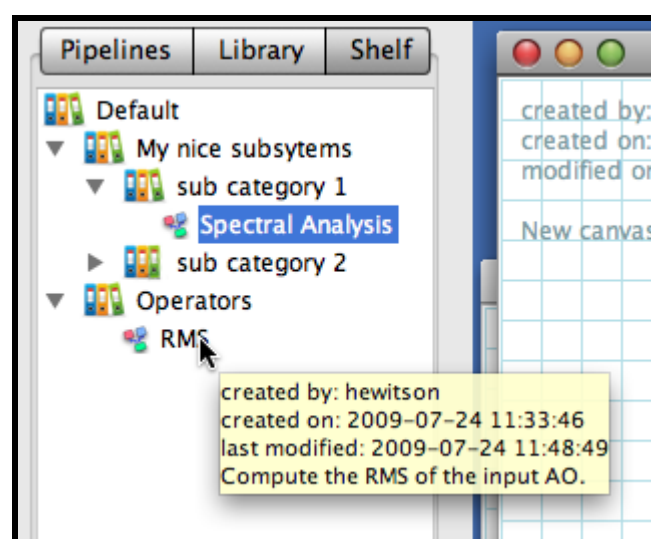
To add a particular subsystem to the shelf, do the following:

1. Select a category in the shelf
2. Right-click on a subsystem on the current canvas and select 'Put on shelf'

Using subsystems from the shelf

Subsystems can be dragged from the shelf to the current pipeline. You can also right-click on a subsystem in the shelf and choose 'Add to pipeline'.

Hovering the mouse over a subsystem in the shelf reveals some information about the subsystem:



If you want to edit or rename a subsystem from the shelf, add it to a pipeline, then make the required edits, then re-add the edited subsystem to the shelf. When satisfied, you can remove the old subsystem from the shelf by right-clicking on the subsystem and choosing 'Remove from shelf'.

◀ Creating shelf categories	Importing and exporting shelf categories ▶

©LTP Team

Importing and exporting shelf categories

Shelf categories can be exported and imported.

Export a category

To export a shelf category, right-click on the shelf category and choose 'Export category...'. Then enter a filename to save the category to. Categories are saved to disk in an XML format; the files have extension '.cat'.

Import a category

To import a category, right click on the shelf root or on an existing category, and choose 'Import category...'. Choose a '.cat' file from the disk.

Execution plans

Content needs written...

 Importing and exporting shelf categories

Editing the plan 

©LTP Team

Editing the plan

Content needs written...

 Execution plans

Linking pipelines 

©LTP Team

Linking pipelines

Content needs written...

◀ Editing the plan

Building pipelines programatically ▶

©LTP Team

Executing pipelines

Content needs written...

◀ Building pipelines programatically

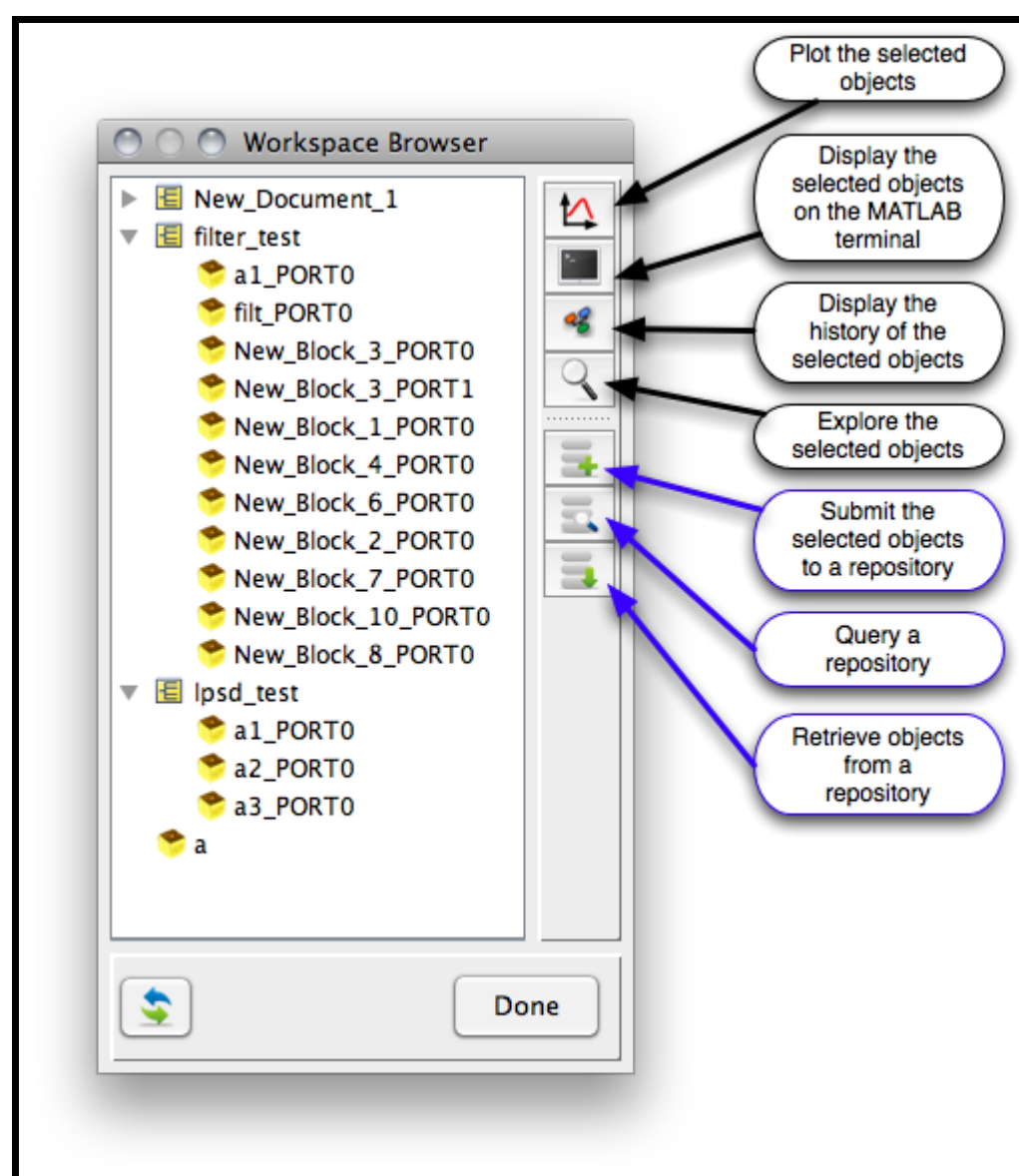
The LTPDA Workspace Browser ▶

©LTP Team

The LTPDA Workspace Browser

The LTPDA Workspace Browser allows you to explore the current MATLAB workspace from the point-of-view of LTPDA objects. Only LTPDA user-objects are shown in the list of current variables. In addition, structures containing LTPDA user-objects are also shown (such as those created by the LTPDA Workbench).

The figure below shows the workspace browser with some LTPDA user objects in the workspace. Hovering the mouse over an object reveals a tooltip containing details about the object.



The button panel on the right of the workspace browser allows you to perform various actions on the selected objects, as well as providing a convenient way to interact with an LTPDA repository (see [Working with an LTPDA Repository](#) for more details).



©LTP Team

The pole/zero model helper

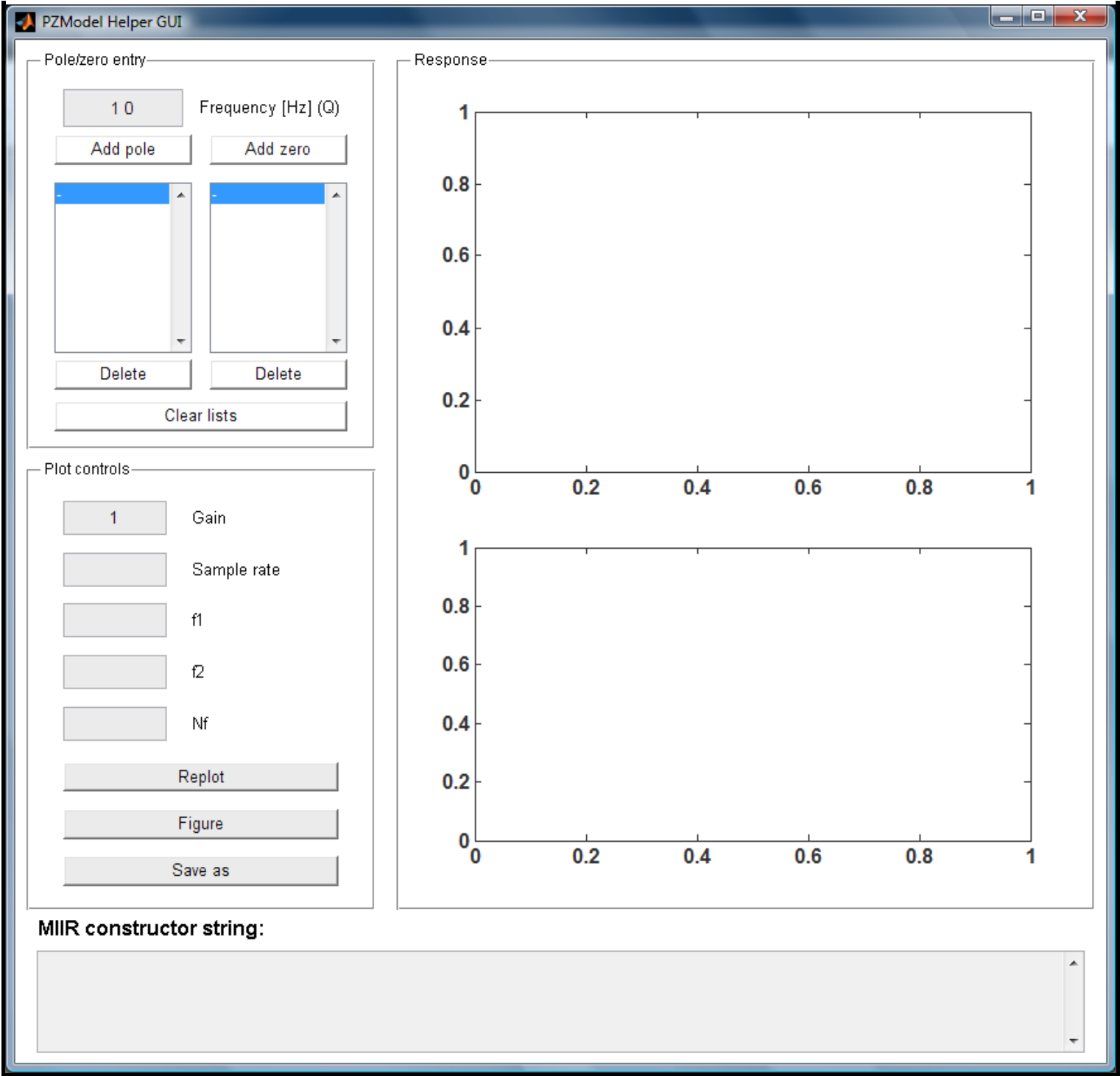
The LTPDA toolbox contains a class (`pzmodel`) for creating and using pole/zero models. The pole/zero model helper GUI allows the user to visualise the pole/zero model as it's being designed. It also allows the user to quickly see how the corresponding IIR filter (`miir` object) will look for different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

To start the pole/zero model helper:

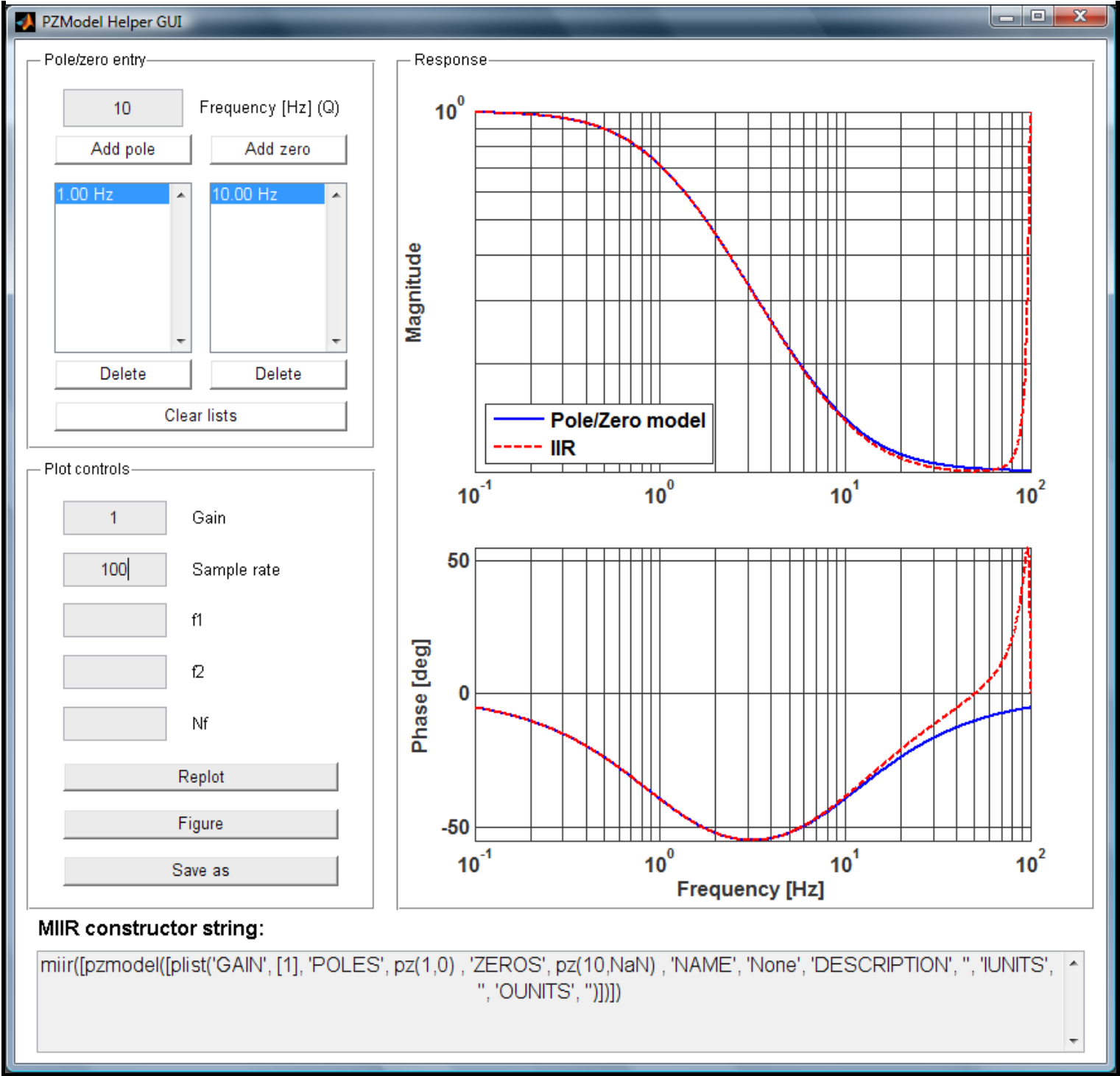
```
>> pzmodel_helper
```

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.



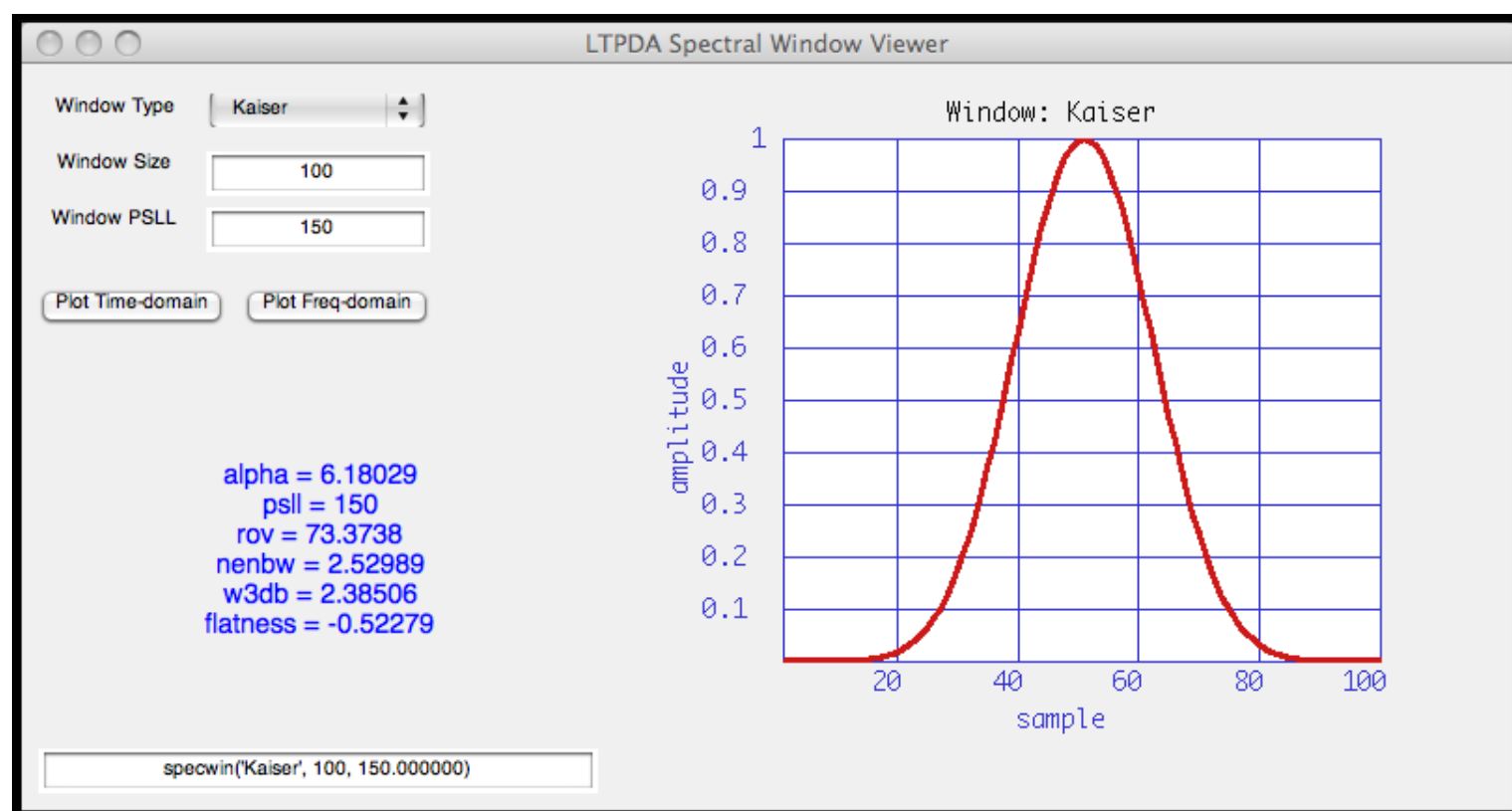
The Spectral Window GUI

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 constructor helper

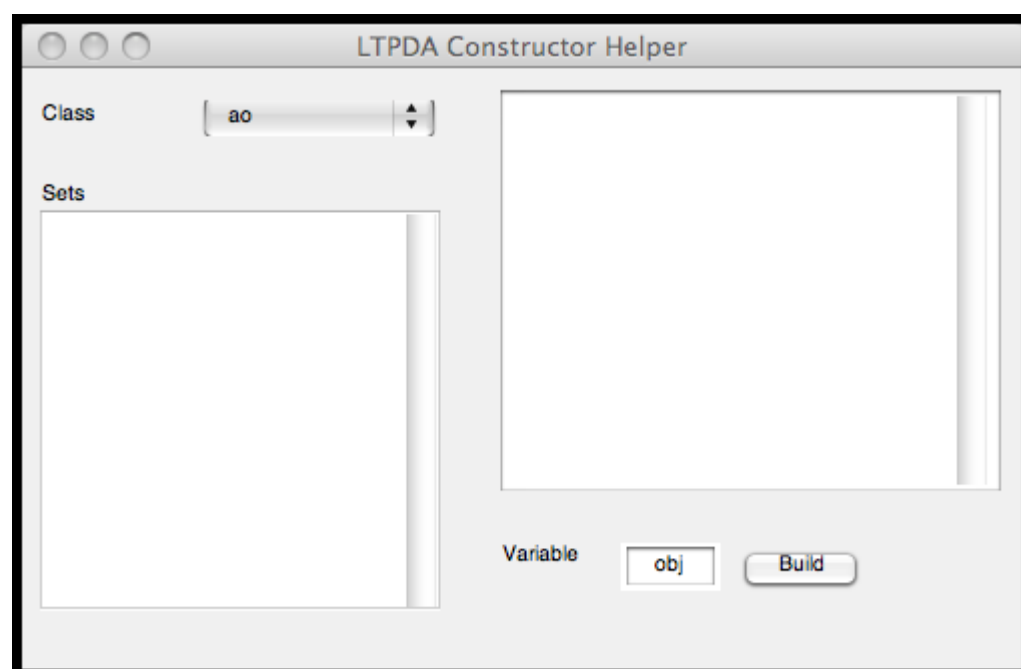
Since LTPDA is an object-oriented system, the user must create objects of different types using the appropriate constructors. The various constructor forms for each different LTPDA class can be explored using the constructor helper.

To start the constructor helper GUI:

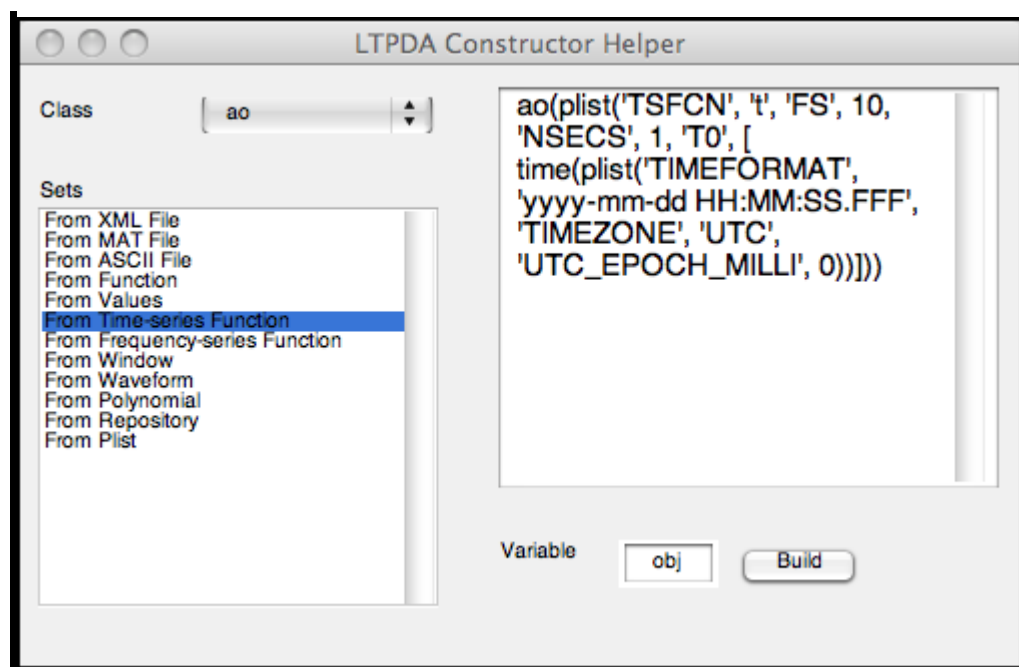
```
>> ltpda_constructor_helper
```

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 ▶

©LTP Team

The LTPDA object explorer

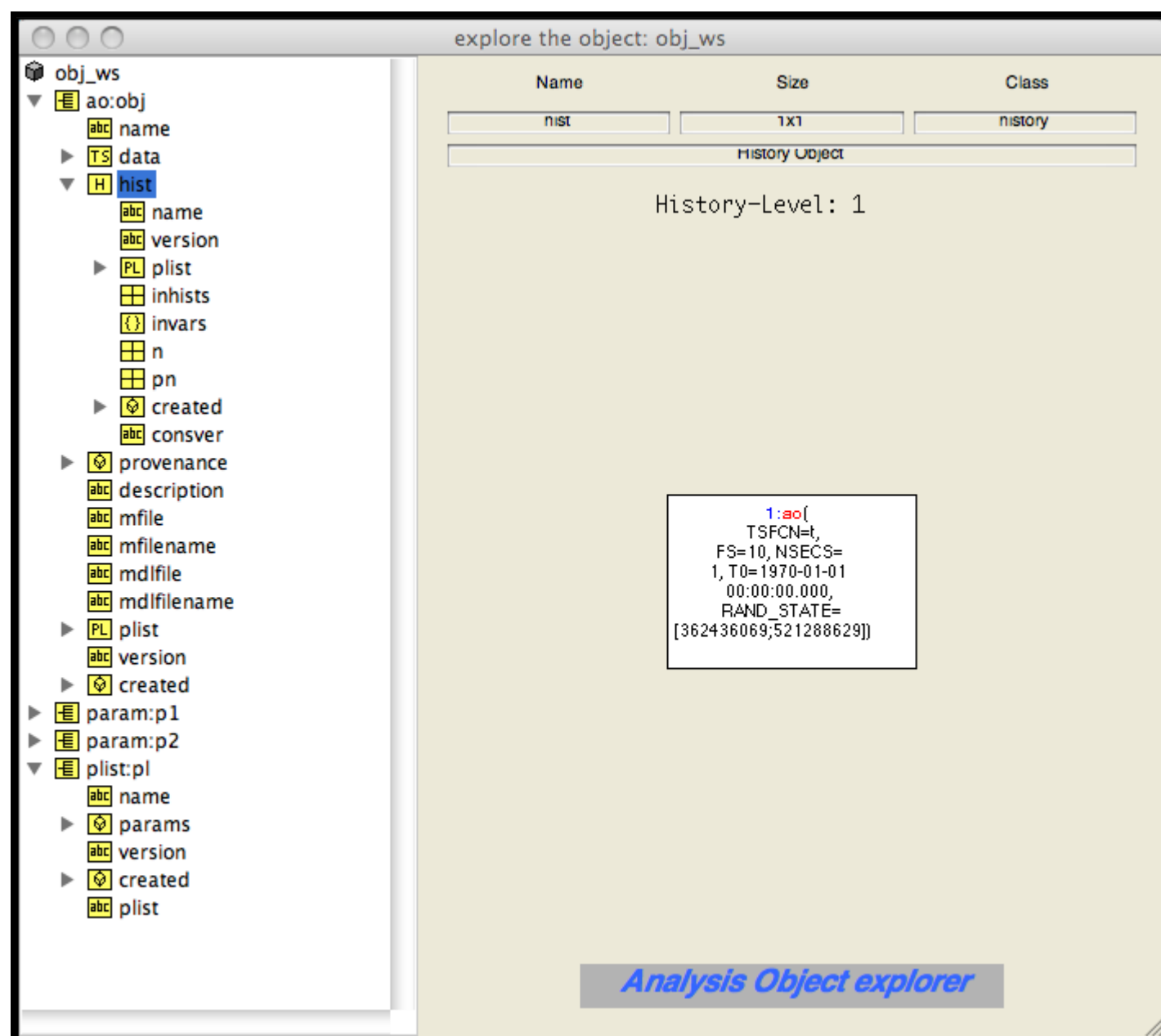
Since LTPDA works mainly with complex object types, it is often useful to explore the content of these objects graphically, particularly for Analysis Objects which may contain deep history trees. To do this, LTPDA offers the object explorer.

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

Working with an LTPDA Repository ▶

©LTP Team

Working with an LTPDA Repository

This section introduces the LTPDA Repository.

- [What is an LTPDA Repository](#)
- [Connecting to an LTPDA Repository](#)
- [Submitting LTPDA objects to a repository](#)
- [Exploring an LTPDA Repository](#)
- [Retrieving LTPDA objects from a repository](#)

What is an LTPDA Repository

Introduction

An LTPDA repository has at its core a database server (in fact, a [MySQL server](#)). A single MySQL server can host multiple databases (LTPDA repositories). A single database/repository comprises a particular set of database tables. These tables hold meta-data about the objects stored in the database.

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 `submit` and the `From Repository` constructors, or the [LTPDA Workspace Browser](#).. In addition, the LTPDA Workbench has full built-in support for interacting with LTPDA Repositories.

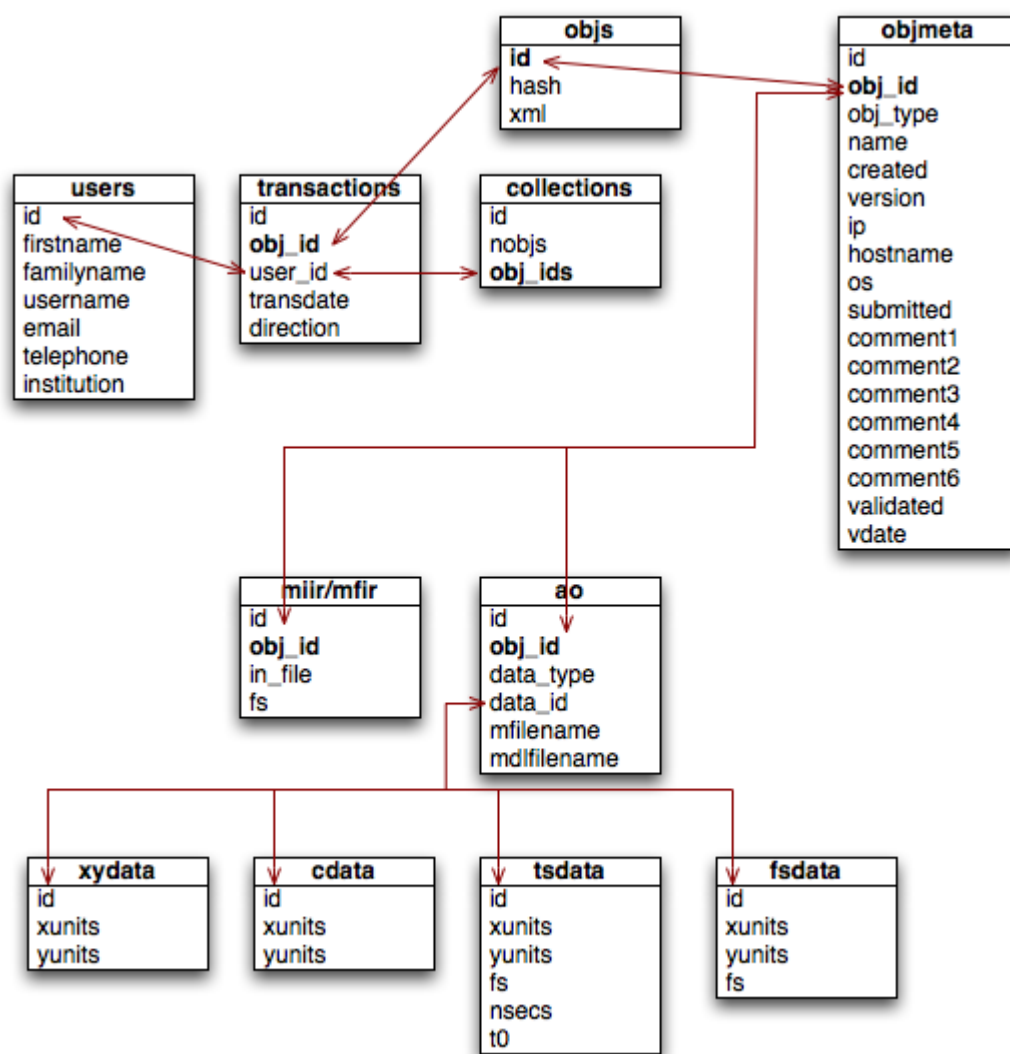
Any standard MySQL client can be used to query and search an LTPDA repository. For example, using a web-client or the standard MySQL command-line interface. In addition, the LTPDA Toolbox provides two ways to search the database: using the workspace browser, or the LTPDA Workbench.. It is also possible 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 Structured 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

The main database tables

An LTPDA repository consists of the following database tables:

objs table

The objs table stores the XML representation of the submitted object. At this point, each object in the database is assigned a unique identifier. Together with the database name and hostname/ip of the server, this forms a unique tag for all LTPDA objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all LTPDA objects in this database. This value is the link between all database tables.

hash	text	An MD5 hash of the XML representation of the object.
xml	longtext	The XML representation of the object. This field can be dumped directly to an XML file and should be readable in LTPDA.

objmeta table

The objmeta table stores various pieces of information associated with the object being submitted. The aim of this table is to provide a lot of useful fields on which to perform searches and queries.

Field	Data Type	Description
id	int(11)	A unique identification for all entries in this table.
obj_id	int(11)	The object id of the object in the objs table.
obj_type	text	The (LTPDA) class of this object.
name	text	The user-assigned name of this object.
created	datetime	The date and time this object was created.
version	text	The CVS tag of the object constructor code.
ip	text	The IP address of the machine which submitted the object.
hostname	text	The hostname of the machine which submitted the object.
os	text	The operating system of the machine which submitted the object.
submitted	datetime	The date and time the object was submitted.
experiment_title	text	A title for the experiment associated with the object.
experiment_desc	text	A description of the experiment associated with the object.
analysis_desc	text	A description of the analysis associated with the object.
quantity	text	If applicable, the physical quantity associated with the data in the object.
additional_authors	text	Any additional people involved in creating

		this object.
additional_comments	text	A free-form field of additional comments.
keywords	text	A list of keywords associated with the object.
reference_ids	text	ID numbers of any other objects associated with this object.
validated	tinyint(1)	A boolean field intended to indicate validated objects.
vdate	datetime	The date/time the object was validated.

transactions table

The transactions table records all user transactions. A transaction corresponds to submitting or retrieving a single or a collection of LTPDA objects.

Field	Data Type	Description
id	int(11)	A unique identification number this table entry.
obj_id	int(11)	The object id of the object in the objs table.
user_id	int(11)	The unique ID number of the user who carried out the transaction.
transdate	datetime	The date/time of the transaction.
direction	text	The direction of the transaction: 'in' or 'out'.

users table

The users table stores information about the users allowed to access the database.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in this table.
firstname	text	The firstname of the user.
familyname	text	The family name of the user.
username	text	The username (login name) of the user.
email	text	A valid e-mail address for the user.
telephone	text	A telephone numnber for the user.

institution	text	The institution of the user.

collections table

The `collections` table stores virtual collections of objects submitted to the database. When the user submits one or more objects at the same time, this constitutes a collection. In this case a collection ID number is assigned next to a list of the object IDs in the collection. This allows the user to retrieve collections of objects based on the collection ID alone; no information about the individual object IDs is required.

Field	Data Type	Description
id	int(11)	A unique identification number for this collection of objects.
nobjs	int	The number of objects in the collection.
obj_ids	text	A comma separated list of object IDs.

Additional database tables

As well as the main database tables, additional meta-data tables are used to capture extra meta-data about some of the LTPDA objects.

ao table

The `ao` table stores additional meta-data specific to analysis objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in the table.
obj_id	int(11)	The unique ID of the object.
data_type	text	The type/class of the data stored in the AO.
data_id	int(11)	The unique ID of the data object listed in one of the data meta-data tables.
description	text	The description property of the AO.
mfilename	text	The filename of any m-file attached to the AO.
mdlfilename	text	The filename of any SIMULINK model file attached to the AO.

miir table

The `miir` table stores additional meta-data specific to miir filter objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in the table.
obj_id	int(11)	The unique ID of the object.
in_file	text	The input filename (if applicable) used to create the filter object
fs	int(11)	The sample rate of the data the filter is designed for.

mfir table

The mfir table stores additional meta-data specific to mfir filter objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in the table.
obj_id	int(11)	The unique ID of the object.
in_file	text	The input filename (if applicable) used to create the filter object
fs	int(11)	The sample rate of the data the filter is designed for.

tsdata table

The tsdata table stores additional meta-data specific to tsdata (time-series data) objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in the table.
xunits	text	The X-units associated with this time-series.
yunits	text	The Y-units associated with this time-series.
fs	int(11)	The sample rate of the data the filter is designed for.
nsecs	int	The duration (number of seconds) of data in the object.
t0	datetime	The date/time associated with the start (first sample) of the time-series.

fsdata table

The fsdata table stores additional meta-data specific to fsdata (frequency-series data) objects.

Field	Data Type	Description

id	int(11)	A unique identification number for all entries in the table.
xunits	text	The X-units associated with this time-series.
yunits	text	The Y-units associated with this time-series.
fs	int(11)	The sample rate of the data the filter is designed for.

cdata table

The cdata table stores additional meta-data specific to cdata (1D data) objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in the table.
xunits	text	The X-units associated with this time-series.
yunits	text	The Y-units associated with this time-series.

xydata table

The xydata table stores additional meta-data specific to xydata (2D data) objects.

Field	Data Type	Description
id	int(11)	A unique identification number for all entries in the table.
xunits	text	The X-units associated with this time-series.
yunits	text	The Y-units associated with this time-series.

Connecting to an LTPDA Repository

Connection to an LTPDA Repository is normally carried out as part of other processes. For example, when submitting an object to the repository using the `submit` command, the user is prompted to login to the chosen repository.

It is also possible to connect to a repository using a script interface. The result is a connection object which can be passed to other methods (for example, `submit`), thus allowing the construction of more automated procedures for interacting with repositories.

To connect to a repository using the script interface, us the command `utils.jmysql.connect`. It takes the following input arguments:

hostname	A hostname for the repository
dbname	A database name to connect to
username	[optional] A username to connect with
username	[optional] A password to connect with

If you don't supply them as input arguments, you will then be prompted for a valid username and password.

The result is a `mpipeline.repository.RepositoryConnection` object which can be further used to interact with the repository.

To disconnect from the server, use the `closeConnection` method of the `mpipeline.repository.RepositoryConnection` class:

```
>> conn.closeConnection
```


Submitting LTPDA objects to a repository

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

There are three different methods which submit/update object(s) to the repository:

- **submit**

Submits the given collection of objects to an LTPDA Repository. If multiple objects are submitted together, a corresponding collection entry will be made. The objects are stored as a XML representation **and** if possible a binary representation (see bsubmit).

- **bsubmit**

Submits the given collection of objects to an LTPDA Repository. If multiple objects are submitted together, a corresponding collection entry will be made. The objects are stored **only** as a binary representation.

In order to retrieve this object by calling a constructor it is necessary to set the key 'BINARY' to 'yes' because the XML representation doesn't exist. For example the AO constructor ['From Repository'](#)

- **update**

Updates the specified object with a new object.

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. An attempt is made to create a binary representation of the object (.MAT). If this is possible, the binary data is inserted in the `bobjs` table.
 8. Various pieces of meta-data (object name, object type, created time, client IP address, etc.) are submitted to the `objmeta` table
 9. Additional meta-data is entered into the table matching the object class (`ao`, `tsdata`, etc.)
 10. 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 `submit`. This command takes at least two inputs:

object	The object(s) to

	submit
pl	A plist containing various pieces of information (see below)

The `plist` details can be found at: [Parameter Sets](#)

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

```
% Connect to a repository
conn = utils.jmysql.connect('localhost', 'ltpda_test');

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

% Build an information plist
pl = plist;
pl.append('conn', conn);
pl.append('experiment title', 'Interferometer noise');
pl.append('experiment description', 'Spectral estimation of interferometer output
signal');
pl.append('analysis description', 'Spectrum of the recorded signal');
pl.append('quantity', 'photodiode output');
pl.append('keywords', 'interferometer, noise, spectrum');
pl.append('reference ids', '');
pl.append('additional comments', 'none');
pl.append('additional authors', 'no one');

% Submit the AO
[ids, cid] = submit(a, pl);

% Close the connection
conn.closeConnection
```

The ID assigned to the submitted object is contained in the first output of the `submit` function:

```
% Inspect the object ID
disp(ids)
212
```

If the `plist` contains sufficient information, it is possible to override the appearance of the 'submission' dialog by giving the key/value pair: 'no dialog', true.

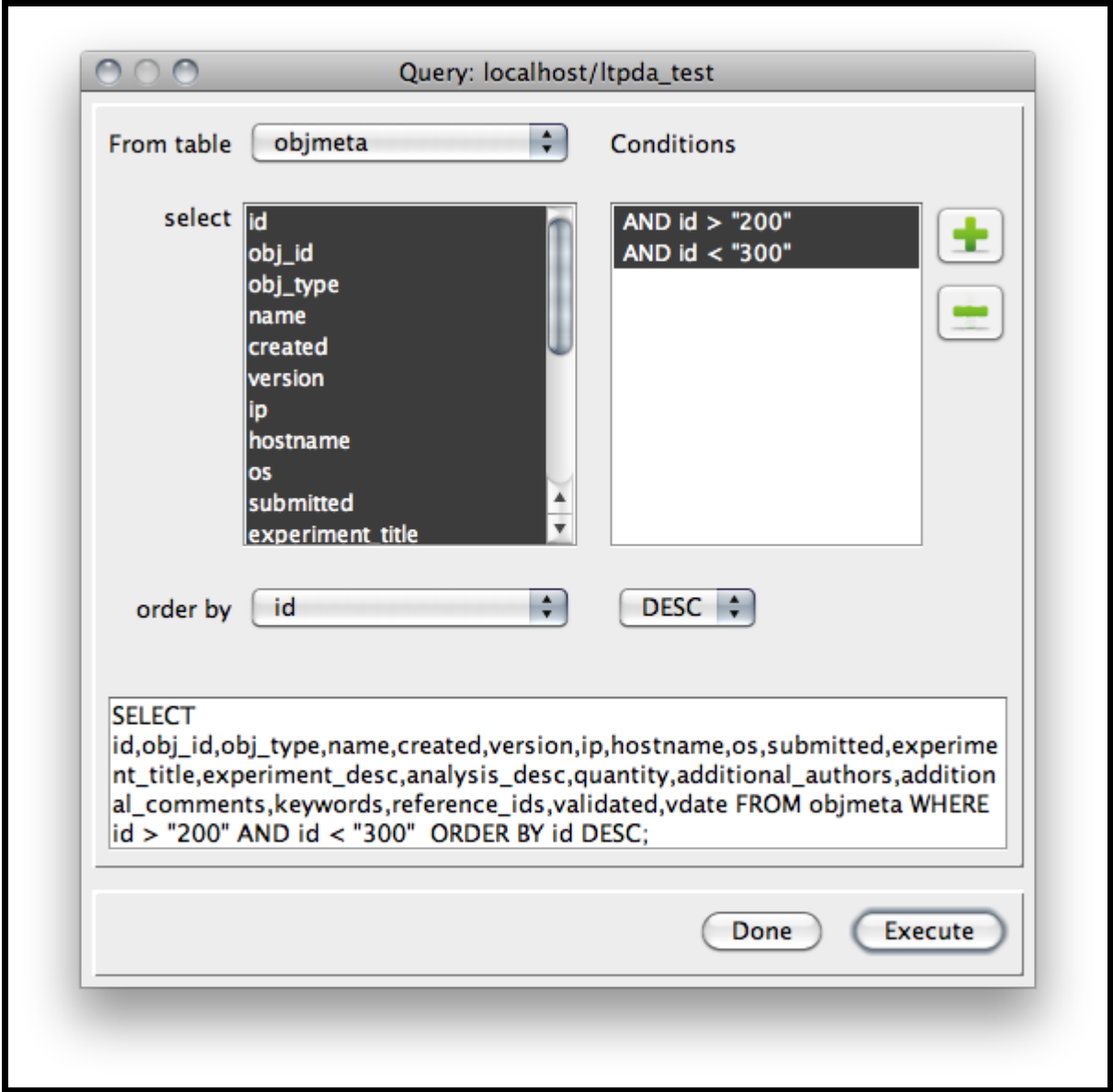
Submitting collections

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.

Exploring an LTPDA Repository

Exploring an LTPDA repository is most easily achieved using the purpose-built graphical user interface. This interface is accessible either from the LTPDA workbench, or via a toolbar button in the workspace browser.

The figure below shows the query dialog that appears through either interface:



You can construct a query by use of the drop-down menus and buttons, then execute the query to retrieve a table of results, like the one shown below:

SELECT
id,obj_id,obj_type,name,created,version,ip,hostname,os,submitted,experiment_title,experiment_desc,analysis_desc,quantity,additional_authors,additional_comments,keywords,reference_ids,validated,vdate
FROM objmeta WHERE id > "200" AND id < "300" ORDER BY id DESC;

id	obj_id	obj_type	name	created	version	ip	hostname	os
299	303	ao	at5	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
298	302	ao	at2	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
297	301	ao	at4	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
296	300	ao	at1	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
295	299	miir	highpass	2009-09...	\$Id: miir....	192.168...	martin-he...	MACi6
294	298	smodel	None	2009-09...	\$Id: smod...	192.168...	martin-he...	MACi6
293	297	ao	at1	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
292	296	ao	at6	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
291	295	ao	at3	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
290	294	ao	at5	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
289	293	ao	at2	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
288	292	ao	at4	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
287	291	ao	at1	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6
286	290	ao	at6	2009-09...	\$Id: ao.m,...	192.168...	martin-he...	MACi6

Done

If the query dialog is launched from within the LTPDA Workbench, the results table has an additional button which can be used to create constructor blocks on the current pipeline corresponding to the selected records in the results table.

◀ Submitting LTPDA objects to a repository

Retrieving LTPDA objects from a repository ▶

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 class constructors to retrieve objects. In addition, to retrieve a collection of objects, the `collection` class can be used.

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, or the ID of the collection that contains that object. If you don't know the class of the object, you can use the `collection` class as a container, as follows:

```
>> a = collection(plist('hostname', 'localhost', 'database', 'ltpda_test', 'id', 1))
** Connection status:
LWB: Disconnected from localhost/ltpda_test as hewitson
---- collection 1 ----
name: none
num objs: 1
01: ao | New Block/tsdataNdata=[10x1], fs=1, nsecs=10, t0=1970-01-01 00:00:00.000
description:
UUID: 9ec7a613-9442-4451-9b40-af18f4afea00
-----
```

If you already know the class of the object (for example, `ao`), you can directly call the class constructor method:

```
% Define the hostname and database
hostname = 'localhost';
database = 'ltpda_test';

% Retrieve the object
q = ao(plist('hostname', hostname, 'database', dbname, 'ID', 12));
```

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

```
>> a = ao(plist('hostname', 'localhost', 'database', 'ltpda test', 'id', [1 2 3]))
```

Retrieving object collections

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

```
>> a = collection(plist('hostname', 'localhost', 'database', 'ltpda_test', 'cid', 1))
```

The output is a `collection` object containing the objects retrieved.

Retrieving binary objects

The retrieval process may be speeded up by taking advantage of the fact that the objects are stored in the databases also in binary form. This can be achieved by using the parameter 'binary', that will build the object from the corresponding binary representation, if stored in the database.

```
% Retrieve the collection  
q = ao(plist('hostname', hostname, 'database', dbname, 'ID', 12, 'binary', 'yes'));
```

If the binary representation is not in the database, the object will be built from the xml one.

◀ Exploring an LTPDA Repository

Using the LTPDA Repository GUI ▶

©LTP Team

Class descriptions

AO class	Implements analysis objects in the LTPDA toolbox
SSM class	Implements statespace model in the LTPDA toolbox
SMODEL class	Implements symbolic model in the LTPDA toolbox
FILTERBANK class	Implements a filterbank within LTPDA toolbox
MFIR class	Implements finite impulse response filter objects within LTPDA toolbox
MIIR class	Implements infinite impulse response filter objects within LTPDA toolbox
PZMODEL class	Implements pole/zero model objects within LTPDA toolbox
PARFRAC class	Implements partial fraction representation of a transfer function within LTPDA toolbox
RATIONAL class	Implements rational representation of a transfer function within LTPDA toolbox
TIMESPAN class	Implements time span objects within LTPDA toolbox
PLIST class	Implements parameter list objects within LTPDA toolbox
MATRIX class	Implements a matrix class within LTPDA toolbox
COLLECTION class	Implements a collection class within LTPDA toolbox

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

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
data	Data object associated with this AO	ao
version	CVS version string of the constructor	ao
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Arithmetic Operator	
Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Operator	Operator methods
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods
Trigonometry	Trigometry methods

[▲ Back to Top](#)

Arithmetic Operator

Methods	Description	Defined in class
minus	MINUS implements subtraction operator for analysis objects.	ao
mpower	MPOWER implements mpower operator for analysis objects.	ao
mrdivide	MRDIVIDE implements mrdivide operator for analysis objects.	ao
mtimes	MTIMES implements mtimes operator for analysis objects.	ao
plus	PLUS implements addition operator for analysis objects.	ao
power	POWER implements power operator for analysis objects.	ao
rdivide	RDIVIDE implements division operator for analysis objects.	ao
times	TIMES implements multiplication operator for analysis objects.	ao

[▲ Back to Top of Section](#)

Constructor

Methods	Description	Defined in class
ao	AO analysis object class constructor.	ao
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
cat	CAT concatenate AOs into a row vector.	ao
convert	CONVERT perform various conversions on the ao.	ao
demux	DEMUX splits the input vector of AOs into a number of output AOs.	ao
dx	DX Get the data property 'dx'.	ao
dy	DY Get the data property 'dy'.	ao
find	FIND particular samples that satisfy the input query and return a new AO.	ao
fs	FS Get the data property 'fs'.	ao
join	JOIN multiple AOs into a single AO.	ao
len	LEN overloads the length operator for Analysis objects. Length of the data samples.	ao
nsecs	NSECS Get the data property 'nsecs'.	ao
search	SEARCH selects AOs that match the given name.	ao
setDx	SETDX sets the 'dx' property of the ao.	ao
setDy	SETDY sets the 'dy' property of the ao.	ao
setFs	SETFS sets the 'fs' property of the ao.	ao
setT0	SETT0 sets the 't0' property of the ao.	ao
setX	SETX sets the 'x' property of the ao.	ao
setXY	SETXY sets the 'xy' property of the ao.	ao
setXunits	SETXUNITS sets the 'xunits' property of the ao.	ao
setY	SETY sets the 'y' property of the ao.	ao

setYunits	SETYUNITS sets the 'yunits' property of the ao.	ao
setZ	SETZ sets the 'z' property of the ao.	ao
simplifyYunits	SIMPLIFYYUNITS simplify the 'yunits' property of the ao.	ao
t0	T0 Get the data property 't0'.	ao
timeshift	TIMESHIFT for AO/tsdata objects, shifts the time axis such that $x(1) = 0$.	ao
validate	VALIDATE checks that the input Analysis Object is reproducible and valid.	ao
x	X Get the data property 'x'.	ao
xunits	XUNITS Get the data property 'xunits'.	ao
y	Y Get the data property 'y'.	ao
yunits	YUNITS Get the data property 'yunits'.	ao
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
char	CHAR overloads char() function for analysis objects.	ao
md5	MD5 computes an MD5 checksum from an analysis objects.	ao
plot	PLOT a simple plot of analysis objects.	ao
pwelch	PSD makes power spectral density estimates of the time-series objects	ao
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

MDC01

Methods	Description	Defined in class
mdc1_cont2act_utn	mdc1_cont2act_utn simulate the effect of retarded actuators	ao
mdc1_ifo2acc_fd	MDC1_IFO2ACC_FS calculates the external acceleration in the frequency-domain.	ao
mdc1_ifo2acc_fd_utn	mdc1_ifo2acc_fd_utn convert ifo data to acceleration	ao
mdc1_ifo2acc_inloop	MDC1_IFO2ACC_INLOOP calculates the inloop acceleration in the time-domain.	ao
mdc1_ifo2cont_utn	mdc1_ifo2cont_utn simulate the effect of the controller	ao
mdc1_ifo2control	MDC1_IFO2CONTROL converts the input	ao

time-series to control forces.

mdc1_x2acc	MDC1_X2ACC converts the input time-series ao to acceleration with a time-domain filter

[▲ Back to Top of Section](#)

Operator

Methods	Description	Defined in class
abs	ABS overloads the Absolute value method for Analysis objects.	ao
angle	ANGLE overloads the angle operator for Analysis objects.	ao
complex	COMPLEX overloads the complex operator for Analysis objects.	ao
conj	CONJ overloads the conjugate operator for Analysis objects.	ao
ctranspose	CTRANSPOSE overloads the ' operator for Analysis Objects.	ao
det	DET overloads the determinant function for Analysis objects.	ao
diag	DIAG overloads the diagonal operator for Analysis Objects.	ao
eig	EIG overloads the eigenvalues/eigenvectors function for Analysis objects.	ao
exp	EXP overloads the exp operator for Analysis objects. Exponential.	ao
hypot	HYPOT overloads robust computation of the square root of the sum of squares for AOs.	ao
imag	IMAG overloads the imaginary operator for Analysis objects.	ao
inv	INV overloads the inverse function for Analysis Objects.	ao
ln	LN overloads the log operator for Analysis objects. Natural logarithm.	ao
log	LOG overloads the log operator for Analysis objects. Natural logarithm.	ao

log10	LOG10 overloads the log10 operator for Analysis objects. Common (base 10) logarithm.	ao
lscov	LSCOV is a wrapper for MATLAB's lscov function.	ao
max	MAX computes the maximum value of the data in the AO.	ao
mean	MEAN computes the mean value of the data in the AO.	ao
median	MEDIAN computes the median value of the data in the AO.	ao
min	MIN computes the minimum value of the data in the AO.	ao
mode	MODE computes the modal value of the data in the AO.	ao
norm	NORM overloads the norm operator for Analysis Objects.	ao
offset	OFFSET adds an offset to the data in the AO.	ao
phase	PHASE overloads the ltpda_phase operator for Analysis objects.	ao
real	REAL overloads the real operator for Analysis objects.	ao
scale	SCALE scales the data in the AO by the specified factor.	ao
sign	SIGN overloads the sign operator for Analysis objects.%	ao
sort	SORT the values in the AO.	ao
sqrt	SQRT computes the square root of the data in the AO.	ao
std	STD computes the standard deviation of the data in the AO.	ao
sum	SUM computes the sum of the data in the AO.	ao
sumjoin	SUMJOIN sums time-series signals together	ao
svd	SVD overloads the determinant function for Analysis objects.	ao
transpose	TRANPOSE overloads the .' operator for Analysis Objects.	ao
uminus	UMINUS overloads the uminus operator for Analysis objects.	ao
unwrap	UNWRAP overloads the unwrap operator for Analysis	ao

objects.

var	VAR computes the variance of the data in the AO.	ao

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
display	DISPLAY implement terminal display for analysis object.	ao
export	EXPORT export the data of an analysis object to a text file.	ao
iplot	IPLOT provides an intelligent plotting tool for LTPDA.	ao
iplotyy	IPLOT provides an intelligent plotting tool for LTPDA.	ao
table	TABLE display the data from the AO in a table.	ao
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
ge	GE overloads >= operator for analysis objects. Compare the y-axis values.	ao
gt	GT overloads > operator for analysis objects. Compare the y-axis values.	ao
le	LE overloads <= operator for analysis objects. Compare the y-axis values.	ao
lt	LT overloads < operator for analysis objects. Compare the y-axis values.	ao
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
bin_data	BIN_DATA rebins aos data, on logarithmic scale, linear scale, or arbitrarily chosen.	ao
cohere	COHERE estimates the coherence between time-series objects	ao
compute	COMPUTE performs the given operations on the input AOs.	ao
confint	CONFINT Calculates confidence levels and variance for psd, lpsd, cohere, lcohere and curvefit parameters	ao
consolidate	CONSOLIDATE resamples all input AOs onto the same time grid.	ao
conv	CONV vector convolution.	ao
corr	CORR estimate linear correlation coefficients.	ao
cov	COV estimate covariance of data streams.	ao
cpsd	CPSD estimates the cross-spectral density between time-series objects	ao
curvefit	CURVEFIT fit a curve to data.	ao
delay	DELAY delays a time-series using various methods.	ao
detrend	DETREND detrends the input analysis object using a polynomial of degree N.	ao
dft	DFT computes the DFT of the input time-series at the requested frequencies.	ao
diff	DIFF differentiates the data in AO.	ao
dopplercorr	Dopplercorr coorects data for Doppler shift	ao
downsample	DOWNSAMPLE AOs containing time-series data.	ao
dropduplicates	DROPDUPPLICATES drops all duplicate samples in time-series AOs.	ao

dsmean	DSMEAN performs a simple downsampling by taking the mean of every N samples.	ao
eqmotion	EQMOTION solves numerically a given linear equation of motion	ao
evaluateModel	EVALUATEMODEL evaluate a curvefit model.	ao
fft	FFT overloads the fft method for Analysis objects.	ao
fftfilt	FFTFILT overrides the fft filter function for analysis objects.	ao
filtSubtract	FILTSUBTRACT subtracts a frequency dependent noise contribution from an input ao.	ao
filter	FILTER overrides the filter function for analysis objects.	ao
filtfilt	FILTFILT overrides the filtfilt function for analysis objects.	ao
firwhiten	FIRWHITEN whitens the input time-series by building an FIR whitening filter.	ao
fixfs	FIXFS resamples the input time-series to have a fixed sample rate.	ao
fngen	FNGEN creates an arbitrarily long time-series based on the input PSD.	ao
gapfilling	GAPFILLING fills possible gaps in data.	ao
gapfillingoptim	GAPFILLINGOPTIM fills possible gaps in data.	ao
getdof	GETDOF Calculates degrees of freedom for psd, lpsd, cohere and lcohere	ao
heterodyne	HETERODYNE heterodynes time-series.	ao
hist	HIST overloads the histogram function (hist) of MATLAB for Analysis Objects.	ao
ifft	IFFT overloads the ifft operator for Analysis objects.	ao
integrate	INTEGRATE integrates the data in AO.	ao
interp	INTERP interpolate the values in the input AO(s) at new values.	ao

interpmissing	INTERPMISSING interpolate missing samples in a time-series.	ao
lcohere	LCOHERE implement magnitude-squared coherence estimation on a log frequency axis.	ao
lcpd	LCPSD implement cross-power-spectral density estimation on a log frequency axis.	ao
linSubtract	LINSUBTRACT subtracts a linear contribution from an input ao.	ao
lincom	LINCOM make a linear combination of analysis objects	ao
linedetect	LINEDETECT find spectral lines in the ao/fsdata objects.	ao
lisovfit	LISOVFIT uses LISO to fit a pole/zero model to the input frequency-series.	ao
lpsd	LPSD implements the LPSD algorithm for analysis objects.	ao
ltfe	LTFE implements transfer function estimation computed on a log frequency axis.	ao
metropolis1D	METROPOLIS1D samples a probability distribution for a SISO problem using a Metropolis/Metropolis-Hastings algorithm.	ao
metropolis2D	METROPOLIS2D samples a probability distribution for a MIMO problem	ao
noisegen1D	NOISEGEN1D generates colored noise from white noise.	ao
noisegen2D	NOISEGEN2D generates cross correlated colored noise from white noise.	ao
normdist	NORMDIST computes the equivalent normal distribution for the data.	ao
polyfit	POLYFIT overloads polyfit() function of MATLAB for Analysis Objects.	ao
psd	PSD makes power spectral density estimates of the time-series objects	ao
psdconf	PSDCONF Calculates confidence levels and variance for psd	ao
resample	RESAMPLE overloads resample function for	ao

AOs.

rms	RMS Calculate RMS deviation from spectrum	ao
sDomainFit	sDomainFit performs a fitting loop to identify model order and	ao
select	SELECT select particular samples from the input AOs and return new AOs with only those samples.	ao
smallvector_lincom	SMALLVECTOR_LINCOM does the linear combination for the small vector model.	ao
smallvectorfit	SMALLVECTORFIT does a linear fit to the small vector model.	ao
smoother	SMOOTHER smooths a given series of data points using the specified method.	ao
spectrogram	SPECTROGRAM computes a spectrogram of the given ao/tsdata.	ao
spikecleaning	spikecleaning detects and corrects possible spikes in analysis objects	ao
split	SPLIT split an analysis object into the specified segments.	ao
straightLineFit	STRAIGHTLINEFIT fits a straight line to the given data series	ao
tdfit	TDFIT fit a set of smodels to a set of input and output signals..	ao
tfe	TFE estimates transfer function between time-series objects.	ao
timedomainfit	TIMEDOMAINFIT uses lscov to fit a set of time-series AOs to a target time-series AO.	ao
upsample	UPSAMPLE overloads upsample function for AOs.	ao
whiten1D	WHITEN1D whitens the input time-series.	ao
whiten2D	WHITEN2D whiten the noise for two cross correlated time series.	ao
xcorr	XCORR makes cross-correlation estimates of the time-series	ao
xfit	XFIT fit a function of x to data.	ao

zDomainFit	zDomainFit performs a fitting loop to identify model order and	ao
zeropad	ZEROPAD zero pads the input data series.	ao

[▲ Back to Top of Section](#)

Trigonometry

Methods	Description	Defined in class
acos	ACOS overloads the acos method for Analysis objects.	ao
asin	ASIN overloads the asin method for Analysis objects.	ao
atan	ATAN overloads the atan method for Analysis objects.	ao
atan2	ATAN2 overloads the atan2 operator for Analysis objects. Four quadrant inverse tangent.	ao
cos	COS overloads the cos method for Analysis objects.	ao
sin	SIN overloads the sin method for Analysis objects.	ao
tan	TAN overloads the tan method for Analysis objects.	ao

[▲ Back to Top of Section](#)

◀ Class descriptions

ssm Class ▶

ssm Class

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

[▲ Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
amats	A matrix representing a difference/differential term in the state equation, block stored in a cell array	ssm
mmats	M matrix representing an inertial coefficient matrix in the state equation, block stored in a cell array	ssm
bmats	B matrix representing an input coefficient matrix in the state equation, block stored in a cell array	ssm
cmats	C matrix representing the state projection in the observation equation, block stored in a cell array	ssm
dmats	D matrix representing the direct feed through term in the observation equation, block stored in a cell array	ssm
timestep	Timestep of the difference equation. Zero means the representation is time continuous and A defines a differential equation.	ssm

<u>inputs</u>	Ssmblock for input blocks	ssm
<u>states</u>	Ssmblock describing state blocks	ssm
<u>outputs</u>	Ssmblock describing the output blocks	ssm
<u>params</u>	Nested plist describing the parameters	ssm
<u>version</u>	CVS version string of the constructor	ssm
<u>Ninputs</u>	Number of input column-blocks, it is a double	ssm
<u>inputsizes</u>	Width corresponding to each input column-block in the B/D matrices. It is a double vector	ssm
<u>Noutputs</u>	Number of output row-blocks, it is a double	ssm
<u>outputsizes</u>	Width corresponding to each output row-block in the C/D matrix. It is a double vector	ssm
<u>Nstates</u>	Number of state-blocks, it is a double	ssm
<u>statesizes</u>	Size corresponding to each state-block in the A/B/C matrices. It is a double vector	ssm
<u>Nparams</u>	Total number of parameters	ssm
<u>isnumerical</u>	This binary tells whether the system has numerical content only, or symbolic as well	ssm
<u>Nss</u>	Number of state-blocks, it is a double	ssm
<u>sssizes</u>	Size corresponding to each input state-block in the A/B/C matrices. It is a double vector	ssm
<u>inputnames</u>	Names corresponding to each input column-block in the B/D matrices. Cell array of strings	ssm
<u>inputvarnames</u>	Names corresponding to each input column in each column-block of the B/D matrices. Cell array of cell array of Strings	ssm
<u>ssnames</u>	Names corresponding to each input state-block in the A/B/C matrices. Cell array of strings	ssm
<u>ssvarnames</u>	Names corresponding to each variable in each input state-block in the A/B/C matrices. Cell array of cell array of Strings	ssm
<u>statenames</u>	Cell array with states blocks names	ssm
<u>statevarnames</u>	Embedded cell array with states ports names	ssm

outputnames	Names corresponding to each output row-block in the C/D matrices. Cell array of strings	ssm
outputvarnames	Names corresponding to each output variable in each row-block in the C/D matrices. Cell array of cell array of Strings	ssm
paramnames	Names of each parameter, stored as a string in a cell array	ssm
paramvalues	Nominal value of each parameter, stored as a double vector	ssm
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Operator	Operator methods
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined

		in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
ssm	SSM statespace model class constructor.	ssm

[▲ Back to Top of Section](#)

Converter

Methods	Description	Defined in class
ssm2miir	SSM2MIIR converts a statespace model object to a miir object	ssm
ssm2pzmodel	SSM2PZMODEL converts a time-continuous statespace model object to a pzmodel	ssm
ssm2rational	SSM2RATIONAL converts a statespace model object to a rational frac. object	ssm
ssm2ss	SSM2SS converts a statespace model object to a MATLAB statespace object.	ssm

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh

setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
addParameters	ADDPARAMETERS Adds the parameters to the model.	ssm
duplicateInput	DUPLICATEINPUT copies the specified input blocks.	ssm
setBlockDescriptions	SETBLOCKDESCRIPTIONS Sets descriptions of the specified SSM blocks.	ssm
setBlockNames	SETBLOCKNAMES Sets names of the specified SSM blocks.	ssm
setParameters	SETPARAMETERS Sets the values of the given parameters.	ssm
setParams	SETPARAMS Sets the parameters of the model to the given plist.	ssm
setPortDescriptions	SETPORTDESCRIPTIONS Sets descriptions of the specified SSM ports.	ssm
setPortNames	SETPORTNAMES Sets names of the specified SSM ports.	ssm
setPortUnits	SETPORTUNITS Sets units of the specified SSM ports.	ssm

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to	ltpda_uoh

recreate the input object(s).

[type](#) TYPE converts the input objects to MATLAB functions. ltpda_uoh

[▲ Back to Top of Section](#)

Operator

Methods	Description	Defined in class
assemble	assembles embedded subsystems, with exogenous inputs	ssm
kalman	KALMAN applies Kalman filtering to a discrete ssm with given i/o	ssm
keepParameters	KEEPPARAMETERS enables to substitute symbolic parameters	ssm
modifTimeStep	MODFTIMESTEP modifies the timestep of a ssm object	ssm
sMinReal	SMINREAL gives a minimal realization of a ssm object by deleting unreachable states	ssm
simplify	SIMPLIFY enables to do model simplification	ssm
simulate	SIMULATE simulates a discrete ssm with given inputs	ssm
subsParameters	SUBSPARAMETERS enables to substitute symbolic parameters	ssm

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a ssm object into a string.	ssm
display	DISPLAY display ssm object.	ssm

dotview	DOTVIEW view an ssm object via the DOT interpreter.	ssm
double	Convert a statespace model object to double arrays for given i/o	ssm
findParameters	findParameters returns parameter names matching the given pattern.	ssm
getParameters	getParameters returns parameter values for the given names.	ssm
getParams	GETPARAMS returns the parameter list for this SSM model.	ssm
isStable	ISSTABLE tells if ssm is numerically stable	ssm
resp	RESP gives the timewise impulse response of a statespace model.	ssm
ssm2dot	SSM2DOT converts a statespace model object a DOT file.	ssm

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
bode	BODE makes a bode plot from the given inputs to outputs.	ssm

[▲ Back to Top of Section](#)

◀ ao Class

smode Class ▶

smodel Class

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

[▲ Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
expr	Expression of the model	smodel
params	Parameters which are used in the model	smodel
values	Default values for the parameters	smodel
xvar	X-variable	smodel
xvals	Values for the x-variable	smodel
xunits	Units of the x-axis	smodel
yunits	Units of the y-axis	smodel
version	CVS version string of the constructor	smodel
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh

name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Arithmetic Operator	
Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Operator	Operator methods
Output	Output methods
Relational Operator	Relational operator methods

[▲ Back to Top](#)

Arithmetic Operator

Methods	Description	Defined in class
minus	MINUS implements subtraction operator for ltpda model objects.	smodel
mrdivide	MRDIVIDE implements mrdivide operator for ltpda model objects.	smodel
mtimes	MTIMES implements mtimes operator for ltpda model objects.	smodel
plus	PLUS implements addition operator for ltpda model objects.	smodel
rdivide	RDIVIDE implements division operator for ltpda model objects.	smodel
times	TIMES implements multiplication operator for ltpda model objects.	smodel

[▲ Back to Top of Section](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
smodel	SMODEL constructor for smodel class.	smodel

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
conj	CONJ gives the complex conjugate of the input smodels	smodel
double	DOUBLE Returns the numeric result of the model.	smodel
eval	EVAL evaluates the symbolic model and returns an AO containing the numeric data.	smodel
fitfunc	FITFUNC Returns a function handle which sets the 'values' and 'xvals' to a ltpda model.	smodel
op	OP Add a operation around the model expression	smodel

setParams	SETPARAMS Set the property 'params' AND 'values'	smodel
setValues	SETVALUES Set the property 'values'	smodel
setXunits	SETXUNITS Set the property 'xunits'.	smodel
setXvals	SETXVALS Set the property 'xvals'	smodel
setXvar	SETXVAR Set the property 'xvar'	smodel
setYunits	SETYUNITS Set the property 'yunits'.	smodel
subs	SUBS substitutes symbolic parameters with the given values.	smodel

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

Operator

Methods	Description	Defined in class
convol_integral	CONVOL_INTEGRAL implements the convolution integral for smodel objects.	smodel
det	DET evaluates the determinant of smodel objects.	smodel
diff	DIFF implements differentiation operator for smodel objects.	smodel

fourier	FOURIER implements continuous f-domain Fourier transform for smodel objects.	smodel
ifourier	IFOURIER implements continuous f-domain inverse Fourier transform for smodel objects.	
ilaplace	ILAPLACE implements continuous s-domain inverse Laplace transform for smodel objects.	smodel
inv	INV evaluates the inverse of smodel objects.	smodel
iztrans	IZTRANS implements continuous z-domain inverse Z-transform for smodel objects.	
laplace	LAPLACE implements continuous s-domain Laplace transform for smodel objects.	smodel
simplify	SIMPLIFY implements simplify operator for smodel objects.	smodel
sop	SOP apply a symbolic operation to the expression.	smodel
sum	SUM adds all the elements of smodel arrays.	smodel
ztrans	ZTRANS implements continuous z-domain Z-transform for smodel objects.	smodel

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a smodel object into a string.	smodel
display	DISPLAY overloads display functionality for smodel objects.	smodel

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

◀ ssm Class	filterbank Class ▶

©LTP Team

filterbank Class

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

 [Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
filters	Filters of the bank	filterbank
type	Type of the bank	filterbank
version	CVS version string of the constructor	filterbank
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
filterbank	FILTERBANK constructor for filterbank class.	filterbank
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh

[setProcinfo](#) SETPROCINFO sets the 'procinfo' property of the ltpda_uoh ao.

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
char	CHAR convert a filterbank object into a string.	filterbank
display	DISPLAY overloads display functionality for filterbank objects.	filterbank
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class

eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

 [Back to Top of Section](#)

 smodel Class	mfir Class 

©LTP Team

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

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
gd		mfir
version	CVS version string of the constructor	mfir
ntaps	Number of coefficients in the filter	mfir
fs	Frequency of the filter	ltpda_filter
infile	Filename which builds the filter	ltpda_filter
a	Set of numerator coefficients	ltpda_filter
histout	Output history values to filter	ltpda_filter
iunits	Input unit of a transfer function	ltpda_tf
ounits	Output unit of a transfer function	ltpda_tf
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh

plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
mfir	MFIR FIR filter object class constructor.	mfir

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
setHistout	SETHISTOUT Set the property 'histout' to a filter object	ltpda_filter
get	GET get a property of a object.	ltpda_obj
setlunits	SETIUNITS sets the 'iunits' property of the transfer functions.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the	ltpda_tf

transfer functions.

created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
redesign	REDESIGN redesign the input filter to work for the mfir given sample rate.	

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a mfir object into a string.	mfir
display	DISPLAY overloads display functionality for mfir objects.	mfir

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
impresp	IMPRESP Make an impulse response of the filter.	ltpda_filter
resp	RESP Make a frequency response of the filter.	ltpda_filter

[▲ Back to Top of Section](#)



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

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
b	Set of numerator coefficients	miir
histin	Input history values to filter	miir
version	CVS version string of the constructor	miir
ntaps	Number of coefficients in the filter	miir
fs	Frequency of the filter	ltpda_filter
infile	Filename which builds the filter	ltpda_filter
a	Set of numerator coefficients	ltpda_filter
histout	Output history values to filter	ltpda_filter
iunits	Input unit of a transfer function	ltpda_tf
ounits	Output unit of a transfer function	ltpda_tf
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained	ltpda_uoh

in the main result of any method (plist-object).

plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
miir	MIIR IIR filter object class constructor.	miir

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
setHistout	SETHISTOUT Set the property 'histout' to a filter object	ltpda_filter
get	GET get a property of a object.	ltpda_obj
setlunits	SETIUNITS sets the 'iunits' property of the transfer ltpda_tf functions.	

setOunits	SETOUNITS sets the 'ounits' property of the transfer functions.	ltpda_tf
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
redesign	REDESIGN redesign the input filter to work for the miir given sample rate.	

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh

type	TYPE converts the input objects to MATLAB functions. ltpda_uoh	
setHistin	SETHISTIN Set the property 'histin'	miir

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a miir object into a string.	miir
display	DISPLAY overloads display functionality for miir objects.	miir

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
impresp	IMPRESP Make an impulse response of the filter.	ltpda_filter
resp	RESP Make a frequency response of the filter.	ltpda_filter

[▲ Back to Top of Section](#)

©LTP Team

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

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
poles	Vector of poles (pz-objects)	pzmodel
zeros	Vector of zeros (pz-objects)	pzmodel
gain	Gain of the model	pzmodel
delay	Delay of the pole/zero Model	pzmodel
version	CVS version string of the constructor	pzmodel
iunits	Input unit of a transfer function	ltpda_tf
ounits	Output unit of a transfer function	ltpda_tf
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo

description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Arithmetic Operator	
Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Operator	Operator methods
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲ Back to Top](#)

Arithmetic Operator

Methods	Description	Defined in class
mrdivide	MRDIVIDE overloads the division operator for pzmodels.	pzmodel
mtimes	MTIMES overloads the multiplication operator for pzmodels.	pzmodel
rdivide	RDIVIDE overloads the division operator for pzmodels.	pzmodel
times	TIMES overloads the multiplication operator for pzmodels.	pzmodel

[▲ Back to Top of Section](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
pzmodel	PZMODEL constructor for pzmodel class.	pzmodel

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
setlunits	SETIUNITS sets the 'iunits' property of the transfer functions.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the transfer functions.	ltpda_tf
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
setDelay	SETDELAY sets the 'delay' property of a pole/zero model.	pzmodel

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in	ltpda_uo

binary form to an LTPDA Repository.

submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh
getlowerFreq	GETLOWERFREQ gets the frequency of the lowest pole or zero in the model.	pzmodel
getupperFreq	GETUPPERFREQ gets the frequency of the highest pole or zero in the model.	pzmodel
respCore	RESPCORE returns the complex response of one pzmodel object.	pzmodel

[▲ Back to Top of Section](#)

Operator

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

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a pzmodel object into a string.	pzmodel

display	DISPLAY overloads display functionality for pzmodel objects.	pzmodel

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
resp	RESP returns the complex response of a transfer function as an Analysis Object.	ltpda_tf
fngen	FNGEN creates an arbitrarily long time-series based on the input pzmodel.	
simplify	SIMPLIFY simplifies pzmodels by cancelling like poles with like zeros.	pzmodel

[▲ Back to Top of Section](#)

parfrac Class

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

[▲ Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name'); % This command applies to obj
```

Properties	Description	Defined in class
res	Residuals of the partial fraction representation	parfrac
poles	Poles (real or complex numbers) of the partial fraction representation	parfrac
pmul	Represents the pole multiplicity	parfrac
dir	Direct terms of the partial fraction representation	parfrac
version	CVS version string of the constructor	parfrac
iunits	Input unit of a transfer function	ltpda_tf
ounits	Output unit of a transfer function	ltpda_tf
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh

name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
parfrac	PARFRAC partial fraction representation of a transfer function.	parfrac

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
setlunits	SETIUNITS sets the 'iunits' property of the transfer functions.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the transfer functions.	ltpda_tf
created	CREATED Returns a time object of the last modification.	ltpda_uoh

creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh
getlowerFreq	GETLOWERFREQ gets the frequency of the lowest pole in the model.	parfrac
getupperFreq	GETUPPERFREQ gets the frequency of the highest pole in the model.	parfrac
respCore	RESPCORE returns the complex response of one	parfrac

parfrac object.

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a parfrac object into a string.	parfrac
display	DISPLAY overloads display functionality for parfrac objects.	parfrac

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
resp	RESP returns the complex response of a transfer function as an Analysis Object.	ltpda_tf

[▲ Back to Top of Section](#)

rational Class

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

[▲ Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
num	Numerator coefficients of the rational representation	rational
den	Denominator coefficients of the rational representation	rational
version	CVS version string of the constructor	rational
iunits	Input unit of a transfer function	ltpda_tf
ounits	Output unit of a transfer function	ltpda_tf
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo

mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
rational	RATIONAL rational representation of a transfer function.	rational

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
setlunits	SETIUNITS sets the 'iunits' property of the transfer functions.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the transfer functions.	ltpda_tf
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh

index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh
getlowerFreq	GETLOWERFREQ gets the frequency of the lowest pole in the model.	rational
getupperFreq	GETUPPERFREQ gets the frequency of the highest pole in the model.	rational
respCore	RESPCORE returns the complex response of one rational object.	rational

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a rational object into a string.	rational
display	DISPLAY overloads display functionality for rational objects.	rational

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

Signal Processing

Methods	Description	Defined in class
resp	RESP returns the complex response of a transfer function as an Analysis Object.	ltpda_tf

[▲ Back to Top of Section](#)

◀ parfrac 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

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
startT	TIME object of the start time	timespan
endT	TIME object of the end time	timespan
timeformat	Time format of the start/end time objects	timespan
timezone	Timezone of the start/end time objects	timespan
interval	Interval string of the start/end time	timespan
version	CVS version string of the constructor	timespan
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo

mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
timespan	TIMESPAN timespan object class constructor.	timespan

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh

setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
setEndT	SETENDT Set the property 'endT'.	timespan
setStartT	SETSTARTT Set the property 'startT'.	timespan
setTimeformat	SETTIMEFORMAT Set the property 'timeformat'.	timespan
setTimezone	SETTIMEZONE Set the property 'timezone'.	timespan

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh

[_____](#)

viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a timespan object into a string.	timespan
display	DISPLAY overloads display functionality for timespan objects.	timespan

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

◀ rational 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

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
params	List of param-objects (key/value pairs)	plist
created	The creation/manipulation time of the object	plist
creator	The creator/manipulator of the object with all used toolboxes	plist
version	CVS version string of the constructor	plist
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
plist	PLIST Plist class object constructor.	plist

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uo
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uo
append	APPEND append a param-object, plist-object or a key/value pair to the parameter list.	plist
combine	COMBINE multiple parameter lists (plist objects) into a single plist.	plist
find	FIND overloads find routine for a parameter list.	plist
getIndexForKey	GETINDEXFORKEY returns the index of a parameter with the given key.	plist
getKeys	GETKEYS Return all the keys of the parameter list.	plist
getOptionsForParam	GETOPTIONSFORPARAM Returns the options for the specified parameter key.	plist
getSelectionForParam	GETSELECTIONFORPARAM Returns the selection mode for the specified parameter key.	plist

isparam	ISPARAM look for a given key in the parameter lists.	plist
nparams	NPARAMS returns the number of param objects in the list.	plist
pset	PSET set or add a key/value pair or a param-object into the parameter list.	plist
remove	REMOVE remove a parameter from the parameter list.	plist
removeKeys	REMOVEKEYS removes keys from a PLIST.	plist
setDescriptionForParam	SETDESCRIPTIONFORPARAM Sets the property 'desc' of the param object in dependencies of the 'key'	plist
setName	SETNAME Set the property 'name'.	plist
setOptionsForParam	SETOPTIONSFORPARAM Sets the options of the param object in dependencies of the 'key'	plist
setSelectionForParam	SETSELECTIONFORPARAM Sets the selection mode of the param object in dependencies of the 'key'	plist
string	STRING converts a plist object to a command string which will recreate the plist object.	plist

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	Itpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	Itpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	Itpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	Itpda_uo
getDescriptionForParam	GETDESCRIPTIONFORPARAM Returns the	plist

	description for the specified parameter key.	
getSetRandState	GETSETRANDSTATE gets or sets the random state of the MATLAB functions 'rand' and 'randn'	plist
parse	PARSE a plist for strings which can be converted into numbers	plist
plist2cmds	PLIST2CMDS convert a plist to a set of commands.	plist

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
char	CHAR convert a parameter list into a string.	plist
display	DISPLAY display plist object.	plist

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj
eq	EQ overloads the == operator for ltpda plist objects.	plist

[▲ Back to Top of Section](#)

◀ timespan Class

matrix Class ▶

matrix Class

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

[▲ Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
objs	Objects in matrix	matrix
version	CVS version string of the constructor	matrix
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Arithmetic Operator	
Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Operator	Operator methods
Output	Output methods
Relational Operator	Relational operator methods

[▲ Back to Top](#)

Arithmetic Operator

Methods	Description	Defined in class
conj	conj implements conj operator for matrix objects.	matrix
ctranspose	CTRANSPOSE implements conjugate transpose operator for matrix objects.	matrix
filter	FILTER implements N-dim filter operator for matrix objects.	matrix
minus	MINUS implements subtraction operator for ltpda model objects.	matrix
mtimes	MTIMES implements mtimes operator for matrix objects.	matrix
plus	PLUS implements addition operator for matrix objects.	matrix
rdivide	RDIVIDE implements division operator for matrix objects.	matrix
times	TIMES implements multiplication operator for matrix objects.	matrix
transpose	TRANSPOSE implements transpose operator for matrix objects.	matrix

[▲ Back to Top of Section](#)

Constructor

Methods	Description	Defined in class
 		

rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh
matrix	MATRIX constructor for matrix class.	matrix

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last modification.	ltpda_uoh
creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh
getObjectAtIndex	GETOBJECTATINDEX index into the inner objects of one matrix object.	matrix
ncols	NCOLS Returns the number of columns of the inner object array.	matrix
nrows	NROWS Returns the number of rows of the inner object array.	matrix
osize	OSIZE Returns the size of the inner object array.	matrix
setObjs	SETOBJS sets the 'objs' property of a matrix object.	matrix

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

Operator

Methods	Description	Defined in class
det	DET evaluates the determinant for matrix object.	matrix
inv	INV evaluates the inverse for matrix object.	matrix

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	CHAR convert a matrix object into a string.	matrix
display	DISPLAY overloads display functionality for matrix objects.	matrix

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

[▲ Back to Top of Section](#)

◀

plist Class

collection Class

▶

collection Class

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

[▲ Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:  
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:  
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)  
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defined in class
objs	Objects in collection	collection
version	CVS version string of the constructor	collection
hist	History object associated with this object	ltpda_uoh
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ltpda_uoh
plotinfo	Plist-object which contains the plot information of the object	ltpda_uoh
name	Name of the object	ltpda_uo
description	Description of the object	ltpda_uo
mdlfile	Full text representation of the mdl-file that created this AO	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defined in class
collection	COLLECTION constructor for collection class.	collection
rebuild	REBUILD rebuilds the input objects using the history.	ltpda_uoh

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defined in class
addObjects	ADDOBJECTS adds the given objects to the collection.	collection
getObjectAtIndex	GETOBJECTATINDEX index into the inner objects of one collection object.	collection
getObjectsOfClass	GETOBJECTSOFCCLASS returns all objects of the specified class in a collection-object.	collection
nobjs	NOBJS Returns the number of objects in the inner object array.	collection
removeObjectAtIndex	REMOVEOBJECTATINDEX removes the object at the specified position from the collection.	collection
setObjectAtIndex	SETOBJECTATINDEX sets an input object to the collection.	collection
setObjs	SETOBJS sets the 'objs' property of a collection object.	collection
get	GET get a property of a object.	ltpda_obj
created	CREATED Returns a time object of the last	ltpda_uoh

modification.

creator	CREATOR Extract the creator(s) from the history.	ltpda_uoh
csvexport	CSVEXPORT Exports the data of an object to a csv file.	ltpda_uoh
index	INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
setDescription	SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.	ltpda_uoh
setMdlfile	SETMDLFILE sets the 'mdlfile' property of the ao.	ltpda_uoh
setName	SETNAME Set the property 'name'.	ltpda_uoh
setPlotinfo	SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.	ltpda_uoh
setProcinfo	SETPROCINFO sets the 'procinfo' property of the ao.	ltpda_uoh

[▲ Back to Top of Section](#)

Internal

Methods	Description	Defined in class
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
bsubmit	BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.	ltpda_uo
submit	SUBMIT submits the given collection of objects to an LTPDA Repository.	ltpda_uo
update	UPDATE updates the given object in an LTPDA Repository.	ltpda_uo
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	ltpda_uoh

[▲ Back to Top of Section](#)

Output

Methods	Description	Defined in class
char	CHAR convert a collection object into a string.	collection
display	DISPLAY overloads display functionality for collection objects.	collection
save	SAVE overloads save operator for ltpda objects.	ltpda_uo
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
viewHistory	VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh

 [Back to Top of Section](#)

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda objects.	ltpda_obj
ne	NE overloads the ~= operator for ltpda objects.	ltpda_obj

 [Back to Top of Section](#)

 matrix Class

Constructor Examples 

Constructor Examples

Constructor examples

[General Constructor examples](#)

[Constructor examples of the AO class](#)

[Constructor examples of the MFIR class](#)

[Constructor examples of the MIIR class](#)

[Constructor examples of the PZMODEL class](#)

[Constructor examples of the PARFRAC class](#)

[Constructor examples of the RATIONAL class](#)

[Constructor examples of the TIMESPAN class](#)

[Constructor examples of the PLIST class](#)

[Constructor examples of the SPECWIN class](#)

◀ collection Class

General constructor examples ▶

©LTP Team

General constructor examples

[Copy a LTPDA object](#)

[Construct an empty LTPDA object](#)

[Construct an empty LTPDA object with the size zero](#)

[Construct a LTPDA object by loading the object from a file](#)

[Construct an LTPDA object from a parameter list object \(PLIST\)](#)

All here described constructors works for all LTPDAobjects.

Copy a LTPDA object

The following example creates a copy of a LTPDA object (blue command) by an example of the PARFRAC object. The copy constructor should work for all classes.

```
>> pf1 = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3)
---- parfrac 1 ----
model:      None
res:        [1;2+i*1;2-i*1]
poles:      [6;1+i*3;1-i*3]
dir:        3
pmul:       [1;1;1]
iunits:     []
ounits:     []
description:
UUID:       7c57ab84-3f49-486b-be49-6ed9926fbd66
-----
>> pf2 = parfrac(pf1)
---- parfrac 1 ----
model:      None
res:        [1;2+i*1;2-i*1]
poles:      [6;1+i*3;1-i*3]
dir:        3
pmul:       [1;1;1]
iunits:     []
ounits:     []
description:
UUID:       c892f70e-d113-407c-acc-67f988b31e7e
-----
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> pf1 = parfrac()
---- parfrac 1 ----
model:      none
res:        []
poles:      []
dir:        0
pmul:       []
iunits:     []
ounits:     []
description:
UUID:       bd7c9c71-e633-402b-964b-0a270f80764a
-----
>> pf2 = pf1;
>> pf2.setName('my new name')
---- parfrac 1 ----
model:      my new name
res:        []
poles:      []
```

```
dir: 0
pmul: []
iunits: []
ounits: []
description:
UUID: 47a5e458-84fd-4d6b-beb2-11ff574ae661
-----
```

If we display pf1 again then we see that the property 'name' was changed although we only have changed pf2.

```
>> pf1
---- parfrac 1 ----
model: my new name
res: []
poles: []
dir: 0
pmul: []
iunits: []
ounits: []
description:
UUID: 47a5e458-84fd-4d6b-beb2-11ff574ae661
-----
```

Construct an empty LTPDA object

The following example creates an empty LTPDA object by an example of a rational object.

```
>> rat = rational()
---- rational 1 ----
model: none
num: []
den: []
iunits: []
ounits: []
description:
UUID: eb240e62-ff2f-4c31-a683-0f395b9a5242
-----
```

Construct an empty LTPDA object with the size zero.

Sometimes it is necessary to create LTPDA objects which have at least one dimension of zero. This mean that the MATLAB method "isempty" is true for these objects. This constructor is shown by an example of a matrix object.

```
>> m = matrix.initObjectWithSize(1,0)
----- matrix -----
empty-object [1,0]
-----
```

It is also possible with this constructor to create a matrix of objects with a different input as zero.

```
>> m = matrix.initObjectWithSize(2,1);
---- matrix 1 ----
name: none
size: 0x0
description:
UUID: ed13da80-92a7-4d1f-8d78-a88f7d77cec3
-----
---- matrix 2 ----
name: none
size: 0x0
description:
UUID: ed13da80-92a7-4d1f-8d78-a88f7d77cec3
```

Construct a LTPDA object by loading the object from a file

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

```
pf = ao('parfrac_object.mat')
pf = ao('parfrac_object.xml')
```

Construct an LTPDA object from a parameter list object (PLIST)

Each constructor have different sets of PLISTS which create the object in a different way.

AO class	Parameter Sets
SSM class	Parameter Sets
SMODEL class	Parameter Sets
TIMESPAN class	Parameter Sets
MATRIX class	Parameter Sets
COLLECTION class	Parameter Sets
PZMODEL class	Parameter Sets
PARFRAC class	Parameter Sets
RATIONAL class	Parameter Sets
FILTERBANK class	Parameter Sets
MIIR class	Parameter Sets
MFIR class	Parameter Sets
PLIST class	Parameter Sets

Constructor examples of the AO class

[Copy an AO](#)

[Construct an AO by loading the AO from a file](#)

[Construct an AO from a data file](#)

[Construct an AO from spectral window](#)

[Construct an AO from a parameter list object \(PLIST\)](#)

Copy an AO

The following example creates a copy of an analysis object (blue command).

```
>> a1 = ao([1:12]);
>> a2 = ao(1)
----- ao: a -----

name: none
creator: created by hewitson@bobmac.aei.uni-hannover.de[130.75.117.65] on MACI/7.6
description:
data: None
hist: ao / ao / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
mfilename:
mdlfilename:
-----
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> a1 = ao()
----- ao 01: a1 -----
name: none
description:
data: None
hist: ao / ao / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
mfilename:
mdlfilename:
-----
>> a2 = a1;
>> a2.setName('my new name')
----- ao 01: my new name -----
name: my new name
description:
data: None
hist: ltpda_uoh / setName / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
mfilename:
mdlfilename:
-----
```

If we display a1 again then we see that the property 'name' was changed although we only have changed a2.

```
>> a1
----- ao 01: my new name -----
name: my new name
description:
data: None
hist: ltpda_uoh / setName / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo 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')
```

or in a PLIST

```
pl = plist('filename', 'a1.xml')
a = ao(pl)
```

Construct an AO from a data 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 determines how the analysis object is created. The valid key/value pairs of the parameter list are:

Key	Description
'type'	'tsdata','fsdata','xydata' [default: 'tsdata']
'fs'	If this value is set, the x-axes is computed by the fs value. [default: empty]
'columns'	[1 2 1 4] Each pair represents the x- and y-axes (each column pair creates an analysis object). If the value 'fs' is set, then each column is converted to the y vector of a time-series AO. [default: [1 2]]
'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 represents the x- and y-axes.
% 'fs' is not used !!!

pl = plist('filename', 'data.dat', ...
'description', 'my ao description', ...
'type', 'xydata', ...
'xunits', 's', ...
'yunits', {'Volt', 'Hz'}, ...
'columns', [1 2 1 3], ...
```

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

Another example where the time vector is specified by the sample rate (f_s) and each column of data is converted in to a single AO.

```
% 'fs' is used. As such, each column in col creates its own AO with the specified sample
rate.
pl = plist('filename', 'data.dat',...
'type', 'tsdata', ...
'fs', 100, ...
't0', {'14:00:00', '14:00:20', '14:00:30'}, ...
'columns', [1 2 3]);
out = ao('data.dat', pl);
out = ao(pl);
```

Construct an AO from a spectral window

The following example creates a cdata type AO containing the window values.

```
win = specwin('Kaiser', 100, 10);
>> a = ao(win)
----- ao 01: Kaiser -----
name: Kaiser
description:
data: 0.7145 0.7249 0.7351 0.7452 0.7551 0.7649 0.7746 0.7840 0.7934 0.8025 ...
----- cdata 01 -----
y: [1x100], double
yunits: []
-----
hist: ao / ao / $Id: fromSpecWin.m,v 1.11 2008/12/05 10:47:14 hewitson Exp -->
mfilename:
mdlfilename:
-----
```

Construct an AO from a parameter list (plist)

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

[Use the key word 'fcn'](#)

[Use the key word 'vals'](#)

[Use the key word 'xvals' AND 'yvals'](#)

[Use the key word 'tsfcn'](#)

[Use the key word 'fsfcn'](#)

[Use the key word 'xyfcn'](#)

[Use the key word 'win'](#)

[Use the key word 'waveform'](#)

[Use the key word 'hostname'](#)

[Use the key word 'polyval'](#)

[Use the key word 'pzmodel'](#)

Use the key word 'fcn'

The following example creates an AO from the description of any valid MATLAB function. The data object is of type cdata.

```
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 'vals'

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)
```

Use the key word 'xvals' and 'yvals'

Construct an AO from a set of values. The data type depends on the parameters. You have to specify the parameters:

Key	Description
'xvals'	a set of x values.
'yvals'	a set of y values.

You can also specify optional parameters

Key	Description
'dtype'	type of the data object [default: tsdata]
'fs'	sampling frequency. Only used if 'dtype' is a time-series object [default: []]

```
x = 1:1000;
y = randn(1000,1);
pl1 = plist('xvals', x, 'yvals', y, 'dtype', 'tsdata');
pl2 = plist('xvals', x, 'yvals', y, 'dtype', 'fsdata');
pl3 = plist('xvals', x, 'yvals', y, 'dtype', 'xydata');
a1 = ao(pl1) % Create an AO with time-series data
a2 = ao(pl2) % Create an AO with frequency-series data
a3 = ao(pl3) % Create an AO with x-y data
```

Use the key word 'tsfcn'

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

Key	Description
'tsfcn'	a function of time

You can also specify optional parameters

Key	Description
'fs'	sampling frequency [default: 1 Hz]
'nsecs'	length in seconds [default: 10 s]
't0'	Start time which is associated with the time-series [default: '1970-01-01 00:00:00.000']

Example:

```
pl = plist('fs', 10, 'nsecs', 10, ...
          'tsfcn', 'sin(2*pi*1.4*t) + 0.1*randn(size(t))', ...
          't0', '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 of type `fsdata` (frequency-series).

Key	Description
'fsfcn'	a function of frequency

You can also specify optional parameters:

Key	Description
'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:

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

```
pl1 = plist('fsfcn', '1./f.^2', 'scale', 'lin', 'nf', 100);
pl2 = plist('fsfcn', '1./f.^2', 'f', logspace(0,3, 1000));
a1 = ao(pl1)
a2 = ao(pl2)
```

Use the key word 'xyfcn'

Construct an AO from a function $f(x)$ string. The data object is of type `xydata`. You have to specify the parameters:

Key	Description
'xyfcn'	specify a function of 'x'

'x' the x values

```
pl = plist('X', [1:50,52:2:100,110:10:1000], 'xyfcn', 'log(x)');
a1 = ao(pl)
----- ao 01: a1 -----
name: none
description:
  data: (1,0) (2,0.6931) (3,1.0986) (4,1.3862) (5,1.6094) ...
        ----- xydata 01 -----
              x: [1 165], double
              y: [1 165], double
            xunits: []
            yunits: []
        -----
hist: ao / ao / $Id: fromXYFcn.m,v 1.3 2009/02/10 20:02:51 ingo Exp -->
mfilename:
mdlfilename:
-----
```

Use the key word 'win'

Construct an AO from a spectral window object.
[List of available window functions](#)

```
pl1 = plist('win', specwin('Hanning', 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

Key	Description
'sine wave'	'A' – Amplitude of the wave 'f' – Frequency of the wave 'phi' – Phase of the wave
'noise'	'type' – can be 'Normal' or 'Uniform','sigma' – specify the standard deviation
'chirp'	'f0', 'f1', 't1' (help chirp)
'Gaussian pulse'	'f0', 'bw' (help gausspuls)
'Square wave'	'f', 'duty' (help square)
'Sawtooth'	'f', 'width' (help sawtooth)

You can also specify additional parameters:

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

't0' time-stamp of the first data sample [default time(0)]

[default waveform: 'sine wave', A: 1, f: 1.23, phi: 0, fs: 10, nsecs: 10, t0: time(0)].

```
% Construct a sine wave
pl = plist('nsecs', 10, 'fs', 1000);
pl_w = pl.append('waveform', 'sine wave', 'phi', 30, 'f', 1.23);
out_sin = ao(pl_w)
% Construct random noise
pl_w = pl.append('waveform', 'noise', 'type', 'Normal');
out_noise1 = ao(pl_w)
% Construct uniform random noise
pl_w = append(pl, 'waveform', 'noise', 'type', 'Uniform');
out_noise2 = ao(pl_w)
% Construct a chirp waveform
pl_w = append(pl, 'waveform', 'chirp', 'f0', 1, 'f1', 50, 't1', 100);
out_chirp = ao(pl_w)
% Construct a Gaussian pulse waveform
pl_w = append(pl, 'waveform', 'Gaussian pulse', 'f0', 10, 'bw', 100);
out_gaus = ao(pl_w)
% Construct a Square wave
pl_w = append(pl, 'waveform', 'Square wave', 'f', 1, 'duty', 50);
out_square = ao(pl_w)
% Construct a Sawtooth wave
pl_w = append(pl, 'waveform', 'Sawtooth', 'width', .5, 'f', 1);
out_saw = ao(pl_w)
```

Use the key word 'hostname'

Construct an AO by retrieving it from a LTPDA repository.

The relevant parameters are:

Key	Description
'hostname'	the repository hostname. [default: 'localhost']
'database'	The database name [default: 'ltpda']
'id'	A vector of object IDs. [default: []]
'cid'	Retrieve all AO objects from a particular collection
'binary'	Set to 'yes' to retrieve from stored binary representation (not always available). [default: yes]

```
pl = plist('hostname', '130.75.117.67', 'database', 'ltpda_test', 'id', 1)
a1 = ao(pl)
```

Use the key word 'polyval'

Construct an AO from a set of polynomial coefficients. The relevant parameters are:

Key	Description
'polyval'	A set of polynomial coefficients. [default: []]

Additional parameters:

Key	Description

'nsecs'	Number of seconds [default: 10]
'fs'	Sample rate[default: 10 s]

or

Key	Description
't'	vector of time vertices. The value can also be an AO, in which case the X vector is used. [default: []]

```
pl = plist('polyval', [1 2 3], 'Nsecs', 10, 'fs', 10);
a1 = ao(pl)
```

Use the key word 'pzmodel'

Generates an AO with a timeseries with a prescribed spectrum.
The relevant parameters are:

Key	Description
'pzmodel'	a pole/zero model which builds the time-series AO
'nsecs'	number of seconds to be generated [default: 0]
'fs'	sampling frequency [default: 0]

You can also specify optional parameters:

Key	Description
'xunits'	unit of the x-axis [default: 's']
'yunits'	unit of the y-axis [default: '']

```
p = [pz(1,2) pz(10)]
z = [pz(4)]
pzm = pzmodel(1, p, z)

fs = 10
nsecs = 100
pl = plist('pzmodel', pzm, 'Nsecs', nsecs, 'Fs', fs)
a1 = ao(pl)
```

Constructor examples of the SMODEL class

[Copy a symbolic model object](#)

[Construct an empty symbolic model object](#)

[Construct an empty symbolic model object with the size zero](#)

[Construct a symbolic model object by loading the object from a file](#)

[Construct a simple example](#)

[Construct an LTPDA object from a parameter list object \(PLIST\)](#)

A simple example

The following example shows you how to create a symbolic model.

```
>> s = smodel('a.*x.^2+b.*x+c');
```

You define with the command above the symbolic model. Please take care that each symbolic can represent a vector/matrix of numbers. This mean that you should use the array operators like '.*' or '.*' or ... and not the matrix operators like '*' or '^' or

Now, it is necessary to define the parameters with their default values

```
>> s.setParams({'a', 'b', 'c'}, {1 2 3})
---- symbolic model 1 ----
  name: None
  expr: a.*x.^2+b.*x+c
  params: {'a', 'b', 'c'}
  values: {1, 2, 3}
  xvar:
  xvals: []
  xunits: []
  yunits: []
  description:
    UUID: 7a084750-89c2-47f0-8c6b-65e9079bf6b3
  -----
```

Finally it is necessary to set the 'x' variable and the values.

```
>> s.setXvar('x')
---- symbolic model 1 ----
  name: None
  expr: a.*x.^2+b.*x+c
  params: {'a', 'b', 'c'}
  values: {1, 2, 3}
  xvar: x
  xvals: []
  xunits: []
  yunits: []
  description:
    UUID: ca9276b0-cc97-4001-9135-00c6799ec8b1
  -----

>> s.setXvals(-100:100)
---- symbolic model 1 ----
  name: None
  expr: a.*x.^2+b.*x+c
  params: {'a', 'b', 'c'}
  values: {1, 2, 3}
  xvar: x
  xvals: [-100 -99 -98 -97 -96 -95 -94 -93 -92 -91 -90 -89 -88 -87 -86 ...]
  xunits: []
  yunits: []
  description:
    UUID: f43e2f9b-3fb6-4e3c-92ba-7cba31c6a6c3
  -----
```

After all steps above it is possible to plot the model.

```
>> s.setXunits('s');  
>> s.setYunits('V');  
>> s.setName('Just a test');  
>> iplot(s.eval)
```

◀ Constructor examples of the AO class

Constructor examples of the MFIR class ▶

©LTP Team

Construct a MFIR object from a difference equation

http://www.lisa.aei-hannover.de/ltpda/usermanual/uq/constructor_examples_mfir.html[2/10/09 10:16:49 AM]

```
name: my new name
-----
```

If we display fir1 again then we see that the property 'name' was changed although we only have changed fir2.

```
>> fir1
----- mfir/1 -----
gd: []
version: $Id: mfir.m,v 1.84 2009/02/24 17:02:44 ingo Exp
ntaps: 0
fs: []
infile:
a: []
histout: []
iunits: [1x1 unit]
ounits: [1x1 unit]
hist: mfir.hist [1x1 history]
name: my new name
-----
```

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.

```
fir = mfir('fir.mat')
fir = mfir('fir.xml')
```

or in a PLIST

```
pl = plist('filename', 'fir.xml');
fir = mfir(pl)
```

Construct a MFIR object from an Analysis Object

An FIR filter object can be generated based on the magnitude of the input AO/fsdata object. In the following example an AO/fsdata object is first generated and then passed to the mfir constructor to obtain the equivalent FIR filter.

```
a1 = ao(plist('fsfcn', '1./(50+f)', 'fs', 1000, 'f', linspace(0, 500, 1000)));
fir = mfir(a1);
ipplot(a1, resp(fir));
```

or in a PLIST with the relevant parameters:

Key	Description
'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]

'type' one of the types: 'highpass', 'lowpass', 'bandpass', 'bandreject'
[default 'lowpass']

You can also specify optional parameters:

Key	Description
'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']
'iunits'	the input unit of the filter
'ounits'	the output unit of the filter

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)
```

Construct a MFIR object from an existing filter

The mfir constructor also accepts as an input existing filters stored 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 an LTPDA repository

The relevant parameters for retrieving a FIR filter from a LTPDA repository are:

Key	Description
'hostname'	the repository hostname. [default: 'localhost']
'database'	The database name [default: 'ltpda']
'id'	A vector of object IDs. [default: []]
'cid'	Retrieve all rational objects from a particular collection
'binary'	Set to 'yes' to retrieve from stored binary representation (not always available). [default: yes]

```
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:

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

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

or in a PLIST

The relevant parameters are:

Key	Description
'a'	vector of A coefficients. (see note ** below) [default: empty]
'fs'	sampling frequency of the filter [default: empty]
'name'	name of filter [default: 'None']

```
a = [-0.8 10];
fs = 1;
pl = plist('a', a, 'fs', fs);
fir = mfir(pl)
```

NOTES:

** The convention used here for naming the filter coefficients is the opposite to MATLAB's convention. The recursion formula for this convention is
$$y(n) = a(1)*x(n) + a(2)*x(n-1) + \dots + a(na+1)*x(n-na).$$

◀ Constructor examples of the SMODEL class

Constructor examples of the MIIR class ▶

©LTP Team

Constructor examples of the MIIR class

[Copy a MIIR object](#)

[Construct a MIIR object by loading the object from a file](#)

[Construct a MIIR object from a parfrac object \(PARFRAC\)](#)

[Construct a MIIR object from a pole/zero model \(PZMODEL\)](#)

[Construct a MIIR object from a standard type](#)

[Construct a MIIR object from an existing model](#)

[Construct a MIIR object from a difference equation](#)

Copy an IIR filter object

The following example creates a copy of an IIR filter object (blue command).

```
>> iir1 = miir(plist('type', 'lowpass'));
>> iir2 = miir(iir1)
----- miir/1 -----
  b: [1 -0.509525449494429]
 histin: 0
version: $Id: miir.m,v 1.98 2009/02/20 15:59:48 nicola Exp
  ntaps: 2
   fs: 1
  infile:
    a: [0.245237275252786 0.245237275252786]
 histout: 0
  iunits: [] [1x1 unit]
  ounits: [] [1x1 unit]
    hist: miir.hist [1x1 history]
   name: lowpass
-----
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> iir1 = miir()
----- miir/1 -----
  b: []
 histin: []
version: $Id: miir.m,v 1.98 2009/02/20 15:59:48 nicola Exp
  ntaps: 0
   fs: []
  infile:
    a: []
 histout: []
  iunits: [] [1x1 unit]
  ounits: [] [1x1 unit]
    hist: miir.hist [1x1 history]
   name: none
-----
>> iir2 = iir1;
>> iir2.setName('my new name')
----- miir/1 -----
  b: []
 histin: []
version: $Id: miir.m,v 1.98 2009/02/20 15:59:48 nicola Exp
  ntaps: 0
   fs: []
  infile:
    a: []
 histout: []
```

```
iunits: [] [1x1 unit]
ounits: [] [1x1 unit]
hist: miir.hist [1x1 history]
name: my new name
-----
```

If we display iir1 again then we see that the property 'name' was changed although we only have changed iir2.

```
>> iir1
----- miir/1 -----
  b: []
 histin: []
version: $Id: miir.m,v 1.98 2009/02/20 15:59:48 nicola Exp
  ntabs: 0
   fs: []
  infile:
    a: []
 histout: []
  iunits: [] [1x1 unit]
  ounits: [] [1x1 unit]
   hist: miir.hist [1x1 history]
   name: my new name
-----
```

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')
```

or in a PLIST

```
pl = plist('filename', 'iir.xml');
iir = miir(pl)
```

Construct a MIIR object from a parfrac object

An IIR filter object can be generated based on a parfrac object. The next example shows how you can convert a parfrac object into a iir filter object.

```
pf = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3);
iir = miir(pf)
```

or in a PLIST with the relevant parameters:

Key	Description
'parfrac'	a parfrac object to construct the filters from [default: empty parfrac]
'fs'	sample rate for the filter(s) [default: 8 * upper frequency of the parfrac object]

The following example creates a IIR filter object from an parfrac object.

```
pf = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3);
```

```
pl = plist('parfrac', pf, 'fs', 100);
```

Construct an IIR filter object from a pole/zero model

The following example creates a new IIR filter object from a pole/zero model.

```
>> pzm = pzmodel(1, {1 2 3}, {4 5}, 'my pzmodel')
----- pzmodel 1 -----
      name: my pzmodel
      gain: 1
      delay: 0
      iunits: []
      ounits: []
      pole 001: (f=1 Hz,Q=NaN)
      pole 002: (f=2 Hz,Q=NaN)
      pole 003: (f=3 Hz,Q=NaN)
      zero 001: (f=4 Hz,Q=NaN)
      zero 002: (f=5 Hz,Q=NaN)
-----
>> iir = miir(pzm) % Use the default sample rate fs=8 * frequency of the highest pole or
zero in the model
>> iir = miir(pzm, plist('fs', 100))
----- miir/1 -----
      b: [1 -0.6485 -1.9619 1.3364 0.9644 -0.6854]
      histin: [0 0 0 0 0]
      version: $Id: miir.m,v 1.98 2009/02/20 15:59:48 nicola Exp
      ntaps: 6
      fs: 100
      infile:
      a: [0.0102 0.0152 -0.0097 -0.0186 0.0019 0.0057]
      histout: [0 0 0 0 0]
      iunits: [] [1x1 unit]
      ounits: [] [1x1 unit]
      hist: miir.hist [1x1 history]
      name: my pzmodel
-----
```

or in a PLIST with the relevant parameters:

Key	Description
'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]

```
>> pzm = pzmodel(1, {1 2 3}, {4 5}, 'my pzmodel')
>> pl = plist('pzmodel', pzm, 'fs', 100)
>> iir = miir(pl)
```

Construct a MIIR object from a standard type

Construct an IIR filter object from a standard type: 'lowpass', 'highpass', 'bandpass' or 'bandreject'

The relevant parameters are:

Key	Description
'type'	one of the types: 'highpass', 'lowpass', 'bandpass', 'bandreject' [default 'lowpass']

You can also specify optional parameters:

Key	Description
'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']
'iunits'	the input unit of the filter
'ounits'	the output unit of the filter

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)
```

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

The relevant parameters for retrieving a IIR filter from a LTPDA repository are:

Key	Description
'hostname'	the repository hostname. [default: 'localhost']
'database'	The database name [default: 'ltpda']
'id'	A vector of object IDs. [default: []]
'cid'	Retrieve all rational objects from a particular collection
'binary'	Set to 'yes' to retrieve from stored binary representation (not always available). [default: yes]

```
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:

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

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

or in a PLIST

The relevant parameters are:

Key	Description
'a'	vector of A coefficients (see note ** below) [default: empty]
'b'	vector of B coefficients (see note ** below) [default: empty]
'fs'	sampling frequency of the filter [default: empty]
'name'	name of filter [default: 'None']

```
a = [0.5 -0.01];
b = [1 0.1];
fs = 1;
name = 'my IIR';
pl = plist('a', a, 'fs', fs, 'name', name);
iir = miir(pl)
```

NOTES:

** The convention used here for naming the filter coefficients is the opposite to MATLAB's convention. The recursion formula for this convention is
$$b(1)*y(n) = a(1)*x(n) + a(2)*x(n-1) + \dots + a(na+1)*x(n-na) - b(2)*y(n-1) - \dots - b(nb+1)*y(n-nb)$$

◀ Constructor examples of the MFIR class

Constructor examples of the PZMODEL class ▶

©LTP Team

Constructor examples of the PZMODEL class

[General information about PZMODEL objects](#)

[Construct empty PZMODEL object](#)

[Construct a PZMODEL object by loading the object from a file](#)

[Construct a PZMODEL object from gain, poles and zeros](#)

[Construct a PZMODEL object from an existing model](#)

[Construct a PZMODEL object from a parameter list \(PLIST\) object](#)

[General information about pole/zero models](#)

[For help in designing a PZMODEL object, see also the PZMODEL Helper GUI documentation](#)

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 MIIR class

Constructor examples of the PARFRAC class ▶

©LTP Team

Constructor examples of the PARFRAC class

[Copy an parfrac object](#)

[Construct an parfrac object by loading the object from a file](#)

[Construct an parfrac object from a rational object](#)

[Construct an parfrac object from a pole/zero model](#)

[Construct an parfrac object from residuals, poles and direct terms](#)

[Construct an parfrac object from a parameter list object \(PLIST\)](#)

Copy an parfrac object

The following example creates a copy of an parfrac object (blue command).

```
>> pf1 = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3)
>> pf2 = parfrac(pf1)
---- parfrac 1 ----
model:      None
res:        [1;2+i*1;2-i*1]
poles:      [6;1+i*3;1-i*3]
dir:        3
pmul:       [1;1;1]
iunits:     []
ounits:     []
-----
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> pf1 = parfrac()
---- parfrac 1 ----
model:      none
res:        []
poles:      []
dir:        0
pmul:       []
iunits:     []
ounits:     []
-----
>> pf2 = pf1;
>> pf2.setName('my new name')
---- parfrac 1 ----
model:      my new name
res:        []
poles:      []
dir:        0
pmul:       []
iunits:     []
ounits:     []
-----
```

If we display pf1 again then we see that the property 'name' was changed although we only have changed pf2.

```
>> pf1
---- parfrac 1 ----
model:      my new name
res:        []
poles:      []
dir:        0
pmul:       []
iunits:     []
ounits:     []
-----
```

Construct an parfrac object by loading the object from a file

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

```
pf = parfrac('parfrac_object.mat')
pf = parfrac('parfrac_object.xml')
```

or in a PLIST

```
pl = plist('filename', 'parfrac_object.xml');
pf = parfrac(pl)
```

Construct an parfrac object from a rational object

The following example creates a new parfrac object from a rational object.

```
>> rat = rational([1 2 3], [4 5 6 7], 'my rational')
---- rational 1 ----
model:      my rational
num:        [1 2 3]
den:        [4 5 6 7]
iunits:     []
ounits:     []
-----
>> pf = parfrac(rat)
---- parfrac 1 ----
model:      parfrac(my rational)
res:        [0.0355+i*0.1682; 0.0355-i*0.1682; 0.1788]
poles:      [-0.021-i*1.2035; -0.0211+i*1.2035; -1.2077]
dir:        0
pmul:       [1;1;1]
iunits:     []
ounits:     []
-----
```

or in a plist

```
>> rat = rational([1 2 3], [4 5 6 7], 'my rational');
>> pl = plist('rational', rat)
>> pf = parfrac(pl)
```

Construct an parfrac object from a pole/zero model

The following example creates a new parfrac object from a pole/zero model.

```
>> pzm = pzmodel(1, {1 2 3}, {4 5})
---- pzmodel 1 ----
name: None
gain: 1
delay: 0
```

```
iunits: []
ounits: []
pole 001: (f=1 Hz, Q=NaN)
pole 002: (f=2 Hz, Q=NaN)
pole 003: (f=3 Hz, Q=NaN)
zero 001: (f=4 Hz, Q=NaN)
zero 002: (f=5 Hz, Q=NaN)
-----
>> pf = parfrac(pzm)
--- parfrac 1 ---
model: parfrac(None)
res: [0.9999999999999999; -6.0000000000000001; 5.999999999999999]
poles: [-18.8495559215388; -12.5663706143592; -6.28318530717959]
dir: 0
pmul: [1;1;1]
iunits: []
ounits: []
-----
```

or in a plist

```
>> pzm = pzmodel(1, {1 2 3}, {4 5})
>> pl = plist('pzmodel', pzm)
>> pf = parfrac(pl)
```

Construct an parfrac object from residuals, poles and direct terms

The following example creates a new parfrac direct from the values.

```
>> res = [1;2+i*1;2-i*1];
>> poles = [6;1+i*3;1-i*3];
>> dir = 3;
>> name = 'my parfrac';
>> iunits = 'Hz';
>> ounits = 'V';

>> pf = parfrac(res, poles, dir)
>> pf = parfrac(res, poles, dir, name)
>> pf = parfrac(res, poles, dir, name, iunits, ounits)
--- parfrac 1 ---
model: my parfrac
res: [1;2+i*1;2-i*1]
poles: [6;1+i*3;1-i*3]
dir: 3
pmul: [1;1;1]
iunits: [Hz]
ounits: [V]
-----
```

Construct an parfrac object from a parameter list (plist)

Constructs an parfrac object from the description given in the parameter list.

- [Use the key word 'hostname'](#)
- [Use the key word 'res' or 'poles' or 'dir'](#)

Use the key word 'hostname'

Construct an parfrac object by retrieving it from a LTPDA repository.

The relevant parameters are:

Key	Description
'hostname'	the repository hostname. [default: 'localhost']

'database'	The database name [default: 'ltpda']
'id'	A vector of object IDs. [default: []]
'cid'	Retrieve all parfrac objects from a particular collection
'binary'	Set to 'yes' to retrieve from stored binary representation (not always available). [default: yes]

```
pl = plist('hostname', '130.75.117.67', 'database', 'ltpda_test', 'id', 1)
a1 = parfrac(pl)
```

Use the key word 'res' or 'poles' or 'dir'

Construct an parfrac object direct from the residual, pole or direct terms.
The relevant parameters are:

Key	Description
'res'	residuals [default: []]
'poles'	poles [default: []]
'dir'	direct terms [default: []]

You can also specify optional parameters:

Key	Description
'name'	name of the parfrac object [default: 'none']
'xunits'	unit of the x-axis
'yunits'	unit of the y-axis

```
res    = [1;2+i*1;2-i*1];
poles  = [6;1+i*3;1-i*3];
dir     = 3;
name    = 'my parfrac';
iunits  = 'Hz';
ounits  = 'V';

pl = plist('res', res, 'poles', poles);
pf = parfrac(pl)
---- parfrac 1 ----
model:      None
res:        [1;2+i*1;2-i*1]
poles:      [6;1+i*3;1-i*3]
dir:        0
pmul:       [1;1;1]
iunits:     []
ounits:     []
-----

pl = plist('res', res, 'poles', poles, 'name', name, 'iunits', iunits, 'ounits',
ounits);
pf = parfrac(pl)
---- parfrac 1 ----
model:      my parfrac
res:        [1;2+i*1;2-i*1]
```

```
poles:    [6;1+i*3;1-i*3]
dir:      0
pmul:     [1;1;1]
iunits:   [Hz]
ounits:   [V]
-----
```

Constructor examples of the RATIONAL class

[Copy an rational object](#)

[Construct an rational object by loading the object from a file](#)

[Construct an rational object from an parfrac object](#)

[Construct an rational object from a pole/zero model](#)

[Construct an rational object from numerator and denominator coefficients](#)

[Construct an rational object from a parameter list object \(PLIST\)](#)

Copy an rational object

The following example creates a copy of an rational object (blue command).

```
>> ra1 = rational([1 2 3], [4 5 6 7], 'my rational');
>> ra2 = rational(ra1)
---- rational 1 ----
model:    my rational
num:      [1 2 3]
den:      [4 5 6 7]
iunits:   []
ounits:   []
-----
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> ra1 = rational()
---- rational 1 ----
model:    none
num:      []
den:      []
iunits:   []
ounits:   []
-----
>> ra2 = ra1;
>> ra2.setName('my new name')
---- rational 1 ----
model:    my new name
num:      [1 2 3]
den:      [4 5 6 7]
iunits:   []
ounits:   []
-----
```

If we display ra1 again then we see that the property 'name' was changed although we only have changed ra2.

```
>> ra1
---- rational 1 ----
model:    my new name
num:      [1 2 3]
den:      [4 5 6 7]
iunits:   []
ounits:   []
-----
```

```
ounits:  []
-----
```

Construct an rational object by loading the object from a file

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

```
ra = rational('rational_object.mat')
ra = rational('rational_object.xml')
```

or in a PLIST

```
pl = plist('filename', 'rational_object.xml');
ra = rational(pl)
```

Construct an rational object from an parfrac object

The following example creates a new rational object from an parfrac object.

```
>> pf = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3, 'my parfrac')
---- parfrac 1 ----
model:    my parfrac
res:      [1;2+i*1;2-i*1]
poles:    [6;1+i*3;1-i*3]
dir:      3
pmul:     [1;1;1]
iunits:   []
ounits:   []
-----
>> ra = rational(pf)
---- rational 1 ----
model:    rational(my parfrac)
num:      [3 -19 30 -110]
den:      [1 -8 22 -60]
iunits:   []
ounits:   []
-----
```

or in a plist

```
>> pf = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3, 'my parfrac');
>> pl = plist('rational', rat)
>> ra = rational(pl)
```

Construct an rational object from a pole/zero model

The following example creates a new rational object from a pole/zero model.

```
>> pzm = pzmodel(1, {1 2 3}, {4 5}, 'my pzmodel')
---- pzmodel 1 ----
name: my pzmodel
gain: 1
delay: 0
iunits: []
ounits: []
pole 001: (f=1 Hz,Q=NaN)
pole 002: (f=2 Hz,Q=NaN)
pole 003: (f=3 Hz,Q=NaN)
zero 001: (f=4 Hz,Q=NaN)
zero 002: (f=5 Hz,Q=NaN)
-----
```

```
>> ra = rational(pzm)
---- rational 1 ----
model:      rational(None)
num:        [0.0012  0.0716  1]
den:        [0.0001  0.0036  0.0401  0.2122  1]
iunits:     []
ounits:     []
-----
```

or in a plist

```
>> pzm = pzmodel(1, {1 2 3}, {4 5}, 'my pzmodel')
>> pl  = plist('pzmodel', pzm)
>> ra  = rational(pl)
```

Construct an rational object from numerator and denominator coefficients

The following example creates a new rational direct from the numerator and denominator coefficients.

```
>> num    = [1 2 3];
>> den    = [4 5 6];
>> name    = 'my rational';
>> iunits  = 'Hz';
>> ounits  = 'V';

>> ra = rational(num, den)
>> ra = rational(num, den, name)
>> ra = rational(num, den, name, iunits, ounits)
---- rational 1 ----
model:      my rational
num:        [1 2 3]
den:        [4 5 6]
iunits:     [Hz]
ounits:     [V]
-----
```

Construct an rational object from a parameter list (plist)

Constructs an rational object from the description given in the parameter list.

[Use the key word 'hostname'](#)

[Use the key word 'num' or 'den'](#)

Use the key word 'hostname'

Construct an rational object by retrieving it from a LTPDA repository.

The relevant parameters are:

Key	Description
'hostname'	the repository hostname. [default: 'localhost']
'database'	The database name [default: 'ltpda']
'id'	A vector of object IDs. [default: []]
'cid'	Retrieve all rational objects from a particular collection

'binary' Set to 'yes' to retrieve from stored binary representation (not always available). [default: yes]

```
pl = plist('hostname', '130.75.117.67', 'database', 'ltpda_test', 'id', 1)
a1 = rational(pl)
```

Use the key word 'num' or 'den'

Construct an rational object direct from the coefficients.
The relevant parameters are:

Key	Description
'num'	numerator coefficients [default: []]
'den'	denominator coefficients [default: []]

You can also specify optional parameters:

Key	Description
'name'	name of the rational object [default: 'none']
'xunits'	unit of the x-axis
'yunits'	unit of the y-axis

```
num    = [1 2 3];
den    = [4 5 6];
name   = 'my rational';
iunits = 'Hz';
ounits = 'V';

pl = plist('num', num, 'den', den);
ra = rational(pl)
---- rational 1 ----
model:      None
num:        [1 2 3]
den:        [4 5 6]
iunits:     []
ounits:     []
-----

pl = plist('num', num, 'den', den, 'name', name, 'iunits', iunits, 'ounits', ounits);
pf = rational(pl)
---- rational 1 ----
model:      my rational
num:        [1 2 3]
den:        [4 5 6]
iunits:     [Hz]
ounits:     [V]
-----
```

Constructor examples of the TIMESPAN class

[Construct empty TIMESPAN object](#)

[Construct a TIMESPAN object by loading the object from a file](#)

[Construct a TIMESPAN object with a start and end time](#)

[Construct a TIMESPAN object from a parameter list \(PLIST\) object](#)

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 PLIST class

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

[Creating parameter lists from parameters](#)

[Creating parameter lists directly](#)

[Appending parameters to a parameter list](#)

[Finding parameters in a parameter list](#)

[Removing parameters from a parameter list](#)

[Setting parameters in a parameter list](#)

[Combining multiple parameter lists](#)

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 TIMESPAN class](#) [Constructor examples of the SPECWIN class ▶](#)

©LTP Team

Constructor examples of the SPECWIN class

[Construct empty SPECWIN object](#)

[Construct a SPECWIN object by loading the object from a file](#)

[Construct a SPECWIN of a particular window type an length](#)

[Construct a SPECWIN Kaiser window with the prescribed psll](#)

[Construct a SPECWIN object from a parameter list \(PLIST\) object](#)

[For help in designing a SPECWIN object, see also the Spectral Window GUI documentation](#)

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

-----
```

Functions – By Category

Categories

- [Constructor](#)
- [Internal](#)
- [Statespace](#)
- [Signal Processing](#)
- [Arithmetic Operator](#)
- [Helper](#)
- [Operator](#)
- [Output](#)
- [Relational Operator](#)
- [Trigonometry](#)
- [MDC01](#)
- [GUI function](#)
- [Converter](#)
- [User defined](#)

Constructor

Function name	Class	Description
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
plist	plist	% PLIST Plist class object constructor.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
pzmodel	pzmodel	% PZMODEL constructor for pzmodel class.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
rational	rational	% RATIONAL rational representation of a transfer function.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
parfrac	parfrac	% PARFRAC partial fraction representation of a transfer function.

rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
timespan	timespan	% TIMESPAN timespan object class constructor.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
ssm	ssm	% SSM statespace model class constructor.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
smodel	smodel	% SMODEL constructor for smodel class.
filterbank	filterbank	% FILTERBANK constructor for filterbank class.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
matrix	matrix	% MATRIX constructor for matrix class.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.
collection	collection	% COLLECTION constructor for collection class.
rebuild	ltpda_uoh	% REBUILD rebuilds the input objects using the history.

[Back to top](#)

Internal

Function name	Class	Description
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
char	ao	% CHAR overloads char() function for analysis objects.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
md5	ao	% MD5 computes an MD5 checksum from an analysis objects.
plot	ao	% PLOT a simple plot of analysis objects.

pwelch	ao	% PWELCH makes power spectral density estimates of the time-series objects
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
getDescriptionForParam	plist	% GETDESCRIPTIONFORPARAM Returns the description for the specified parameter key.
getSetRandState	plist	% GETSETRANDSTATE gets or sets the random state of the MATLAB functions 'rand' and 'randn'
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
parse	plist	% PARSE a plist for strings which can be converted into numbers
plist2cmds	plist	% PLIST2CMDS convert a plist to a set of commands.
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).

submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
setHistin	miir	% SETHISTIN Set the property 'histin'
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
getlowerFreq	pzmodel	% GETLOWERFREQ gets the frequency of the lowest pole or zero in the model.
getupperFreq	pzmodel	% GETUPPERFREQ gets the frequency of the highest pole or zero in the model.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
respCore	pzmodel	% RESPCORE returns the complex response of one pzmodel object.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects

		to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
getlowerFreq	rational	% GETLOWERFREQ gets the frequency of the lowest pole in the model.
getupperFreq	rational	% GETUPPERFREQ gets the frequency of the highest pole in the model.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
respCore	rational	% RESPCORE returns the complex response of one rational object.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
getlowerFreq	parfrac	% GETLOWERFREQ gets the frequency of the lowest pole in the model.
getupperFreq	parfrac	% GETUPPERFREQ gets the frequency of the highest pole in the model.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
respCore	parfrac	% RESPCORE returns the complex response of one

		parfrac object.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA

		Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects

		to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.
bsubmit	ltpda_uo	% BSUBMIT submits the given collection of objects in binary form to an LTPDA Repository.
isprop	ltpda_obj	% ISPROP tests if the given field is one of the object properties.
string	ltpda_uoh	% STRING writes a command string that can be used to recreate the input object(s).
submit	ltpda_uo	% SUBMIT submits the given collection of objects to an LTPDA Repository.
type	ltpda_uoh	% TYPE converts the input objects to MATLAB functions.
update	ltpda_uo	% UPDATE updates the given object in an LTPDA Repository.

[Back to top](#)

Statespace

Function name	Class	Description

[Back to top](#)

Signal Processing

Function name	Class	Description
bin_data	ao	% BIN_DATA rebins aos data, on logarithmic scale, linear scale, or arbitrarily chosen.
compute	ao	% COMPUTE performs the given operations on the input AOs.
confint	ao	% CONFINT Calculates confidence levels and variance for psd, lpsd, cohere, lcohere and curvefit parameters

consolidate	ao	% CONSOLIDATE resamples all input AOs onto the same time grid.
conv	ao	% CONV vector convolution.
corr	ao	% CORR estimate linear correlation coefficients.
cov	ao	% COV estimate covariance of data streams.
curvefit	ao	% CURVEFIT fit a curve to data.
delay	ao	% DELAY delays a time-series using various methods.
detrend	ao	% DETREND detrends the input analysis object using a polynomial of degree N.
dft	ao	% DFT computes the DFT of the input time-series at the requested frequencies.
diff	ao	% DIFF differentiates the data in AO.
dopplercorr	ao	% Dopplercorr coorects data for Doppler shift
downsample	ao	% DOWNSAMPLE AOs containing time-series data.
dropduplicates	ao	% DROPDUPLICATES drops all duplicate samples in time-series AOs.
dsmean	ao	% DSMEAN performs a simple downsampling by taking the mean of every N samples.
eqmotion	ao	% EQMOTION solves numerically a given linear equation of motion
evaluateModel	ao	% EVALUATEMODEL evaluate a curvefit model.
fft	ao	% FFT overloads the fft method for Analysis objects.
fftfilt	ao	% FFTFILT overrides the fft filter function for analysis objects.
filtSubtract	ao	% FILTSUBTRACT subtracts a frequency dependent noise contribution from an input ao.
filter	ao	% FILTER overrides the filter function for analysis objects.

filtfilt	ao	% FILTFILT overrides the filtfilt function for analysis objects.
fixfs	ao	% FIXFS resamples the input time-series to have a fixed sample rate.
gapfilling	ao	% GAPFILLING fills possible gaps in data.
gapfillingoptim	ao	% GAPFILLINGOPTIM fills possible gaps in data.
getdof	ao	% GETDOF Calculates degrees of freedom for psd, lpsd, cohere and lcohere
heterodyne	ao	% HETERODYNE heterodynes time-series.
hist	ao	% HIST overloads the histogram function (hist) of MATLAB for Analysis Objects.
ifft	ao	% IFFT overloads the ifft operator for Analysis objects.
integrate	ao	% INTEGRATE integrates the data in AO.
interp	ao	% INTERP interpolate the values in the input AO(s) at new values.
interpmissing	ao	% INTERPMISSING interpolate missing samples in a time-series.
linSubtract	ao	% LINSUBTRACT subtracts a linear contribution from an input ao.
lincom	ao	% LINCOM make a linear combination of analysis objects
linedetect	ao	% LINEDETECT find spectral lines in the ao/fsdata objects.
lisovfit	ao	% LISOVFIT uses LISO to fit a pole/zero model to the input frequency-series.
ltp_ifo2acc	ao	% LTP_IFO2ACC convert ifo data to acceleration
metropolis1D	ao	% METROPOLIS1D samples a probability distribution for a SISO problem using a Metropolis/Metropolis-Hastings algorithm.
metropolis2D	ao	% METROPOLIS2D samples a probability distribution for a MIMO problem

noisegen1D	ao	% NOISEGEN1D generates colored noise from withe noise.
noisegen2D	ao	% NOISEGEN2D generates cross correleted colored noise from withe noise.
normdist	ao	% NORMDIST computes the equivalent normal distribution for the data.
polyfit	ao	% POLYFIT overloads polyfit() function of MATLAB for Analysis Objects.
psdconf	ao	% PSDCONF Calculates confidence levels and variance for psd
resample	ao	% RESAMPLE overloads resample function for AOs.
rms	ao	% RMS Calculate RMS deviation from spectrum
sDomainFit	ao	% sDomainFit performs a fitting loop to identify model order and
select	ao	% SELECT select particular samples from the input AOs and return new AOs with only those samples.
smallvector_lincom	ao	% SMALLVECTOR_LINCOM does the linear combination for the small vector model.
smallvectorfit	ao	% SMALLVECTORFIT does a linear fit to the small vector model.
smoother	ao	% SMOOTHER smooths a given series of data points using the specified method.
spikecleaning	ao	% spikecleaning detects and corrects possible spikes in analysis objects
split	ao	% SPLIT split an analysis object into the specified segments.
straightLineFit	ao	% STRAIGHTLINEFIT fits a straight line to the given data series
tdfit	ao	% TDFIT fit a set of smodels to a set of input and output signals..
timedomainfit	ao	% TIMEDOMAINFIT uses Iscov to fit a set of time–

		series AOs to a target time-series AO.
upsample	ao	% UPSAMPLE overloads upsample function for AOs.
whiten2D	ao	% WHITEN2D whiten the noise for two cross correlated time series.
xcorr	ao	% XCORR makes cross-correlation estimates of the time-series
xfit	ao	% XFIT fit a function of x to data.
zDomainFit	ao	% zDomainFit performs a fitting loop to identify model order and
zeropad	ao	% ZEROPAD zero pads the input data series.
impresp	ltpda_filter	% IMPRESP Make an impulse response of the filter.
resp	ltpda_filter	% RESP Make a frequency response of the filter.
impresp	ltpda_filter	% IMPRESP Make an impulse response of the filter.
resp	ltpda_filter	% RESP Make a frequency response of the filter.
resp	ltpda_tf	% RESP returns the complex response of a transfer function as an Analysis Object.
simplify	pzmodel	% SIMPLIFY simplifies pzmodels by cancelling like poles with like zeros.
resp	ltpda_tf	% RESP returns the complex response of a transfer function as an Analysis Object.
resp	ltpda_tf	% RESP returns the complex response of a transfer function as an Analysis Object.
bode	ssm	% BODE makes a bode plot from the given inputs to outputs.

[Back to top](#)

Arithmetic Operator

Function name	Class	Description
minus	ao	% MINUS implements subtraction operator for analysis

		objects.
mpower	ao	% MPOWER implements mpower operator for analysis objects.
mrdivide	ao	% MRDIVIDE implements mrdivide operator for analysis objects.
mtimes	ao	% MTIMES implements mtimes operator for analysis objects.
plus	ao	% PLUS implements addition operator for analysis objects.
power	ao	% POWER implements power operator for analysis objects.
rdivide	ao	% RDIVIDE implements division operator for analysis objects.
times	ao	% TIMES implements multiplication operator for analysis objects.
rdivide	pzmodel	% RDIVIDE overloads the division operator for pzmodels.
times	pzmodel	% TIMES overloads the multiplication operator for pzmodels.
minus	smodel	% MINUS implements subtraction operator for ltpda model objects.
mrdivide	smodel	% MRDIVIDE implements mrdivide operator for ltpda model objects.
mtimes	smodel	% MTIMES implements mtimes operator for ltpda model objects.
plus	smodel	% PLUS implements addition operator for ltpda model objects.
rdivide	smodel	% RDIVIDE implements division operator for ltpda model objects.
times	smodel	% TIMES implements multiplication operator for ltpda model objects.
conj	matrix	% conj implements conj operator for matrix objects.

ctranspose	matrix	% CTRANSPOSE implements conjugate transpose operator for matrix objects.
filter	matrix	% FILTER implements N-dim filter operator for matrix objects.
minus	matrix	% MINUS implements subtraction operator for ltpda model objects.
mtimes	matrix	% MTIMES implements mtimes operator for matrix objects.
plus	matrix	% PLUS implements addition operator for matrix objects.
rdivide	matrix	% RDIVIDE implements division operator for matrix objects.
times	matrix	% TIMES implements multiplication operator for matrix objects.
transpose	matrix	% TRANSPOSE implements transpose operator for matrix objects.

[Back to top](#)

Helper

Function name	Class	Description
cat	ao	% CAT concatenate AOs into a row vector.
convert	ao	% CONVERT perform various conversions on the ao.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
demux	ao	% DEMUX splits the input vector of AOs into a number of output AOs.
dx	ao	% DX Get the data property 'dx'.

dy	ao	% DY Get the data property 'dy'.
find	ao	% FIND particular samples that satisfy the input query and return a new AO.
fs	ao	% FS Get the data property 'fs'.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
join	ao	% JOIN multiple AOs into a single AO.
len	ao	% LEN overloads the length operator for Analysis objects. Length of the data samples.
nsecs	ao	% NSECS Get the data property 'nsecs'.
search	ao	% SEARCH selects AOs that match the given name.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setDx	ao	% SETDX sets the 'dx' property of the ao.
setDy	ao	% SETDY sets the 'dy' property of the ao.
setFs	ao	% SETFS sets the 'fs' property of the ao.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setT0	ao	% SETT0 sets the 't0' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
setX	ao	% SETX sets the 'x' property of the ao.

setXY	ao	% SETXY sets the 'xy' property of the ao.
setXunits	ao	% SETXUNITS sets the 'xunits' property of the ao.
setY	ao	% SETY sets the 'y' property of the ao.
setYunits	ao	% SETYUNITS sets the 'yunits' property of the ao.
setZ	ao	% SETZ sets the 'z' property of the ao.
simplifyYunits	ao	% SIMPLIFYYUNITS simplify the 'yunits' property of the ao.
t0	ao	% T0 Get the data property 't0'.
timeshift	ao	% TIMESHIFT for AO/tsdata objects, shifts the time axis such that $x(1) = 0$.
validate	ao	% VALIDATE checks that the input Analysis Object is reproducible and valid.
x	ao	% X Get the data property 'x'.
xunits	ao	% XUNITS Get the data property 'xunits'.
y	ao	% Y Get the data property 'y'.
yunits	ao	% YUNITS Get the data property 'yunits'.
append	plist	% APPEND append a param-object, plist-object or a key/value pair to the parameter list.
combine	plist	% COMBINE multiple parameter lists (plist objects) into a single plist.
find	plist	% FIND overloads find routine for a parameter list.
get	ltpda_obj	% GET get a property of a object.
getIndexForKey	plist	% GETINDEXFORKEY returns the index of a parameter with the given key.
getKeys	plist	% GETKEYS Return all the keys of the parameter list.
getOptionsForParam	plist	% GETOPTIONSFORPARAM Returns the options for the specified parameter key.

getSelectionForParam	plist	% GETSELECTIONFORPARAM Returns the selection mode for the specified parameter key.
isparam	plist	% ISPARAM look for a given key in the parameter lists.
nparams	plist	% NPARAMS returns the number of param objects in the list.
pset	plist	% PSET set or add a key/value pair or a param-object into the parameter list.
remove	plist	% REMOVE remove a parameter from the parameter list.
removeKeys	plist	% REMOVEKEYS removes keys from a PLIST.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setDescriptionForParam	plist	% SETDESCRIPTIONFORPARAM Sets the property 'desc' of the param object in dependencies of the 'key'
setMdlfile	ltpda_uo	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	plist	% SETNAME Set the property 'name'.
setOptionsForParam	plist	% SETOPTIONSFORPARAM Sets the options of the param object in dependencies of the 'key'
setSelectionForParam	plist	% SETSELECTIONFORPARAM Sets the selection mode of the param object in dependencies of the 'key'
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
string	plist	% STRING converts a plist object to a command string which will recreate the plist object.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.

get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
redesign	mfir	% REDESIGN redesign the input filter to work for the given sample rate.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setHistout	ltpda_uoh	
setlunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the transfer functions.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setOunits	ltpda_tf	% SETOUNITS sets the 'ounits' property of the transfer functions.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
redesign	miir	% REDESIGN redesign the input filter to work for the given sample rate.

setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setHistout	ltpda_uoh	
setlunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the transfer functions.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setOunits	ltpda_tf	% SETOUNITS sets the 'ounits' property of the transfer functions.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
setDelay	pzmodel	% SETDELAY sets the 'delay' property of a pole/zero model.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setlunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the transfer functions.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.

setOunits	ltpda_tf	% SETOUNITS sets the 'ounits' property of the transfer functions.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setIunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the transfer functions.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setOunits	ltpda_tf	% SETOUNITS sets the 'ounits' property of the transfer functions.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.

creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setlunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the transfer functions.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setOunits	ltpda_tf	% SETOUNITS sets the 'ounits' property of the transfer functions.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.

setEndT	timespan	% SETENDT Set the property 'endT'.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setStartT	timespan	% SETSTARTT Set the property 'startT'.
setTimeformat	timespan	% SETTIMEFORMAT Set the property 'timeformat'.
setTimezone	timespan	% SETTIMEZONE Set the property 'timezone'.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
addParameters	ssm	% ADDPARAMETERS Adds the parameters to the model.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
duplicateInput	ssm	% DUPLICATEINPUT copies the specified input blocks.
get	ltpda_obj	% GET get a property of a object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
setBlockDescriptions	ssm	% SETBLOCKDESCRIPTIONS Sets descriptions of the specified SSM blocks.
setBlockNames	ssm	% SETBLOCKNAMES Sets names of the specified SSM blocks.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.

setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setParameters	ssm	% SETPARAMETERS Sets the values of the given parameters.
setParams	ssm	% SETPARAMS Sets the parameters of the model to the given plist.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setPortDescriptions	ssm	% SETPORTDESCRIPTIONS Sets descriptions of the specified SSM ports.
setPortNames	ssm	% SETPORTNAMES Sets names of the specified SSM ports.
setPortUnits	ssm	% SETPORTUNITS Sets units of the specified SSM ports.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
conj	smodel	% CONJ gives the complex conjugate of the input smodels
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
double	smodel	% DOUBLE Returns the numeric result of the model.
eval	smodel	% EVAL evaluates the symbolic model and returns an AO containing the numeric data.
fitfunc	smodel	% FITFUNC Returns a function handle which sets the 'values' and 'xvals' to a ltpda model.
get	ltpda_obj	% GET get a property of a object.

index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
op	smodel	% OP Add a operation around the model expression
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setParams	smodel	% SETPARAMS Set the property 'params' AND 'values'
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
setValues	smodel	% SETVALUES Set the property 'values'
setXunits	smodel	% SETXUNITS Set the property 'xunits'.
setXvals	smodel	% SETXVALS Set the property 'xvals'
setXvar	smodel	% SETXVAR Set the property 'xvar'
setYunits	smodel	% SETYUNITS Set the property 'yunits'.
subs	smodel	% SUBS substitutes symbolic parameters with the given values.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.

index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
getObjectAtIndex	matrix	% GETOBJECTATINDEX index into the inner objects of one matrix object.
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
ncols	matrix	% NCOLS Returns the number of columns of the inner object array.
nrows	matrix	% NROWS Returns the number of rows of the inner object array.
osize	matrix	% OSIZE Returns the size of the inner object array.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.

setObjs	ltpda_uoh	
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.
addObjects	ltpda_uoh	
created	ltpda_uoh	% CREATED Returns a time object of the last modification.
creator	ltpda_uoh	% CREATOR Extract the creator(s) from the history.
csvexport	ltpda_uoh	% CSVEXPORT Exports the data of an object to a csv file.
get	ltpda_obj	% GET get a property of a object.
getObjectAtIndex	ltpda_uoh	
getObjectsOfClass	ltpda_uoh	
index	ltpda_uoh	% INDEX index into a 'ltpda_uoh' object array or matrix. This properly captures the history.
nobjs	collection	% NOBJS Returns the number of objects in the inner object array.
removeObjectAtIndex	ltpda_uoh	
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setMdlfile	ltpda_uoh	% SETMDLFILE sets the 'mdlfile' property of the ao.
setName	ltpda_uoh	% SETNAME Set the property 'name'.
setObjectAtIndex	ltpda_uoh	
setObjs	ltpda_uoh	
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an

		ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setUUID	ltpda_uo	% SETUUID Set the property 'UUID'.

[Back to top](#)

Operator

Function name	Class	Description
abs	ao	% ABS overloads the Absolute value method for Analysis objects.
angle	ao	% ANGLE overloads the angle operator for Analysis objects.
complex	ao	% COMPLEX overloads the complex operator for Analysis objects.
conj	ao	% CONJ overloads the conjugate operator for Analysis objects.
ctranspose	ao	% CTRANSPOSE overloads the ' operator for Analysis Objects.
det	ao	% DET overloads the determinant function for Analysis objects.
diag	ao	% DIAG overloads the diagonal operator for Analysis Objects.
eig	ao	% EIG overloads the eigenvalues/eigenvectors function for Analysis objects.
exp	ao	% EXP overloads the exp operator for Analysis objects. Exponential.
hypot	ao	% HYPOT overloads robust computation of the square root of the sum of squares for AOs.
imag	ao	% IMAG overloads the imaginary operator for Analysis objects.
inv	ao	% INV overloads the inverse function for Analysis Objects.

ln	ao	% LN overloads the log operator for Analysis objects. Natural logarithm.
log	ao	% LOG overloads the log operator for Analysis objects. Natural logarithm.
log10	ao	% LOG10 overloads the log10 operator for Analysis objects. Common (base 10) logarithm.
lscov	ao	% LSCOV is a wrapper for MATLAB's lscov function.
max	ao	% MAX computes the maximum value of the data in the AO.
mean	ao	% MEAN computes the mean value of the data in the AO.
median	ao	% MEDIAN computes the median value of the data in the AO.
min	ao	% MIN computes the minimum value of the data in the AO.
mode	ao	% MODE computes the modal value of the data in the AO.
norm	ao	% NORM overloads the norm operator for Analysis Objects.
offset	ao	% OFFSET adds an offset to the data in the AO.
phase	ao	% PHASE overloads the ltpda_phase operator for Analysis objects.
real	ao	% REAL overloads the real operator for Analysis objects.
scale	ao	% SCALE scales the data in the AO by the specified factor.
sign	ao	% SIGN overloads the sign operator for Analysis objects.%
sort	ao	% SORT the values in the AO.
sqrt	ao	% SQRT computes the square root of the data in the AO.

std	ao	% STD computes the standard deviation of the data in the AO.
sum	ao	% SUM computes the sum of the data in the AO.
sumjoin	ao	% SUMJOIN sums time-series signals together
svd	ao	% SVD overloads the determinant function for Analysis objects.
transpose	ao	% TRANSPOSE overloads the .' operator for Analysis Objects.
uminus	ao	% UMINUS overloads the uminus operator for Analysis objects.
unwrap	ao	% UNWRAP overloads the unwrap operator for Analysis objects.
var	ao	% VAR computes the variance of the data in the AO.
tomfir	pzmodel	% TOMFIR approximates a pole/zero model with an FIR filter.
tomiir	pzmodel	% TOMIIR converts a pzmodel to an IIR filter using a bilinear transform.
assemble	ssm	% assembles embedded subsystems, with exogenous inputs
kalman	ssm	% KALMAN applies Kalman filtering to a discrete ssm with given i/o
keepParameters	ssm	% KEEPPARAMETERS enables to substitute symbolic parameters
modifTimeStep	ssm	% MODFTIMESTEP modifies the timestep of a ssm object
sMinReal	ssm	% SMINREAL gives a minimal realization of a ssm object by deleting unreachable states
simplify	simplify	
simulate	ssm	% SIMULATE simulates a discrete ssm with given inputs
subsParameters	ssm	% SUBSPARAMETERS enables to substitute symbolic parameters

convol_integral	smodel	% CONVOL_INTEGRAL implements the convolution integral for smodel objects.
det	smodel	% DET evaluates the determinant of smodel objects.
diff	smodel	% DIFF implements differentiation operator for smodel objects.
fourier	smodel	% FOURIER implements continuous f-domain Fourier transform for smodel objects.
ifourier	smodel	% IFOURIER implements continuous f-domain inverse Fourier transform for smodel objects.
ilaplace	smodel	% ILAPLACE implements continuous s-domain inverse Laplace transform for smodel objects.
inv	smodel	% INV evaluates the inverse of smodel objects.
iztrans	smodel	% IZTRANS implements continuous z-domain inverse Z-transform for smodel objects.
laplace	smodel	% LAPLACE implements continuous s-domain Laplace transform for smodel objects.
simplify	smodel	% SIMPLIFY implements simplify operator for smodel objects.
sop	smodel	% SOP apply a symbolic operation to the expression.
sum	smodel	% SUM adds all the elements of smodel objects arrays.
ztrans	smodel	% ZTRANS implements continuous z-domain Z-transform for smodel objects.
det	matrix	% DET evaluates the determinant for matrix object.
inv	matrix	% INV evaluates the inverse for matrix object.

[Back to top](#)

Output

Function name	Class	Description
display	ao	% DISPLAY implement terminal display for analysis object.

export	ao	% EXPORT export the data of an analysis object to a text file.
iplot	ao	% IPLOT provides an intelligent plotting tool for LTPDA.
iploity	ao	% IPLOT provides an intelligent plotting tool for LTPDA.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
table	ao	% TABLE display the data from the AO in a table.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	plist	% CHAR convert a parameter list into a string.
display	plist	% DISPLAY display plist object.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
char	mfir	% CHAR convert a mfir object into a string.
display	mfir	% DISPLAY overloads display functionality for mfir objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	miir	% CHAR convert a miir object into a string.
display	miir	% DISPLAY overloads display functionality for miir objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.

viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	pzmodel	% CHAR convert a pzmodel object into a string.
display	pzmodel	% DISPLAY overloads display functionality for pzmodel objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	rational	% CHAR convert a rational object into a string.
display	rational	% DISPLAY overloads display functionality for rational objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	parfrac	% CHAR convert a parfrac object into a string.
display	parfrac	% DISPLAY overloads display functionality for parfrac objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	timespan	% CHAR convert a timespan object into a string.
display	timespan	% DISPLAY overloads display functionality for timespan objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input

		objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	ssm	% CHAR convert a ssm object into a string.
display	ssm	% DISPLAY display ssm object.
dotview	ssm	% DOTVIEW view an ssm object via the DOT interpreter.
double	ssm	% Convert a statespace model object to double arrays for given i/o
findParameters	ssm	% findParameters returns parameter names matching the given pattern.
getParameters	ssm	% getParameters returns parameter values for the given names.
getParams	ssm	% GETPARAMS returns the parameter list for this SSM model.
isStable	ssm	% ISSTABLE tells if ssm is numerically stable
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
resp	ssm	% RESP gives the timewise impulse response of a statespace model.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
ssm2dot	ssm	% SSM2DOT converts a statespace model object a DOT file.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	smodel	% CHAR convert a smodel object into a string.
display	smodel	% DISPLAY overloads display functionality for smodel objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.

save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	filterbank	% CHAR convert a filterbank object into a string.
display	filterbank	% DISPLAY overloads display functionality for filterbank objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	matrix	% CHAR convert a matrix object into a string.
display	matrix	% DISPLAY overloads display functionality for matrix objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.
char	collection	% CHAR convert a collection object into a string.
display	collection	% DISPLAY overloads display functionality for collection objects.
report	ltpda_uoh	% REPORT generates an HTML report about the input objects.
save	ltpda_uo	% SAVE overloads save operator for ltpda objects.
viewHistory	ltpda_uoh	% VIEWHISTORY Displays the history of an object as a dot-view or a MATLAB figure.

[Back to top](#)

Relational Operator

Function name	Class	Description
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ge	ao	% GE overloads >= operator for analysis objects. Compare the y-axis values.
gt	ao	% GT overloads > operator for analysis objects. Compare the y-axis values.
le	ao	% LE overloads <= operator for analysis objects. Compare the y-axis values.
lt	ao	% LT overloads < operator for analysis objects. Compare the y-axis values.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	plist	% EQ overloads the == operator for ltpda plist objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.

eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.
eq	ltpda_obj	% EQ overloads the == operator for ltpda objects.
ne	ltpda_obj	% NE overloads the ~= operator for ltpda objects.

[Back to top](#)

Trigonometry

Function name	Class	Description
acos	ao	% ACOS overloads the acos method for Analysis objects.
asin	ao	% ASIN overloads the asin method for Analysis objects.
atan	ao	% ATAN overloads the atan method for Analysis objects.
atan2	ao	% ATAN2 overloads the atan2 operator for Analysis objects. Four quadrant inverse tangent.
cos	ao	% COS overloads the cos method for Analysis objects.
sin	ao	% SIN overloads the sin method for Analysis objects.
tan	ao	% TAN overloads the tan method for Analysis objects.

[Back to top](#)

MDC01

Function name	Class	Description
ifo2acc	ao	% IFO2ACC convert ifo data to acceleration
mdc1_cont2act_utn	ao	% mdc1_cont2act_utn simulate the effect of retarded actuators
mdc1_ifo2acc_fd	ao	% MDC1_IFO2ACC_FS calculates the external acceleration in the frequency-domain.
mdc1_ifo2acc_fd_utn	ao	% mdc1_ifo2acc_fd_utn convert ifo data to acceleration
mdc1_ifo2acc_inloop	ao	% MDC1_IFO2ACC_INLOOP calculates the inloop acceleration in the time-domain.
mdc1_ifo2cont_utn	ao	% mdc1_ifo2cont_utn simulate the effect of the controller
mdc1_ifo2control	ao	% MDC1_IFO2CONTROL converts the input time-series to control forces.
mdc1_x2acc	ao	% MDC1_X2ACC converts the input time-series to acceleration with a time-domain filter

[Back to top](#)

GUI function

Function name	Class	Description

[Back to top](#)

Converter

Function name	Class	Description
ssm2miir	ssm	% SSM2MIIR converts a statespace model object to a miir object
ssm2pzmodel	ssm	% SSM2PZMODEL converts a time-continuous statespace model object to a pzmodel
ssm2rational	ssm	% SSM2RATIONAL converts a statespace model object to a rational frac. object

ssm2ss	ssm	% SSM2SS converts a statespace model object to a MATLAB statespace object.

[Back to top](#)

User defined

Function name	Class	Description

[Back to top](#)

◀ Constructor examples of the SPECWIN class	LTPDA Training Session 1 ▶

©LTP Team

LTPDA Training Session 1

This series of help pages constitute the first training session of LTPDA. The various data-packs used throughout the tutorials are available for download on the [LTPDA web-site](#).

1. [Topic 1 – The basics of LTPDA](#)
2. [Topic 2 – Pre-processing of data](#)
3. [Topic 3 – Spectral Analysis](#)
4. [Topic 4 – Transfer function models and digital filtering](#)
5. [Topic 5 – Model fitting](#)

In addition, throughout the course of this training session, we will perform a full analysis of some lab data. The inputs to the analysis are two time-series data streams, the first is the recorded output of an interferometer, the second is a recording of the room temperature in the vicinity of the interferometer. Both are recorded with different sample rates and on different sampling grids. The temperature data is unevenly sampled, and may even have missing samples.

During each topic of the training session, the data will be manipulated using the tools introduced in that topic (and previous topics). The aim of the data analysis is to determine the influence of temperature on the interferometer output. In particular the steps will be:

1. [Topic 1](#) Loading and calibrating the raw data.
 1. Read in the raw data files and convert them to AOs
 2. Plot the two data streams
 3. Calibrate the interferometer output to meters (from radians)
 4. Calibrate the temperature data to degrees Kelvin from degrees Celcius
 5. Save the calibrated data series to XML files, ready for the input to the next topic
2. [Topic 2](#) Pre-processing and data conditioning.
 1. Read in the calibrated AOs from XML files
 2. Trim the data streams to the same time segments
 3. Resample the temperature on to an even sampling grid with no missing samples
 4. Resample to the two data streams to a common 1Hz sample rate
 5. Interpolate the two data streams on to the same time grid
 6. Save the cleaned data to AO XML files
3. [Topic 3](#) Spectral analysis.
 1. Load the time-series data from Topics 1 and 2
 2. Compare PSDs of the time-series data before and after pre-processing
 3. Check the coherence of temperature and IFO output before and after pre-processing
 4. Measure the transfer function from temperature to IFO output
 5. Save the measured transfer function to disk as an AO XML file
4. [Topic 4](#) Simulation of the system under investigation.
 1. Make approximate noise-shape models for the temperature and IFO displacement input spectra
 2. Make digital IIR filters matching these noise-shape models
 3. Filter white-noise data streams to produce simulated versions of the temperature and IFO inputs
 4. Make a model of the temperature to IFO coupling
 5. Construct a filter representing this coupling
 6. Filter the simulated temperature data and add it to the simulated IFO input data

7. Save the simulated temperature and the simulated IFO output data to disk
8. Repeat the steps from Topic 3, this time using the simulated data
5. [Topic 5](#) Model fitting and system identification.
 1. Load the measured transfer function from the end of Topic 3
 2. Fit a model transfer function to this measurement
 3. Make a digital filter representation of the fitted model
 4. Filter the temperature data with this filter
 5. Compare the PSD of the filtered temperature data and the IFO output
 6. Subtract the filtered temperature data from the IFO output
 7. Compare the IFO data with the temperature influence subtracted to the original IFO output
 8. (Time permitting) Repeat the exercise for the simulated from Topic 4
 9. (Still need something to do?) Repeat the steps of Topic 4 but this time fit a model to the measured temperature data and use a noise generator to make a simulated temperature data stream

◀ Functions – By Category

Topic 1 – The basics of LTPDA ▶

©LTP Team

Topic 1 – The basics of LTPDA

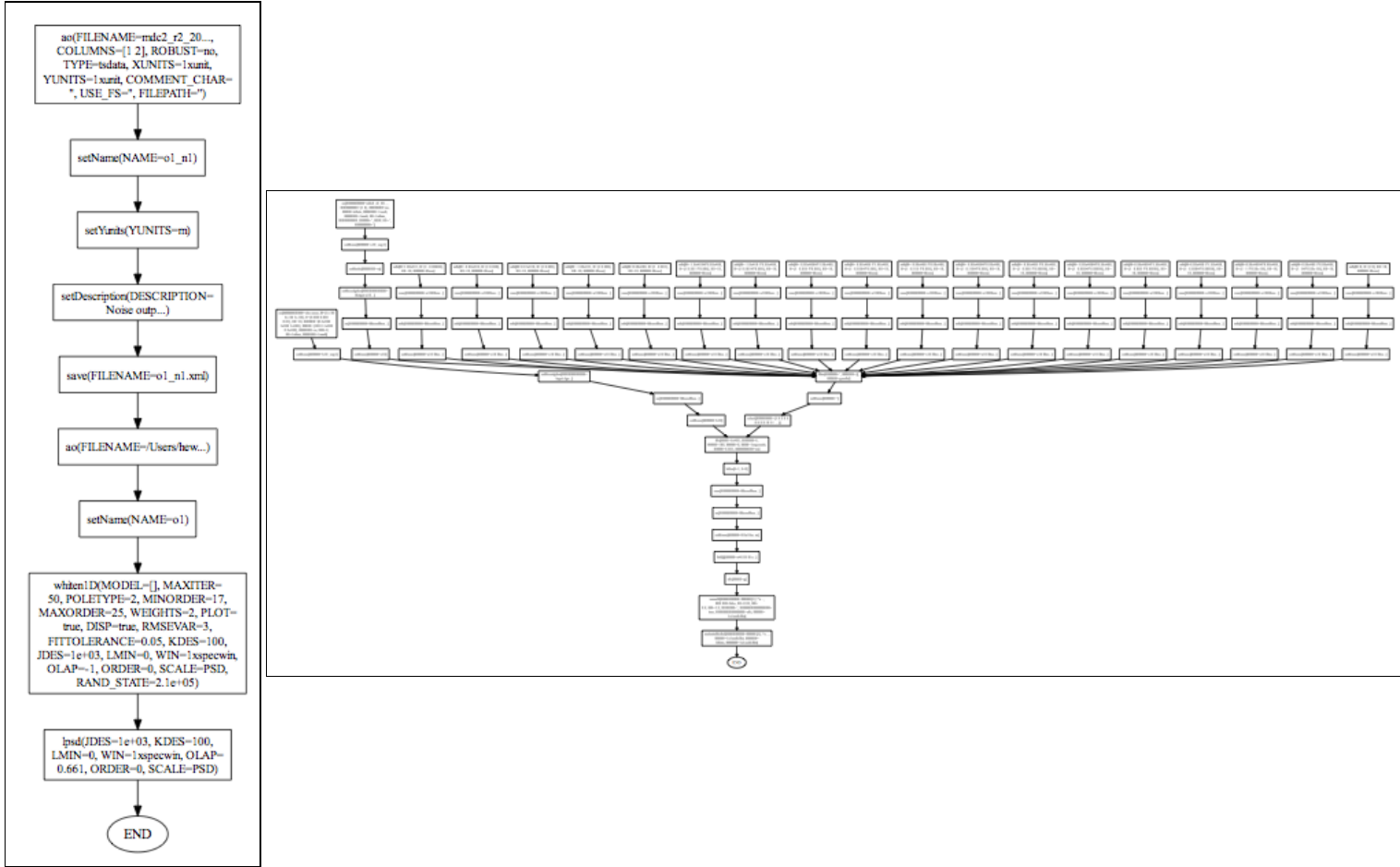
Topic 1 of the training session aims to introduce the basic use of LTPDA. After working through the examples you should be familiar with:

- constructing Analysis Objects (AOs) from simulated data
- constructing AOs from data files
- setting the properties of an AO
- basic operations on AOs, for example, adding and multiplying AOs together
- plotting the data in AOs
- viewing the history of an AO

Introducing Analysis Objects

Analysis objects are one type of LTPDA user object. The job of an analysis object is to bring together some numerical data with a set of descriptive meta-data. In addition, AOs are clever and can keep track of all the things you do to them. So at any time, you can look at the history of the AO to see what processing steps have happened in the past.

Example history trees are shown below:



Making AOs

Exercise 1 – Your first Analysis Object

AOs can be constructed in many different ways. Each of the different ways is called a constructor. For example, there is a constructor to make an AO from a set of numeric values, there is also a constructor to make an AO from a data file. Each time you construct an AO, you make an instance of the class, `ao`. The variable you have in MATLAB is then just a reference to the object you constructed.

This may all sound confusing to start with, but will become clearer as we go through the examples below.

Let's make an AO. On the MATLAB terminal, type the following: `a = ao` and hit return. You should see an output like the following:

```
>> a = ao
M: running ao/ao
M: running ao/display
----- ao 01: a -----

      name:  none
      data:  None
      hist:  ao / ao / SId: ao.m,v 1.282 2009/08/20 19:38:09 nicola Exp S
      mdlfile:  empty
description:
      UUID:   e2b1c39e-a0ef-42d0-a72c-68958f6a57a5
-----
```

Note that the number of lines beginning with the "M:" syntax may vary depending on the level of "verbosity" that can be set via the [LTPDA Preferences](#). You have just made your first AO. It's not a very exciting AO since it contains no data, but it is an AO nonetheless. So now let's make an AO with some data in it. Type the following in to the MATLAB terminal: `a = ao(1)` and hit return. You should see

```
>> a = ao(1)
M: running ao/ao
M: constructing from values
M: running ao/display
----- ao 01: a -----

      name:  None
      data:  1
              ----- cdata 01 -----
              y:  [1x1], double
              dy:  [0x0], double
              yunits:  []
              -----

      hist:  ao / ao / SId: fromVals.m,v 1.29 2009/08/07 17:43:55 ingo Exp $-->$Id: ao.m,v 1.282 2009/08/20 19:38:09 nicola Exp S
      mdlfile:  empty
description:
      UUID:   9ef97503-a12d-4651-9025-a792ddc745c5
-----
```

Now you can see that your AO has some data. The data is of type `cdata` (more on that later), it has no Y units, and it contains a single value, 1.

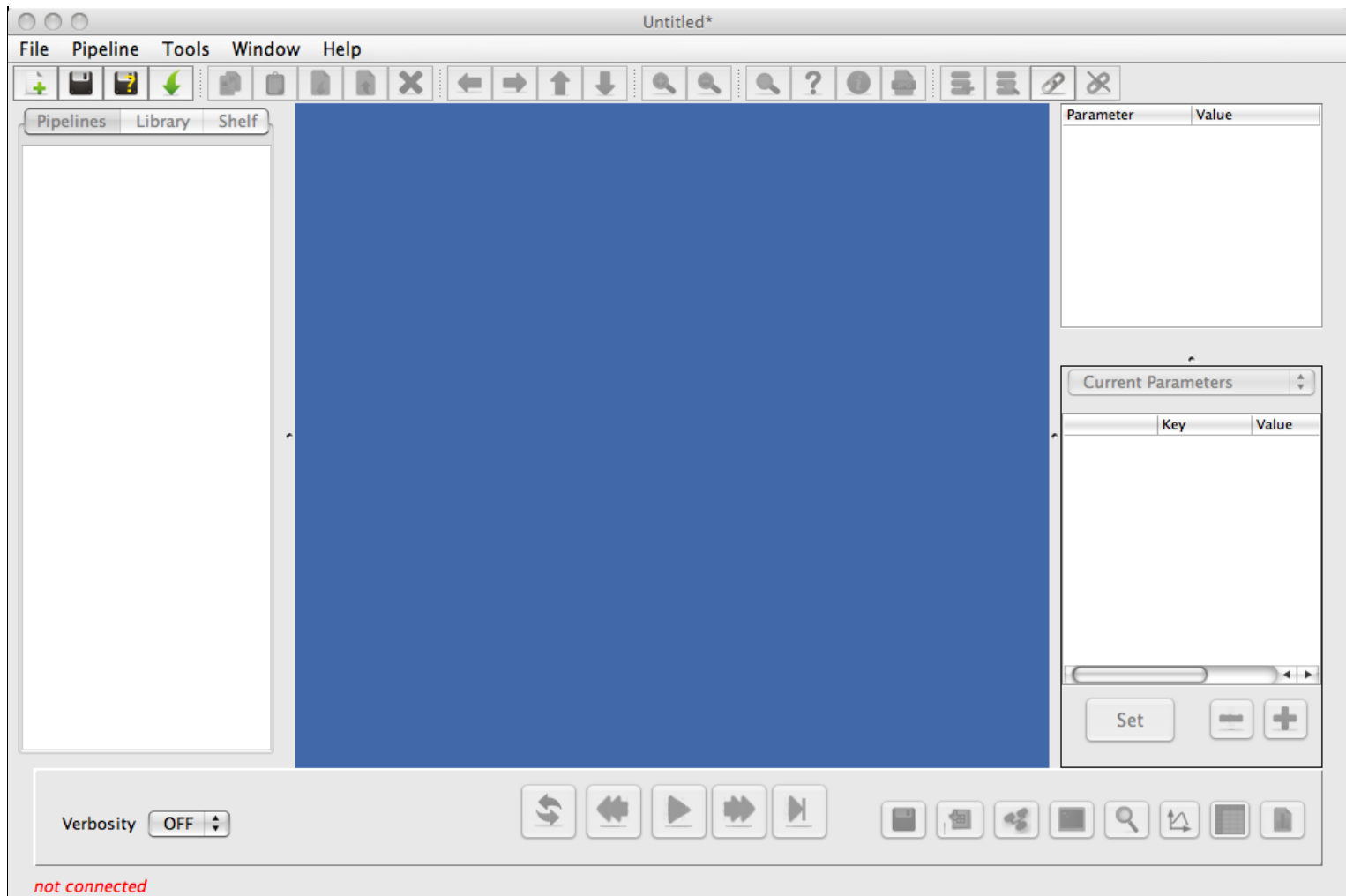
Note also that the information shown in the "hist" field may vary depending on the version of the LTPDA TOolbox you installed.

In addition to the standard MATLAB scripting interface, LTPDA offers a graphical programming environment where the user can put together signal processing pipelines by dragging and dropping blocks on to a canvas, then joining up the blocks.

This graphical programming environment is called an LTPDA Workbench. To start the workbench, issue the following command on the MATLAB terminal, or click on the "LTPDA Workbench" button on the launch bay.

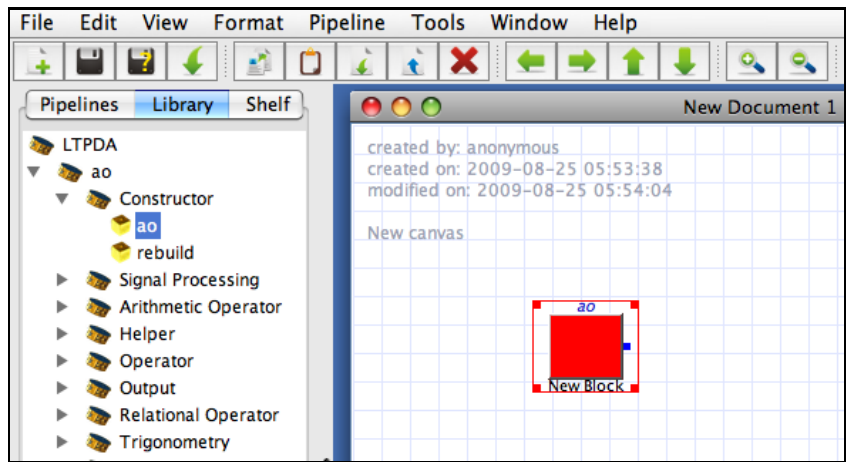
```
LTPDAworkbench
```

You should see a window like the one below:



The use of the LTPDA workbench is quite intuitive, so hopefully playing around is sufficient. Further details of using the workbench environment can be found in the [appropriate section](#) of the user manual.

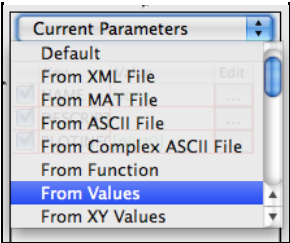
To construct the simple AO as we did above on the terminal, first create an empty pipeline in the workbench by going to "Pipeline->New Pipeline" or hit `ctrl-n` (`cmd-n` on OS X). Then you can drag an AO constructor block from the Library on the left. You can also double click on the block in the library to add it to the canvas.



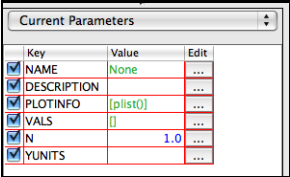
Blocks can be added to the canvas using the "Quick Block" dialog. Hit `ctrl-b` (`cmd-b` on OS X) on the canvas to open the quick block dialog. Begin typing to find the block you want (e.g. 'a' 'o') the hit `enter` to add that block to the canvas. Hit `enter` to add multiple blocks the same. Hit `escape` to dismiss the dialog.

To set the value as we did on the terminal (`ao(1)`) we set some parameters on the block. Follow these steps:

1. Click on the AO block to select it
2. Select the pre-defined parameter set "From Values" from the drop-down menu at the right of the workbench

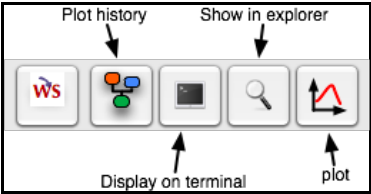


3. You should see the parameter list like



- 4. To set this parameter list to the block, click the **Set** button below the parameter table
- 5. You can now edit this parameter list, for example, add or remove parameters or edit parameter key names and values.
- 6. Edit the value for the key "VALS" by double clicking on the table cell
- 7. A more sophisticated way to edit the parameter is available by clicking on the "Edit" symbol
- 8. Enter a new value (any MATLAB expression) in the dialog box and click the OK button
- 9. To set the name of the block, double-click it and enter a new name in the dialog box

You can now execute the pipeline by clicking on the play button . To display the result of your pipeline, select the blocks you are interested in the outputs of, then you can click on the various 'output' buttons:



These output buttons only work if the pipeline has been successfully executed so that the variables corresponding to the various blocks are in the MATLAB workspace. If the block property "Keep Result" is set to "false", then the output buttons won't work for that block because the result of executing that block is cleared from the MATLAB workspace as soon as the result is no longer needed by other blocks.

Exercise 2 – Setting properties of AOs

We can now go on and manipulate this AO. For example, suppose we want to set its name. Type the following in to the MATLAB terminal: `a.setName('Bob')` and hit enter. You should see:

```
>> a.setName('Bob')
M: running ao/display
----- ao 01: Bob -----
name: Bob
data: 1
----- cdata 01 -----
      y: [1x1], double
      dy: [0x0], double
yunits: []
-----
hist: ltpda_uoh / setName / Sid: setName.m,v 1.9 2009/01/23 09:48:02 hewitson Exp S
mdlfile: empty
description:
  UUID: 7450cdc4-c218-438f-94fe-d9cb44c6e43e
-----
```

The ao has a new name. The function (or more strictly, method) `setName` has acted on the AO, `a`. An equivalent statement would be: `setName(a, 'Bob')`.

By doing this, you have modified `a`. If instead you do

```
b = setName(a, 'Bob')
```

or

```
b = a.setName('Bob')
```

then you get a new variable, `b`, which is a distinct (deep) copy of `a`. The original AO, `a`, has not been modified. Try this out.

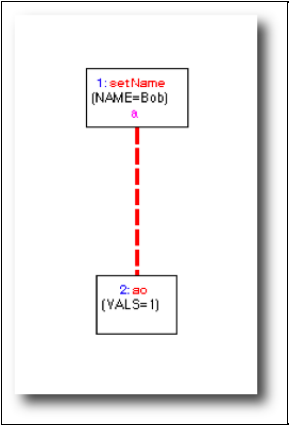
You can do the same on the workbench by using a `setName` block. To use the 'modifier' behaviour, you set the block to be a modifier in the 'block properties' table:

Property	Value
Name	New Block
Keep Result	☒
Modifier	☐
Attach workbench	☐

On the workbench, the `setName` method is automatically called on the output of constructor blocks so that the 'name' you give to a constructor block is set to the object that's constructed. More about this later.

Exercise 3 – Viewing the history

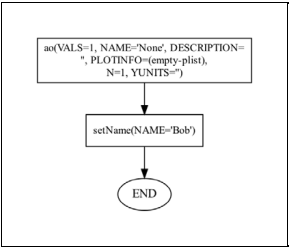
We can now look at the history of this object (in case our memory is really short). The history can be viewed in different ways. For short history trees, the easiest is to use the MATLAB-based history plotter built in to LTPDA. In the MATLAB terminal, type `plot(a.hist)` and hit return. You should get a MATLAB figure looking something like the picture below. You can see the only things we have done are to construct the object and set its name.



For very complicated history plots, LTPDA also supports viewing the history using [Graphviz](#). If your machine has graphviz already installed, and you've set this up in the [LTPDA Preferences](#), then you can immediately do:

```
dotview(a.hist, plist('filename', 'tmp.pdf'))
```

and you should get a figure something like that below in your system pdf viewer.

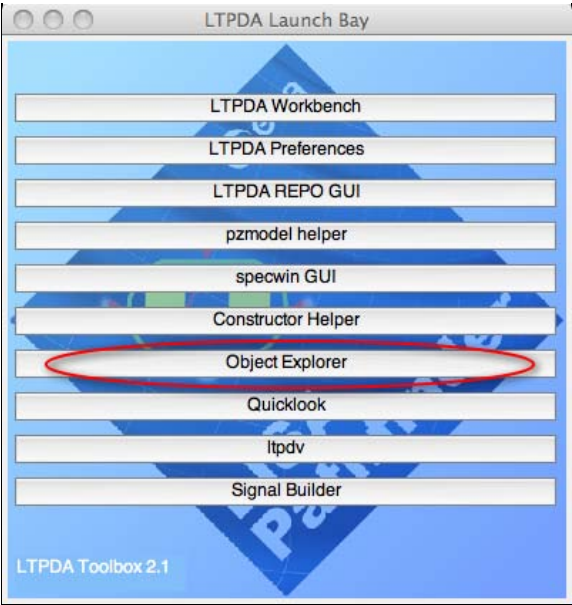


 When you click on the 'display history' button on the workbench, a filename created from combining the pipeline name and the block name is used in the call to `dotview`. You will find the PDF file in the current MATLAB working directory.

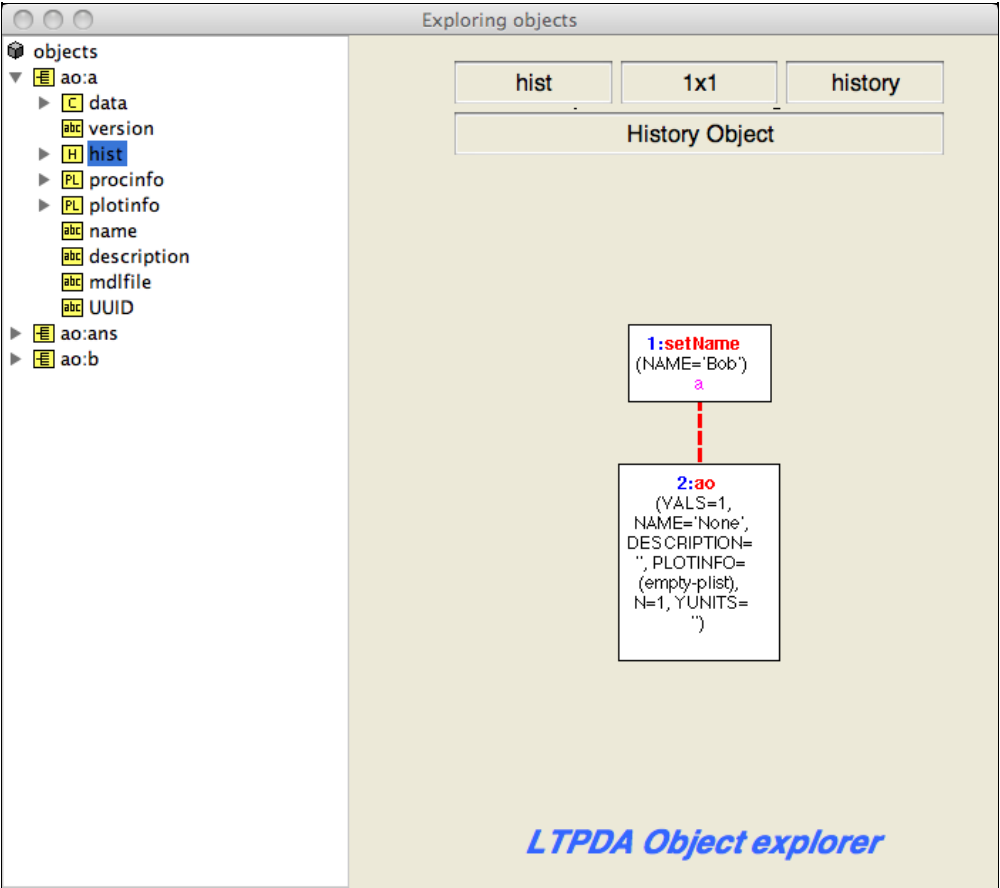
Don't worry about all this `plist` business, we'll get to that soon enough. For now it's enough to know that the conversion to pdf is done by the graphviz engine, and this needs to write the pdf to a file. That's the 'filename' specified in that last command.

Installation of graphviz is covered in the LTPDA user manual under the section [System Requirements](#).

There is a third option for viewing the history: using the LTPDA Explorer. On the MATLAB terminal, type `ltpda_explorer` and hit return, or click the "Object Explorer" button on the LTPDA launch bay (see figure below). If the launch bay is not open, you can open it with the command: `ltpdalauncher`.



Once you have launched the explorer, you can navigate through the various LTPDA objects that are in your MATLAB workspace. Currently, if you add objects to the MATLAB workspace, they will not appear in the LTPDA Explorer until you restart it.



We said earlier that the AO we created has no Y units set. If you look at the output on the MATLAB terminal you will see that the Y units is actually a property of the data, not of the AO. This is because the data inside the AO is actually an object in its own right. There exist 4* data types in LTPDA:

Data class	Description
cdata	Intended for storing an arbitrary matrix of values. This class has two main fields: the data itself is stored in the field <code>y</code> , and the units of the data in <code>yunits</code> .
tsdata	Intended for storing time-series data. More details on this one later.
fsdata	For storing frequency-series data.
xydata	For storing an arbitrary set of x-y data pairs.

* there is actually a 5th data type in development for storing X-Y-Z data, for example for time-frequency maps.

Getting back to our Y units. To set the value of the Y units, the AO class has a method called (not surprisingly) `setYunits`. To set the Y

units of this AO, type the following in to the MATLAB terminal: `a.setYunits('km')` and hit return. You should see the following output:

```
>> a.setYunits('km')
M: running ao/setYunits
M: running ao/display
----- ao 01: Bob -----
      name: Bob
      data: 1
            ----- cdata 01 -----
                  y: [1x1], double
                  dy: [0x0], double
            yunits: [km]
            -----
      hist:  ao / setYunits / SId: setYunits.m,v 1.17 2009/07/30 15:26:05 ingo Exp S
      mdlfile: empty
      description:
      UUID:      e933138b-736b-4c40-9f69-7d9ffb630aa2
      -----
```

Now you see that the AO has Y units of 'km'. (To get a list of supported units in ltpda, type the following command in to the MATLAB terminal: `unit.supportedUnits`. To get a list of supported prefixes, type `unit.supportedPrefixes`.)

Making a time-series AO

Exercise 4

Time-series data are stored in a data object of the class `tsdata`. As a user, you don't need to care about this, but it's sometimes nice to know how things work. There are various ways (constructors) to build time-series AOs. For example, you can give a set of values and a sample rate like

```
a = ao([1 2 3 4 5], 2)
```

The first argument is the Y data vector; the second, the sample rate.

If you run this command in the MATLAB terminal you should see

```
>> a = ao([1 2 3 4 5], 2)
M: running ao/ao
M: constructing from Y values and fs
M: running ao/display
----- ao 01: a -----

      name: None
      data: (0,1) (0.5,2) (1,3) (1.5,4) (2,5)
            ----- tsdata 01 -----

              fs: 2
              x: [1 5], double
              y: [1 5], double
              dx: [0 0], double
              dy: [0 0], double
      xunits: [s]
      yunits: []
      nsecs: 2.5
      t0: 1970-01-01 00:00:00.000
      -----

      hist: ao / ao / SId: fromVals.m,v 1.26 2009/07/02 13:45:21 nicola Exp $-->$Id: ao.m,v
1.282 2009/08/20 19:38:09 nicola Exp S
      mdlfile: empty
      description:
      UUID: 76663c08-e0e7-4753-9a3b-4d72619ff9ea
      -----
```

Now you see that the data type is `tsdata` and the X units are automatically set to seconds ('s'). You can also see that the data series spans 2.5s and that the first sample corresponds to 1970-01-01 00:00:00.000 UTC. You can set further properties of the object, for example

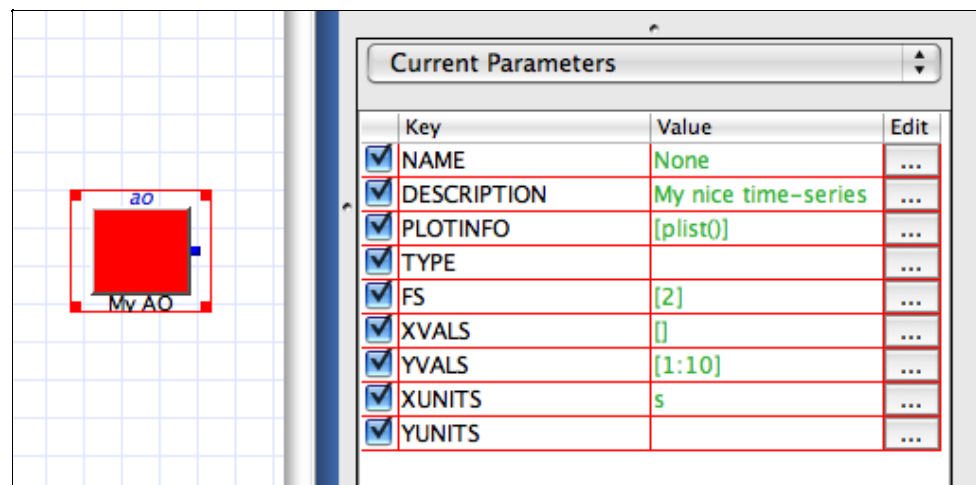
```
a.setT0('2009-02-03 12:23:44')
a.setDescription('My lovely time-series')
```

You can do all of this in one block on the workbench. To do that:

1. Start the workbench and create a new pipeline
2. Drag an AO constructor block from the library (or use "Quick Block")
3. Select the block and select the "From XY Values" parameter set
4. Click the "Set" button to set the parameters to the block
5. Double-click the value cell for the key "YVALS" and enter some values, e.g., 1:10
6. Double-click the value cell for the key "FS" and enter a sample frequency, e.g., 2. By setting a set of values for the Y-data and a sample rate, we tell the AO constructor that we want to build a `tsdata` AO.
7. To set the name of the block, double click the block and enter a name in the dialog box. Automatic setting of AO names from the block name only happens for constructor blocks. To

- the set the name of AOs which are outputs of all other block types, use the `setName` block.
- You'll notice that the parameter list doesn't contain a `T0` parameter by default, but you can easily add this parameter by clicking on the "plus" button below the parameter list. Enter the key `T0` in the dialog box, and an appropriate value in the next dialog box. (Note: parameter key names are case insensitive.)
 - You can do the same for the description, or any other property of the AO.

The final parameter list in this case might look like:



Digression: Introducing parameter lists

The time has come to go back to that `plist` command we saw earlier when plotting the AO history via the `graphviz` renderer.

The following two commands are equivalent:

```
a = ao([1 2 3 4 5], 2)
a = ao(plist('yvals', [1 2 3 4 5], 'fs', 2))
```

Here we introduce the idea of parameter lists (`plist`). A `plist` is a list of parameters, each parameter being defined by a key/value pair. The key of a `plist` is always a string and is always case insensitive. The value can be anything: a number, a string, another LTPDA object, a cell-array, a structure, etc. For more information about parameter lists, see the [appropriate section](#) of the LTPDA user manual.

Going on with time-series objects: The following is almost equivalent:

```
a = ao(plist('xvals', [0 0.5 1 1.5 2], 'yvals', [1 2 3 4 5]))
```

The difference is, if you run this command, you will see that the resulting AO has data of type `xydata`. To make this a time-series object, we need to tell the constructor some more information. Either you need to specify the sample-rate, or you can explicitly set the data type:

```
a = ao(plist('xvals', [0 0.5 1 1.5 2], 'yvals', [1 2 3 4 5], 'fs', 2))
a = ao(plist('xvals', [0 0.5 1 1.5 2], 'yvals', [1 2 3 4 5], 'dtype', ...
            'tsdata'))
```

The elipsis (...) in MATLAB means join the two lines.

If you specify the samples rate with the key `'fs'`, then the `'xvals'` are just ignored. If you tell the data type with the key `'dtype'`, then the sample rate is computed from the `'xvals'`.

You can add additional parameters to these constructor lines. For example,

```
a = ao(plist('xvals', [0 0.5 1 1.5 2], 'yvals', [1 2 3 4 5], ...
            'dtype', 'tsdata', ...
            'name', 'Bob',
            't0', '2008-09-01'))
```

There are other constructors which make constructing time-series AOs from simulated data more convenient. Two of these are discussed below.

Times-series AO as a function of t

If you want to specify your time-series as a function of the variable t , then you can use the following constructor:

```
a = ao(plist('tsfcn', 't.^2 + t',
            'fs', 10, 'nsecs', 1000))
```

You specify the function of t with the key 'tsfcn', then give the sample rate and the number of seconds. If you run this command you should see the output:

```
>> a = ao(plist('tsfcn', 't.^2 + t', 'fs', 10, 'nsecs', 1000))
M: running ao/ao
M: constructing from plist
M: running ao/display
----- ao 01: a -----

name: None
data: (0,0) (0.1,0.11) (0.2,0.24) (0.3,0.39) (0.4,0.56) ...
----- tsdata 01 -----

fs: 10
x: [10000 1], double
y: [10000 1], double
dx: [0 0], double
dy: [0 0], double
xunits: [s]
yunits: [m]
nsecs: 1000
t0: 1970-01-01 00:00:00.000
-----

hist: ao / ao / SId: fromTSfcn.m,v 1.20 2009/08/05 10:35:55 hewitson Exp $-->$Id: ao.m,v
1.282 2009/08/20 19:38:09 nicola Exp S
mdlfile: empty
description:
UUID: 9cac42cc-4779-4204-8885-5d0503649db8
-----
```

You can write any valid MATLAB expression as a function of t .

Plists can be reused, of course. Suppose we define a recipe for an AO as

```
pl = plist('tsfcn', 'randn(size(t))', 'fs', 10, 'nsecs', 1000)
```

then we can make repeated AOs from this recipe:

```
a1 = ao(pl)
a2 = ao(pl)
```

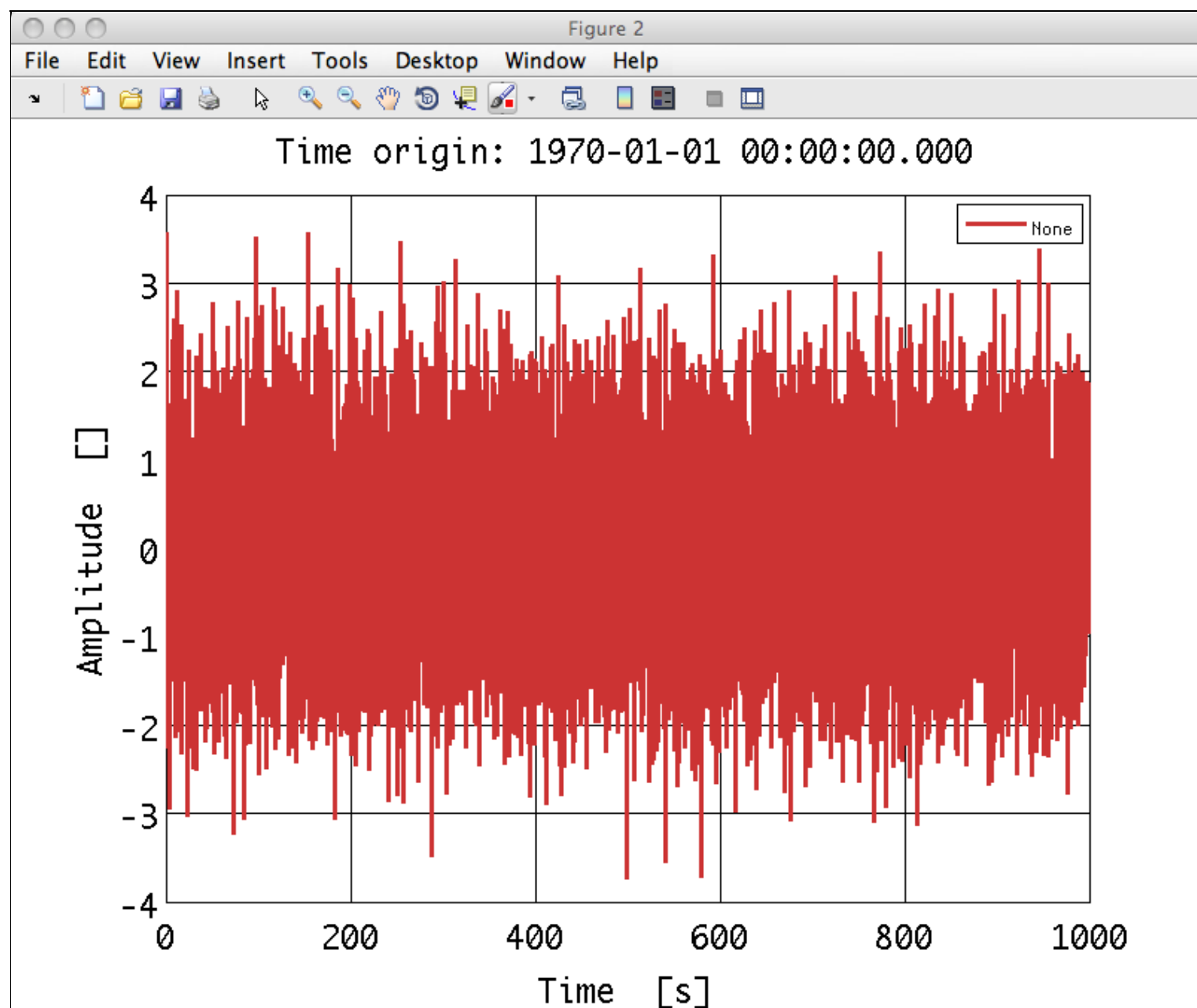
Here we have made two AOs with different random white-noise data vectors.

Digression: plotting the data

To plot the data in the AO, you can use the intelligent plotting method, `ipplot`. For example, type in the MATLAB terminal:

```
a1.ipplot
```

and you should see a plot like the one below.



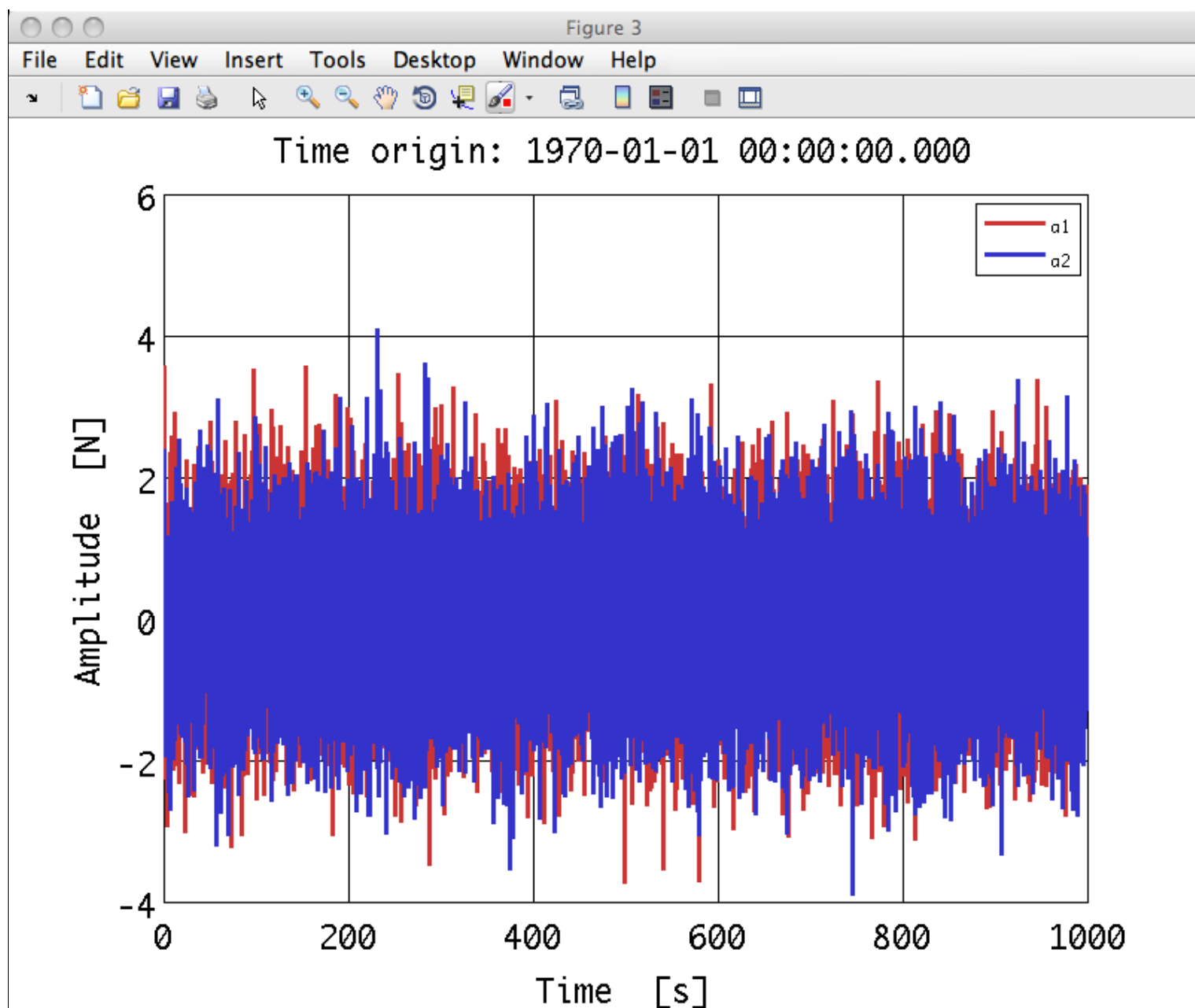
We can make a more interesting plot if we first specify some of the properties of the AOs. For example, type the following commands to set the names and Y units of the two AOs we made earlier:

```
a1.setName
a2.setName
setYunits(a1,a2, 'N')
```

Now plot both time-series together with:

```
ipplot(a1,a2)
```

and you should see a plot like the following:



Calling the `setName` method with no input argument causes the AO to be named with the variable name.

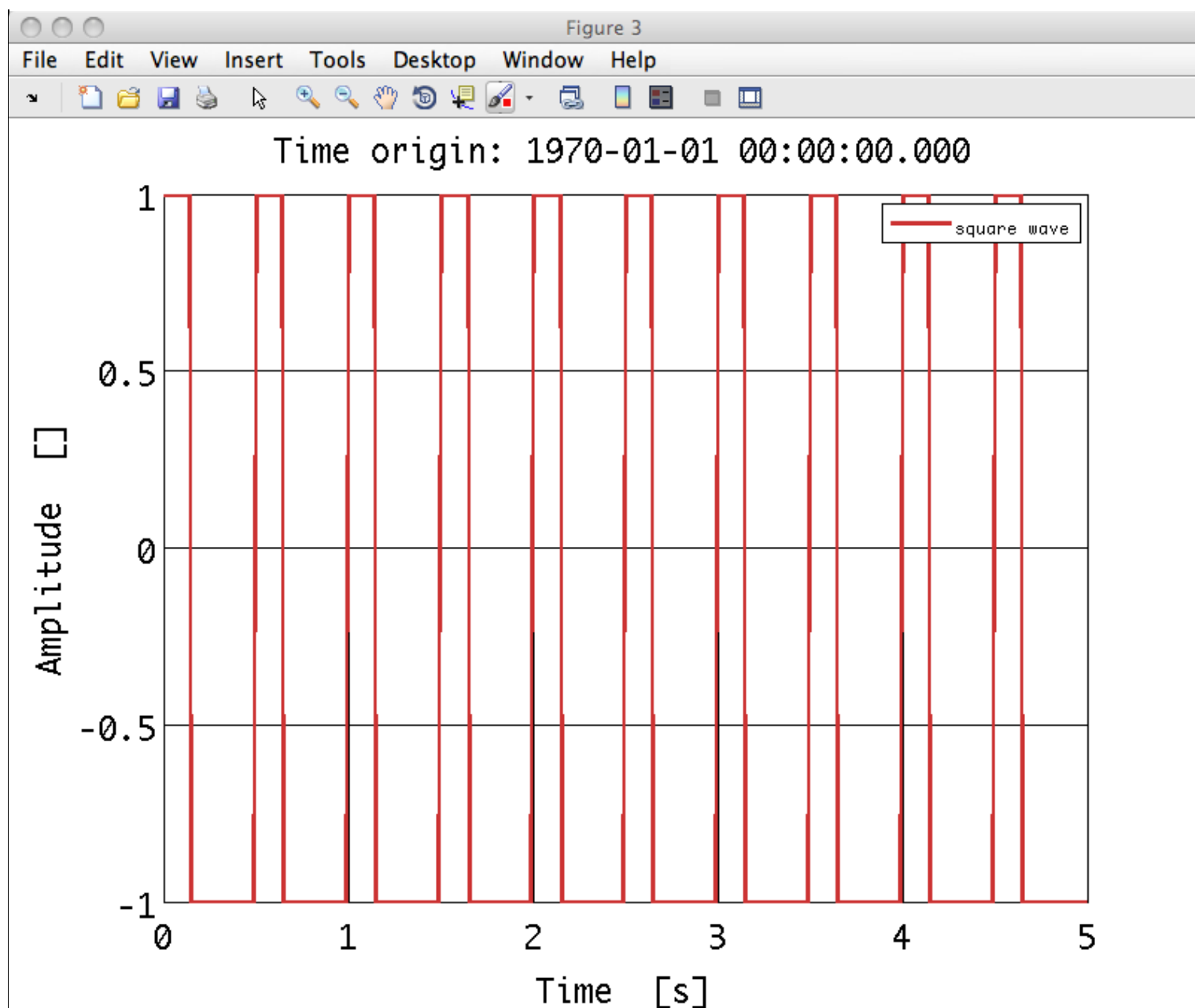
`iplot` has many configurable parameters which are (mostly) documented in the help.

Times-series AO from built in waveforms

MATLAB has various functions for creating standard waveforms, for example, sine waves, square waves, and saw-tooth signals. These are available as convenient AO constructors. For example suppose we want to create a square-wave pulse train with a 30% duty cycle at 2Hz sampled at 100Hz lasting for 5s, then we can do

```
sw = ao(plist('waveform', 'square wave', 'f', 2, 'duty', 30, ...
            'fs', 100, 'nsecs', 5))
```

If you run that command and plot the result, you should see the square wave you were expecting:



You can construct various different waveforms, but each has different parameters to set. The help of the AO method details the possibilities (`help ao`); here is the relevant extract:

```
'waveform' - a waveform description (see options below).

You can also specify additional parameters:
'fs'      - sampling frequency [default: 10 Hz]
'nsecs'   - length in seconds [default: 10 s]
't0'      - time-stamp of the first data sample [default time(0)]

and, for the following waveform types:
'sine wave' - 'A', 'f', 'phi', 'nsecs', 'toff'
              (can be vectors for sum of sine waves)
      'A'    - Amplitude of the wave
      'f'    - Frequency of the wave
      'phi'   - Phase of the wave
      'nsecs' - Number of seconds (in seconds)
      'toff'  - Offset of the wave (in seconds)
'noise'     - 'type' (can be 'Normal' or 'Uniform')
              'sigma' specify the standard deviation
'chirp'     - 'f0', 'f1', 't1' (help chirp)
'gaussian pulse' - 'f0', 'bw' (help gausspuls)
'square wave' - 'f', 'duty' (help square)
'sawtooth'   - 'f', 'width' (help sawtooth)

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.
```


Basic math with AOs

Most of the basic math operations supported by MATLAB have been implemented as AO class methods. For example, suppose you want to add two AOs together, then you can do

```
a = ao(1)
b = ao(2)
c = a+b
plot(c.hist)
```

Note: the units of the two AOs for addition and subtraction must be the same. You can't add apples to oranges, but you can add dimensionless (empty units) to oranges.

Some of the standard operators can act as modifiers. For example, if you want to square an AO:

```
a = ao(2)
a.^2
```

will do the job.

The operators follow MATLAB rules whenever possible. So if you want to add a single AO to a vector of AOs, you can. However, if you want to add two vectors of AOs together, the two vectors must contain the same number of AOs. For example,

```
a = [ao(1) ao(2) ao(3)]
b = ao(4)
c = a + b
```

will work fine and result in `b` being added to each element of `a`. However,

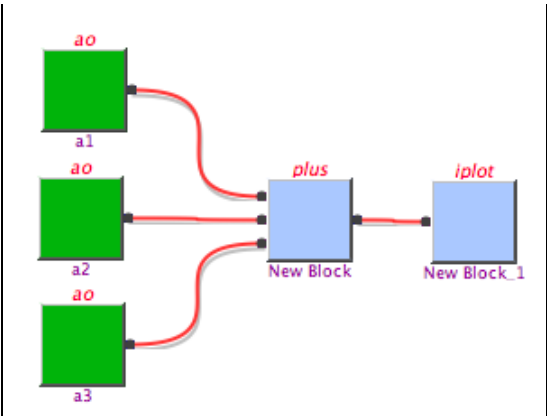
```
a = [ao(1) ao(2) ao(3)]
b = [ao(4) ao(5)]
c = a + b
```

will give an error.

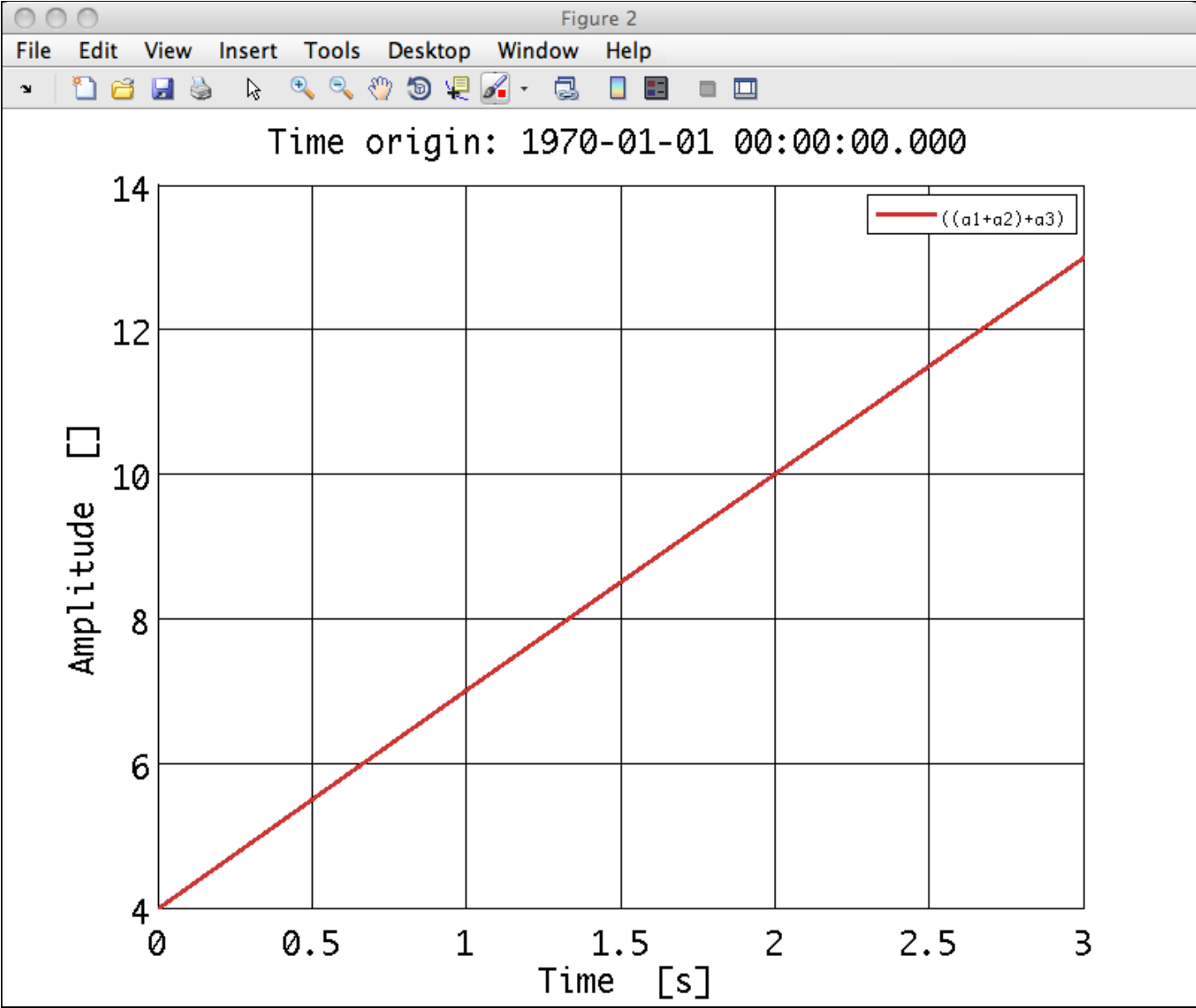
You can do all of this on the workbench as well, of course.

Try the following:

1. Start up the workbench, and/or open a new pipeline
2. Drag an `ao` constructor block to the canvas
3. Set the AO block to be constructed "From XY Values"
4. Duplicate the block two more times (`ctrl/cmd-d`)
5. Enter a number for the sampling frequency `fs`
6. Enter a vector the same length for the `y` value of each block
7. Drag a `plus` block to the canvas
8. You'll see that by default the "plus" block has two inputs. To add another input, right-click on the block and choose "Add input" from the context menu
9. Connect each AO block to the plus block. The easiest way to do that is to select the AO block (source) and then `ctrl+left-click` (`cmd+left-click` on Mac systems) on the plus (destination) block. You can also drag a pipe from the output terminal of the AO blocks to the input terminals of the plus block if you want to be explicit about which input ports are used.
10. Add an `iplot` block to the canvas and connect the output of the `plus` block to the input of the `iplot` block. You should now have a pipeline something like:



11. Execute the pipeline and you should see a plot something like the one below, depending on what data values you gave to the ao constructors



Saving and loading AOs

Having made all these nice AOs, you will be keen to save them to disk. You'll be delighted to hear that this is easy.

To file formats are supported by LTPDA: the MATLAB binary MAT format, and an XML file format. The choice is made by the file extension.

```
save(a1, 'foo.xml');
save(a1, 'foo.mat');
```

To load the files again, you use the AO constructor:

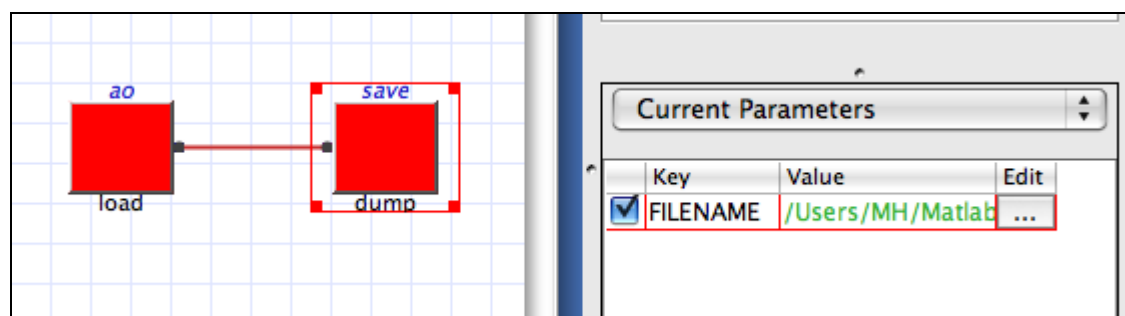
```
a1 = ao('foo.xml');
a2 = ao('foo.mat');
```

That's all there is to it.

To load files on the workbench, you use the relevant constructor block. For example, to load an AO XML file from disk, use the `ao` constructor with the parameter set "From XML File" (or "From MAT File" to load from a MAT file). This parameter set has one default parameter: `FILENAME`. If you double click the cell corresponding to the parameter value you will be presented with a 'load file' dialog from which you can choose the file to load.

To save files from within the workbench, use the `save` block. The parameter list and key is the same. The only difference will be you will be presented with a 'save file' dialog box when editing the parameter value.

The following pipeline shows the load and save in action:



Constructing AOs from data files

You can build AOs from existing ASCII data files. Various formats are supported, for example, multiple columns, files containing comments.

Install the data pack

The data-pack for this training session should be downloaded from the [LTPDA web-site](#). The zip file will expand to a top-level directory. This should contain sub-directories for each topic of the training session.

The rest of the tutorial will assume that you have changed directories in MATLAB to the data-pack directory so that filenames are relative to that directory. To change the working directory of MATLAB, either use the MATLAB interface or type

```
cd /path/to/my/data/pack
```

Create an AO from a simple ASCII file

The data-pack contains a simple two-column text file which represents a time-series sampled at 10Hz. The first column contains the time-stamps, the second column the amplitude values.

To convert this data file to an AO, use the following command:

```
>> a = ao('topic1/simpleASCII.txt')
M: running ao/ao
M:   load file: simpleASCII.txt
M:   constructing from filename and/or plist
M: running ao/ao
M:   constructing from data object tsdata
M: running ao/ao
M: running ao/display
----- ao 01: topic1/simpleASCII.txt_01_02 -----
      name:  topic1/simpleASCII.txt_01_02
      data: (0,1.13549060238654) (0.1,2.37202031808076) (0.2,-0.531181753855465)
(0.3,1.25684477541224) (0.4,1.30895517480354) ...
      ----- tsdata 01 -----
            fs: 10
            x: [1000 1], double
            y: [1000 1], double
            dx: [0 0], double
            dy: [0 0], double
      xunits: [s]
      yunits: []
      nsecs: 100
      t0: 1970-01-01 00:00:00.000
      -----
      hist:  ao / ao / SId: fromDatafile.m,v 1.21 2009/08/05 10:35:55 hewitson Exp S-->SId:
ao.m,v 1.284 2009/08/25 15:01:16 mauro Exp S
      mdlfile:  empty
      description:
      UUID:  de31042b-cded-4542-999b-cf010b702355
      -----
```

From the output on the screen you can see that

1. the name of the AO has automatically been set based on the filename and the columns of data loaded
2. the sample rate of the data is 10Hz, as expected
3. the length of the data is 100s

You can plot this data and see that it is just a random noise time-series.

Create an AO from a multi-column ASCII file

The data-pack contains a data file which contains multiple columns of data. Here we will load only selected columns from the file and produce multiple AOs, one for each column loaded. Column 1 of the file contains the time-stamps; columns 2-6 contain sine waves at frequencies 1-5Hz.

Let's load the 2Hz and 4Hz sine waves from the file. At the same time, we'll give the AOs names, Y units, and descriptions.

```
sigs = ao(plist('filename', 'topic1/multicolumnASCII.txt', ...
'columns', [1 3 1 5], ...
'name', {'sin2', 'sin4'}, ...
'yunits', {'m', 'm'}, ...
'description', {'sine wave at 2Hz', 'sine wave at 4Hz'}))
```

You can plot the results and focus on the first 2 seconds of data

```
sigs.ipplot(plist('XRanges', [0 2]))
```

◀ Saving and loading AOs

Writing LTPDA scripts ▶

©LTP Team

Writing LTPDA scripts

Up to now, all the activity of the tutorial has been carried out on the MATLAB command terminal. It is, of course, much more convenient to collect the commands together in to a MATLAB script. In this sense, LTPDA scripting is just the same as normal MATLAB scripting; just using LTPDA commands.

There are, however, one or two caveats to that, especially when it comes to preserving the history. Most notably, indexing and concatenating objects is not captured by the history if you use standard MATLAB notation.

For example, suppose you have two AOs which you want to make a Transfer function between them. The following script shows three different ways to do this, but only two will properly track the history.

```
%% Make two test AOs
a1 = ao(plist('tsfcn', 'randn(size(t))', 'fs', 10, 'nsecs', 100));
a2 = ao(plist('tsfcn', 'randn(size(t))', 'fs', 10, 'nsecs', 100));

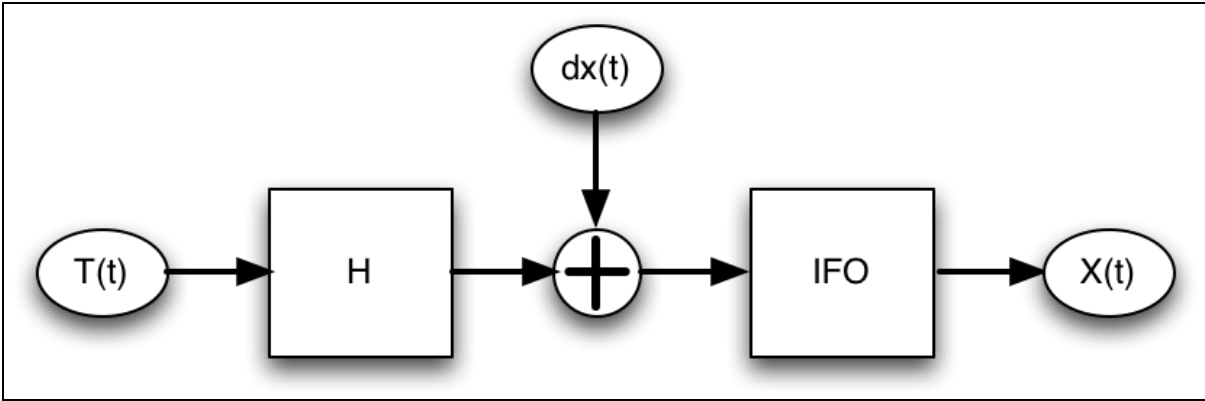
%% Make TFE
t11 = tfe(a1, a1);
t12 = tfe(a1, a2);
t21 = tfe(a2, a1);
t22 = tfe(a2, a2);
Axx = t12 ./ t21;
```

You can explore the commands used to rebuild the object by using the `type` method. Or you can plot the history as we did earlier. For example, try this command for each of the three cases above:

```
type(Axx)
```


IFO/Temperature Example – Introduction

In your data pack, you will find two raw data series. These are real measurements of a system that looks like



We have measured the two signals, $\tau(t)$ and $x(t)$. The displacement input to the interferometer, $dx(t)$, is inaccessible to us, and is in fact the data we want to recover from the data analysis.

Reading and calibrating the interferometer data

The interferometer data, $X(t)$, is saved in the file `ifo_temp_example/ifo_training.dat`. This is a two-column ASCII file with the first column giving the time-stamps of the data and the second column the measured IFO output in radians.

To read in the data, we can use the AO constructor, with the set of parameters "From ASCII File". The key parameters are:

Key	Value	Description
FILENAME	'ifo_temp_example/ifo_training.dat'	The name of the file to read the data from.
TYPE	'tsdata'	Interpret the data in the file as time-series data.
COLUMNS	[1 2]	Load the data x-y pairs from columns 1 (as x) and 2 (as y).
XUNITS	's'	Set the units of the x-data to seconds (s).
YUNITS	'rad'	Set the units of the y-data to radians (rad).
ROBUST	'no'	Use fast data reading for

DESCRIPTION **'Interferometer data'**

this simple file format.

Set some text to the 'description' field of the AO.

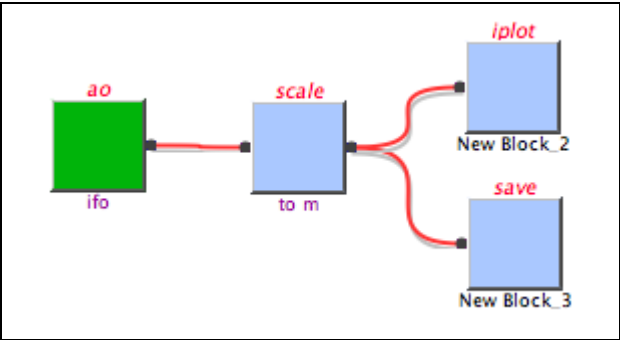
Once we've loaded the data we can calibrate it to displacement using the following equation:

$$Y(t)[m] = X(t) \frac{\lambda}{2\pi} [rad]$$

where lambda=1064nm.

To do the calibration, you can use the method `ao/scale`, specifying the `factor` and the new `yunits`. Then save the resulting time-series to disk in `ifo_temp_example/ifo_disp.xml`.

A finished pipeline might look something like:



Reading and calibrating the temperature data

The temperature data, $T(t)$, is saved in the file `ifo_temp_example/temp_training.dat`. Again, this is a two-column ASCII file; the first column contains the time-stamps of the data, the second column contains the temperature values in degrees Celsius.

To read in the data, we can use the AO constructor, with the set of parameters "From ASCII File". The key parameters are:

Key	Value	Description
FILENAME	'ifo_temp_example/temp_training.dat'	The name of the file to read the data from.
TYPE	'tsdata'	Interpret the data in the file as time-series data.
COLUMNS	[1 2]	Load the data x-y pairs from columns 1 (as x) and 2 (as y).
XUNITS	's'	Set the units of the x-data to seconds (s).
YUNITS	'degC'	Set the units of the y-data to degrees Celcius.

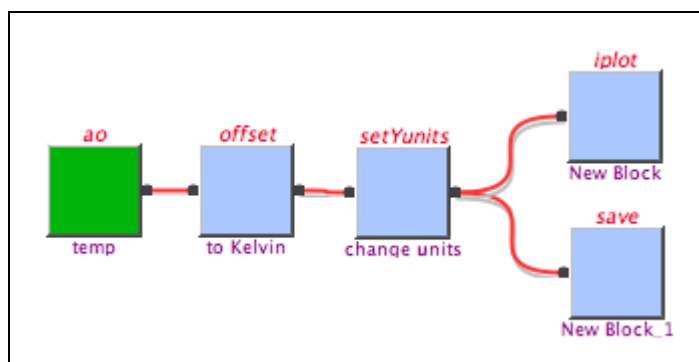
ROBUST	'no'	Use fast data reading for this simple file format.
DESCRIPTION	'Temperature data'	Set some text to the 'description' field of the AO.

Having loaded the temperature data, we can proceed to calibrate it to Kelvin by adding 273.15 to the data and then change the y-units. You can do this using the two AO methods:

- `offset --` to add the offset
- `setYunits --` to change the y-units of the data

The final step is to save this calibrated temperature data to disk as an AO XML file called `ifo_temp_example/temp_kelvin.xml`, ready for input to the next topic.

A complete pipeline for this step might look like:



To plot the data with hours on the x-axis instead of seconds, use the `iplot` parameter `XUNITS`. For example:

```
iplot(a, plist('XUNITS', 'h'))
```

Topic 2 – Pre-processing of data

In the signal pre-processing session we will learn about a variety of functions with which we can process our data prior to further analysis. Below are the functions we will learn about in this topic.

- [Topic 2.1 – Downsampling a time-series AO](#)
- [Topic 2.2 – Upsampling a time-series AO](#)
- [Topic 2.3 – Resampling a time-series AO](#)
- [Topic 2.4 – Interpolation of a time-series AO](#)
- [Topic 2.5 – Remove trends from a time-series AO](#)
- [Topic 2.6 – Whitening data](#)
- [Topic 2.7 – Select and find data from an AO](#)
- [Topic 2.8 – Split and join AOs](#)
- [Topic 2.9 – Pre-processing the IFO/Temperature Example](#)

◀ IFO/Temperature Example – Introduction

Downsampling a time-series AO ▶

©LTP Team

Downsampling a time-series AO

Downsampling reduces the sampling rate of the input AOs by an integer factor, which can be very useful for example to reduce data load.

The `downsample` method takes the following parameters:

Key	Description
FACTOR	The decimation factor [by default is 1: no downsampling] (must be an integer)
OFFSET	The sample offset for where the downsampling starts counting. By default, this value is zero so it starts counting from the first sample.

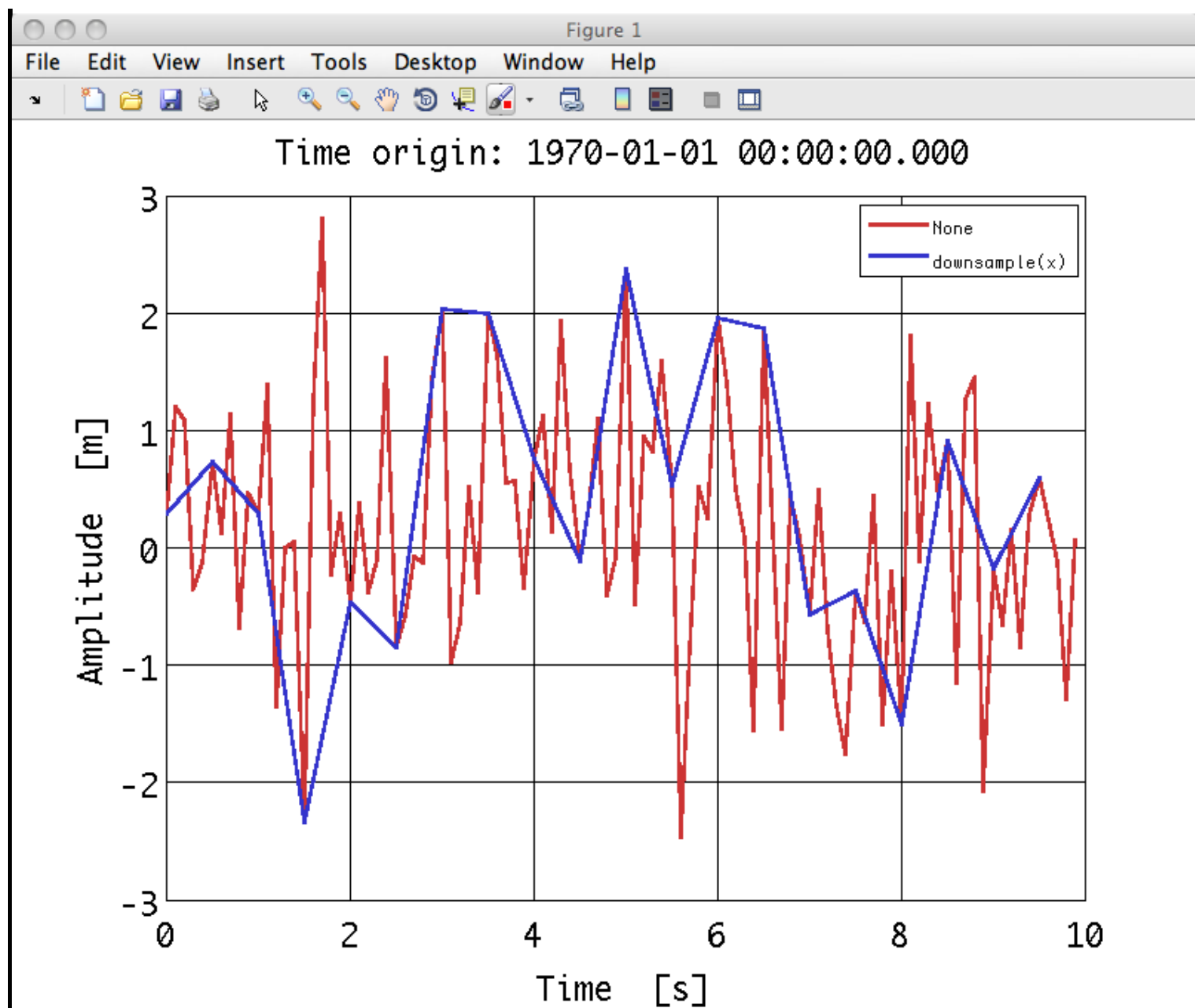
Example 1

First we'll create a time-series of random white noise at 10Hz. To do that, define a `plist` with the key/value pairs shown below, and pass these to the `ao` constructor. If you're using the workbench, add an `ao` constructor block to the canvas and select the parameter set "From Time-series Function" and set the parameters as they are in the `plist` below.

```
% create an AO of random data with fs = 10 Hz;
pl      = plist('tsfcn', 'randn(size(t))', ...
               'fs', 10, 'yunits', 'm');
x       = ao(pl); % create AO
```

Now we will downsample this data by a factor 5 to 2Hz. We use the method `ao/downsample` and set the downsample factor in the `plist` as shown below.

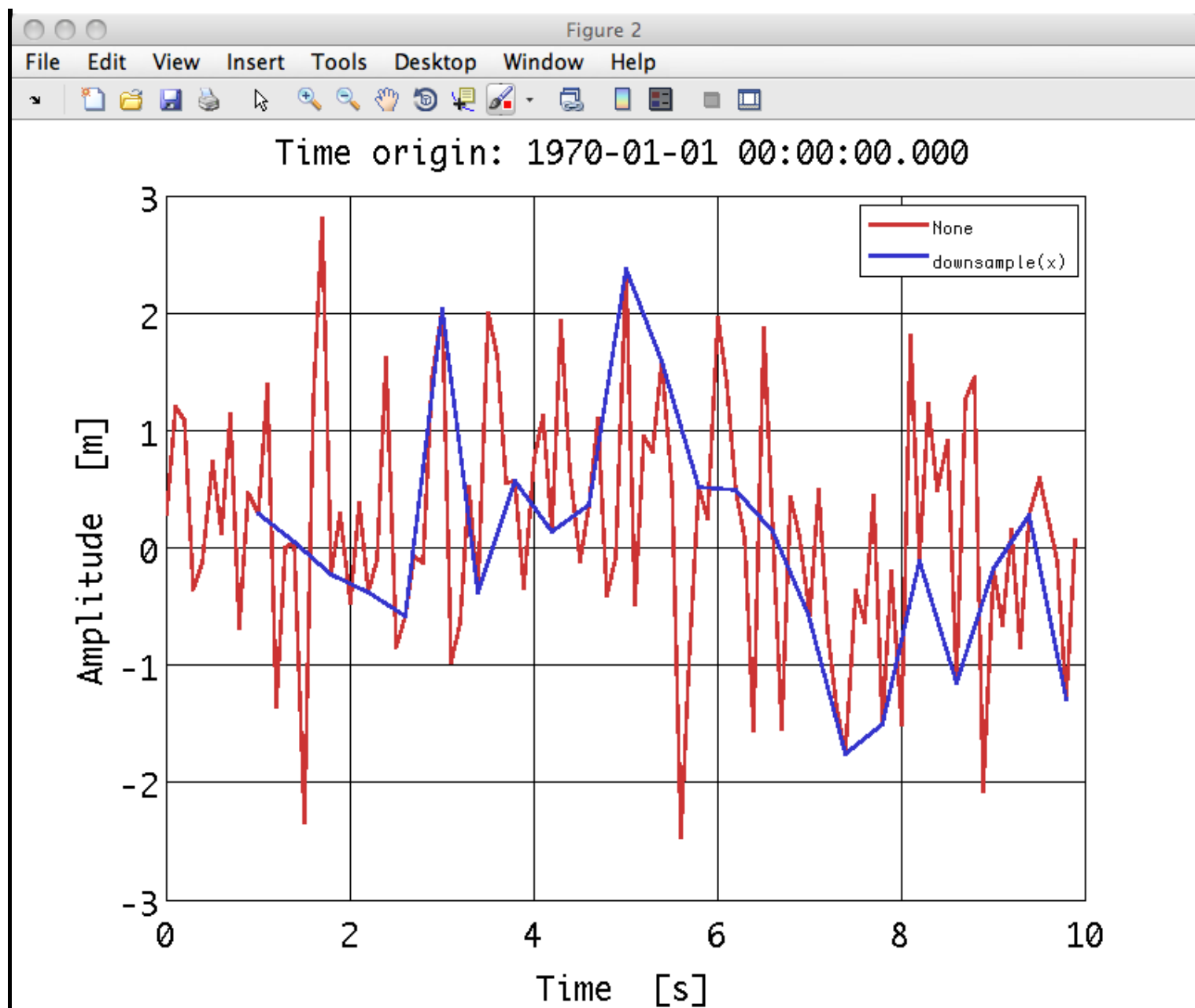
```
pl_down = plist('factor', 5); % add the decimation factor
x_down  = downsample(x, pl_down); % downsample the input AO, x
ipplot(x, x_down) % plot original, x, and decimated, x_down, AOs
```



Example 2

The `downsample` method takes an 'offset' key which controls where the counting starts. The default is to output every Nth sample starting with the first. The 'offset' key tells `downsample` to start the counting from a sample other than the first. The example below outputs every 4th sample starting from sample 10.

```
pl_downoff = plist('factor', 4, 'offset', 10); % add decimation factor and offset parameter
x_downoff = downsample(x, pl_downoff); % downsample the input AO, x
ipplot(x, x_downoff) % plot original, x, and decimated, x_downoff, AOs
```



Upsampling a time-series AO

Upsampling increases the sampling rate of the input AOs by an integer factor

The `ao/upsample` method can take the following parameters:

Key	Description
N	The upsample factor. The algorithm places 'N-1' zeros between each of the original samples.
PHASE	This parameter specifies an additional sample offset. The value must be between 0 and N-1.

Example 1

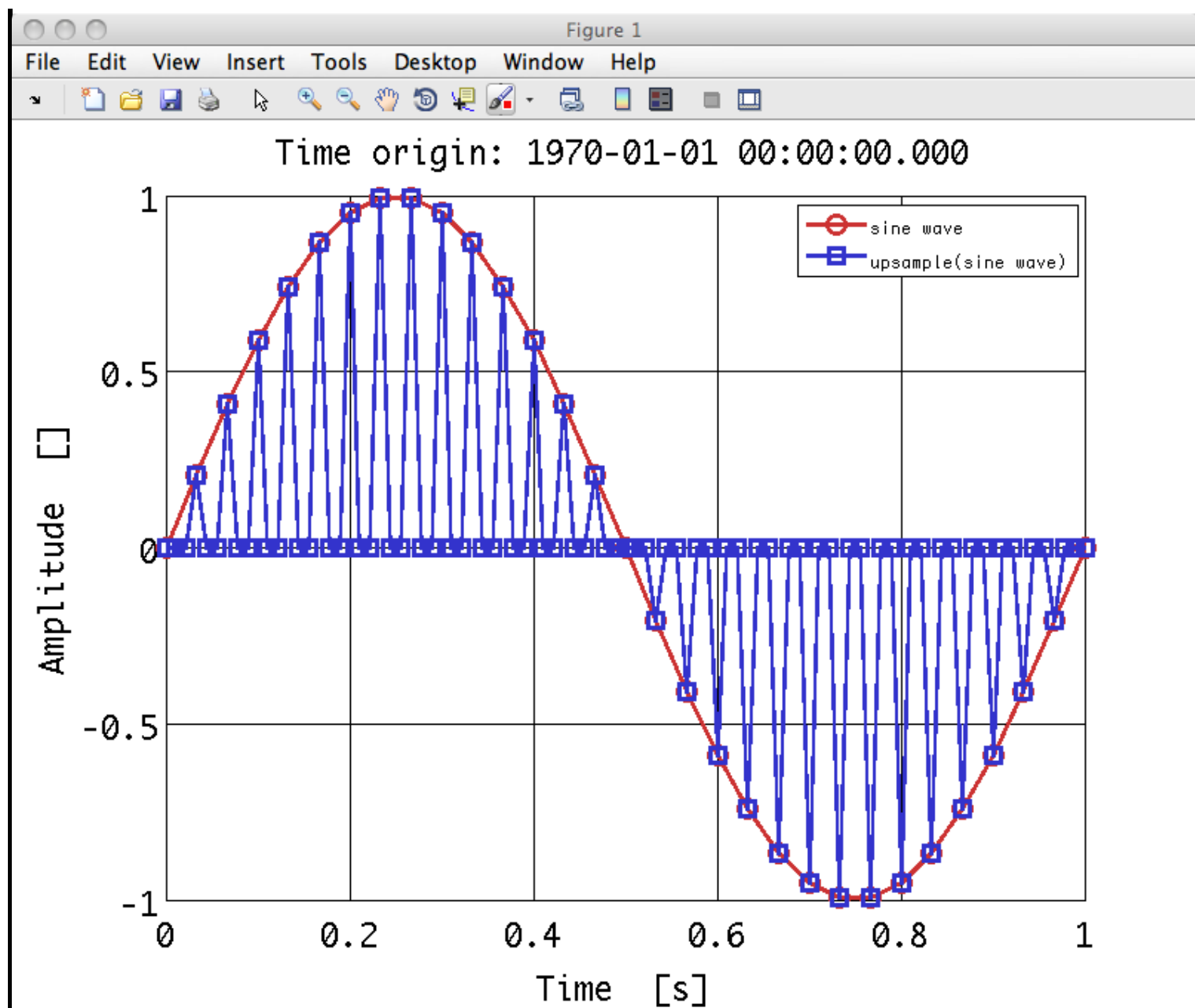
We will upsample a sine-wave by a factor of 3 with no initial phase offset.

Start by creating a sine-wave at 1Hz with a 30Hz sample rate and 10 seconds long. We can use the `ao` "From Waveform" parameter set to do this. (Equally, we can do this with the "From Time-series Function" parameter set.)

```
pl = plist('Waveform', 'sine wave', 'f', 1, 'fs', 30, 'nsecs', 10)
x = ao(pl);
```

Now we can proceed to upsample this data by a factor 3. This will place 2 zero samples between each of the original samples.

```
pl_up = plist('N', 3); % increase the sampling frequency by a factor of 10
x_up = upsample(x, pl_up); % resample the input AO (x) to obtain the upsampled AO (y)
data iplot(x, x_up, plist('XRanges', [0 1], 'Markers', {'o', 's'})) % plot original and upsampled
```

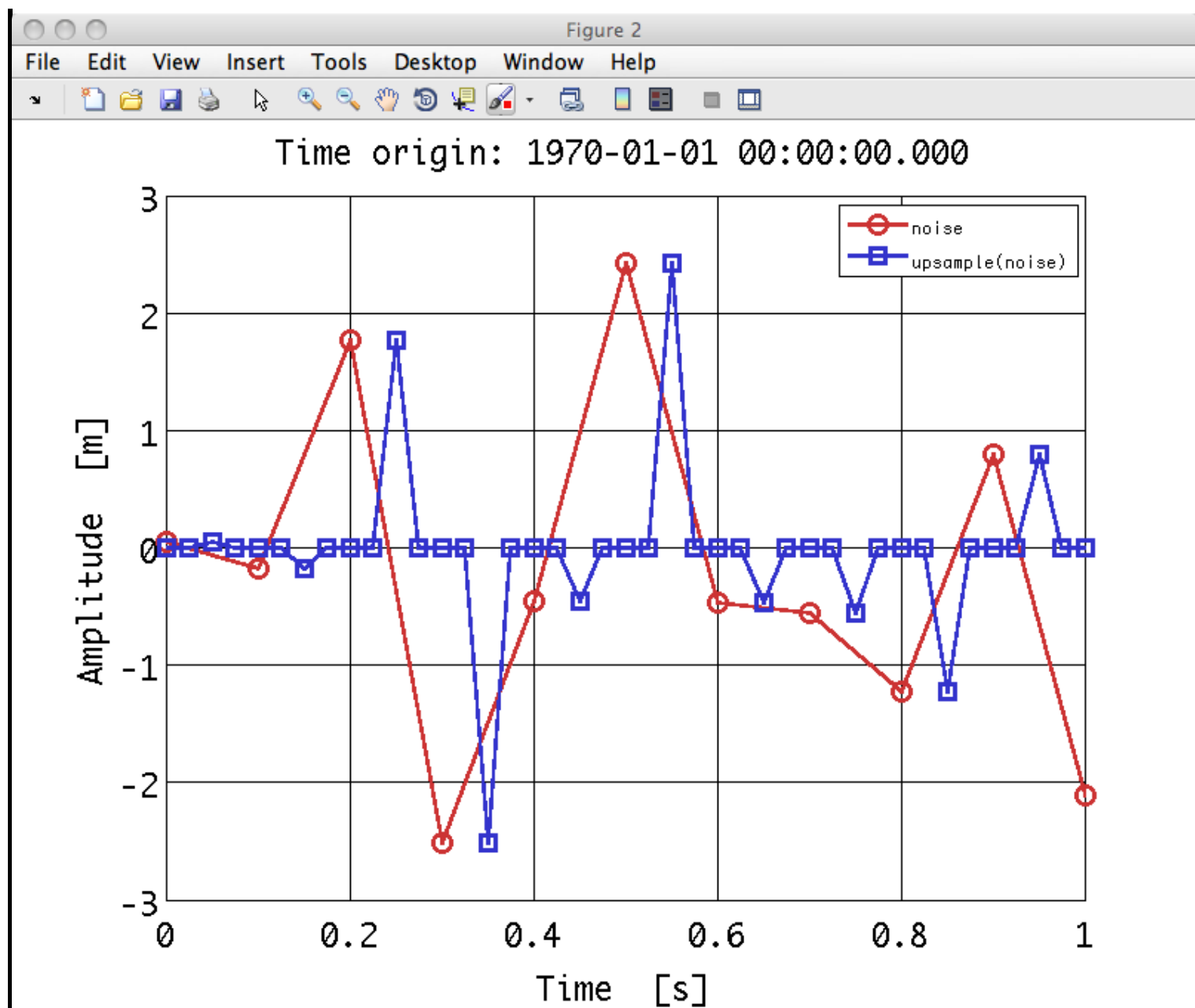


Example 2

In this second example, we will upsample some random noise by a factor 4 with a phase offset of 2 samples.

Again, start by constructing some test data, in this case a white-noise data stream. We can do this again using the "From Waveform" parameter set with an ao constructor.

```
pl = plist('Waveform', 'noise', 'fs', 10, 'nsecs', 10, 'yunits', 'm')
x = ao(pl);
pl_upphase = plist('N', 4, 'phase', 2); % increase the sampling frequency and add phase of 2
samples to the upsampled data
x_upphase = upsample(x, pl_upphase); % resample the input AO (x) to obtain the upsampled and
delayed AO
iplot(x, x_upphase, plist('XRanges', [0 1], 'Markers', {'o', 's'})) % plot original and upsampled
data
```

◀ Downsampling a time-series AO

Resampling a time-series AO ▶

©LTP Team

Resampling a time-series AO

Resampling is the process of changing the sampling rate of data. The method `ao/resample` changes the sampling rate of the input AOs to the desired output sampling frequency by performing band-limited interpolation, or interpolation.

If the ratio of the input and the desired output sample rate can be expressed as an integer ratio P/Q where both P and Q then band-limited interpolation can be performed. In this case, the data-series is upsampled (by inserting zeros) and then a low-pass filter is applied at the original Nyquist frequency.

The `ao/resample` method can be called with the syntax:

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

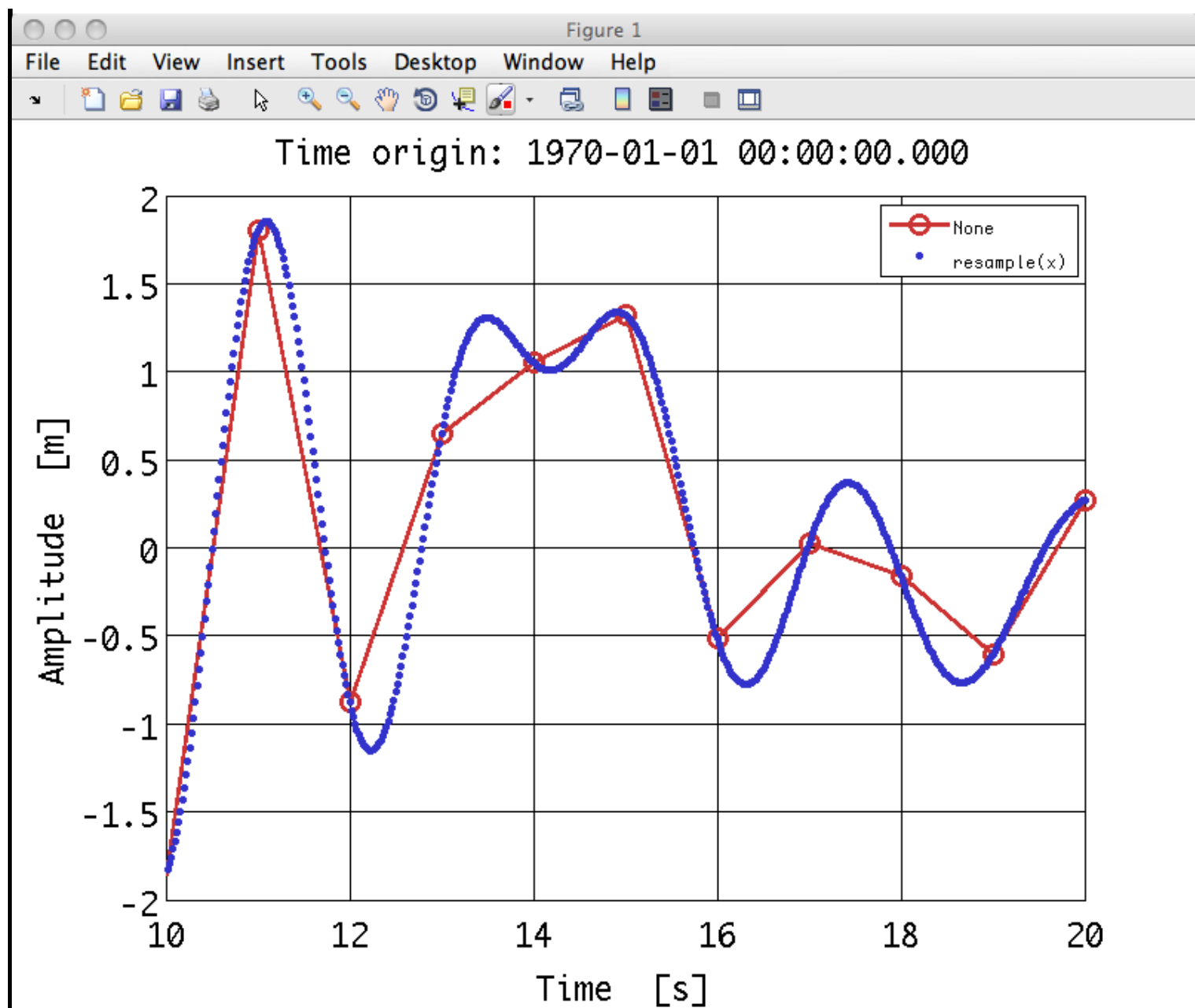
and can accept the following parameter:

Key	Description
FSOUT	The desired output sample.

Example 1

Here we will resample a sequence of random data from the original sampling rate of 1 Hz to an output sampling of 50 Hz.

```
pl    = plist('tsfcn', 'randn(size(t))','nsecs',100,'fs',1,'yunits','m');
x     = ao(pl)
pl_re = plist('fsout', 50);
x_re  = resample(x, pl_re); % resample the input AO (x) to obtain the resampled output AO (y)
ipplot(x, x_re, plist('XRanges', [10 20], '...',
                      'Markers', {'o', '.'},
                      'LineStyles', {'-', 'none'})) % plot original and resampled data
```



◀ Upsampling a time-series AO

Interpolation of a time-series AO ▶

©LTP Team

Interpolation of a time-series AO

The `ao` class has a method for interpolating data using different forms of interpolation. This method is called `ao/interp`.

To configure `ao/interp`, use the following parameters:

Key	Description
VERTICES	A new set of vertices (relative to the <code>t0</code>) on which to resample.
METHOD	The method by which to interpolate. Choose from • 'nearest' – nearest neighbour • 'linear' – linear interpolation • 'spline' – for spline interpolation • 'cubic' – for cubic interpolation

Example

Here we will interpolate a sinusoid signal on to a new time-grid. The result will be to increase the sample rate by a factor 2.

First we create a time-series `ao`:

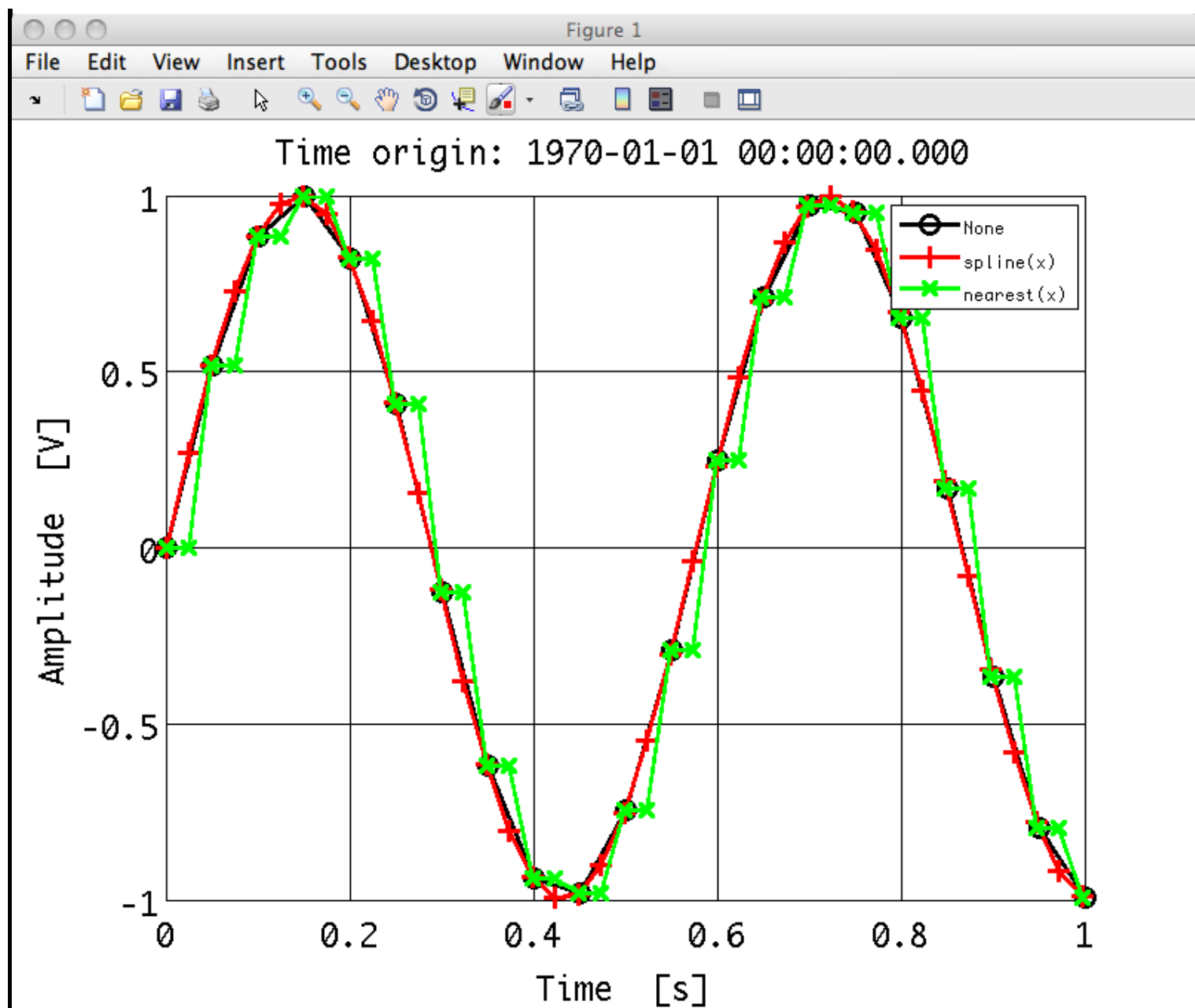
```
pl = plist('tsfcn', 'sin(2*pi*1.733*t)', 'fs', 20, 'nsecs', 10, 'units', 'V');
x = ao(pl);
```

Then we create the new time-grid we want to resample on to.

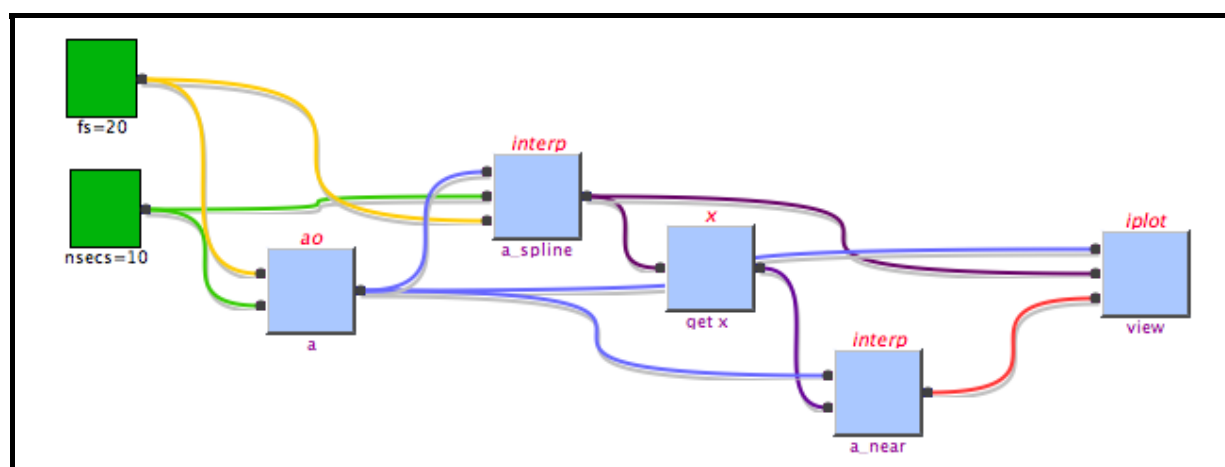
```
t = linspace(0, x.nsecs - 1/x.fs, 2*(x.len));
```

And finally we can apply our new time-grid to the data using `interp`. We test two of the available interpolation methods:

```
pl_spline = plist('vertices', t);
pl_nearest = plist('vertices', t, 'method', 'nearest');
x_spline = interp(x, pl_spline);
x_nearest = interp(x, pl_nearest);
ipplot(x, x_spline, x_nearest, plist('Markers', {'o', '+', 'x'}, ...
  'LineColors', {'k', 'r', 'g'}, ...
  'XRanges', [0 1]));
```



To do the same activity on the workbench, we can use a pipeline like:



This teaches some important aspects of the use of the workbench, so it's worth stepping through its construction slowly.

To build this pipeline:

1. Create a new empty canvas
2. Add two MATLAB Expression Blocks:
 1. Right-click on the canvas and choose "Additional Blocks->MATBlock". A `MATBlock` is a block which can evaluate any valid MATLAB expression and pass that to further blocks via its single output.
 2. Select the block then change its name in the block property table (top right) to 'fs'. Alternatively, the name of the block can be changed by right-clicking on the block and choosing "Set name"
 3. Double-click the block to get a pop-up dialog where you can enter the MATLAB expression. In this case just enter the value 20.
 4. Select this 'fs' block and hit `ctrl-d` (`cmd-d` on OS X) to duplicate the block.
 5. Select the new block and change its name to 'nsecs'
 6. Double-click the 'nsecs' block to change its expression. Enter the value 10.
3. Next we need some additional blocks. Add an `ao` block, two `ao/interp` blocks, an `ao/x` block, and an `ao/iplot` block.
4. Connect up the blocks as shown on the pipeline above.

To add inputs (or outputs) to a block, right-click on the block and choose "Add input".

Double-click an LTPDA Block to get a dialog box to enter a new name for the block.

To set the color of the pipes emanating from a particular block, right-click on the block and choose "Set output pipe color" from the context menu. You can also set the color of individual pipes by right-clicking on a pipe and choosing "Set color" from the context menu.

5. Next we need to set the various properties of each block. Follow these steps:
 1. Set the properties of the AO block to look like:

Key	Value
TSFCN	'sin(2*pi*1.733*t)'
FS	PORT_0
NSECS	PORT_1
T0	1970-01-01 00:00:00.000
XUNITS	s
YUNITS	V

The single quotes around the `TSFCN` value are not strictly necessary, but it can avoid problems, for example in the case you have a variable `t` already defined in the MATLAB workspace.

2. Set the properties of the first interpolate block (`a_spline`) to look like:

Key	Value
VERTICES	linspace(0, PORT_1-1/PORT_2, 2*PORT_1*PORT_2)
METHOD	spline

Notice that here we have used the keywords `PORT_1` and `PORT_2` to build the expression. These refer to the ports of that block, and are connected to the MATLAB Expression Blocks which represent the values we are interested in.

3. The block `ao/x` has no properties and simply gets the full `x`-vector from the output of `a_spline`. This is then passed to the next interpolation block where we use the values as the vertices for the next interpolation step, thus ensuring that the two interpolations are done on the same grid.
4. Set the properties of the second interpolate block (`a_near`) to look like:

Key	Value
VERTICES	PORT_1
METHOD	nearest

5. Finally, for the `iplot` block, add three new parameters and give them key names and values like:

Key	Value
XRanges	[0 1]
Markers	{'o', '+', 'x'}
LineColors	{'r', 'k', 'g'}

It should now be possible to run this pipeline and see a plot very similar to the one produced above.

◀ Resampling a time-series AO

Remove trends from a time-series AO ▶

Remove trends from a time-series AO

The `ao/detrend` method offers the possibility to remove polynomial trends from a data series.

The method can be configured with the following parameter:

Key	Description
N	The order of the polynomial to fit and remove. For orders below 10, a very fast C-code algorithm is used. For higher orders, the MATLAB functions <code>polyfit</code> and <code>polyval</code> are used to construct the polynomial which is then subtracted from the data.

Example 1

In this example we will construct a time-series consisting of noise plus a known quadratic trend. We will then remove that trend using `ao/detrend` and compare the detrended time-series with the original noise.

First let's create the time-series series consisting of the noise plus trend.

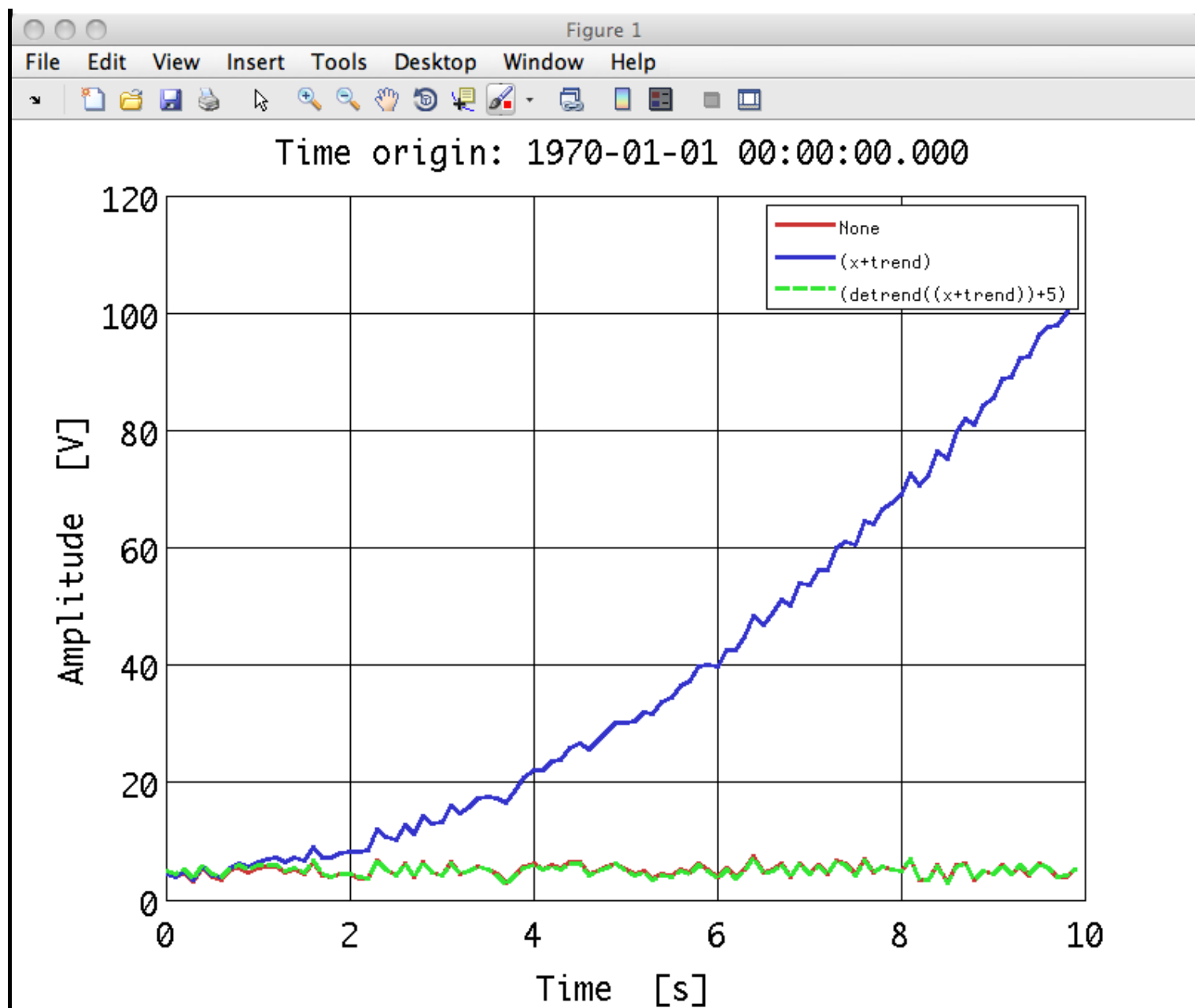
```
% Construct noise data stream
fs = 10;
nsecs = 10;
pl = plist('tsfcn', '5+randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'V');
x = ao(pl);
% Construct a quadratic data series
pl_trend = plist('tsfcn', 't.^2', 'fs', fs, 'nsecs', nsecs);
trend = ao(pl_trend);
% Add them together
fcn_trend = x + trend;
```

The offset of 5 is added to the noise to ensure the data series doesn't come close to zero; we want to divide by it later in the example.

Next we will detrend the data and compare the result to the noise data `x` we made above.

```
pl_detr = plist('N', 2);
detr = detrend(fcn_trend, pl_detr);
ipplot(x, fcn_trend, detr, plist('LineStyle', {'', '', '--'}));
```

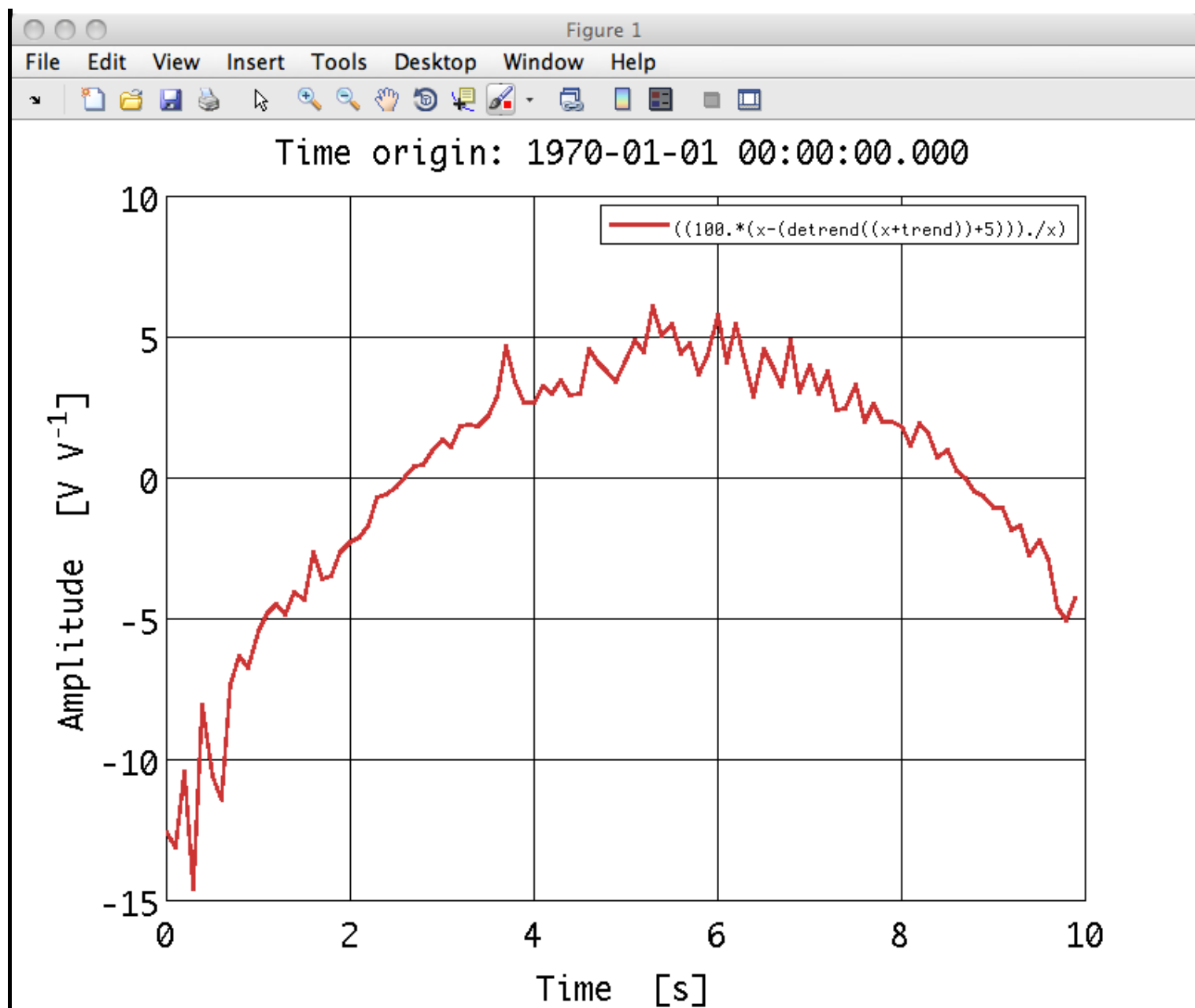
In the `plist` we specified `'LineStyle'` as empty strings. These just serve as place holders and can be interpreted as "just to the default". If you want a data-series plotted with no line, then specify `'none'`, for example, `{'none', '-', '--'}`.



From this plot, it is not very easy to see how well our detrending worked. Let's form the fractional difference of the original x data and the detrended data and plot that instead.

```
detr = detr + 5;
diff = 100.*(x-detr)./x;
iplot(diff);
```

The result is shown below. We added the value 5 to the detrended time-series just to ensure that we don't divide by any values close to zero.

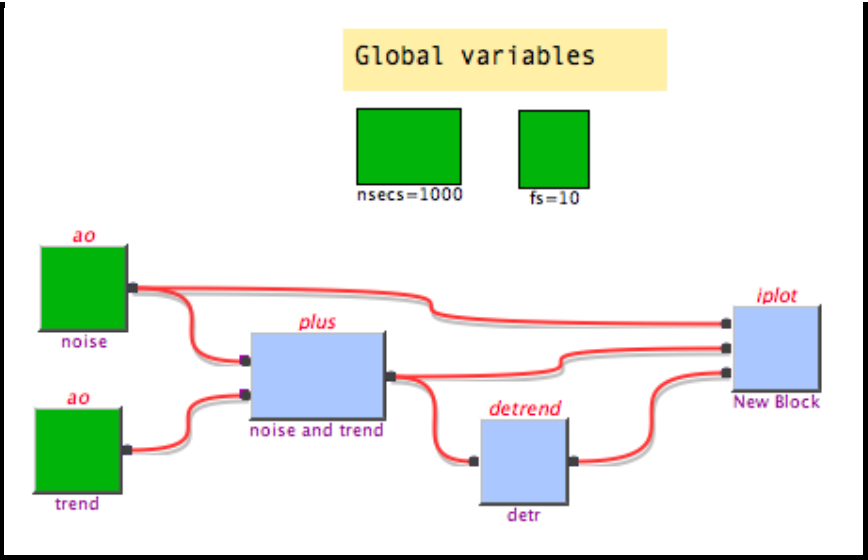


Try increasing the length of the data series to say, 1000 or 10000 seconds, to see how the detrending improves.

The value of the coefficient describing the subtracted trend are included in the field `procinfo` of the `ao` objects. The `procinfo` is actually a `plist` object, so we can search for parameters, in this case the key is `'coeffs'`:

```
c = find(detr.procinfo, 'coeffs');
```

Below is an example pipeline to perform the steps we did above:



This introduces a new concept to the pipelines, namely, the use of constant blocks. Constant blocks are executed before the rest of the pipeline and the values are placed in the MATLAB workspace. This means that all parameter lists on the pipeline can refer to these constants. For example, the pipeline above declares two constants: 'fs' and 'nsecs'. The two ao blocks refer to these. Below is the parameter list for the first ao block, noise.

Key	Value
TSFCN	5+randn(size(t))
FS	fs
NSECS	nsecs
T0	1970-01-01 00:00:00.000
XUNITS	s
YUNITS	V

If you want to change the length of this simulation, then you just need to change the value in the constant block, nsecs.

To add constant blocks to your pipeline, right-click on the canvas and select "Additional Blocks->Constant" from the context menu. You can also add an annotation from the same context menu. The above pipeline shows one annotation. To edit the text on an annotation, double-click it. Right-clicking on an annotation gives a context menu that allows you to configure its appearance.

Whitening noise

The LTPDA toolbox offers various ways in which you could whiten data. Perhaps you know the whitening filter you want to use, in which case you can build the filter and filter the data. Alternatively, you may have a model for the spectral content of the data, in which case you can use the method `ao/whiten1D` if you are dealing with single, uncorrelated data streams, or `ao/whiten2D` if you have a pair of correlated data streams. You can also use `ao/whiten1D` in the case where you don't have a model for the spectral content of the data. In this case, the method calculates the spectrum of the data and fits a model of the spectrum as a series of partial fractions in `s`. to that.

The whitening algorithms are highly configurable and accept a large number of parameters. The main ones that we will change from the defaults in the following examples are

Key	Description
PLOT	Plot the result of the fitting as it proceeds.
MAXORDER	Specify the maximum allowed model order that can be fit.
WEIGHTS	Choose the way the data is weighted in the fitting procedure.
RMSVAR	Check if the variation of the RMS error is smaller than $10^{(-b)}$, where <code>b</code> is the value given in the plist.

We will start by whitening some data using this last method, i.e., allowing `whiten1D` to determine the whitening filter from the data itself.

The data we will whiten can be found in your data packet in the 'topic2' sub-directory.

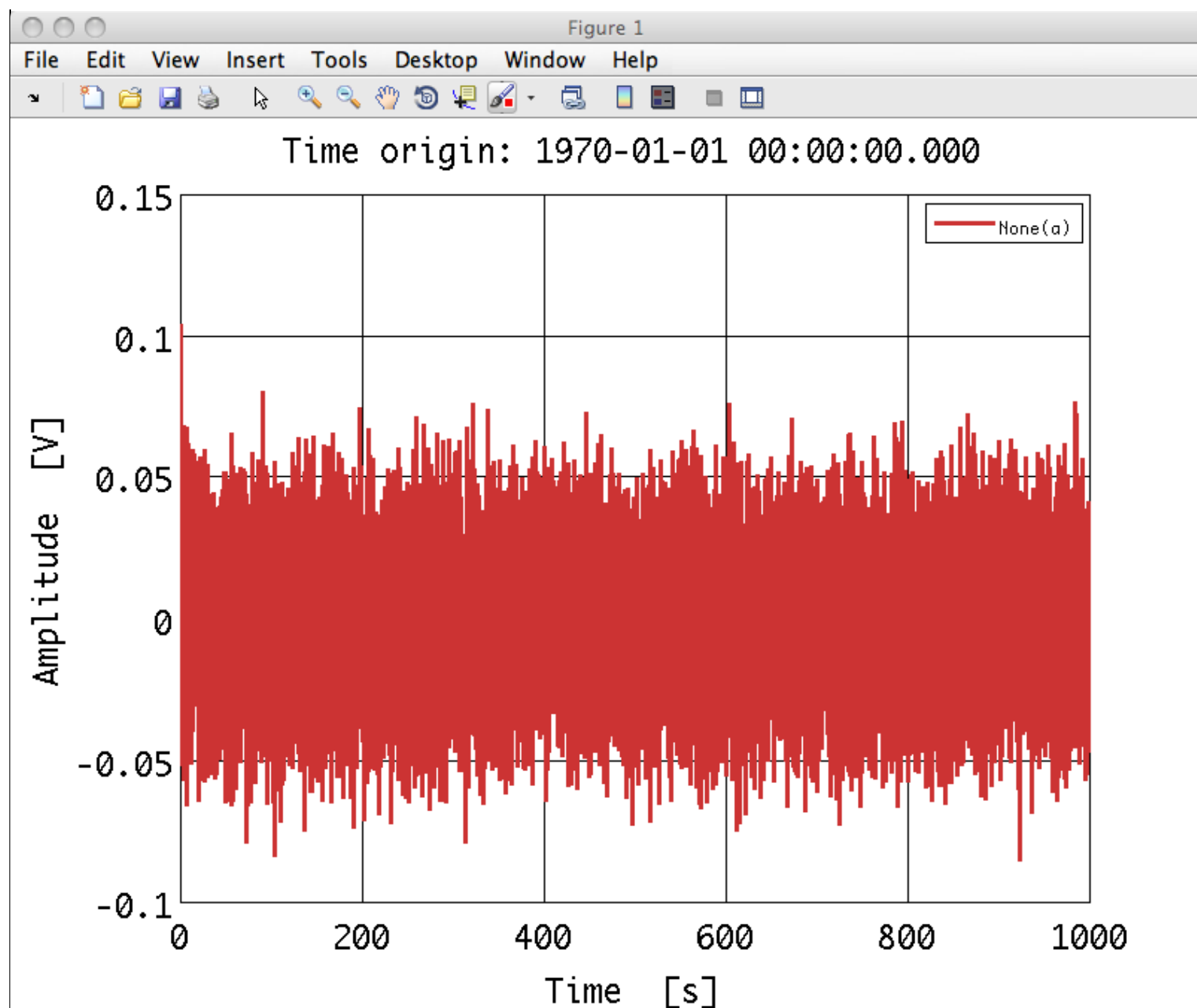
We start by loading the mat file:

```
a = ao('topic2/whiten.mat');
```

The AO stored in the variable `a` is a coloured noise time-series. Let's have a look at this times series using `ipplot`.

```
>> ipplot(a);
```

The result should be similar to:



Before we can whiten the data, we have to define the parameter list for the whitening tool:

```
pl = plist(...
    'Plot', true, ...
    'MaxOrder', 9, ...
    'Weights', 2, ...
    'RMSEVar', 5);
```

Now we can call the whitening function `whiten1D` with our input `AO`, `a` and the parameter list `pl`:

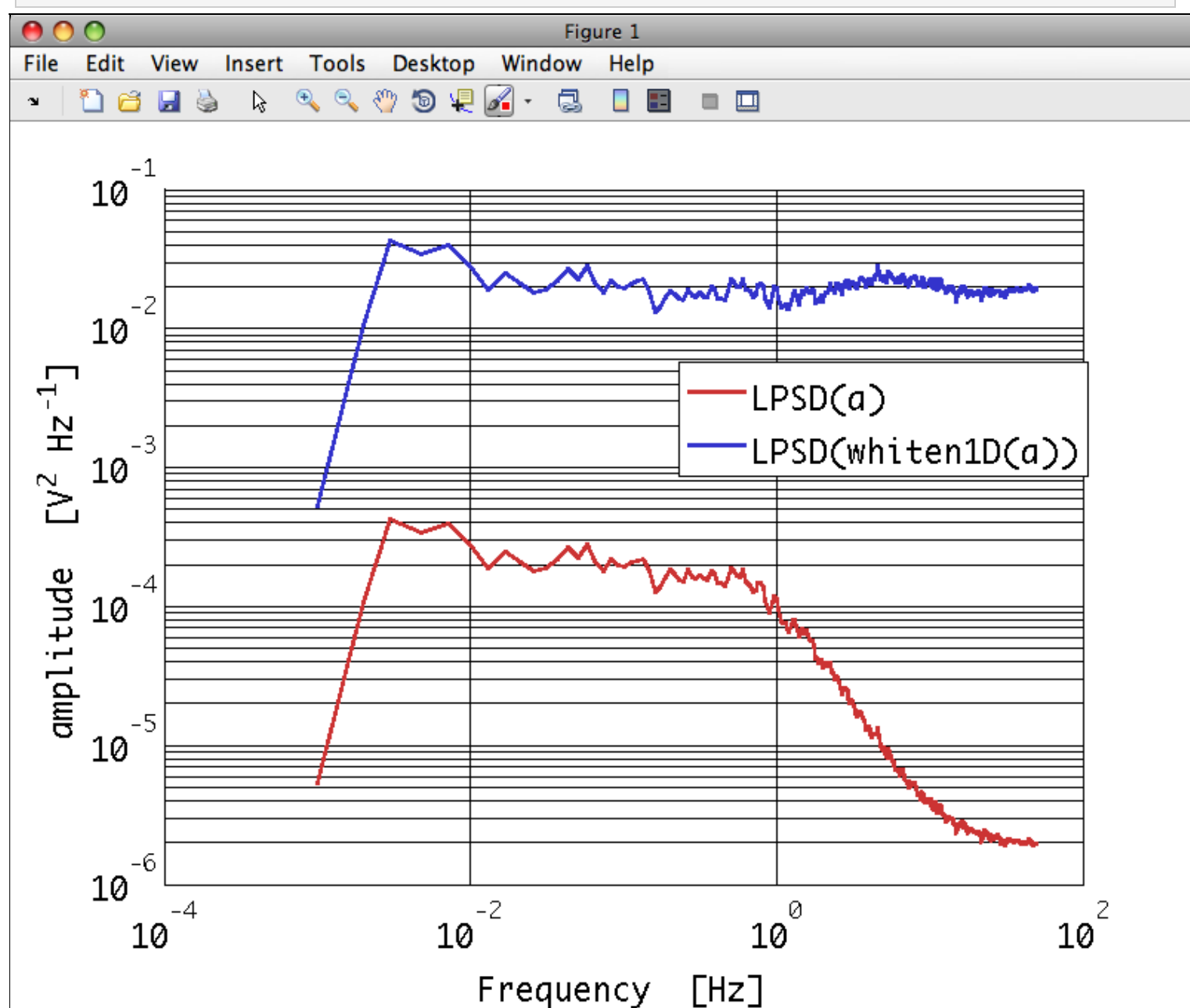
```
>> aw = whiten1D(a,pl);
```

To compare the whitened data with the coloured noise we compute the power spectrum (for details see [Power spectral density estimation](#)):

```
awxx = aw.lpsd;
axx = a.lpsd;
```

and finally plot our result in the frequency domain; in particular we plot the whitened data (`awxx`) compared to the coloured noise that was our input (`axx`).

```
iplot(axx, awxx);
```



◀ Remove trends from a time-series AO

Select and find data from an AO ▶

©LTP Team

Select and find data from an AO

LTPDA contains a set of methods that can be used for the selection of data from an AO. In this section we will look at `ao/find` and `ao/select` in particular.

We will start by generating a sine wave:

```
pl = plist('waveform', 'sine wave', 'f', 1, 'fs', 100, 'nsecs', 10, 'yunits', 'V');  
a = ao(pl)
```

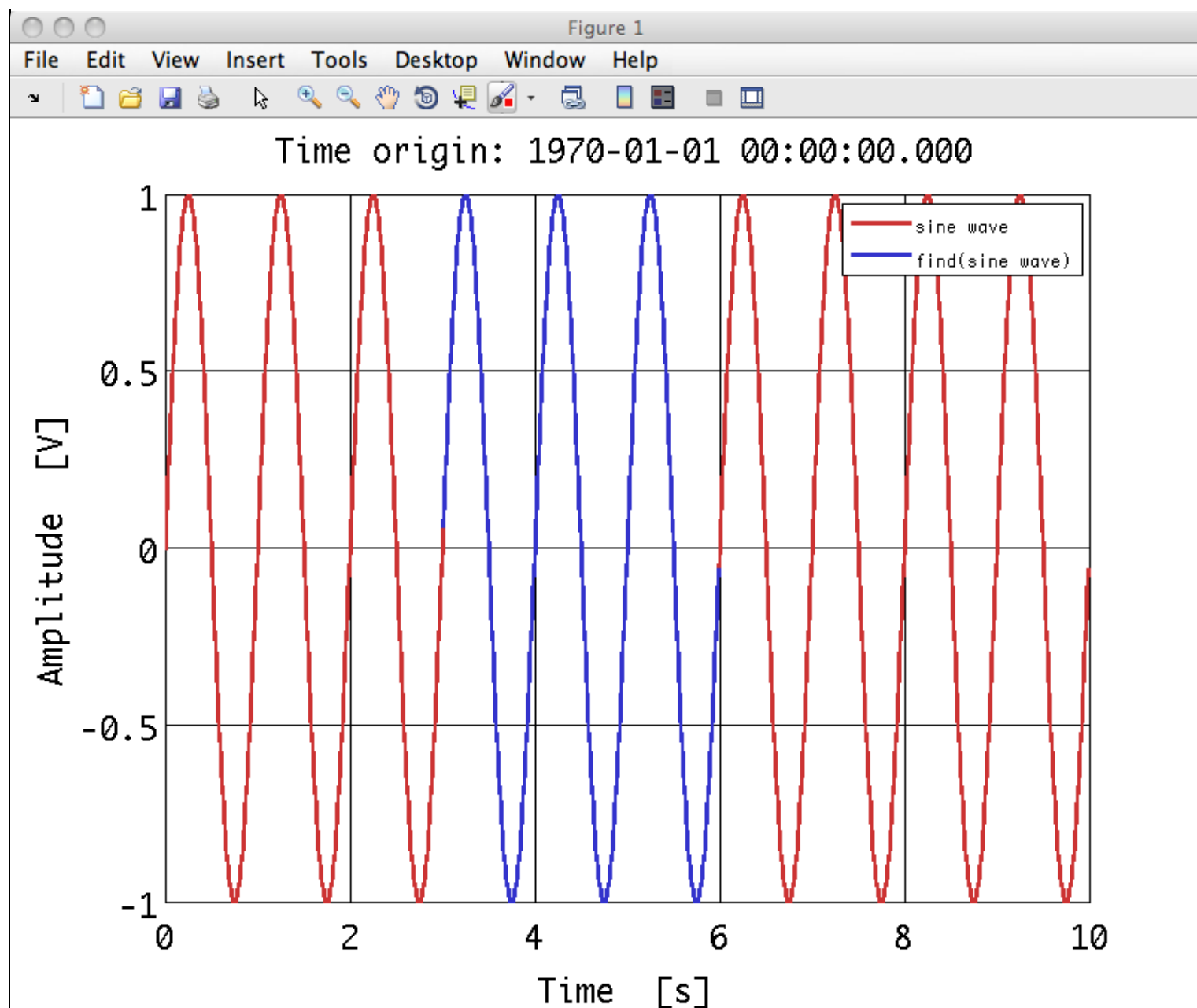
Example 1 – using find

Now let us use the `find` method to extract parts of the data we are interested in.

The `find` method can take a parameter 'query' for defining which data points you want to find. The query string can be any valid MATLAB logical expression, and in particular can be expressed in terms of the x and y data of the input AO.

In this example, we want to find all x values between 3 and 6. The following code does just that:

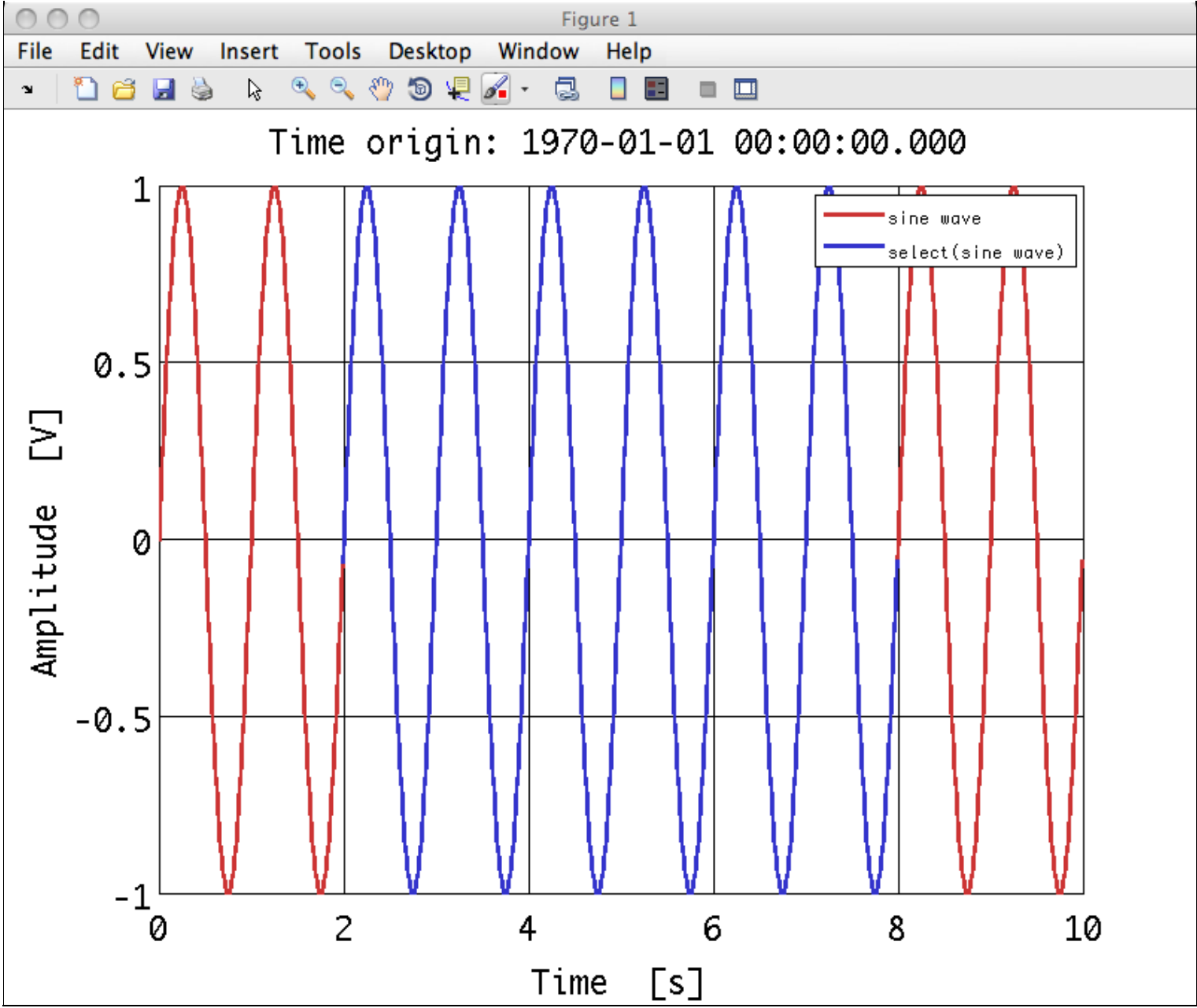
```
a_find = find(a, plist('query', 'x>3 & x<6'));  
ipplot(a, a_find)
```



Example 2 – using select

The `select` method lets us select a set of data samples from our AO. For this we need a `plist` containing an array of samples we want to select. We take our sine wave again which is stored in the variable `a` to see how it works.

```
a_select = select(a, plist('samples', 200:800));
iplot(a, a_select)
```

◀ Whitening noise

Split and join AOs ▶

Split and join AOs

You can split the data inside an AO to produce one or more output AOs. The `ao/split` method splits an AO by samples, times (if the AO contains time series data), frequencies (if the AO contains frequency data), intervals, or a number of pieces. We can control this as usual by defining our parameters.

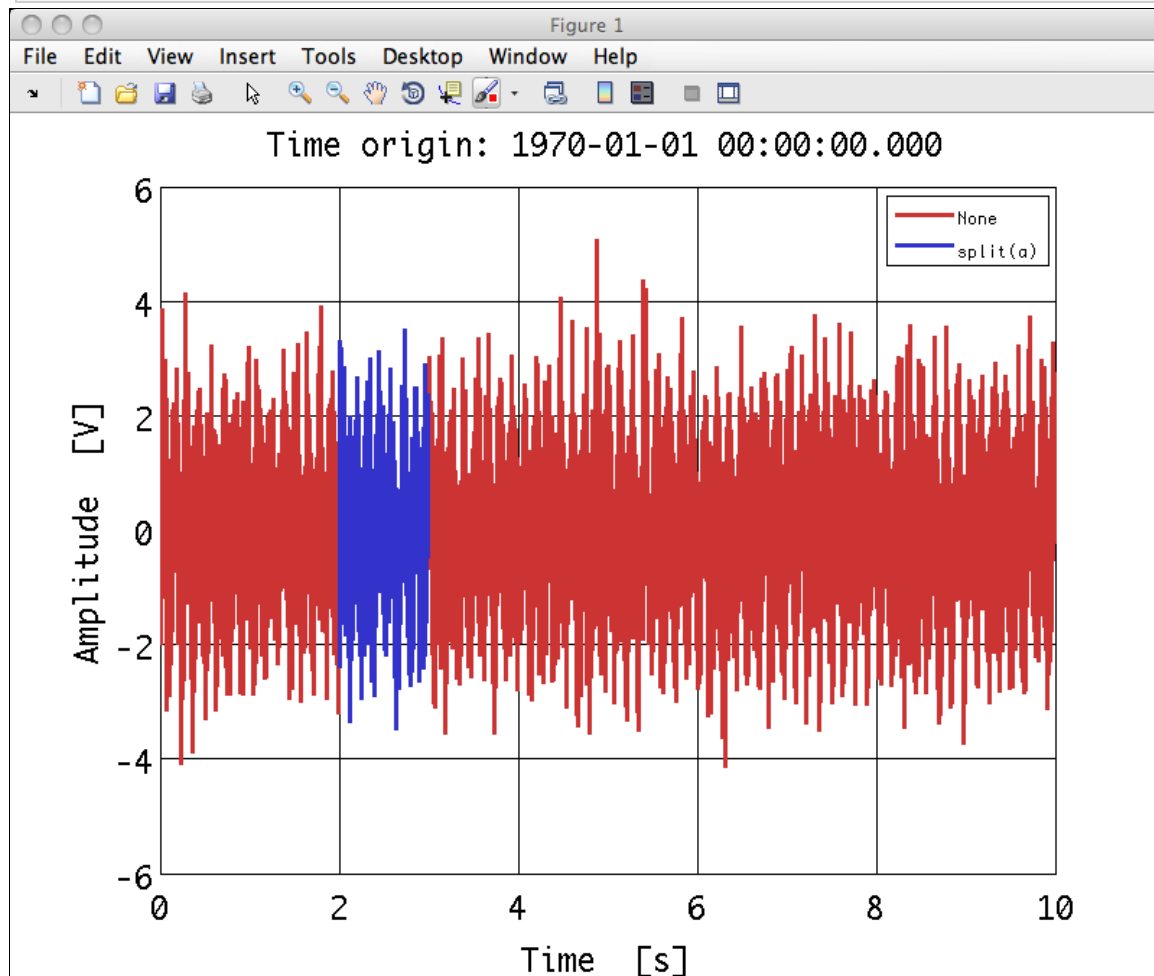
Split by times

Let us create a new time series AO for these examples.

```
pl = plist('nsecs', 10, 'fs', 1000, 'tsfcn', 'sin(2*pi*7.433*t) + randn(size(t))', 'yunits', 'V');
a = ao(pl);
```

For splitting in time we need to define a time vector for the parameter list and pass it to `ao/split`

```
pl_time = plist('times', [2 3]);
a_time = split(a, pl_time);
ipplot(a, a_time)
```



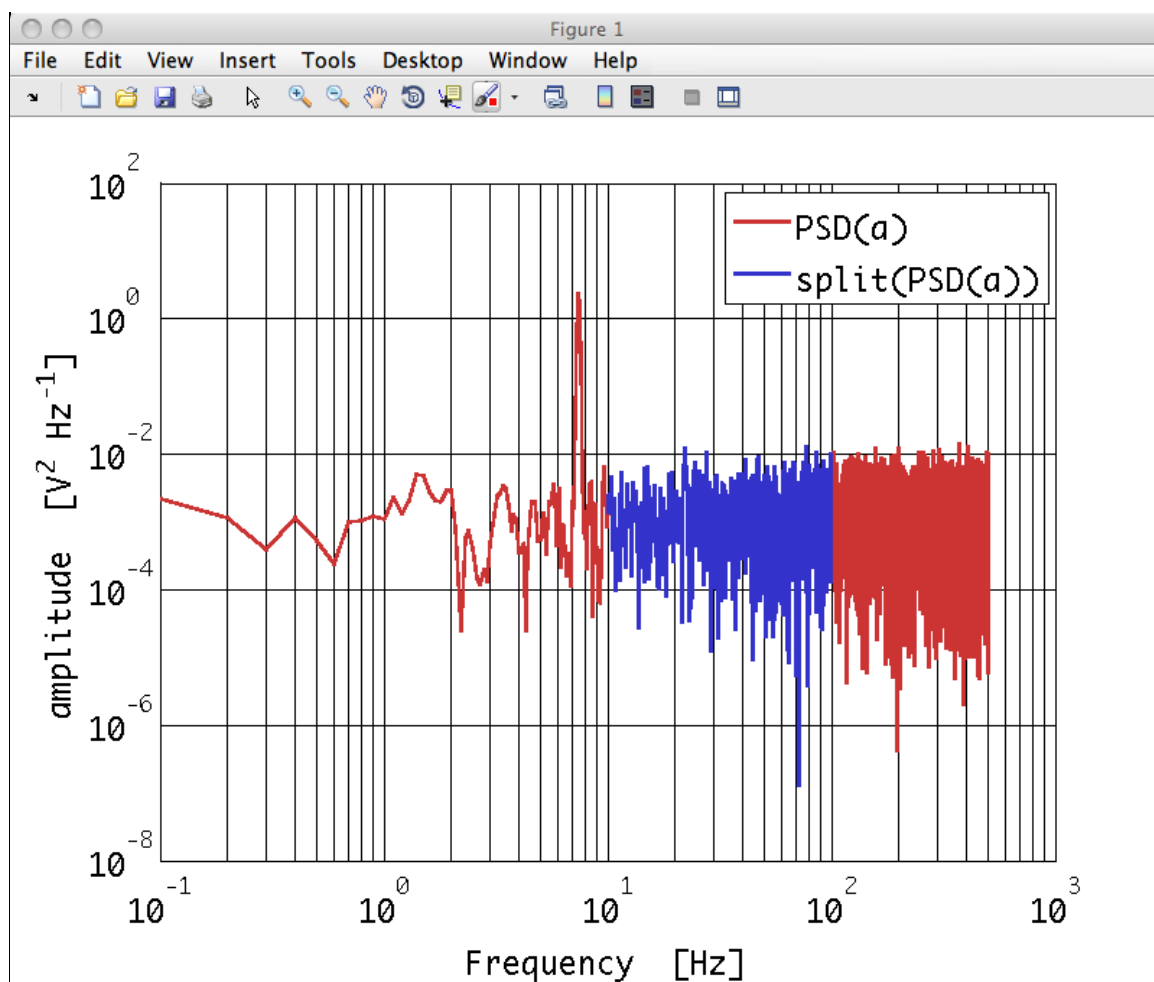
Split by frequencies

For this we need a frequency data AO. One easy way to get this is by computing the power spectrum using `ao/psd`.

```
axx = a.psd;
```

Again we need a vector for the parameter list and pass it to `ao/split`

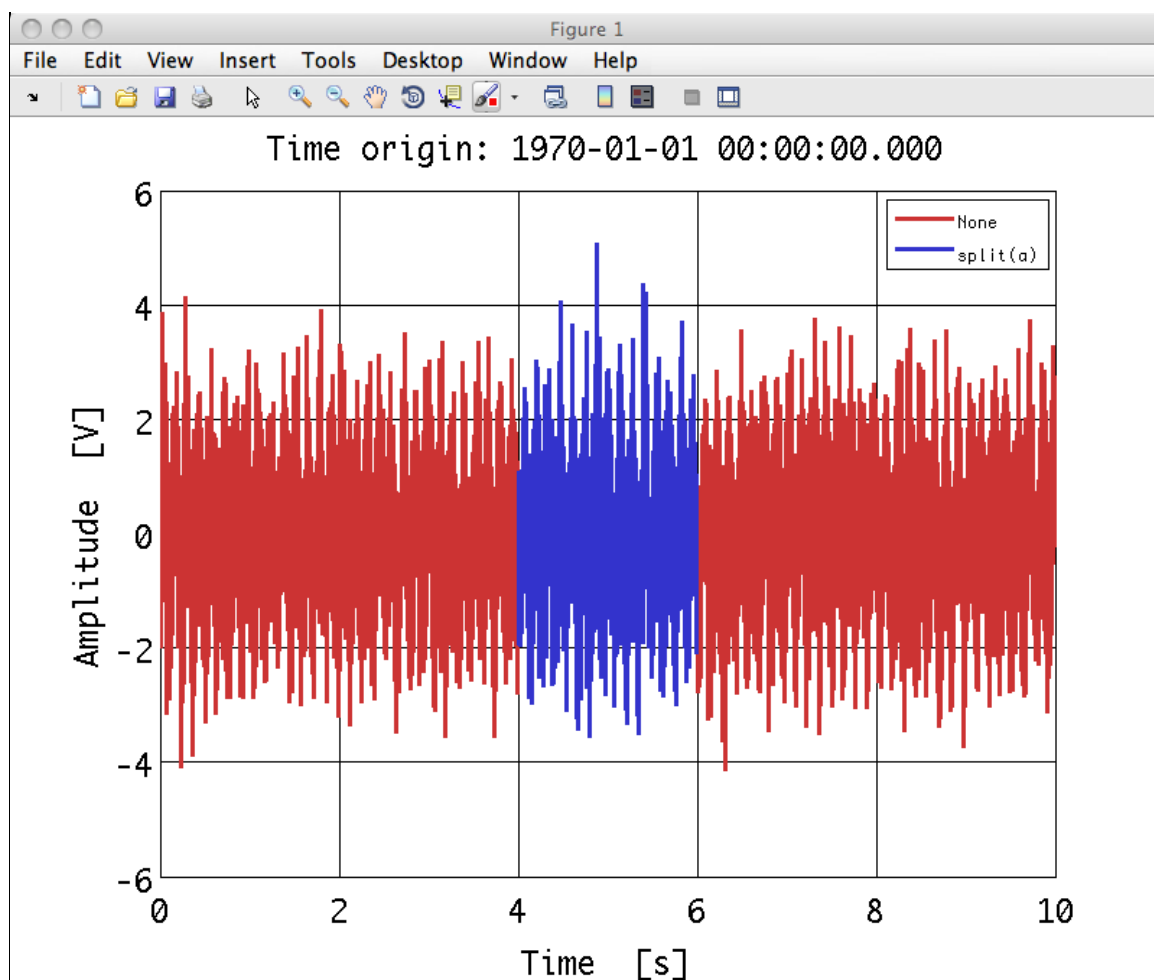
```
pl_freq = plist('frequencies', [10 100]);
axx_freq = split(axx, pl_freq);
ipplot(axx, axx_freq)
```



Split by intervals

We can also split the AO by passing a time interval to the `ao/split` method.

```
pl_interv = plist('start_time', 4, 'end_time', 6);
a_interv= split(a, pl_interv);
iplot(a, a_interv)
```

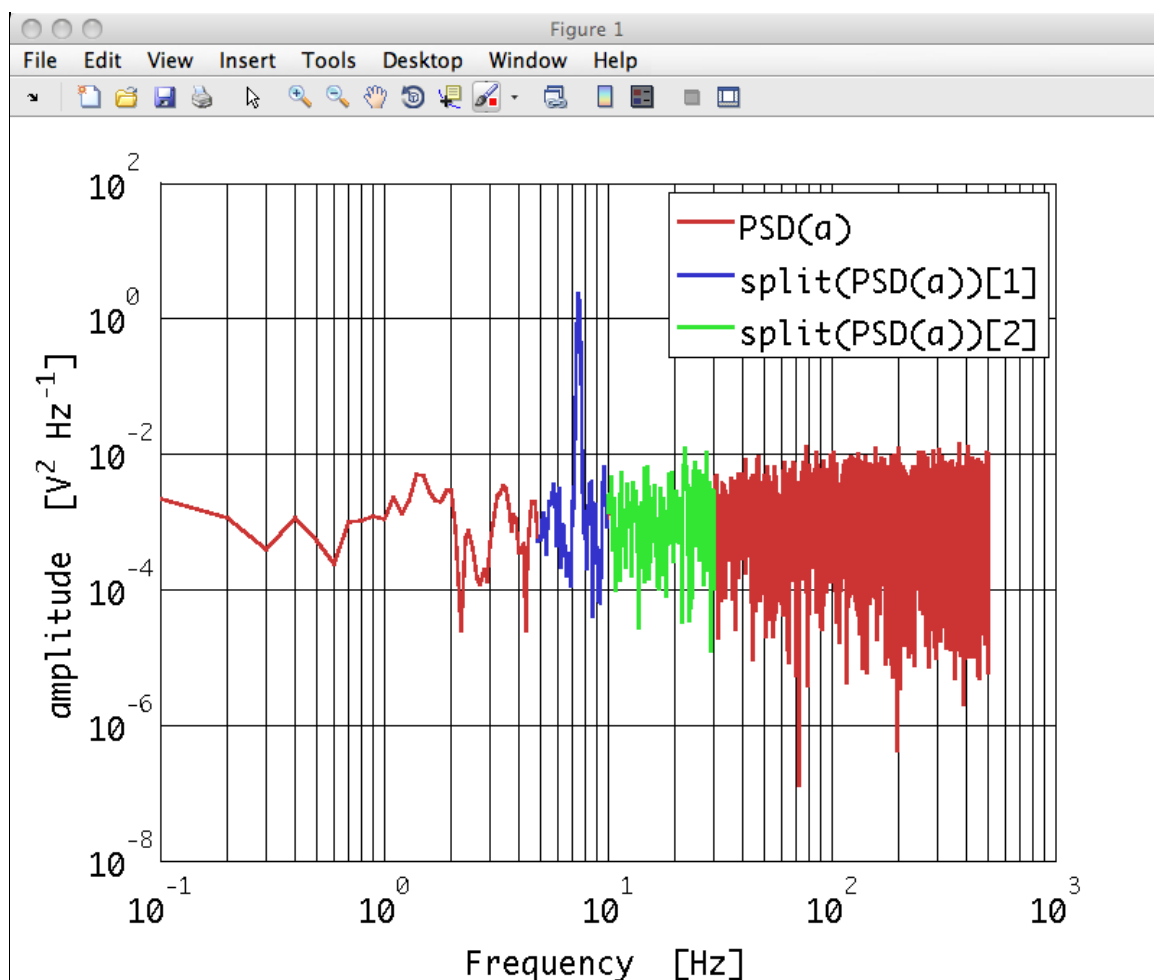


Split by samples

This type of splitting method we can use on any type of data. Let us use the frequency type, `axx`.

Again we need a vector for the parameter list and pass it to `ao/split`, only that this time we will split our AO in to two parts.

```
pl_samp = plist('samples', [50 100 101 300]);
[axx_samp1 axx_samp2]= split(axx, pl_samp)
iplot(axx, axx_samp1, axx_samp2)
```

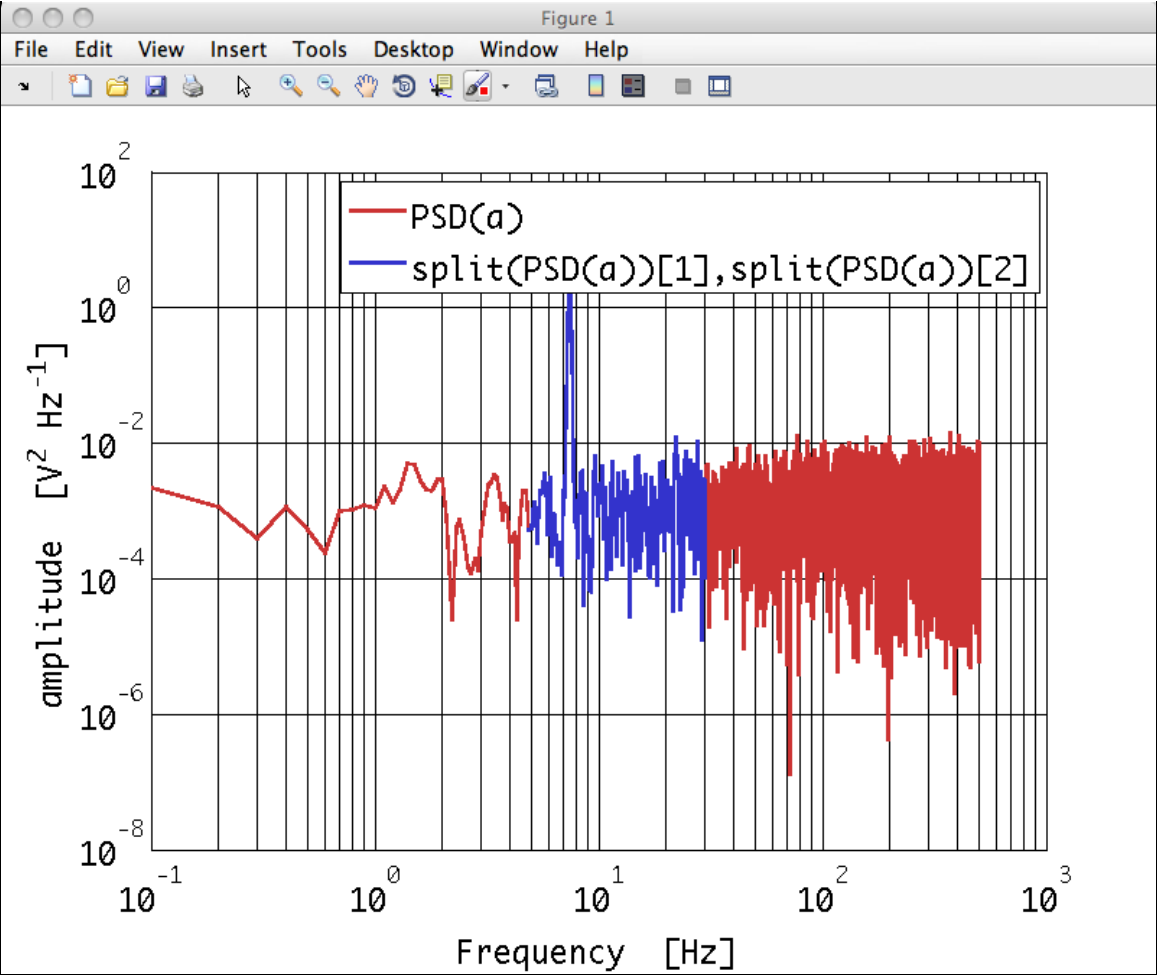


Although in this example the two resulting AOs are contiguous, they need not to be.

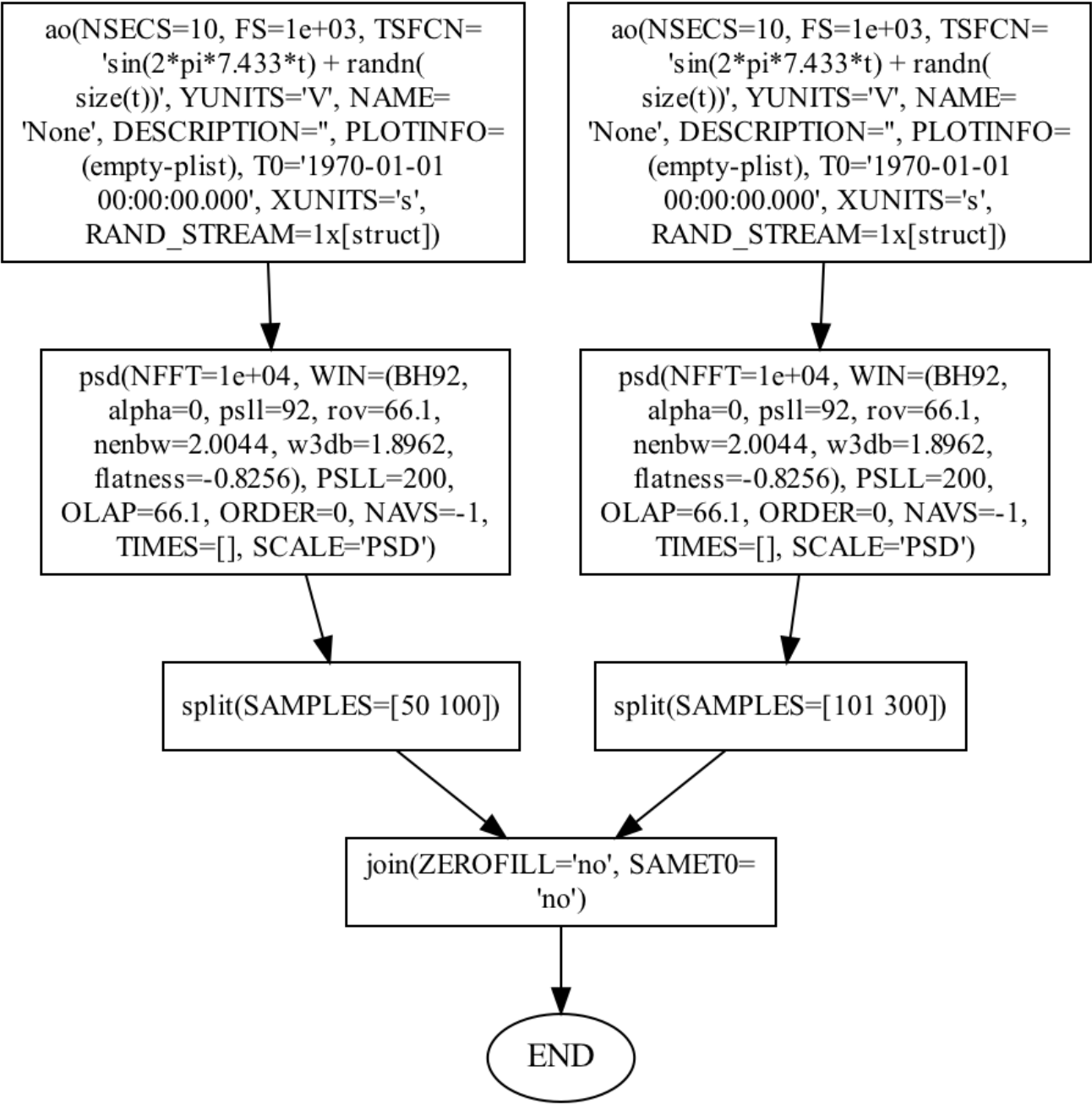
Join AOs

We can join our two AOs back together using `ao/join`

```
axx_join = join(axx_samp1, axx_samp2);
ipplot(axx, axx_join)
```



If we look at the history for axx_join (for instance, by entering `dotview(axx_join.hist)`), we will see the following:



Since the two AOs that are output from the 'split by samples' stage are independent, the history tree reflects this, showing two independent branches leading to the join step.

IFO/Temperature Example – Pre-processing

Now we return to the IFO/Temperature example that was started in Topic 1.

Loading and checking the calibrated data sets from topic2

In the last topic you should have saved your calibrated data files as

ifo_temp_example/ifo_disp.xml and
ifo_temp_example/temp_kelvin.xml

Now load each file into an AO, and simplify the name of the objects by assigning them the name of the Matlab workspace variable containing them:

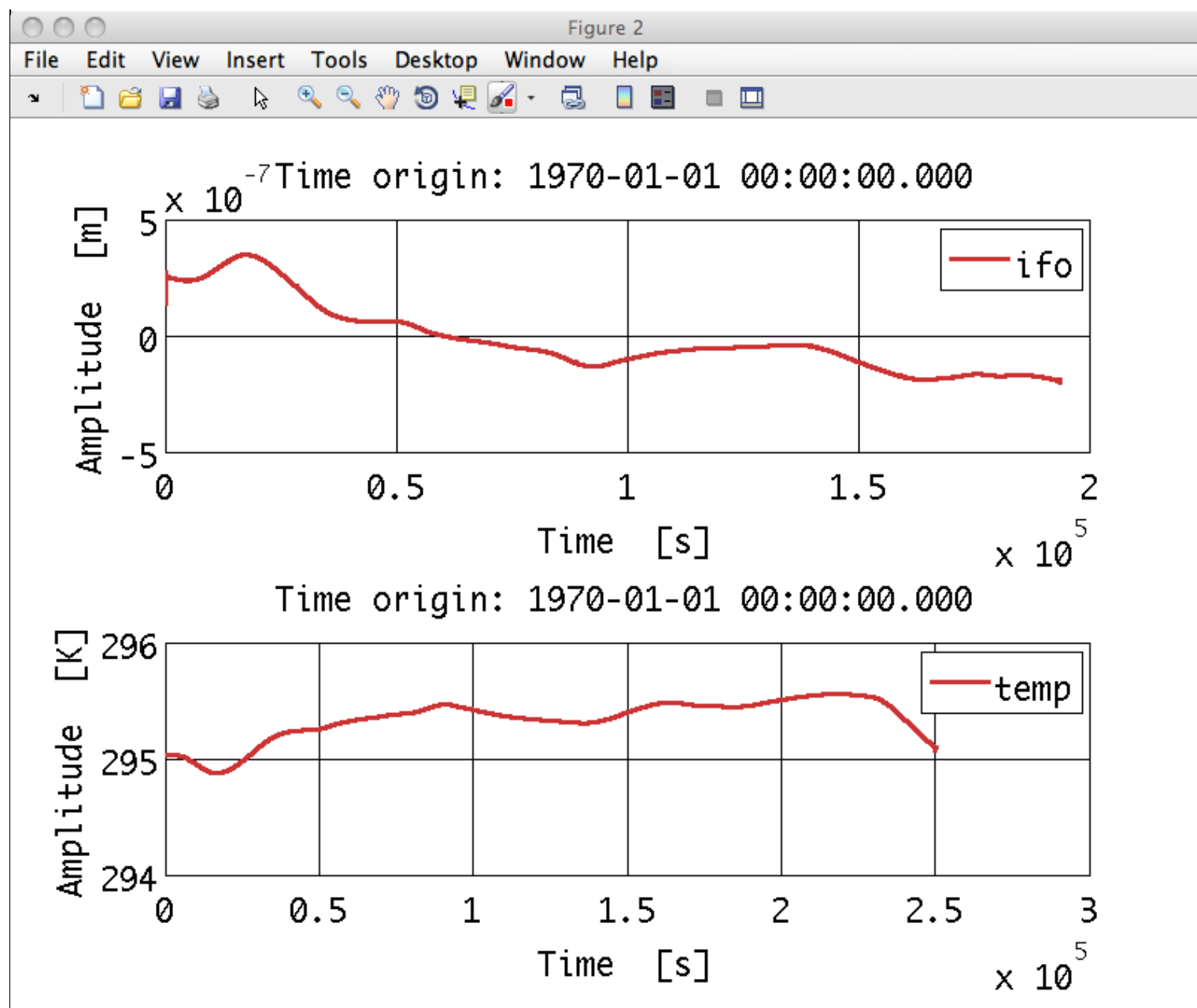
```
ifo = ao('ifo_temp_example/ifo_disp.xml');  
temp = ao('ifo_temp_example/temp_kelvin.xml');  
ifo.setName;  
temp.setName;
```

Let's see what kind of pre-processing we have to apply to our data prior to further analysis.

You can have a look at the data by for example displaying the AOs on the terminal and by plotting them, of course. Since the two data series have different Y units, we should plot them on subplots. To do that with `iplot`, pass the key 'arrangement' in a plist. For example:

```
pl = plist('arrangement', 'subplots');
```

If you plot the two time-series you should see something like the following:



Some points to note:

1. The two data streams:
 - do not have the same sampling frequency.
 - are not of the same length (nsecs).
2. The interferometer data has a small transient at the start

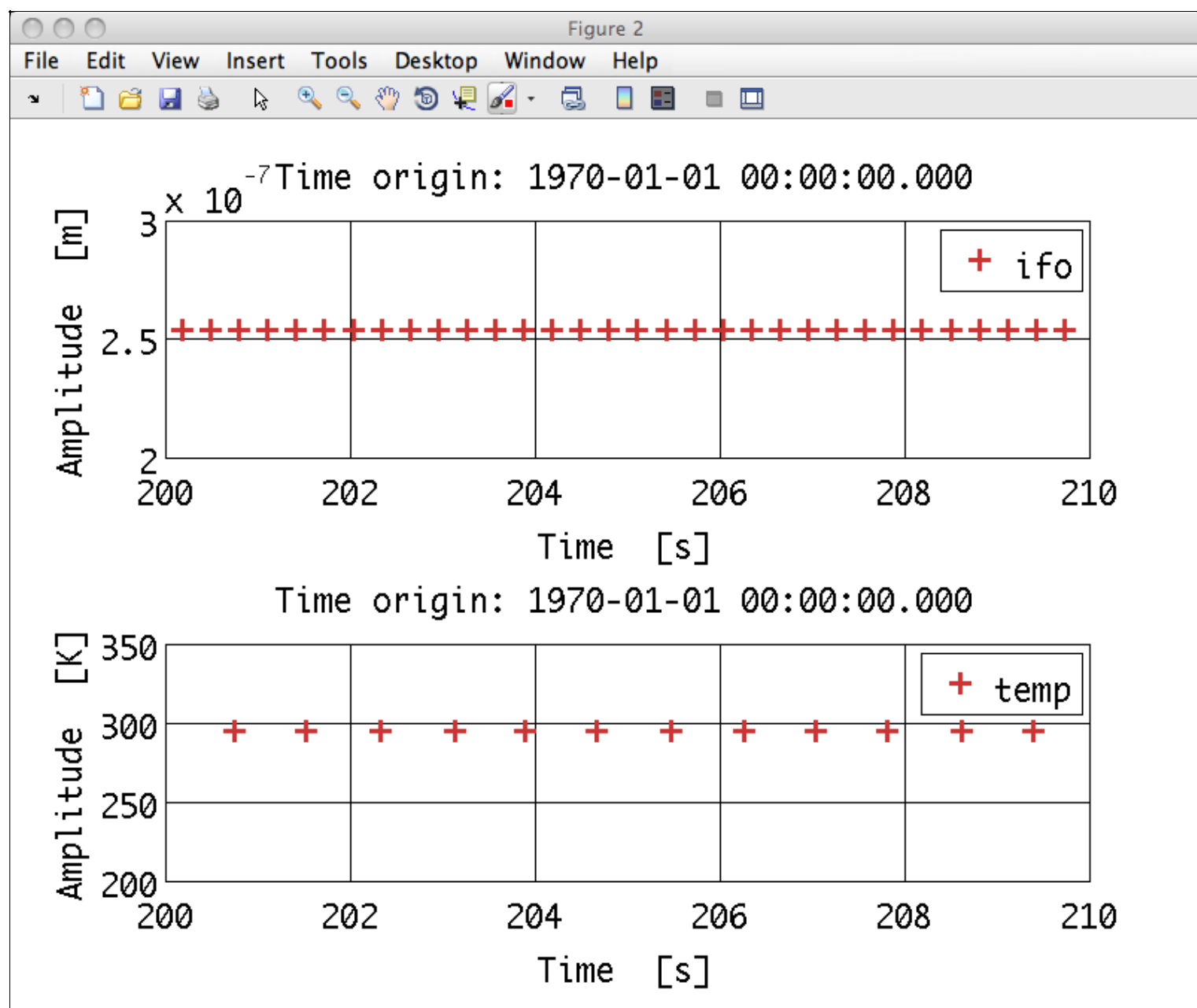
To have a closer look at the samples we plot markers at each sample and only plot a zoomed-in section. You can do this by creating a parameter list with the following key/value pairs (remember the 'key' properties for plist entries is not case sensitive):

Key	Value
ARRANGEMENT	'subplots'
LINESTYLES	{'none','none'}
MARKERS	{'+','+'}
XRANGES	{'all', [200 210]}
YRANGES	{[2e-7 3e-7], [200 350]}

Notice the use of the keyword 'all' in the value for the 'XRANGES' parameter. Many of the `iplot` options support this keyword, which tells `iplot` to use the same value for all plots and subplots. For the 'YRANGES' we specify different values for each subplot.

Please store your parameter lists in 2 different variables. We can reuse them for plotting our results later. If you are working on a pipeline instead of a script, you can use two `plist` constructor blocks and pass these as an input to an `iplot` block.

Passing such a parameter list to `iplot` together with the two AO time-series should yield a plot something like:



From this plot you may be able to see that the temperature data is unevenly sampled.

To confirm this, enter the following (standard) MATLAB commands on the terminal:

```
dt = diff(temp.x);
min(dt)
max(dt)
```

Don't forget the semicolon at the end of the `diff` calculation; this is a long data series and will be printed to the terminal if you do forget.

You see that the minimum and maximum difference in the time-stamps of the data is different, showing that the data are not evenly sampled.

Before we proceed with the later analysis of this data, we need to

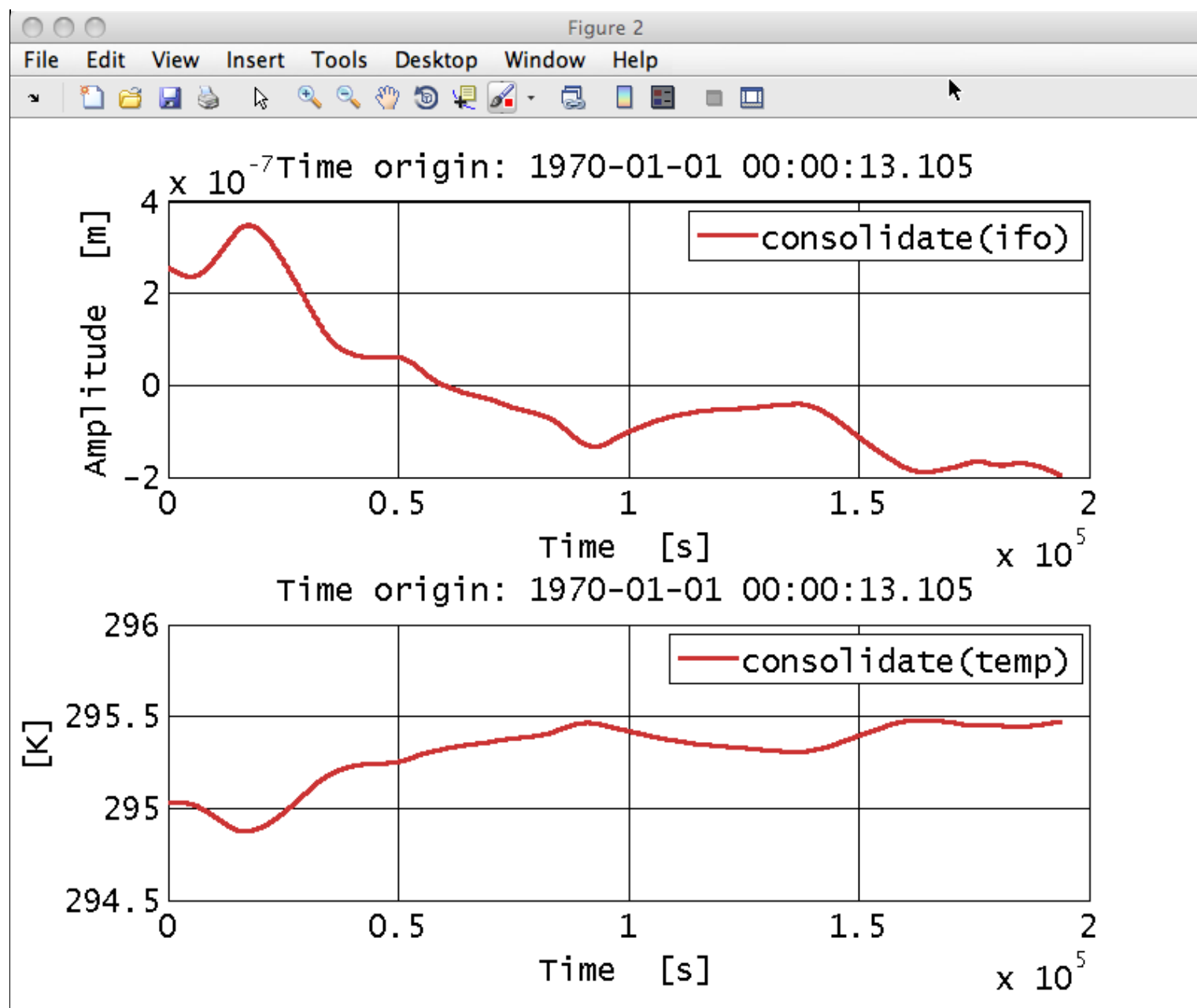
- Fix the uneven sampling of the temperature data
- Resample both data streams to the same rate
- Resample both data streams on to the same timing grid
- Select the segment of interferometer data that matches the temperature data

Each of these steps can, in principle, be done by hand. However, LTPDA provides a 'data fixer' method called `ao/consolidate` which attempts to automate this process. The call to `consolidate` is shown below:

```
[temp_fixed ifo_fixed] = consolidate(temp, ifo, plist('fs',1));
```

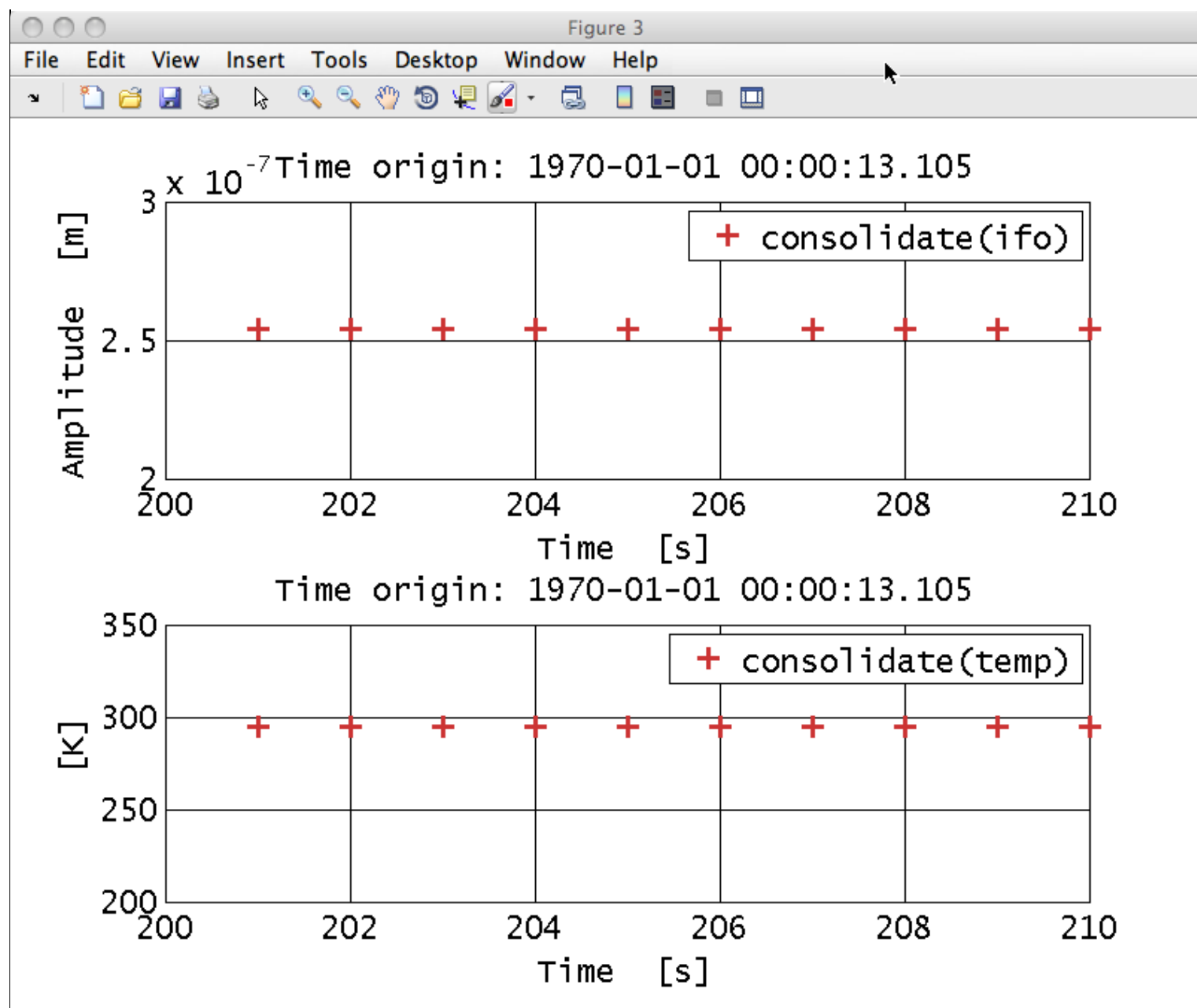
We tell `consolidate` that we want to have our data resampled to 1 Hz by specifying the parameter key 'fs'.

Now we can inspect the time-series of these data. The result should look something like the figure below:



Note that the time origin above the plots has now changed from zero to 13.105 which was the time of the first sample in the temperature measurement.

If we also plot the zoomed-in view again, we should see something like:



As you can see `consolidate` solved all our issues with these two data streams. They now start at the same time and are evenly sampled at the same sampling frequency.

In the next topic, we will look at the spectral content and coherence of the data before and after the pre-processing. For now, finish by saving the consolidated data ready for the next topic.

```
save(temp_fixed, 'ifo_temp_example/temp_fixed.xml');
save(ifo_fixed, 'ifo_temp_example/ifo_fixed.xml');
```

Topic 3 – Spectral Analysis

Training session 3 is a tutorial of how to estimate spectral properties of the signals, employing the instruments provided by the LTPDA Toolbox. As described in the devoted User Manual [section](#), spectral estimation is a branch of the signal processing, performed on data and based on frequency-domain techniques.

The focus of the tools is on time-series objects, whose spectral content needs to be estimated.

We will learn here how to use the tools to evaluate univariate and multivariate analyses.

The topic is divided as follows:

- [Exercises on Power Spectral Density estimation](#)
- [Exercises on Transfer Functions estimation](#)
- [IFO/Temperature Example – Spectral Analysis](#)

◀ IFO/Temperature Example – Pre-processing

Power Spectral Density estimation ▶

©LTP Team

Power Spectral Density estimation

In this subsection we will focus on the evaluation of the PSD (Power Spectral Density) for a given time-series signal. The functionality is provided by a method of the `ao` class called

ao/psd

which implements the Welch method of averaging modified periodograms (also referred to as WOSA). More details can found in the dedicated [section](#) of the user manual.

In this tutorial, we propose the following exercises based on the estimation of the PSD of suitable time-series data:

1. [Simply PSD](#): a very basic starting exercise
2. [Windowing data](#): introducing the usage of segment averaging/windowing/detrending
3. [Log-scale PSD on MDC1 data](#): a "realistic" example from the first LTPDA Mock Data Challenge, employing the log-scale PSD estimation method

Example 1: Simply PSD

Let's run our first exercise by means of the graphical programming environment called LTPDA Workbench. As you remember, details on how to use the Workbench were given in previous steps of this tutorial, such as [here](#). Anyways ...

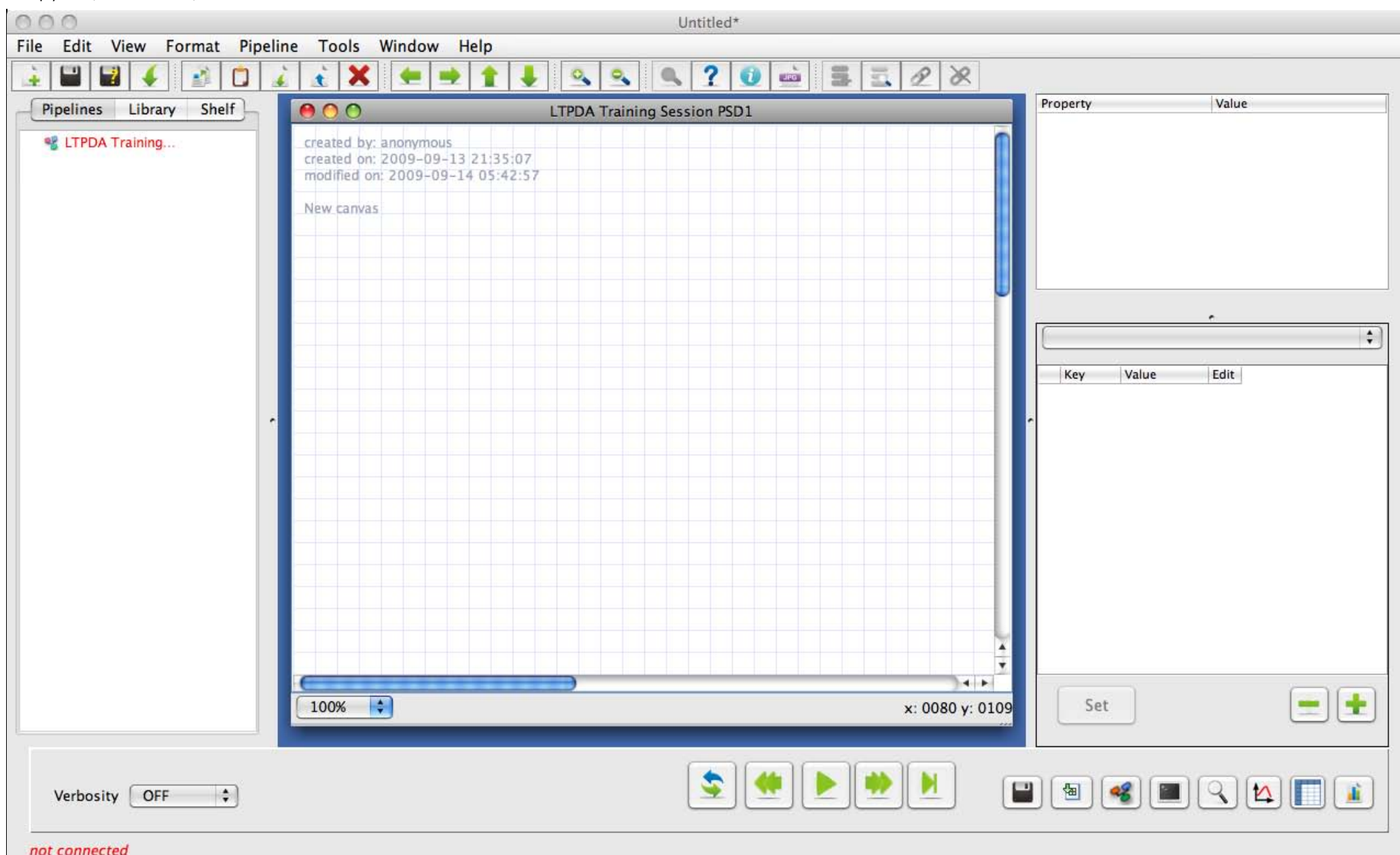
To start the workbench, issue the following command on the MATLAB terminal, or click on the "LTPDA Workbench" button on the launch bay.

```
LTPDAworkbench
```

Now let's go ahead and create a new pipeline, or analysis diagram. There are many ways to do this: hit `ctrl-n` (`cmd-n` on Mac OS X), or select "New Pipeline" from the "Pipeline" menu. (For more details on this and the other commands using the workbench environment please refer to the [appropriate section](#) of the user manual.)

Let's use the command "Pipeline -> Rename Pipeline" to give this diagram a more significant name, such as, for instance, "LTPDA Training Session PSD1". You should see a window like the one below:

Example 1: Simply PSD (LTPDA Toolbox)



The idea of the first exercise is the following:

1. simulate a time-series of white noise data
2. evaluate the Power Spectrum of the data

3. calculate the square root of the calculated power spectrum
4. plot the results

In a flow diagram, the representation is as follows:

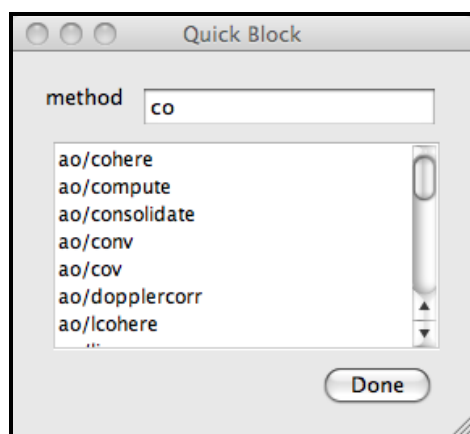


Simulate a time-series of white noise data

There are many different ways to simulate a white noise time series data. Here we choose a pretty powerful one. Add an LTPDA Algorithm block to the canvas, selecting the block in the LTPDA library, and either

1. drag the block to the canvas
2. hit `return` to add the block to the canvas
3. right-click on the library entry and select 'add block'

You can also use the "Quick Block" dialog. This is especially useful if you know the name of the block you are looking for. To open the Quick Block dialog, hit `ctrl-b` (`cmd-b` on Mac OS X) on the Canvas.



To get the `ao` constructor block we want, just start typing in the "method" edit field. Once the block `ao` is top of the list, just hit `enter` to add it to the canvas. You can also double-click on the block list to add any of the blocks in the list. Hit `escape` or click "Done" to dismiss the Quick Block dialog.

Now select the new AO block and choose the "From Waveform" option set from the "Parameters" drop-down list. Hit "Set" to assign this choice to the currently selected `ao` constructor block.

We can now tune the parameters of the `ao` constructor: in particular, let's double click on the value of the parameter line with the key `WAVEFORM`, so we can choose "noise" from the drop-down list that will appear. If needed, more help can be found [here](http://www.lisa.aei-hannover.de/ltpda/usermanual/ug/ltpda_training_topic_3_2_1.html).

Then let's make the time series last longer by setting the number of seconds (`nsecs`) to 1000.

Eventually, we set the units of the noise to be `meters` by selecting the "Yunits" parameter and entering 'm'.

Evaluate the Power Spectrum of the data

Now let's go ahead and search within the library for the `psd` method of the `ao` class. To do that, just click on the "Library" button on the top left of the screen, and type the word in the "search" box. Once we found the `psd` method, let's add it to the diagram, and then connect its input to the output of the `ao` constructor block. Some details and hints on connecting blocks can be found [here](#).

The next step is to choose the parameters. After selecting the "Default" set and clicking "Set", we can proceed to modify the parameters. Let's discuss the parameters and their meaning:

- `Nfft` allows to choose the number of samples in each periodogram evaluation
- `win` allows to choose the type of spectral window to be employed to reduce the edge effects at beginning/end of the data sections.
- `Psll` allows to choose the peak-sidelobe level in the case of Kaiser windows.
- `olap` allows to choose the percentual overlap between subsequent segments
- `Order` allows to choose the degree of detrending applied to each segment prior to windowing
- `Navs` allows to choose the number of averages to perform for each frequency bin
- `Times` allows to choose a reduced set of the inputs by asking the are within a given time range
- `Scale` allows to choose the quantity to be sent in output (ASD, PSD, AS, PS)

If the user does not specify any value for the parameters, the routine applies the default values. When called within the LTPDAworkbench graphical environment, the default parameters are explicitly shown after selecting the "Default" set and clicking "Set".

In this case, we will use the default parameters:

- `Nfft` = -1 so to estimate the PSD only on one window, including the whole data set
- `win` = The value set by the user in the preferences
- `Psll` = 200 in the case of Kaiser window
- `olap` = -1 so the overlap will be chosen based on the windows properties
- `Order` = 0 so to remove the mean value from the data before applying the window
- `Navs` = -1 so to set the length of the window (all data, in this case) rather than asking for a given number of means at each frequency
- `Times` = [] so to analyze all the input data
- `Scale` = 'PSD' so we evaluate the Power Spectral Density [output units] will be [input units]^2 / Hz

Extract the square root of the calculated power spectrum

For this exercise, we proceed by computing the square root of the calculated spectrum. Notice: by default, as described in its help, `ao/sqrt` applied to `fsdata` and `tsdata` objects acts only on the y (dependent) variable. We'll see later that is possible to choose different outputs so as to avoid this step, if we wanted to, by selecting the "Value" 'ASD' for the "Key" 'Scale'.

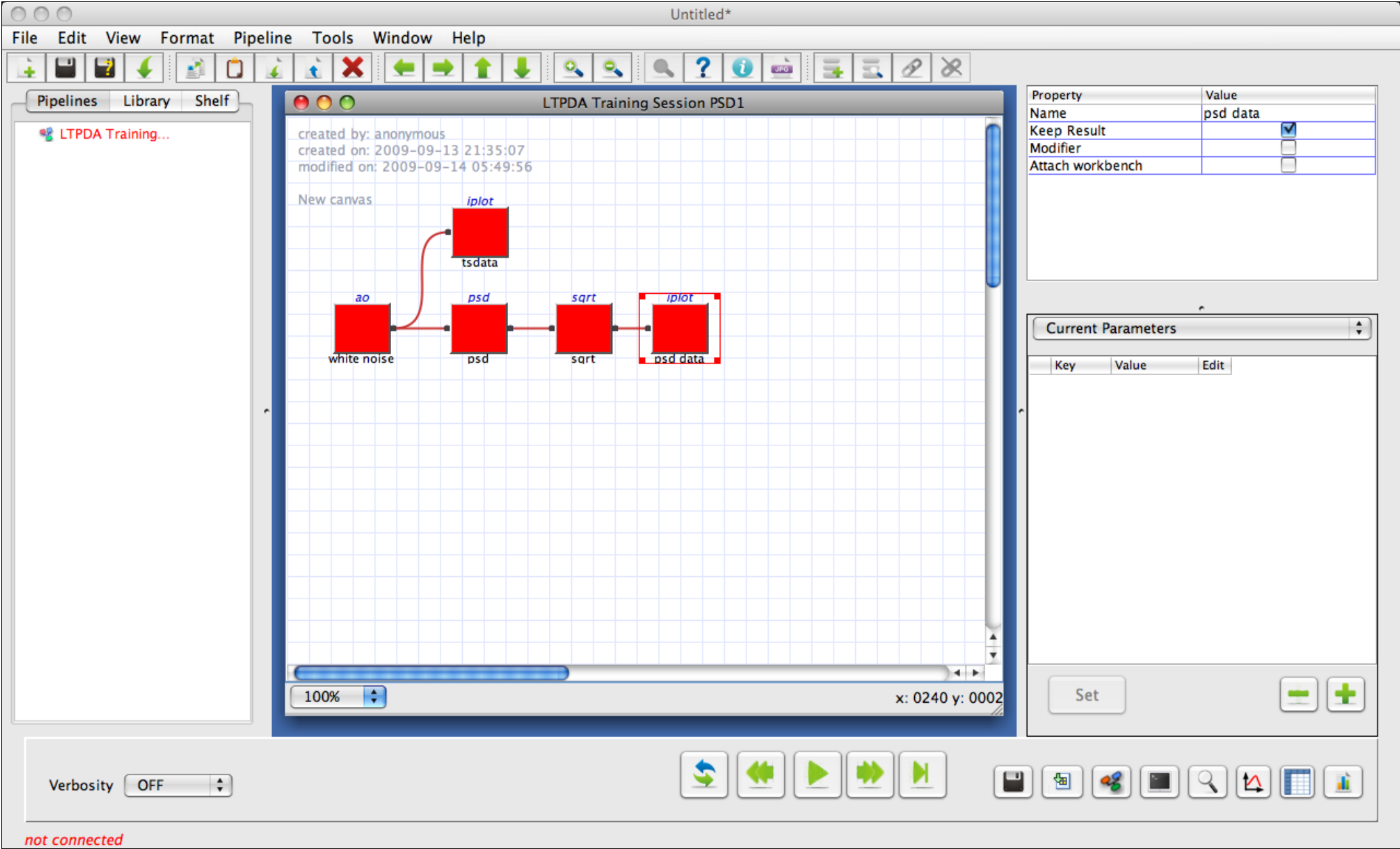
For now, let's add the `sqrt` method to the diagram and connect it to the output of the `psd` method.

Plot the results

What remains is just to plot the calculated square root of the PSD of the input white noise. Just add an `iplot` block and connect it ... and why not add another `ipplot` to look at the original time-series signal.

So at the end we should have a situation like the following:

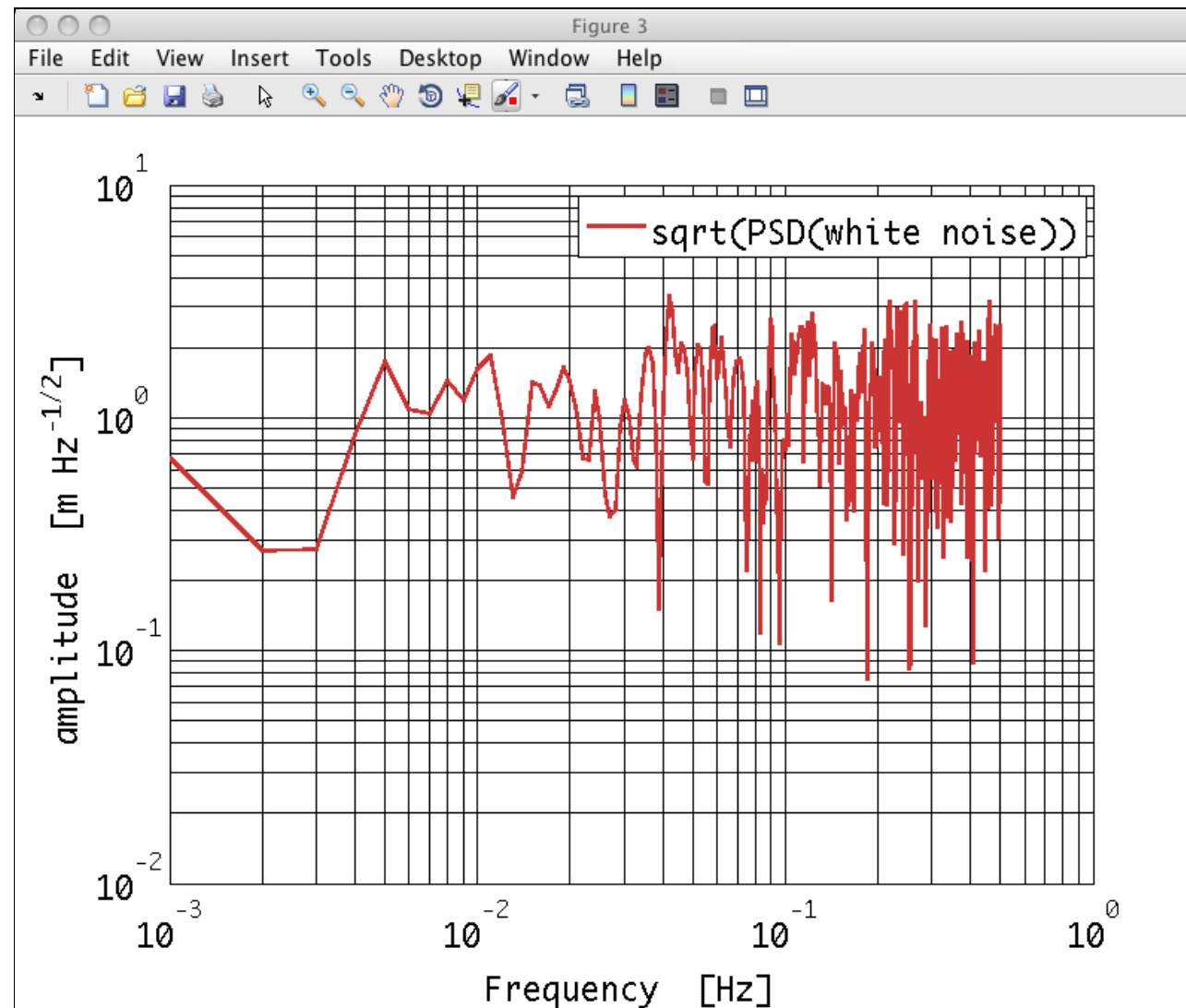
Example 1: Simply PSD (LTPDA Toolbox)



Notice that we also renamed the individual blocks, by double clicking on them and typing in the new names. We also changed the zoom amount by using the mouse scroll wheel or the commands available under the "View" menu.

The shape of the arrows, the color of the blocks and many other features can be adjusted by selecting "File->Preferences" when a Workbench is selected.

Now we can execute the diagram, by clicking on the "Run" button on the bottom center, and the calculation should end up with 2 figures. The second one should be the following:



Please notice:

- The log-log shape of the plot
- The x-axis units
- The y-axis units
- The history steps reported on the legend

Example 2: Windowing data

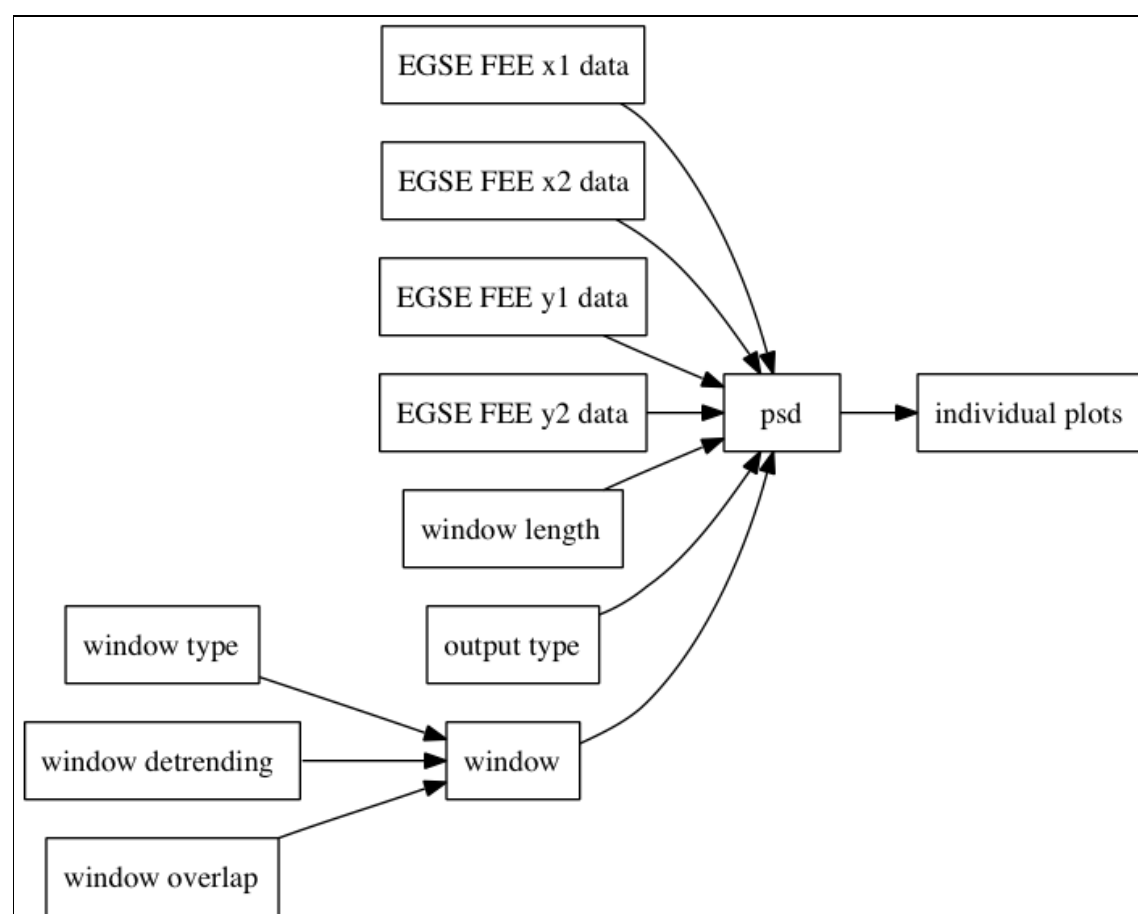
In the second exercise we will use again the graphical programming environment called LTPDA Workbench. After checking that the workbench is loaded, let's go ahead and create a new pipeline, or analysis diagram.

Let's use the command "Pipeline -> Rename Pipeline" to give this diagram a more significant name, such as, for instance, "LTPDA Training Session PSD2".

The idea of the second exercise is the following:

1. load a list of time-series with noise data from disk
2. evaluate the Power Spectrum of the different data sets and:
 - study the effect of the usage of different windows lengths
 - study the effect of the usage of different windows types
 - choose different outputs
3. plot the results in different plot styles

In a flow diagram, the representation is as follows:



Let's create a new pipeline and then use the command "Pipeline -> Rename Pipeline" to give this diagram a more significant name, like "LTPDA Training Session PSD2".

Loading experimental data time-series from data files

This step was touched upon in [previous steps](#) and in the [user manual](#). Here we go ahead by adding an ao constructor method/block, that we can retrieve from the library or with the "quick block"

shortcut.

We give the block a sensible name by double-clicking on it, and then we proceed with setting the parameters as follows:

- 1. Let's first choose, from the "Parameters" drop-down list, the "From ASCII File" set, and hit "Set" to assign this choice to the currently selected ao constructor block.
- 2. We can now tune the key parameters of the ao constructor: in particular, let's double click on the first parameter line, within the "Value" column, so we can choose the filename from the file browser that will appear.
- 3. Similarly, let's go ahead and insert the values for the others parameters.

To add parameters, we click on the "+" button, subsequently define the "Key" entry, which is the parameter *name*, and the "Value" entry, which contains the parameter *value*.

Key	Value	Description
FILENAME	'topic3/EGSE_FEE_x1.dat'	The name of the file to read the data from.
TYPE	'tsdata'	Interpret the data in the file as time-series data.
COLUMNS	[1 2]	Load the data x-y pairs from columns 1 (as x) and 2 (as y).
XUNITS	's'	Set the units of the x-data to seconds (s).
YUNITS	'F'	Set the units of the y-data to farad (F).
COMMENT_CHAR	'%'	Indicateds which header lines to skip in the ASCII data file.
FS	[]	Indicates to load time series from the first data column.
ROBUST	'yes'	Use robust data reading for this file format.
DESCRIPTION	'EGSE FEE x1 data'	Set some text to the 'description' field of the AO.

This procedure can be repeated for all 4 channels we want to analyze:

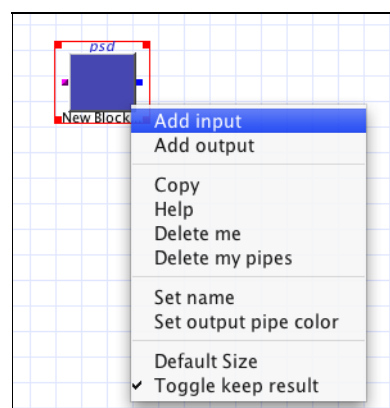
Key	Value
FILENAME	'topic3/EGSE_FEE_x1.dat'
FILENAME	'topic3/EGSE_FEE_x2.dat'
FILENAME	'topic3/EGSE_FEE_y1.dat'
FILENAME	'topic3/EGSE_FEE_y2.dat'

To speed up the procedure, we can copy & paste (using the contextual menu or the menu items or the shortcuts) the ao constructor block we just set up, and just change the filenames. We can also

'duplicate' blocks using `ctrl-d` (cmd-d on OS X), or by "Edit->duplicate".

Evaluate the Power Spectrum of the different data sets

Now let's proceed by adding the `psd` block to the diagram, as we did previously. Once we're done with this, we realize we want to call the method only once, without the need of putting 4 identical blocks; we also want to be sure to apply the same parameters to all the input objects. There are different ways to achieve this; one possibility is to take advantage of the multiple input handling allowed by the `ao/psd` method. So we go ahead and add 3 inputs to the block; to do that, place the mouse over the block, right-click and select "Add input":

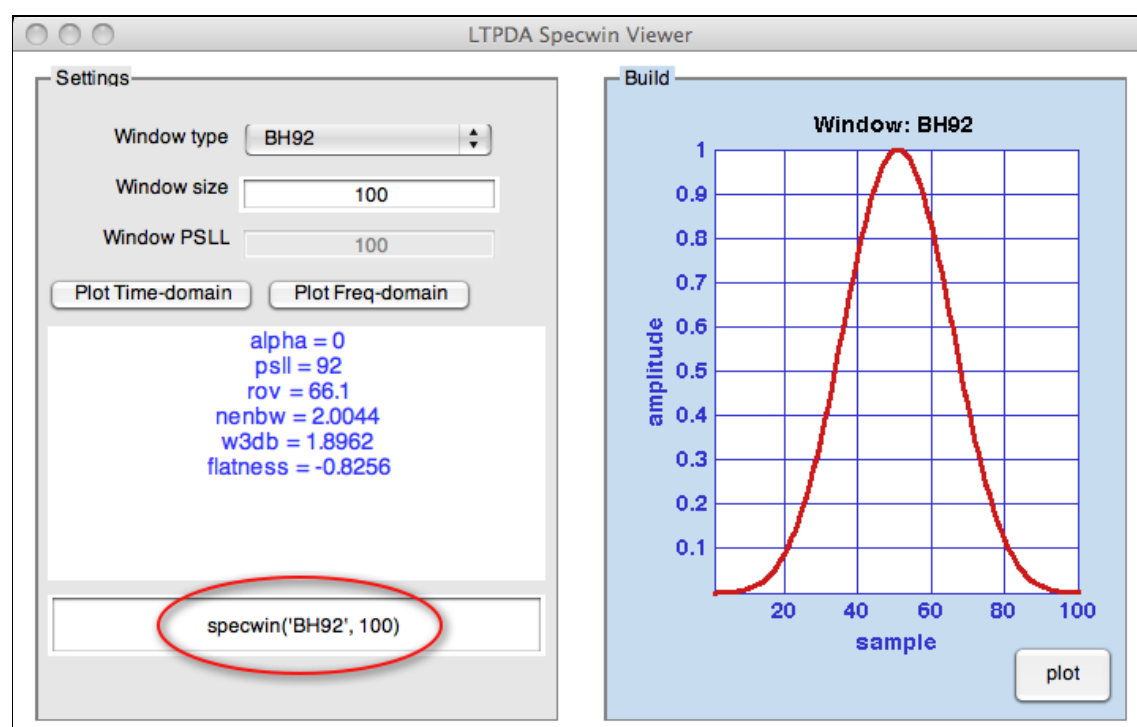


Then obviously we need to connect each of the `ao` constructor blocks to an input of the `psd` block.

The parameters that we will choose for the `psd` method will be applied to all the 4 objects.

Choosing the spectral window

The spectral leakage is different with different [windows](#). In order to choose the proper one for our needs, we can use the [specwin helper](#), to visualize the window object both in time-domain and frequency domain.



Once we are happy with the choice, we can go back to the workbench, double click on the "Value" field for the "Key" `win` in the parameters of the `psd` block. Then let's set the little panel to select the window. One parameter is already selected, based on the user preferences. Here I choose to use the Blackman-Harris window, called 'BH92'.

The length of the specwin object is irrelevant, because it will be ignored by the psd method, that will rebuild the window only based on the definition (the name).

The effective window length is decided by setting the "Nfft" parameter!

Choosing the window length

In order to reduce the variance in the estimated PSD, the user may decide to split the input data objects into shorter segments, estimate the fft of the individual segments and then average them. This is set by the "Nfft" parameter. A value of -1 will mean one single window.

Let's choose a window length of 20000 points (2000 seconds at 10 Hz).

Choosing the window overlap

In principle, we can decide the amount of overlap among consecutive windows, by entering a percentage value.

Let's do nothing here, leave -1 and let the psd use the recommended value which is stored inside the window object and shown as "Rov".

Choosing the scale

We can decide to give as an output directly the 'ASD' (Amplitude Spectral Density), rather than the 'PSD' (Power Spectral Density). We can also have 'AS' (Amplitude Spectrum) or 'PS' (Power Spectrum).

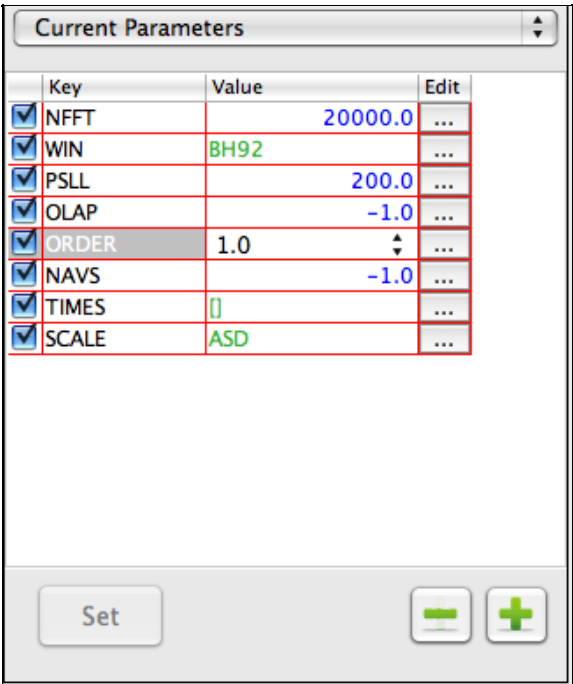
Here I choose to use ASD, so I double-click on the "Value" corresponding to the "Scale" entry and go ahead and enter the string, 'ASD'.

Choosing the detrending

Detrending here refers to an additional detrending performed for each individual segment, prior to applying the window.

In this case we want to subtract a linear drift, so just enter 1.

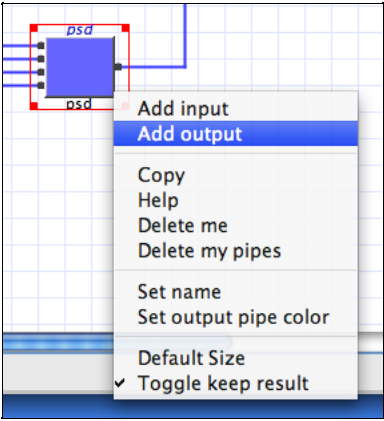
We should be ending in some parameter section like:



Plotting the spectra

By default, psd would give as an output an array of aos corresponding to each input. So in our case,

we'd have an array with 4 aos. We can also make them individuals, by adding outputs to the psd block. We just need to right-click and select "Add output":

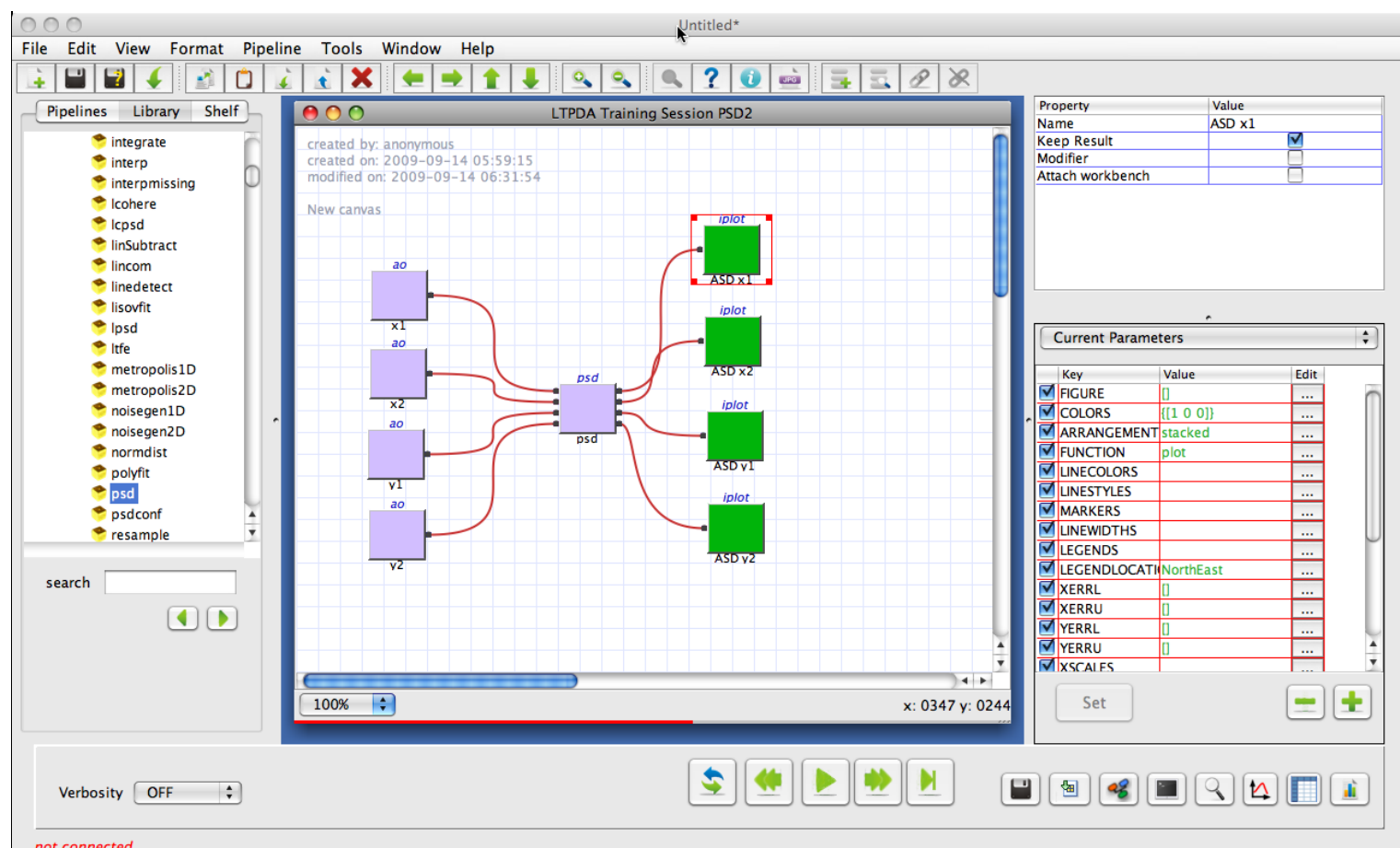


We can now go ahead and plot the individual aos in many plots, just by adding `ipplot` blocks and connecting them to each output port of the `psd` block.

We can also define parameters for each plot, such as colors an so on. Just to exercise let's set the colors to:

Object	Color
x1 data PSD	{[1 0 0]}
x2 data PSD	{[0 1 0]}
y1 data PSD	{[0 0 1]}
y2 data PSD	{[1 1 1]}

So the workbench is ready for execution, and it should be similar to:

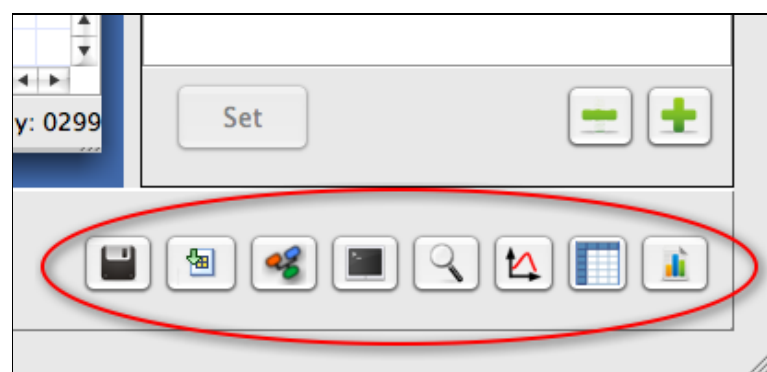


We can go ahead and execute it ...

At the end we can look at the output plots and check the results, the units, the frequency range.

More info on the spectra

We can extract more informations about the results we obtained, by exploiting the toolbox functionality via the buttons located in the right-bottom side of the GUI:



They allow us to:

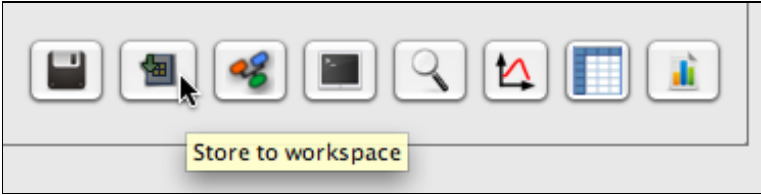
- Display the history of the selected object
- Display the selected object on the Matlab terminal
- Explore the selected object
- Plot the selected object
- ... and more!

A first introduction on the role/usage of these buttons was given [previously](#) so ... let's try them, for instance to obtain the information on how many windows were effectively applied (useful to evaluate

the statistical properties of the estimated ASD).

Storing the spectra to the workspace

One useful option is storing the data to the workspace, for further investigations or to store them to a database. After selecting the psd block, we hit the "Store to workspace" button.



Now, if we switch to the Matlab terminal, we can see between our variables that we have stored the outputs corresponding to the output ports of the psd block:

```
>> who
Your variables are:
LTPDA_Training_Session_PSD2   psd_PORT0                psd_PORT2
ans                          psd_PORT1                psd_PORT3
>>
```

As we did from the GUI, we can display the objects simply by:

```
>> psd_PORT0
----- ao 01: ASD(x1) -----
name: ASD(x1)
data: (0,2.5372422083703e-18) (0.00051203275,4.68581398941709e-18) ...
----- fsdata 01 -----
      fs: 10.2407
      x: [10001 1], double
      y: [10001 1], double
      dx: [0 0], double
      dy: [10001 1], double
xunits: [Hz]
yunits: [F Hz^(-1/2)]
      t0: 01-01-1970 00:00:00.000
      navs: 31
-----
hist: ao / psd / SId: psd.m,v 1.52 2009/09/05 05:57:32 mauro Exp S
mdlfile: empty
description: EGSE FEE x1 data
UUID: 4702fadd-d37a-49a7-be00-bc651258d31f
-----
>>
```

Saving the spectra

Once we dumped the objects we are interested on, we can go ahead and save at least the *x1* data as described [here](#) and [here](#).

We could also include a save step in the pipeline, tough, by inserting a save block. Let's grab it from the library, typing "save" in the "search" box, or selecting it under ao -> Output. Then let's connect it to the output of the psd block. After selecting, as usual, the "Default" parameter set from the parameters drop-down list, we can go ahead and enter a suitable file name and file location.



©LTP Team

Example 3: Log-scale PSD on MDC1 data

Let's run our third exercise by means of the LTPDA Workbench. If it is not already open, start the workbench issuing the following command on the MATLAB terminal, or clicking on the "LTPDA Workbench" button on the launch bay.

```
LTPDAworkbench
```

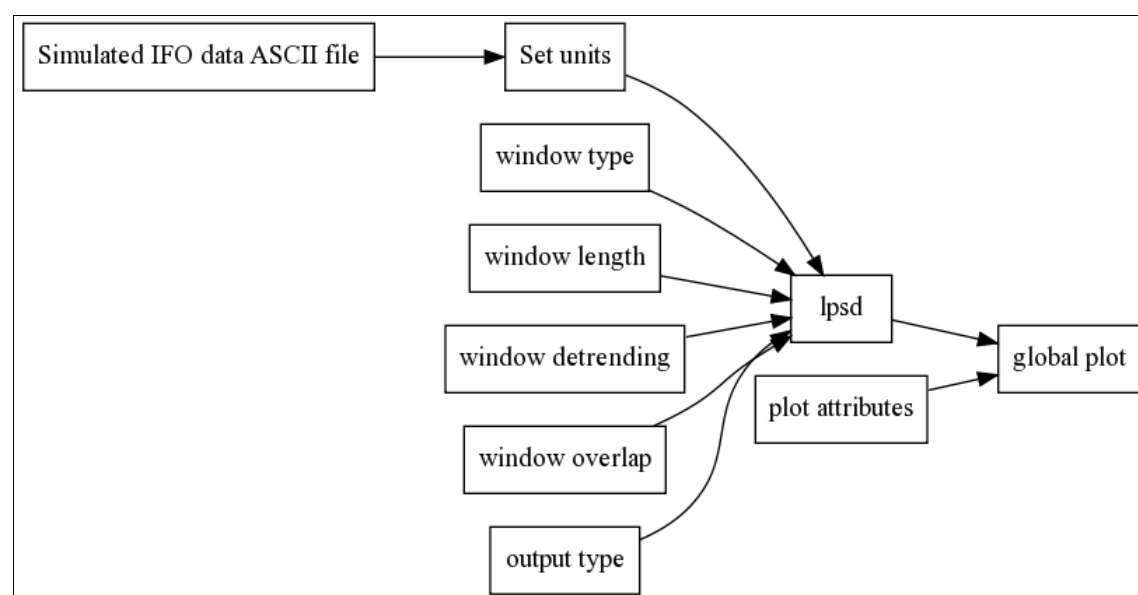
As we did before, let's go ahead and create a new pipeline, or analysis diagram.

Again, use the command "Pipeline -> Rename Pipeline" to give this diagram a more significant name, such for instance "LTPDA Training Session PSD3".

The idea of the third exercise is the following:

1. load a time-series of IFO noise data simulated for the first LTPDA Mock Data Challenge (MDC1)
2. set data units
3. evaluate the Log-Scale Power Spectrum of the 2 IFO channels data
4. set the plot properties
5. plot the results

In a flow diagram, the representation is as follows:



Load a time-series of IFO noise data simulated for the first LTPDA Mock Data Challenge (MDC1)

The step of building aos by loading data from files was touched upon in [previous steps](#) and in the [user manual](#). Here we go ahead by adding a ao constructor method/block, that we can retrieve from the library or with the "quick block" shortcut.

We give the block a sensible name by double-clicking on it, and then we proceed with setting the parameters as follows:

1. Let's first choose, from the "Parameters" drop-down list, the "From ASCII File" set, and hit "Set" to assign this choice to the currently selected ao constructor block.
2. We can now tune the key parameters of the ao constructor: in particular, let's double click on the first parameter line, within the "Value" column, so we can choose the filename from the file

browser that will appear. Similarly, let's go ahead and insert the vaues for the others parameters.

To add parameters, we click on the "+" button, subsequently define the "Key" entry, which is the parameter name, and the "Value" entry, which contains the parameter value.

Key	Value	Description
FILENAME	'topic3/mockdata_16_48_17_11_2007_1.dat'	The name of the file to read the data from.
TYPE	'tsdata'	Interpret the data in the file as time-series data.
COLUMNS	[1 2 1 3]	Load the data x-y pairs from columns 1 (as x) and 2 (as y), in the first ao, and from columns 1 (as x) and 3 (as y), in the second ao.
XUNITS	's'	Set the units of the x-data to seconds (s).
YUNITS	"	We leave this empty for now.
COMMENT_CHAR	'%'	Indicateds which header lines to skip in the ASCII data file.
FS	[]	Indicates to load time series from the first data column.
ROBUST	'no'	We don't need robust data reading for these simulated data.
DESCRIPTION	'MDC1 set #1, 17/11/2007'	Set some text to the 'description' field of the AO.

The output will be a vector of aos, containing:

- the x1 IFO measurement (TM1 to SC x position)
- the x12 IFO measurement (TM1 to TM2 relative x position)

We can then run all the analyses, and be sure to be applying the same parameters, by passing the vector of aos to the various methods.

Set data units

We forgot to set the units ... luckily both displacements are expressed in meters, so let's add a block

ao/setYunits

The relevant parameters to set, (after choosing, from the "Parameters" drop-down list, the "Default" set, and hitting "Set" to assign this choice to the currently selected block), are:

Key	Value	Description
YUNITS	'm'	The unit object to set as y-units

Evaluate the Log-Scale Power Spectrum of the 2 IFO channels data

Now let's go ahead and search within the library for the `lpsd` method on the `ao` class. To do that, just click on the "Library" button on the top left of the screen, and type the word in the "search" box. Once we found the `lpsd` method, let's add it to the diagram, and then connect its input to the output of the `ao` constructor block. Some details and hints on connecting blocks can be found [here](#).

The next step is choosing the parameters. After selecting the "Default" set and clicking "Set", we can proceed and modify the parameters. Four of these parameters are the same as we already discussed for `ao/psd`:

- `win` Allows to choose the type of spectral window to be employed to reduce the edge effects at beginning/end of the data sections.
- `olap` Allows to choose the percentual overlap between subsequent segments
- `order` Allows to choose the degree of detrending applied to each segment prior to windowing
- `scale` Allows to choose the quantity to be sent in output (ASD, PSD, AS, PS)

In this case, we will use the following parameters:

Key	Value	Description
WIN	'BH92'	Or a different one, if you want.
OLAP	-1	Overlap will be chosen based on the window properties
ORDER	2	Segment-wise detrending up to order 2
SCALE	'ASD'	Evaluate the Amplitude Spectral Density so that [output units] will be [input units] / sqrt(Hz)

We also need to set other 2 parameters that are typical of this method, discussed in the devoted user manual [section](#).

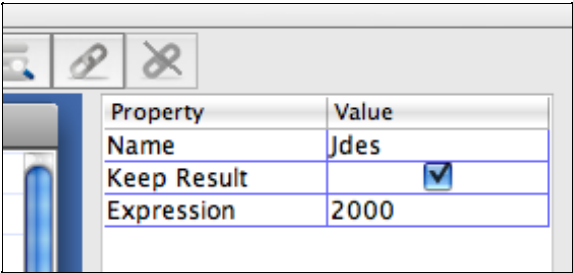
- `'Jdes'` – the number of spectral frequencies to compute
- `'Kdes'` – the desired number of averages
- `'Lmin'` – the minimum segment length [default: 0]

We will act on the first one, so to decide how many ASD bins to estimate, leaving to the algorithm the choice of their location and the length of the windows on each bin.

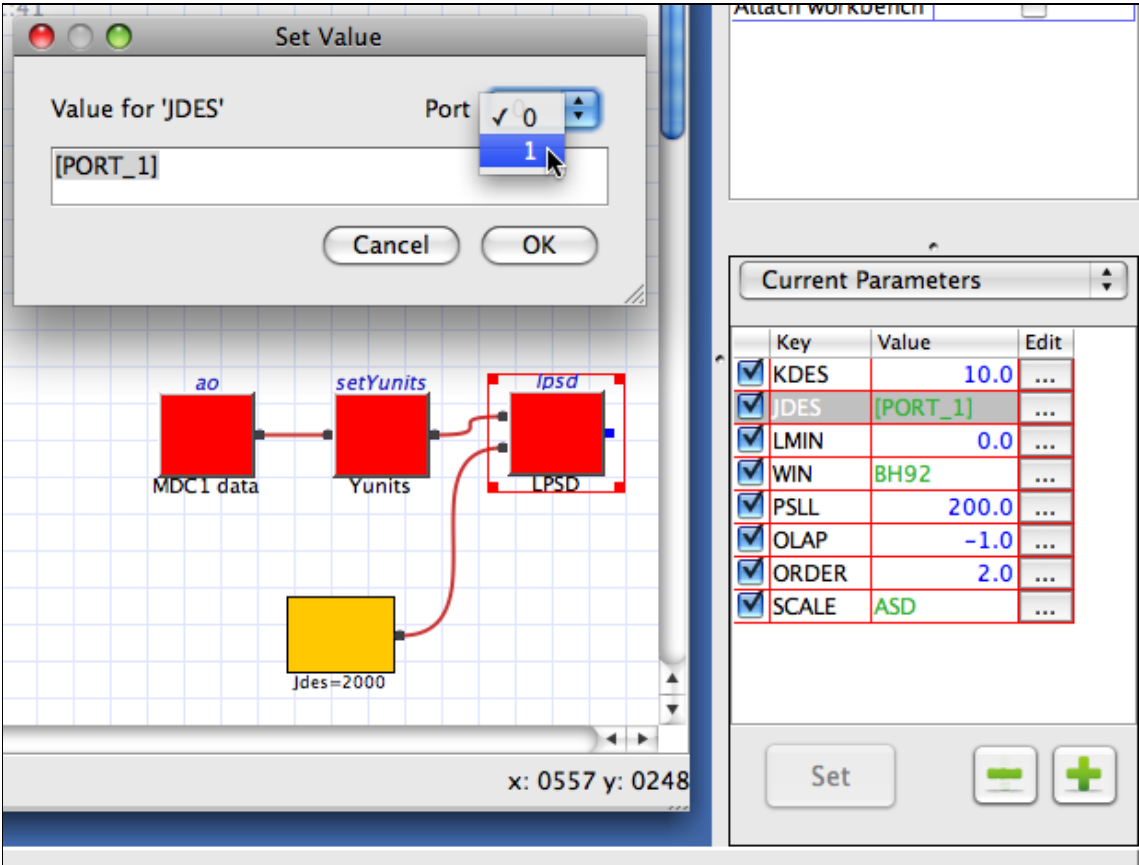
Key	Value	Description
JDES	2000	Compromising execution time and resolution
KDES	10	Slightly less than the default value
LMIN	0	The default value

In order to learn features of the GUI, let's introduce a "MATBlock" by right-clicking on some empty

part of the canvas and selecting "Add block ...->MATBlock". This provides the possibility to have defined numerical values to be passed to different blocks so we can change the value and be sure it's applied to all sensible blocks. As an example, we can type the value for the frequency bins: 2000. To do that, either we double click on the block or, with the block selected, we enter the values in the Property Table located in the upper-right corner of the workbench:



That's not all though. We need to connect the MATBlock to the ao/lpsd block, so we go ahead and right-click on the ao/lpsd block, select "Add input". Now we can actually connect the MATBlock to the ao/lpsd block; after that, let's move to the Parameter Set table, double-click on the "Edit" entry corresponding to the "JDES" key, and select the now available "PORT_1" on the drop-down list, or type "PORT_1" in the box. As an alternative to the last step, we can also type "PORT_1" in the "Value" entry.



Plot the results

As we did in previous topics, let's add one ao/iplot block to the output of the ao/lpsd block; additionally, let's also have another ao/iplot block for displaying the IFO time series.

We make a little additional exercise: let's add one block of the type

plist

As usual, to add parameters, just hit the "+" sign, so we can define some "key" and "value" properties:

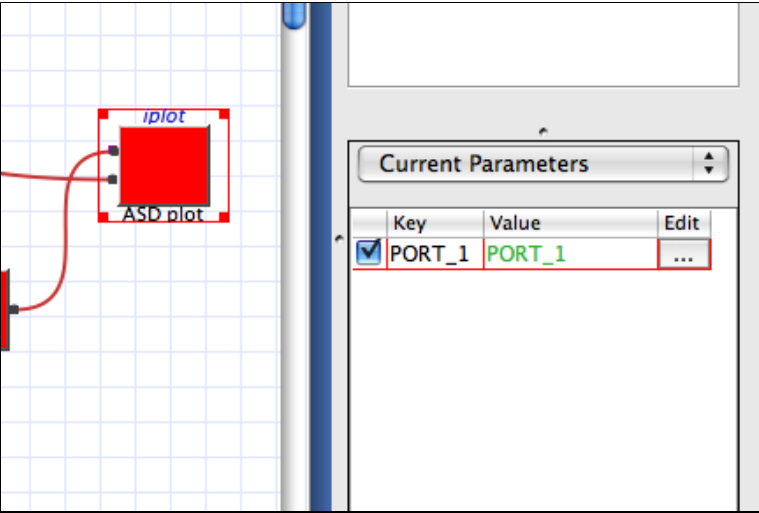
Key	Value	Description

COLORS

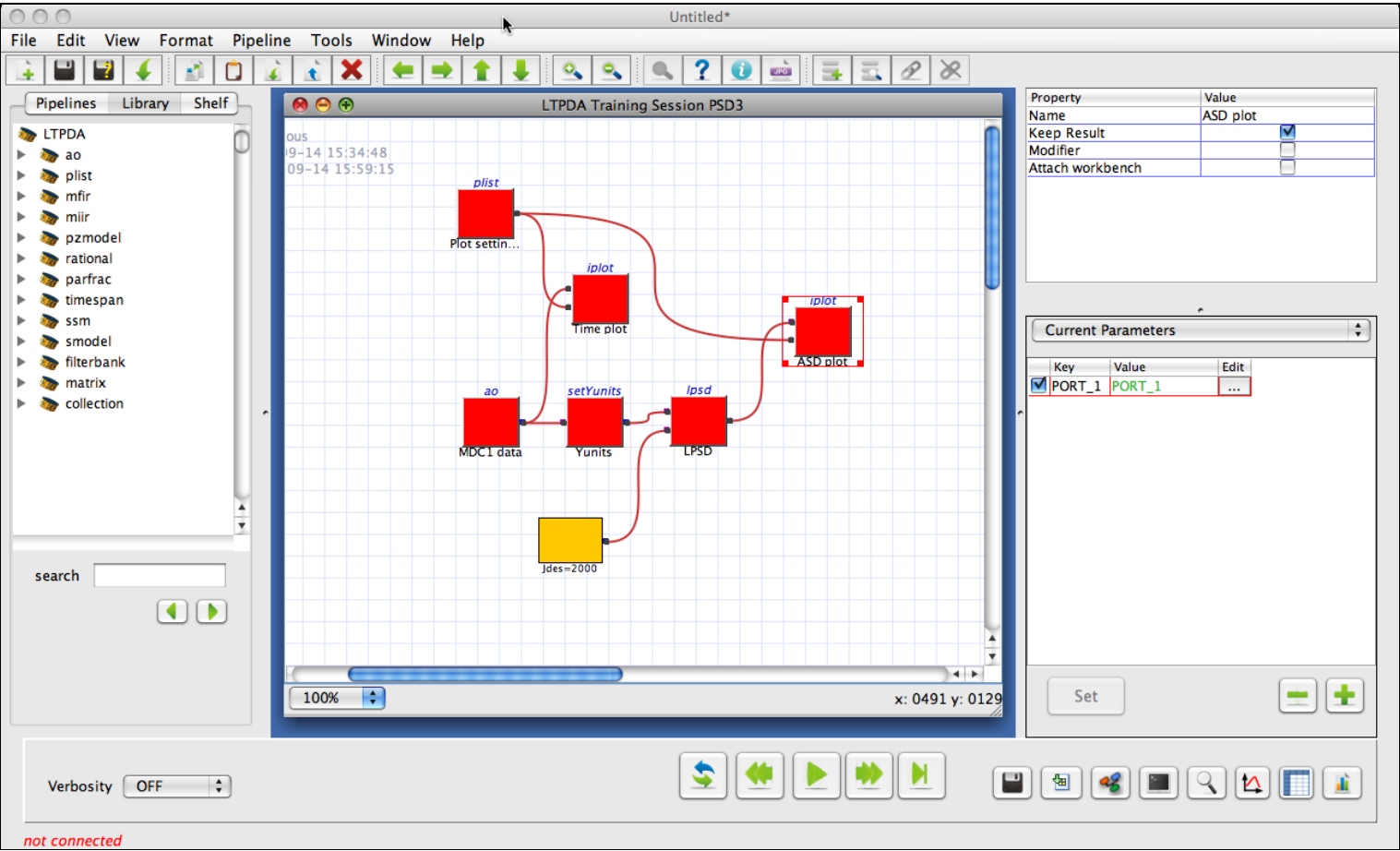
`{[0 0 1],[1 0 0]}`

Setting x1 to blue and x12 to red

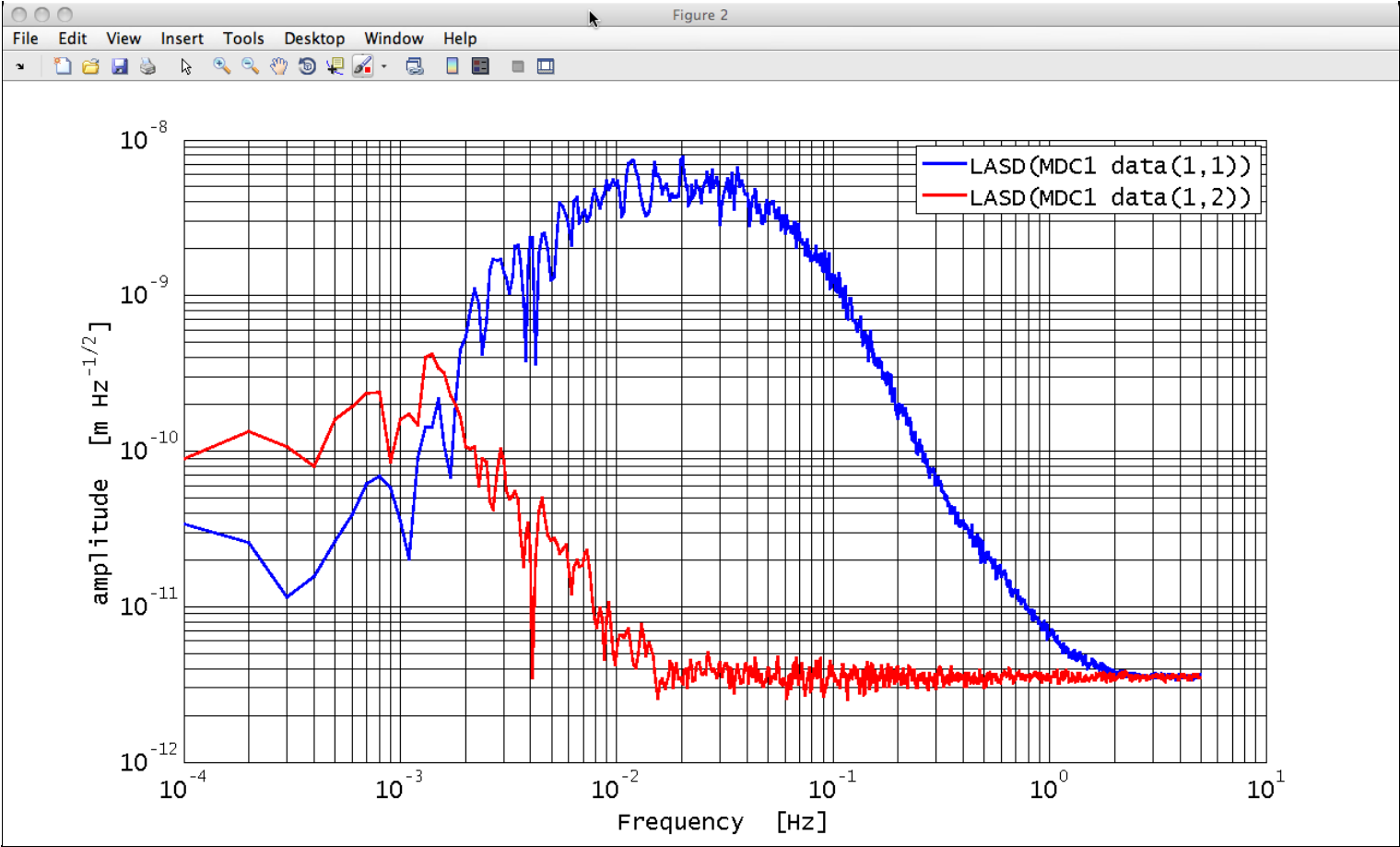
Now we can input this `plists` to the `iplot` blocks, by adding an input to them, then going into the Parameter Set area, hitting the "+" button, and selecting "PORT_1" for both the "key" and "value" entries. Like:



Ready for execution then ... and that's how the workbench should look like:



The execution may take some while ... and here are the results:



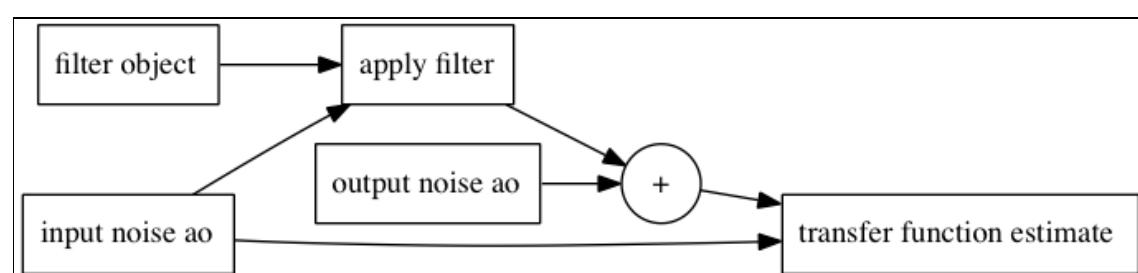
Empirical Transfer Function estimation

Let's run this exercise on empirical estimation of Transfer Functions on the Matlab terminal.

The idea of the exercise is the following:

1. simulate some white noise $x(t)$
2. build a band-pass filter F
3. pass the input noise $x(t)$ through the filter and add some more noise $y_n(t)$ at the output so to have $y = F*x(t) + y_n(t)$
4. evaluate and plot the transfer function $x \rightarrow y$

In a flow diagram, the representation is as follows:



The command-line sequence is the following:

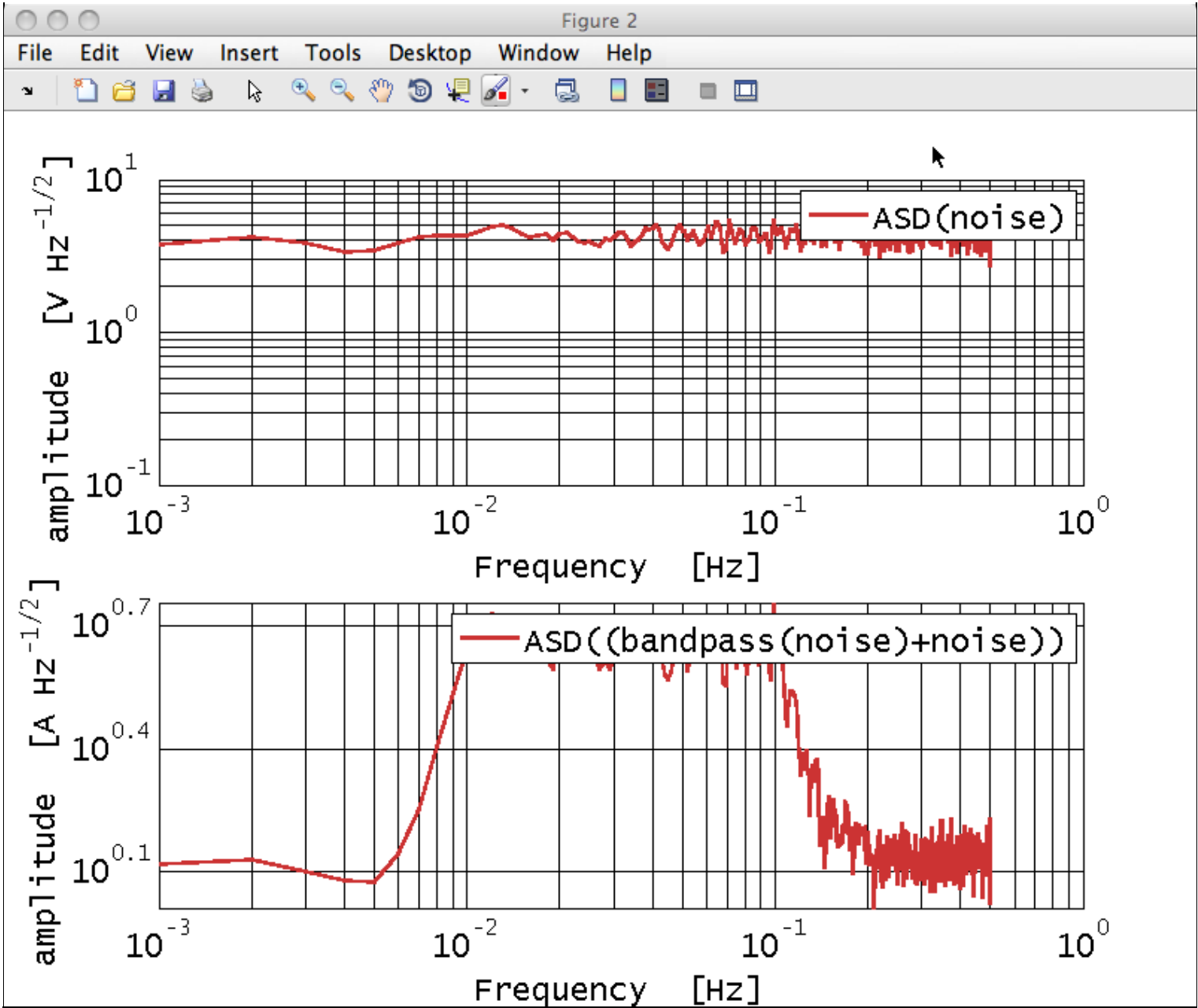
```

%% General definitions
nsecs = 10000;
fs = 1;
%% Input noise
x = ao(plist('waveform', 'noise', 'sigma', 3, 'fs', fs, 'nsecs', nsecs, 'yunits', 'V'))

%% Filter
bp_filter = miir(plist('type', 'bandpass', 'fc', [0.01 0.1], 'fs', 1, 'order', 3, 'iunits', 'V',
'ounits', 'A'))
xf = simplifyYunits(filter(x, bp_filter))

%% Output noise
yn = ao(plist('waveform', 'noise', 'sigma', 1, 'fs', fs, 'nsecs', xf.nsecs, 'yunits', 'A'))
y = xf + yn

%% Plotting input and output noise
xx = psd(x, plist('scale', ...
                  'ASD', ...
                  'nfft', 1000))
yy = psd(y, plist('scale', 'ASD', ...
                  'nfft', 1000))
iplot(xx, yy, plist('Arrangement', 'subplots', 'YRanges', {[1e-1 1e1], [1e-2 1e2]}));
  
```



Now we can proceed with the call to the `ao/tfe` method. The parameter list is very similar to the one employed for the other spectral estimators:

Key	Value	Description
NFFT	1000	The number of samples defining the length of the window to apply
WIN	'BH92'	Or a different one, if you want.
OLAP	-1	Overlap will be chosen based on the window properties
ORDER	0	Segment-wise detrending up to order 0

The command line is the following:

```
% Estimate the x->y transfer function
tfx = tfe(x, y, plist('nfft', 1000, 'win', 'BH92', 'olap', -1, 'order', 0));
```

We also would like to evaluate the expected transfer function $x \rightarrow y$, which is obviously the filter

transfer function, or response. This can be calculated by means of the

miir/resp

method. A detailed description of digital filtering is available in the User Manual dedicated [section](#) and will be touched upon in [this](#) topic; here let's just use the simplest form, where the needed parameter is a list of the frequency to evaluate the response at:

Key	Value	Description
F	tfxy.x	a vector of frequency values or an ao whereby the x-axis is taken for the frequency values

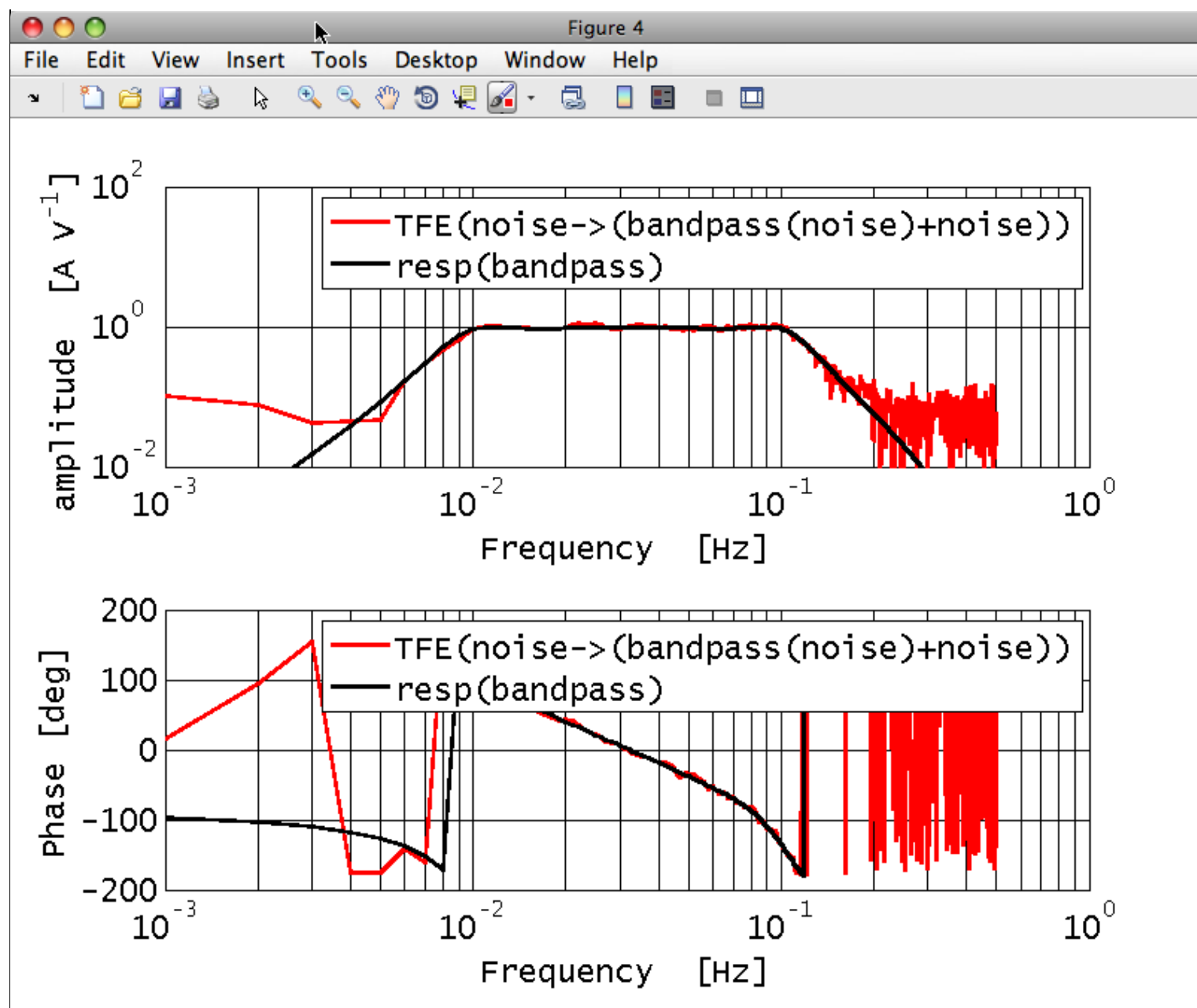
So we can just pass the x field of the fsdata ao containing the transfer function estimate. However, we can also just pass the AO itself. In which case, the resp function will take the X values from the AO.

The command line is the following:

```
%% Evaluate the expected x->y transfer function
rf = resp(bp_filter, plist('f', tfxy))
```

Eventually let's look at the results:

```
%% Plotting estimated and expected transfer functions
ipplot(tfxy, rf, plist('colors',[1 0 0],[0 0 0]','YRanges',[1e-2 1e2],[-200 200])))
```



◀ Example 3: Log-scale PSD on MDC1 data

IFO/Temperature Example – Spectral Analysis ▶

©LTP Team

IFO/Temperature Example – Spectral Analysis

Loading the consolidated data sets from topic2

In the last topic you should have saved your consolidated data files as

- ifo_temp_example/temp_fixed.xml
- ifo_temp_example/ifo_fixed.xml

In order to proceed with the spectral analysis, we need to use the `ao` constructor, with the set of parameters "From XML File". The key parameters are:

Key	Value	Description
FILENAME	'ifo_temp_example/ifo_fixed.xml'	The name of the file to read the data from.
FILENAME	'ifo_temp_example/temp_fixed.xml'	The name of the file to read the data from.

Hint: the command-line sequence may be similar to the following:

```
%% Get the consolidated data
% Using the xml format

T_filename = 'ifo_temp_example/temp_fixed.xml';
x_filename = 'ifo_temp_example/ifo_fixed.xml';

pl_load_T = plist('filename', T_filename);
pl_load_x = plist('filename', x_filename);

% Build the data aos
T = ao(pl_load_T);
x = ao(pl_load_x);
```

Estimating the PSD of the signals

To perform the PSD estimation, you can use the method

`ao/lpsd`

Hint: the command-line sequence may be similar to the following:

```
%% plists for spectral estimations

%% PSD
x_psd = lpsd(x)
x_psd.setName('Interferometer');

T_psd = lpsd(T)
T_psd.setName('Temperature');

% Plot estimated PSD
pl_plot = plist('Arrangement', 'subplots', 'LineStyles', {'-', '-'}, 'Linecolors', {'b', 'r'});
```

```
iplot(sqrt(x_psd), sqrt(T_psd), pl_plot);
```

Reducing time interval

Looking at the output of the analysis it is easy to recognize in the IFO PSD the signature of some strong "spike" in the data. Indeed, if we plot the x data, we can find it around $t = 40800$. There is also a leftover from the filtering process performed during consolidation right near to the last data.

We can then try to estimate the impact of the "glitch" by comparing the results we obtain by passing to `ao/lpsd` a reduced fraction of the data.

In order to select a fraction of the data we use the method:

```
ao/split
```

The relevant parameters for this method are listed here, together with their recommended values:

Key	Value	Description
SPLIT_TYPE	'interval'	The method for splitting the <code>ao</code>
START_TIME	<code>x.t0 + 40800</code>	A time-object to start at
END_TIME	<code>x.t0 + 193500</code>	A time-object to end at

Notice that in order to perform this action, we access one property of the `ao` object, called "t0". The call to the `t0` methods gives as an output an object of the `time` class. Additionally, it's possible to directly add a number (in seconds) to obtain a new time object.

Hint: the command-line sequence may be similar to the following:

```
%% Skip some IFO glitch from the consolidation
pl_split = plist('start_time', x.t0 + 40800, ...
    'end_time', x.t0 + 193500);
x_red = split(x, pl_split);
T_red = split(T, pl_split);
```

Note: you can also use the parameter 'times' for split to specify times relative to the first sample. In this case:

```
plist('times', [40800 193500]).
```

would work.

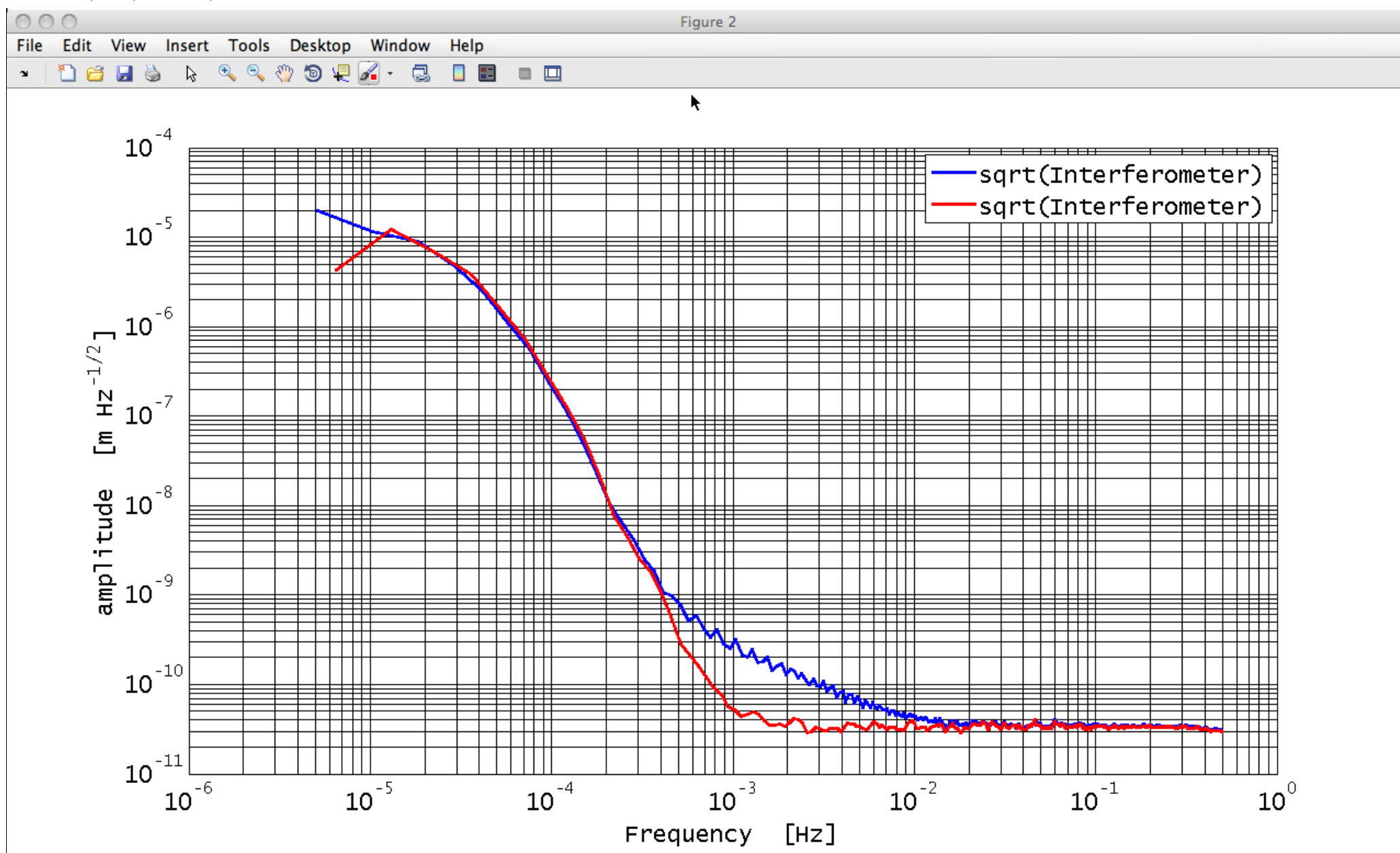
And we can go proceed, evaluating 2 more `aos` to compare the effect of skipping the "glitch". After doing that, we can plot them in comparison.

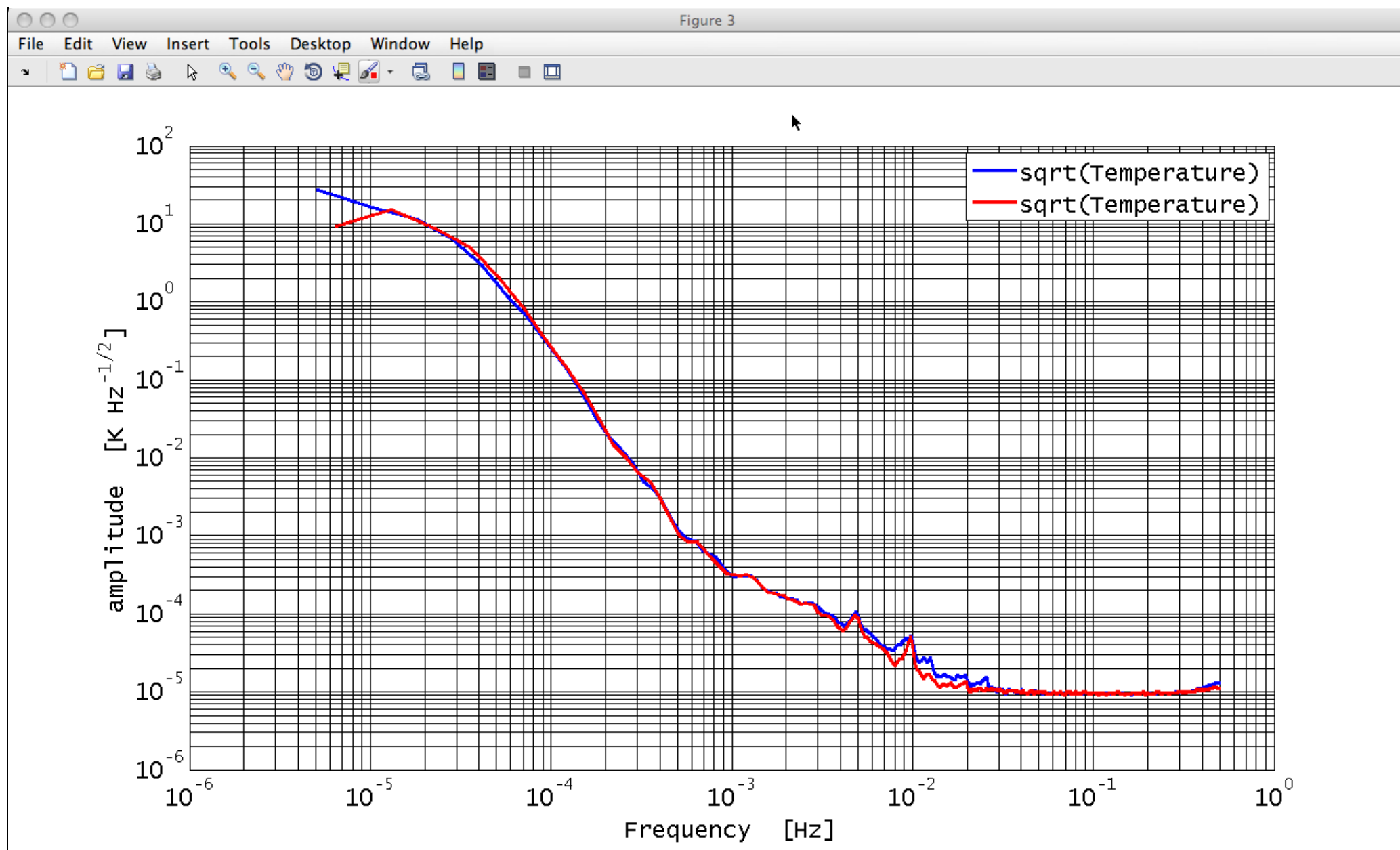
Hint:

```
%% PSD
x_red_psd = lpsd(x_red);
x_red_psd.setName('Interferometer');
T_red_psd = lpsd(T_red)
```

```
T_red_psd.setName('Temperature');

% Plot estimated PSD
pl_plot = plist('Arrangement', 'stacked' 'LineStyles', {'-', '-'}, 'Linecolors', {'b', 'r'});
iplot(sqrt(x_psd), sqrt(x_red_psd), pl_plot);
iplot(sqrt(T_psd), sqrt(T_red_psd), pl_plot);
```





Estimating the cross-spectra

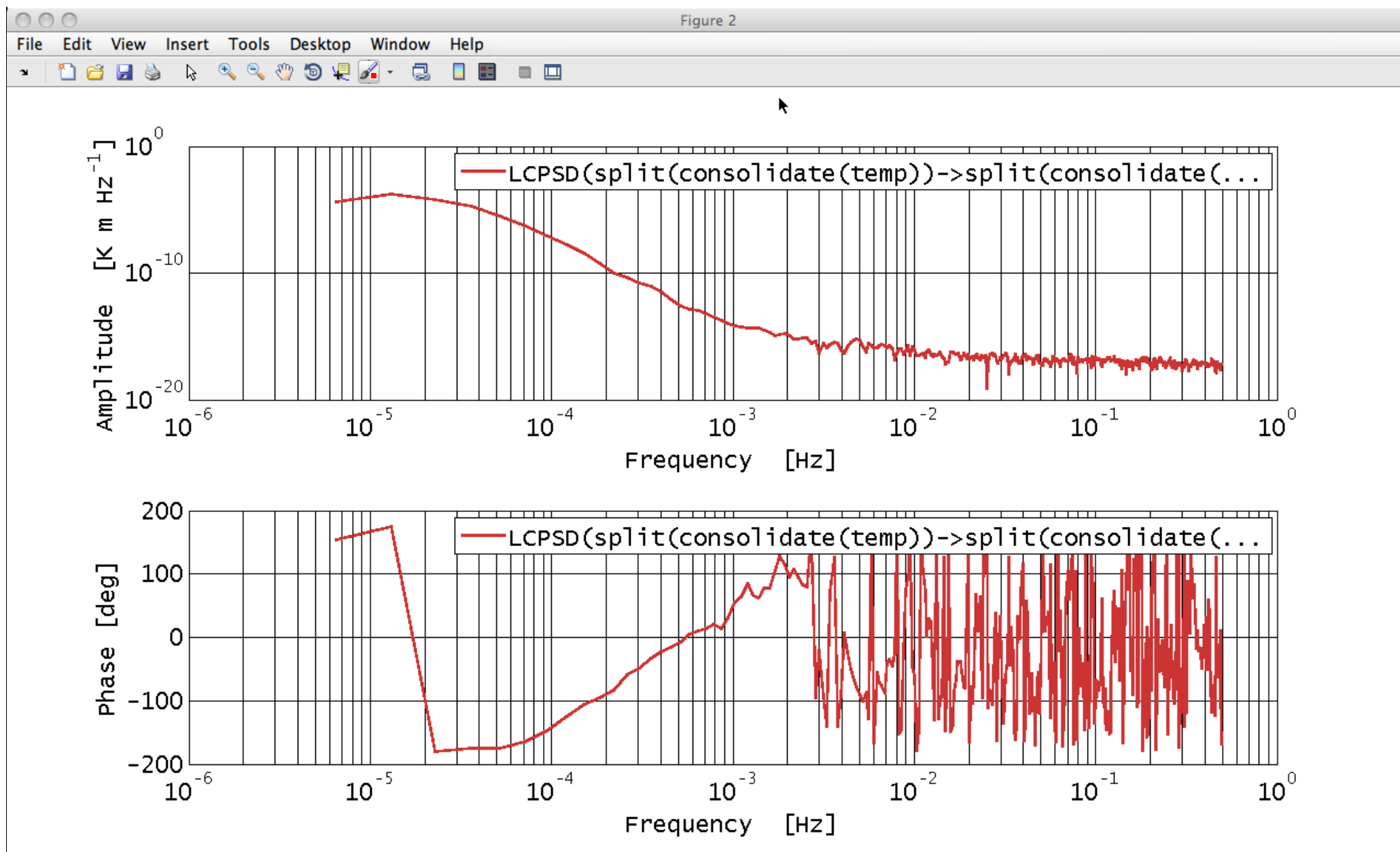
We can now proceed and use the

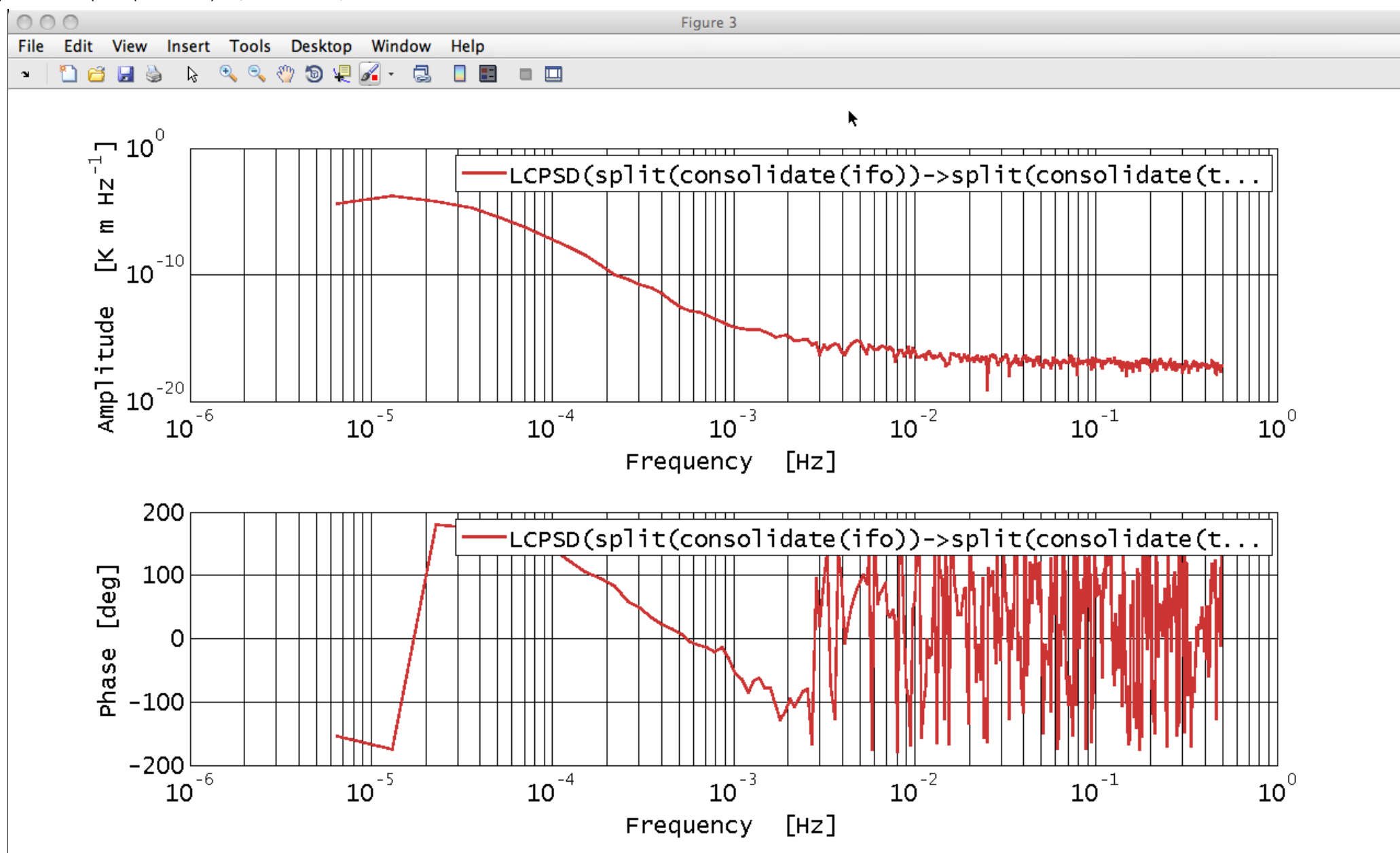
ao/lcpsd

method to evaluate the cross-spectra of the signals, employing a shorter window in order to reduce the scatter of the estimated values.

Hint:

```
%% CPSD estimate
CTx = lcpsd(T_red, x_red);
CxT = lcpsd(x_red, T_red);
% Plot estimated CPSD
iplot(CTx);
iplot(CxT);
```



As expected, there is a strong low-frequency correlation between the IFO data x and the temperature data τ .

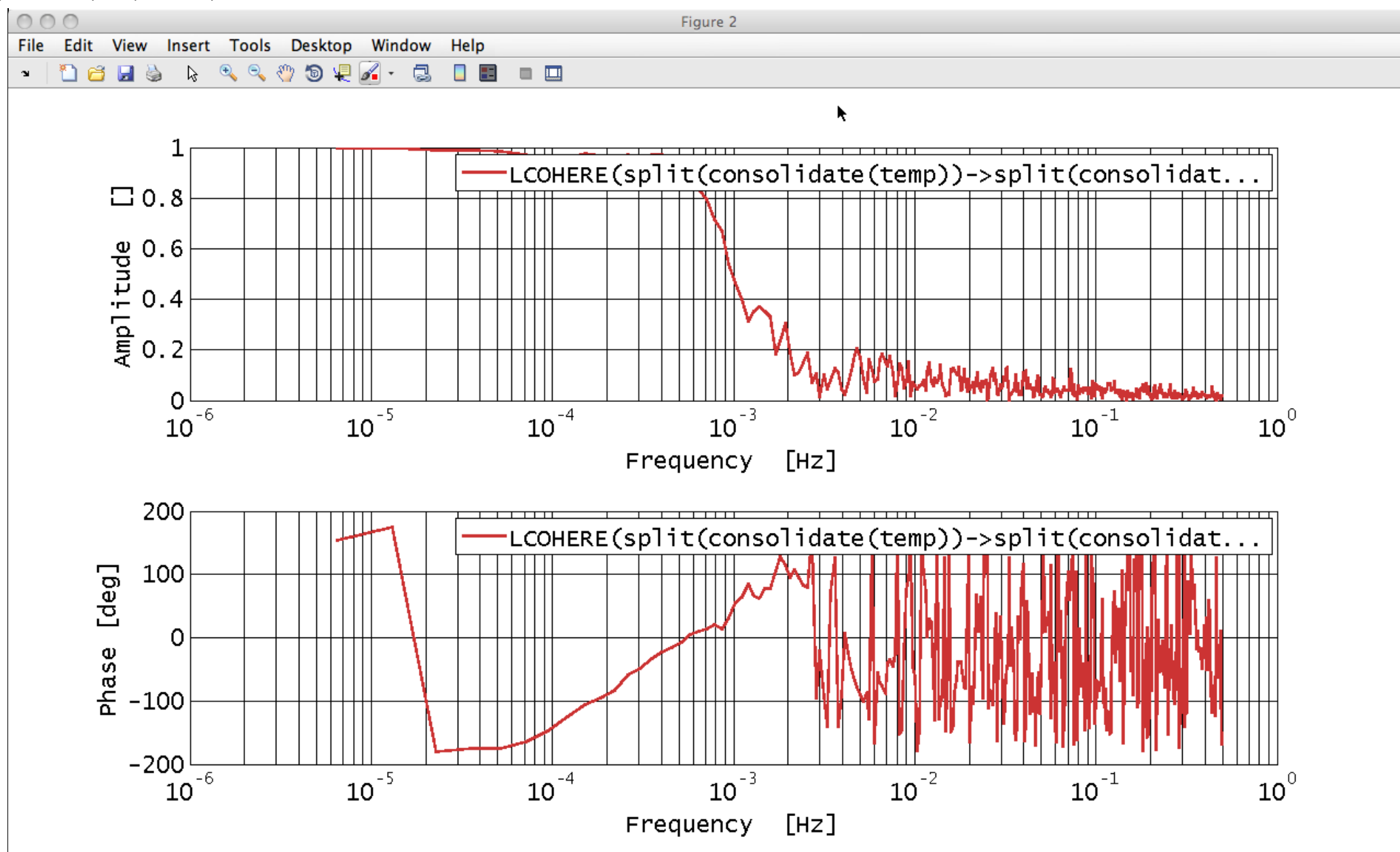
Estimating the cross-coherence

Similarly, we can now proceed and use the

`ao/lcohere`

method to evaluate the cross-coherence of the signals. The output will be a [2x2] matrix of `aos`, and we want to look at one of the off-diagonal terms:

```
%% Coherence estimate  
coh = lcohere(T_red, x_red);  
% Plot estimated cross-coherence  
ipplot(coh, plist('YScales', 'lin'))
```



The coherence approaches 1 at low frequency.

Estimating the transfer function of temperature

We want now to perform noise projection, trying to estimate the transfer function of the temperature signal into the interferometer output. In order to do that, we can use the

`ao/ltf`

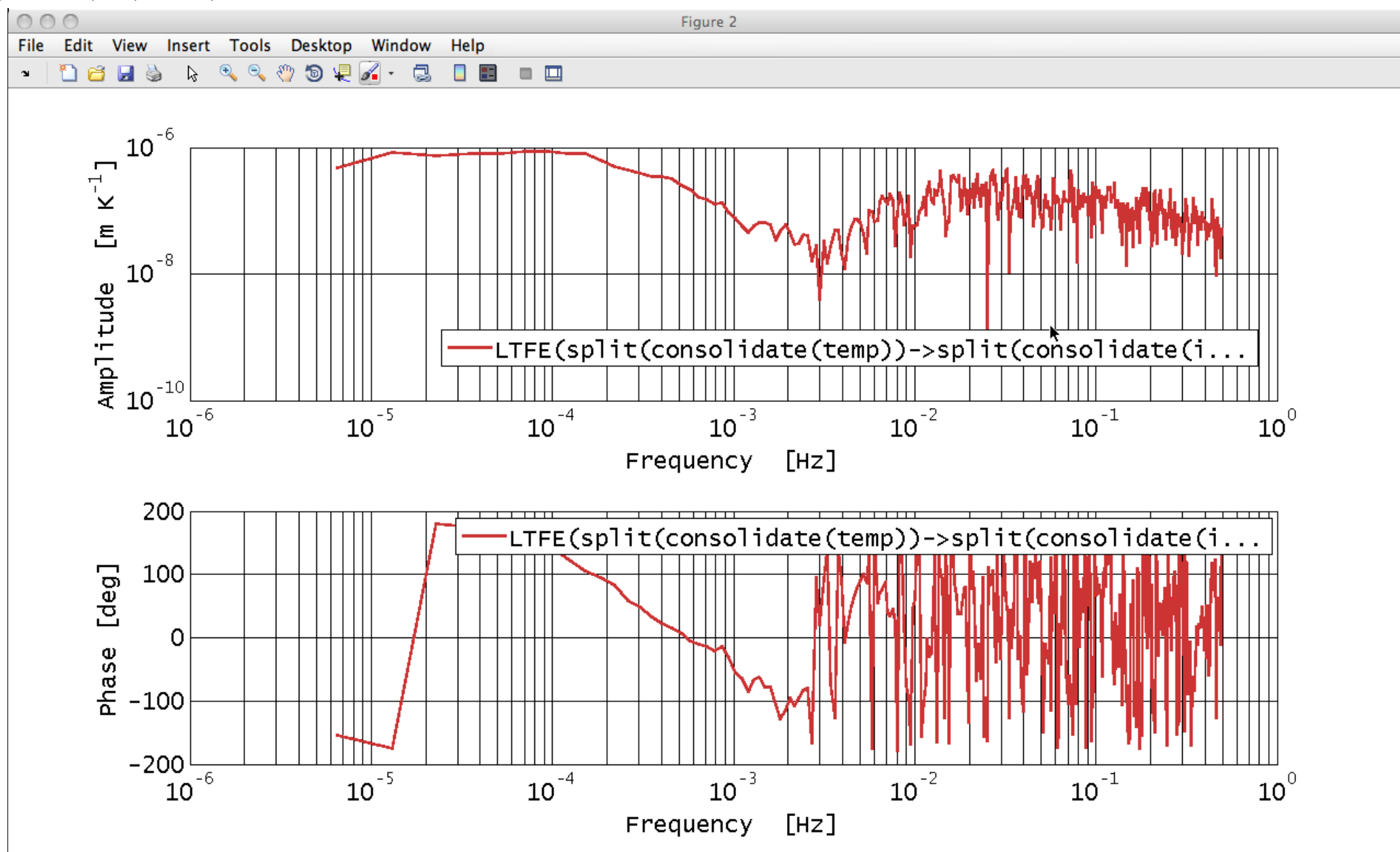
method.

We also want to plot this transfer function to check that the units are correct.

Hint:

```
%% transfer function estimate
tf = ltfe(T_red, x_red)

% Plot estimated TF
iplot(tf);
```



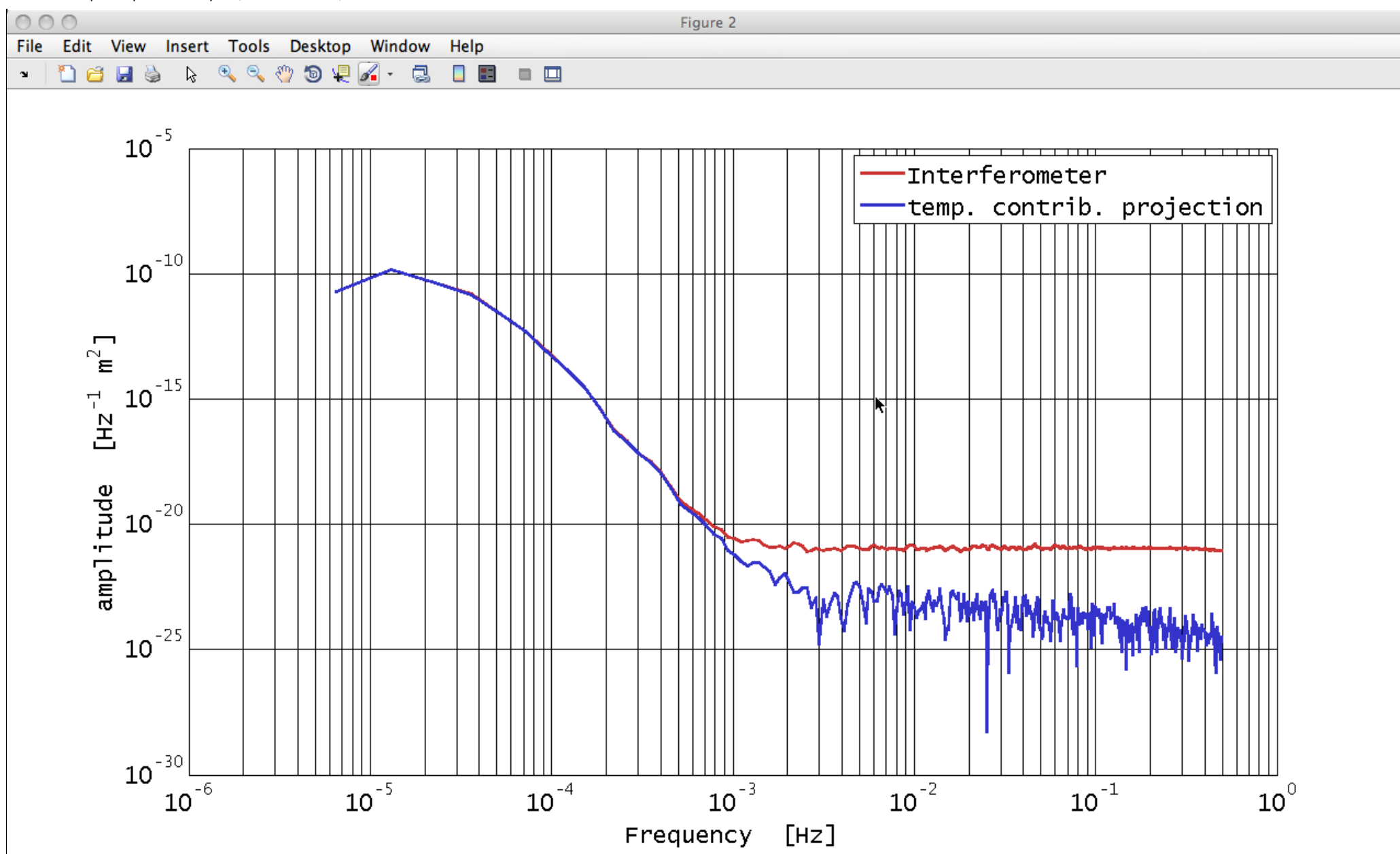
As expected, the transfer function $T \rightarrow \text{IFO}$ is well measured at low frequencies.

Noise projection

We can eventually perform the noise projection, estimating the amount of the noise in the IFO being actually caused by temperature fluctuations. It's a frequency domain estimate:

```
%% Noise projection in frequency domain  
proj = T_red_psd.*(abs(tf)).^2;  
proj.simplifyYunits;  
proj.setName('temp. contrib. projection')  
%% Plotting the noise projection in frequency domain  
iplot(x_red_psd, proj);
```

The contribution of the temperature fluctuations is clearly estimated.



Saving the results

Let's finish this section of the exercise by saving the results on disk, in xml format. We want to keep the results about the Power Spectral Density and Transfer Function Estimates, at least.

Hint: the command-line sequence may be similar to the following:

```
%% Save the PSD data
% Plists for the xml format

pl_save_x_PSD = plist('filename', 'ifo_temp_example/ifo_psd.xml');
pl_save_T_PSD = plist('filename', 'ifo_temp_example/T_psd.xml');

pl_save_xT_CPSD = plist('filename', 'ifo_temp_example/ifo_T_cpsd.xml');
pl_save_xT_cohere = plist('filename', 'ifo_temp_example/ifo_T_cohere.xml');

pl_save_xT_TFE = plist('filename', 'ifo_temp_example/T_ifo_tf.xml');
```

or

```
% Plists for the mat format

pl_save_x_PSD = plist('filename', 'ifo_temp_example/ifo_psd.mat');
pl_save_T_PSD = plist('filename', 'ifo_temp_example/T_psd.mat');

pl_save_xT_CPSD = plist('filename', 'ifo_temp_example/ifo_T_cpsd.mat');
pl_save_xT_cohere = plist('filename', 'ifo_temp_example/ifo_T_cohere.mat');

pl_save_xT_TFE = plist('filename', 'ifo_temp_example/T_ifo_tf.mat');
```

and

```
% Save
x_red_psd.save(pl_save_x_PSD);
T_red_psd.save(pl_save_T_PSD);
CxT.save(pl_save_xT_CPSD);
coh.save(pl_save_xT_cohere);
tf.save(pl_save_xT_TFE);
```

Topic 4 – Transfer function models and digital filtering

Training session 4 is a tutorial of how to build transfer function models. These transfer functions can be either defined by the user or derived from an input/output time series. The tutorial also shows how to extract digital filters from the transfer functions and how to use these them to filter data. The topic is divided as follows:

- build transfer functions models in s domain
- model system and basic operation with models
- extract digital filters from transfer functions
- build digital filters
- filter data

◀ IFO/Temperature Example – Spectral Analysis Create transfer function models in s domain ▶

©LTP Team

Create transfer function models in s domain

Building transfer function models

In this first part we will focus on the creation of transfer function models to then see how these can be translated into digital filters and applied to data. In the LTPDA toolbox, there are three possible ways to express a transfer function which can be more or less suitable depending on your particular application:

- Pole zero models
- Partial fraction models
- Rational models

Pole zero model representation

Creating a pole zero model

Let's build a pole zero model with the following characteristics:

Key	Value
GAIN	5
POLES	(f = 10 Hz, Q = 2)
ZEROS	(f = 1 Hz), (f = 0.1 Hz)

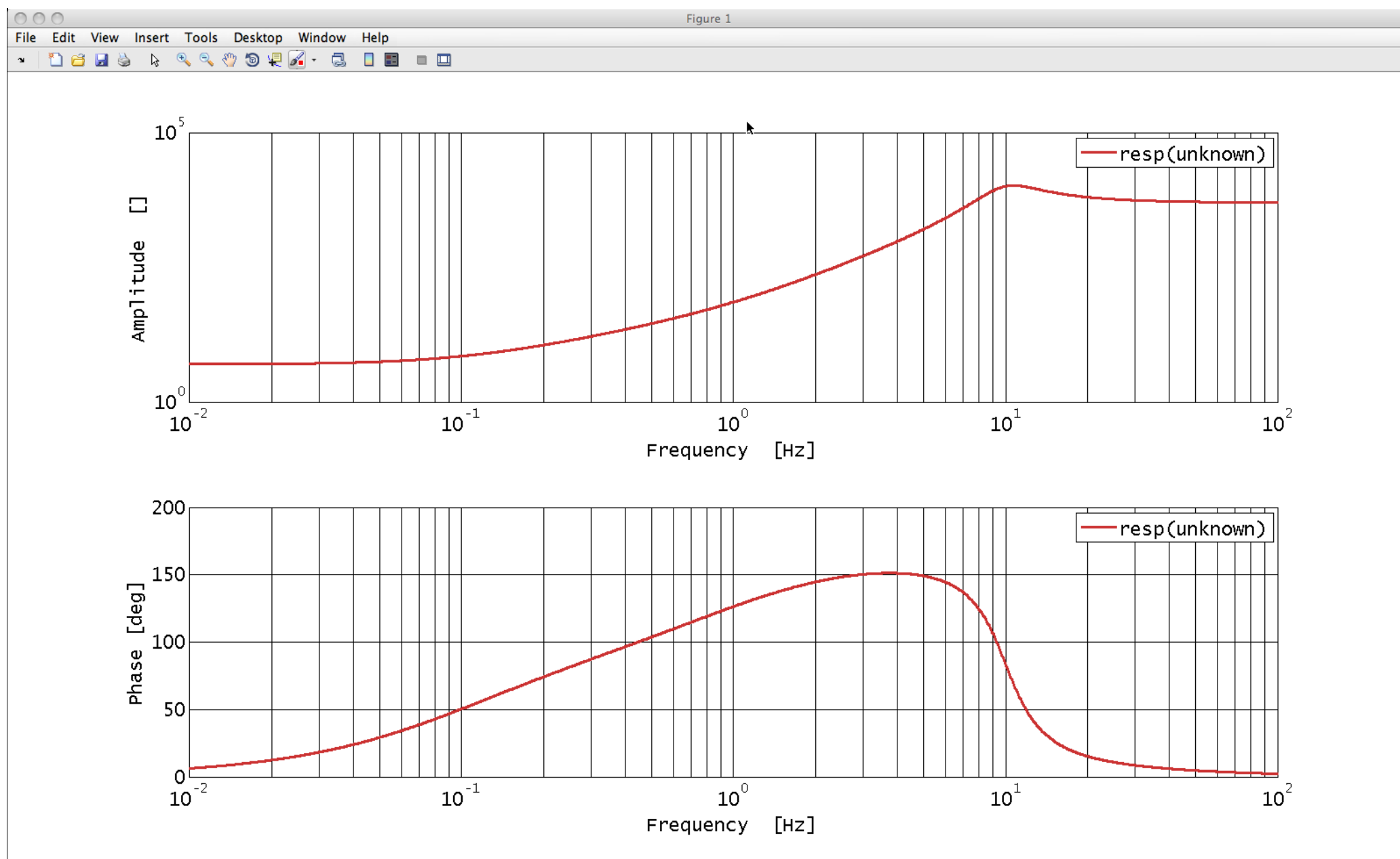
Our pole zero model has a pole at 10 Hz with a quality factor of $Q = 2$, and two zeros, one at 1 Hz, one at 0.1 Hz.

```
m = pzmodel(5, [10 2], {1, 0.1})
```

We can easily obtain the response of this transfer function by using the `resp` method.

```
resp(m)
```

The result is the figure shown below.



About the quality factor Q

The quality factor notation comes from the mechanical analogy of the harmonic oscillator. In short, we can classify systems in terms of Q as

- **Underdamped system ($Q > 0.5$):** the system is described by a complex conjugate pair that represents an oscillating solution
- **Critically damped system ($Q = 0.5$):** the system is described by two equal real poles. The system decays exponentially to equilibrium faster than in any other case.
- **Overdamped system ($Q < 0.5$):** the system is described by two real poles. The response is an exponential decay.

Partial fraction representation

Creating a partial fraction model

We can also specify a transfer function as a sum of partial fraction using the `parfrac` object. This will define a transfer function with the following shape (you can take a look at `help parfrac`):

```
% DESCRIPTION: PARFRAC partial fraction representation of a transfer function.
%
%
%      R(1)      R(2)      R(n)
%      H(s) =  ---- + ---- + ... + ---- + K(s)
%      s - P(1)  s - P(2)  s - P(n)
%
%
```

In this case we will need to define an array of residues (R), an array of poles (P) and direct terms K. All of them either real or complex, for example the ones defined by:

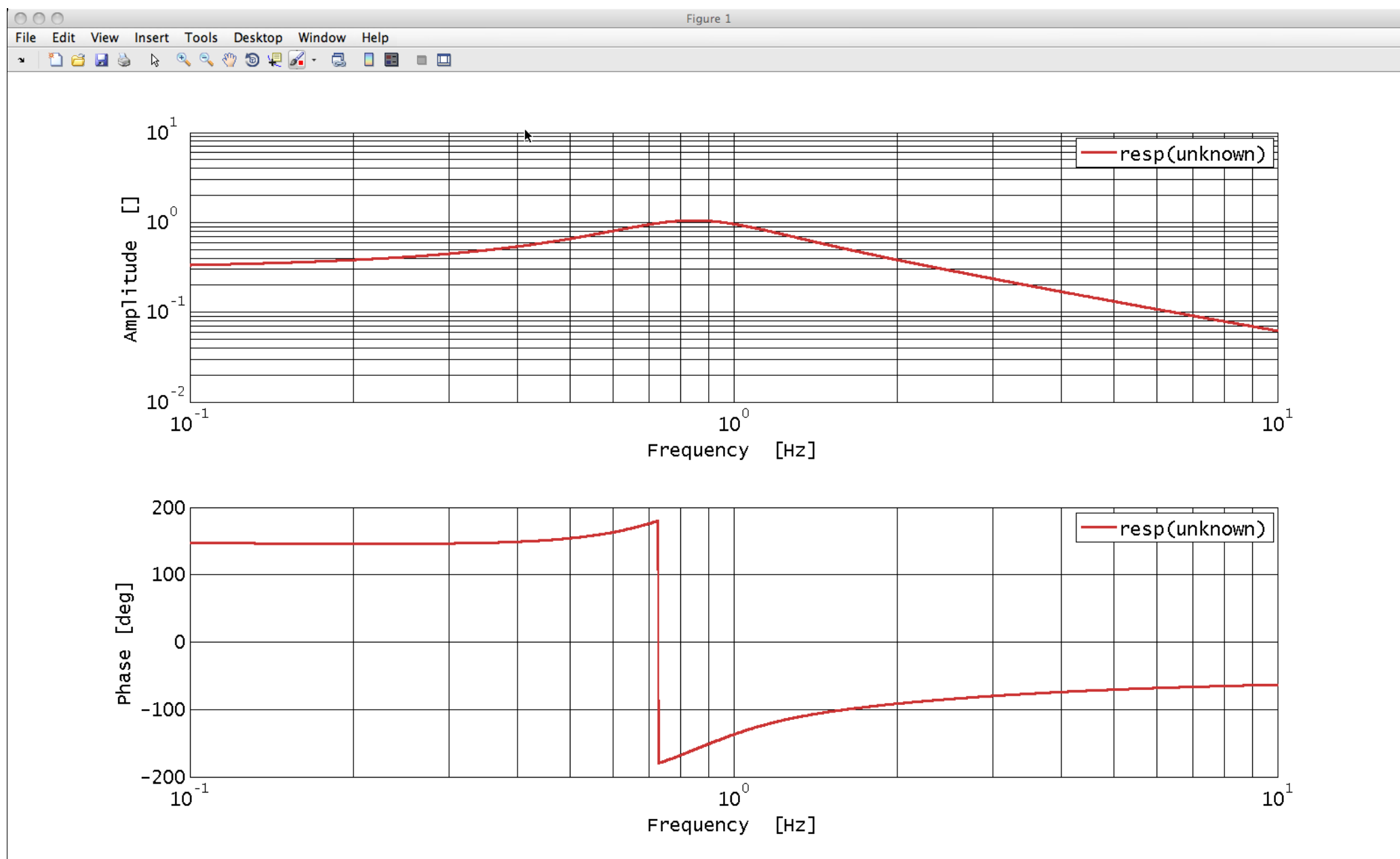
- $R(1) = 1+2i$, $R(2) = 2$
- $P(1) = 10+i$, $P(2) = 2+5*i$
- $K = 0$

which can be inserted directly in the `parfrac` constructor:

```
m = parfrac([1+2i; 2],[10+i; 2+5*i],[])
```

We can now take a look at the response of the transfer function that we have just created. We can specify the frequency region in what we are interested with the `plist` of the `resp` function. For example, we would like to evaluate our model from $f1 = 0.1$ Hz to $f2 = 10$ Hz.

```
resp(m,plist('f1',0.1,'f2',10))
```



◀ Pole zero model representation

Rational representation ▶

Rational representation

Creating a rational model

The last possibility to build a transfer function model is to use the `rational` constructor, which implements a transfer function as a quotient of polynomials

```
% DESCRIPTION: RATIONAL rational representation of a transfer function.
%
%      a(1)s^m + a(2)s^{m-1} + ... + a(m+1)
%      H(s) = -----
%      b(1)s^n + b(2)s^{n-1} + ... + b(n+1)
%
```

For example if the numerator and denominator coefficients are given by:

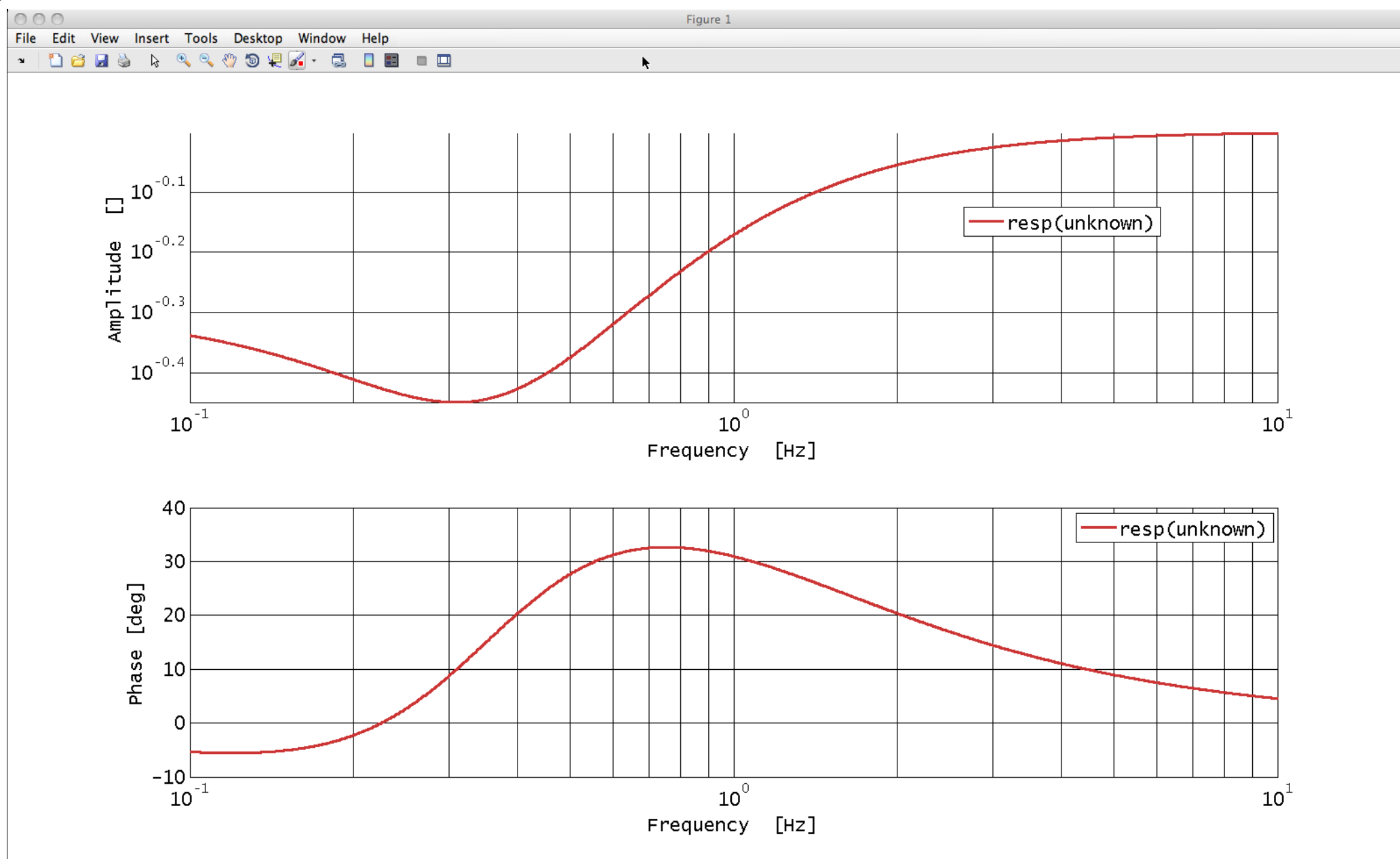
- `num = [1 3 5]`
- `den = [1 8 10]`

then we need to write down the following

```
m = rational([1 3 5],[1 8 10])
```

And now, as with the previous models, we can evaluate the response in a given frequency region

```
resp(m,plist('f1',0.1,'f2',10))
```



Transforming models between representations

Transforming between different representations

The LTPDA toolbox allows you to go from one representation to the other. However, in the current version (v2.1), only the transformations shown in the following table are allowed

	Pole/Zero	Rational	Partial Fraction
Pole/Zero		✓	✗
Rational	✓		✗
Partial Fraction	✗	✗	

We can see how this work for the allowed transformation using the models that we have just created. To turn our `pzmodel` into a `rational` one, we just need to insert the first into the `rational` constructor

For example, if we consider again our `pzmodel`

```
pzm = pzmodel(5, [10 2], {1, 0.1}, 'mymodel')
---- pzmodel 1 ----
  name: mymodel
  gain: 5
  delay: 0
  iunits: []
  ounits: []
  description:
    UUID: 9206f503-69c9-4c20-963e-b5604e59b450
  pole 001: (f=10 Hz, Q=2)
  zero 001: (f=1 Hz, Q=NaN)
  zero 002: (f=0.1 Hz, Q=NaN)
  -----
```

To transform it into its rational equivalent we just need to give it as a input to the `rational` constructor

```
rat = rational(pzm)
---- rational 1 ----
  model: rational(mymodel)
  num: [1.26651479552922 8.75352187005424 5]
  den: [0.000253302959105844 0.00795774715459477 1]
  iunits: []
  ounits: []
  description:
    UUID: 04b7990b-293c-41a7-a3c9-a311e6f3974b
  -----
```

The transformation returns again the first `pzmodel` however the name property allows us to follow the operations that we've been performing to this object.

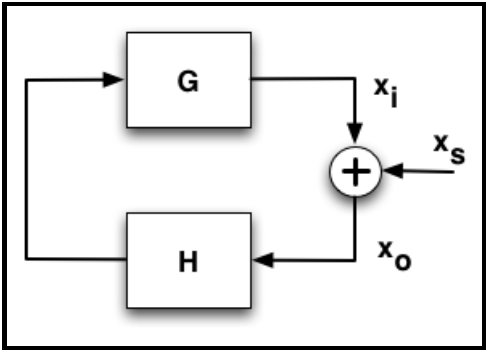
```
pzm2 = pzmodel(rat)
---- pzmodel 1 ----
    name: pzmodel(rational(mymodel))
    gain: 5
    delay: 0
    iunits: []
    ounits: []
description:
    UUID: b2177fde-1465-4c5c-a94f-29d73d9aed67
pole 001: (f=10 Hz,Q=2)
zero 001: (f=1 Hz,Q=NaN)
zero 002: (f=0.1 Hz,Q=NaN)
-----
```

[← Rational representation](#)
[Modelling a system →](#)

©LTP Team

Modelling a system

To show some of the possibilities of the toolbox to model digital system we introduce the usual notation for a closed loop model



In our example we will assume that we know the `pzmodel` of the filter, `H`, and the open loop gain (OLG). These are related with the closed loop gain (CLG) by the following equation

$$\frac{x_o}{x_s} = \text{CLG} = \frac{1}{1 - \text{OLG}} = \frac{1}{1 - H \cdot G}$$

We want to determine `H` and `CLG`. We would also like to find a digital filter for `H`, but we will deal with this in the following section.

Loading the starting models

Imagine that we have somehow managed to find the following model for `OLG`

Key	Value
GAIN	4e6
POLES	1e-6

then we can create a `pzmodel` with these parameters as follows

```
OLG = pzmodel(4e6,1e-6,[], 'OLG')
---- pzmodel 1 ----
  name: OLG
  gain: 4000000
  delay: 0
  iunits: []
  ounits: []
description:
  UUID: 3d8bce32-a9a9-4e72-ab4d-183da69a9b5d
  pole 001: (f=1e-06 Hz,Q=NaN)
-----
```

To introduce the second model, the one describing H , we will show another feature of the `pzmodel` constructor. We will read it from a LISO file, since this constructor accepts this files as inputs. We can then type

```
H = pzmodel('topic4/LISOfile.fil')
---- pzmodel 1 ----
      name: none
      gain: 10000000000
      delay: 0
      iunits: []
      ounits: []
description:
      UUID: e683b32f-4653-474e-b457-939d33aeb63c
pole 001: (f=1e-06 Hz,Q=NaN)
pole 002: (f=1e-06 Hz,Q=NaN)
zero 001: (f=0.001 Hz,Q=NaN)
-----
```

and we see how the constructor recognizes and translates the poles and zeros in the file. The model gets the name from the file but we can easily change it to have the name of our model

```
H.setName;
```

According to our previous definition we can get the `plant` by dividing the OLG by H . We can do so directly when dealing with `pzmodel` objects since multiplication and division are allowed for these objects, then

```
G = OLG/H
---- pzmodel 1 ----
      name: (OLG./H)
      gain: 0.004
      delay: 0
      iunits: []
      ounits: []
description:
      UUID: fcd166f7-d726-4d39-ad2e-0f3b7141415b
pole 001: (f=1e-06 Hz,Q=NaN)
pole 002: (f=0.001 Hz,Q=NaN)
zero 001: (f=1e-06 Hz,Q=NaN)
zero 002: (f=1e-06 Hz,Q=NaN)
-----
```

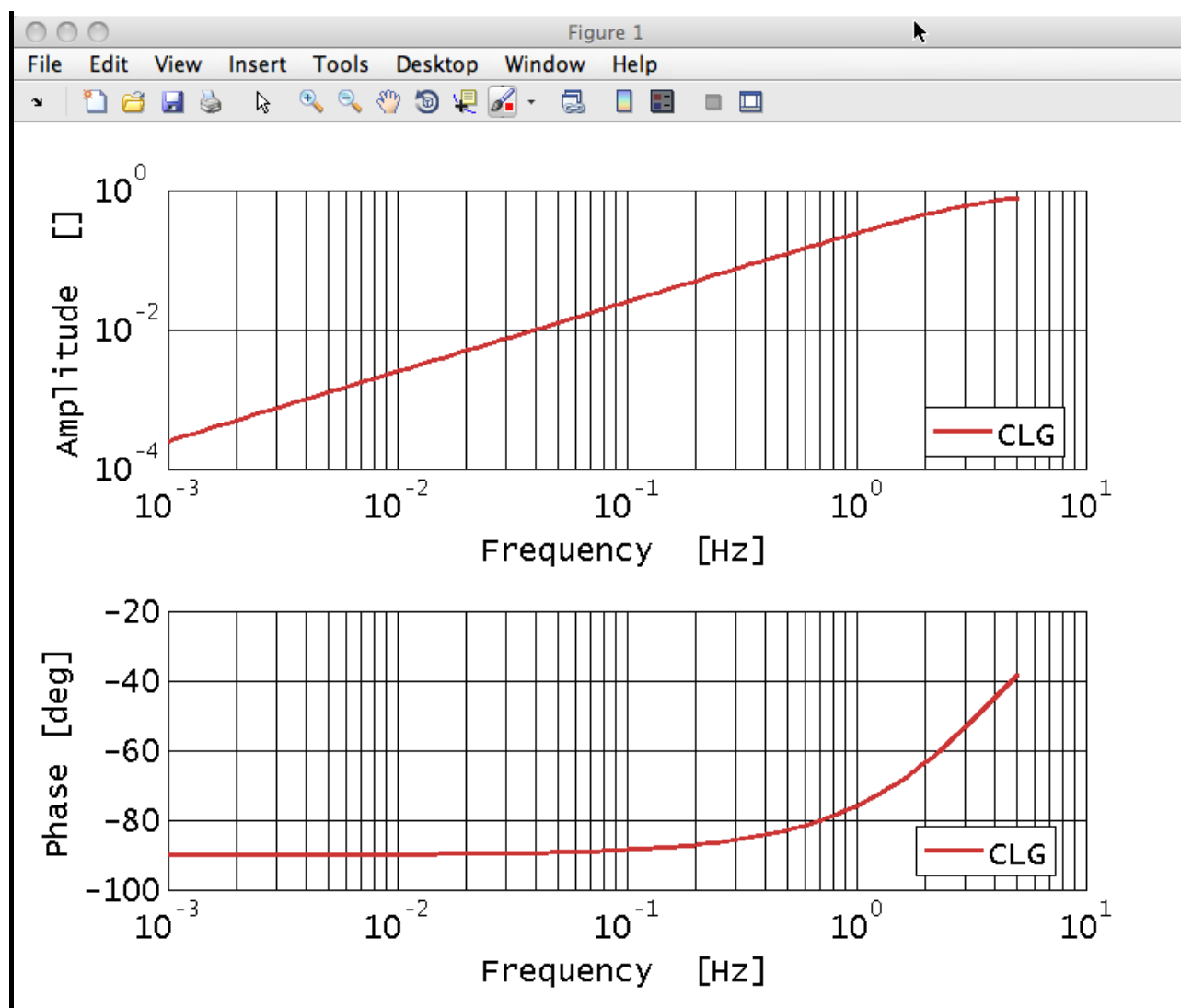
which we need to simplify to get rid of cancelling poles and zeros. We also set the model name here.

```
G.setName;
G.simplify
---- pzmodel 1 ----
      name: simplify(G)
      gain: 0.004
      delay: 0
      iunits: []
      ounits: []
description:
      UUID: c1883713-b860-4942-a127-e42ea565460f
pole 001: (f=0.001 Hz,Q=NaN)
zero 001: (f=1e-06 Hz,Q=NaN)
-----
```

The CLG requires more than a simple multiplication or division between models and we will not be able to derive a `pzmodel` for it. However, we can evaluate the response of this object as follows

```
pl = plist('f1',1e-3,'f2',5,'nf',100);
CLG = 1/(1-resp(OLG,pl));
CLG.setName;
```

which gives us an AO that we can plot

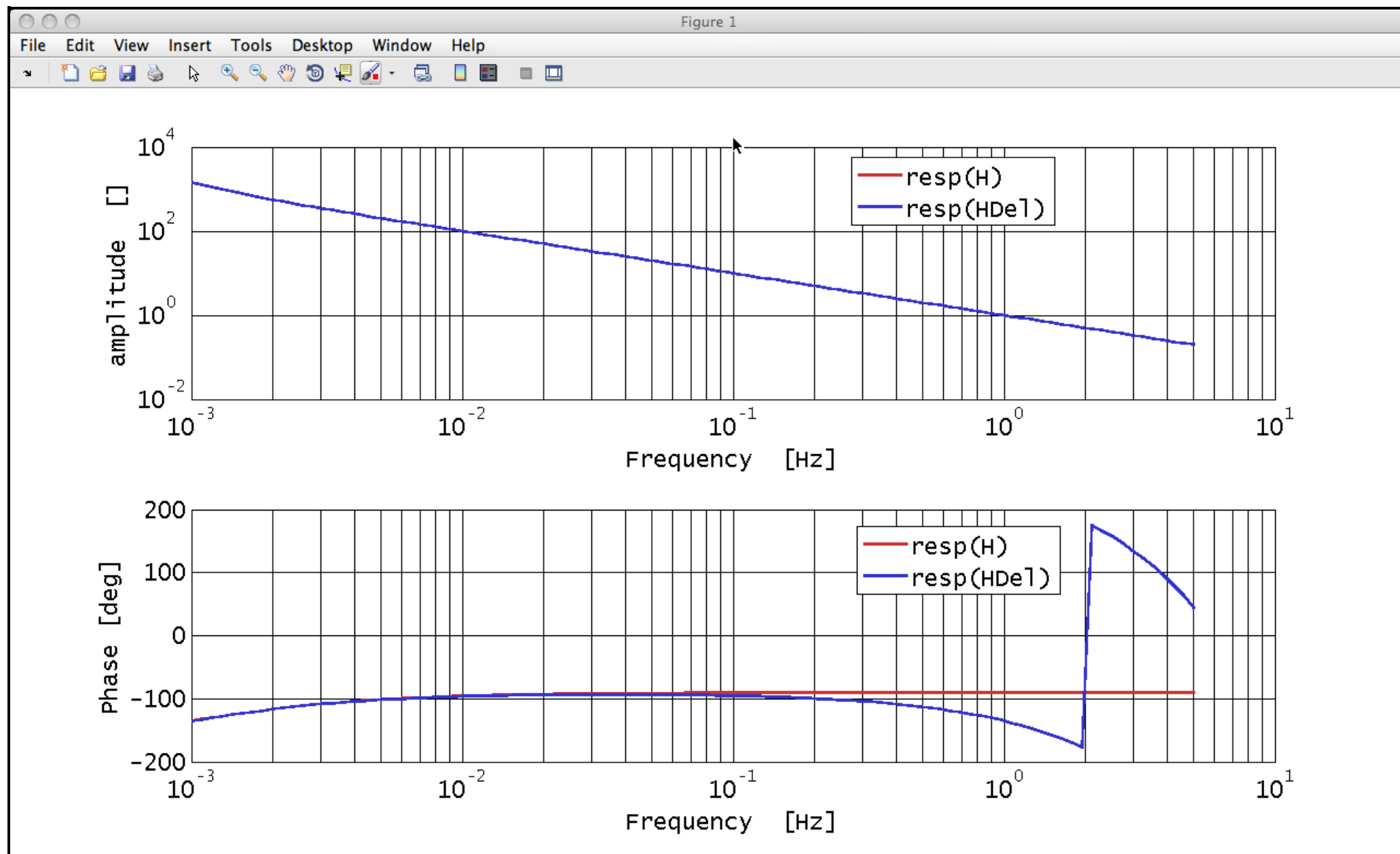


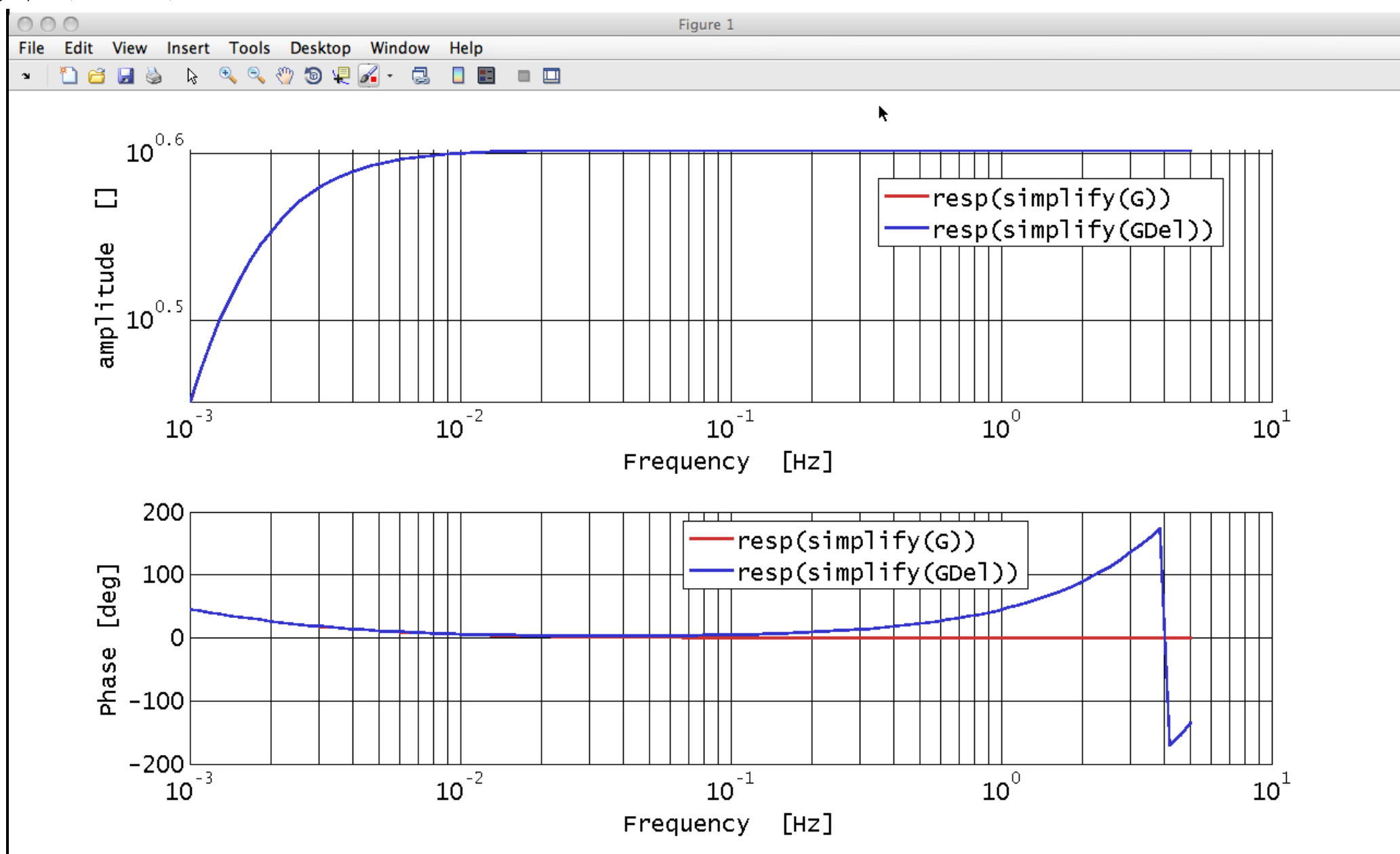
Fine, but my (real) system has a delay...

You can now repeat the same procedure but loading a model with a delay from the LISO file 'LISOFileDelay.fil'

```
>> HDel = pzmodel('topic4/LISOFileDelay.fil')
---- pzmodel 1 ----
    name: none
    gain: 10000000000
    delay: 0.125
    iunits: []
    ounits: []
description:
    UUID: 6dfbdc5-b186-4405-8f6e-02a2822a22c5
pole 001: (f=1e-06 Hz,Q=NaN)
pole 002: (f=1e-06 Hz,Q=NaN)
zero 001: (f=0.001 Hz,Q=NaN)
```

you will see how the delay is correctly handled, meaning that it is added when we multiply two models and subtracted if the models are divided.





How to filter data

How to filter data

In the previous sections we've been dealing with continuous (in s domain) systems. Now we want to apply this to data so we will need first to discretize to obtain a digital filter to then apply it to the data. Another typical application is not to derive the filter from a model but design it from the properties we want it to have (cut-off frequency, order, lowpass...). Both topics will be covered here with two examples:

- **Obtain a digital filter from a model:** we continue with the previous closed loop example to translate the obtained models into filters.
- **Define filter properties:** in this section we design a bandpass filter that will allow us to estimate the noise spectrum of interferometer data in the desired bandwidth.

 Modelling a systemBy discretizing transfer function models 

©LTP Team

By discretizing transfer function models

By discretizing transfer functions

In the following we want to show how to go from continuous (s) domain to digital (z) domain when working with transfer function models. We will keep working with the models from our previous closed loop example.

Discretizing a transfer function model

Once we have our continuous models is fairly simple to obtain their digital representation. We can insert them into the `miir` constructor specifying a sample frequency, for instance `fs = 10 Hz`, as follows

```
Gd = miir(G,plist('fs',10));
Hd = miir(H,plist('fs',10));
OLGd = miir(OLG,plist('fs',10));
```

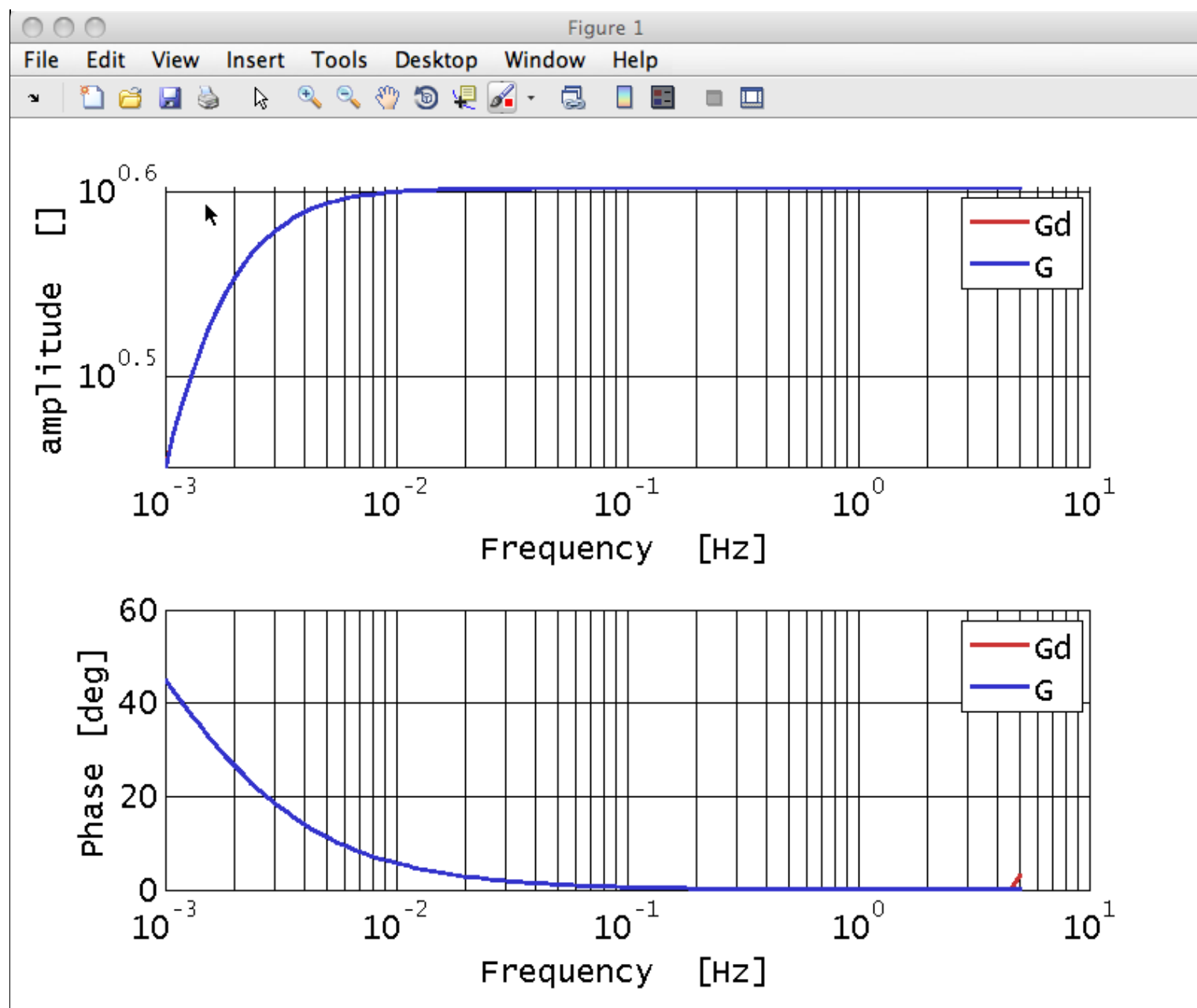
We want to check if the discretization went right, but to do that we need to compute the response for the digital and the continuous apart since the `resp` method can not process inputs from different types. We can do the following

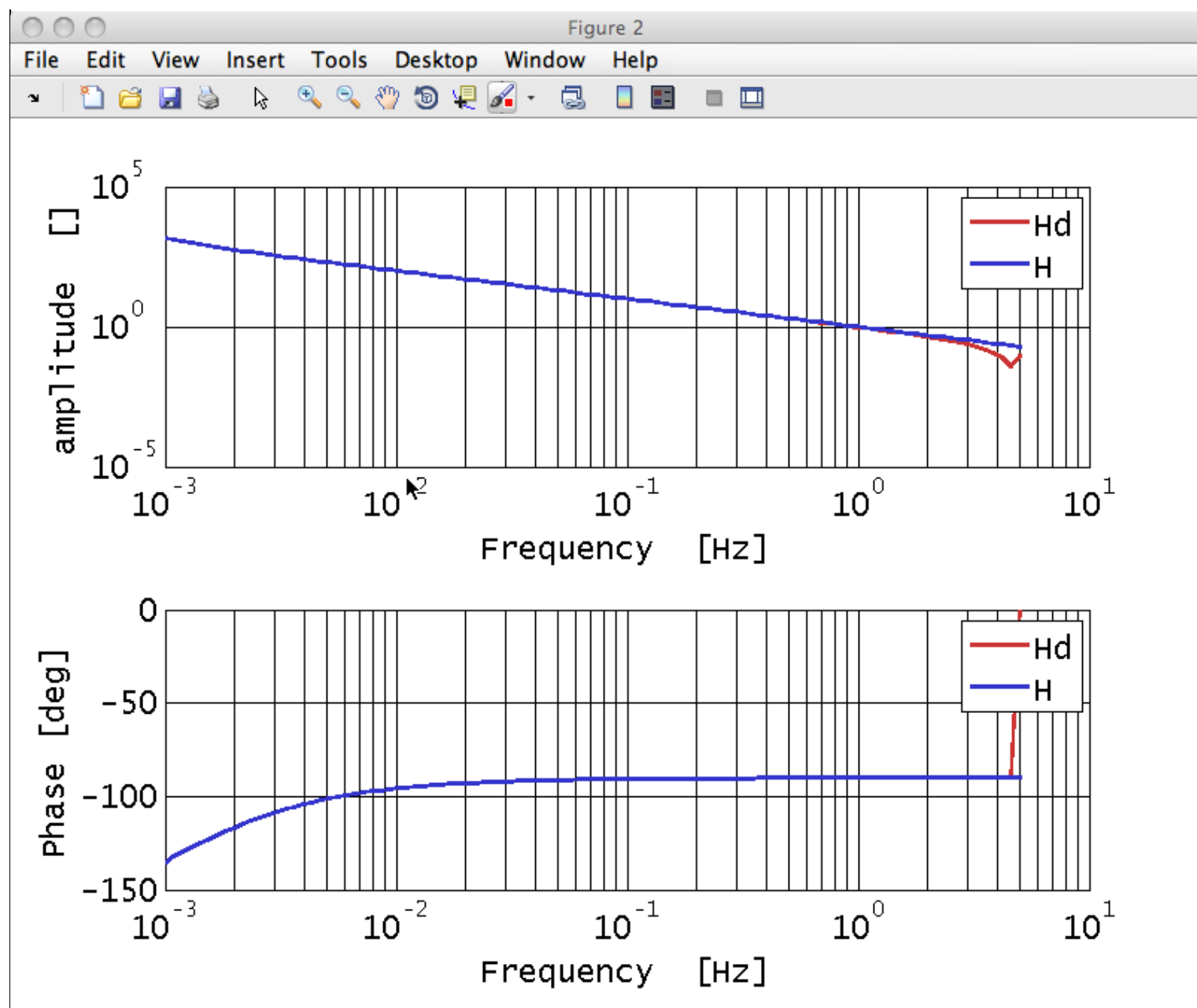
```
%% Compare response
pl = plist('f1',1e-3,'f2',5,'nf',100);

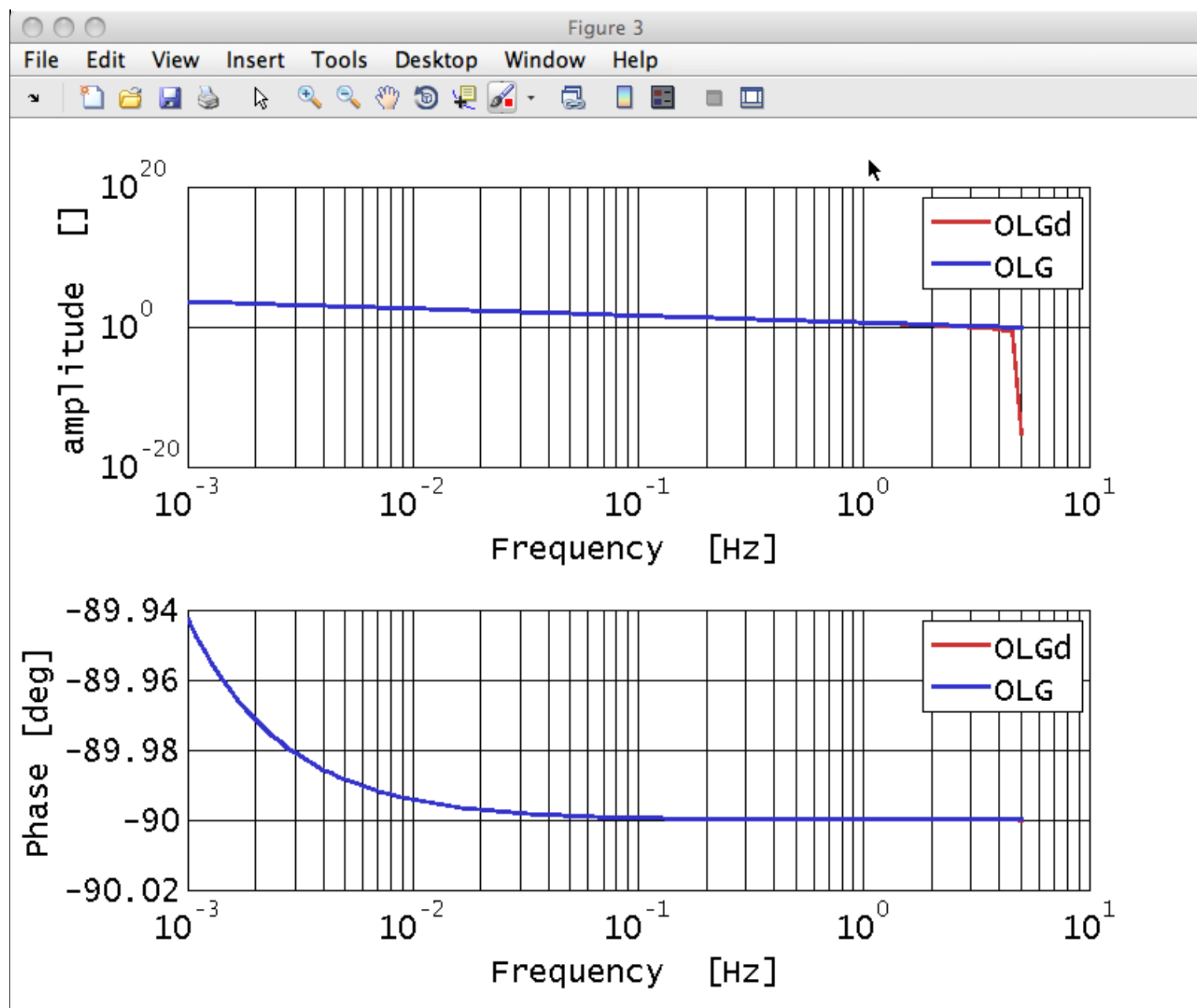
% Digital
rGd = resp(Gd,pl);
rGd.setName('Gd');
rHd = resp(Hd,pl);
rHd.setName('Hd');
rOLGd = resp(OLGd,pl);
rOLGd.setName('OLGd');

% Continuous
rG = resp(G,pl);
rG.setName('G');
rH = resp(H,pl);
rH.setName('H');
rOLG = resp(OLG,pl);
rOLG.setName('OLG');

% Plot
iplot(rGd,rG)
iplot(rHd,rH)
iplot(rOLGd,rOLG)
```







Once the comparison is done, one is usually interested in the coefficients of the digital implementation. These are directly accessible as, for example, `Gd.a` or `Gd.b`

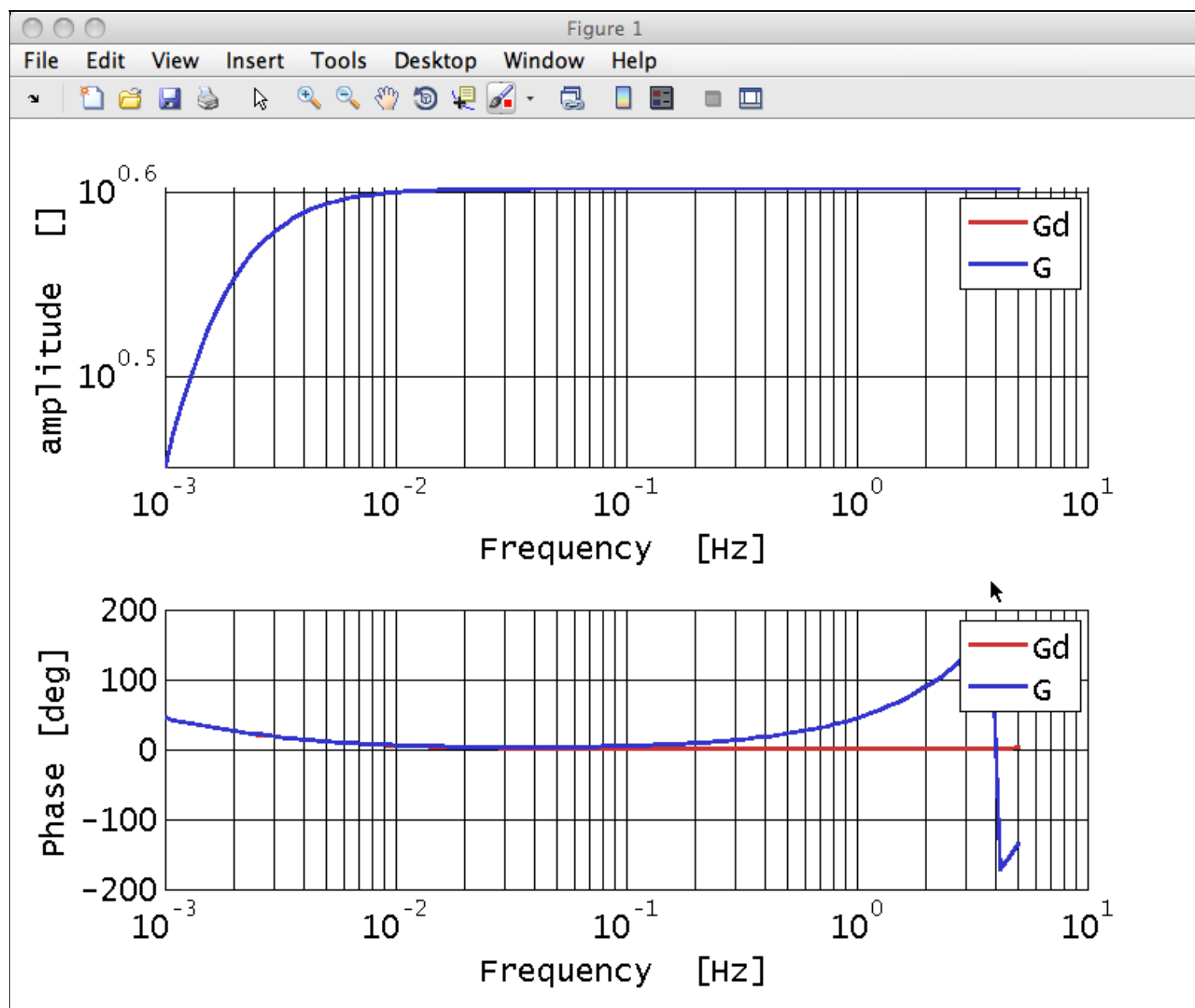
```
>> Gd.a
ans =
3.99874501384116    2.51248480253707e-06    -3.99874250135635
>> Gd.b
ans =
1    0.000628121200622944    -0.999371878799377
```

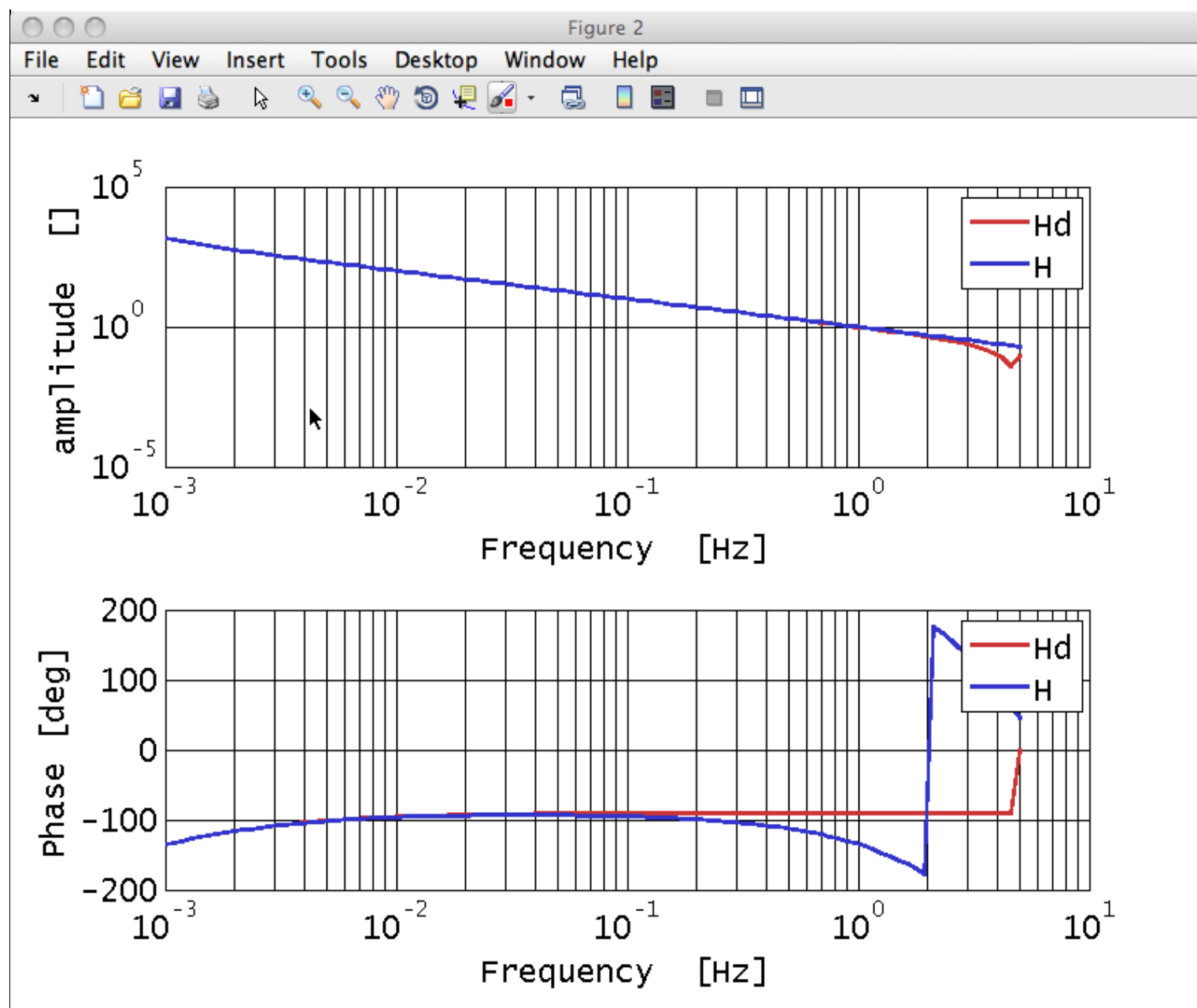
The Delay strikes back

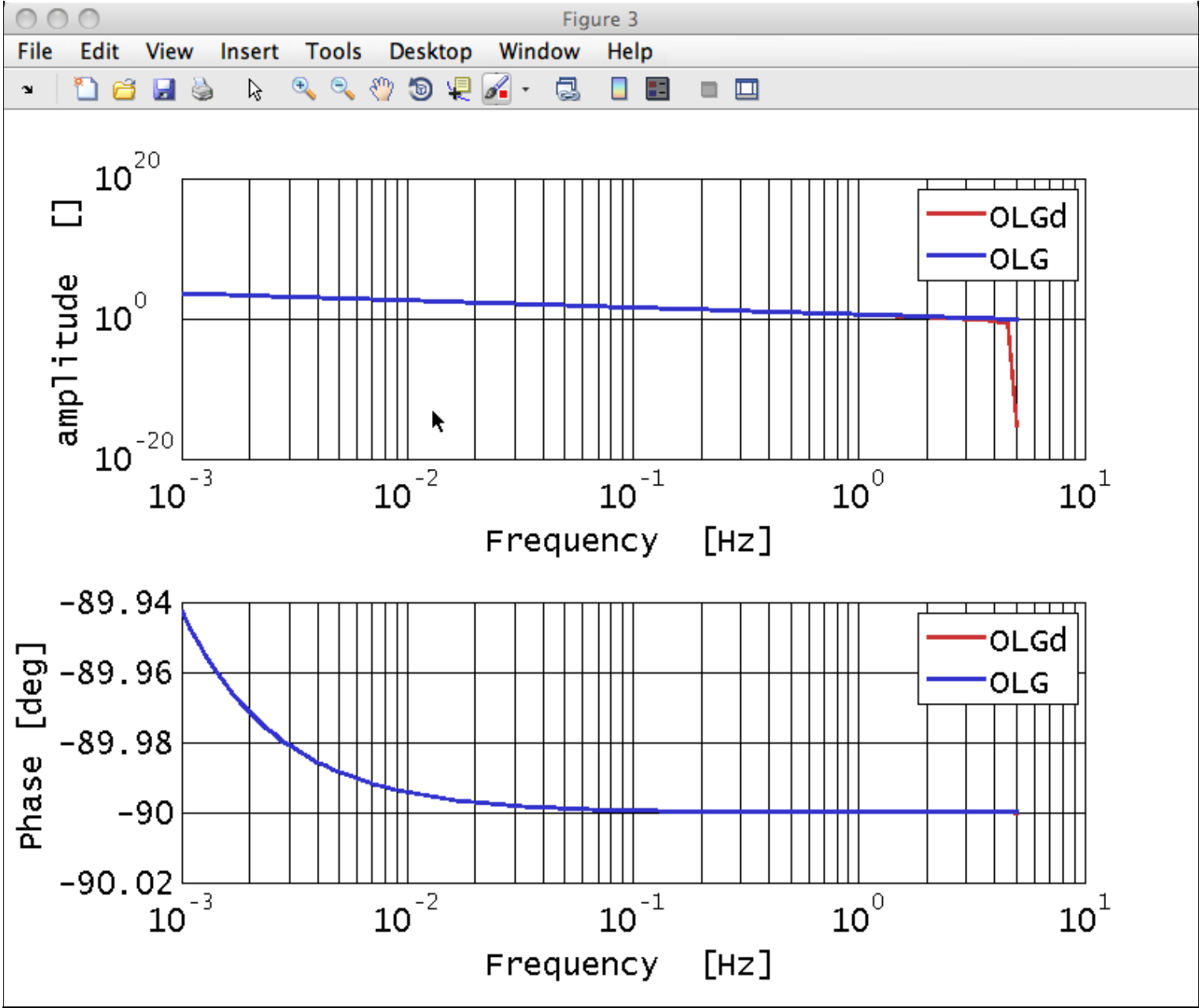
The current version of the toolbox (v2.1) is not able to translate a model with a delay into a digital filter. In such a case, the delay is ignored and a warning is thrown.

```
M: running miir/miir
!!! PZmodel delay is not used in the discretization
```

You can repeat the previous example with the model with delay ('LISOFileDelay.fil') to obtain the following







◀ How to filter data By defining filter properties ▶

By defining filter properties

By defining filter properties

In the LTPDA toolbox you have several ways to define digital filters. You can take a look at the [digital filtering](#) help pages. In this tutorial we will see how to create a bandpass filter and how to use it to filter data.

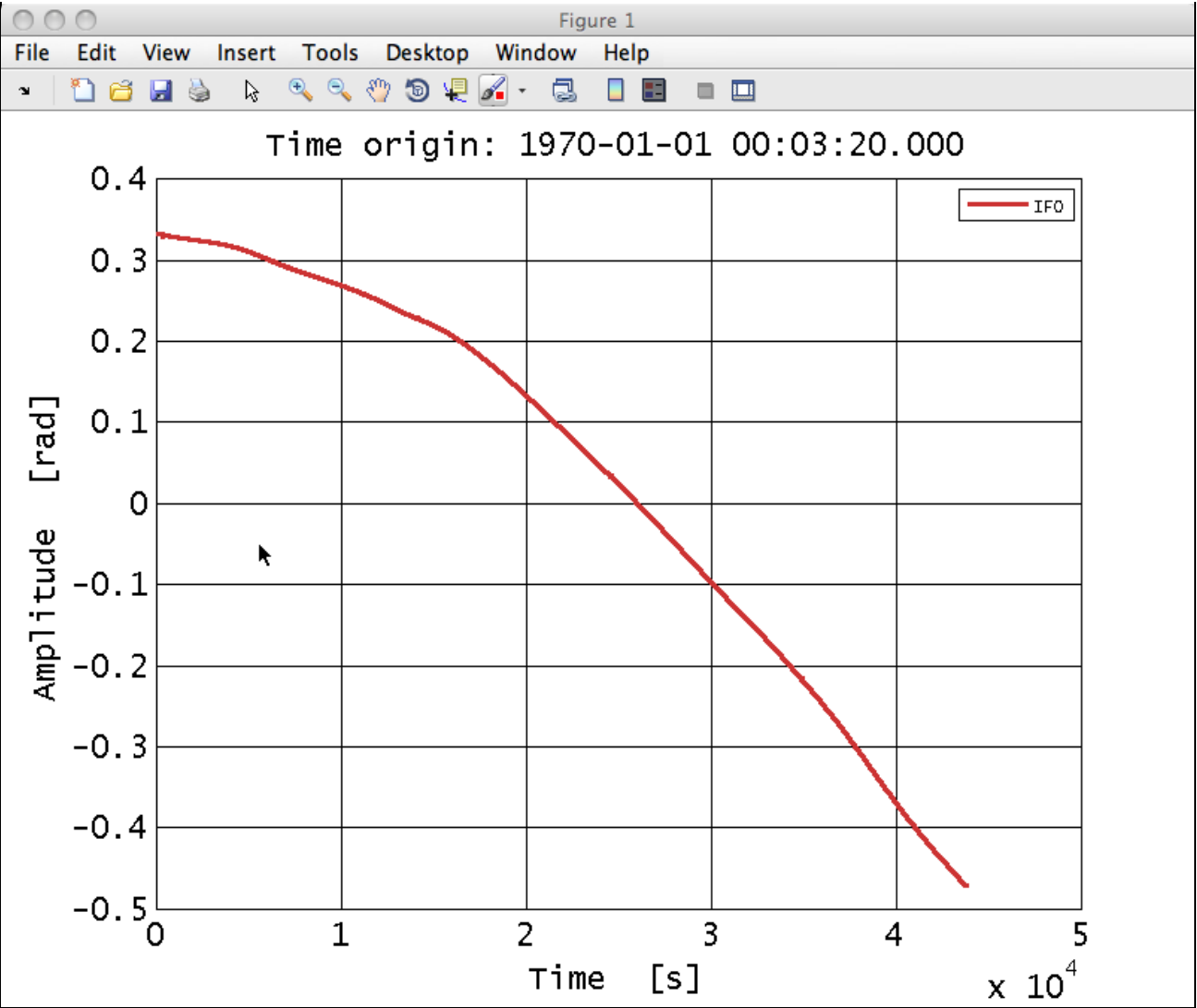
Estimating power spectrum in a limited band

In this example we want to estimate the power spectrum of an interferometer time series in the LTP bandwidth, $[1\text{e-}3, 30\text{e-}3]$ Hz. We need first to load it from the data package as follows

```
d = ao('topic4/BandpassExample.xml');
```

We can now take a look at the data

```
d.iplot
```



Before estimating the power spectrum we would like to remove all the contributions outside our band and, in particular, the trend which will add up as a low frequency feature into our spectrum. We could apply the `detrend` method but we can also design a bandpass filter to do the job. The properties of this filter would be the following

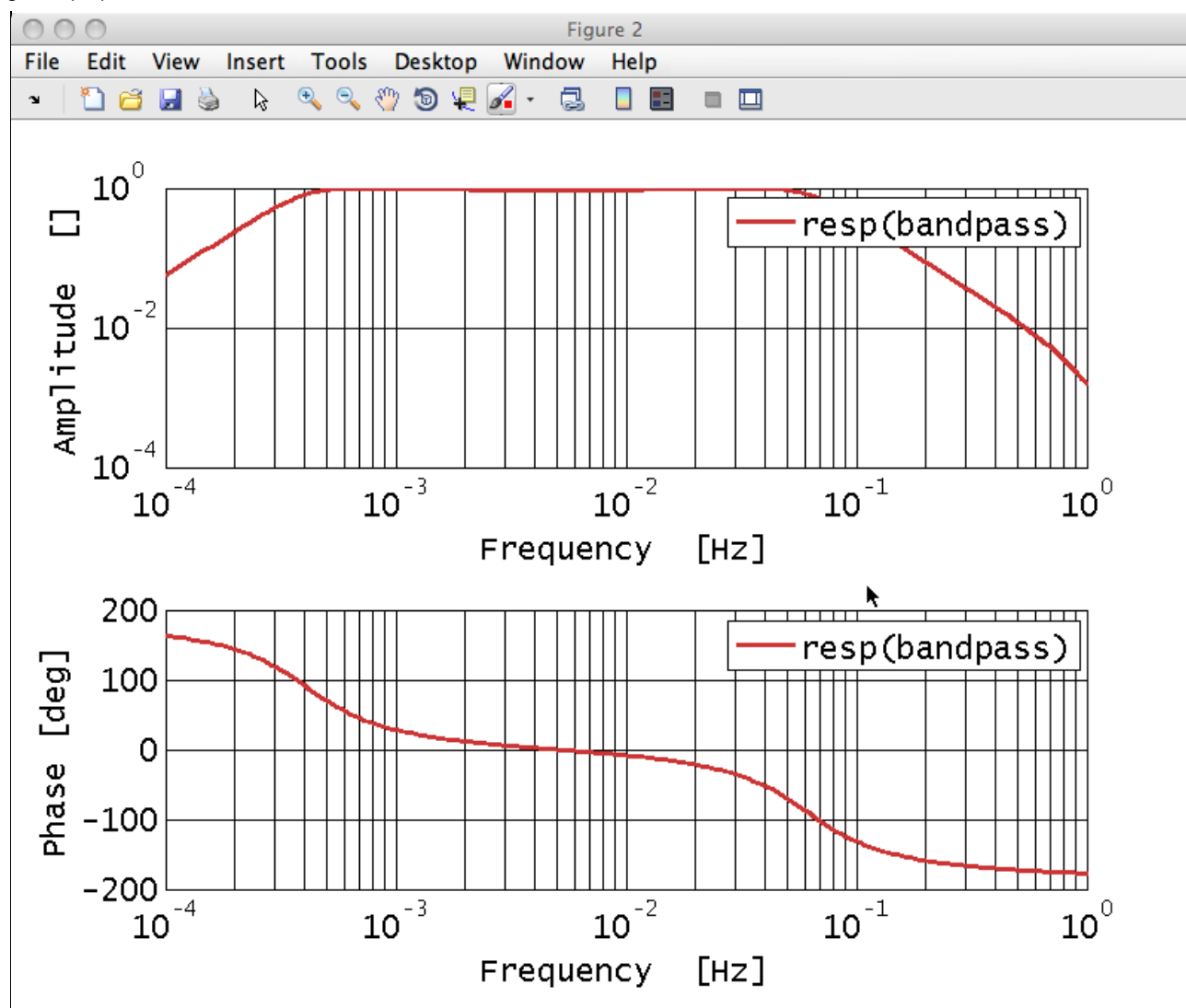
Key	Value
TYPE	'bandpass'
ORDER	2
FS	3.247
FC	[5e-4 5e-2]

We have set the cut-off frequencies slightly lower and above the interesting bandwidth trying to avoid any kind of unwanted effect in the band. All the previous properties of the filter can be set as parameters in a `plist` that we can then insert in the constructor. We will use in our example the `miir` constructor to create a IIR filter.

```
pl = plist('type', 'bandpass', 'order', 2, 'fs', 3.247, 'fc', [5e-4 5e-2]);  
bp = miir(pl);
```

`bp` is a `filter` object. Although we could set units, we don't need units here since the input and output are `rad` and therefore the filter will remain unitless. The response of the filter can be easily evaluated using the `resp` method, showing us the expected shape. We specify the frequency range and the number of points we want with a `plist`.

```
bp.resp(plist('f1', 1e-4, 'f2', 1, 'nf', 200))
```

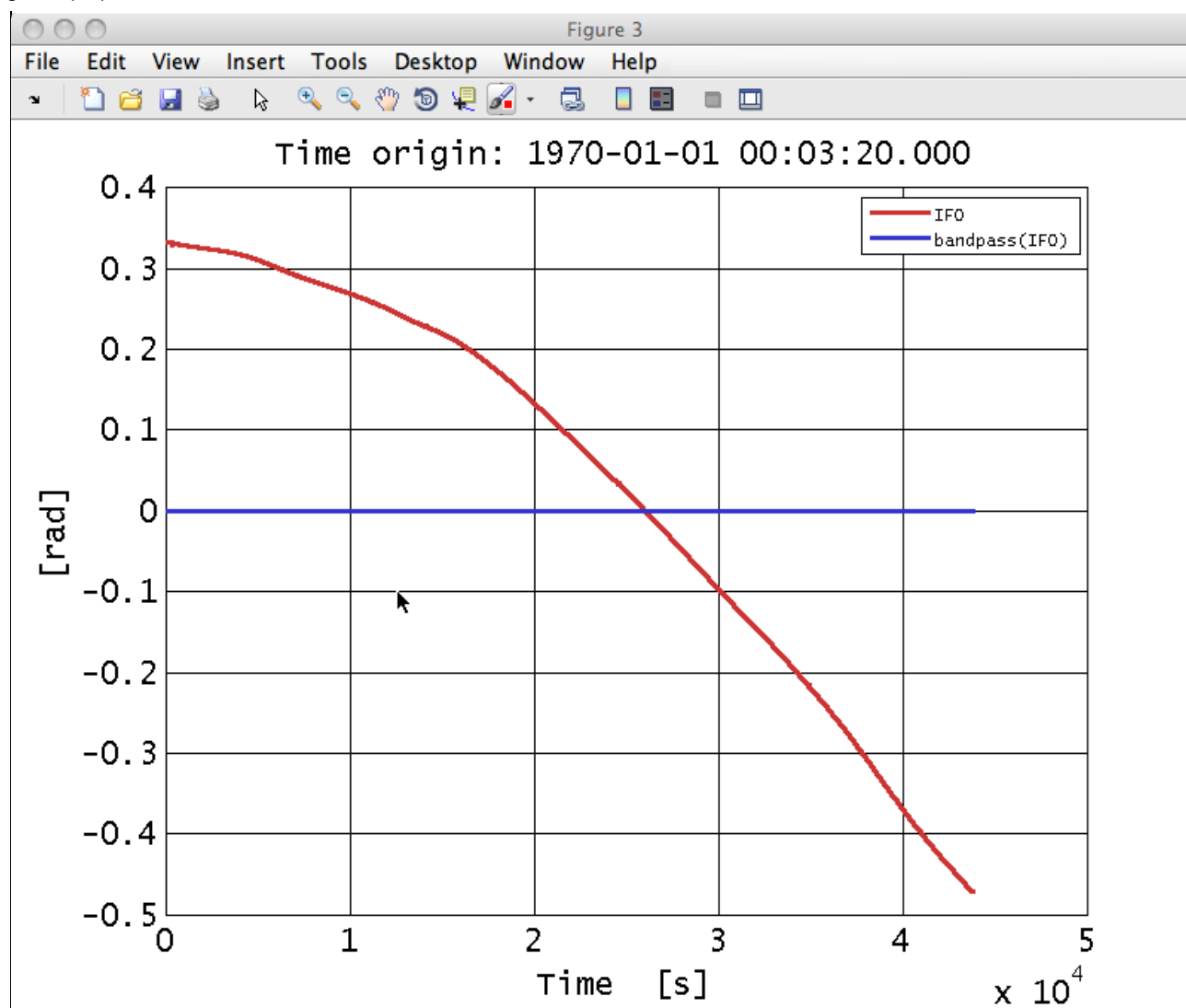


Once we have the `filter` object defined, to apply it to the data is straightforward. The standard method to do so is `filter(A0,filter_obj)`, however we will use here `filtfilt` which processes the data in both the forward and reverse direction producing a zero-phase filter but also removing transients.

```
df = filtfilt(d,bp)
```

We can see how the filter has removed the trend by plotting both time series

```
iplot(d,df)
```

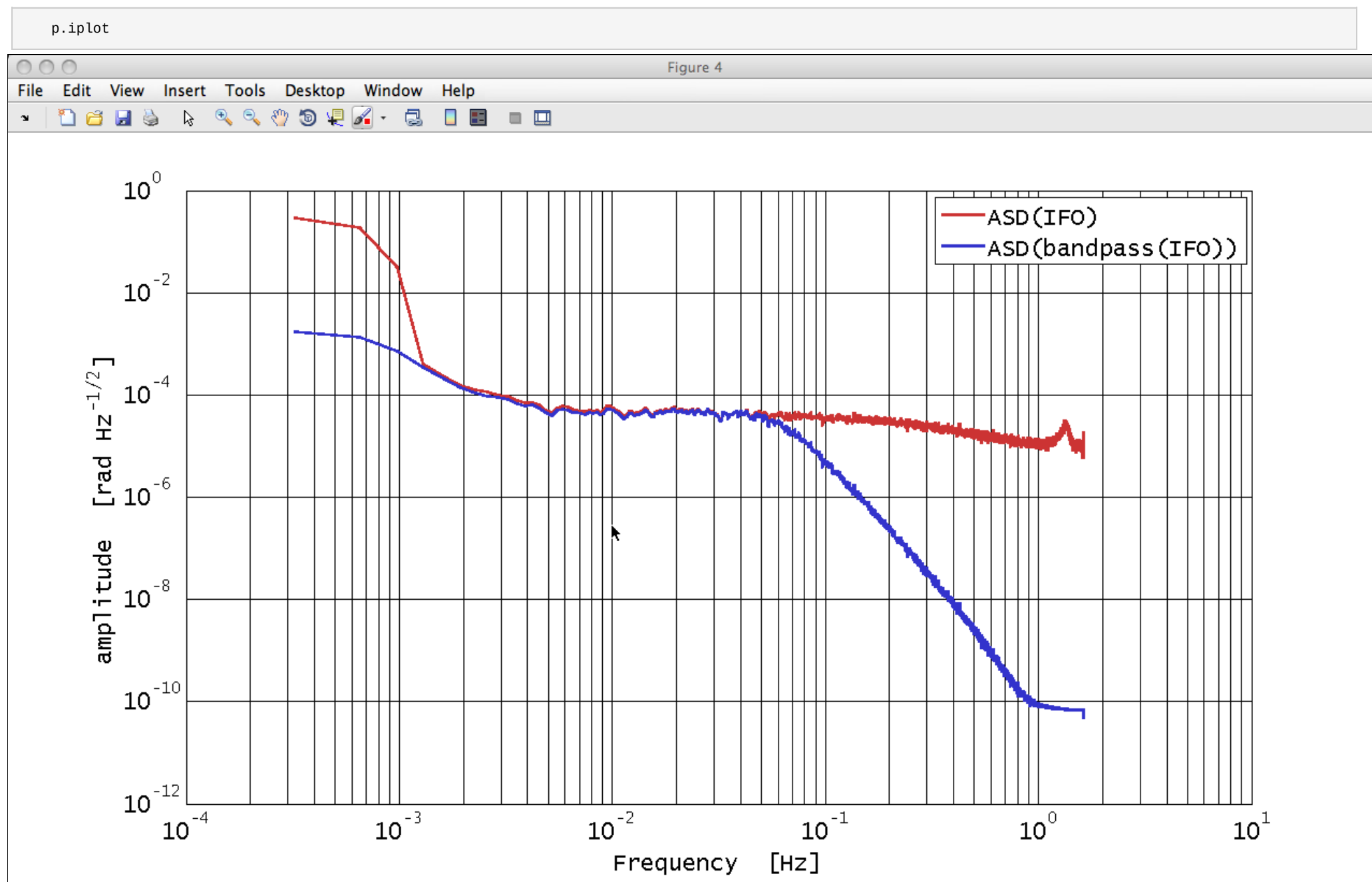
We are now ready to estimate the power spectrum using `psd` and the following parameters

Key	Value
NFFT	1e4
SCALE	'ASD'

This can be done by typing the following in the command window

```
pl = plist('Nfft',1e4, 'scale', 'ASD')
p = psd(d,df,pl)
```

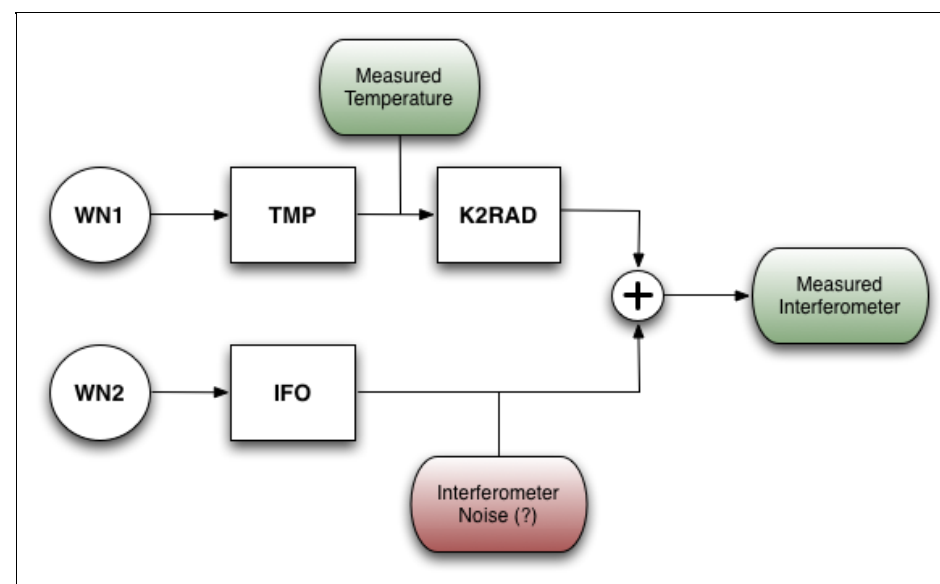
Since we entered two objects to `psd`, we get two objects at the output which can be easily plot using `ipplot`



IFO/Temperature Example – Simulation

We now come back to the IFO/Temperature working example. Our interest here is not to add more tools to the data analysis chain that you've been developing but to create a simple toy model that allows you to reproduce the steps done up to now but with synthetic data.

The problem is shown schematically in the figure below. We will generate temperature and interferometer noisy data by applying a filter (**TMP** and **IFO**) to white noise data (**WN1** and **WN2**). We will then add a temperature coupling (**K2RAD**) to the interferometer and from the measured interferometer and temperature data we will then estimate the temperature to interferometer coupling.



Since we've been through the same steps that you need to apply here in the previous section we will give here the step by step description of the task and let you play with the models.

Build the models

We need three models: one to generate temperature-like data, another modelling the interferometer and a third one acting as the K-to-rad transfer function.

STEP 1: Build a temperature noise PZMODEL with the following properties

Key	Value
'name'	'TMP'
'ounits'	'K'

GAIN	10
POLE 1	1e-5

For example, this few lines would do the job

```
TMP = pzmodel(10,1e-5, []);
TMP.set0units('K');
TMP.setName('TMP');
```

STEP 2: Build a interferometer noise PZMODEL with the following properties

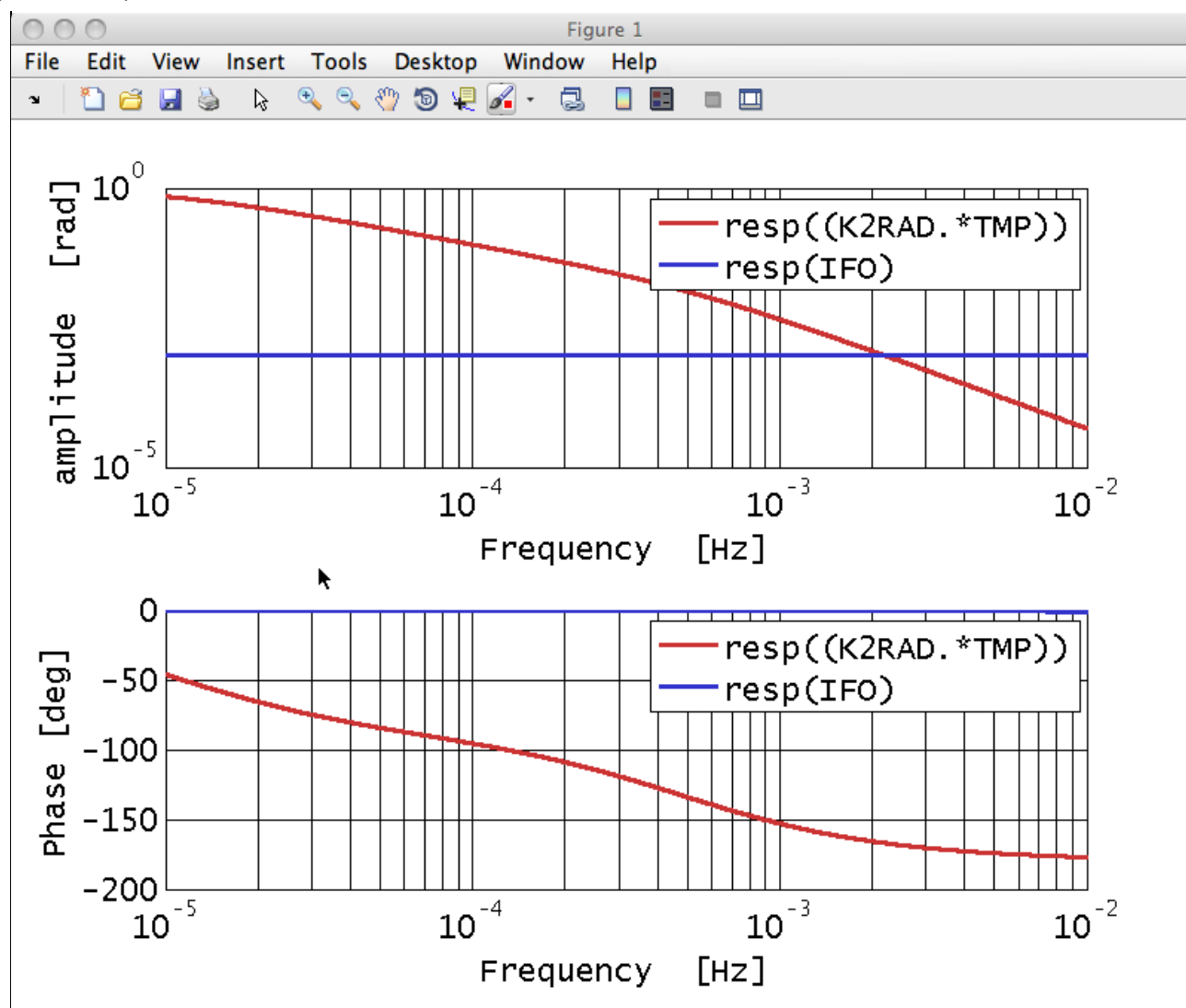
Key	Value
'name'	'IFO'
'ounits'	'rad'
GAIN	1e-3
POLE 1	0.4

STEP 3: Build temperature to interferometer coupling PZMODEL with the following properties

Key	Value
'name'	'K2RAD'
'iunits'	'K'
'ounits'	'rad'
GAIN	1e-1
POLE 1	5e-4

You can take a look at your models. Since we are interested in the projection of temperature into interferometric data, we can plot the response of TMP*K2RAD against the IFO

```
pl = plist('f1',1e-5,'f2',0.01)
resp(K2RAD*TMP, IFO, pl)
```



Discretize the models

Now discretize the models at $f_s = 1\text{Hz}$ using the `miir` constructor. After that you will obtain three digital filters

STEP 4: Discretize the three transfer (TMP,IFO,K2RAD) with the MIIR constructor

For example, the model related to temperature noise would be discretized like this:

```
TMPd = miir(TMP,plist('fs',1));
```

Generate white noise data

We will need two initial white noise time series, WN1 and WN2, that we will use as a seed to apply our filters and get noise shaped time series.

STEP 5: Generate white noise with the AO constructor

You will need the ao constructor for that. You could use the following settings

Key	Value
'name'	'WN1'
'tsfcn'	'randn(size(t))'
'fs'	1
'nsecs'	250000

Key	Value
'name'	'WN2'
'tsfcn'	'randn(size(t))'
'fs'	1
'nsecs'	250000

Generate the noise data streams

For each noise source you will need to apply the filter that you have designed to the white noise data:

STEP 6: Filter white noise WN1 with the TMPd filter

For example, following our notation:

```
T = filter(WN1,TMPd);
```

STEP 7: Filter white noise WN2 with the IFOd filter

Temperature and interferometric noise are uncorrelated, so we need to use here the second noise time series **WN2**

For example, following our notation:

```
IFO = filter(WN2,IFOd);
```

STEP 8: Filter white noise WN1 with the TMPd and the K2RADd filter

In this case you need to apply both filters in serial, you can do this in one command by using the 'bank' property of the `filter` method.

Hint: you can input a vector of filters into the `filter` method and ask it to filter the data in 'parallel' or in 'serial' (the one we are interested here) by doing the

following

```
b = filter(WN1,[TMPd K2RADd],plist('bank','serial'));
```

STEP 9: Add the IFO noise to the K2RAD noise

At this point the IFO represents the purely interferometric noise and the K2RAD the contribution to interferometric noise coming from temperature. You need to add both to get the final interferometric data. This only requires to add both AOs.

Perform the noise projection

Here we will reproduce the main steps performed in topic 3 analysis: power spectral and transfer function estimation.

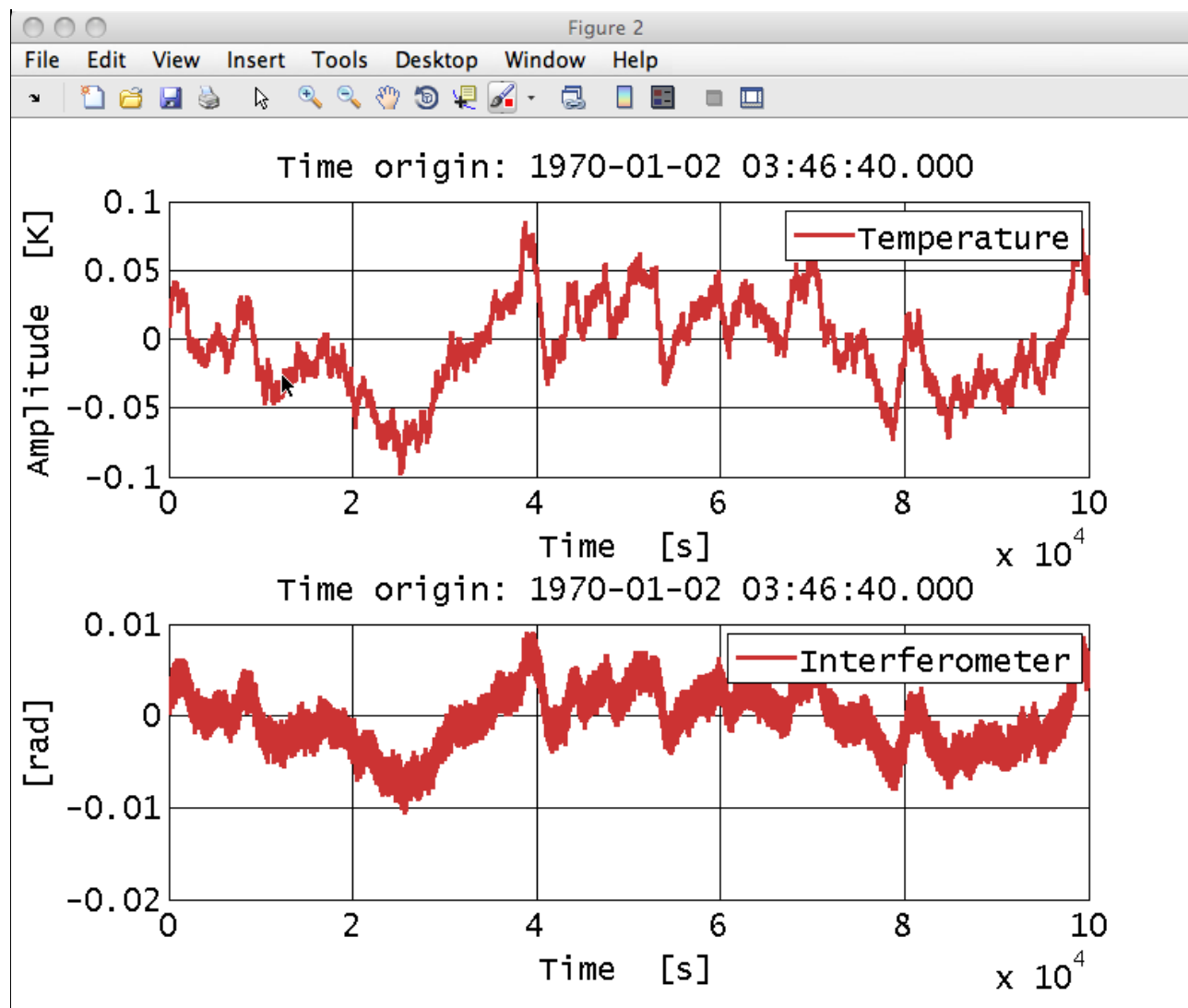
STEP 10: Split the data streams

We will do the analysis with data in the region going from 1e5 to 2e5 seconds to avoid initial transients. You must then split your two data streams introducing the following parameters in the `split` method.

Key	Value
'times'	[1e5 2e5]

After the splitting you must have two data streams that plot together should look like the ones below. The code should be similar to one in the following lines

```
Ts = split(T,plist('times',[1e5 2e5]))
Ts.setName('Temperature')
IFOs = split(IFO_all,plist('times',[1e5 2e5]))
IFOs.setName('Interferometer')
% Plot in different panels with 'subplots' options
iplot(Ts,IFOs,plist('arrangement','subplots'))
```

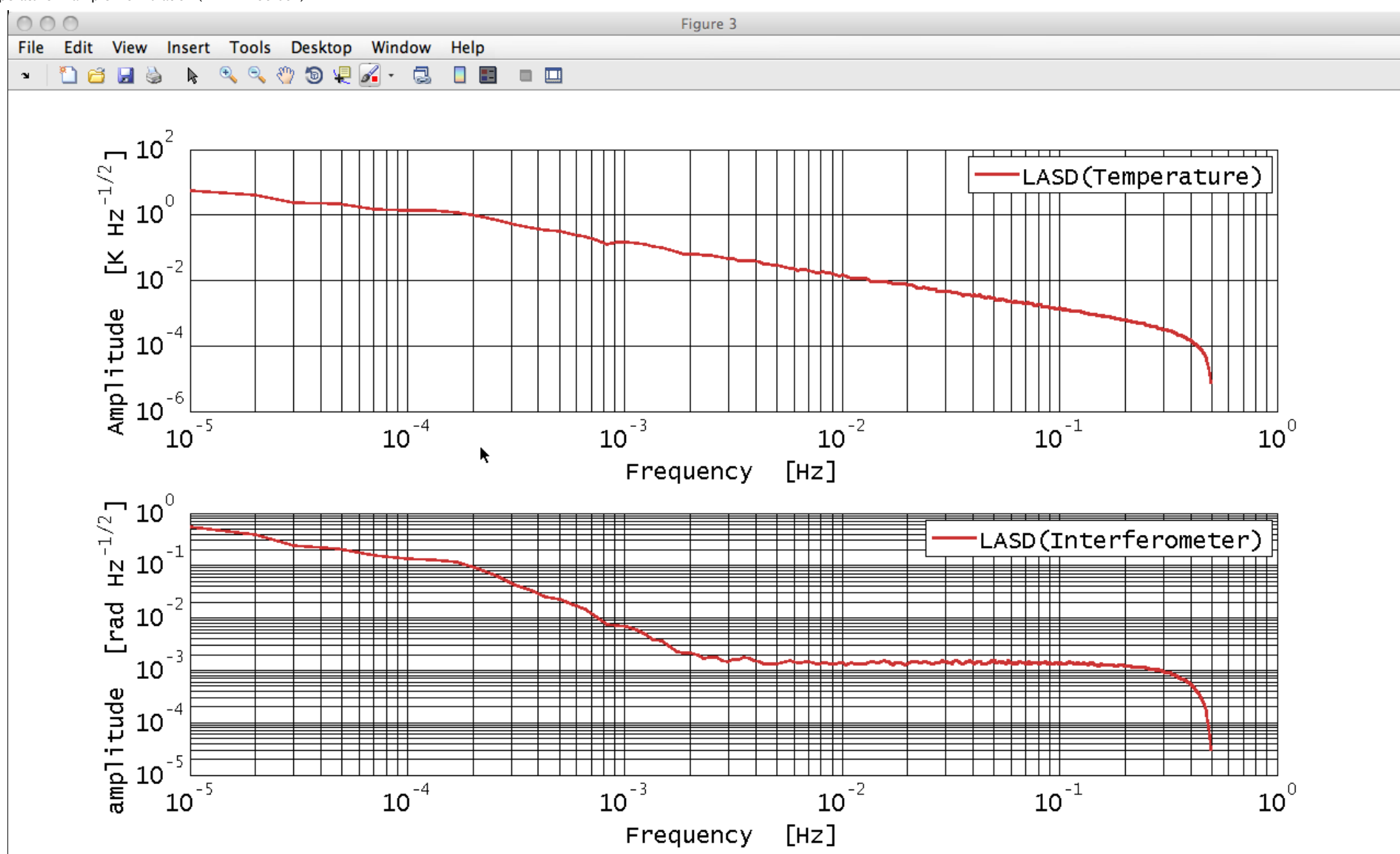



STEP 11: Compute power spectral estimates for the temperature and interferometric data

Here you need to apply `lpsd` or `psd` methods. For example:

```
pl = plist('order',1,'scale','ASD')
psd_T = lpsd(Ts,pl)
psd_IFO = lpsd(IFOs,pl)
```

The resulting spectrum should look like this



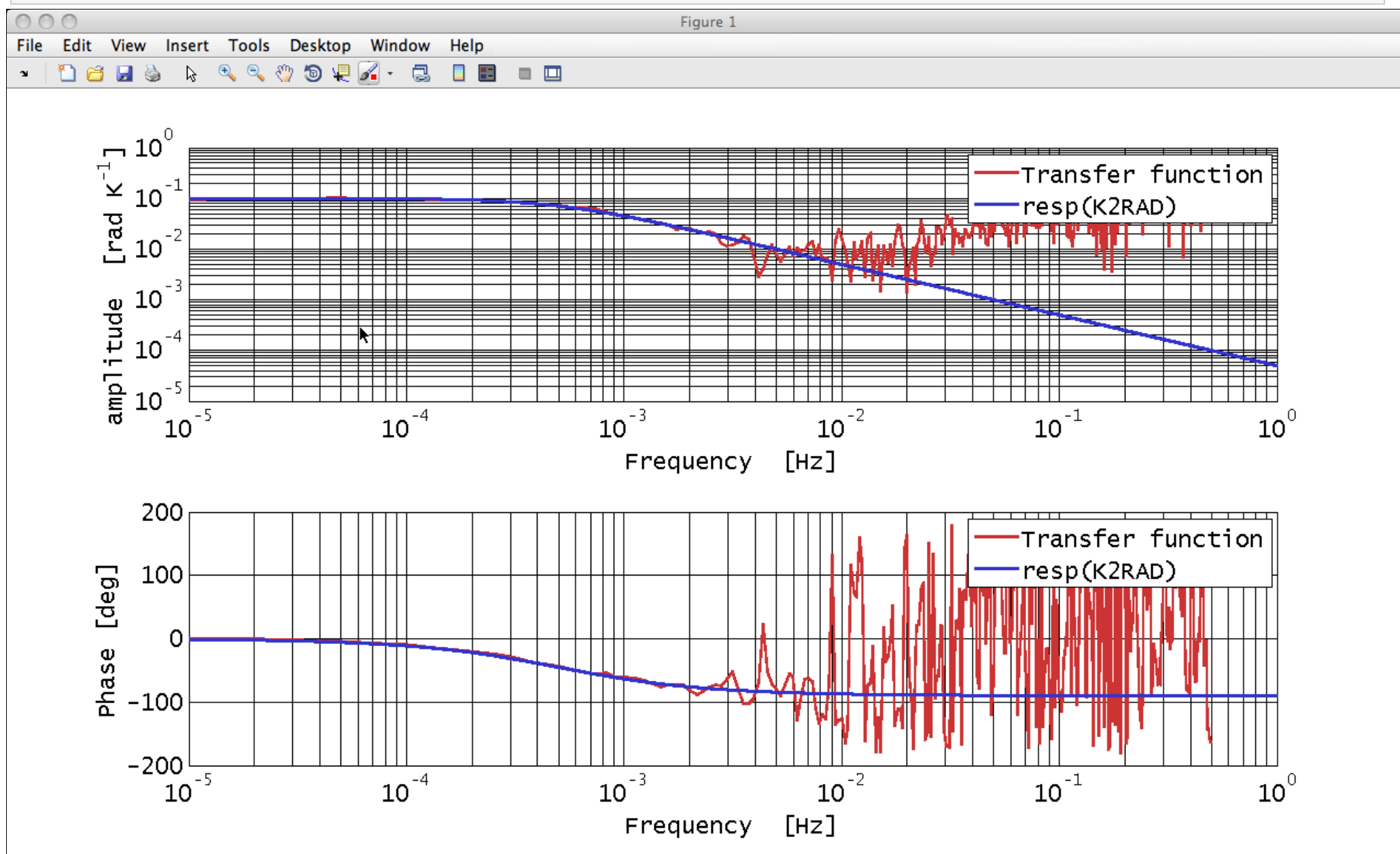
STEP 12: Compute transfer function estimate for the temperature and interferometric data

Here you need to apply `ltfe` or `tfe` methods. For example:

```
T2IFO = ltfe(Ts, IFOs)
T2IFO.setName('transfer function')
```

You can now compare the transfer function model with the estimation obtained from the data:

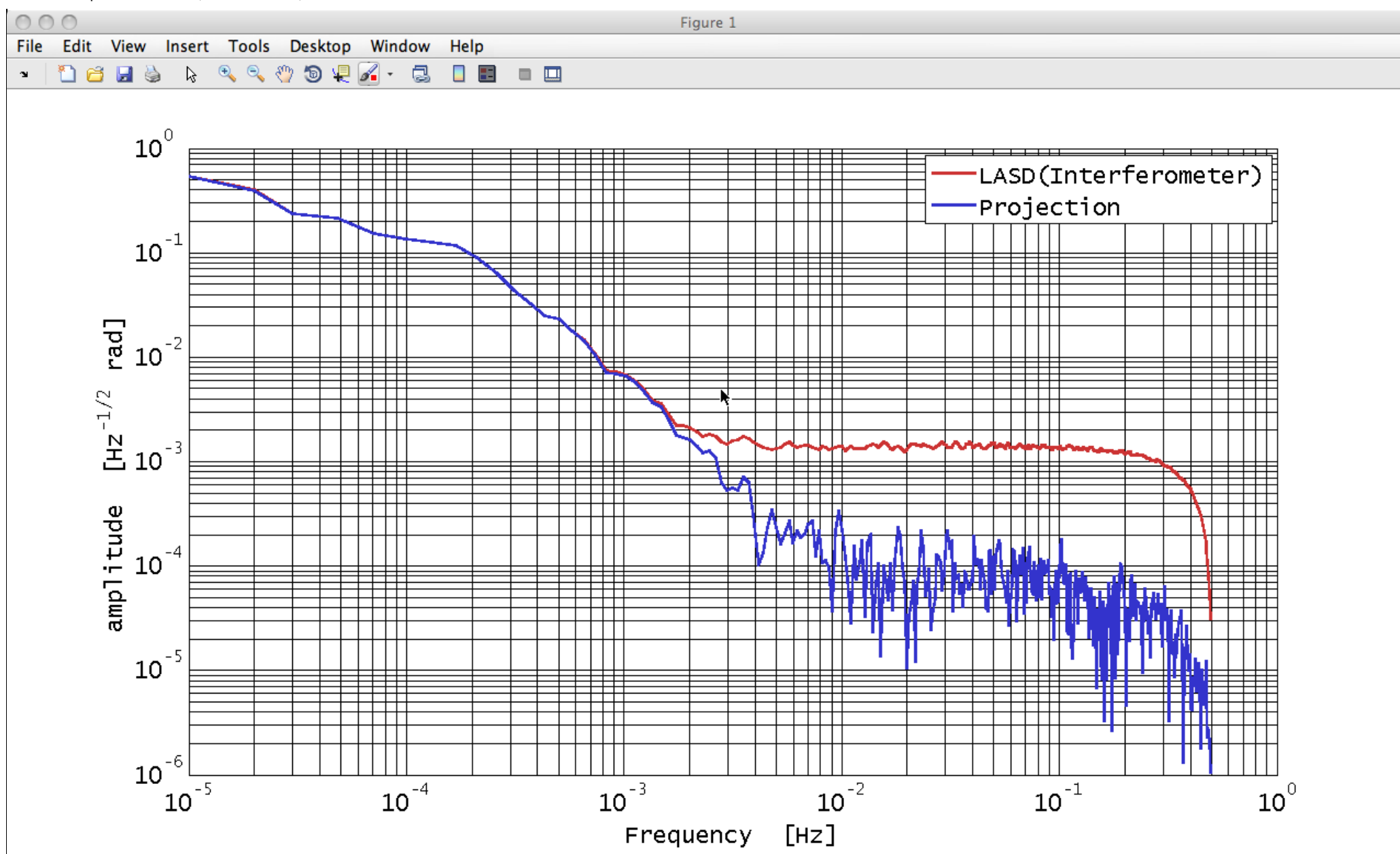
```
pl = plist('f1',1e-5,'f2',1)
ipplot(T2IFO,resp(K2RAD,pl))
```



STEP 13: Project the temperature noise

Reproducing the analysis performed in topic 3 you will be able to project the temperature noise contribution into interferometric noise. The result obtained should be the one in the figure below and the code you will need for that should be similar to this:

```
% Compute projection
Projection = abs(T2IF0).*psd_T;
Projection.simplifyYunits
Projection.setName;
% Plot against interferometer noise
iplot(psd_IF0,Projection)
```



◀ By defining filter properties

Topic 5 – Model fitting ▶

Topic 5 – Model fitting

Topic 5 of the training session aims to briefly introduce the advanced fitting capabilities offered by LTPDA. After working through the examples you should be familiar with:

- [System identification in z-domain](#)
- [Generation of noise with given psd](#)
- [Fitting time series with polynomials](#)
- [Non-linear least square fitting of time series](#)
- [Time-domain subtraction of temperature contribution to interferometer signal](#)

◀ IFO/Temperature Example – Simulation

System identification in z-domain ▶

©LTP Team

System identification in z-domain

System identification in Z-domain is performed with the function `ao/zDomainFit`. It is based on a modified version of the vector fitting algorithm that was adapted to fit in the Z-domain. Details of the algorithm can be found in the [Z-domain fit documentation page](#).

System identification in Z-domain

During this exercise we will:

1. Generate white noise
2. Filter white noise with a `miir` filter generated by a `pzmodel`
3. Extract the transfer function from data
4. Fit the transfer function with `ao/zDomainFit`
5. Check results

Let's start by generating some white noise.

```
a = ao(plist('tsfcn', 'randn(size(t))', 'fs', 1, 'nsecs', 10000, 'yunits', 'm'));
```

This command generates a time series of gaussian distributed random noise with a sampling frequency ('fs') of 1 Hz, 10000 seconds long ('nsecs') and with 'yunits' set to meters ('m').

Now we are ready to move on the second step where we will:

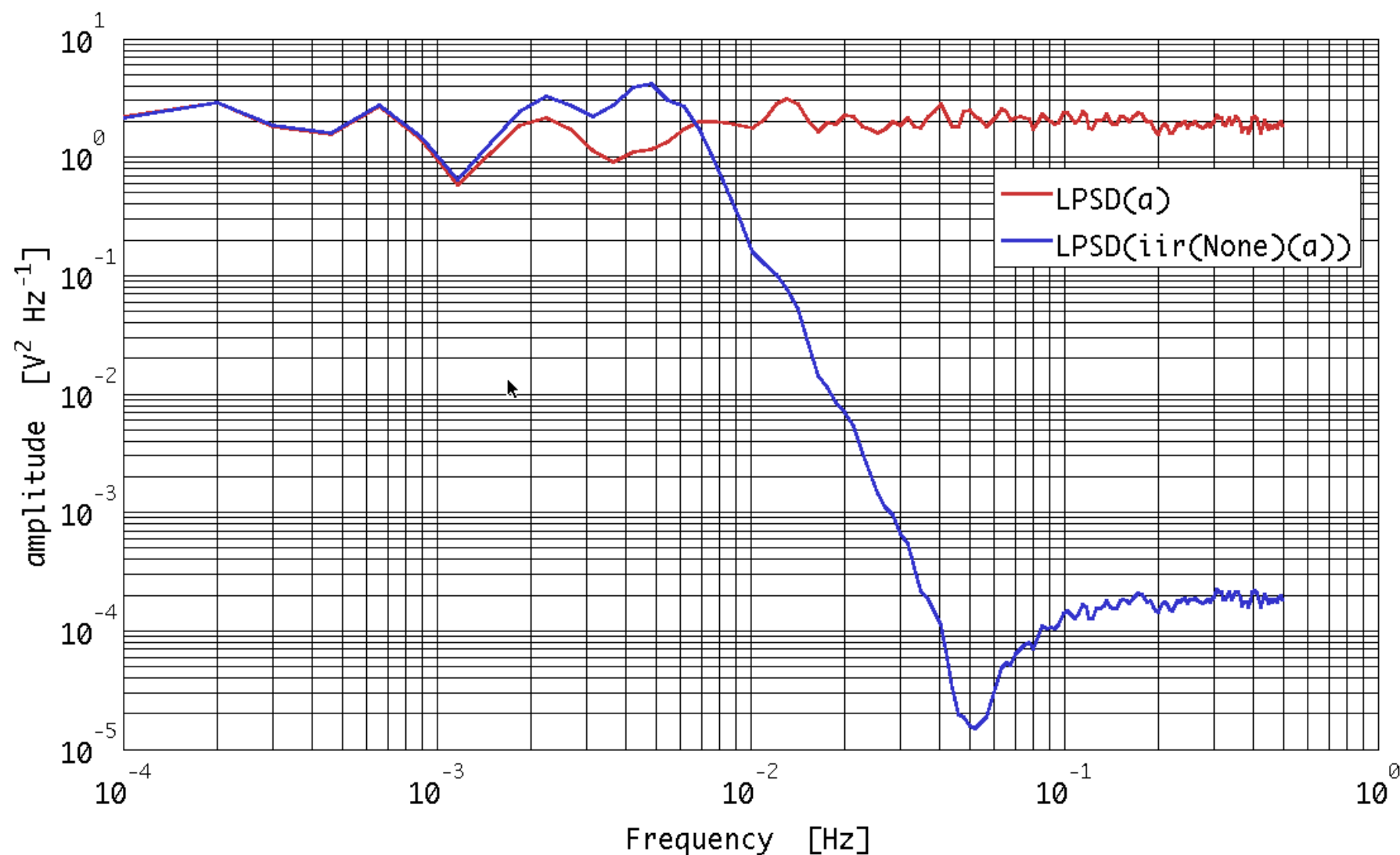
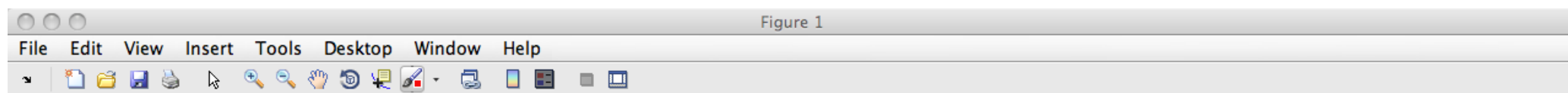
- Build a pole-zero model (`pzmodel`)
- Construct a `miir` filter from the `pzmodel`
- filter white noise data with the `miir` filter in order to obtain a colored noise time series.

```
pzm = pzmodel(1, [0.005 2], [0.05 4]);  
filt = miir(pzm, plist('fs', 1));  
filt.setIunits('m');  
filt.setOunits('V');  
  
% Filter the data  
ac = filter(a, filt);  
ac.simplifyYunits;
```

We can calculate the PSD of the data streams in order to check the difference between the coloured noise and the white noise. Let's choose the log-scale estimation method.

```
axx = lpsd(a);  
acxx = lpsd(ac);  
iplot(axx, acxx)
```

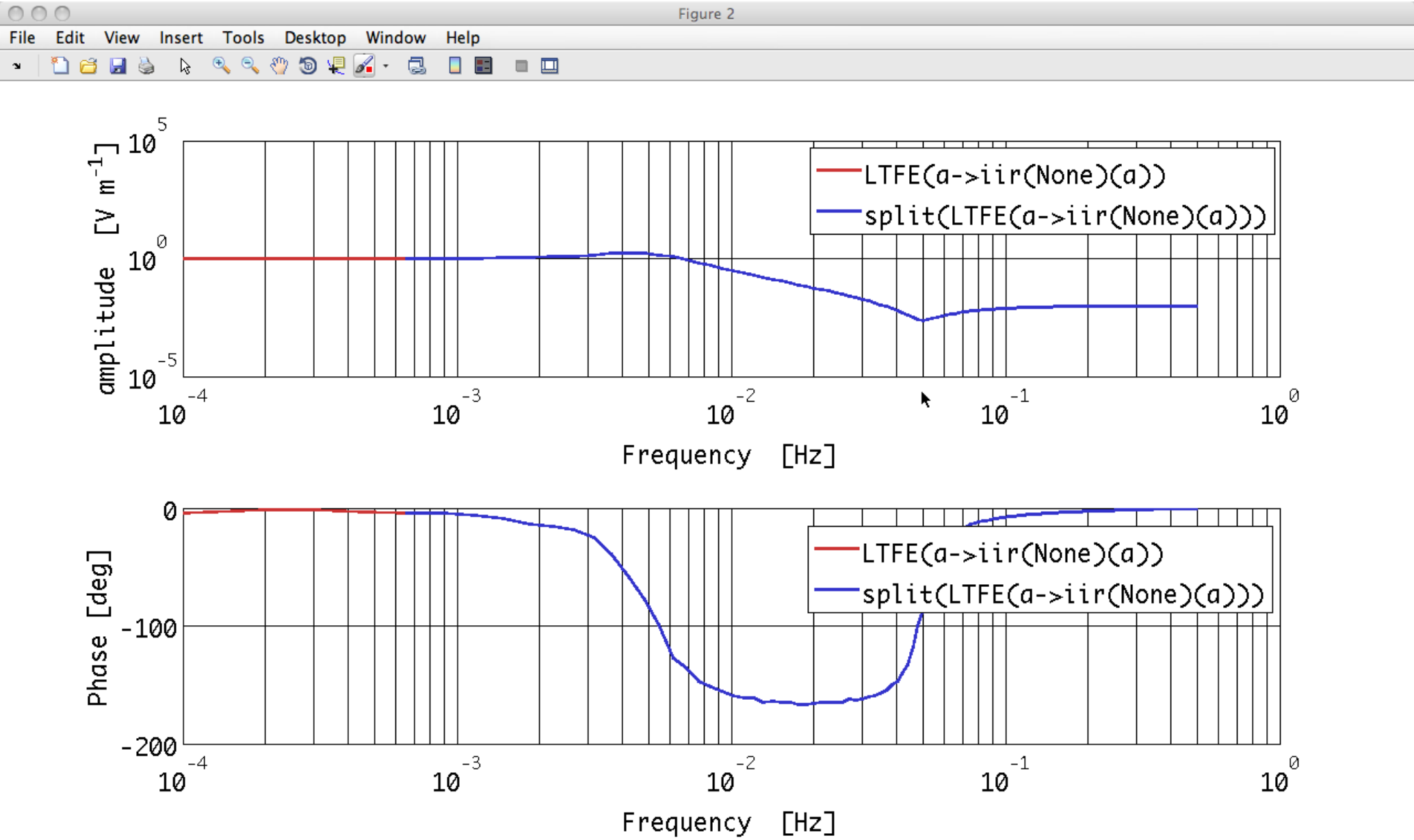
You should obtain a plot similar to this:



Let us move to the third step. We will generate the transfer function from the data and split it in order to remove the first 3 bins. The last operation is useful for the fitting process.

```
tf = ltfe(a,ac);
tfsp = split(tf,plist('frequencies', [5e-4 5e-1]));
ipplot(tf,tfsp)
```

The plot should look like the following:



It is now the moment to start fitting with `zDomainFit`. As reported in the function help page we can run an automatic search loop to identify proper model order. In such a case we have to define a set of conditions to check fit accuracy and to exit the fitting loop. We can start checking "Residuals Logarithm Difference" and "Root Mean Squared Error". It is worth remembering that:

Key	Description

ResLogDiff

This makes `zDomainFit` check that the normalized difference between the data and the fit residuals (in log scale) is larger than the value specified. If `d` is the input value for the parameter, then the algorithm checks that $\log_{10}(\text{data}) - \log_{10}(\text{res}) > d$.

RMSE

If `r` is the parameter value, then the algorithm checks that the step-by-step Root Mean Squared Error variation is less than 10^{-r} .

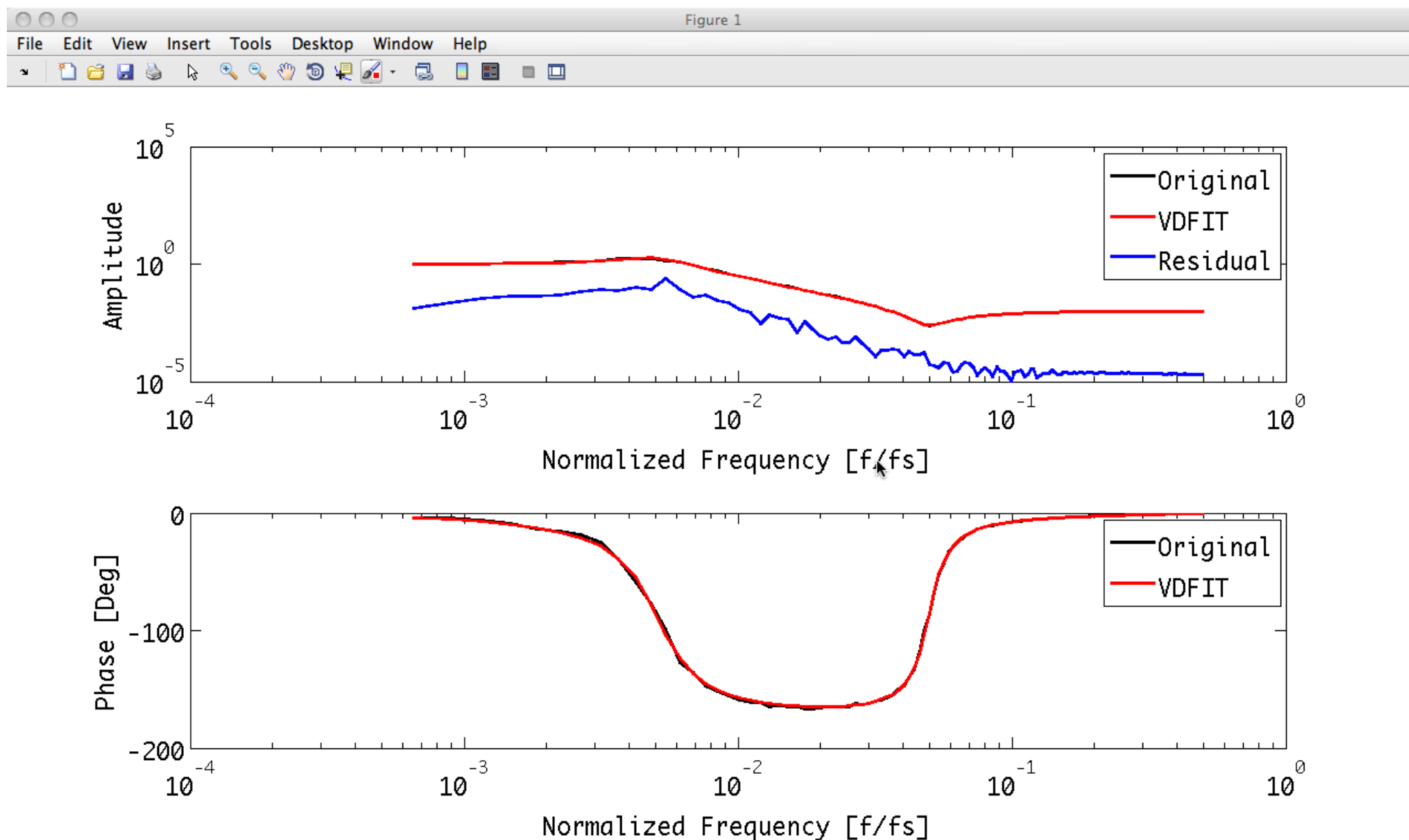
Now let's run the fit: we set '**ResLogDiff**' to a small number (0.5) because with noisy data we do not want/expect a perfect match.

```
% Set up the parameters
plfit = plist('fs',1,... % Sampling frequency for the model filters
'AutoSearch','on',... % Automatically search for a good model
'StartPolesOpt','clog',... % Define the properties of the starting poles - complex distributed in the unitary circle
'maxiter',50,... % maximum number of iteration per model order
'minorder',2,... % minimum model order
'maxorder',9,... % maximum model order
'weightparam','abs',... % assign weights as 1./abs(data)
'ResLogDiff',0.5,... % Residuals log difference
'ResFlat',[],... % Residuals spectral flatness
'RMSE',5,... % Root Mean Squared Error Variation
'Plot','on',... % set the plot on or off
'ForceStability','on',... % force to output a stable poles model
'CheckProgress','off'); % display fitting progress on the command window

% Do the fit
fobj = zDomainFit(tfsp,plfit);

% Set the input and output units for fitted model
fobj.setIunits('m');
fobj.setOunits('V');
```

When '**Plot**' parameter is set to '**on**' the function plots the fit progress.



We can now check the result of our fitting procedures. We calculate the frequency response of the fitted models (filters) and compare them with the starting IIR filter response, then we will plot the percentage error on the filters magnitudes.

Note that the result of the fitting procedure is a `matrix` object containing a `filterbank` object, which itself contains a parallel bank of 3 IIR filters.

Note that at the moment we need to access the individual filter objects inside the `matrix` object that was the result of the fitting procedure.

```
% set plist for filter response
plrsp = plist('bank','parallel','f1',1e-5,'f2',0.5,'nf',100,'scale','log');

% compute the response of the original noise-shape filter
rfilt = resp(filt,plrsp);
rfilt.setName;

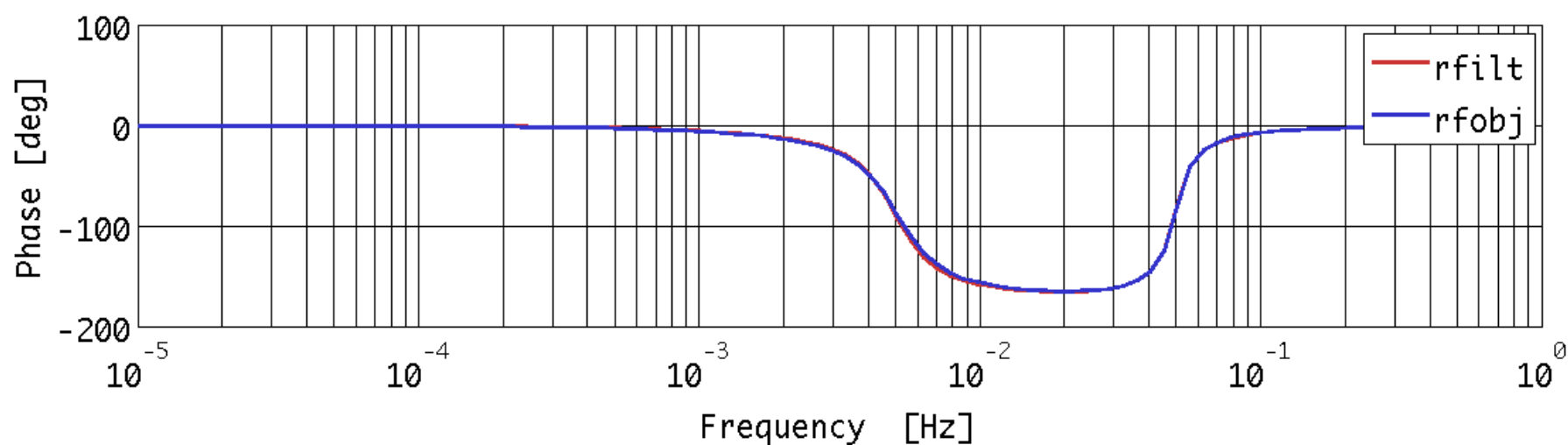
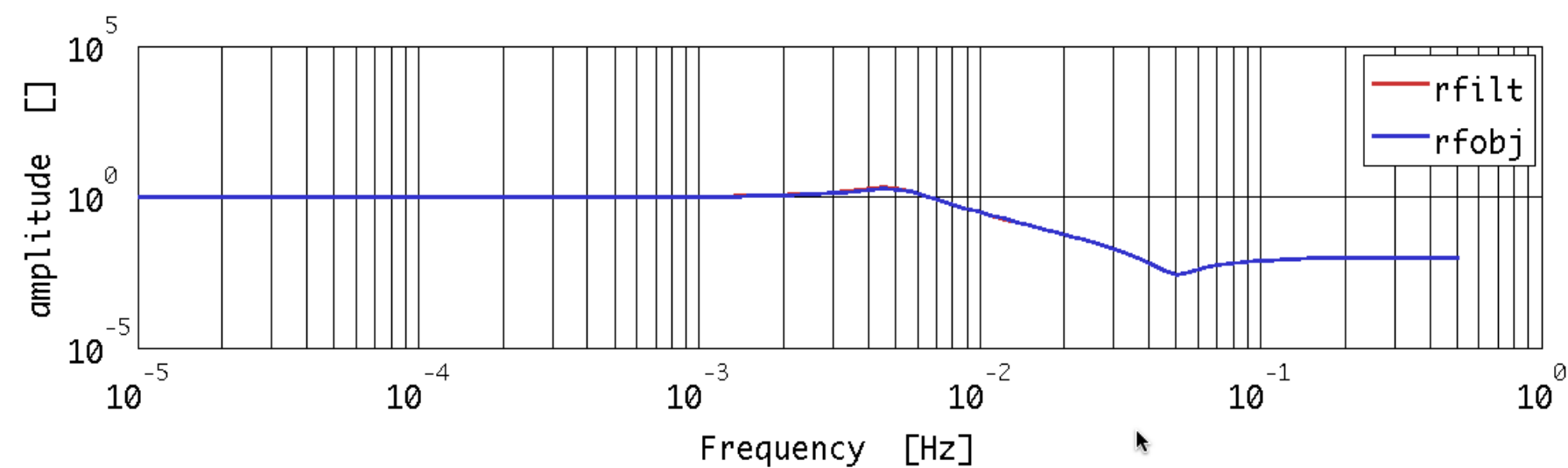
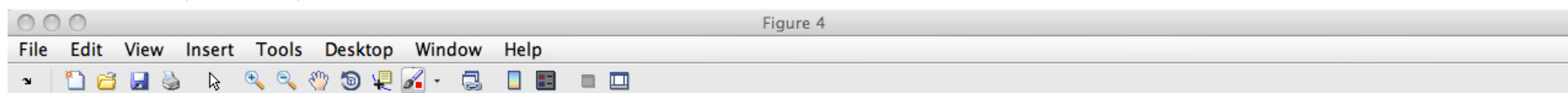
% compute the response of our fitted filter bank
rfobj = resp(fobj.objs.filters,plrsp);
rfobj.setName;

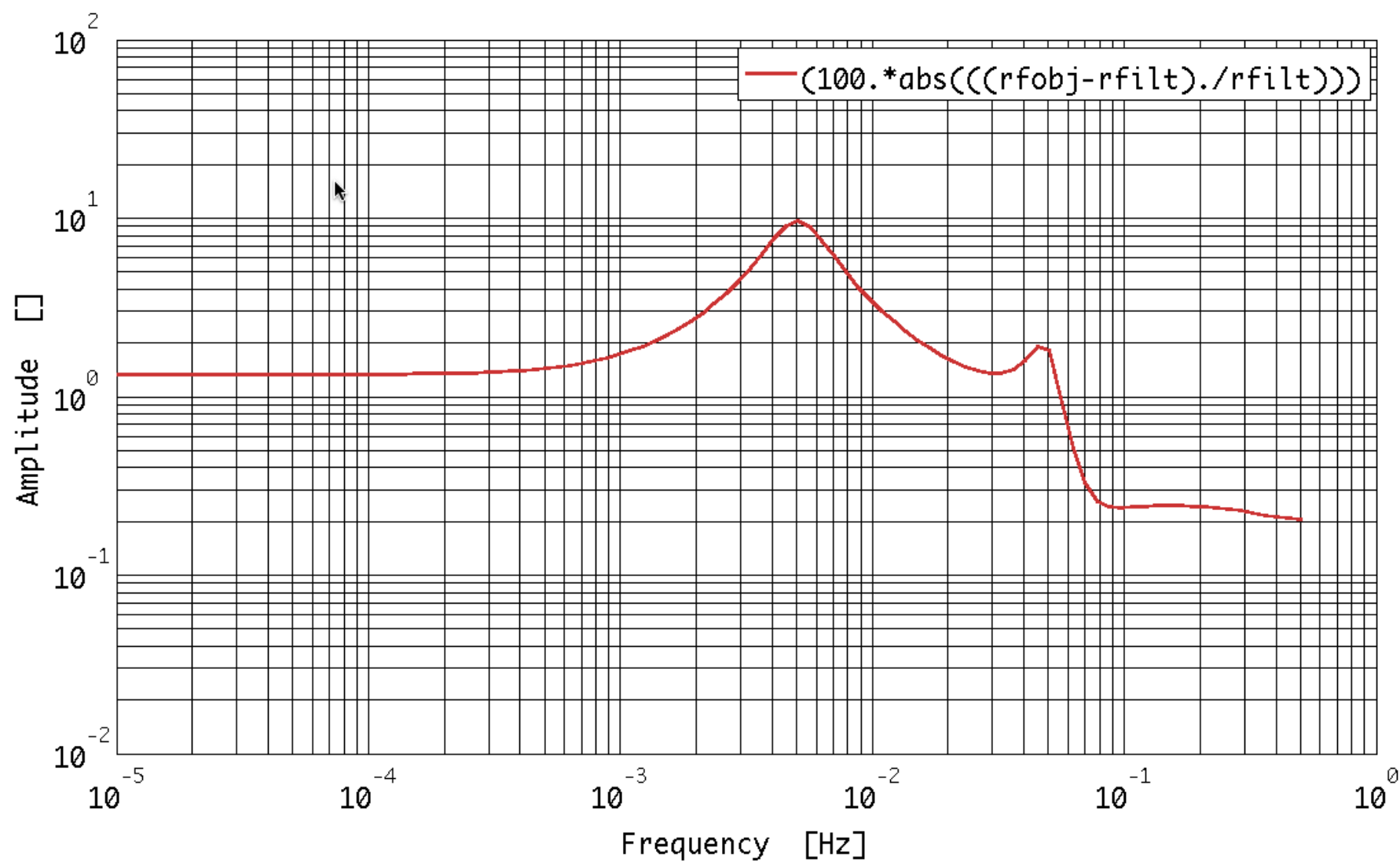
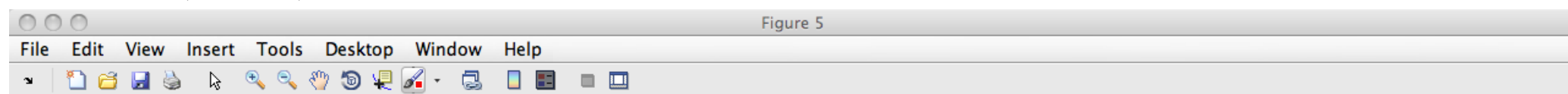
% compare the responses
iplot(rfilt,rfobj)

% and the percentage error on the magnitude
pdiff = 100.*abs((rfobj-rfilt)./rfilt);
pdiff.simplifyYunits;
iplot(pdiff,plist('YRanges',[1e-2 100]))
```

The first plot shows the response of the original filter and the fitted filter bank, whereas the second plot reports the percentage difference between fitted model and target filter magnitude.

As can be seen, the difference between filters magnitude is at most 10%.





Generation of noise with given psd

Generation of model noise is performed with the function `ao/noisegen1D`. Details on the algorithm can be found in [noisegen1D help page](#).

Generation of noise with given psd

During this exercise we will:

1. Load fsdata object from file
2. Fit psd of test data with `zDomainFit`
3. Genarate noise from fitted model
4. Compare results

Let us open a new editor window and load test data.

```
tn = ao(plist('filename', 'topic5/T5_Ex03_TestNoise.xml'));
tn.setName;
```

This command will load an analysis object containing a test time series 10000 seconds long and sampled at 1 Hz.

The command `setName` set the name of the AO to `tn`.

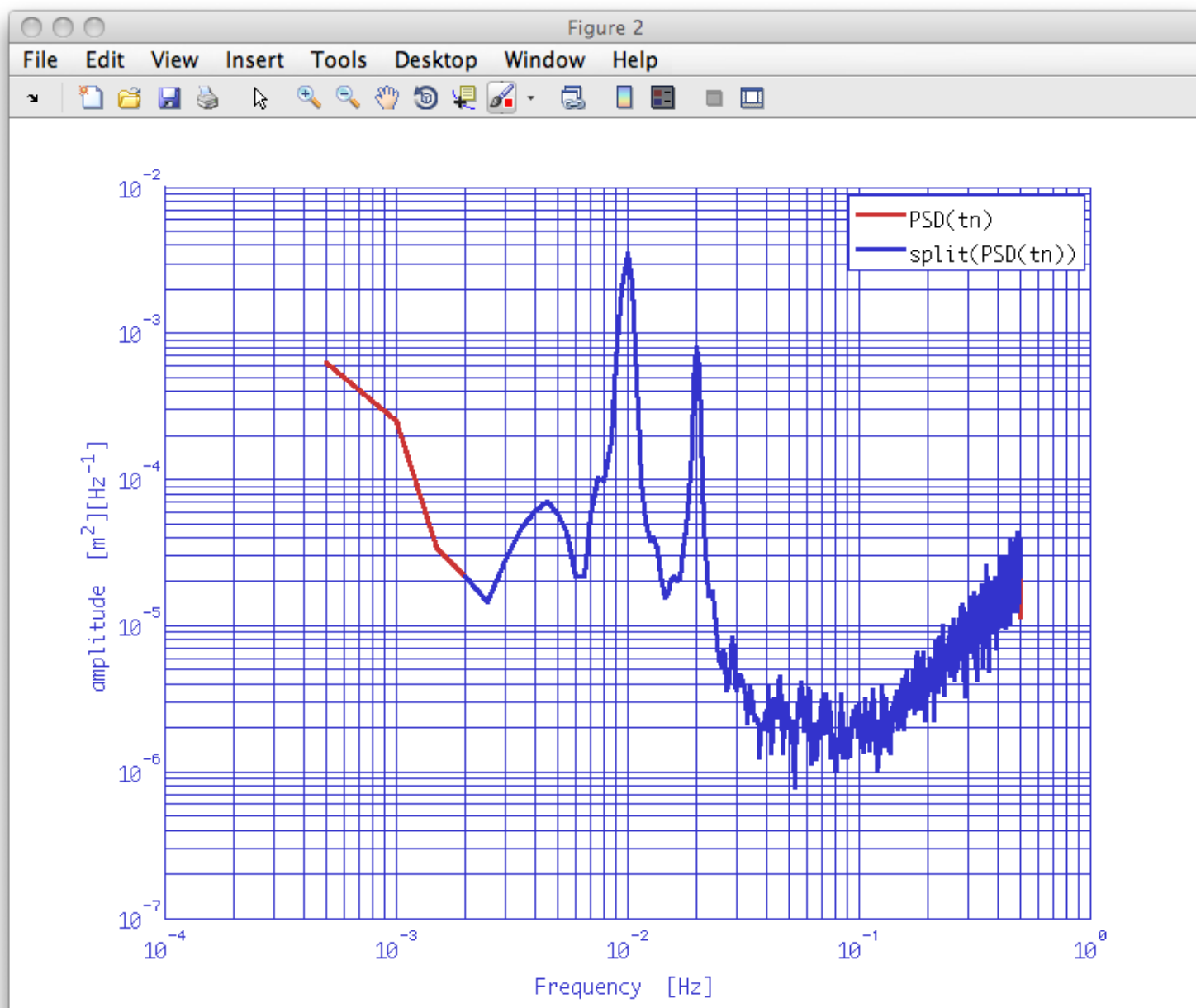
Now do the psd of our data. we need some average otherwise the fitting algorithm is not able to run correctly.

```
tnxx = tn.psd(plist('Nfft', 2000));
```

In order to extract a reliable model from psd data we need to discharge the first bins.

```
tnxxr = split(tnxx,plist('frequencies', [2e-3 5e-1]));
ipplot(tnxx,tnxxr)
```

The result should look like

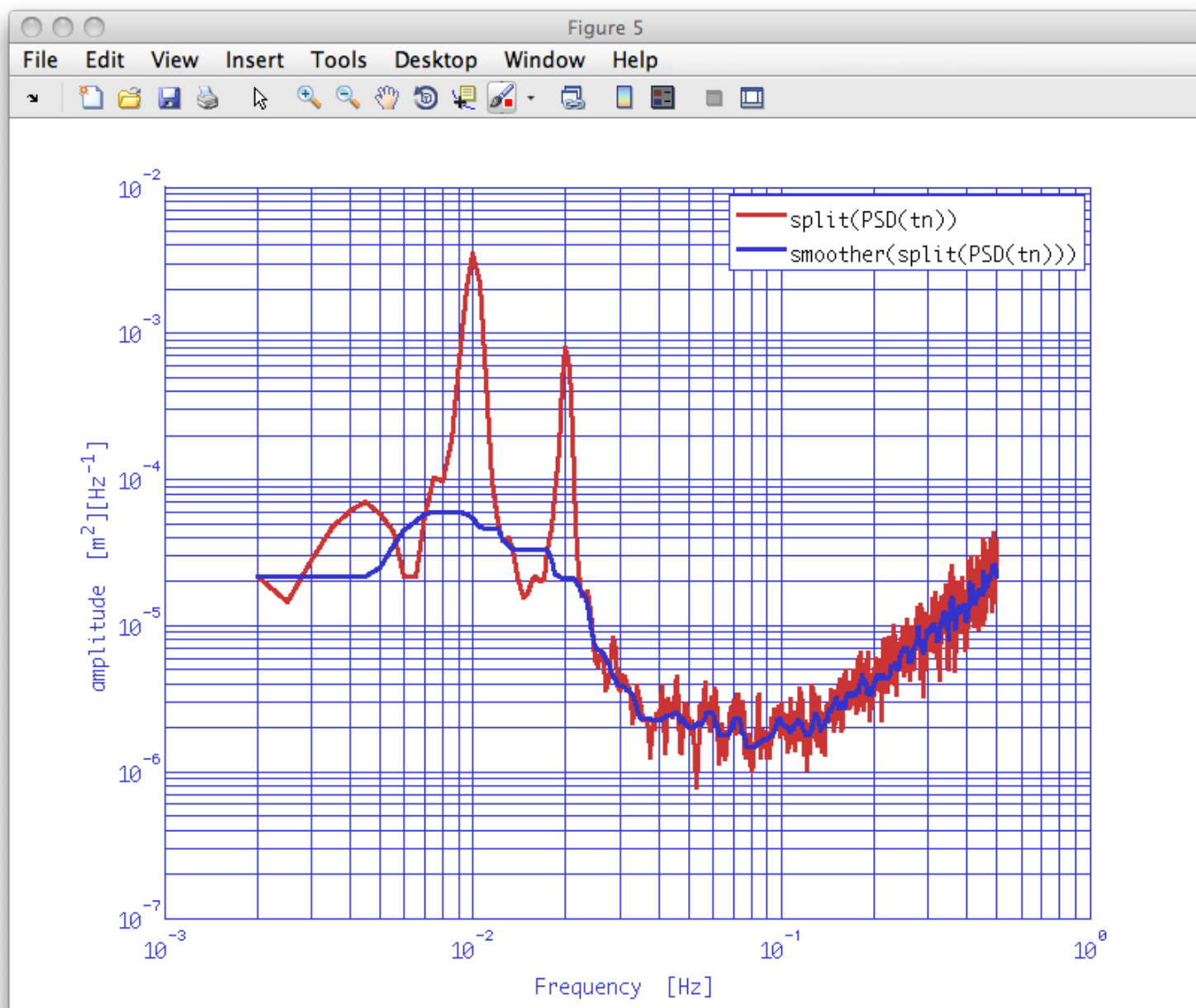


Now it is the moment to fit our PSD to extract a smooth model to pass to the noise generator. First of all we could define a set of proper weights for our fit process. We smooth our PSD data and then define the weights as the inverse of the absolute value of smoothed PSD. This should help the fit function to do a good job with noisy data. It is worth to note that weights are not univocally defined and there could be lots of better ways to define them.

```
stnxx = smoother(tnxxr);
ipplot(tnxxr, stnxx);
wgh = 1./abs(stnxx);
```

The result of the `smoother` method is shown in the plot below:

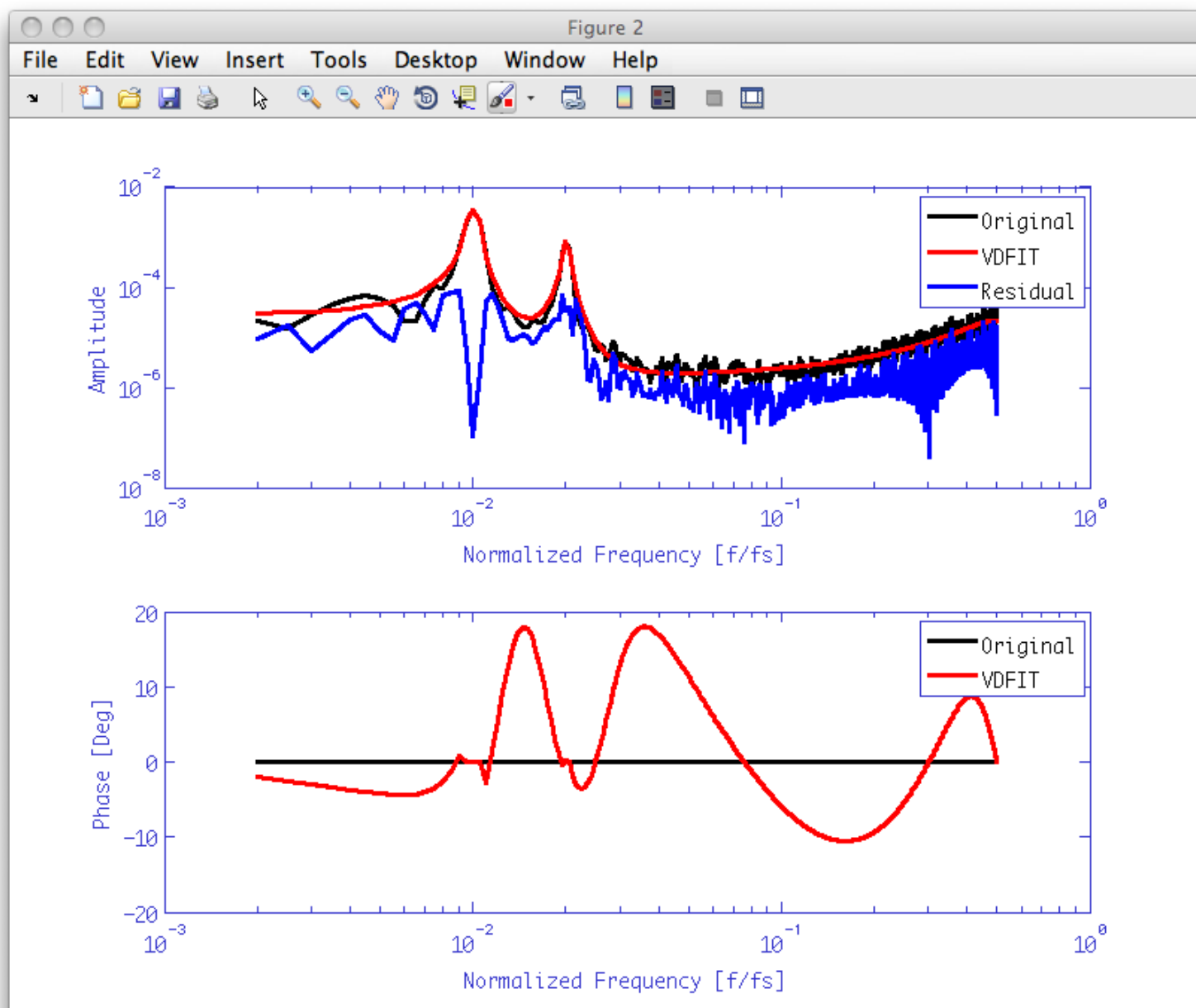
Now let us run an automatic search for the proper model and pass the set of externally defined weights. The first output of `zDomainFit` is a `miir` filter model; the second output is the model response. Note that we are setting **'ResFlat'** parameter to define the exit condition. **'ResFlat'** check the spectral flatness of the absolute value of the fit residuals.



```
plfit = plist('FS',1,...
    'AutoSearch','on',...
    'StartPolesOpt','clog',...
    'maxiter',50,...
    'minorder',10,...
    'maxorder',45,...
    'weights',wgh,... % assign externally calculated weights
    'ResLogDiff',[],...
    'ResFlat',0.77,...
    'RMSE',5,...
    'Plot','on',...
    'ForceStability','off',...
    'CheckProgress','off');

% Do the fit
[param,fmod] = zDomainFit(tnxxr,plfit);
```

Fit result should look like



We can now start the noise generation process. The first step is to generate a white time series AO

```
a = ao(plist('tsfcn', 'randn(size(t))', 'fs', 1, 'nsecs', 10000, 'yunits', 'm'));
```

Then start noise coloring process calling `noisegen1D`

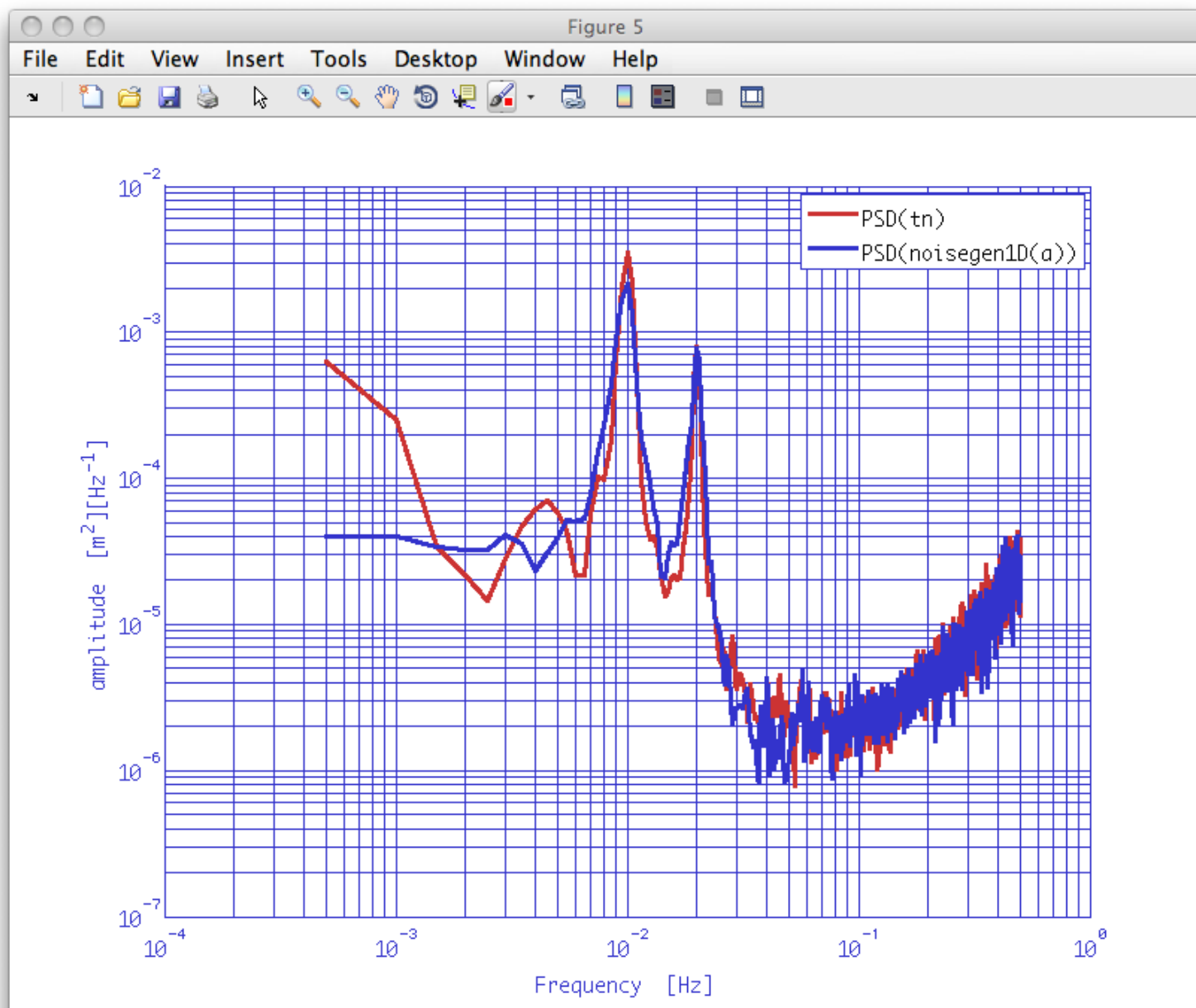
```
plng = plist(...
    'model', abs(fmod), ... % model for colored noise psd
    'MaxIter', 50, ... % maximum number of fit iteration per model order
    'PoleType', 2, ... % generates complex poles distributed in the unitary circle
    'MinOrder', 20, ... % minimum model order
    'MaxOrder', 50, ... % maximum model order
    'Weights', 2, ... % weight with 1/abs(model)
    'Plot', false, ... % on to show the plot
    'Disp', false, ... % on to display fit progress on the command window
    'RMSEVar', 7, ... % Root Mean Squared Error Variation
    'FitTolerance', 2); % Residuals log difference

ac = noisegen1D(a, plng);
```

Let us check the result. Calculate the PSD of the generated noise and compare it with the PSD of starting data.

```
acxx = ac.psd(plist('Nfft', 2000));
iplot(tnxx, acxx)
```

As can be seen, the result is in quite satisfactory agreement with the original data



◀ System identification in z-domain

Fitting time series with polynomials ▶

©LTP Team

Fitting time series with polynomials

Fitting time series with polynomials exploits the function `ao/polyfit`. Details on the algorithm can be found in the [appropriate help page](#).

Fitting time series with polynomials

During this exercise we will:

1. Load time series noise
2. Fit data with `ao/polyfit`
3. Check results

Let's open a new editor window and load test data.

```
a = ao(plist('filename', 'topic5/T5_Ex04_TestNoise.xml'));
a.setYunits('');
a.setName;
```

Try to fit data with `ao/polyfit`. We decide to fit with a 6th order polynomial.

```
plfit = plist('N', 6);
p = polyfit(a, plfit, plist('vector_out', false));
```

The variable `p` is a `cdata` analysis object containing the coefficients of the fitted polynomial.

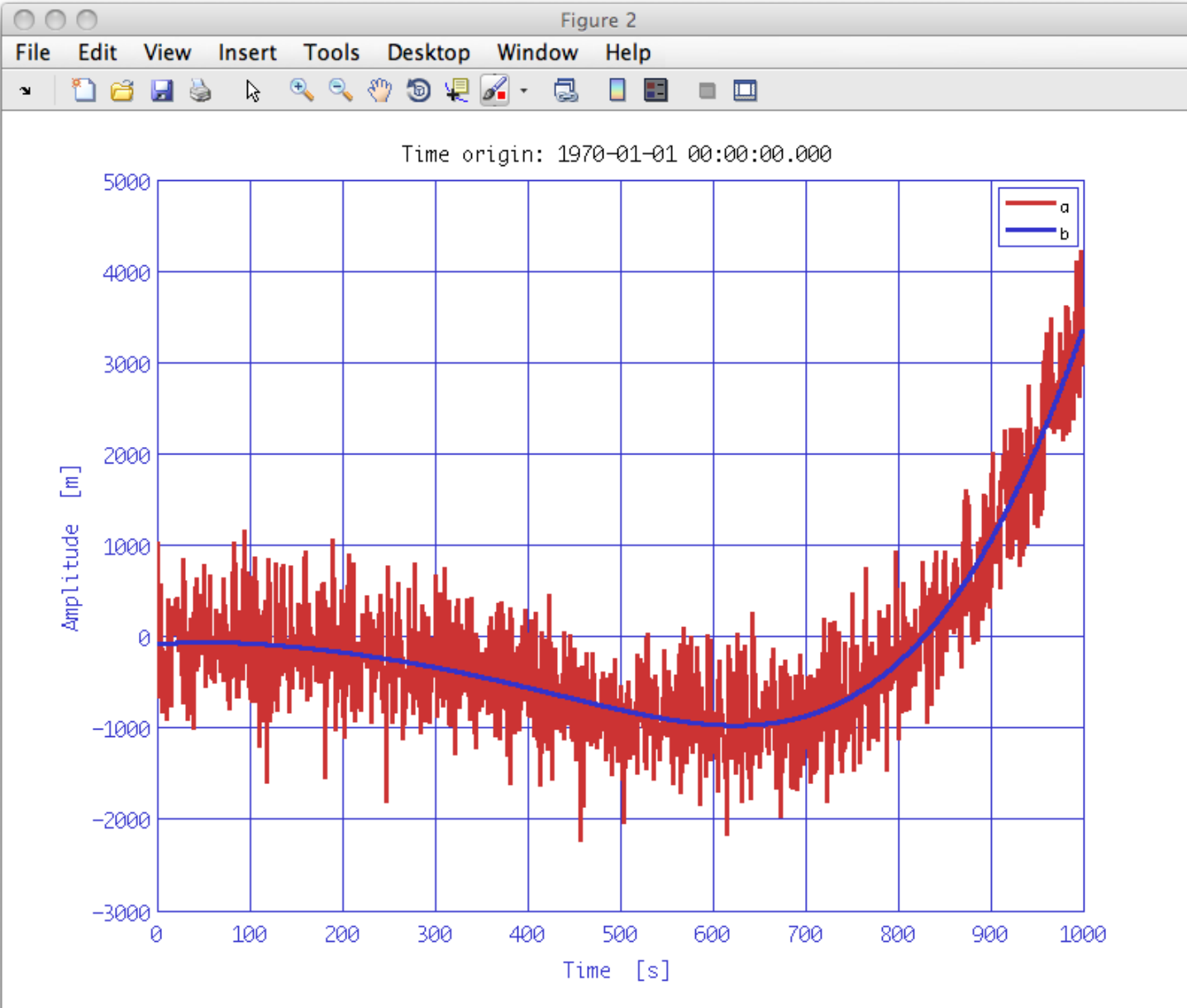
```
----- ao 01: polyfit(a) -----
name: polyfit(a)
data: -3.36e-14 1.00e-10 -9.19e-08 3.98e-05 -0.0123 1.0209 -87.6426
----- cdata 01 -----
      y: [1x7], double
      dy: [0x0], double
      yunits: []
-----
hist: ao / polyfit / SId: polyfit.m,v 1.32 2009/08/27 19:31:28 hewitson Exp S
mdlfile: empty
description:
  UUID: eeed5493-1695-40d3-bb63-0268d275b52a
-----
```

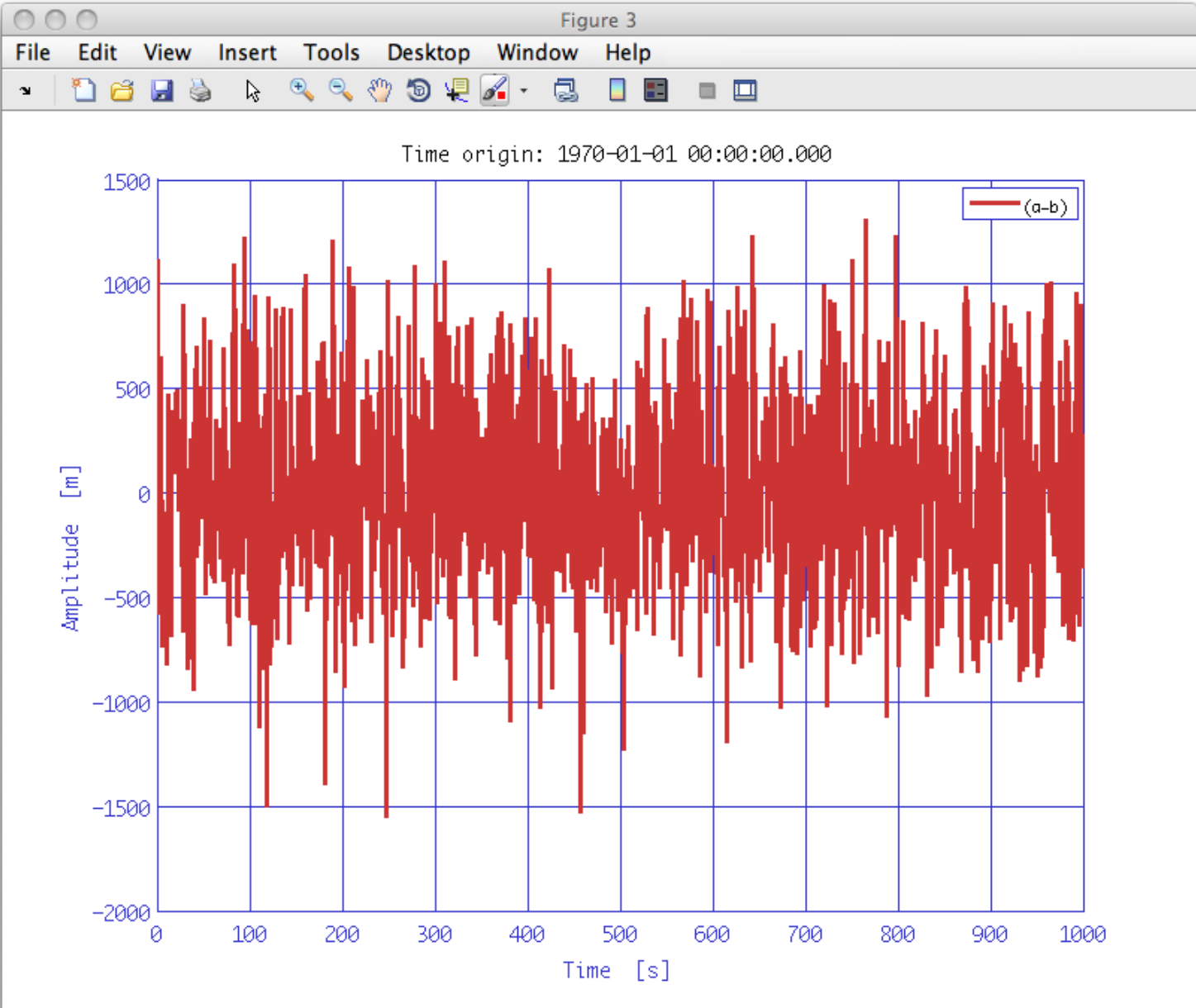
Once we have the model coefficients, we can evaluate the model. We simply need to construct an analysis object with the parameters `polyval` and `t`. We pass as `polyval` the AO containing the fit coefficients and as `t` the AO with original data so that time base of the objects will be the same.

```
b = ao(plist('polyval', p, 't', a));
b.setYunits(a.yunits);
b.setName;
```

Now, check fit result with some plotting. Compare data with fitted model and look at the fit residuals.

```
ipplot(a,b)
ipplot(a-b)
```





You could also try using `ao/detrend` on the input time-series to yield a very similar result as that shown in the last plot.

Non-linear least square fitting of time series

Non-linear least square fitting of time-series exploits the function `ao/curvefit`.

Non-linear least square fitting of time series

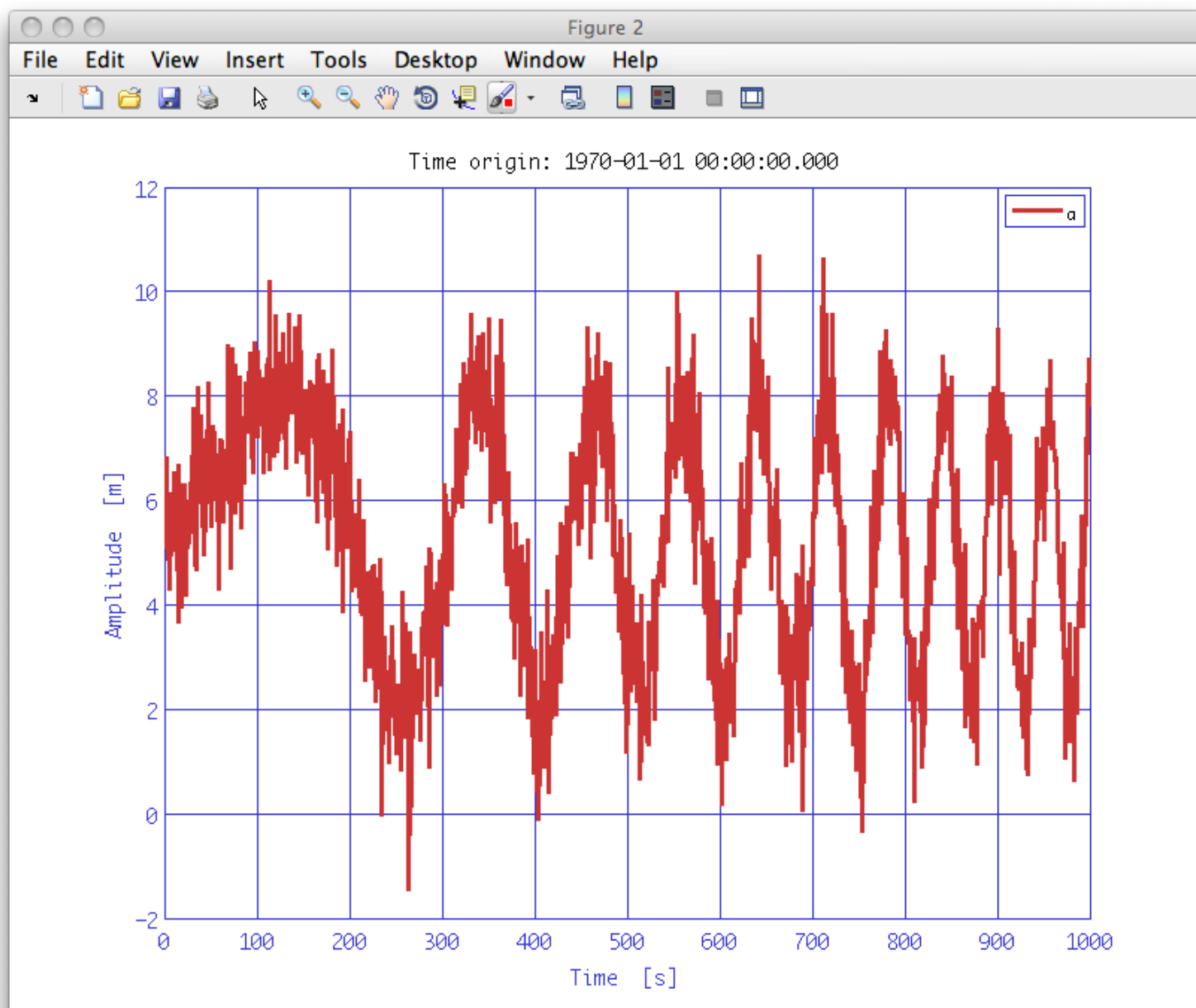
During this exercise we will:

1. Load time series data
2. Fit data with `ao/curvefit`
3. Check results
4. Refine the fit
5. Redo a full parametrized fit

Let us open a new editor window and load test data.

```
a = ao(plist('filename', 'topic5/T5_Ex05_TestNoise.xml'));  
a.setName;  
iplot(a)
```

As can be seen this is a chirped sine wave with some noise.



We could now try the fit. The first parameter to pass to `curvefit` is a fit model. In this case we assume that we are dealing with a linearly chirped sine wave according to the equation:

$$F(t) = A \sin[2\pi(f_0 + kt)t + \varphi]$$

We need to specify a starting guess for the function parameters and, if we want, we can set lower and upper bounds for the fit parameters. Further parameters (unchanged by the fit) could be added as a cell-array in the parameter `ADDP`. In the present case we added a bias of 5 for our data.

The output of `ao/curvefit` is a `cdata` analysis objects containing fit parameters.

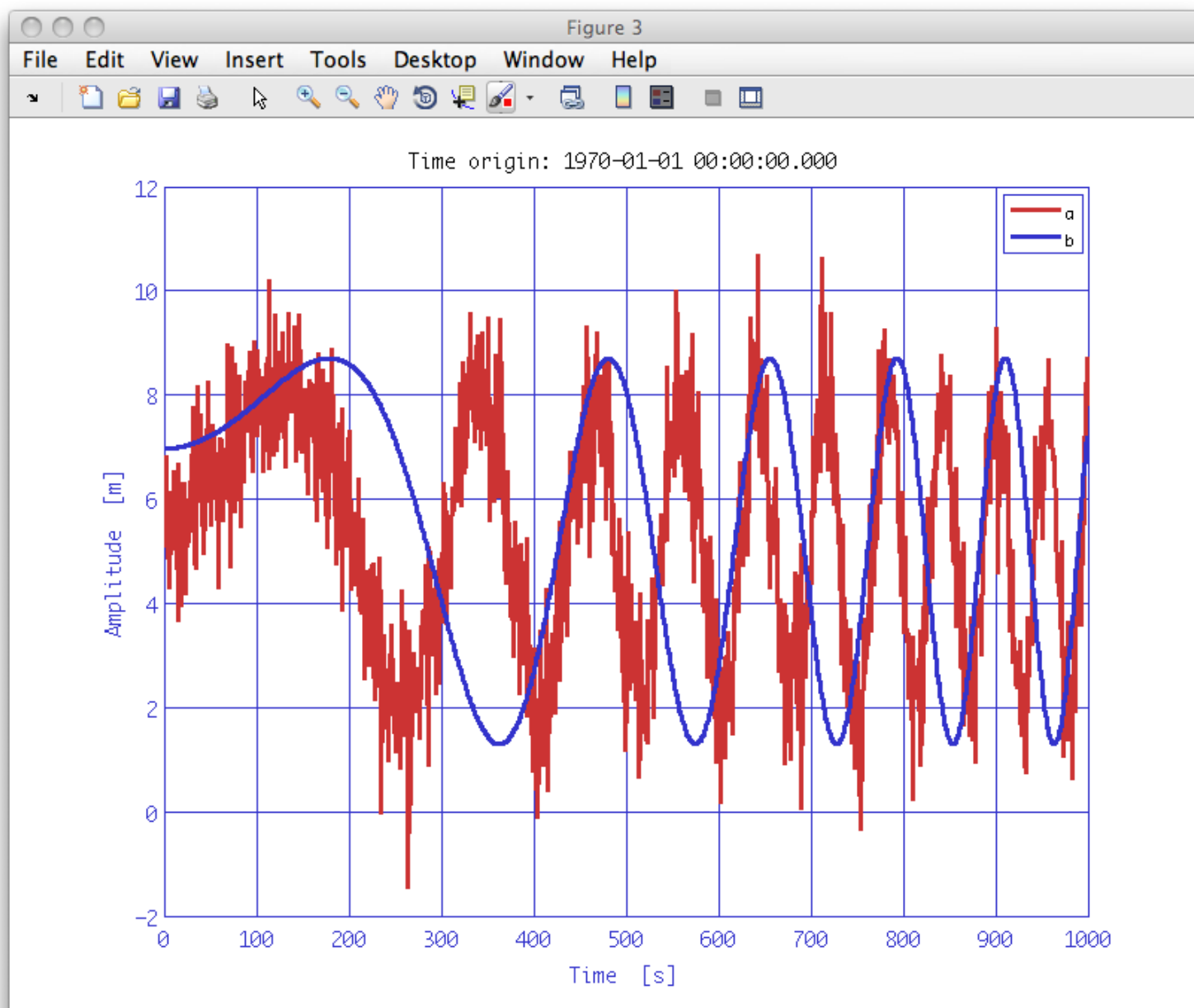
```
plfit = plist('Function', 'ADDP{1} + P(1).*sin(2.*pi.*(P(2) + P(3).*Xdata).*Xdata + P(4))', ...
    'P0', [4 3e-5 5e-6 0.5], ...
    'LB', [1 0 0 -pi], ...
    'UB', [5 1 1 pi], ...
    'ADDP', {5});
params = curvefit(a, plfit);
```

Once the fit is done. We can evaluate our model to check fit results.

```
pleval = plist('Function', 'ADDP{1} + P(1).*sin(2.*pi.*(P(2) + P(3).*Xdata).*Xdata + P(4))', ...
    'Xdata', a, 'tsdata', ...
    'ADDP', {5});
b = evaluateModel(params, pleval);
```

```
b.setYunits(a.yunits);
b.setXunits(a.xunits);
b.setName;
ipplot(a,b)
```

As you can see, the fit is not accurate. One of the great problems of non-linear least square methods is that they easily find a local minimum of the chi square function and stop there without finding the global minimum. So, let us try to look at the data and to refine step by step the fit.



Looking at data we could try to fix some parameters like the amplitude and phase. The y starting level is near 0.4 + bias corresponding to a phase angle around 0.4 rad; the amplitude is close to 3. Now run the fit again with a reduced set of parameters.

```
% Do the fit again
plfit = plfit('Function', 'ADDP{1} + 3.*sin(2.*pi.*(P(1) + P(2).*Xdata).*Xdata + 0.4)', ...
    'P0', [7e-4 9e-6], ...
    'LB', [1e-7 1e-7], ...
    'UB', [1 1e-4], ...
    'ADDP', {5});

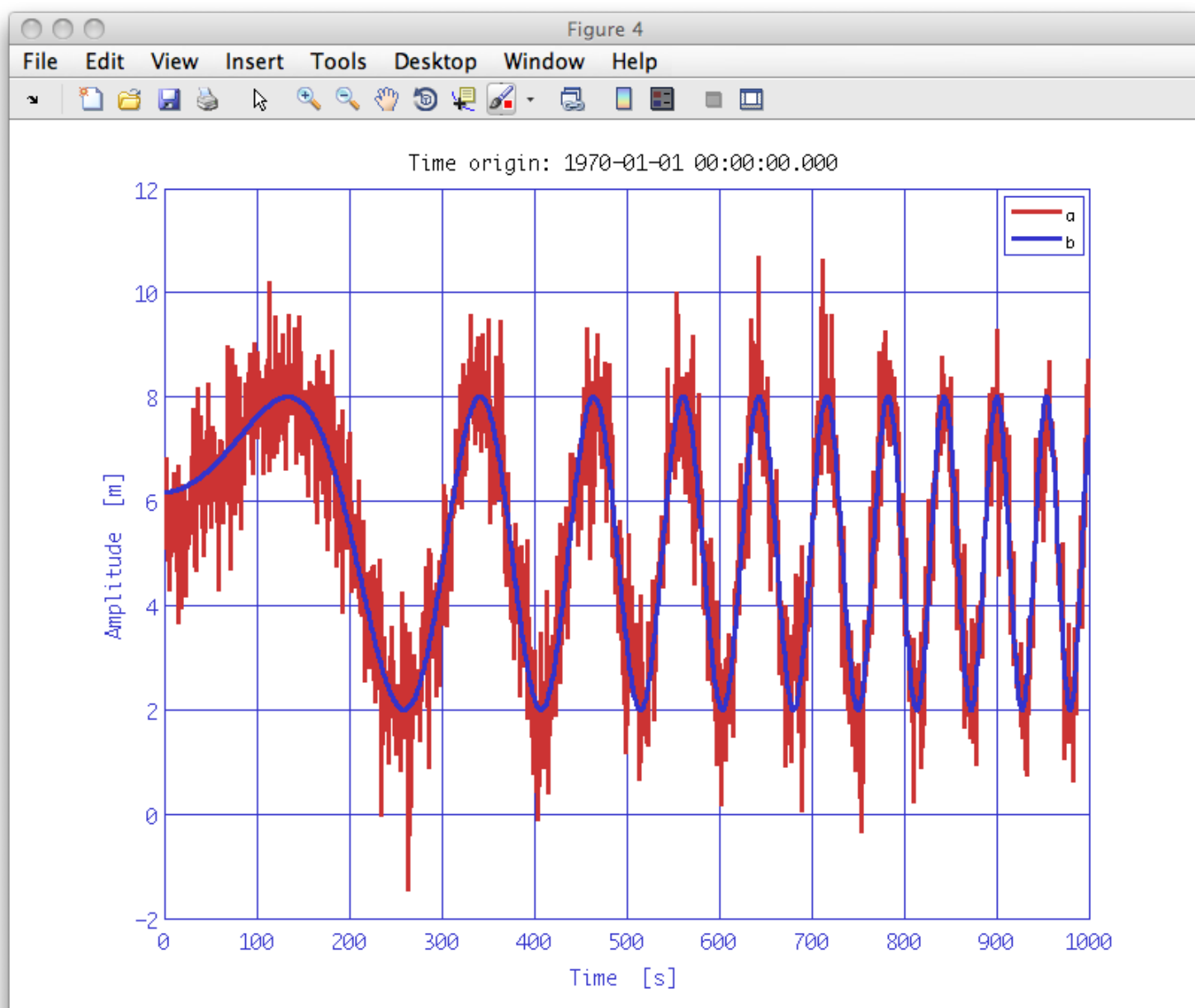
params = curvefit(a, plfit);

% Evaluate the model
pleval = plfit('Function', 'ADDP{1} + 3.*sin(2.*pi.*(P(1) + P(2).*Xdata).*Xdata + 0.4)', ...
    'Xdata', a, ...
    'dtype', 'tsdata', ...
    'ADDP', {5});

b = evaluateModel(params, pleval);
b.setYunits(a.yunits);
b.setXunits(a.xunits);
```

```
b.setName;  
ipplot(a,b)
```

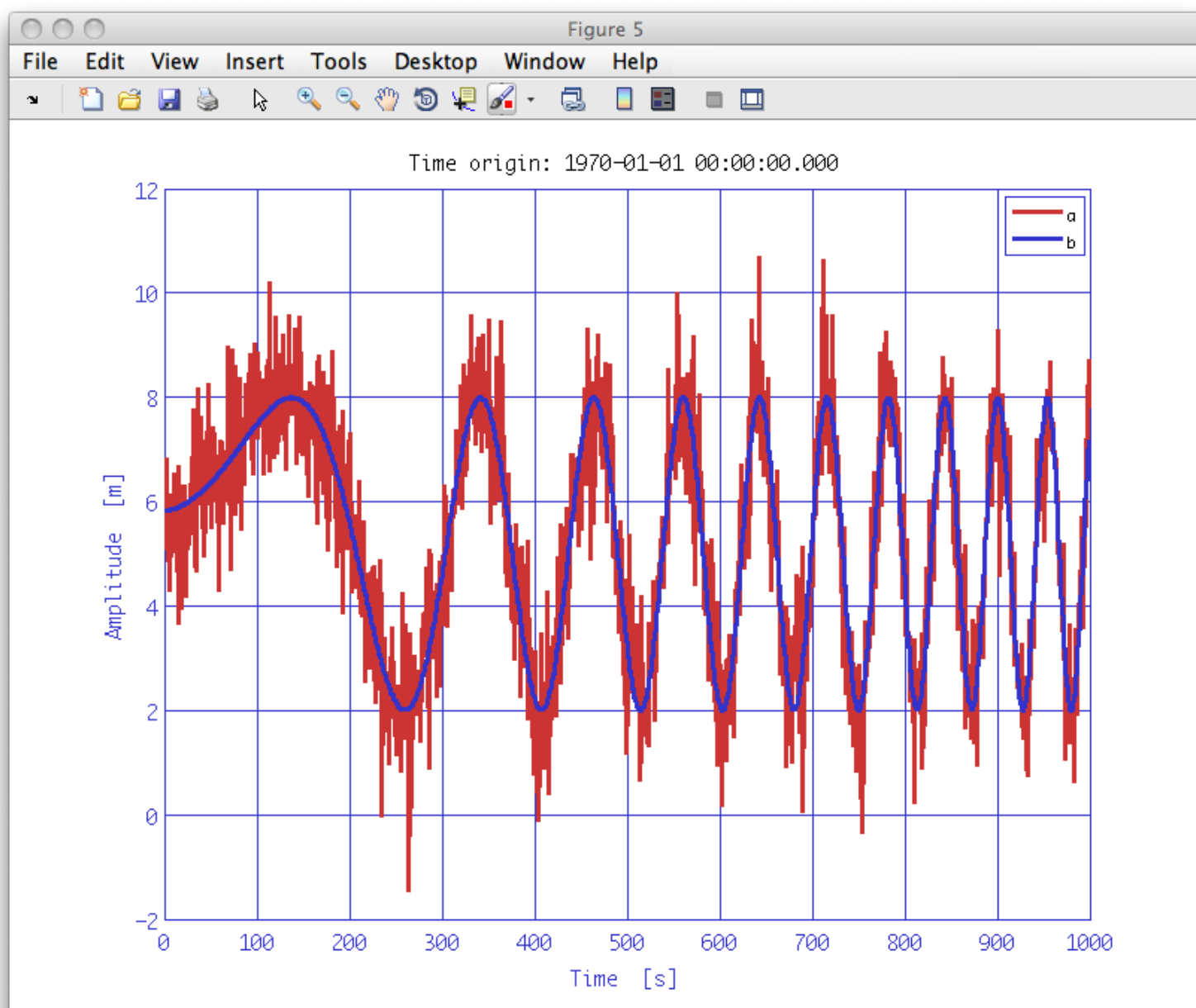
The fit now looks like better...



Actual parameters contained in `params` are now a good starting guess for the wave frequency and chirp parameter.

We could try a new fit with the complete set of parameters...

```
% Do the fit again  
plfit = plist('Function', 'ADDP{1} + P(1).*sin(2.*pi.*(P(2) + P(3).*Xdata).*Xdata + P(4))', ...  
            'P0', [3 5e-5 1e-5 0.4], ...  
            'LB', [2.8 1e-5 1e-6 0.2], ...  
            'UB', [3.2 5e-4 5e-4 0.5], ...  
            'ADDP', {5});  
  
params = curvefit(a, plfit);  
  
% Evaluate the model  
pleval = plist('Function', 'ADDP{1} + P(1).*sin(2.*pi.*(P(2) + P(3).*Xdata).*Xdata + P(4))', ...  
            'Xdata', a, ...  
            'dtype', 'tsdata', ...  
            'ADDP', {5});  
  
b = evaluateModel(params, pleval);  
b.setYunits(a.yunits);  
b.setXunits(a.xunits);  
b.setName;  
  
ipplot(a,b)
```



Let us compare fit results with nominal parameters.
Data were generated with the following set of parameters:

```
ADDP = 5
P(1) = 3
P(2) = 1e-4
P(3) = 1e-5
P(4) = 0.3
```

Fitted parameters are instead:

```
ADDP = 5
P(1) = 2.993
P(2) = 0.000121
P(3) = 9.983e-006
P(4) = 0.278
```

The correlation matrix of the parameters is stored in the `procinfo` (processing information) field of the AO. This field is a `plist` and is used to additional information that can be returned from algorithms. In this case it returns the following `plist`:

Key	Description
COR	The covariance matrix for the parameters.
DPARAMS	A structure of additional information about the

parameters, for example, upper and lower values at the 1-sigma significance level.

GOF A 'goodness of fit' structure which contains fields like, the Chi^2 value and the degrees of freedom.

To extract the covariance matrix from the procinfo you can do

```
params.procinfo.find('cor')
Columns 1 through 3
    1      0.0220543553134523      0.00840698749447142
    0.0220543553134523      1      -0.963274881180157
    0.00840698749447142      -0.963274881180157      1
    -0.0911417933676055      -0.833580057704702      0.692767145487321

Column 4
    -0.0911417933676055
    -0.833580057704702
    0.692767145487321
    1
```

Not so bad!

IFO/Temperature Example – signal subtraction

During this exercise we will:

1. Load the AOs with the IFO and Temperature data
2. Load the transfer function of the data
3. Split the TF to select the meaningful region only
4. Fit the TF with `zDomainFit`
5. Subtract the temperature contribution from the IFO signal

Let us load the test data and split out the 'good' part as we did in Topic 3:

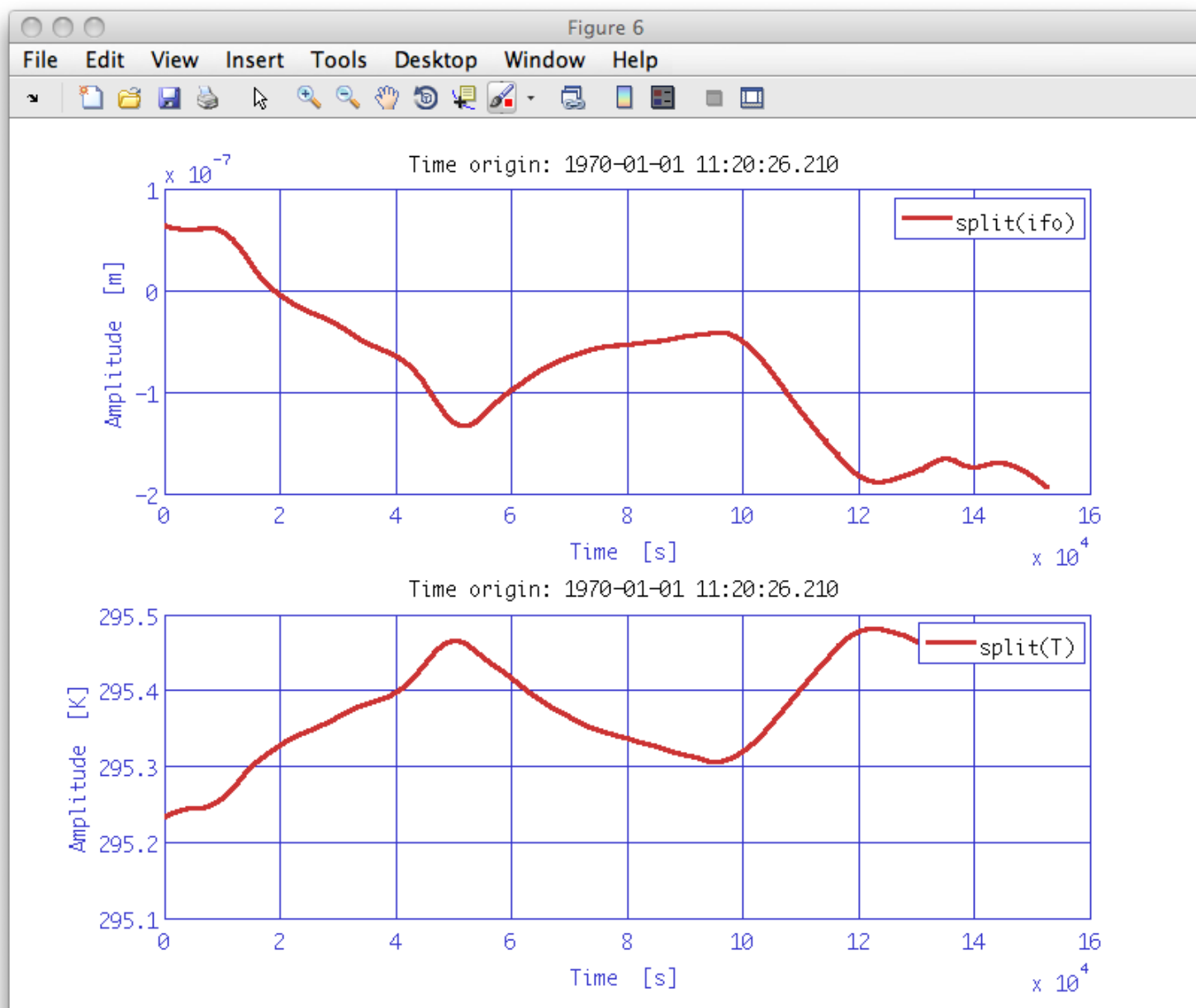
```
ifo = ao(plist('filename', 'ifo_temp_example/ifo_fixed.xml'));
ifo.setName;
T = ao(plist('filename', 'ifo_temp_example/temp_fixed.xml'));
T.setName;

% Split out the good part of the data
pl_split = plist('split_type', 'interval', ...
    'start_time', ifo.t0 + 40800, ...
    'end_time', ifo.t0 + 193500);

ifo_red = split(ifo, pl_split);
T_red = split(T, pl_split);
```

These data are already preprocessed with `ao/consolidate` in order to set the sampling frequency to 1Hz. We could look at the data...

```
iplot(ifo_red, T_red, plist('arrangement', 'subplots'))
```



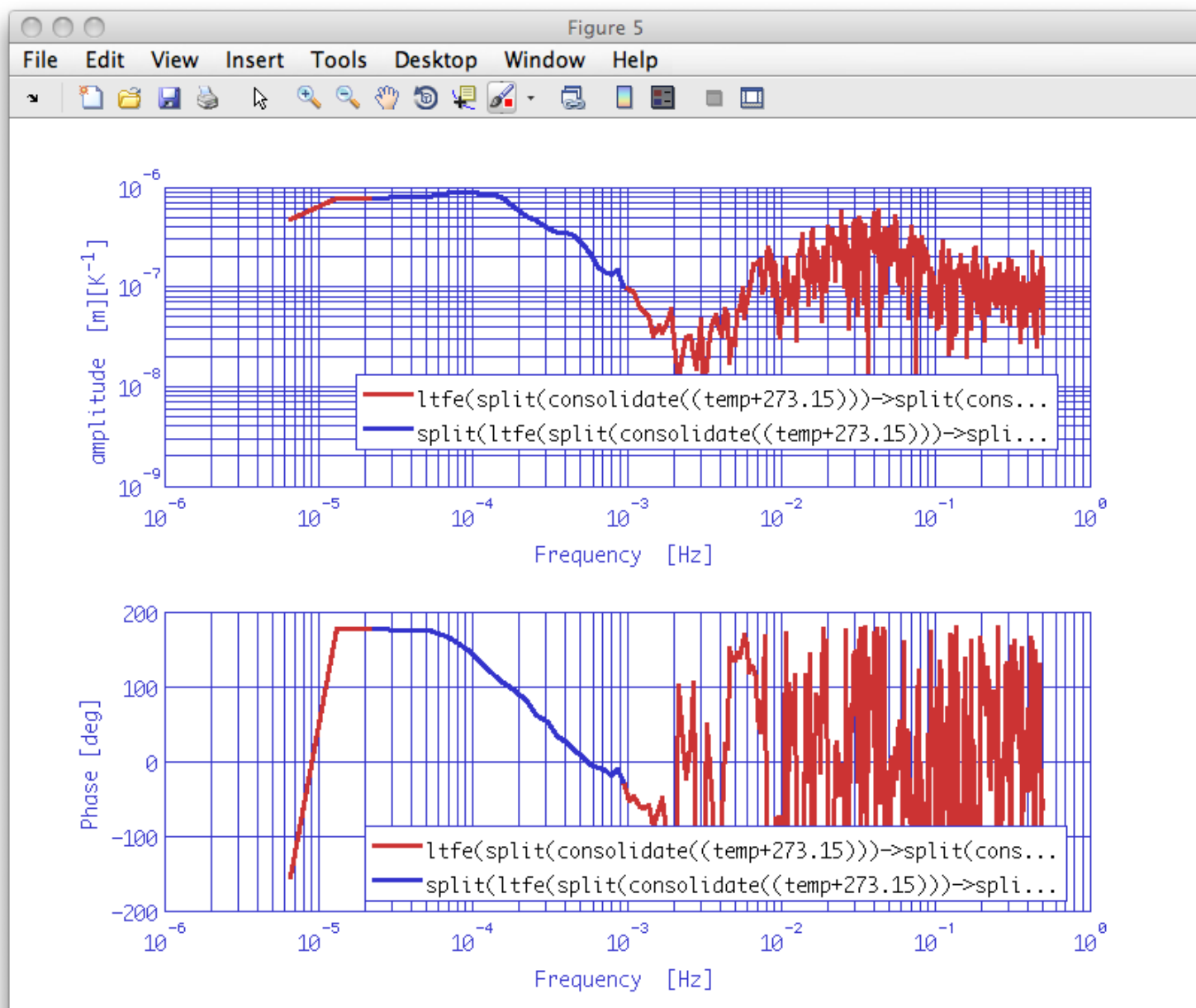
Let us load the transfer function estimate we made in Topic 3.

```
tf = ao('ifo_temp_example/T_ifo_tf.xml');
```

The meaningful frequency region is in the range $2e-5$ Hz – $1e-3$ Hz. Therefore we split the transfer function to extract only meaningful data.

```
tfsp = split(tf,plist('frequencies', [2e-5 1e-3]));
ipplot(tf,tfsp)
```

The plot compares full range TF with splitted TF

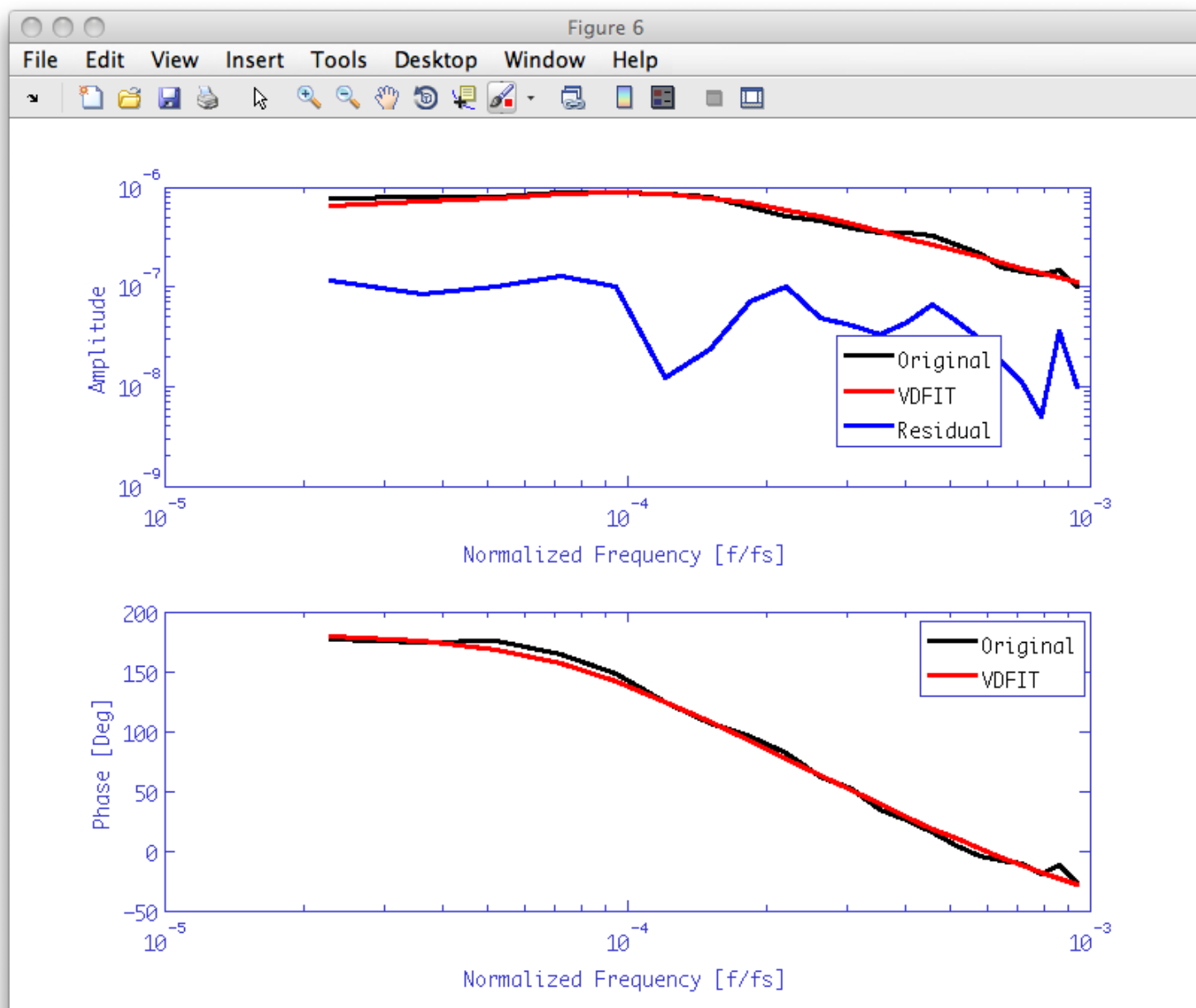


Once we have the proper transfer function, we could start the fitting process. A rapid look to the TF data should convince us that we need a very simple object to fit our data so we could try a fitting session "by hand". In other words, it is more convenient to skip the automatic functionality of `zDomainFit`. Moreover, we force `zDomainFit` to fit a stable model to data because we want to output a stable filter.

```
plfit = plist('FS',1,...
    'AutoSearch','off',...
    'StartPolesOpt','clog',...
    'maxiter',20,...
    'minorder',3,...
    'maxorder',3,...
    'weightparam','abs',...
    'Plot','on',...
    'ForceStability','on',...
    'CheckProgress','off');

fobj = zDomainFit(tfsp,plfit);

% workaround to set the I-/O-units
fobj.objs.filters.setIunits('K');
fobj.objs.filters.setOunits('m');
```

It is time to filter temperature data with the fit output in order to extract temperature contribution to interferometer output. Detrend after the filtering is performed to subtract mean to data (bias subtraction).

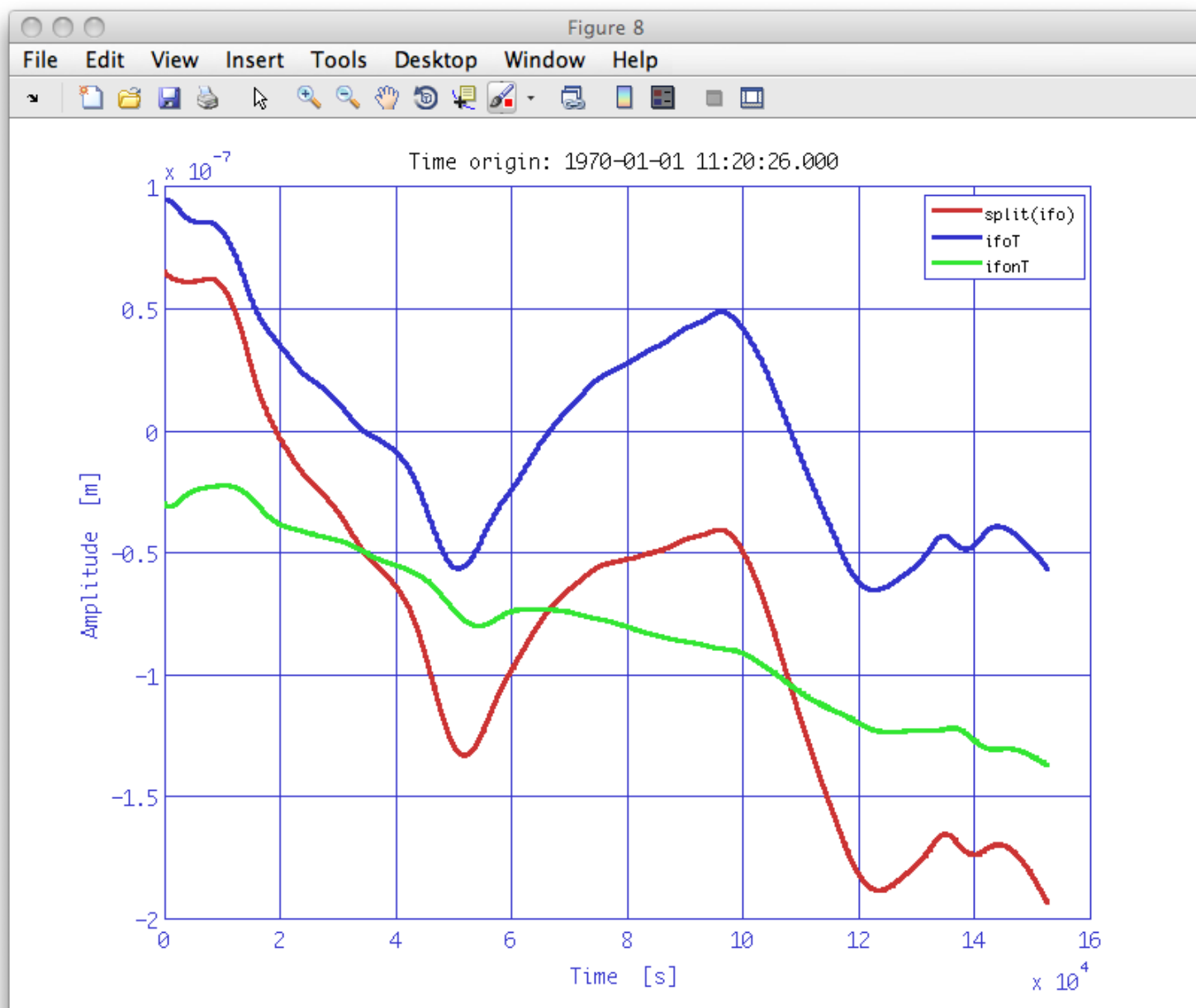
```
ifoT = filter(T_red,fobj,plist('bank','parallel'));
ifoT.detrend(plist('order',0));
ifoT.simplifyUnits;
ifoT.setName;
```

Then we subtract temperature contribution from measured interferometer data

```
ifonT = ifo_red - ifoT;
ifonT.setName;
```

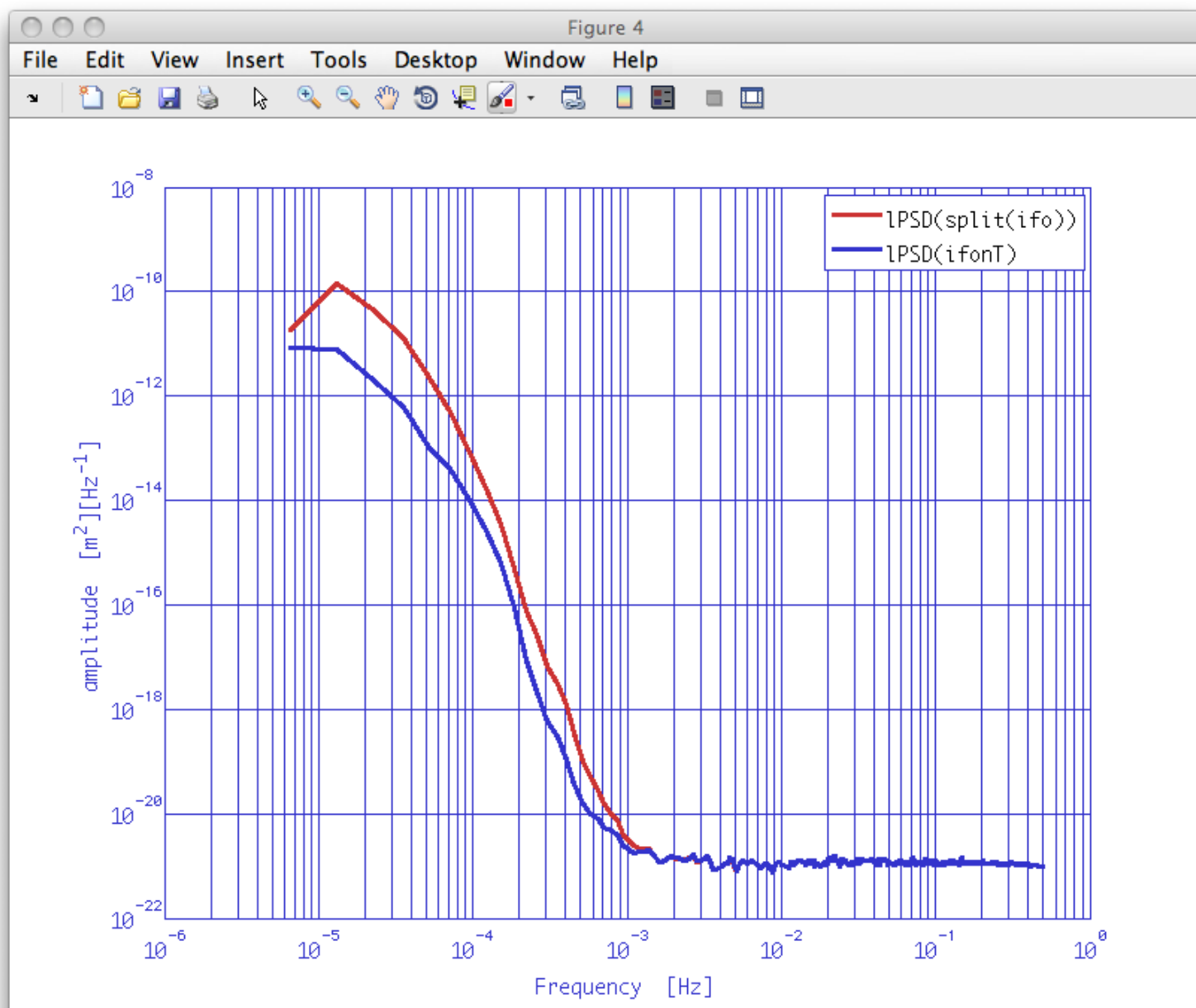
The figure reports measured interferometer data, temperature contribution to interferometer output and interferometer output without thermal drifts.

```
iplot(ifo_red,ifoT,ifonT)
```



If you now compare spectra of the original IFO signal and the one with the temperature contribution removed, you should see something like the figure below:

```
ifoxx = ifo_red.lpsd;
ifonTxx = ifonT.lpsd;
iplot(ifoxx,ifonTxx)
```



◀ Non-linear least square fitting of time series

LTPDA Toolbox ▶

©LTP Team