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

Getting Started with the LTPDA Toolbox

What is the LTPDA Toolbox

This section covers the following topics:

Overview

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

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 (R2008a)	
Signal Processing Toolbox	>6.9	
Database Toolbox	>3.4.1	Needed to interact with an LTPDA repository
Symbolic Math Toolbox	>3.2.3	
Optimization Toolbox	>4.0	

Setting-up MATLAB

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

Add the LTPDA Toolbox to the MATLAB path

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

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

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.

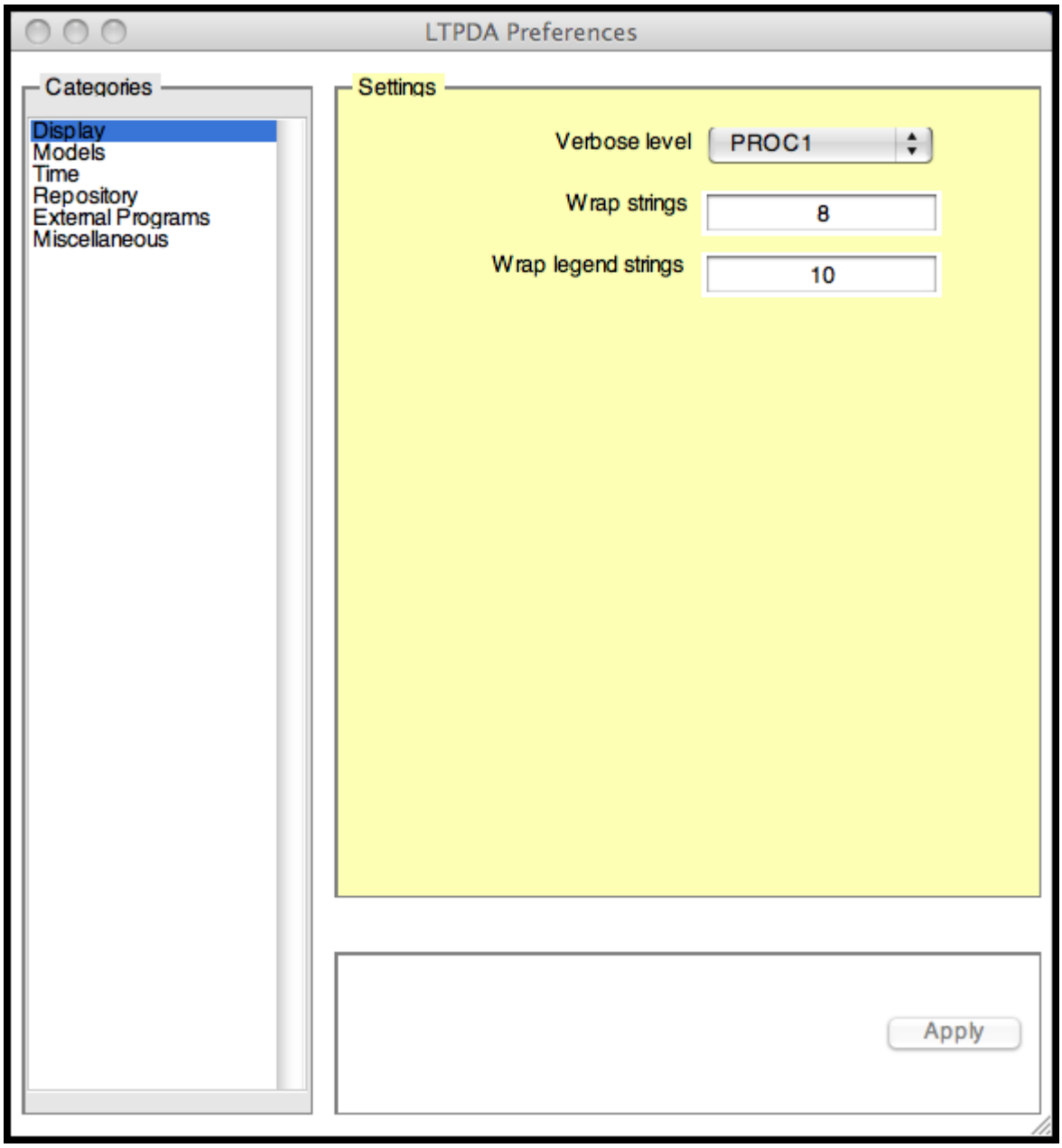
Edit the LTPDA Preferences

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

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

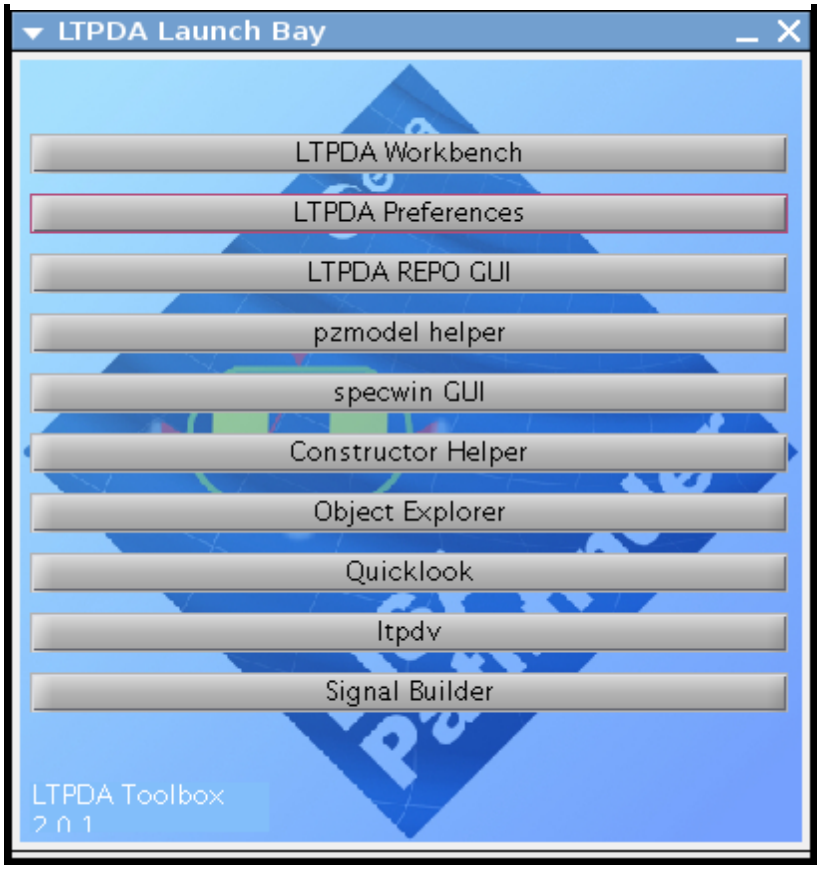
- Installation Guide for Windows
- Installation Guide for Mac OS X
- Installation Guide for Linux

Windows

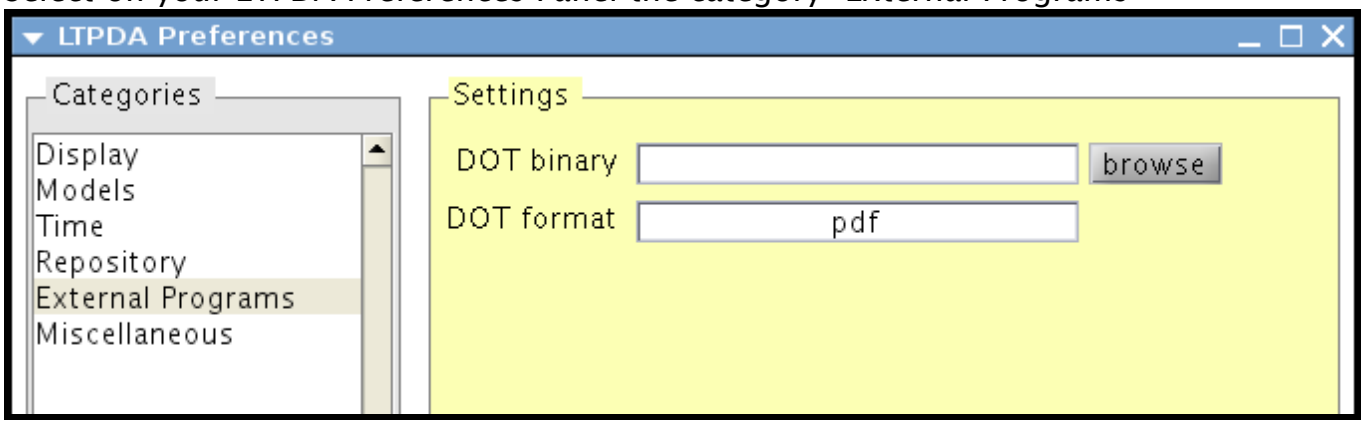
1. Download the relevant package from Downloads section of www.graphviz.org.
2. Install the package by following the relevant instructions.
3. Set the two relevant preferences with your LTPDA Preferences Panel
    1. For this start the LTPDApres

        >> LTPDAprefs

    2. 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 packes 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 programm in MATLAB which opens the file.
   1. The default programm to open a pdf file is the Acrobat Reader
   2. Define another program under File -> Preferences -> Help -> PDF Reader

Setting-up MATLAB Trouble-shooting

Trouble-shooting

A collection of trouble-shooting steps.

1. ## Java Heap Problem

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

   Recommended:

   ```
   -Xmx536870912
   ```

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

   This can be set by performing the following steps:

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

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

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 useing 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/ltpda/src
- Official Microsoft web page
  - Windows 32 bit
  - Windows 64 bit

Additional 3rd-party software Examples

Examples

General

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

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

Constructor examples

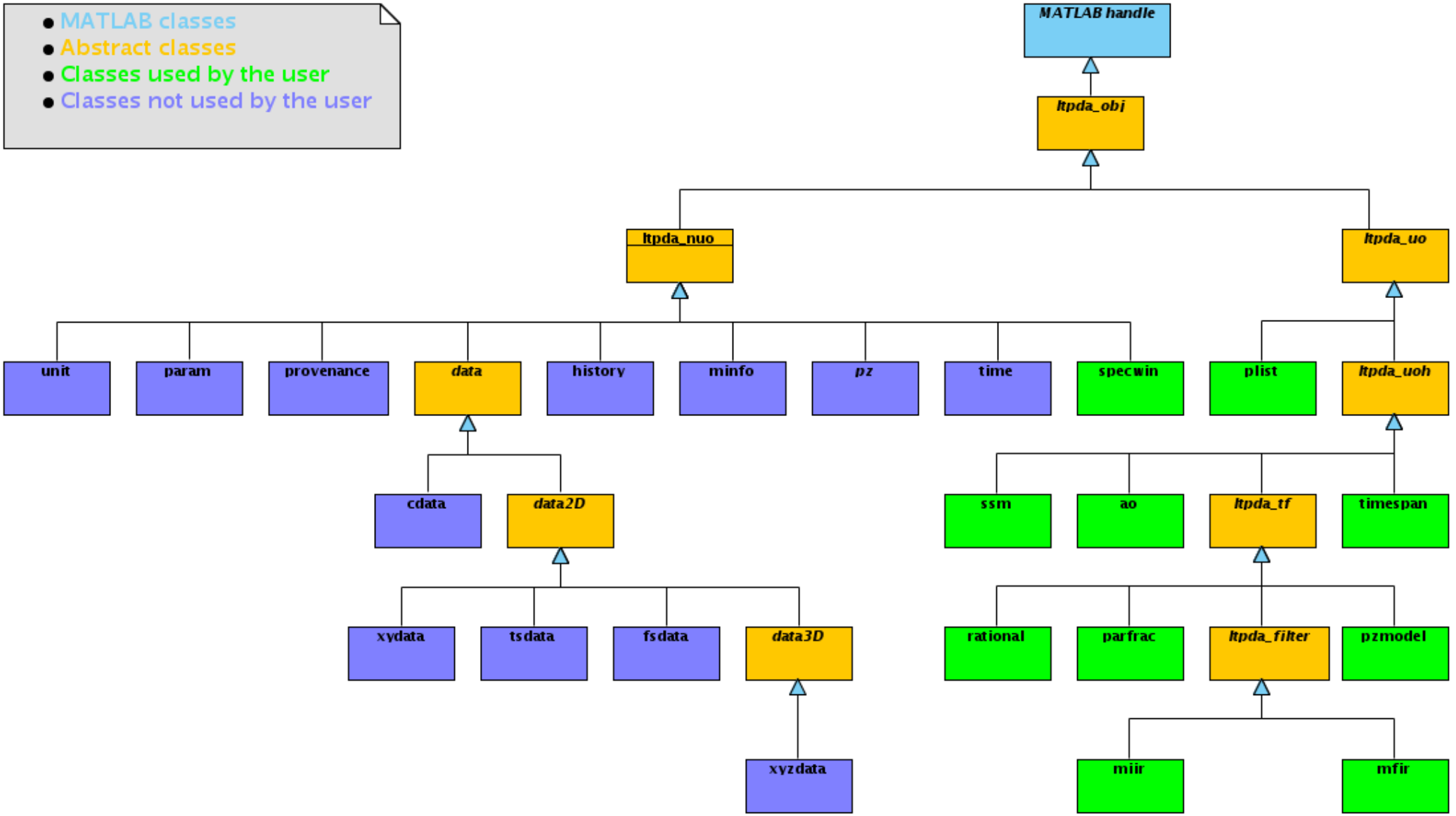
Introducing LTPDA Objects

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

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

LTPDA Classes

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



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:

Examples Creating LTPDA Objects

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.

Introducing LTPDA Objects Working with LTPDA objects

Working with LTPDA objects

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

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

Calling object methods

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
      M: running ao/display
      ----------- ao: a -----------

      name:  none
      creator:  created by hewitson@bobmac.aei.uni-hannover.de[130.75.117.65] on MACI/7.6
(R2008a)/1.9.1 beta (R2008a)
      description:
      data: (0,-1.75921525387737) (0.1,-0.323940403980841) (0.2,1.70580759558634)
(0.3,0.74566737561773) (0.4,-0.386452719524098) ...
      -------- tsdata 01 ------------

      fs:  10
      x:  [1 100], double
      y:  [1 100], double
      xunits:  s
      yunits:  V
      nsecs:  10
      t0:  1970-01-01 00:00:00.000
      -------------------------------

      hist:  ao / ao / $Id: objects_working_content.html,v 1.4 2009/03/19 19:11:14 ingo Exp
$-->$Id: objects_working_content.html,v 1.4 2009/03/19 19:11:14 ingo Exp $
      mfilename:
      mdlfilename:
      -------------------------------
```

Then call the `psd` method:

```
      >> a.psd(pl)
      M: running ao/psd
      M:   using default Nfft of 100
      M:   reset window to BH92(100)
      M:   using default overlap of 66.1%
      M: running ao/display
      ----------- ao: PSD(a) -----------

      name:  PSD(a)
      creator:  created by hewitson@bobmac.aei.uni-hannover.de[130.75.117.65] on MACI/7.6
(R2008a)/1.9.1 beta (R2008a)
      description:
      data: (0,0.0488141703757124) (0.1,0.109407517445348) (0.2,0.194309804548859)
(0.3,0.453109075098881) (0.4,0.650807772380848) ...
      ----------- fsdata 01 -----------

      fs:  10
      x:  [1 51], double
      y:  [1 51], double
      xunits:  empty
      yunits:  V^2/Hz
      t0:  1970-01-01 00:00:00.000
      --------------------------------

      hist:  ao / psd / $Id: objects_working_content.html,v 1.4 2009/03/19 19:11:14 ingo Exp
$
      mfilename:
      mdlfilename:
      --------------------------------
```

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`.

Creating LTPDA Objects Analysis Objects

Analysis Objects

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

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

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

The AO will therefore hold

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

The majority of algorithms in the LTPDA Toolbox will operate on AOs only (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.

Working with LTPDA objects Creating Analysis Objects

Creating Analysis Objects

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

Examples of creating AOs

The following examples show some ways to create Analysis Objects.

Creating AOs from text files.

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

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

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

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

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

Creating AOs from XML or .mat files

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

files.

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

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

Creating AOs from MATLAB functions

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

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

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

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

Creating AOs from functions of time

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

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

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

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

Creating AOs from window functions

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

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

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

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

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

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

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

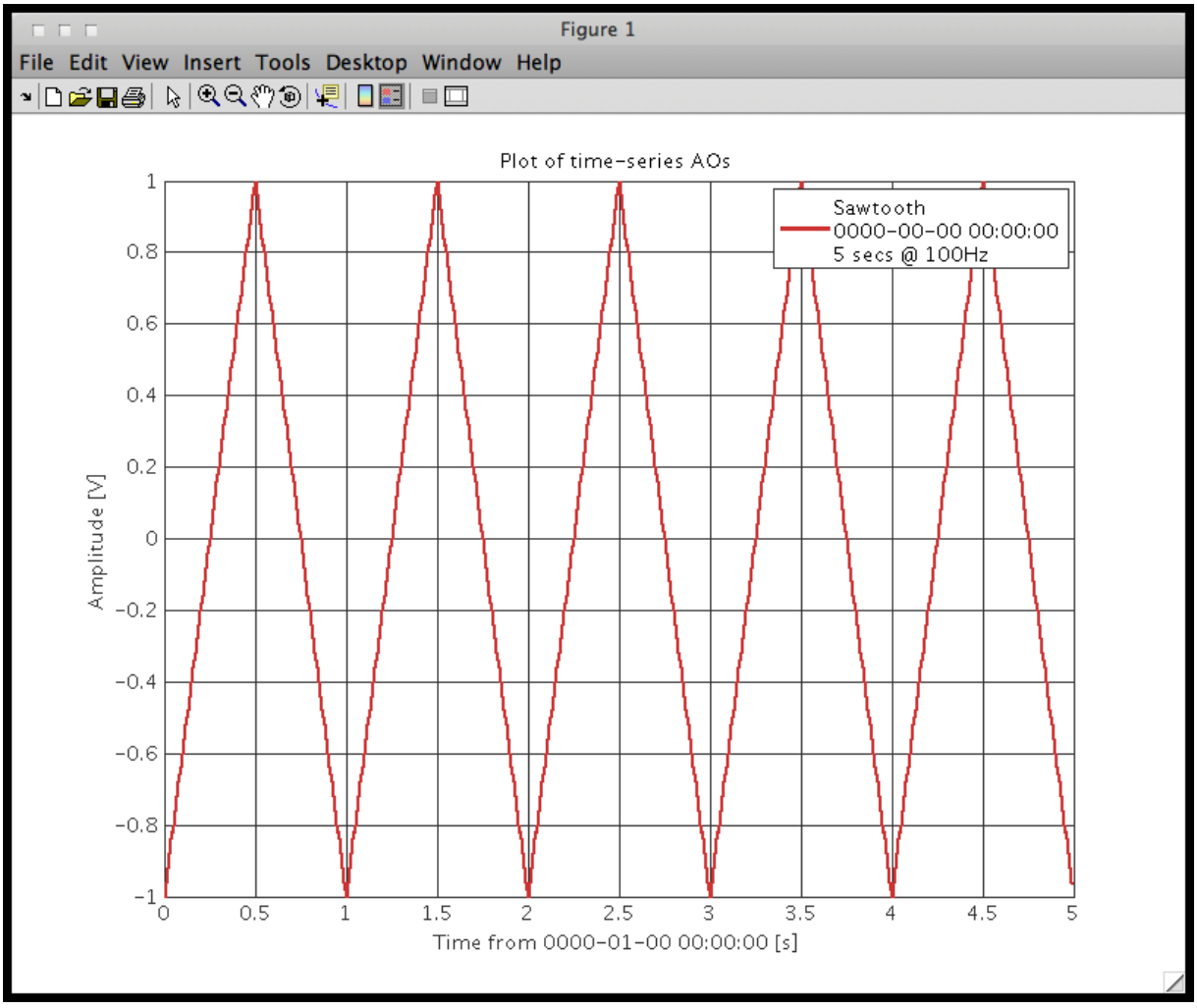
Creating AOs from waveform descriptions

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

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

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

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

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

```
        iplot(asaw);
```



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:

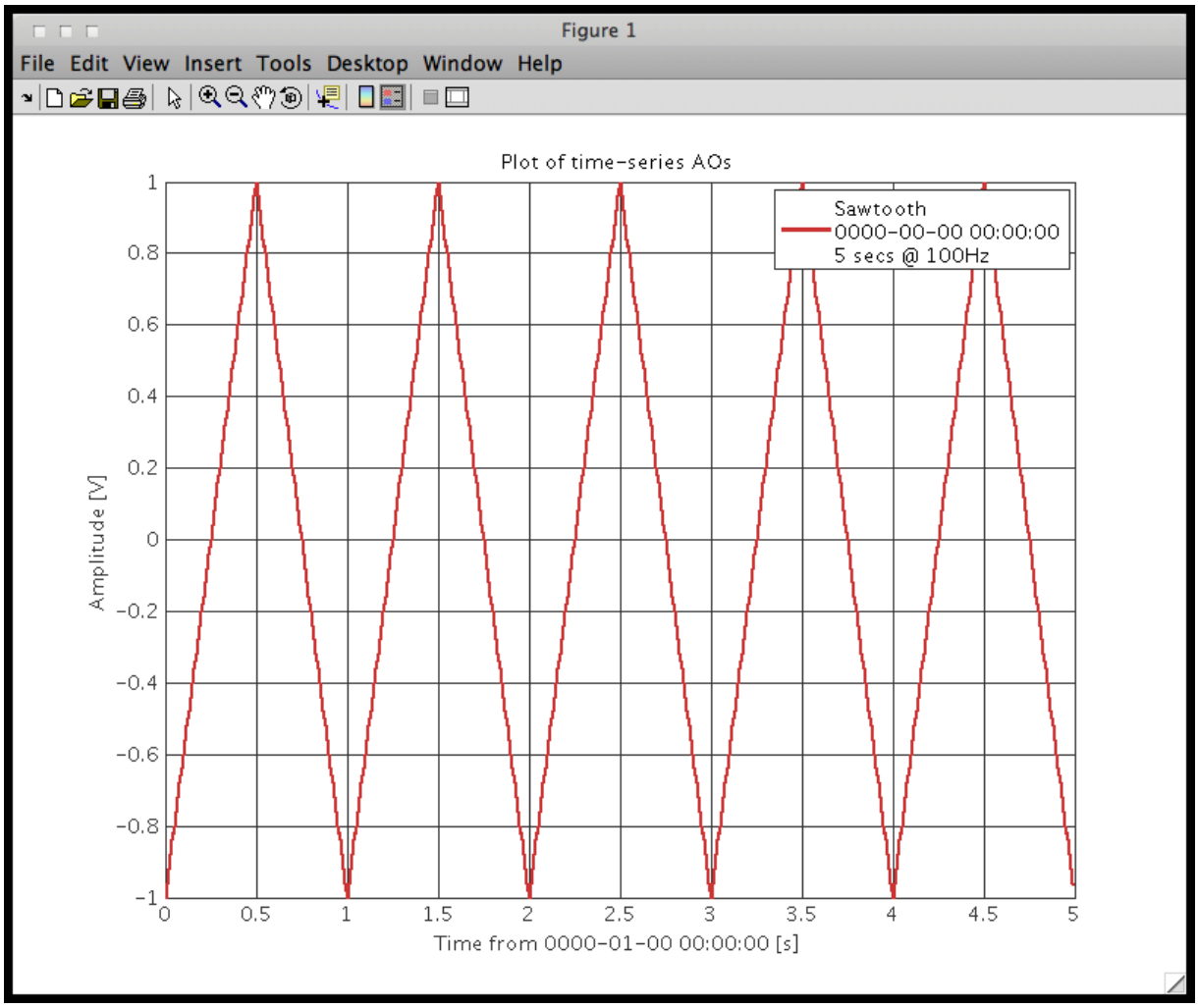
Franklin, Joel N.: *Numerical simulation of stationary and non-stationary gaussian random processes* , SIAM review, Volume { 7}, Issue 1, page 68--80, 1965.

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 frequancy
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)
```

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

```
iplot(asaw);
```



Analysis Objects Saving Analysis Objects

Saving Analysis Objects

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

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

Creating Analysis Objects Plotting Analysis Objects

Plotting Analysis Objects

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

The `iplot` 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.

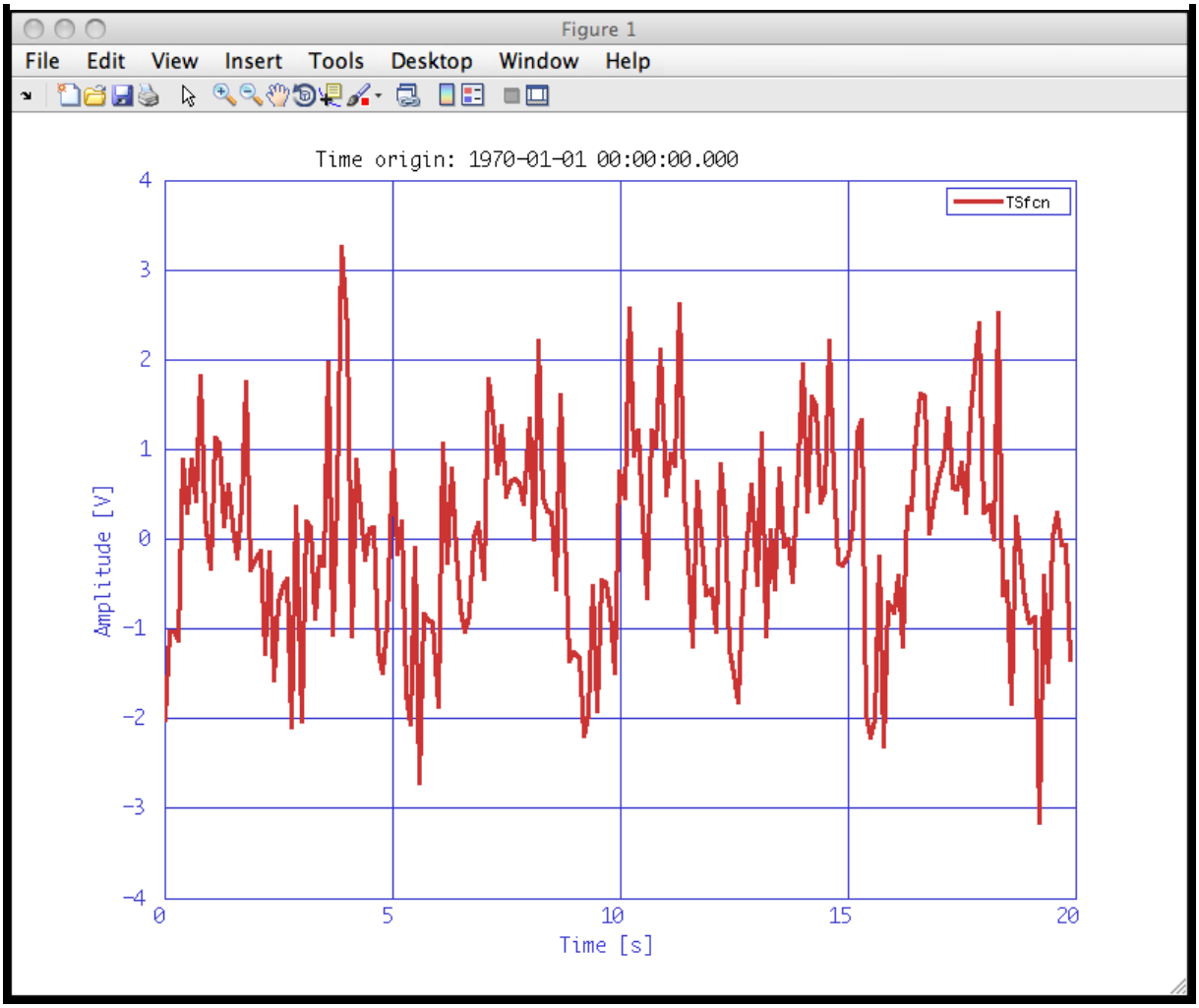
In addition, `iplot` can be configured using a input `plist`. The following examples show some of the possible ways to use `iplot`

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

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

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

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

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

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

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



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

Saving Analysis Objects Parameter Lists

Parameter Lists

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

The following sections introduce parameters and parameter lists:

Creating Parameters

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

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

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

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

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

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

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

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

Parameter Lists Creating lists of Parameters

Creating lists of Parameters

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

Creating parameter lists from parameters.

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

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

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

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


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

Creating parameter lists directly.

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

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

Appending parameters to a parameter list.

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

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

Finding parameters in a parameter list.

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

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

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

Removing parameters from a parameter list.

You can also remove parameters from a parameter list:

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

Setting parameters in a parameter list.

You can also set parameters contained in a parameter list:

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

Combining multiple parameter lists.

Parameter lists can be combined:

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

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

Simulation/modelling

Content needs written...

Creating lists of Parameters Built-in models of LTPDA

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

Built-in Analysis Object Models

Introduction

Built-in Analysis Object models provide a convenient way to add parametric contructors 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

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
```

Built-in Analysis Object Models Generating model noise

Generating model noise

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

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

One way of doing this is to apply digital filters (FIR or IIR) to white input noise.
This approach is 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

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 `T = 1/fs`, where `fs` is the sampling frequency.

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

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

Description

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 CSD

Noise generation with given CSD

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

Multichannel Spectra

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

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

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

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

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

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

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:

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

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

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

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 $\ddot{\vec{\Sigma}}$. A common assumption is $\ddot{\vec{\Sigma}} = \ddot{\vec{1}}$ (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].

Theory

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

$$\ddot{P}_{xx}(\Omega) = \ddot{H}'\left(e^{j\Omega}\right)\ddot{I}\ddot{H}'^{H}\left(e^{j\Omega}\right)$$

$\ddot{H}'(z)$ is a multichannel digital filter that generating colored noise data with given cross-spectrum $\ddot{P}_{xx}(\Omega)$ 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:

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

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

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 noisegen1D

noisegen1D

Description

noisegen1D is a coloring tool allowing the generation of colored noise from withe 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.

Call

```
b = noisegen1D(a, pl);
[b1,b2,...,bn] = noisegen1D(a1,a2,...,an, pl);
```

Inputs

- a - is a tsdata analysis object or a vector of tsdata analysis objects
- pl - is a plist with the input parameters. See the list of function parameters below

Outputs

- b - Colored time-series AOs. The coloring filters used are stored in the objects procinfo field under the parameter 'Filt'.

Algorithm

1. Fit a set of partial fraction z-domain filters using utils.math.psd2tf.
2. Convert to array of MIIR filters.
3. Filter time-series in parallel.

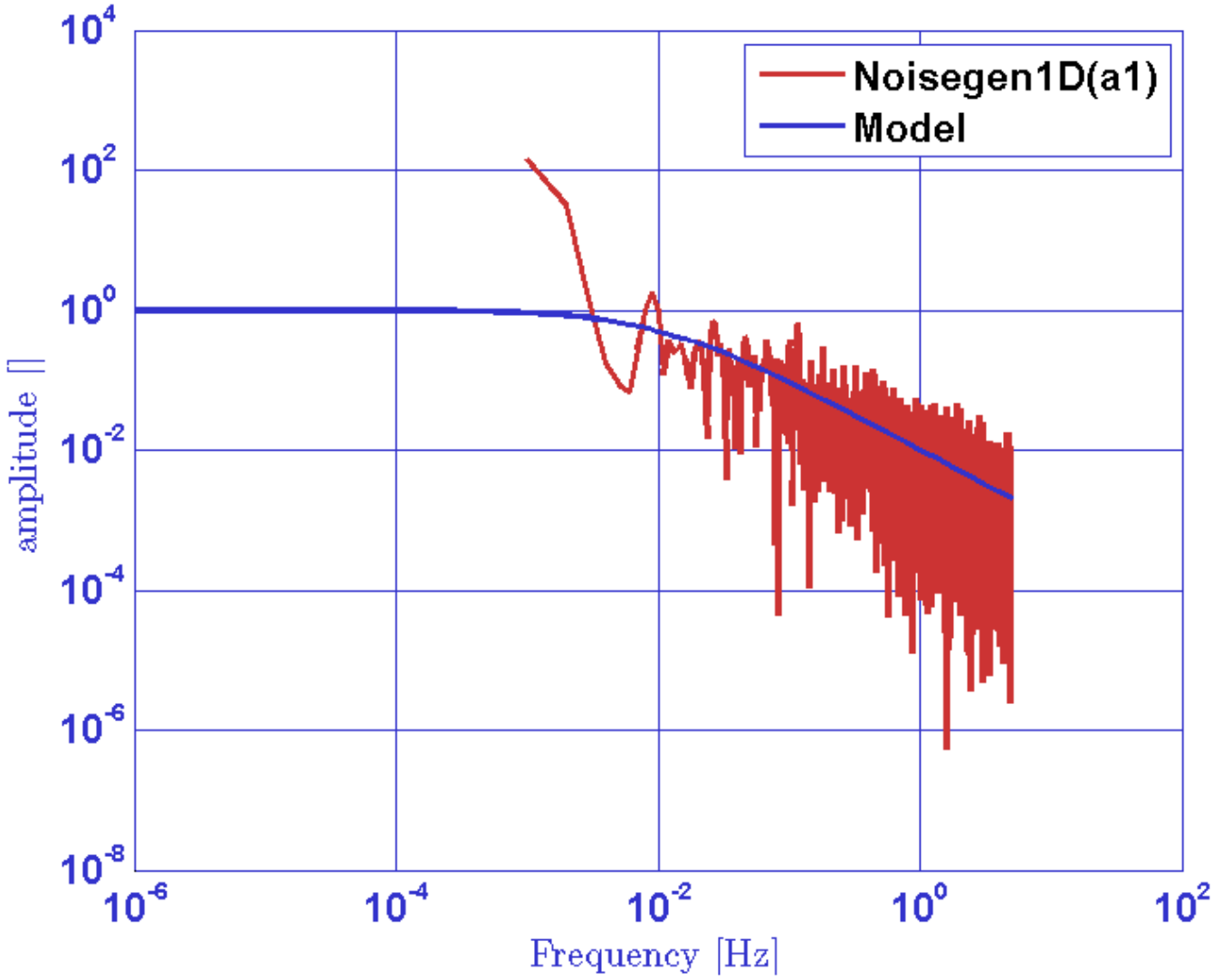
Parameters

- 'Model' - a frequency-series AO describing the model psd.
- 'MaxIter' - Maximum number of iterations in fit routine [default: 30]
- 'PoleType' - Choose the pole type for fitting:
  - 1 - use real starting poles.
  - 2 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+1).
  - 3 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+2) [default].
- 'MinOrder' - Minimum order to fit with. [default: 2].
- 'MaxOrder' - Maximum order to fit with. [default: 25]
- 'Weights' - choose weighting for the fit: [default: 2]
  - 1 - equal weights for each point.

    - 2 - weight with 1/abs(model).
    - 3 - weight with 1/abs(model).^2.
    - 4 - weight with inverse of the square mean spread of the model.
- 'Plot' - plot results of each fitting step. [default: false]
- 'Disp' - Display the progress of the fitting iteration. [default: false]
- 'FitTolerance' - 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 2 order of magnitude lower than data itsleves. [Default: 2].
- 'RMSEVar' - Root Mean Squared Error Variation - Check if the variation of the RMS error is smaller than 10^(-b), where b is the value given to the variable. This option is useful for finding the minimum of Chi squared. [default: 7].

Example

```
%% Noise generation from fsdata model object %%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Description:
% 1) Generate a fsdata object to be used as psd model
% 2) Generate a random series of data (white)
% 3) Generate colored noise with noisegen1D
% 4) calculated psd of generated data
% 5) check result by plotting

% 1)
fs = 10; % sampling frequency
pl_mod1 = plist('fsfcn', '0.01./(0.01+f)', 'f1', 1e-6, 'f2', 5, 'nf', 100);
mod1 = ao(pl_mod1); % fsdata model object

% 2)
% generating white noise
a1 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', 1000));

% 3) Noise generation

pl1 = plist(...
    'model', mod1, ...
    'MaxIter', 30, ...
    'PoleType', 2, ...
    'MinOrder', 10, ...
    'MaxOrder', 20, ...
    'Weights', 2, ...
    'Plot', false,...
    'Disp', false,...
    'RMSEVar', 5,...
    'FitTolerance', 2);

ac1 = noisegen1D(a1, pl1);

% 4)
acxx1 = ac1.psd;
% 5)
iplot(acxx1, mod1);
```



Noise generation with given CSD noisegen2D

noisegen2D

Description

noisegen2D generates colored noise from withe noise with a given cross spectrum. The coloring filter is constructed by a fitting procedure to the models provided. If no model is provided an error is prompted. The cross-spectral matrix is assumed to be frequency by frequency of the type:

```
                 / csd11(f)  csd12(f) \
CSD(f) =         |                    |
                 \ csd21(f)  csd22(f) /
```

Note: The function output colored noise data with one-sided cross spectral density corresponding to the model provided.

Call

```
b = noisegen2D(a, pl)
[b1,b2] = noisegen2D(a1, a2, pl)
[b1,b2,...,bn] = noisegen2D(a1,a2,...,an, pl);
```

- Note1: input AOs must come in couples.
- Note2: this method cannot be used as a modifier, the call `a.noisegen2D(pl)` is forbidden.

Inputs

- a is at least a couple of time series analysis objects
- pl is a parameter list, see the list of accepted parameters below

Outputs

- b are a couple of colored time-series AOs. The coloring filters used are stored in the objects procinfo field under the parameters:
  - b(1): 'Filt11' and 'Filt12'
  - b(2): 'Filt21' and 'Filt22'

Algorithm

1. Fit a set of partial fraction z-domain filters using utils.math.psd2tf
2. Convert to bank of mIIR filters
3. Filter time-series in parallel The filtering process is:

b(1) = Filt11(a(1)) + Filt12(a(2))
b(2) = Filt21(a(1)) + Filt22(a(2))

Parameters

- 'csd11' - a frequency-series AO describing the model csd11
- 'csd12' - a frequency-series AO describing the model csd12
- 'csd21' - a frequency-series AO describing the model csd21
- 'csd22' - a frequency-series AO describing the model csd22
- 'MaxIter' - Maximum number of iterations in fit routine [default: 30]
- 'PoleType' - Choose the pole type for fitting:
  - 1 - use real starting poles.
  - 2 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+1).
  - 3 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+2) [default].
- 'MinOrder' - Minimum order to fit with. [default: 2].
- 'MaxOrder' - Maximum order to fit with. [default: 25]
- 'Weights' - choose weighting for the fit: [default: 2]
  - 1 - equal weights for each point.
  - 2 - weight with 1/abs(model).
  - 3 - weight with 1/abs(model).^2.
  - 4 - weight with inverse of the square mean spread of the model.
- 'Plot' - plot results of each fitting step. [default: false]
- 'Disp' - Display the progress of the fitting iteration. [default: false]
- 'FitTolerance' - 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 2 order of magnitude lower than data itsleves. [default: 2].
- 'RMSEVar' - Root Mean Squared Error Variation - Check if the variation of the RMS error is smaller than 10^(-b), where b is the value given to the variable. This option is useful for finding the minimum of Chi squared. [default: 7].
- 'UseSym' - Use symbolic calculation in eigendecomposition. [default: 0]
  - 0 - perform double-precision calculation in the eigendecomposition procedure to identify 2dim systems and for poles stabilization
  - 1 - uses symbolic math toolbox variable precision arithmetic in the eigendecomposition for 2dim system identification and double-precison for poles stabilization
  - 2 - uses symbolic math toolbox variable precision arithmetic in the eigendecomposition for 2dim system identification and for poles stabilization

noisegen1D Multichannel Noise Generator

Multichannel Noise Generator

---

to come after version 2...

Statespace models

introduce what this is all about

Multichannel Noise Generator Introduction to Statespace Models with LTPDA

Introduction to Statespace Models with LTPDA

---

introduce how statespace models are implemented in ltpda

Statespace models Building Statespace models

Building Statespace models

describe how we can build ssm models; index to more specific methods that follow.

Introduction to Statespace Models with LTPDA Building from scratch

Building from scratch

describe how to build ssm models from description. Introduce the constructor and setter methods for modifying the ssm object. give some examples

Building Statespace models Building from built-in models

Building from built-in models

how to use built-in models. How to write your own built-in models. This should somehow link to the other section of the user manual 'Built-in models of LTPDA'

Building from scratch Modifying systems

Modifying systems

Content needs written...

Building from built-in models Assembling systems

Assembling systems

how to assemble multiple models in to a system

Modifying systems Simulations

Simulations

describe how to make a simulation - time-domain via simulate - bode plots

Assembling systems Transfer Function Modelling

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

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

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)
```

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

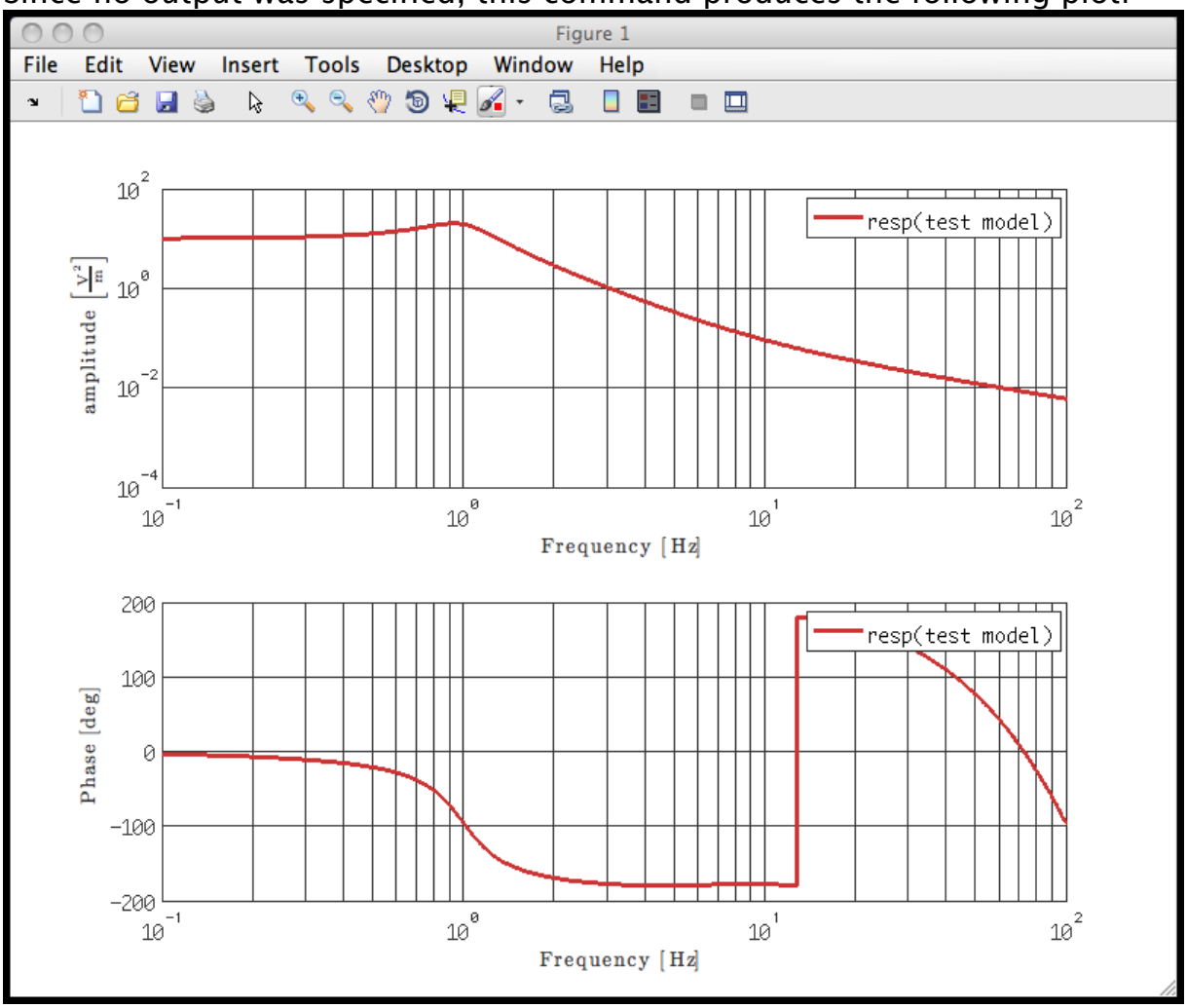
Here we also specified the input units of the transfer function ('iunits') and the output units, ('ounits'). In this case, the model represents a transfer function from metres to Volts squared.

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

Model helper GUI

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

```
>> pzmodel_helper
```

More details on the PZModel Helper GUI.

Building a model Sum of partial fractions representation

Sum of partial fractions representation

Transfer functions can be expressed as a quocient 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

Rational representation

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

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

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_mame');
>> par   = parfrac(pl)
---- rational 1 ----
model:    filter_mame
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

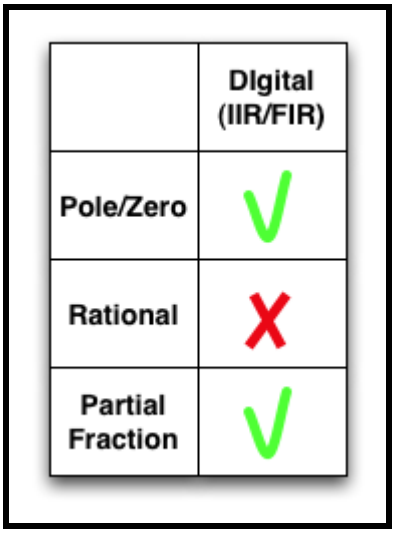
Converting models between different representations

Content needs written...

Rational representation Converting models to digital filters

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:



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:

Converting models to digital filters Downsampling data

Downsampling data

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

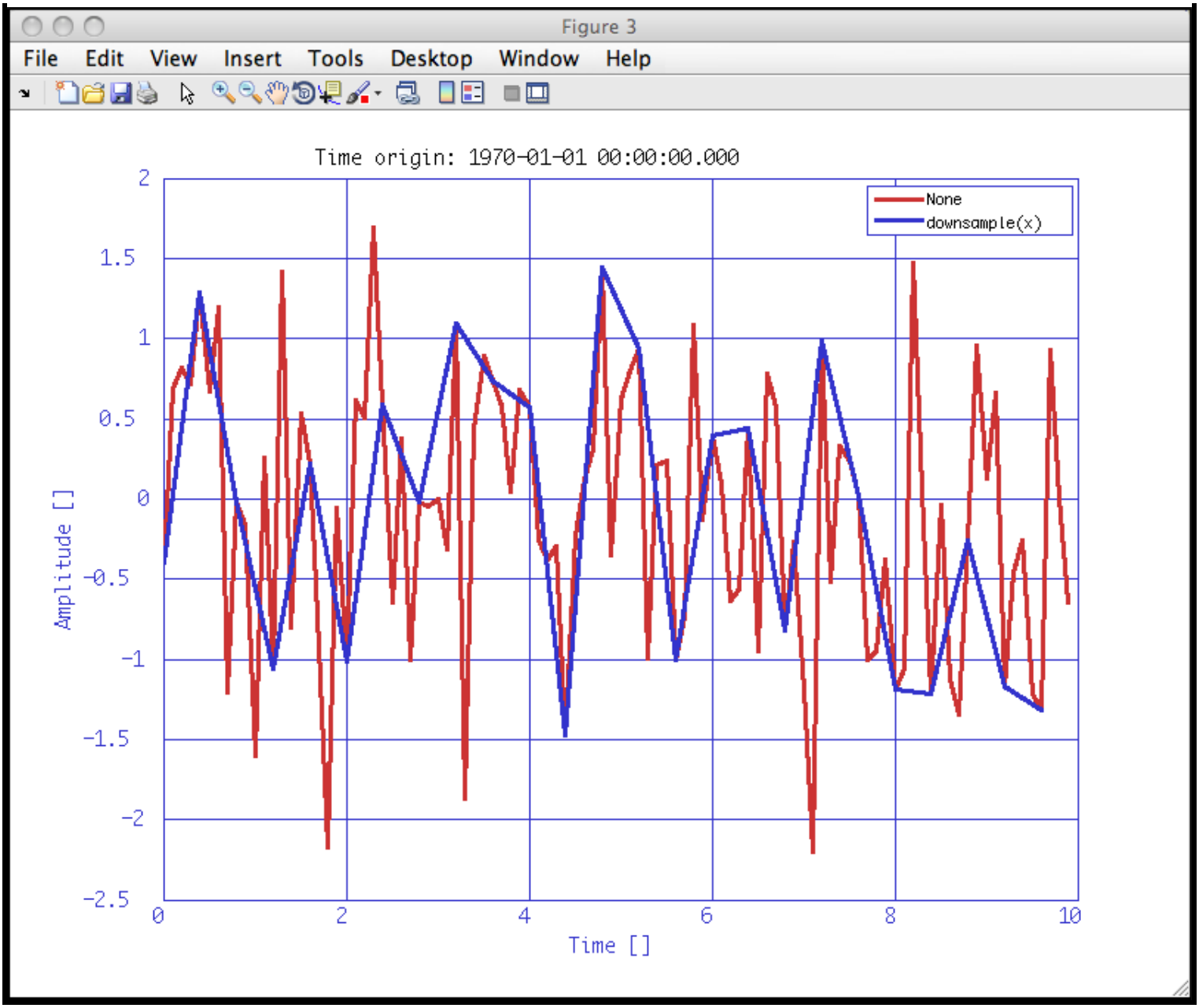
With the following parameters:

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

Examples

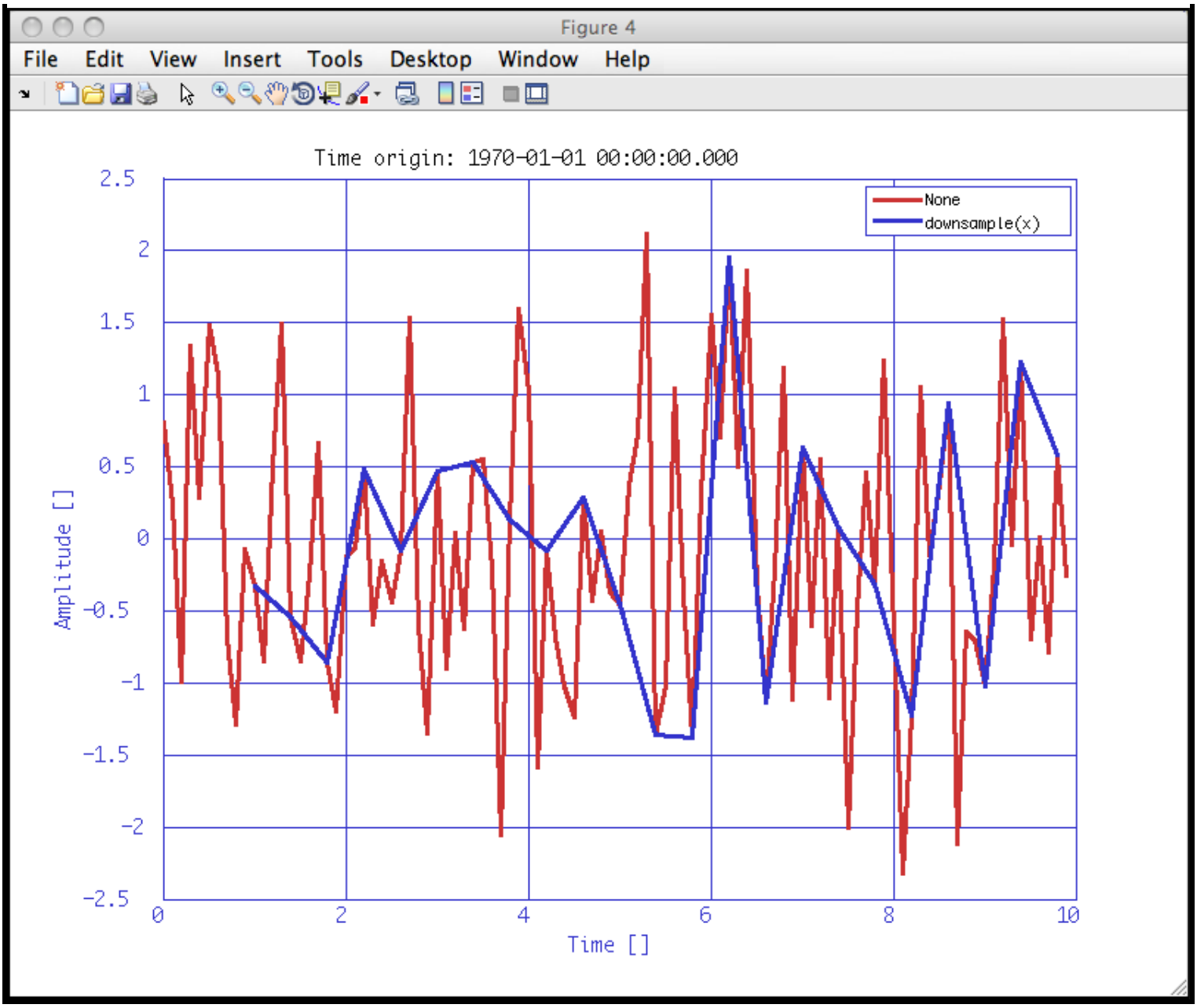
1. Downsampling a sequence of random data at original sampling rate of 10 Hz by a factor of 4 (`fsout` = 2.5 Hz) and no `offset`.

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



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

```
        % 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
        iplot(x, y) % plot original,x, and decimated,y, AOs
```



Signal Pre-processing in LTPDA Upsampling data

Upsampling data

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

Syntax

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

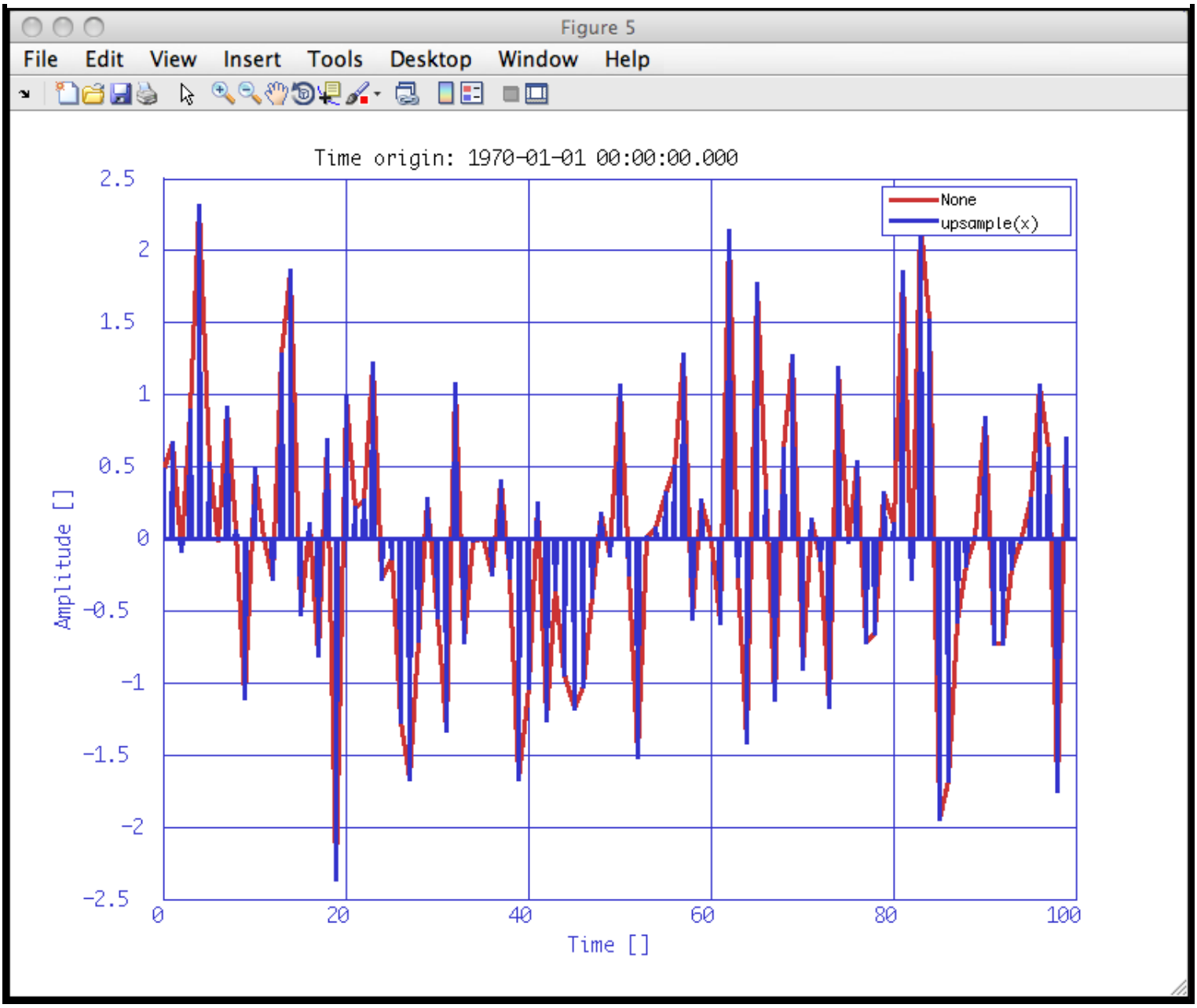
With the following parameters:

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

Examples

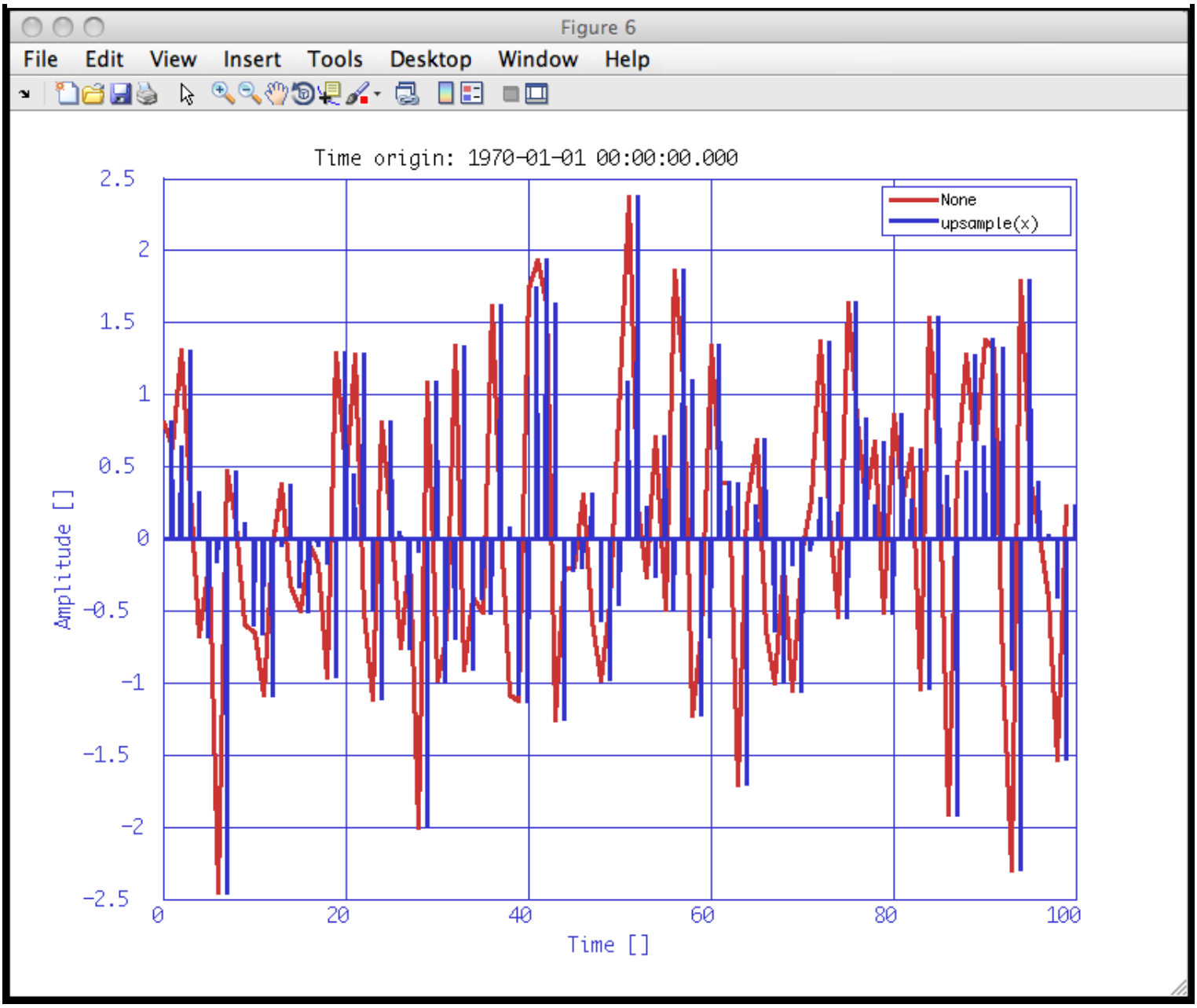
1. Upsampling a sequence of random data at original sampling rate of 1 Hz by a factor of 10 with no initial phase.

```
pl    = plist('tsfcn', 'randn(size(t))','fs',1,'yunits','m');
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) % plot original and upsampled data
```



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

```
  pl         = plist('tsfcn', 'randn(size(t))','fs',1,'yunits','m');
  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 AO (x) to obtain the upsampled and
delayed AO (y)
  iplot(x, y) % plot original and upsampled data
```



Downsampling data Resampling data

Resampling data

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

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

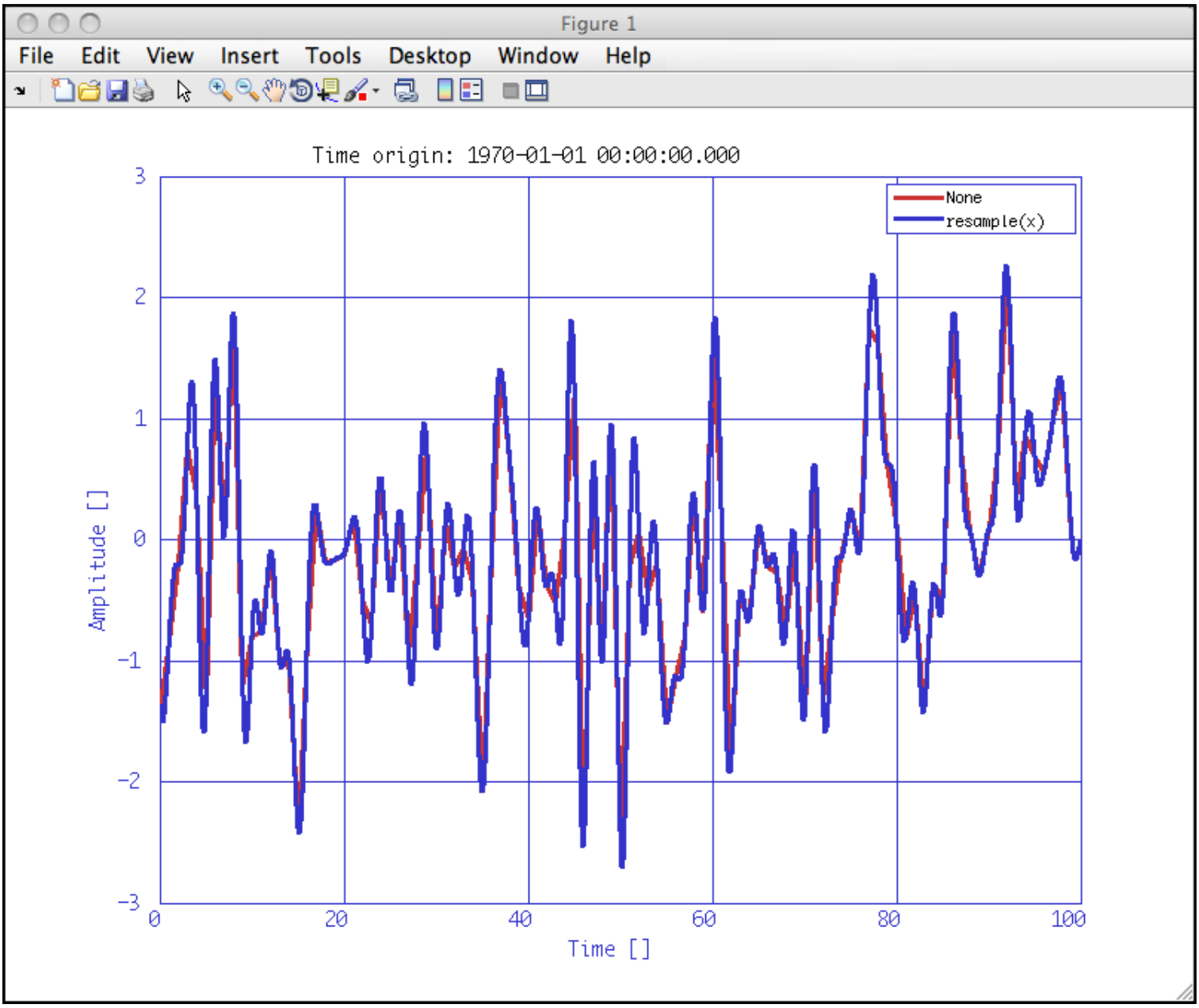
With the following parameters:

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

Examples

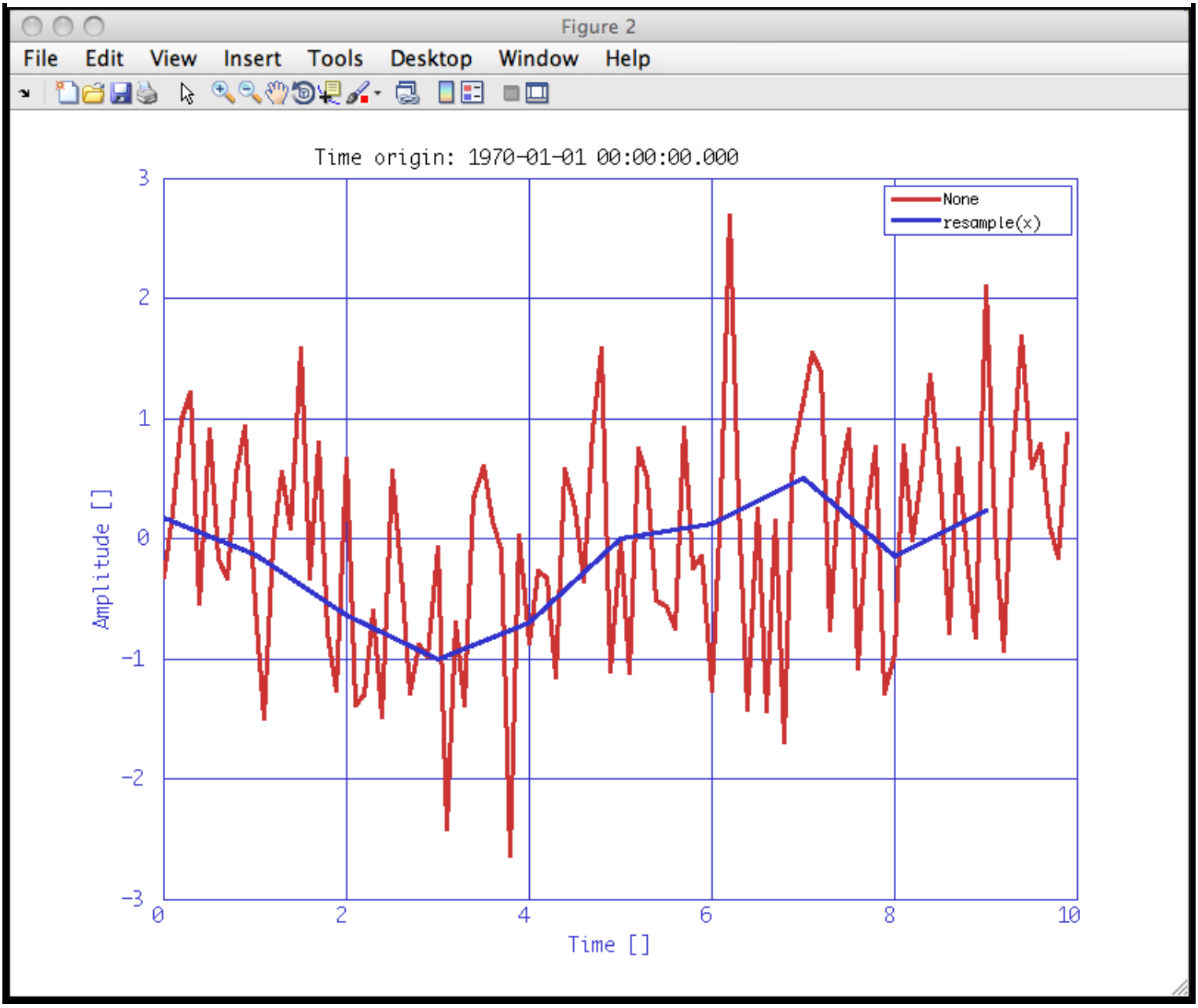
1. Resampling a sequence of random data at original sampling rate of 1 Hz at an output sampling of 50 Hz.

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



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

```
    % 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
    % create parameters list for the filter
    plfilter = plist('type','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)
    iplot(x, y) % plot original and resampled data
```



Upsampling data

Interpolating data

Interpolating data

Interpolation of data can be done in the LTPDA Toolbox by means of `interp`. This function interpolates the values in the input AO(s) at new values specified by the input parameter list.

`Interp` overloads `interp1` function of Matlab Signal Processing Toolbox for AOs.

Syntax

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

With the following parameters:

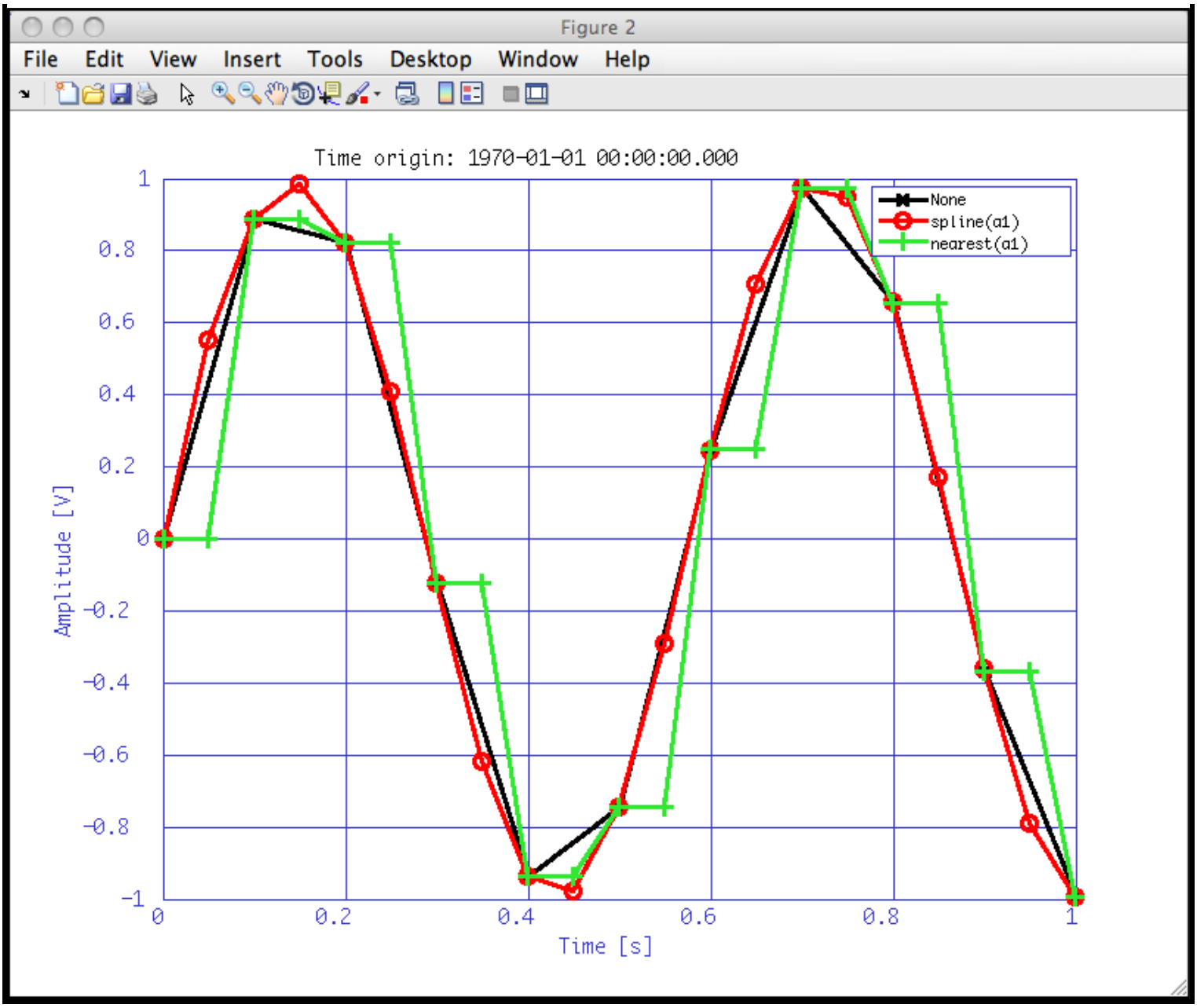
- `vertices` - specify the new vertices to interpolate on
- `method` - four methods are available for interpolating data
  - 'nearest'- nearest neighbor interpolation
  - 'linear' - linear interpolation
  - 'spline' - spline interpolation (default option)
  - 'cubic' - shape-preserving piecewise cubic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

For details see `interp1` help of Matlab.

Examples

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

```
% Signal generation
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);
iplot([x x_spline x_nearest], plist('Markers', {'x', 'o', '+'}, ...
           'LineColors', {'k', 'r'}));
```



Resampling data

Spikes reduction in data

Spikes reduction in data

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

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

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

Syntaxis

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

With the following parameters:

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

Methods explained

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

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

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

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

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

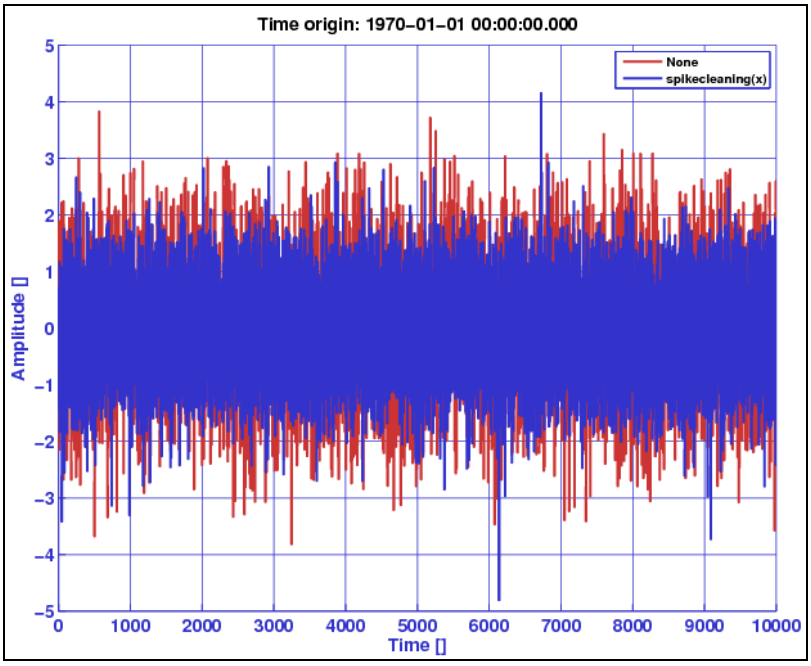
3. `Previous` : the spike is substitued by the previous sample, i.e.:

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

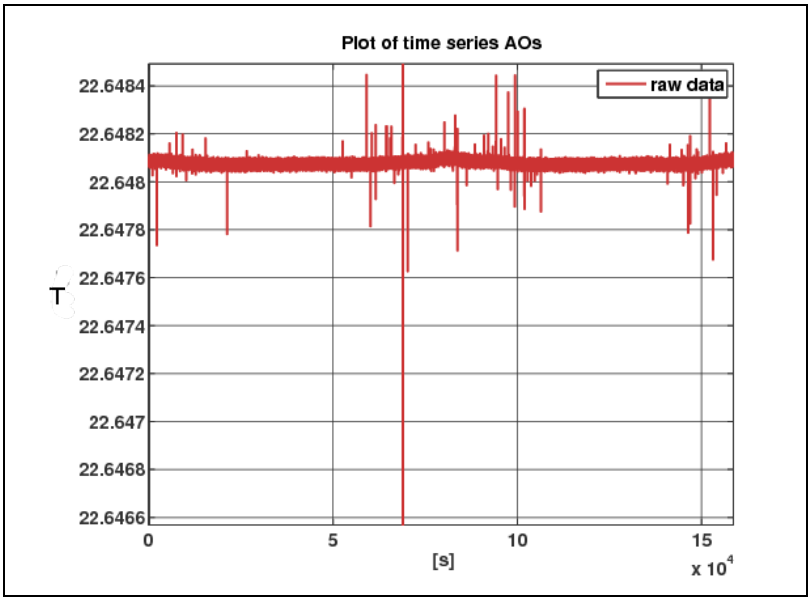
Examples

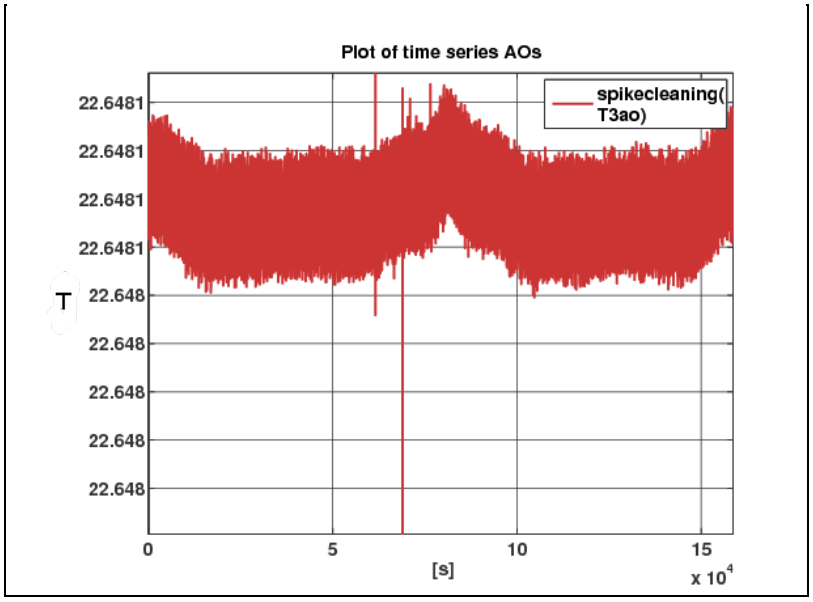
1. Spike cleaning of a sequence of random data with `kspike=2`.

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



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





Interpolating data

Data gap filling

Data gap filling

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

Syntaxis

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

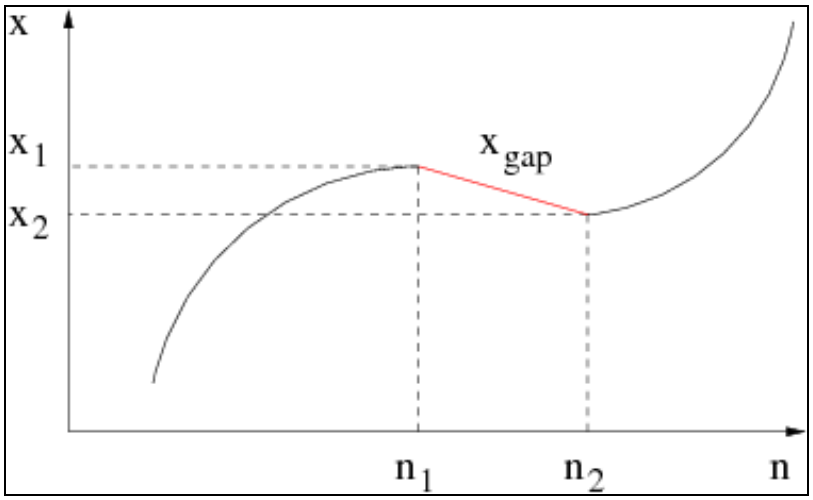
where `a1` and `a2` are the two segments to joint. The parameters are:

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

Interpolation methods

1. `Linear` :

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

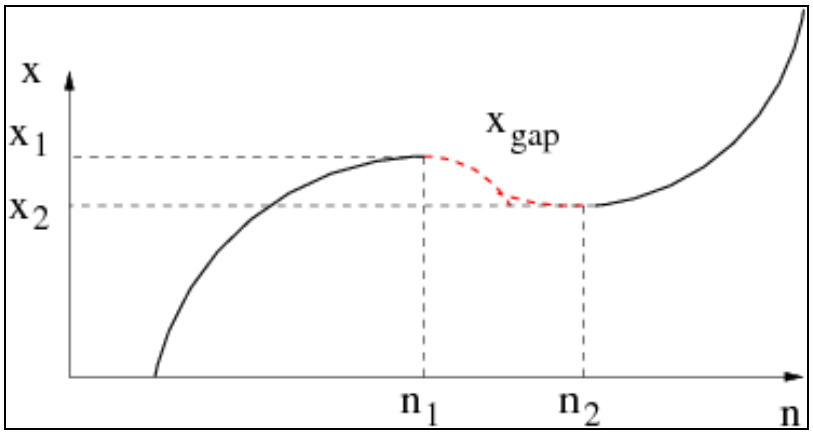


2. `Spline (or third order interpolation)` :

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

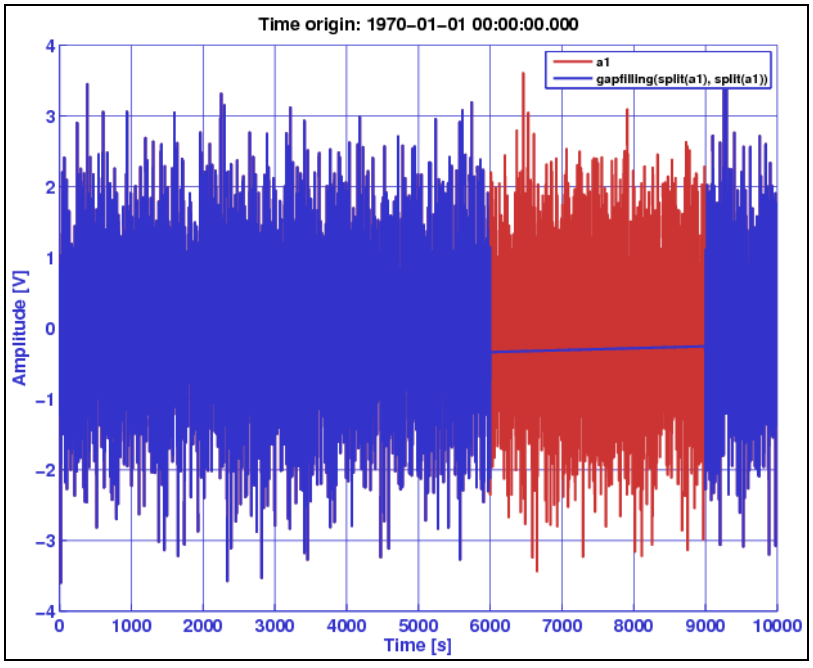
The parameters `a`, `b`, `c` and `d` are calculated by solving next system of equations:

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

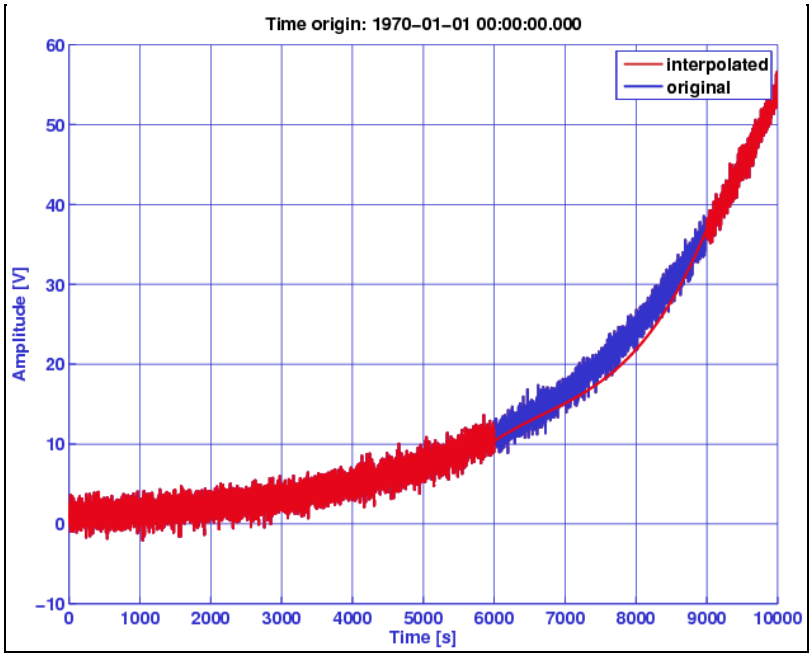


Examples

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



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



Spikes reduction in data

Noise whitening

Noise whitening

Noise whitening tools in LTPDA are:

- Whiten1D
- Whiten2D

whiten1D

Description

whiten1D whitens input time-series. Whitening filter is constructed by a fitting procedure to the model provided. If no model is provided, a fit is made to a log-spectral-density estimate of the time-series (made using lpsd). Note: The function assumes that the input model corresponds to the one-sided psd of the data to be whitened.

Call

```
b = whiten1D(a, pl)
[b1,b2,...,bn] = whiten1D(a1,a2,...,an, pl);
```

Inputs

- a - is a tsdata analysis object or a vector of tsdata analysis objects
- pl - is a plist with the input parameters. See the list of function parameters below

Outputs

- b "whitened" time-series AOs. The whitening filters used are stored in the objects procinfo field under the parameter 'Filt'.

Algorithm

1. If no model provided, make lpsd of time-series and take it as a model for the data power spectral density
2. Fit a set of partial fraction z-domain filters using utils.math.psd2wf. The fit is automatically stopped when the accuracy tolerance is reached.
3. Convert to bank of MIIR filters.
4. Filter time-series in parallel

Accuracy tolerance criteria

- No model provided
  In such a case the algorithm try to extract a smooth model from lpsd noisy data. Fit residuals spectral flatness is compared with the 'FitTolerance' parameter. Fit is stopped when residuals spectral flatness is larger than the 'FitTolerance' parameter. Admitted values are 0 < tol < 1. Recommended values are 0.5 < tol < 0.7. If out of range values are provided the parameter is set to 0.5.
- Model provided
  In such a case the algorithm try to exactly fit the input model whitin the accuracy reported in 'FitTolerance'. Check if the minimum of the logarithmic difference between data and residuals is

larger than a specified value. Admitted values are tol>0. Recommended values are 0.5 < tol < 2. if the conditioning value is 2, the function ensures that the difference between data and residuals is at lest 2 order of magnitude lower than data itsleves. If a negative value is provided the tolerance is set to 1.

Parameters

- 'Model' - a frequency-series AO describing the model response to build the filter from. [default: lpsd of time-series].
- 'MaxIter' - Maximum number of iterations in fit routine [default: 30]
- 'PoleType' - Choose the pole type for fitting:
  - 1 - use real starting poles.
  - 2 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+1).
  - 3 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+2) [default].
- 'MinOrder' - Minimum order to fit with. [default: 2].
- 'MaxOrder' - Maximum order to fit with. [default: 25]
- 'Weights' - choose weighting for the fit: [default: 2]
  - 1 - equal weights for each point.
  - 2 - weight with 1/abs(model).
  - 3 - weight with 1/abs(model).^2.
  - 4 - weight with inverse of the square mean spread of the model.
- 'Plot' - plot results of each fitting step. [default: false]
- 'Disp' - Display the progress of the fitting iteration. [default: false]
- 'FitTolerance' - Stopping fit tolerance condition. Be sure to read the algorithm description to provide the correct value. [default: 0.6]
- 'RMSEVar' - Root Mean Squared Error Variation - Check if the variation of the RMS error is smaller than 10^(-b), where b is the value given to the variable. This option is useful for finding the minimum of Chi squared. [default: 7].

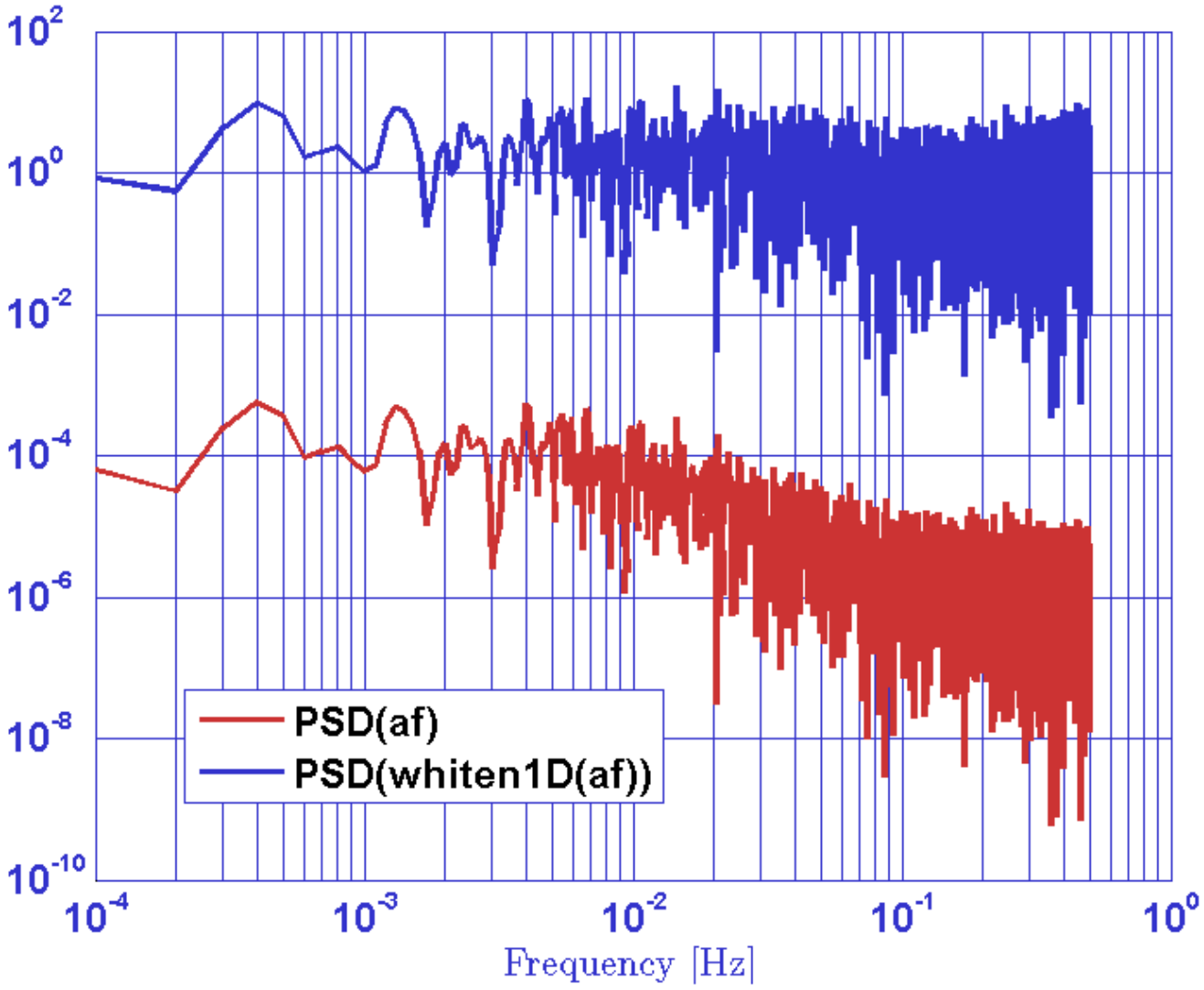
parameters passed to lpsd()

- 'Jdes' - The number of points in the power spectrum. [default: help lpsd].
- 'Win' - Spectral window used in spectral estimation. [default: help lpsd].
- 'Order' - order of segment detrending: [default: help lpsd]
  - -1 - no detrending
  - 0 - subtract mean
  - 1 - subtract linear fit
  - N - subtract fit of polynomial, order N

Example

```
% Generate white noise
fs = 1;
a = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', 10000, 'yunits','m'));

% filter
pzm = pzmodel(1e-2, {0.01}, {0.1});
ft = miir(pzm,plist('fs',fs));

% coloring white noise
af = filter(a, ft);

% Whitening colored noise
pl = plist(...
    'model', [], ...
    'MaxIter', 30, ...
    'PoleType', 2, ...
    'MinOrder', 2, ...
    'MaxOrder', 9, ...
    'Weights', 2, ...
    'Plot', false,...
    'Disp', false,...
    'RMSEVar', 3,...
    'FitTolerance', 0.6); % tolerancee on fit residuals spectral flatness
```

```
aw = whiten1D(af,pl);

% Calculate psd of colored and whitened data
afxx = af.psd;
awxx = aw.psd;

% plotting
iplot(afxx,awxx)
```



Noise whitening whiten2D

whiten2D

Description

whiten2D whitens cross-correlated time-series. Whitening filters are constructed by a fitting procedure to the corss-spectrum models provided. Note: The function assumes that the input model corresponds to the one-sided csd of the data to be whitened.

Call

```
b = whiten2D(a, pl)
[b1,b2] = whiten2D(a1, a2, pl)
[b1,b2,...,bn] = whiten2D(a1,a2,...,an, pl);
```

- Note1: input AOs must come in couples.
- Note2: this method cannot be used as a modifier, the call `a.whiten2D(pl)` is forbidden.

Inputs

- a is at least a couple of time series analysis objects
- pl is a parameter list, see the list of accepted parameters below

Outputs

- b is at least a couple of "whitened" time-series AOs. The whitening filters used are stored in the objects procinfo field as.
  - b(1): 'Filt11' and 'Filt12'
  - b(2): 'Filt21' and 'Filt22'

Algorithm

1. Fit a set of partial fraction z-domain filters using utils.math.psd2wf
2. Convert to bank of mIIR filters
3. Filter time-series in parallel The filtering process is:
   b(1) = Filt11(a(1)) + Filt12(a(2))
   b(2) = Filt21(a(1)) + Filt22(a(2))

Parameters

- 'csd11' - a frequency-series AO describing the model csd11
- 'csd12' - a frequency-series AO describing the model csd12
- 'csd21' - a frequency-series AO describing the model csd21

'csd22' - a frequency-series AO describing the model csd22

- 'MaxIter' - Maximum number of iterations in fit routine [default: 30]
- 'PoleType' - Choose the pole type for fitting:
  - 1 - use real starting poles.
  - 2 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+1).
  - 3 - generates complex conjugate poles of the type a.*exp(theta*pi*j) with theta = linspace(0,pi,N/2+2) [default].
- 'MinOrder' - Minimum order to fit with. [default: 2].
- 'MaxOrder' - Maximum order to fit with. [default: 25]
- 'Weights' - choose weighting for the fit: [default: 2]
  - 1 - equal weights for each point.
  - 2 - weight with 1/abs(model).
  - 3 - weight with 1/abs(model).^2.
  - 4 - weight with inverse of the square mean spread of the model.
- 'Plot' - plot results of each fitting step. [default: false]
- 'Disp' - Display the progress of the fitting iteration. [default: false]
- 'FitTolerance' - 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 2 order of magnitude lower than data itsleves. [default: 2].
- 'RMSEVar' - Root Mean Squared Error Variation - Check if the variation of the RMS error is smaller than 10^(-b), where b is the value given to the variable. This option is useful for finding the minimum of Chi squared. [default: 7].
- 'UseSym' - Use symbolic calculation in eigendecomposition. [default: 0]
  - 0 - perform double-precision calculation in the eigendecomposition procedure to identify 2dim systems and for poles stabilization
  - 1 - uses symbolic math toolbox variable precision arithmetic in the eigendecomposition for 2dim system identification and double-precison for poles stabilization
  - 2 - uses symbolic math toolbox variable precision arithmetic in the eigendecomposition for 2dim system identification and for poles stabilization

whiten1D Signal Processing in LTPDA

Signal Processing in LTPDA

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

- Digital filtering
- Discrete Derivative
- Spectral estimation
- Fitting algorithms

whiten2D Digital Filtering

Digital Filtering

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

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

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

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

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

- Stable: if finite input results in finite output.

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

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

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

Digital filters classification

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

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

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

  AR processes can be also classified as IIR Filters.

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

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

  MA processes can be also classified as FIR Filters.

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

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

Signal Processing in LTPDA IIR Filters

IIR Filters

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

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

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

Creating a IIR filter in the LTPDA

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

Creating from a plist

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

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

Creating from a pzmodel

IIR filters can also be created from a pzmodel .

Creating from a difference equation

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

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

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

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

Importing an existing model

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

-
- LISO files:

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

- XML files:

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

- MAT files:

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

- From repository:

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

Digital Filtering FIR Filters

FIR Filters

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

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

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

Creating a FIR filter in the LTPDA

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

Creating from a plist

The following example creates an order 64 highpass filter with high frequency gain 2. 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:

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

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

Creating from an Analysis Object

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

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

```
>> f = mfir(ao(fsd));
```

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

Importing an existing model

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

-
- LISO files:

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

- XML files:

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

- MAT files:

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

- From repository:

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

IIR Filters Discrete Derivative

Discrete Derivative

Derivative calculation for dicrete data series.

Description

Derivative estimation on discrete data series is implemented by the function `ao/diff`. This function implements 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].

Call

```
b = diff(a,pl)
[b1,b2,...,bn] = diff(a1,a2,...,an, pl);
```

Inputs

- a - is an analysis object or a vector of analysis objects
- pl - is a plist containing input parameters. See the list of function parameters below.

Outputs

- b - output analysis objects containing the differentiated data.

Parameters

Key	Parameter Values	
	Value	**Description**
'METHOD'	'2POINT'	Two point derivative.
	'3POINT'	Three point derivative.
	'5POINT'	5 point derivative.
	'ORDER2'	Compute derivative using a 2nd order method.
	'ORDER2SMOOTH'	Compute derivative using a 2nd order method with a parabolic fit to 5 consecutive samples.
	'FPS'	Calculate five points derivative using the generalized five point method

described in [1]. If you call with this oprtion you may add also the parameters:

- 'ORDER'
  Supperted values are:
  - 'ZERO' - Data smoother
  - 'FIRST' - First Derivative
  - 'SECOND' - Second Derivative
- 'COEFF' - coefficient used for the derivative

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: $$\frac{dy[k]}{dx} \approx \frac{-y[k+2]+8y[k+1]-8y[k-1]+y[k-2]}{3\left(x[k+2]-x[k-2]\right)}$$
'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 1/7
- 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

Frequency response of first and second order estimators is reported in figures 1 and 2 respectively.

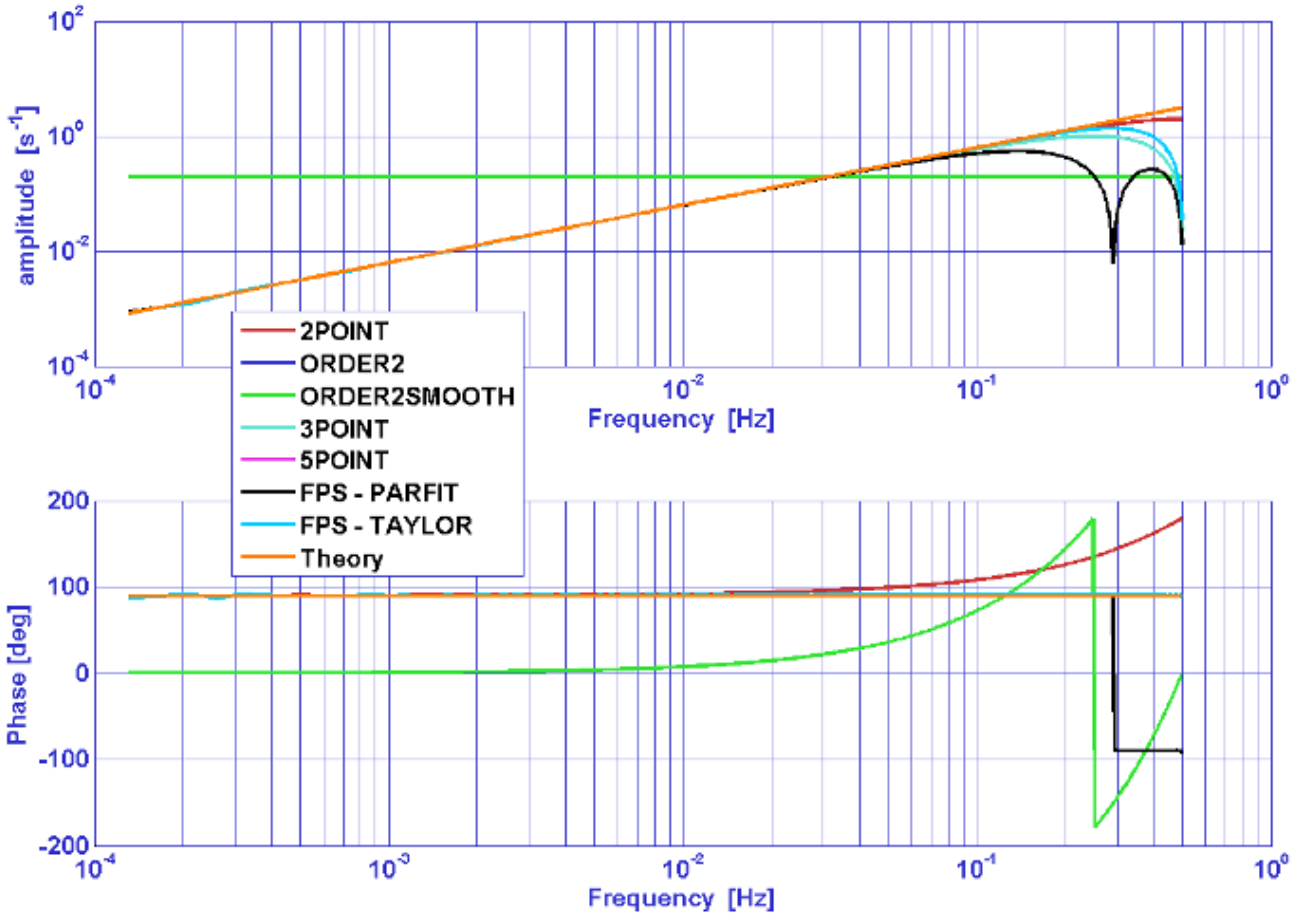


Figure 1: Frequency response of first derivative estimators.

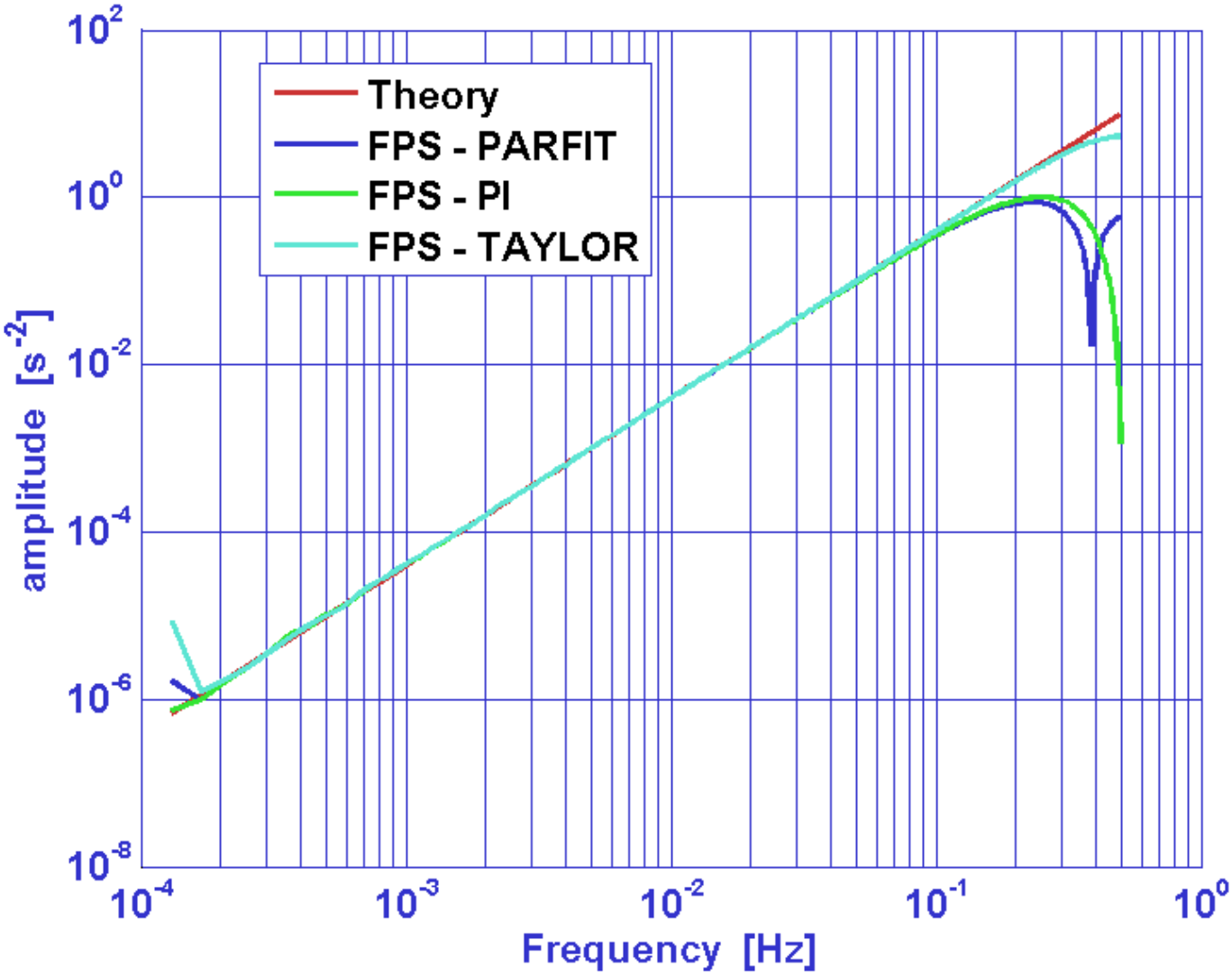


Figure 2: Frequency response of second derivative estimators.

References

1. L. Ferraioli, M. Hueller and S. Vitale, Discrete derivative estimation in LISA Pathfinder data reduction, Class. Quantum Grav., 7th LISA Symposium special issue.
   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.*

FIR Filters *Spectral Estimation*

Spectral Estimation

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

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

- `ao/psd` power spectral density estimates
- `ao/cpsd` cross-spectral density estimates
- `ao/cohere` magnitude squared coherence estimates
- `ao/tfe` transfer function estimates

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

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

- `ao/lpsd` log-scale power spectral density estimates
- `ao/lcpsd` log-scale cross-spectral density estimates
- `ao/lcohere` log-scale magnitude squared coherence estimates
- `ao/ltfe` log-scale transfer function estimates

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

Introduction

This introduction is directly adapted from Matlab documentation regarding the Signal Processing Toolbox. More detailed description, including the whole document referred before, can be found typing:

```
>> doc signal
```

in the Matlab terminal.

On this page...

Background Information

The goal of *spectral estimation* is to describe the distribution (over frequency) of the power contained in a signal, based on a finite set of data. Estimation of power spectra is useful in a variety of applications, including the detection of signals buried in wide-band noise.

Power Spectral Density Function

The *power spectral density* (PSD) of a stationary random process x_n is mathematically related to the correlation sequence by the discrete-time Fourier transform. In terms of normalized frequency, this is given by

$$P_{xx}(\omega) = \frac{1}{2\pi} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-j\omega m}$$

This can be written as a function of physical frequency f (e.g., in hertz) by using the relation $\omega = 2\pi f / f_s$, where f_s is the sampling frequency.

$$P_{xx}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{\infty} R_{xx}(m) e^{-2\pi j f m / f_s}$$

The correlation sequence can be derived from the PSD by use of the inverse discrete-time Fourier transform:

$$R_{xx}(m) = \int_{-\pi}^{\pi} (P_{xx}(\omega) e^{j\omega m}) d\omega = \int_{-f_s/2}^{f_s/2} (P_{xx}(f) e^{2\pi j f m / f_s}) df$$

The average power of the sequence x_n over the entire Nyquist interval is represented by

$$R_{xx}(0) = \int_{-\pi}^{\pi} P_{xx}(\omega) d\omega = \int_{-f_s/2}^{f_s/2} P_{xx}(f) df$$

The average power of a signal over a particular frequency band $[\omega_1,\omega_2]$, $0 \le \omega_1 < \omega_2 \le \pi$, can be found by integrating the PSD over that band:

$$\overline{P}_{[\omega_1,\omega_2]} = \int_{\omega_1}^{\omega_2} P_{xx}(\omega)d\omega + \int_{-\omega_2}^{-\omega_1} P_{xx}(\omega)d\omega$$

You can see from the above expression that $P_{xx}(\omega)$ represents the power content of a signal in an *infinitesimal* frequency band, which is why it is called the power spectral *density*.

The units of the PSD are power (e.g., watts) per unit of frequency. In the case of $P_{xx}(\omega)$, this is watts/radian/sample or simply watts/radian. In the case of $P_{xx}(f)$, the units are watts/hertz. Integration of the PSD with respect to frequency yields units of watts, as expected for the average power $\overline{P}_{[\omega_1,\omega_2]}$.

For real signals, the PSD is symmetric about DC, and thus $P_{xx}(\omega)$ for $0 \le \omega < \pi$ is sufficient to completely characterize the PSD. However, to obtain the average power over the entire Nyquist interval, it is necessary to introduce the concept of the *one-sided* PSD.

The one-sided PSD is given by

$$P_{onesided}(\omega) = \begin{cases} 0, & -\pi \le \omega < 0 \\ 2P_{xx}(\omega), & 0 \le \omega < \pi \end{cases}$$

The average power of a signal over the frequency band $[\omega_1,\omega_2]$, $0 \le \omega_1 < \omega_2 \le \pi$, can be computed using the one-sided PSD as

$$\overline{P}_{[\omega_1,\omega_2]} = \int_{\omega_1}^{\omega_2} P_{onesided}(\omega)d\omega$$

Cross-Spectral Density Function

The PSD is a special case of the *cross spectral density* (*CPSD*) function, defined between two signals x_n and y_n as

$$P_{xy}(\omega) = \frac{1}{2\pi}\sum_{m=-\infty}^{\infty} R_{xy}(m)e^{-j\omega m}$$

As is the case for the correlation and covariance sequences, the toolbox *estimates* the PSD and CPSD because signal lengths are finite.

To estimate the cross-spectral density of two equal length signals `x` and `y` using Welch's method, the `cpsd` function forms the periodogram as the product of the FFT of `x` and the conjugate of the FFT of `y`. Unlike the real-valued PSD, the CPSD is a complex function. `cpsd` handles the sectioning and windowing of `x` and `y` in the same way as the `pwelch` function:

```
Sxy = cpsd(x,y,nwin,noverlap,nfft,fs)
```

Transfer Function Estimate

One application of Welch's method is nonparametric system identification. Assume that *H* is a linear, time invariant system, and *x*(*n*) and *y*(*n*) are the input to and output of *H*, respectively. Then the power spectrum of *x*(*n*) is related to the CPSD of *x*(*n*) and *y*(*n*) by

$$P_{yx}(\omega) = H(\omega)P_{xx}(\omega)$$

An estimate of the transfer function between $x(n)$ and $y(n)$ is

$$\hat{H}(\omega) = \frac{\hat{P}_{yx}(\omega)}{\hat{P}_{xx}(\omega)}$$

This method estimates both magnitude and phase information. The `tfe` function uses Welch's method to compute the CPSD and power spectrum, and then forms their quotient for the transfer function estimate. Use `tfe` the same way that you use the `cpsd` function.

Coherence Function

The magnitude-squared coherence between two signals $x(n)$ and $y(n)$ is

$$C_{xy}(\omega) = \frac{\left|P_{xy}(\omega)\right|^2}{P_{xx}(\omega)P_{yy}(\omega)}$$

This quotient is a real number between 0 and 1 that measures the correlation between $x(n)$ and $y(n)$ at the frequency ω.

The `cohere` function takes sequences `x` and `y`, computes their power spectra and CPSD, and returns the quotient of the magnitude squared of the CPSD and the product of the power spectra. Its options and operation are similar to the `cpsd` and `tfestimate` functions.

If the input sequence length `nfft`, window length `window`, and the number of overlapping data points in a window `numoverlap`, are such that `cohere` operates on only a single record, the function returns all ones. This is because the coherence function for linearly dependent data is one.

Spectral Estimation Methods

The various methods of spectrum estimation available in the toolbox are categorized as follows:

- Nonparametric methods

Nonparametric methods are those in which the PSD is estimated directly from the signal itself. The simplest such method is the *periodogram*. An improved version of the periodogram is *Welch's method* [8].

Nonparametric Methods

The following sections discuss the periodogram, modified periodogram, and Welch methods of nonparametric estimation.

Periodogram

In general terms, one way of estimating the PSD of a process is to simply find the discrete-time Fourier transform of the samples of the process (usually done on a grid with an FFT) and take the magnitude squared of the result. This estimate is called the *periodogram*.

The periodogram estimate of the PSD of a length-L signal $x_L[n]$ is

$$\hat{P}_{xx}(f) = \frac{\left|X_L(f)\right|^2}{f_s L}$$

where

$$X_L(f) = \sum_{n=0}^{L-1} x_L[n] e^{-2\pi j f n / f_s}$$

The actual computation of $X_L(f)$ can be performed only at a finite number of frequency points, N, and usually employs the FFT. In practice, most implementations of the periodogram method compute the N-point PSD estimate

$$\hat{P}_{xx}[f_k] = \frac{|X_L[f_k]|^2}{f_s L}, \qquad f_k = \frac{k f_s}{N}, \qquad k = 0, 1, \ldots, N-1$$

where

$$X_L[f_k] = \sum_{n=0}^{N-1} x_L[n] e^{-2\pi j k n / N}$$

It is wise to choose $N > L$ so that N is the next power of two larger than L. To evaluate $X_L[f_k]$, we simply pad $x_L[n]$ with zeros to length N. If $L > N$, we must wrap $x_L[n]$ modulo-N prior to computing $X_L[f_k]$.

Performance of the Periodogram

The following sections discuss the performance of the periodogram with regard to the issues of leakage, resolution, bias, and variance.

Spectral Leakage. Consider the PSD of a finite-length signal $x_L[n]$, as discussed in the Periodogram section. It is frequently useful to interpret $x_L[n]$ as the result of multiplying an infinite signal, $x[n]$, by a finite-length rectangular window, $w_R[n]$:

$$x_L[n] = x[n] \cdot w_R[n]$$

Because multiplication in the time domain corresponds to convolution in the frequency domain, the Fourier transform of the expression above is

$$X_L(f) = \frac{1}{f_s} \int_{-f_s/2}^{f_s/2} X(\rho) W_R(f-\rho) d\rho$$

The expression developed earlier for the periodogram,

$$\hat{P}_{xx}(f) = \frac{|X_L(f)|^2}{f_s L}$$

illustrates that the periodogram is also influenced by this convolution.

The effect of the convolution is best understood for sinusoidal data. Suppose that $x[n]$ is composed of a sum of M complex sinusoids:

$$x[n] = \sum_{k=1}^{M} A_k e^{j\omega_k n}$$

Its spectrum is

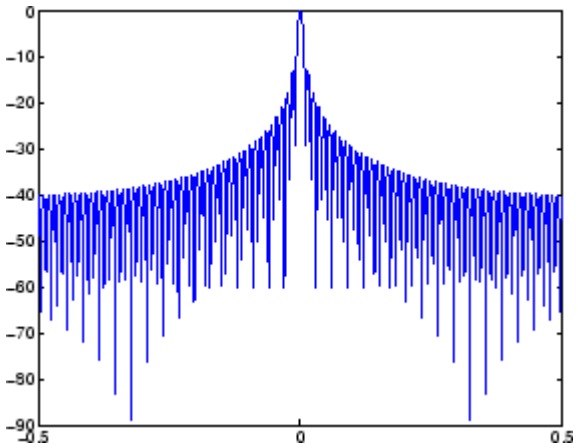
$$X(f) = f_s \sum_{k=1}^{M} A_k \delta(f - f_k$$

which for a finite-length sequence becomes

$$X_L(f) = \int_{-f_s/2}^{f_s/2} \left(\sum_{k=1}^{M} A_k \delta(\rho - f_k) \right) W_R(f - \rho) d\rho = \sum_{k=1}^{M} A_k W_R(f - f_k)$$

So in the spectrum of the finite-length signal, the Dirac deltas have been replaced by terms of the form $W_R(f - f_k)$, which corresponds to the frequency response of a rectangular window centered on the frequency f_k.

The frequency response of a rectangular window has the shape of a sinc signal, as shown below.



The plot displays a main lobe and several side lobes, the largest of which is approximately 13.5 dB below the mainlobe peak. These lobes account for the effect known as *spectral leakage*. While the infinite-length signal has its power concentrated exactly at the discrete frequency points f_k, the windowed (or truncated) signal has a continuum of power "leaked" around the discrete frequency points f_k.

Because the frequency response of a short rectangular window is a much poorer approximation to the Dirac delta function than that of a longer window, spectral leakage is especially evident when data records are short.

It is important to note that the effect of spectral leakage is contingent solely on the length of the data record. It is *not* a consequence of the fact that the periodogram is computed at a finite number of frequency samples.

Resolution. *Resolution* refers to the ability to discriminate spectral features, and is a key concept on the analysis of spectral estimator performance.

In order to resolve two sinusoids that are relatively close together in frequency, it is necessary for the difference between the two frequencies to be greater than the width of the mainlobe of the leaked spectra for either one of these sinusoids. The mainlobe width is defined to be the width of the mainlobe at the point where the power is half the peak mainlobe power (i.e., 3 dB width). This width is approximately equal to f_s / L.

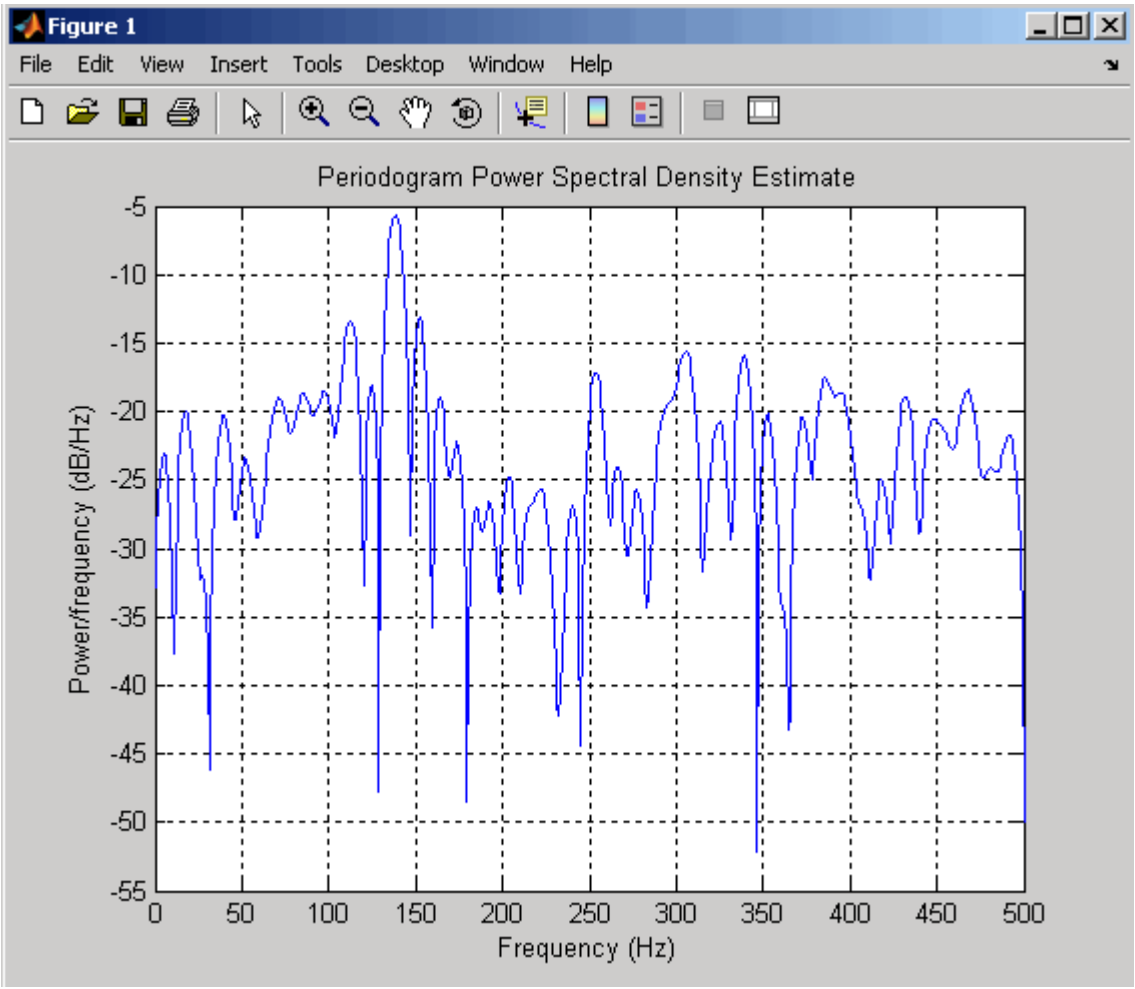
In other words, for two sinusoids of frequencies f_1 and f_2, the resolvability condition requires that

$$\Delta f = (f_1 - f_2) > \frac{f_s}{L}$$

In the example above, where two sinusoids are separated by only 10 Hz, the data record must be greater than 100 samples to allow resolution of two distinct sinusoids by a periodogram.

The above discussion about resolution did not consider the effects of noise since the signal-to-noise ratio (SNR) has been relatively high thus far. When the SNR is low, true spectral features are much harder to distinguish, and noise artifacts appear in spectral estimates based on the periodogram. The example below illustrates this:

```
randn('state',0)
fs = 1000;                      % Sampling frequency
t = (0:fs/10)./fs;              % One-tenth second worth of samples
A = [1 2];                      % Sinusoid amplitudes
f = [150;140];                  % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 2*randn(size(t));
Hs=spectrum.periodogram;
psd(Hs,xn,'Fs',fs,'NFFT',1024)
```



Bias of the Periodogram. The periodogram is a biased estimator of the PSD. Its expected value can be shown to be

$$E\left\{\frac{|X_L(f)|^2}{f_s L}\right\} = \frac{1}{f_s L}\int_{-f_s/2}^{f_s/2} P_{xx}(\rho)|W_R(f-\rho)|^2 d\rho$$

which is similar to the first expression for $X_L(f)$ in Spectral Leakage, except that the expression here is in terms of average power rather than magnitude. This suggests that the estimates produced by the periodogram correspond to a *leaky* PSD rather than the true PSD.

Note that $|W_R(f-\rho)|^2$ essentially yields a triangular Bartlett window (which is apparent from the

fact that the convolution of two rectangular pulses is a triangular pulse). This results in a height for the largest sidelobes of the leaky power spectra that is about 27 dB below the mainlobe peak; i.e., about twice the frequency separation relative to the non-squared rectangular window.

The periodogram is asymptotically unbiased, which is evident from the earlier observation that as the data record length tends to infinity, the frequency response of the rectangular window more closely approximates the Dirac delta function (also true for a Bartlett window). However, in some cases the periodogram is a poor estimator of the PSD even when the data record is long. This is due to the variance of the periodogram, as explained below.

Variance of the Periodogram. The variance of the periodogram can be shown to be approximately

$$var\left\{\frac{|X_L(f)|^2}{f_s L}\right\} = P_{xx}^2(f)\left[1+\left(\frac{\sin(2\pi L f/f_s)}{L\sin(2\pi f/f_s)}\right)^2\right]$$

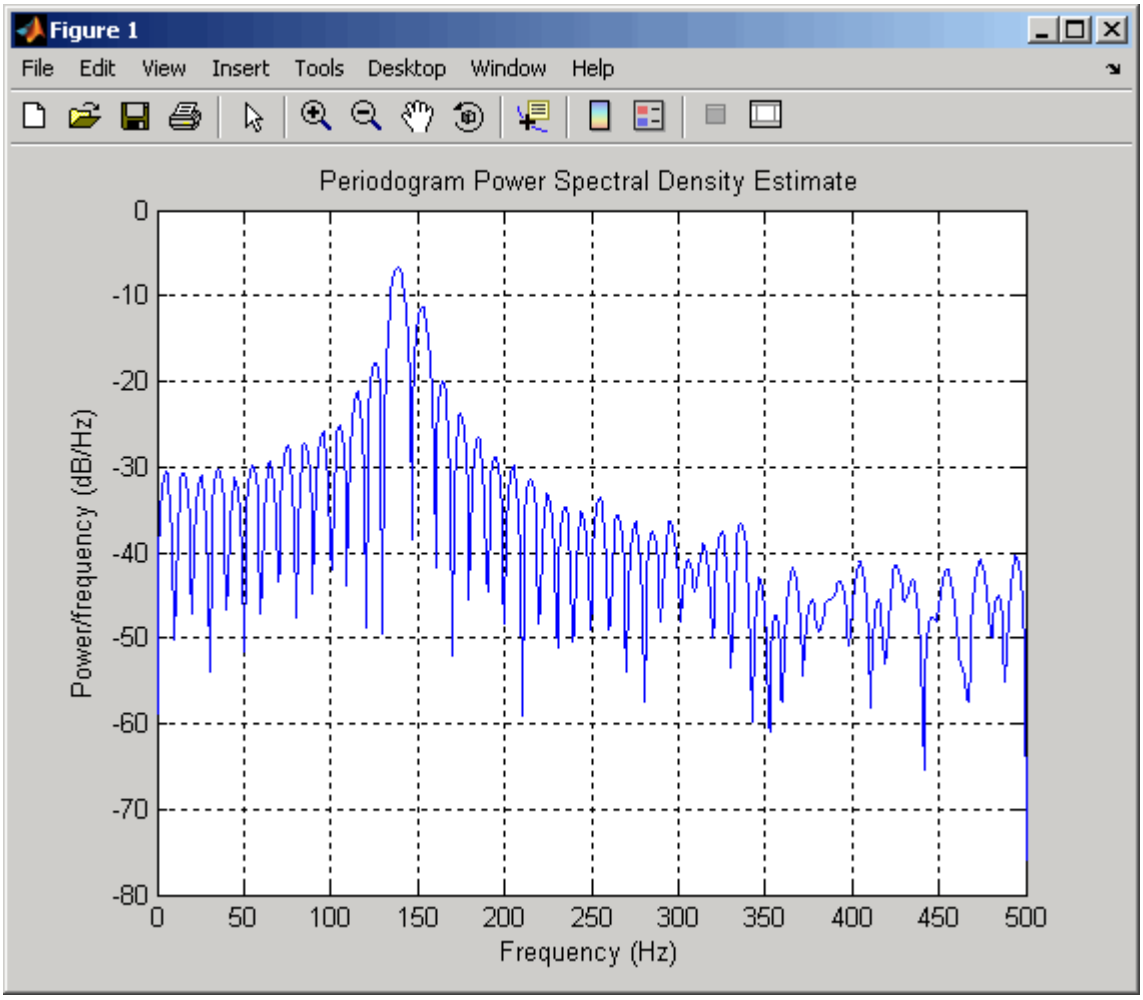
which indicates that the variance does not tend to zero as the data length *L* tends to infinity. In statistical terms, the periodogram is not a consistent estimator of the PSD. Nevertheless, the periodogram can be a useful tool for spectral estimation in situations where the SNR is high, and especially if the data record is long.

The Modified Periodogram

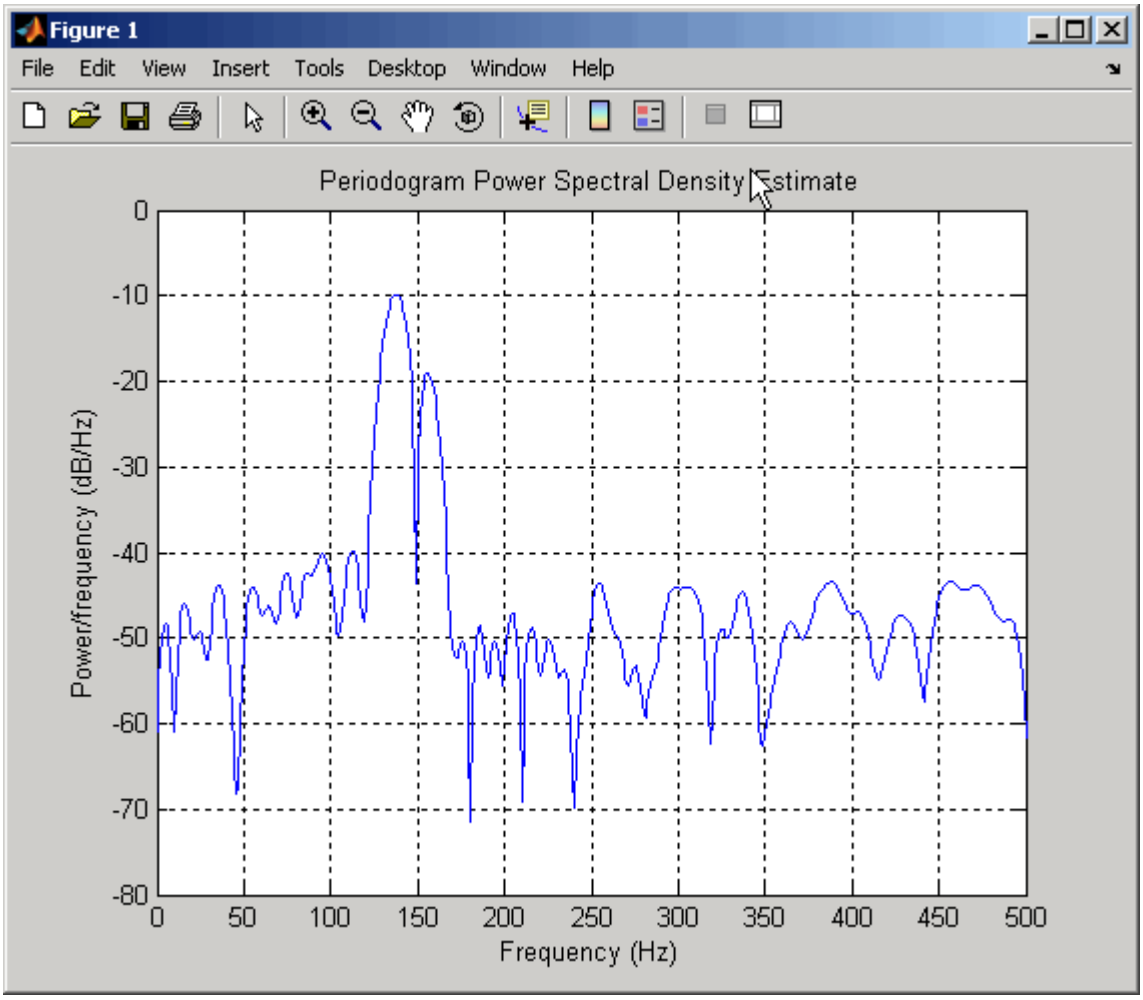
The *modified periodogram* windows the time-domain signal prior to computing the FFT in order to smooth the edges of the signal. This has the effect of reducing the height of the sidelobes or spectral leakage. This phenomenon gives rise to the interpretation of sidelobes as spurious frequencies introduced into the signal by the abrupt truncation that occurs when a rectangular window is used. For nonrectangular windows, the end points of the truncated signal are attenuated smoothly, and hence the spurious frequencies introduced are much less severe. On the other hand, nonrectangular windows also broaden the mainlobe, which results in a net reduction of resolution.

The `periodogram` function allows you to compute a modified periodogram by specifying the window to be used on the data. For example, compare a default rectangular window and a Hamming window:

```
randn('state',0)
fs = 1000;                    % Sampling frequency
t = (0:fs/10)./fs;            % One-tenth second worth of samples
A = [1 2];                    % Sinusoid amplitudes
f = [150;140];                % Sinusoid frequencies
xn = A*sin(2*pi*f*t) + 0.1*randn(size(t));
Hrect = spectrum.periodogram;
psd(Hrect,xn,'Fs',fs,'NFFT',1024);
```




```
Hhamm = spectrum.periodogram('Hamming');
psd(Hhamm,xn,'Fs',fs,'NFFT',1024);
```



You can verify that although the sidelobes are much less evident in the Hamming-windowed periodogram, the two main peaks are wider. In fact, the 3 dB width of the mainlobe corresponding to a Hamming window is approximately twice that of a rectangular window. Hence, for a fixed data length, the PSD resolution attainable with a Hamming window is approximately half that attainable with a rectangular window. The competing interests of mainlobe width and sidelobe height can be resolved to some extent by using variable windows such as the Kaiser window.

Nonrectangular windowing affects the average power of a signal because some of the time samples are attenuated when multiplied by the window. To compensate for this, the `periodogram` function normalizes the window to have an average power of unity. This way the choice of window does not affect the average power of the signal.

The modified periodogram estimate of the PSD is

$$\hat{P}_{xx}(f) = \frac{|X_L(f)|^2}{f_s L U}$$

where *U* is the window normalization constant

$$U = \frac{1}{L}\sum_{n=0}^{L-1} |w(n)|^2$$

which is independent of the choice of window. The addition of *U* as a normalization constant ensures that the modified periodogram is asymptotically unbiased.

Welch's Method

An improved estimator of the PSD is the one proposed by Welch [8]. The method consists of dividing the time series data into (possibly overlapping) segments, computing a modified periodogram of each segment, and then averaging the PSD estimates. The result is Welch's PSD estimate.

Welch's method is implemented in the toolbox by the `spectrum.welch` object or `pwelch` function. By default, the data is divided into eight segments with 50% overlap between them. A Hamming window is used to compute the modified periodogram of each segment.

The averaging of modified periodograms tends to decrease the variance of the estimate relative to a single periodogram estimate of the entire data record. Although overlap between segments tends to introduce redundant information, this effect is diminished by the use of a nonrectangular window, which reduces the importance or *weight* given to the end samples of segments (the samples that overlap).

However, as mentioned above, the combined use of short data records and nonrectangular windows results in reduced resolution of the estimator. In summary, there is a trade-off between variance reduction and resolution. One can manipulate the parameters in Welch's method to obtain improved estimates relative to the periodogram, especially when the SNR is low.

For a more detailed discussion of Welch's method of PSD estimation, see Kay [2] and Welch [8].

Bias and Normalization in Welch's Method

Welch's method yields a biased estimator of the PSD. The expected value can be found to be

$$E\{\hat{P}_{welch}\} = \frac{1}{f_s L_s U} \int_{-f_s/2}^{f_s/2} P_{xx}(\rho)|W(f-\rho)|^2 d\rho$$

where L_S is the length of the data segments and U is the same normalization constant present in the definition of the modified periodogram. As is the case for all periodograms, Welch's estimator is asymptotically unbiased. For a fixed length data record, the bias of Welch's estimate is larger than that of the periodogram because $L_S < L$.

The variance of Welch's estimator is difficult to compute because it depends on both the window used and the amount of overlap between segments. Basically, the variance is inversely proportional to the number of segments whose modified periodograms are being averaged.

References

[1] Hayes, M.H. *Statistical Digital Signal Processing and Modeling*. New York: John Wiley & Sons, 1996.

[2] Kay, S.M. *Modern Spectral Estimation*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[3] Marple, S.L. *Digital Spectral Analysis*. Englewood Cliffs, NJ: Prentice Hall, 1987.

[4] Orfanidis, S.J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996.

[5] Percival, D.B., and A.T. Walden. *Spectral Analysis for Physical Applications: Multitaper and Conventional Univariate Techniques*. Cambridge: Cambridge University Press, 1993.

[6] Proakis, J.G., and D.G. Manolakis. *Digital Signal Processing: Principles, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1996.

[7] Stoica, P., and R. Moses. *Introduction to Spectral Analysis*. Upper Saddle River, NJ: Prentice Hall, 1997.

[8] Welch, P.D. " *The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms.IEEE Trans. Audio Electroacoust*. Vol. AU-15 (June 1967). Pgs.70-73.

Spectral Estimation Spectral Windows

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?

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

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

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

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

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

Spectral Windows Create spectral windows

Create spectral windows

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

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

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

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

List of available window functions

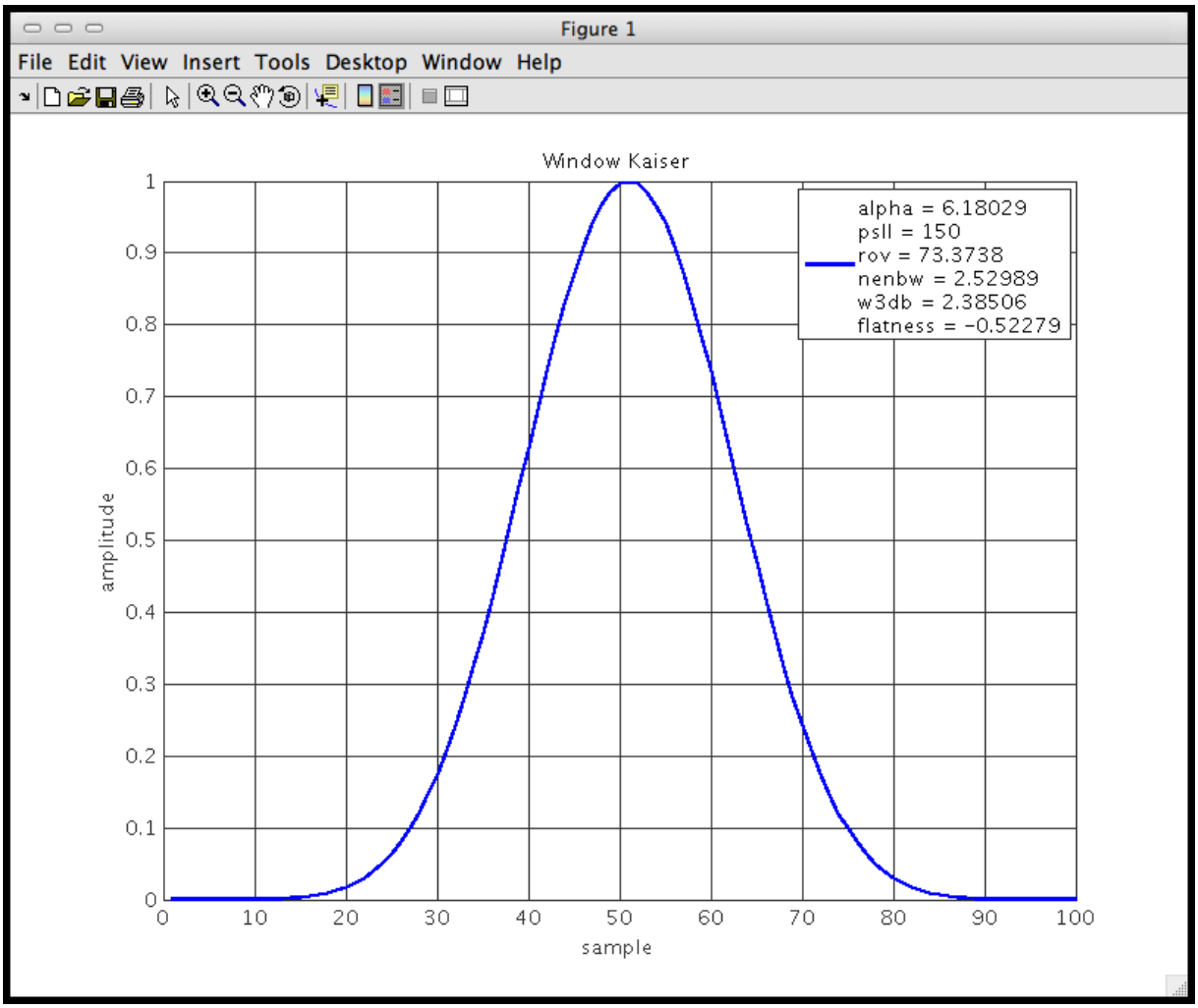
In the special case of creating a Kaiser window, the additional input parameter, PSLL, must be supplied. For example, the following code creates a 100-point Kaiser window with -150dB peak side-lobe level:

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

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

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

Visualising spectral windows

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

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



Windows can also be visualised using the Spectral Window Viewer.

Using spectral windows

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

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

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

Many of the LTPDA algorithms that expect a spectral window parameter can also work with just the name of the window being specified. For example, the above code can also be written as

```
axx = psd(a_in, 'Win', 'Kaiser');
```

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

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

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

Power spectral density estimates

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

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). 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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]

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

As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'psll'`.

As an alternative to setting the number of points `'Nfft'` in each window, it's possible to ask for

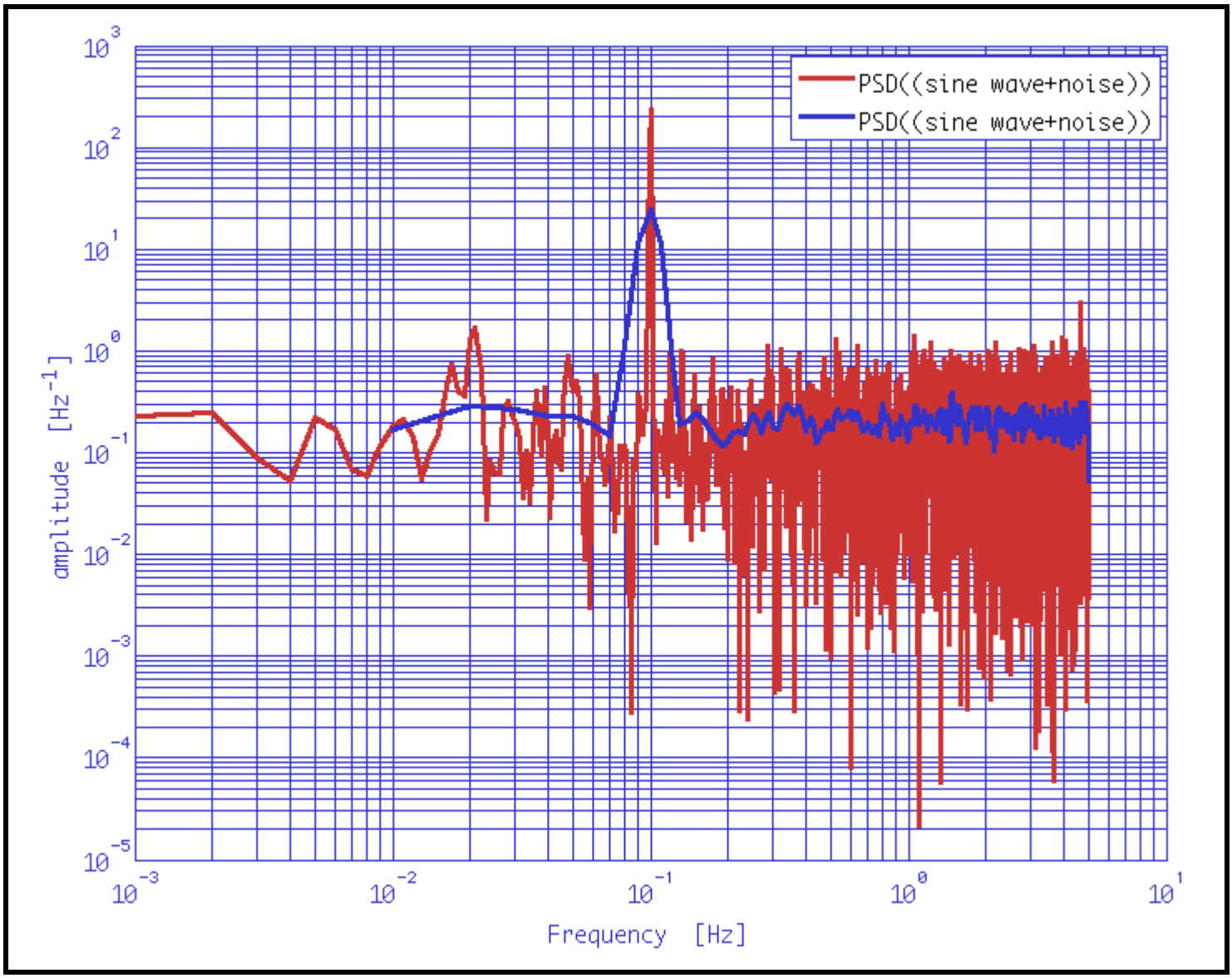
a given number of PSD estimates by setting the `'Navs'` parameter, and the algorithm takes care of calculating the correct window length, according to the amount of overlap between subsequent segments.

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

Examples

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

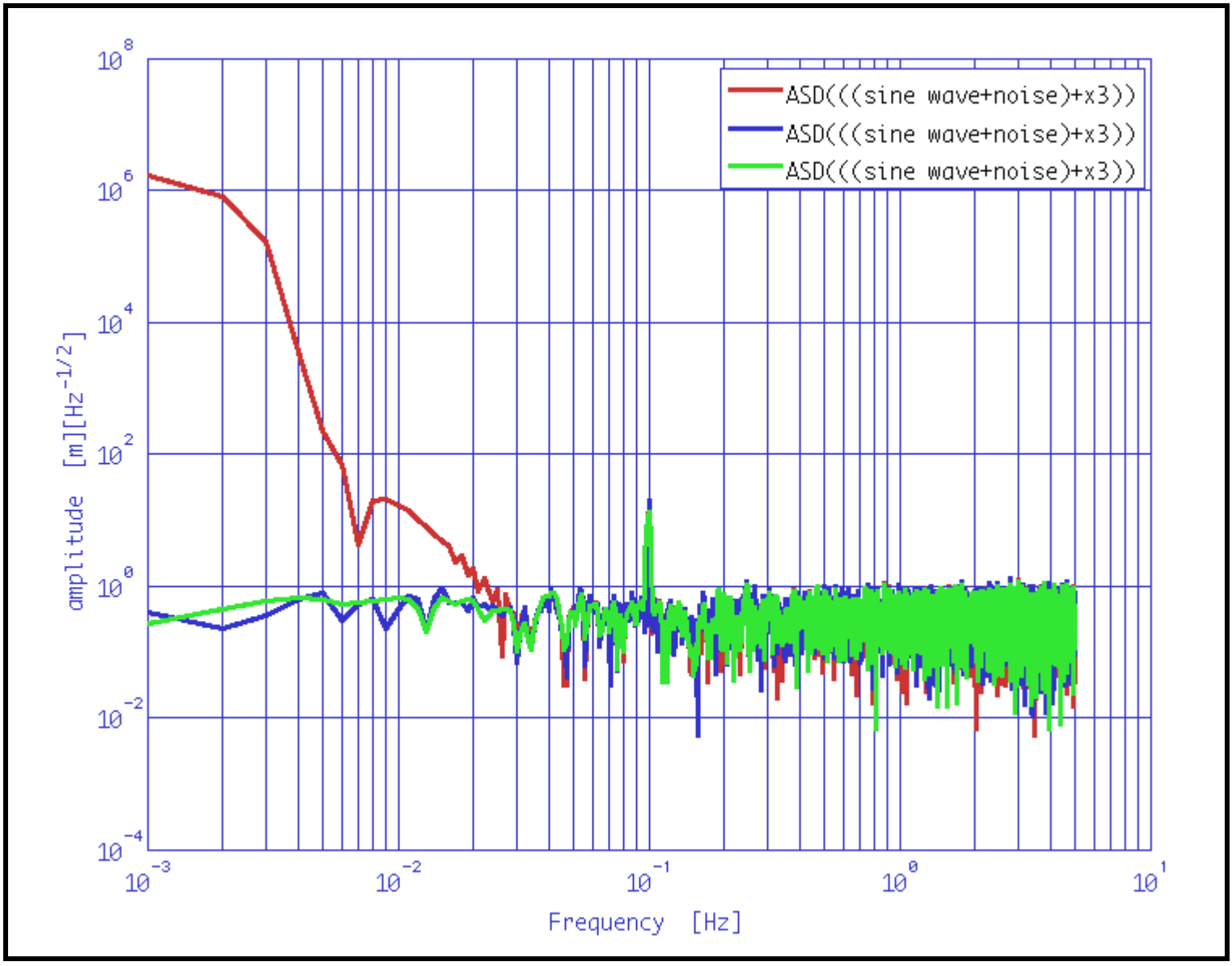
```
x1   = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10));
x2   = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
x    = x1 + x2;
y_lf = psd(x);
y_hf = psd(x,plist('nfft',1000));
iplot(y_lf, y_hf)
```



2. Evaluation of the PSD of a time-series represented by a low frequency sinewave signal, superimposed to white noise and to a low frequency linear drift. In the example, the same spectrum is computed with different spectral windows.

```
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'));
x = x1 + x2 + x3;
y_1 = psd(x,plist('scale','ASD','order',1,'win',specwin('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));
iplot(y_1, y_2, y_3);
```



Using spectral windows

Cross-spectral density estimates

Cross-spectral density estimates

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

Syntaxis

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

`a1` and `a2` are the 2 `ao`s containing the input time series to be evaluated; `b` is the output object. 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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]

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

As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'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.

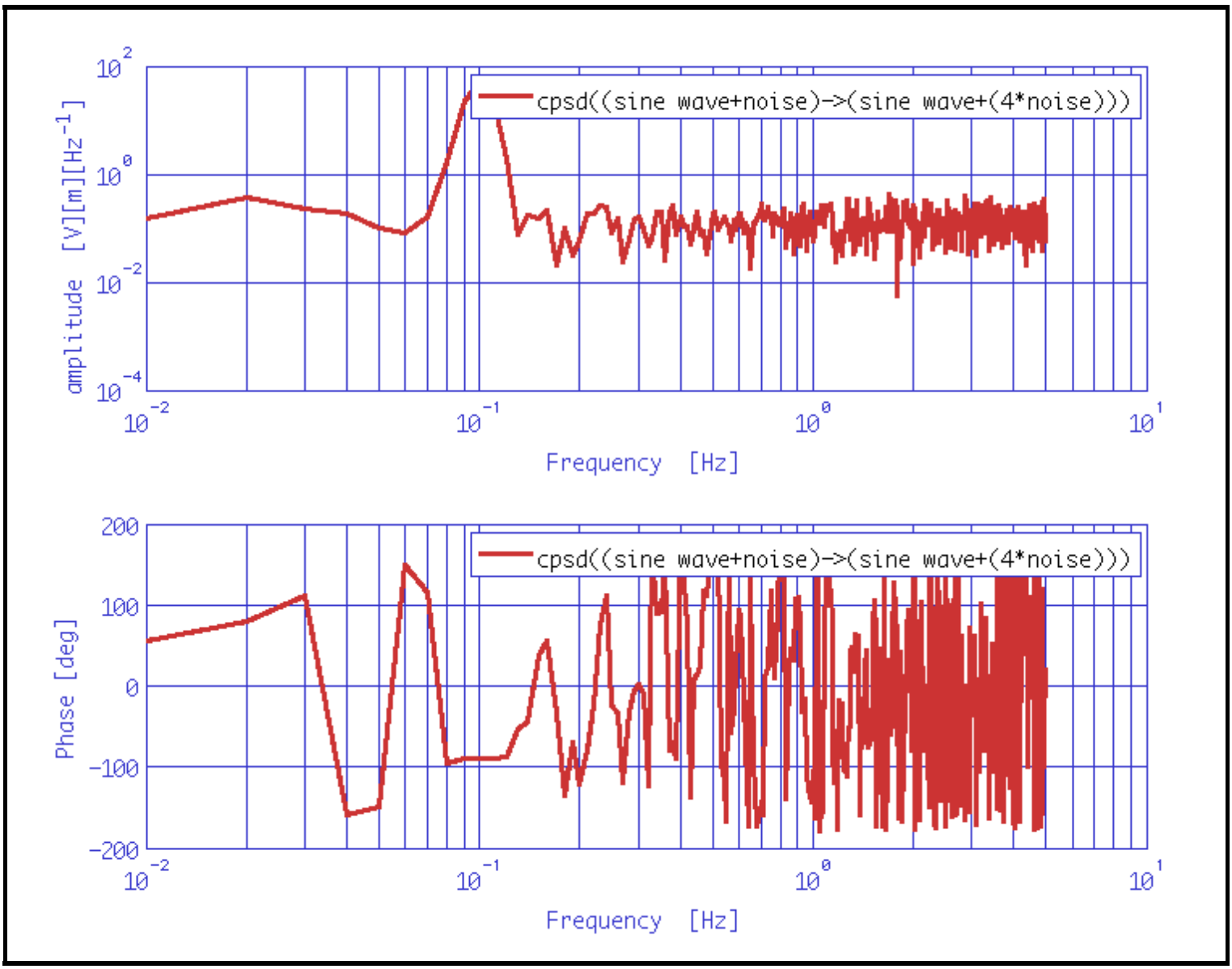
The function makes CPSD estimates between the 2 input `ao`s. The input argument list must

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)`.

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));
iplot(z);
```



Power spectral density estimates
Cross coherence estimates

Cross coherence estimates

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

Syntaxis

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

`a1` and `a2` are the 2 `ao`s containing the input time series to be evaluated; `b` is the output object. 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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]

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

As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'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.

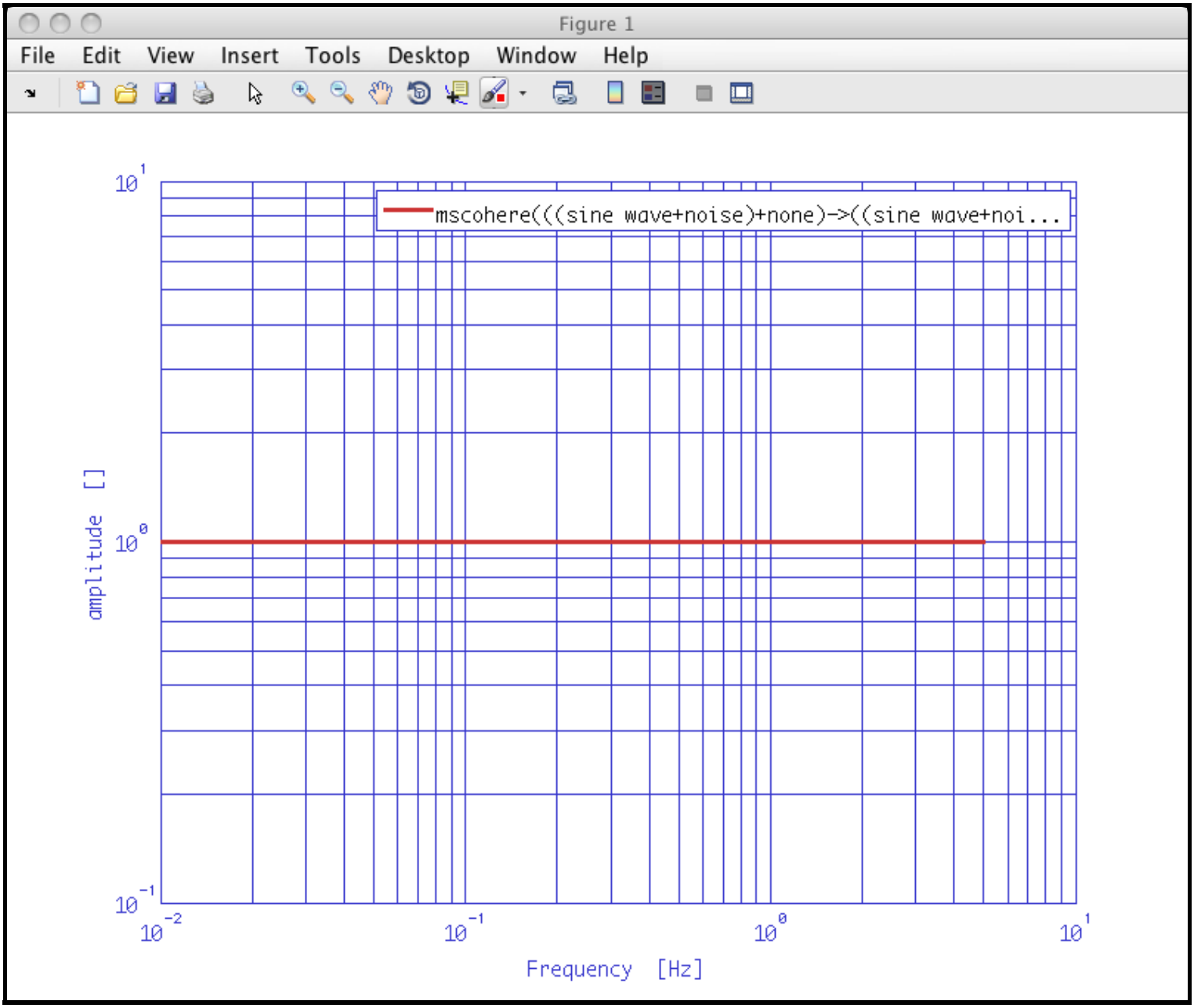
The function makes magnitude-squadred coherence estimates between the 2 input `ao`s. If passing two identical objects `ai` or linearly combined signals, the output will be 1 at all frequencies. The same will happen if analyzing only a single window.

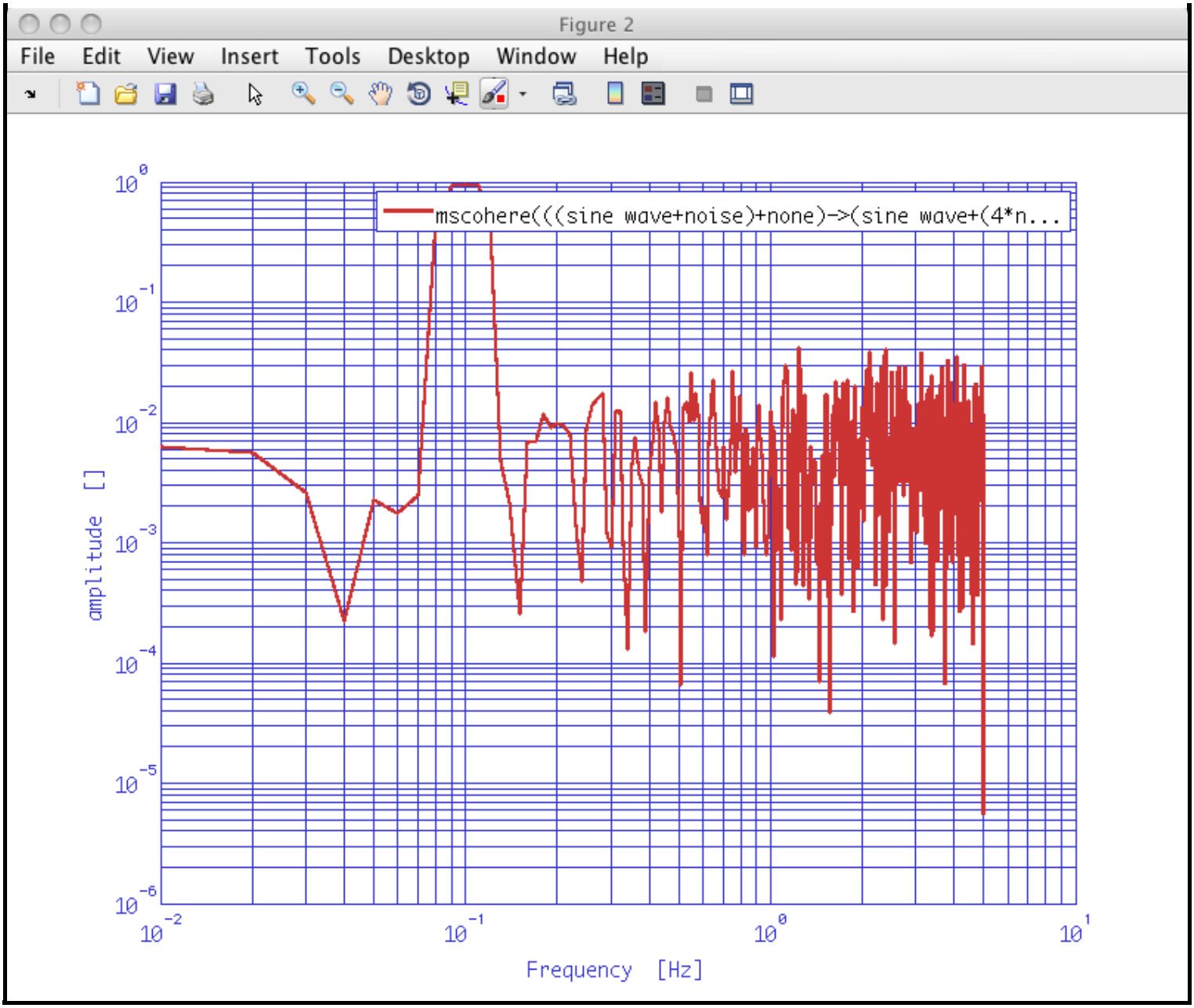
Example

Evaluation of the magnitude-squadred 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.

```
nsecs = 5000;
fs = 10;
x = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',nsecs,'fs',fs,'yunits','m')) + ...
    ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs,'yunits','m')) + ...
    ao(plist('tsfcn', 't','nsecs',nsecs,'fs',fs,'yunits','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');
nfft = 1000;
pl = plist('win',specwin('BH92'),'nfft',nfft, 'order',1);
Cxx  = cohere(x,x,pl);
Cxy  = cohere(x,y,pl);
iplot(Cxx);
iplot(Cxy);
```





Cross-spectral density estimates
Transfer function estimates

Transfer function estimates

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

Syntaxis

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

`a1` and `a2` are the 2 `ao`s containing the input time series to be evaluated; `b` is the output object. 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - - a string value containing the window name e.g., 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
- `'Variance'` - computes transfer function variance.
  - 'yes' - compute tf variance
  - 'no' - do not compute tf variance [default]
- `'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]

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

As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'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.

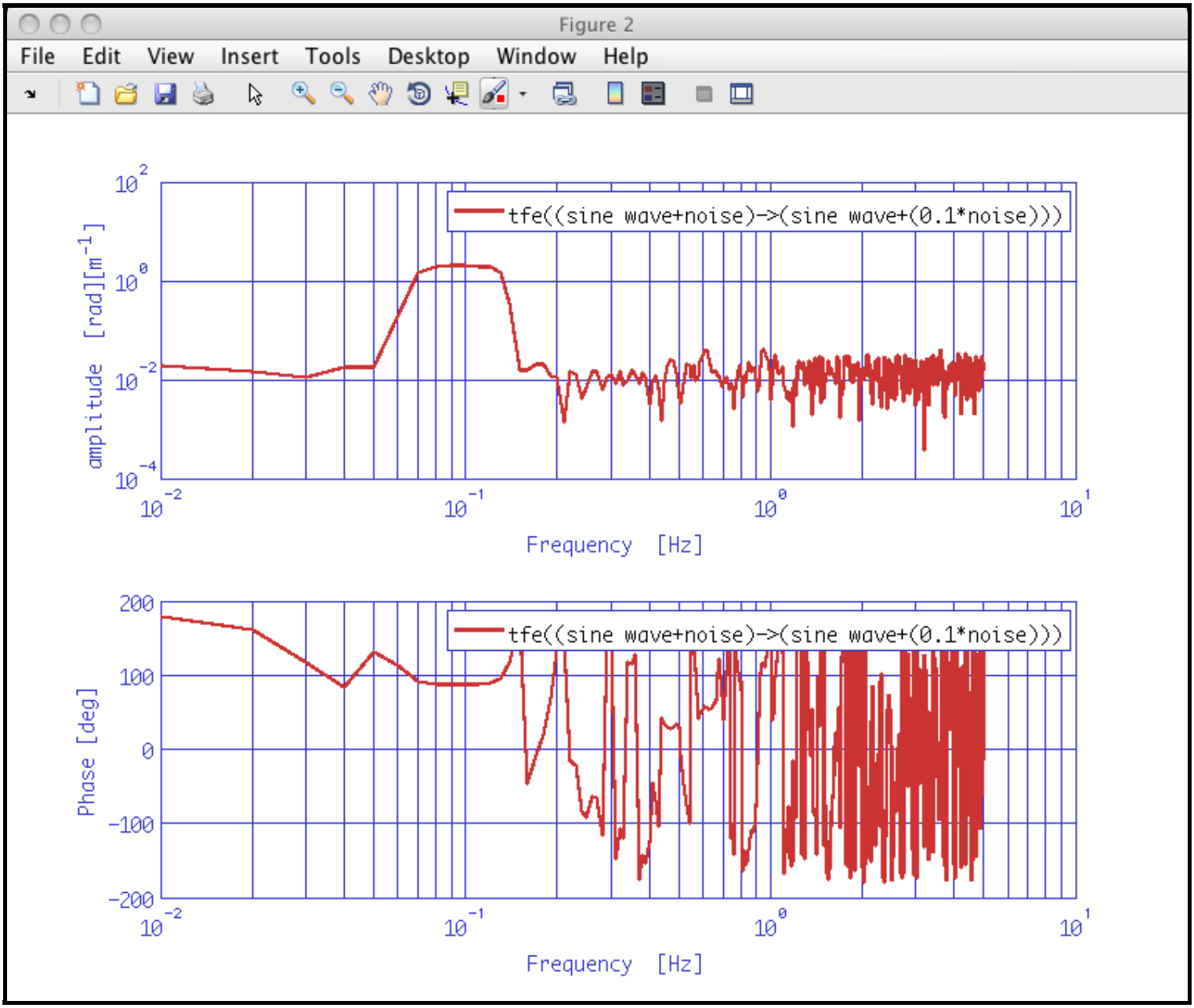
If the user doesn't specify the value of a given parameter, the default value is used.

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

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.

```
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)) + ...
    0.1*ao(plist('waveform','noise','type','normal','nsecs',nsecs,'fs',fs));
y.setYunits('rad');
nfft = 1000;
psll = 200;
Txy = tfe(x,y,plist('win','Kaiser','psll',psll,'nfft',nfft));
Tyx = tfe(y,x,plist('win','Kaiser','psll',psll,'nfft',nfft));
Txx = tfe(x,x,plist('win','Kaiser','psll',psll,'nfft',nfft));
Tyy = tfe(y,y,plist('win','Kaiser','psll',psll,'nfft',nfft));
iplot(Txy);
iplot(Tyx);
iplot(Txx);
iplot(Tyy);
```



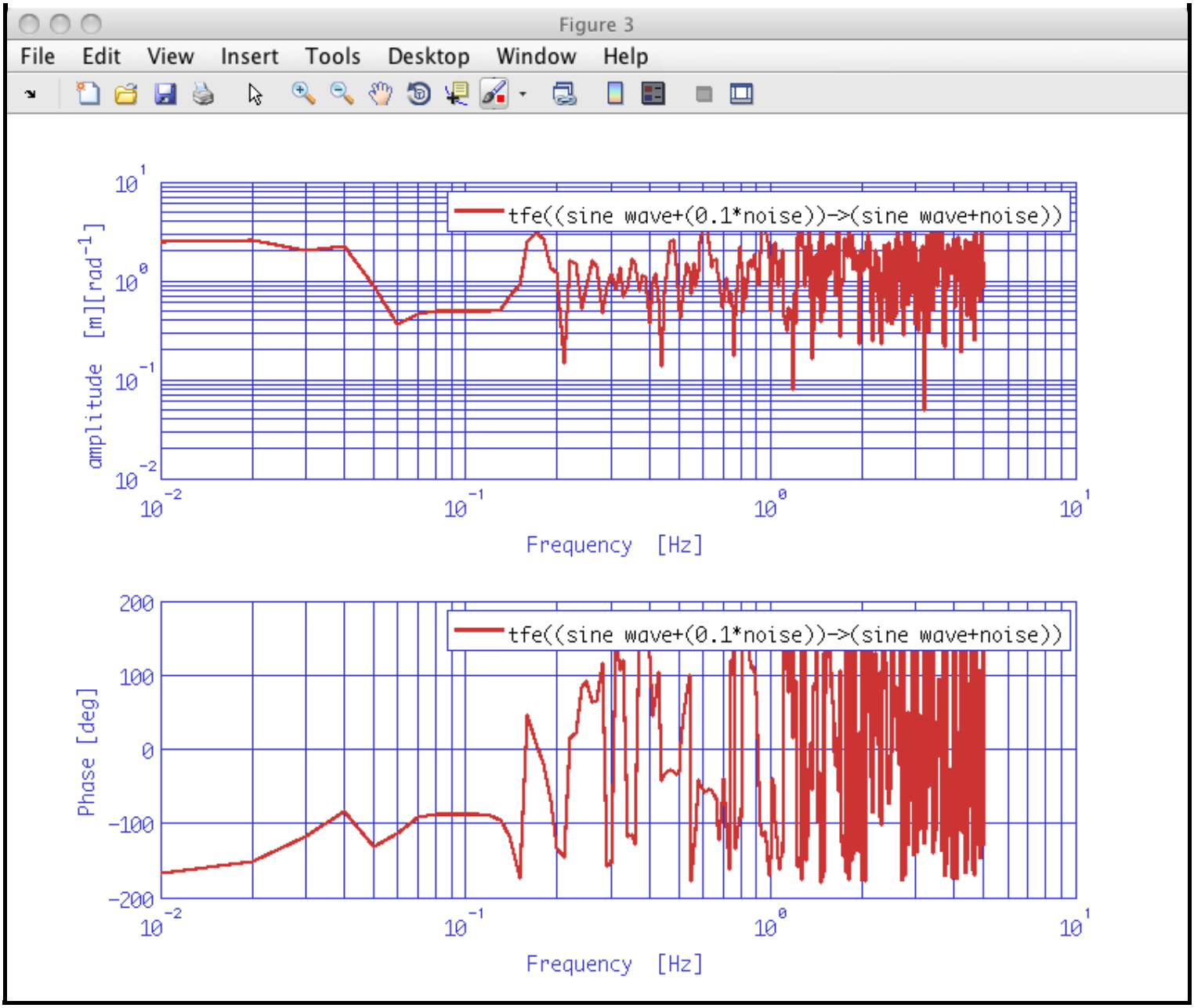


Figure 3
File Edit View Insert Tools Desktop Window Help
tfe((sine wave+(0.1*noise))->(sine wave+noise))
amplitude [m][rad^-1]
Frequency [Hz]
tfe((sine wave+(0.1*noise))->(sine wave+noise))
Phase [deg]
Frequency [Hz]

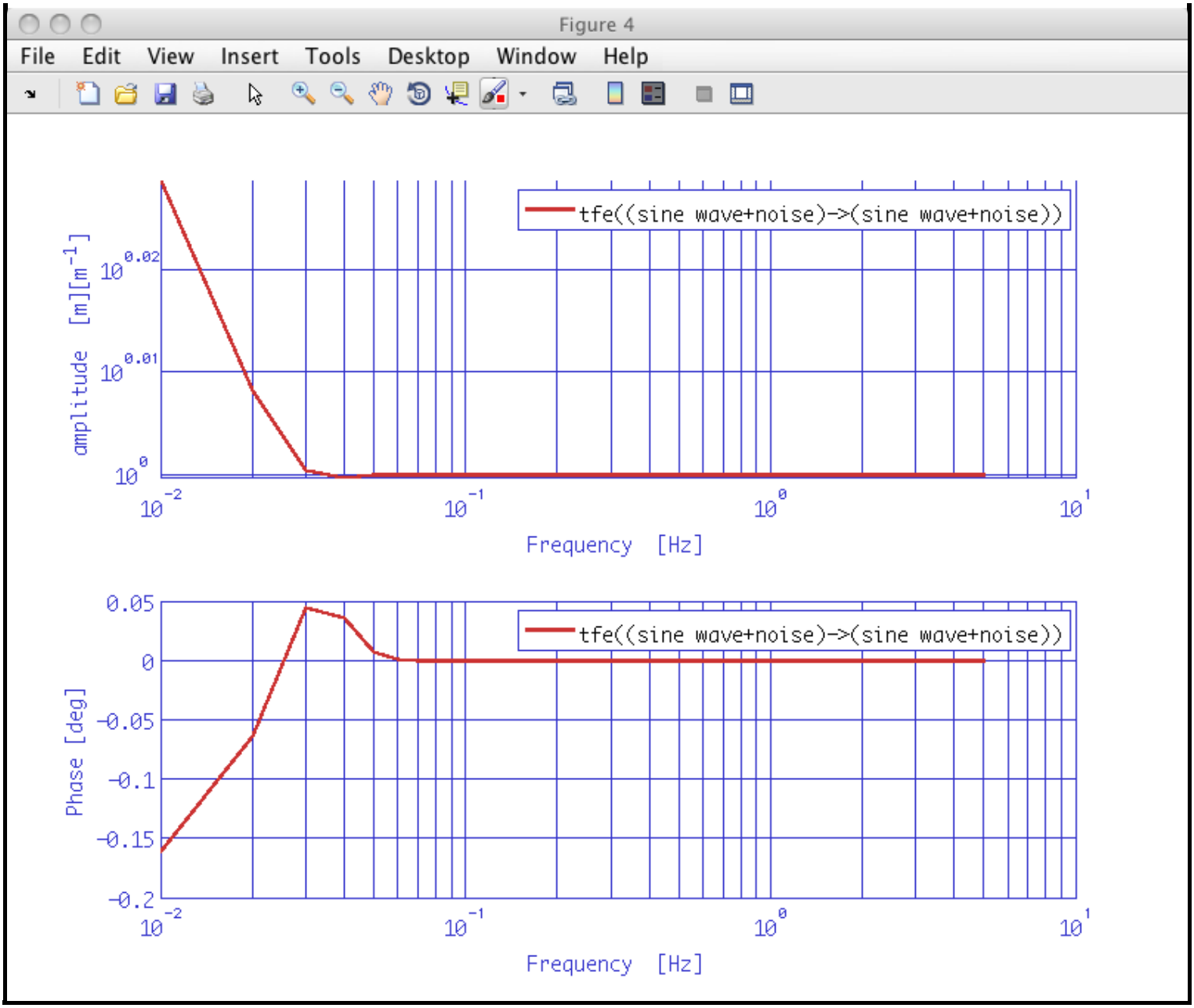
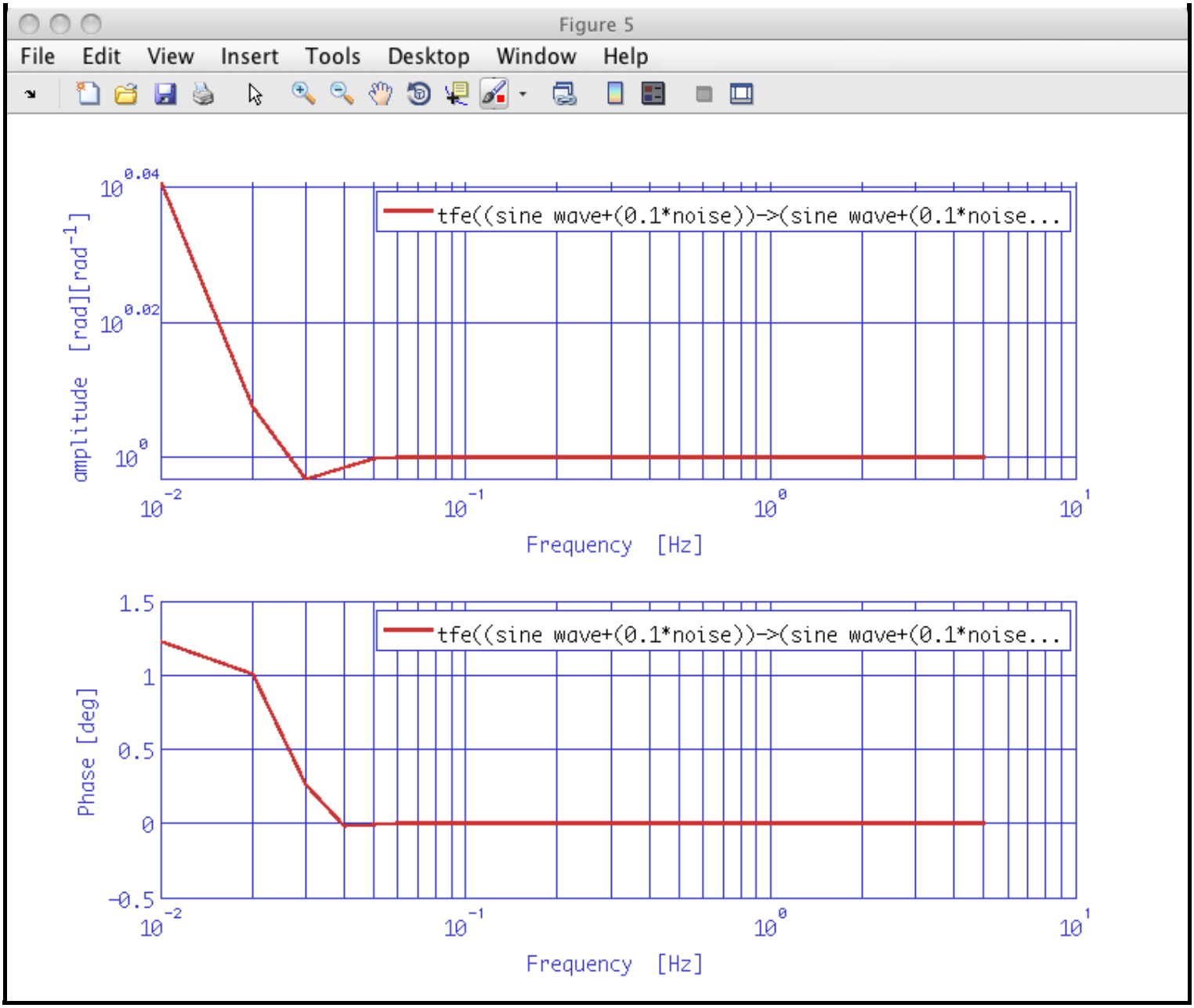


Figure 4
File Edit View Insert Tools Desktop Window Help
tfe((sine wave+noise)->(sine wave+noise))
amplitude [m][m^-1]
10^0.02
10^0.01
10^0
10^-2
10^-1
10^0
10^1
Frequency [Hz]
tfe((sine wave+noise)->(sine wave+noise))
Phase [deg]
0.05
0
-0.05
-0.1
-0.15
-0.2
10^-2
10^-1
10^0
10^1
Frequency [Hz]



Cross coherence estimates

Log-scale power spectral density estimates

Log-scale power spectral density estimates

Univariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm (Measurement 39 (2006) 120-129), which is an application of Welch's averaged, modified periodogram method. Spectral density estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T` where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

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

Syntaxis

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

`a1`, `a2`, `a3`, ... are `ao`(s) containing the input time series to be evaluated. `bs` includes the output object(s). 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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 Keiser windows, the PSLL.

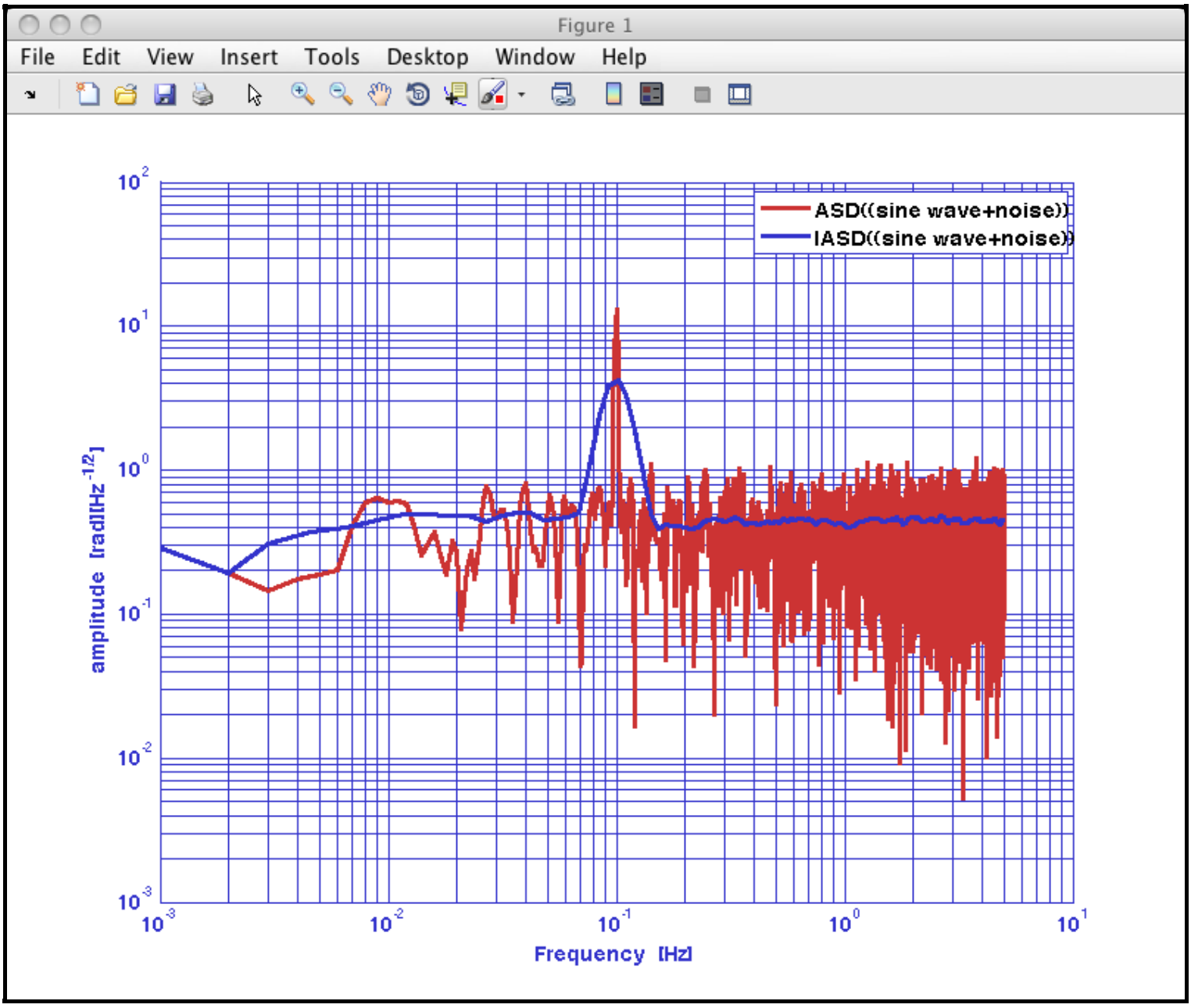
As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'psll'`.

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

Examples

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

```
x1 = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10,'yunits','rad'));
x2 = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10,'yunits','rad'));
x  = x1 + x2;
pl = plist('scale','ASD','order',-1,'win','Kaiser','psll',200);
y1 = psd(x, pl);
y2 = lpsd(x, pl);
iplot(y1, y2)
```



Transfer function estimates
Log-scale cross-spectral density estimates

Log-scale cross-spectral density estimates

Multivariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm (Measurement 39 (2006) 120-129), which is an application of Welch's averaged, modified periodogram method. Cross-spectral density estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T` where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

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

Syntaxis

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

`a1` and `a2` are the 2 `ao`s containing the input time series to be evaluated; `b` is the output object. 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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 Keiser windows, the PSLL.

As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'psll'`.

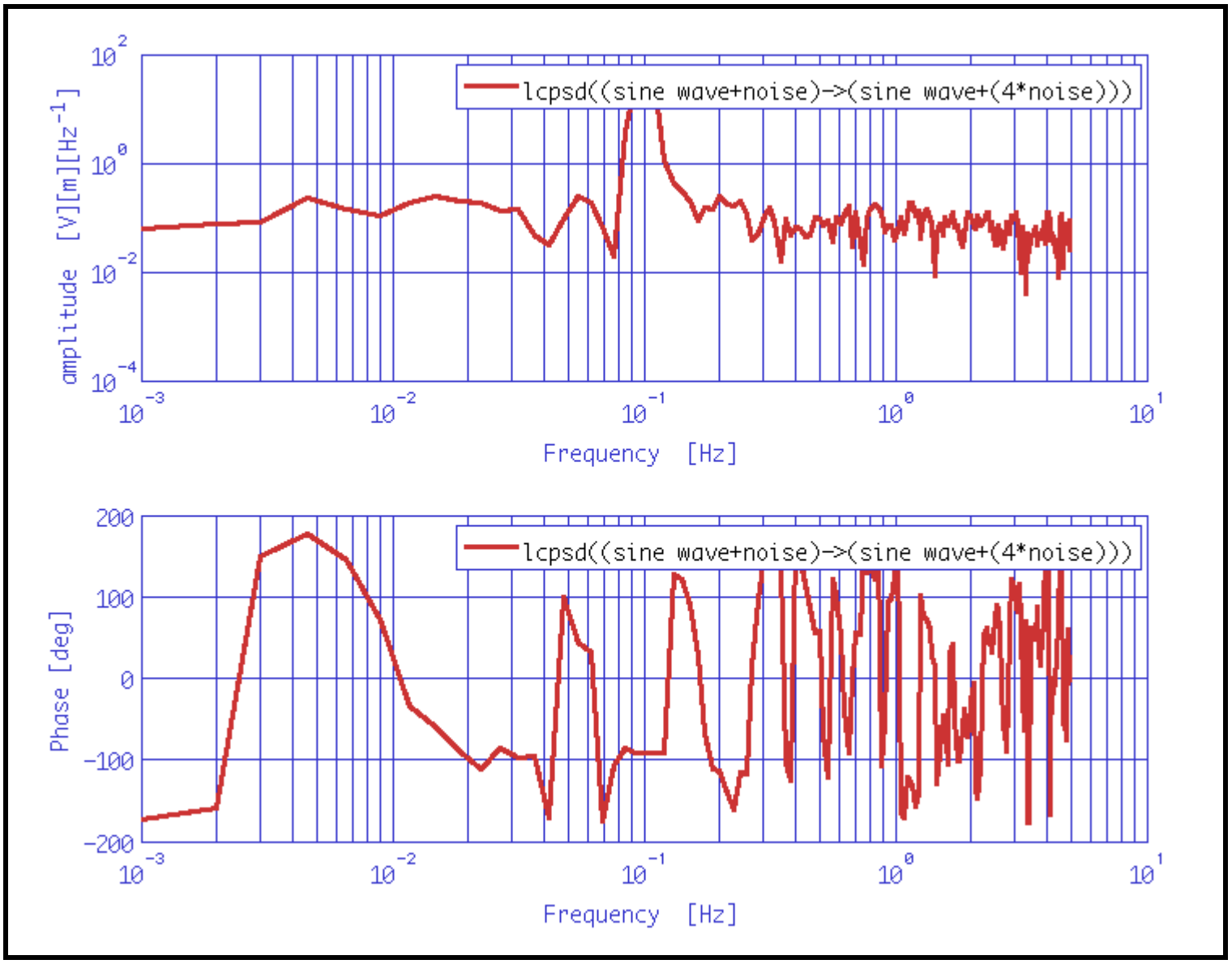
If the user doesn't specify the value of a given parameter, the default value is used.

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 ai, the output will be equivalent to the output of lpsd(ai).

Example

Evaluation of the log-scale 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;
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');
z = lcpsd(x,y,plist('nfft',1000));
iplot(z);
```



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

Log-scale cross coherence density estimates

Multivariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm (Measurement 39 (2006) 120-129), which is an application of Welch's averaged, modified periodogram method. The magnitude-squadred coherence of time-series signals, included in the input aos are not evaluated at freqencies which are linear multiples of the minimum frequency resolution 1/T where T is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

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

Syntaxis

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

a1 and a2 are the 2 aos containing the input time series to be evaluated; b is the output object. 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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
- 'Variance' - computes transfer function variance.
  - 'yes' - compute tf variance
  - 'no' - do not compute tf variance [default]

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

As an alternative, the user can input, as a value for the 'Win' key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter 'psll'.

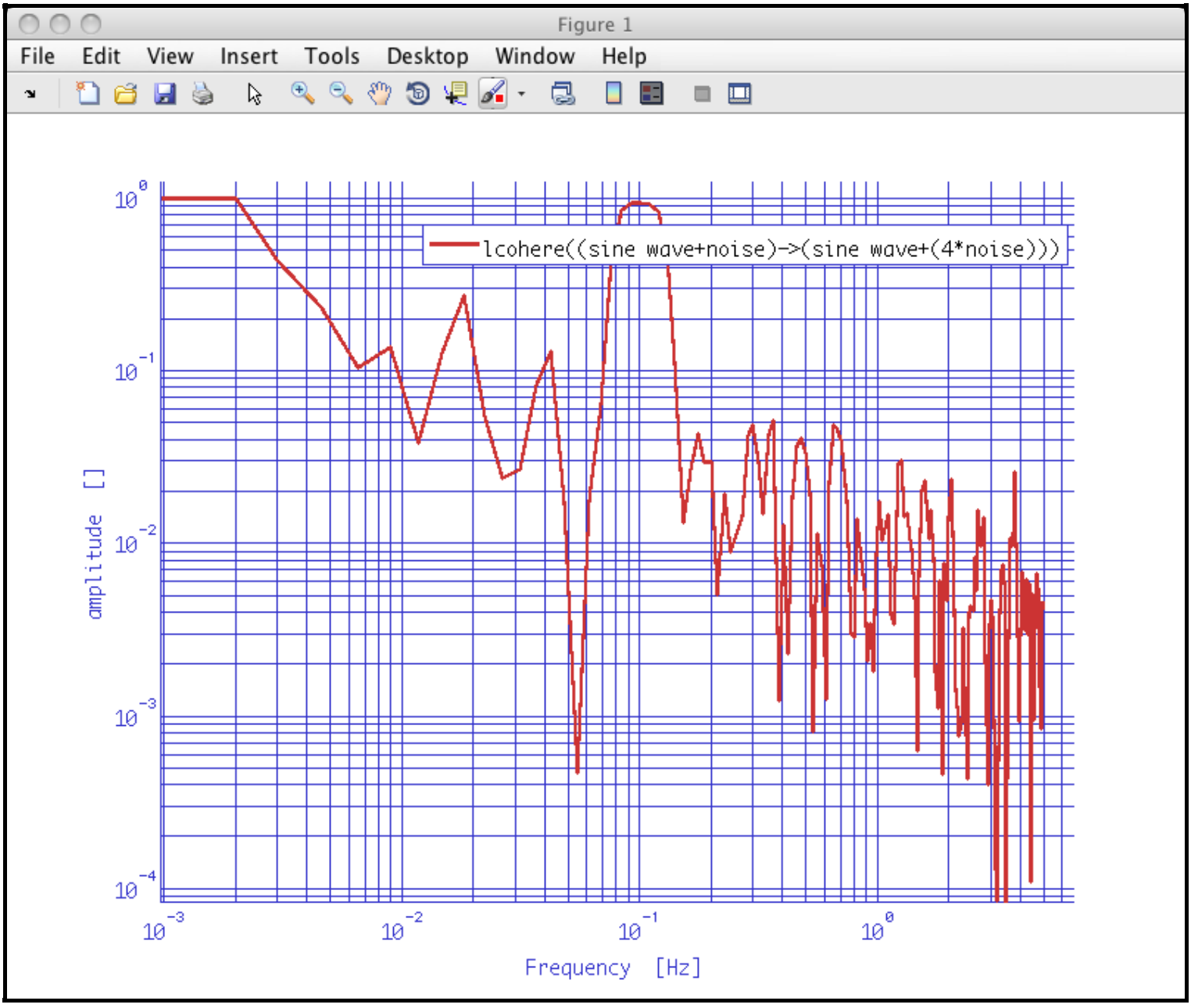
If the user doesn't specify the value of a given parameter, the default value is used.

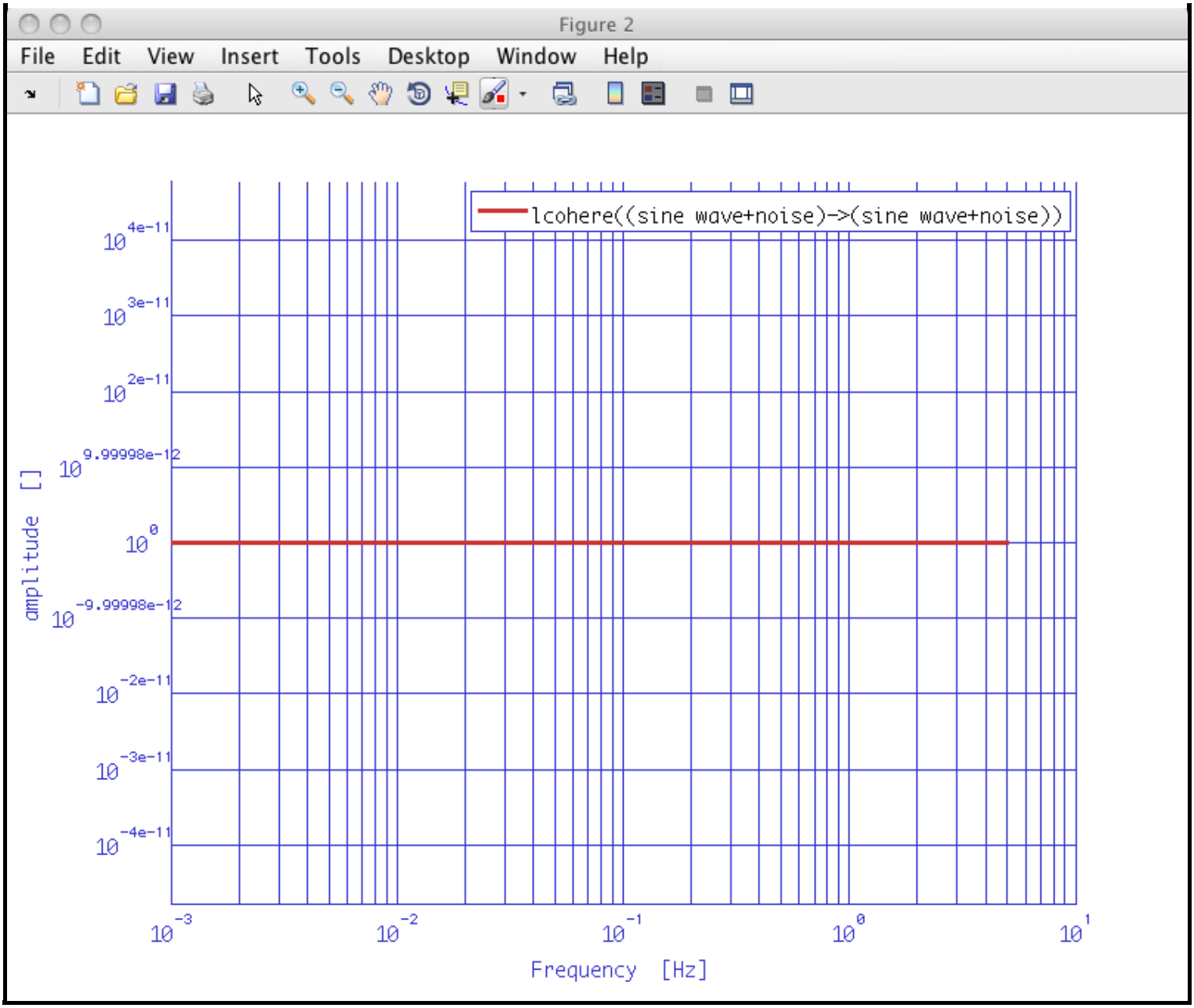
The function makes magnitude-squadred coherence estimates between the 2 input aos, on a logaritmic frequency scale. If passing two identical objects ai or linearly combined signals, the output will be 1 at all frequencies.

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.

```
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');
Cxx = lcohere(x,x,plist('win','Kaiser','psll',200));
Cxy = lcohere(x,y,plist('win','Kaiser','psll',200));
iplot(Cxx);
iplot(Cxy);
```





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

Log-scale transfer function estimates

Multivariate power spectral density on a logarithmic scale can be performed by the LPSD algorithm (Measurement 39 (2006) 120-129), which is an application of Welch's averaged, modified periodogram method. The transfer function estimates are not evaluated at freqencies which are linear multiples of the minimum frequency resolution `1/T`, where `T` is the window lenght, but on a logarithmic scale. The algorithm takes care of calculating the frequencies at which to evaluate the spectral estimate, aiming at minimizing the uncertainty in the estimate itself, and to recalculate a suitable window length for each frequency bin.

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

Syntaxis

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

`a1` and `a2` are the 2 `ao`s containing the input time series to be evaluated; `b` is the output object. 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] Only the design parameters of the window object are used. Enter either:
  - a specwin window object OR
  - a string value containing the window name e.g., 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
- `'Variance'` - computes transfer function variance.
  - 'yes' - compute tf variance
  - 'no' - do not compute tf variance [default]

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

As an alternative, the user can input, as a value for the `'Win'` key, a string corresponding to the name of the window. In the case of Kaiser window, it's necessary to specify the additional parameter `'psll'`.

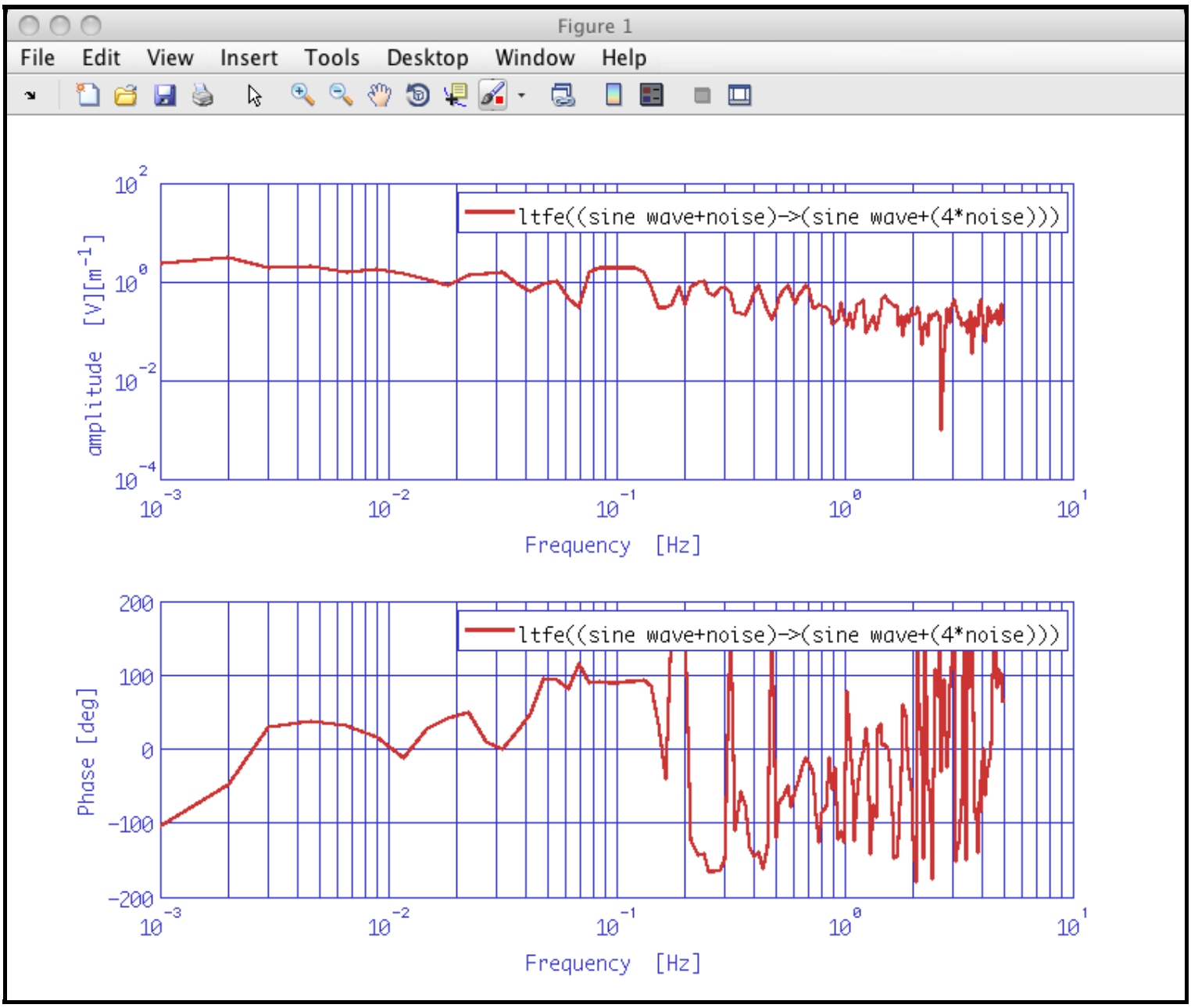
If the user doesn't specify the value of a given parameter, the default value is used.

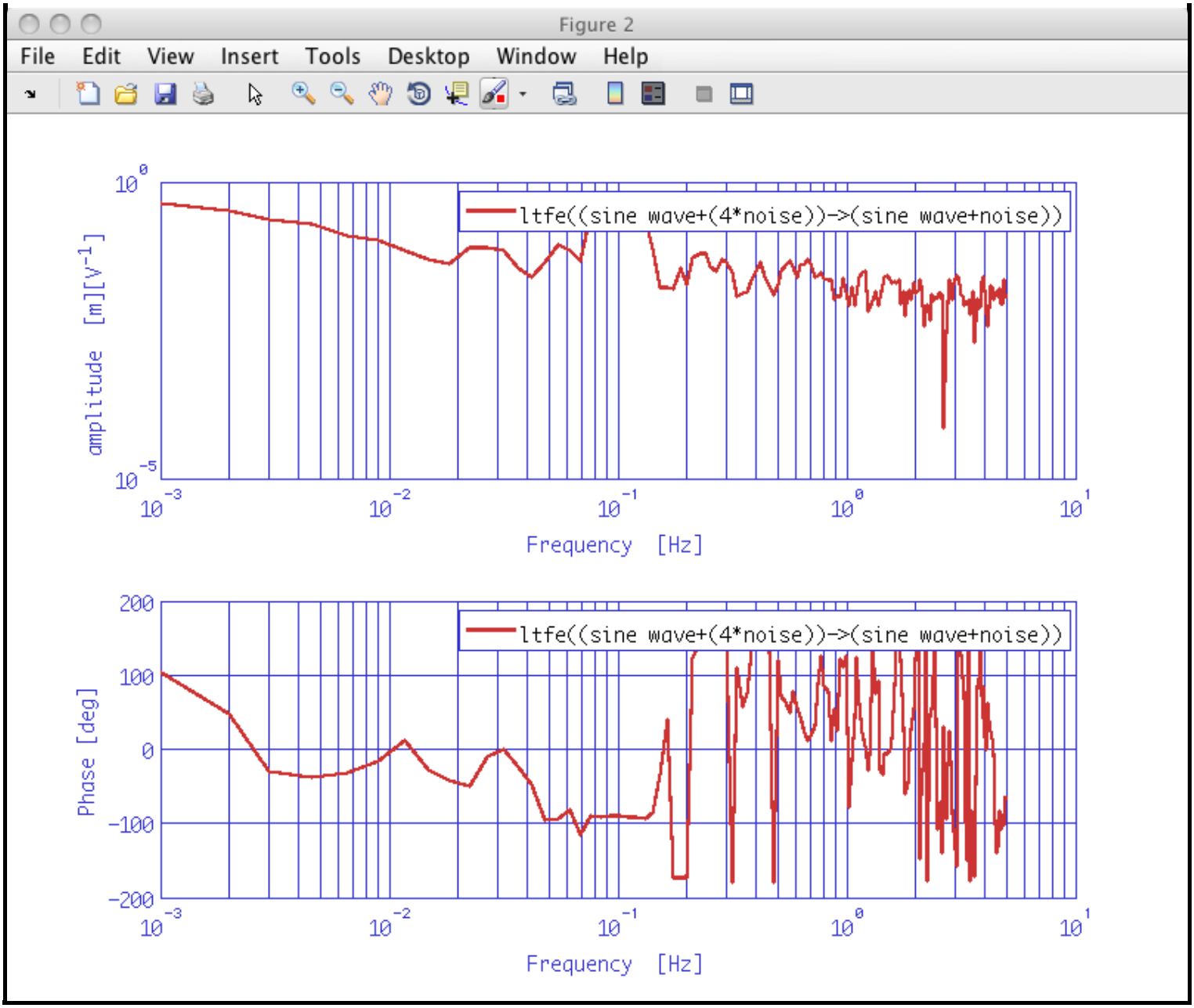
The function makes logaritmic frequencyscale transfer functions estimates between the 2 input `ao`s, and the output will contain the transfer function estimate from the first `ao` to the second.

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.

```
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');
nfft = 1000;
Txy = ltfe(x,y,plist('win',specwin('Kaiser',1,200),'nfft',nfft));
Tyx = ltfe(y,x,plist('win',specwin('Kaiser',1,200),'nfft',nfft));
iplot(Txy);
iplot(Tyx);
```





Log-scale cross coherence density estimates

Fitting Algorithms

Fitting Algorithms

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

Log-scale transfer function estimates Polynomial Fitting

Polynomial Fitting

---

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

- polyfit.m
- polyval.m

Usage

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

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

Time domain Fit

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

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

An example:

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

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

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

Z-Domain Fit

System identification in z-domain is performed with the function `ao/zDomainFit`. It is based on a modeified version of the vector fitting algorithm that was chenged to fit in z-domain. Details on the agorithm can be found in [1 - 3].

Call

```
 mod = zDomainFit(a, pl)
[mod, resp] = zDomainFit(a, pl)
[mod, resp, resids] = zDomainFit(a, pl)
[mod, resp, resids, rmse] = zDomainFit(a, pl)
```

Inputs

- a - input AOs to fit to. 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 will be ignored. Input objects must have the same number of elements.
- pl - parameter list. See the list of function parameeters below.

Outputs

- mod - model, a bank of parallel miir filters for each input AO.
- resp - model frequency response.
- resids - analysis object containing the fit residuals.
- rmse - analysis object containing the root mean squared error progression during the fitting loop.

Parameters

Key	Value	Description
'FS'		Sampling frequency. If it is left empty, the sampling frequency is searched in the input AOs or it is calculated as 2 times the maximum frequency reported in AOs x values. [Default []]
'AutoSearch'	'on'	Parform a full automatic search for the transfer function order. The fitting procedure stops when stop conditions defined by 'ResLogDiff' and 'RMSE' are satisfied. [Default]
'AutoSearch'	'off'	Perform a fitting loop as long as the number of

		iteration reach 'maxiter'. The order of the fitting function is that specified in 'minorder'.
'StartPoles'	[]	A vector of starting poles. Providing a fixed set of starting poles fixes the function order. If it is left empty, the starting poles are internally assigned. [Default []].
'StartPolesOpt'	'real'	Start with linearly spaced real poles.
'StartPolesOpt'	'c1'	Complex poles distributed inside the unitary circle of the complex plane. [Default]
'StartPolesOpt'	'c2'	Complex poles distributed inside the unitary circle of the complex plane.
'maxiter'		Maximum number of allowed iteration. [Deafult 50]
'minorder'		Minimum model function order. [Default 2]
'maxorder'		Maximum model function order. [Default 20]
'weights'	[]	A vector with the desired weights. [Default []]
'weightparam'	'ones'	Assign equal weights to each element of the dataset.
'weightparam'	'abs'	Assign weights as `1./abs(y)`. [Default]
'weightparam'	'sqrt'	Assign weights as `1./sqrt(abs(y))`.
'ResLogDiff'		Check if the log difference between data and residuals is larger than the value indicated. Leave it empty ([]) if you want to use the residuals spectral flatness criterion for checking fit goodness. [Default 2]
'ResFlat'		Check if the spectral flatness coefficient for the rersiduals is larger than the value assigned. Only values `sp` such that `0 < sp < 1` are allowed. If 'ResLogDiff' is not empty this parameter is ignored. [Default 0.5]
'RMSE'		Check that the variation of root mean squared error is lower than `10^(-1*value)`. [Default 8]
'ForceStability'	'on'	Force fitted poles to be stable
'ForceStability'	'off'	Do not force fitted poles to be stable. [Default]
'Plot'	'on'	Plot fit result.

'Plot'	'off'	Do not plot fit result. [Default]
'CheckProgress'	'on'	Disply the status of the fit iteration.
'CheckProgress'	'off'	Do not disply the status of the fit iteration. [Default]

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 (a parallel bank of `miir` objects):

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

Identification loop stops when the stop condition is reached. Stop criteria are based on two different approachs:

1. 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*value)`.
2. 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*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'.

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

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

Call

```
 mod = sDomainFit(a, pl)
[mod, resp] = sDomainFit(a, pl)
[mod, resp, resids] = sDomainFit(a, pl)
[mod, resp, resids, rmse] = sDomainFit(a, pl)
```

Inputs

- a - input AOs to fit to. 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 will be ignored. Input objects must have the same number of elements.
- pl - parameter list. See the list of function parameeters below.

Outputs

- mod - model, a parfrac object for each input AO.
- resp - model frequency response.
- resids - analysis object containing the fit residuals.
- rmse - analysis object containing the root mean squared error progression during the fitting loop.

Parameters

Key	Value	Description
'AutoSearch'	'on'	Parform a full automatic search for the transfer function order. The fitting procedure stops when stop conditions defined by 'ResLogDiff' and 'RMSE' are satisfied. [Default]
'AutoSearch'	'off'	Perform a fitting loop as long as the number of iteration reach 'maxiter'. The order of the fitting function is that specified in 'minorder'.
'StartPoles'	[]	A vector of starting poles. Providing a fixed set of starting poles fixes the function order. If it is

		left empty, the starting poles are internally assigned. [Default []].
'StartPolesOpt'	'real'	Start with linearly spaced real poles.
'StartPolesOpt'	'clog'	Start with log-spaced complex poles [Default].
'StartPolesOpt'	'clin'	Start with linearly spaced complex poles.
'maxiter'		Maximum number of allowed iteration. [Deafult 50]
'minorder'		Minimum model function order. [Default 2]
'maxorder'		Maximum model function order. [Default 20]
'weights'	[]	A vector with the desired weights. [Default []]
'weightparam'	'ones'	Assign equal weights to each element of the dataset.
'weightparam'	'abs'	Assign weights as `1./abs(y)`. [Default]
'weightparam'	'sqrt'	Assign weights as `1./sqrt(abs(y))`.
'ResLogDiff'		Check if the log difference between data and residuals is larger than the value indicated. Leave it empty ([]) if you want to use the residuals spectral flatness criterion for checking fit goodness. [Default 2]
'ResFlat'		Check if the spectral flatness coefficient for the rersiduals is larger than the value assigned. Only values `sp` such that `0 < sp < 1` are allowed. If 'ResLogDiff' is not empty this parameter is ignored. [Default 0.5]
'RMSE'		Check that the variation of root mean squared error is lower than `10^(-1*value)`. [Default 8]
'ForceStability'	'on'	Force fitted poles to be stable
'ForceStability'	'off'	Do not force fitted poles to be stable. [Default]
'direct term'	'on'	Fit with direct term.
'direct term'	'off'	Fit without direct term. [Default]
'Plot'	'on'	Plot fit result.
'Plot'	'off'	Do not plot fit result. [Default]
'CheckProgress'	'on'	Disply the status of the fit iteration.

'CheckProgress' 'off'	Do not disply the status of the fit iteration. [Default]

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:

```
          r1                rN
f(s) = ------- + ... + ------- + d
       s - p1            s - pN
```

Identification loop stops when the stop condition is reached. Stop criteria are based on two different approachs:

1. 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*value)`.
2. 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*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'.

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

Graphical User Interfaces in LTPDA

LTPDA has a variety of Graphical User Interfaces:

S-Domain Fit The LTPDA Launch Bay

The LTPDA Launch Bay

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

```
>> ltpdalauncher
```

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

The LTPDA 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.

The LTPDA Launch Bay Loading the LTPDA Workbench

Loading the LTPDA Workbench

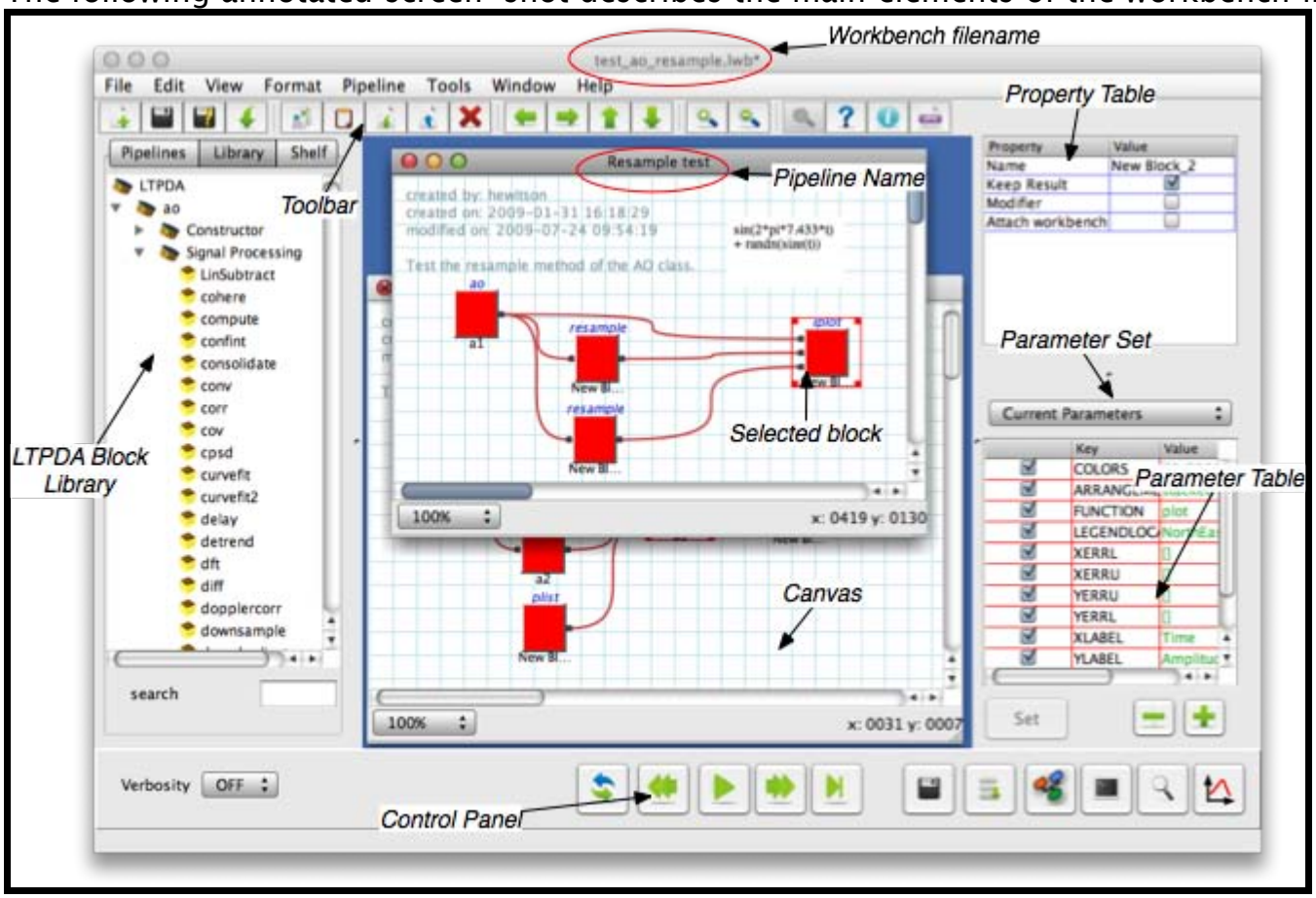
Overview

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

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

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.

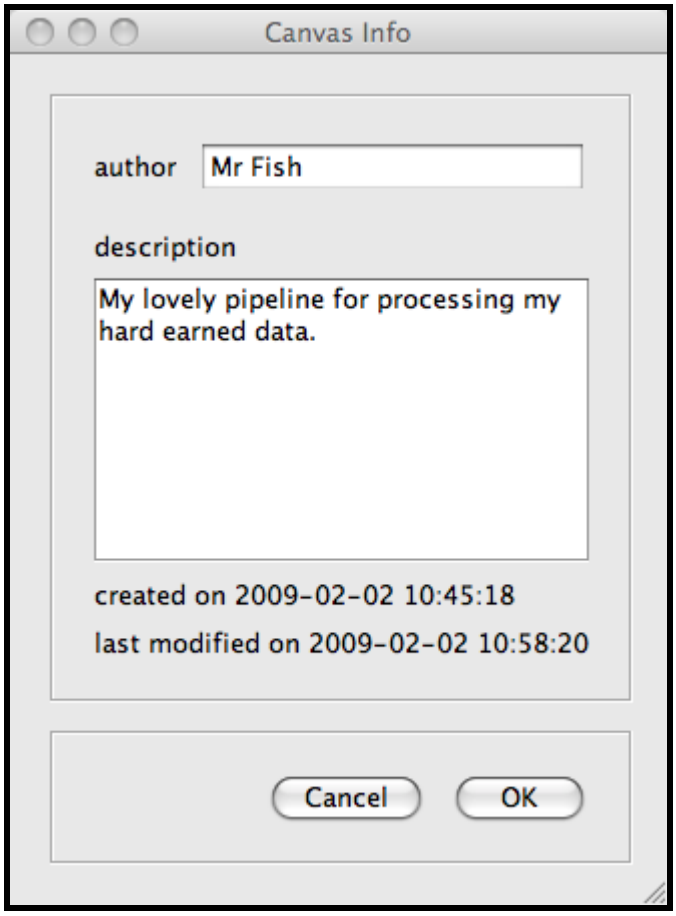
Loading the LTPDA Workbench The canvas

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.

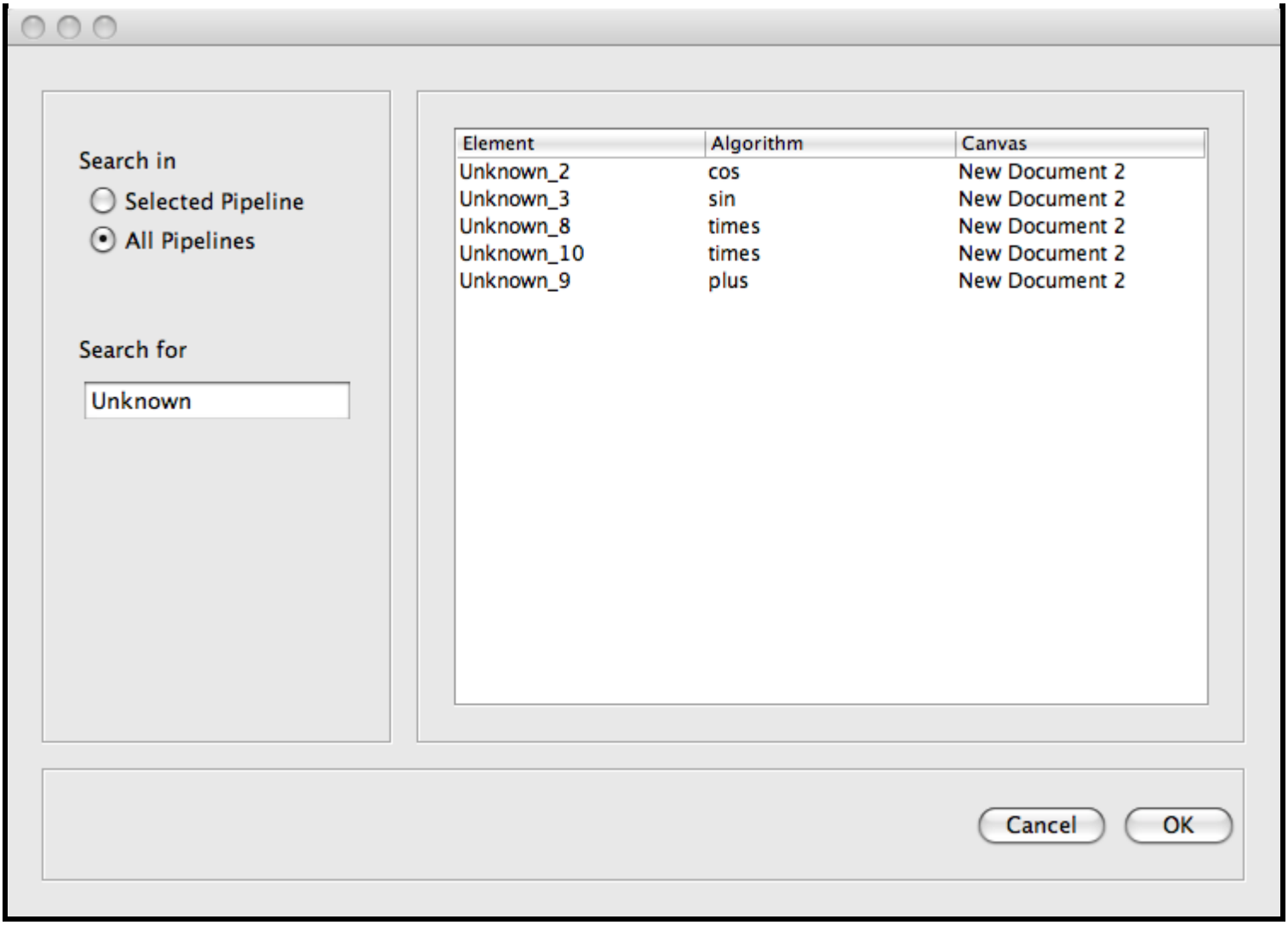


With the canvas inspector you can edit

- 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.



Mouse and keyboard actions

Building pipelines by hand

Building pipelines by hand

This section describes how to compose pipelines by hand.

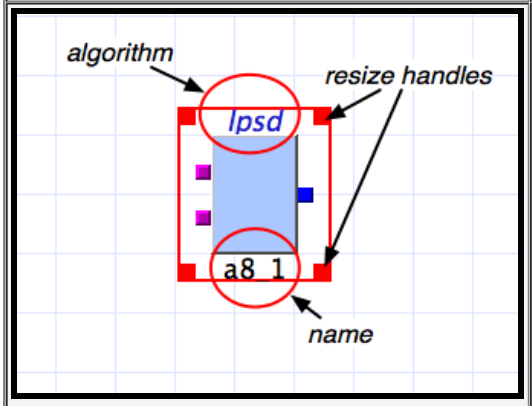
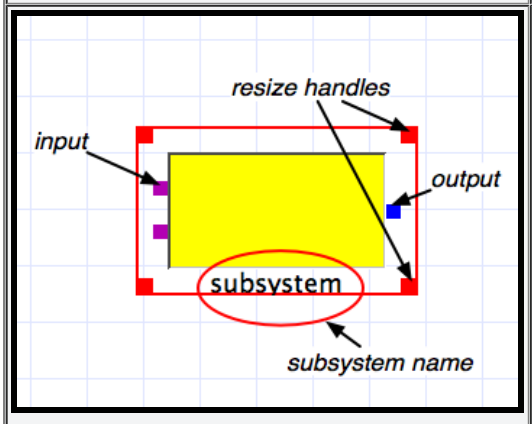
The canvas Block types

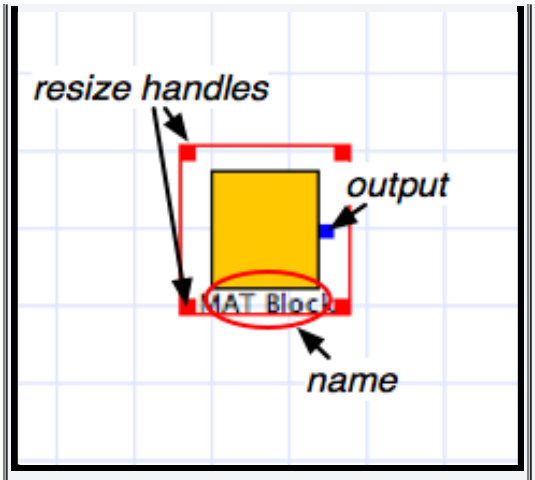
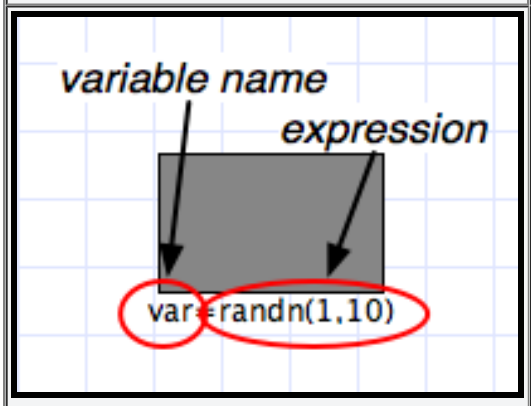
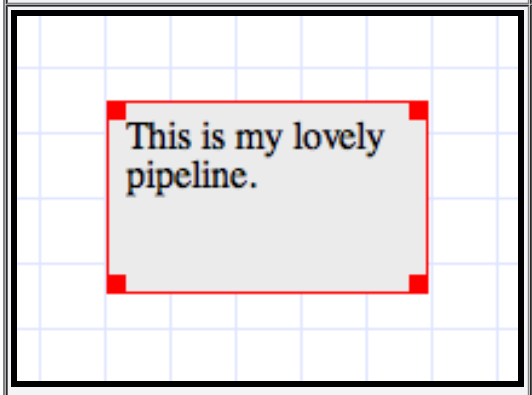
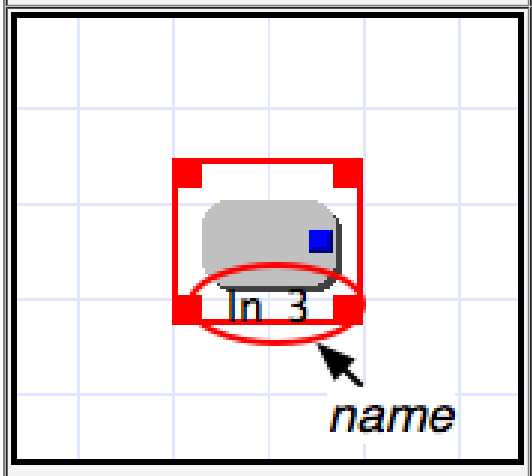
Block types

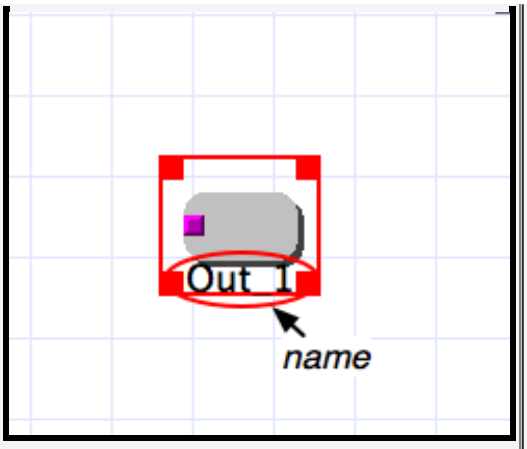
Various types of elements can be present on a pipeline. Blocks can have different states: idle, ready, or executed. These are color coded as

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 canavas.
	Output Terminal	MTerminal	A block which represents an output terminal of a subsystem. These blocks

			can only be placed on a subsystem canavas.

Building pipelines by hand Adding blocks to the canvas

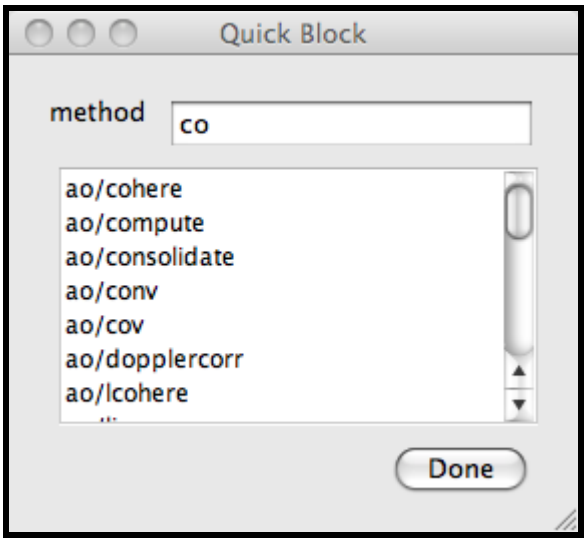
Adding blocks to the canvas

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.

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.

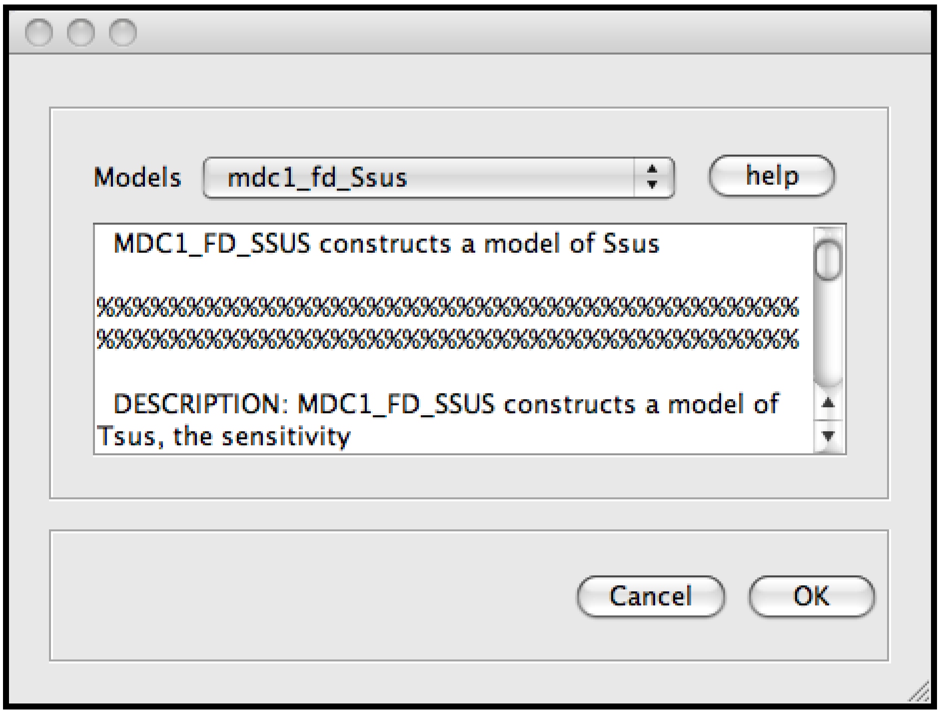
To add or remove parameters from this list use the 'plus' and 'minus' buttons.

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:

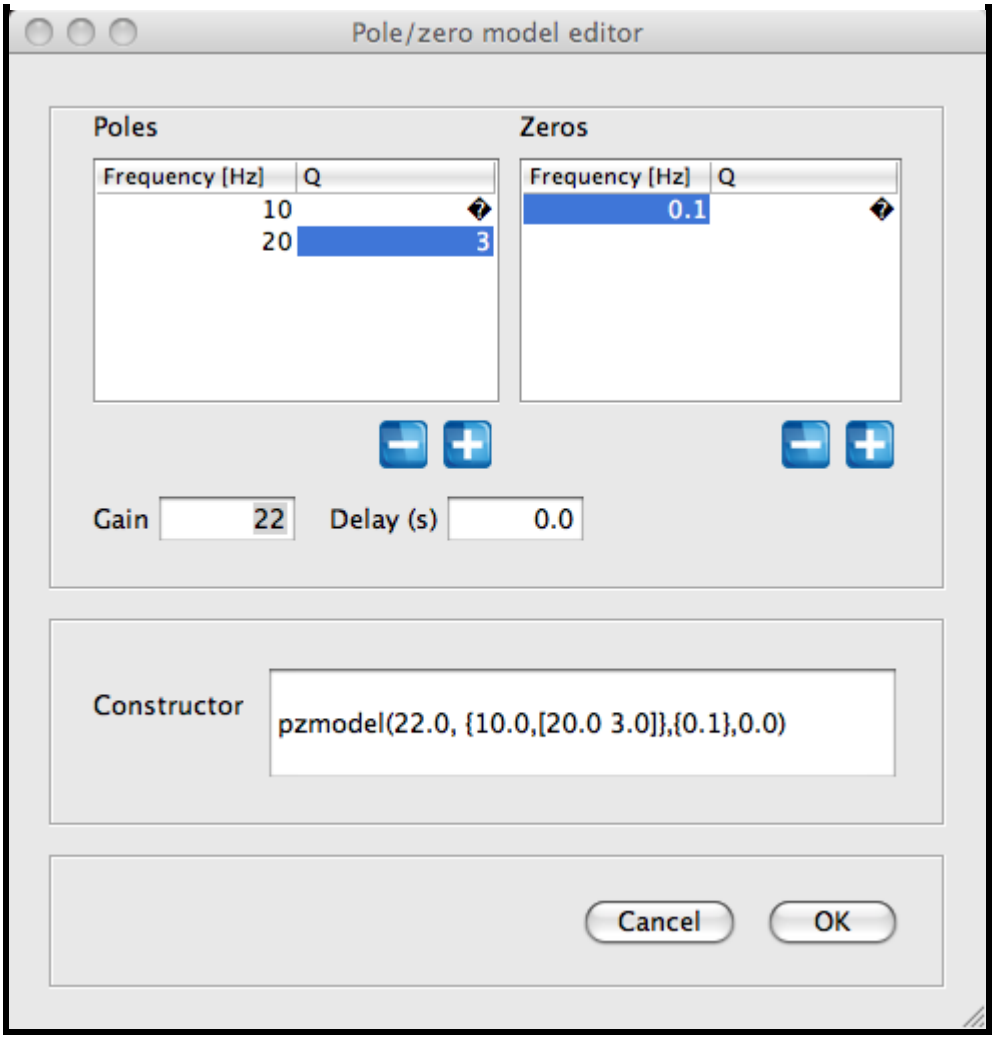
Built-in models of AO and SSM classes

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.



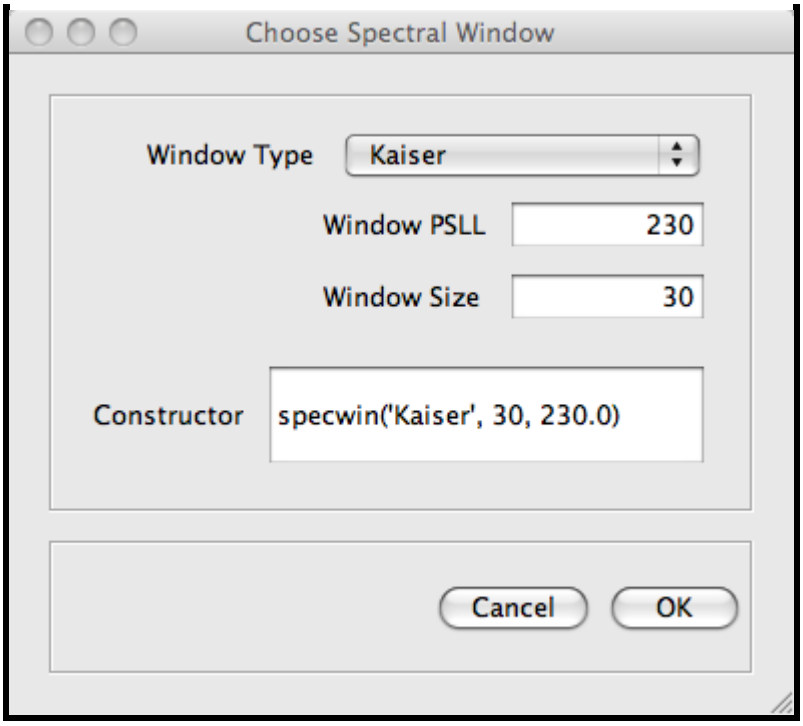
Pole/zero model editor

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 propogate.

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

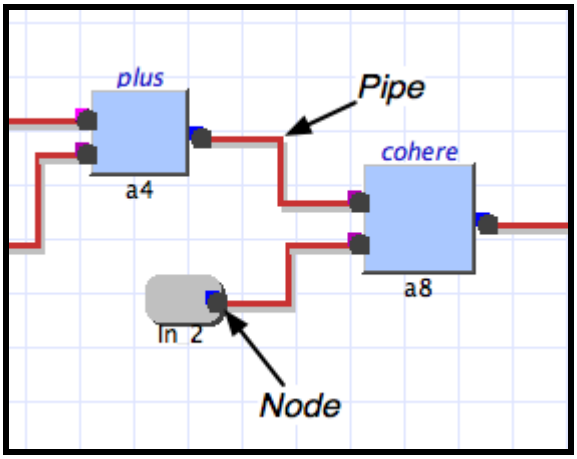
To set the text of an Annotation block, double click on the text area to start editing. Click off the block to end editing.

Adding blocks to the canvas Connecting blocks

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.

Setting block properties and parameters Creating subsystems

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

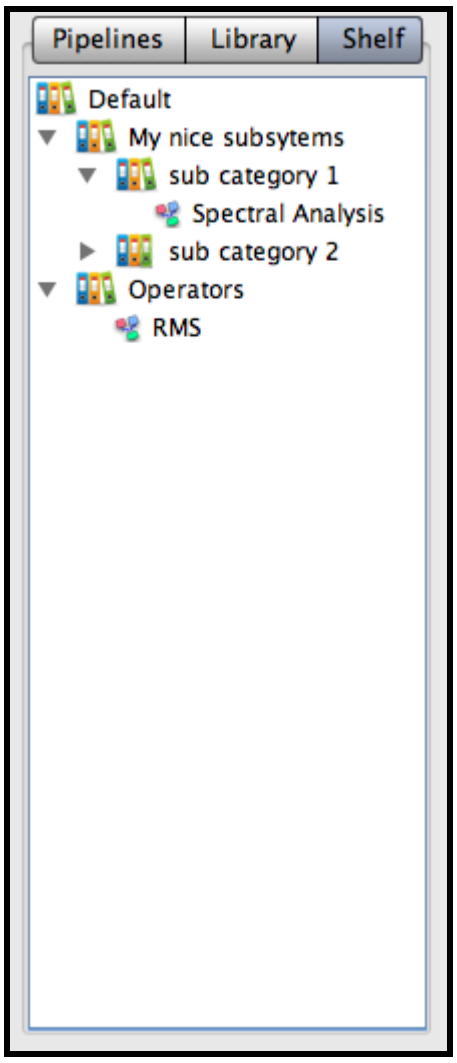
The Workbench 'Shelf' offers the user the possibility for storing subsystems for later use. Different categories and sub-categories can be created and subsystems can be added from the current pipeline to the shelf. The subsystems in the shelf can then be added to other pipelines at a later time. The 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.

Creating subsystems Accessing the shelf

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

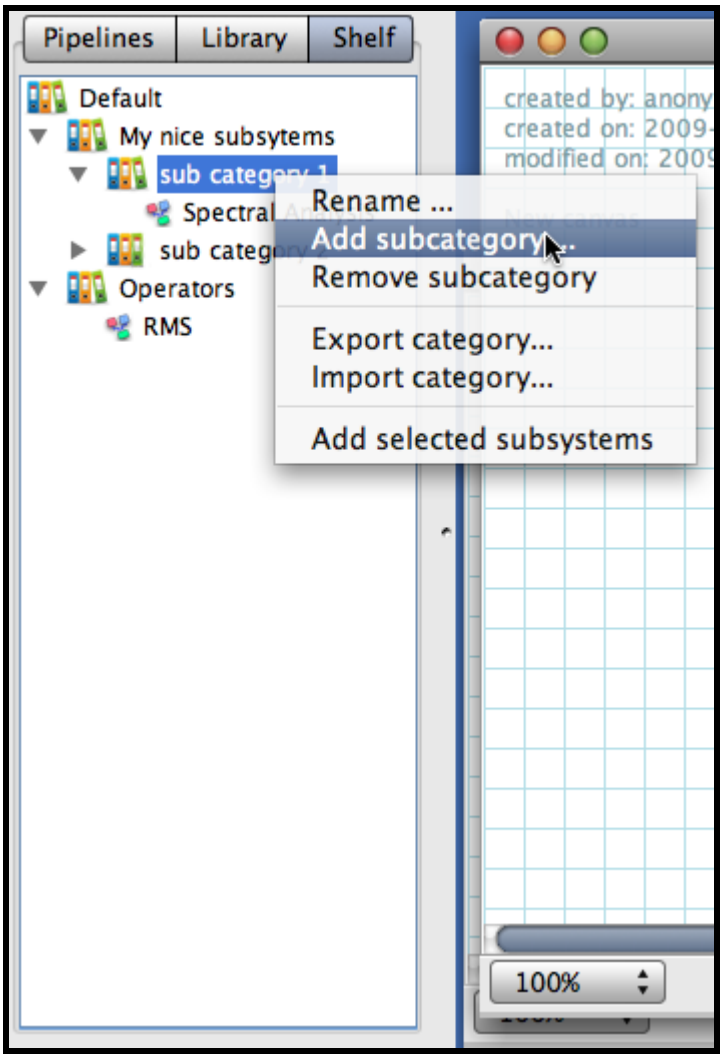
Adding and removing categories is done via the shelf context menu.

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...'.



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

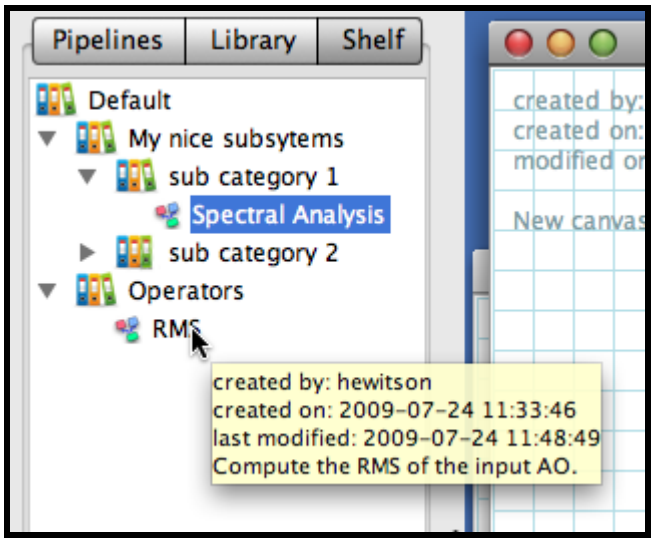
To add a particular subsystem to the shelf, do the following:

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

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.

Adding and using shelf subsystems Execution plans

Execution plans

Content needs written...

Importing and exporting shelf categories Editing the plan

Editing the plan

Content needs written...

Execution plans Linking pipelines

Linking pipelines

Content needs written...

Editing the plan Building pipelines programatically

Building pipelines programatically

Content needs written...

Linking pipelines Executing pipelines

Executing pipelines

Content needs written...

Building pipelines programatically The LTPDA Repository GUI

The LTPDA Repository GUI

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

Executing pipelines The pole/zero model helper

The pole/zero model helper

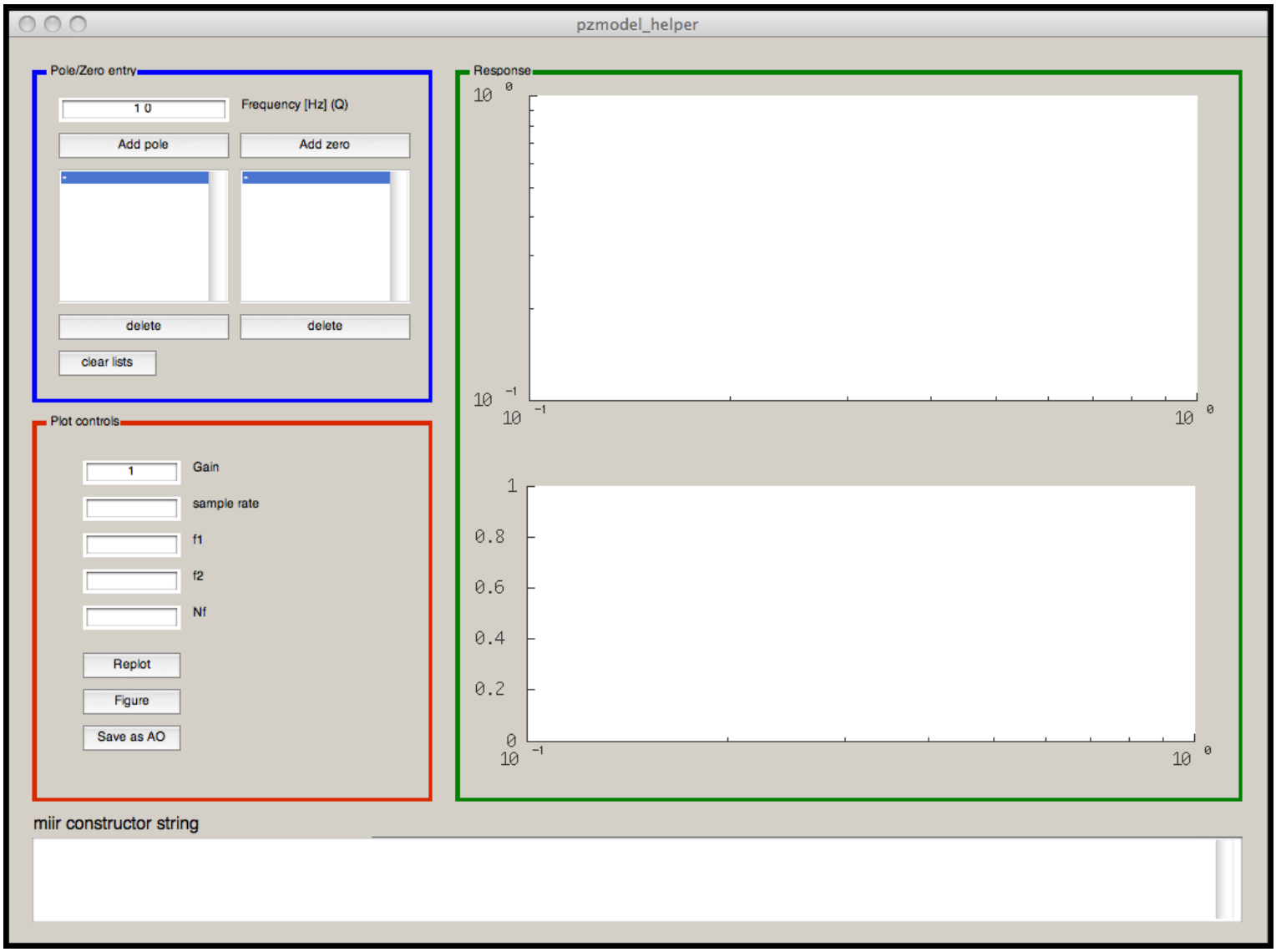
The LTPDA toolbox contains a class (`pzmodel`) for creating and using pole/zero models. The pole/zero model helper GUI allows the user to visualise the pole/zero model as it's being designed. It also allows the user to quickly see how the corresponding IIR filter (`miir` object) will look for different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

To start the pole/zero model helper:

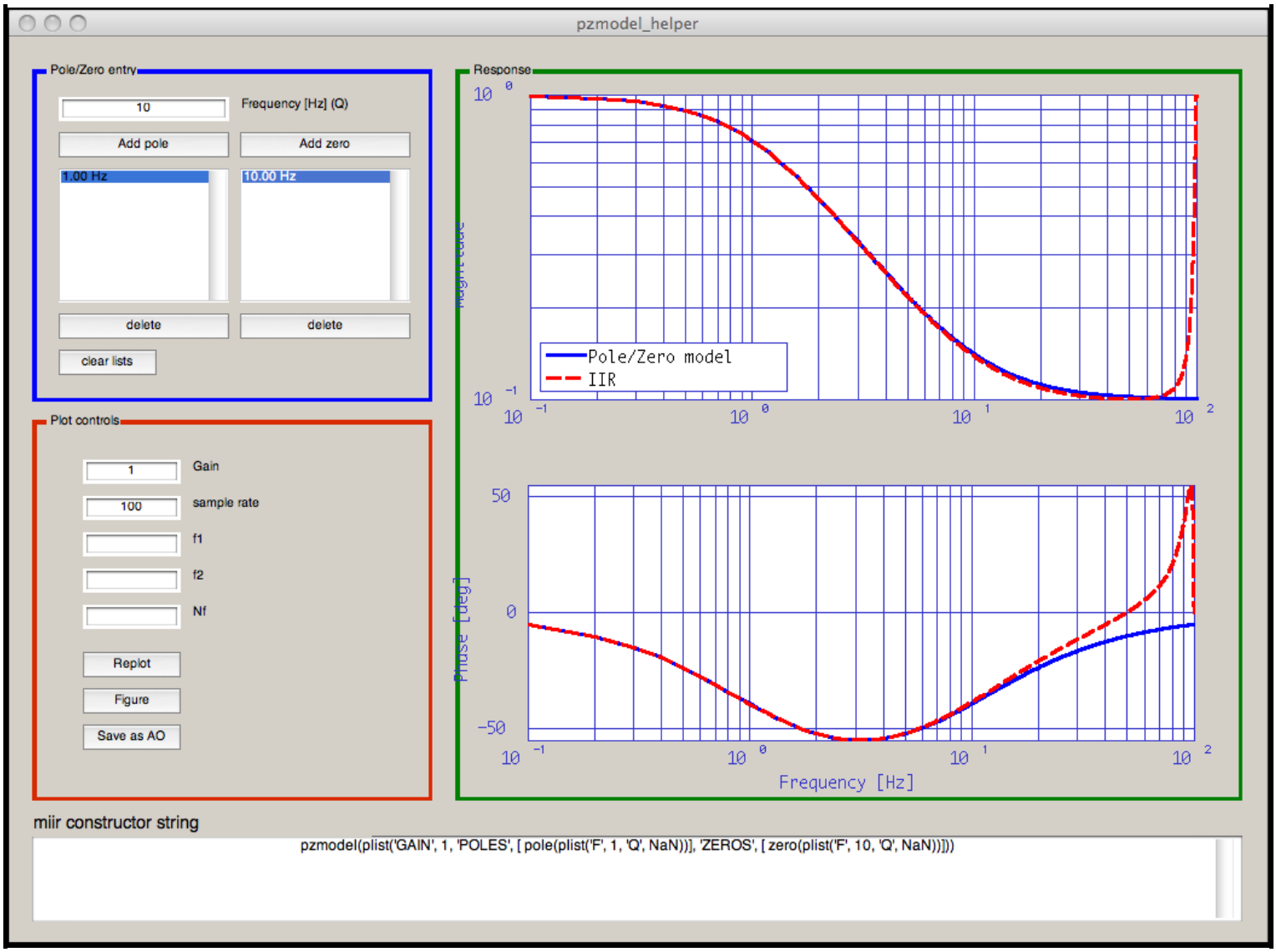
```
>> pzmodel_helper
```

or click the appropriate button on the LTPDA Launch Bay.

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



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



The LTPDA Repository GUI

The Spectral Window GUI

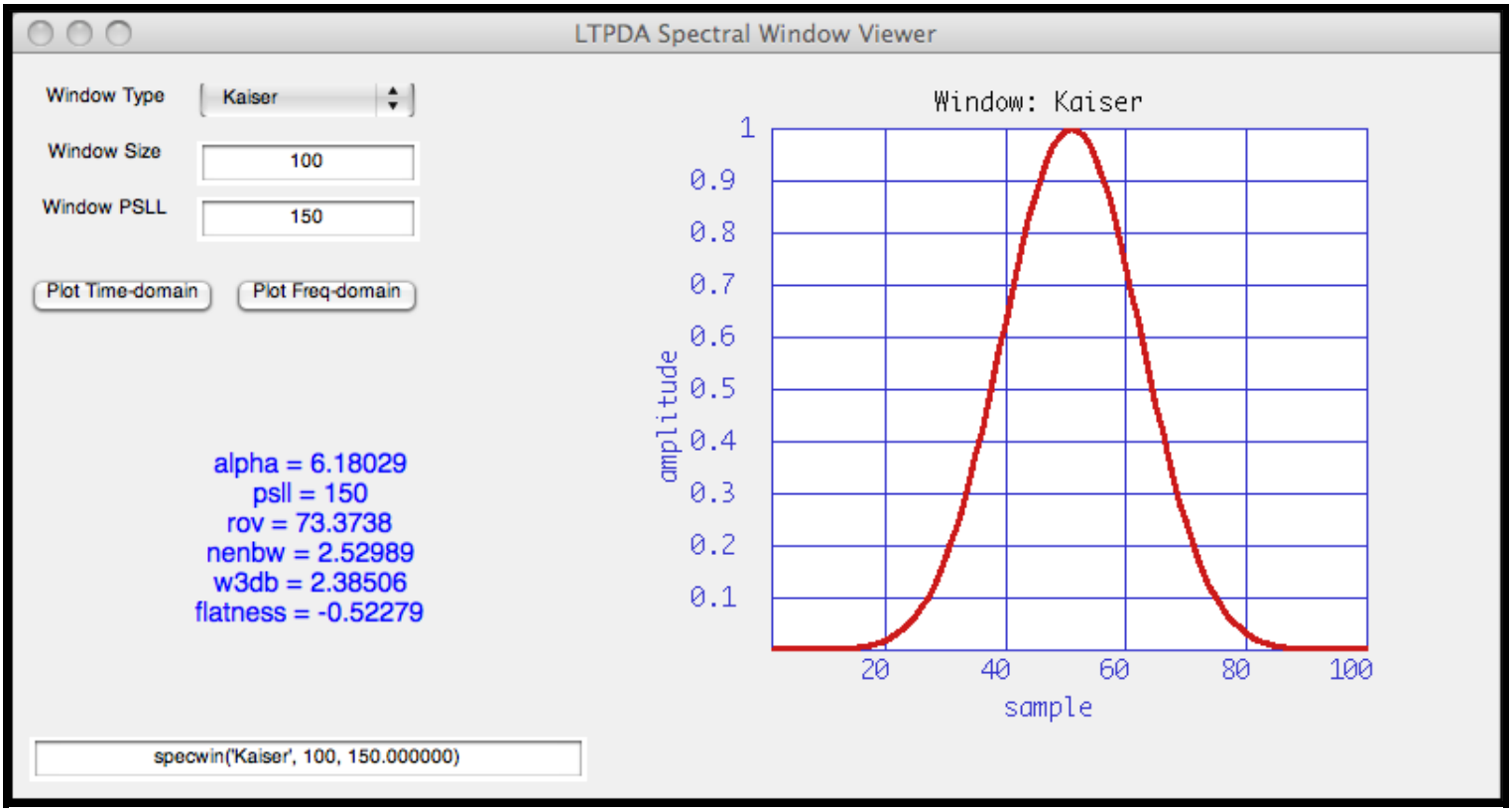
The Spectral Window GUI

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

To start the GUI:

```
>> specwin_viewer
```

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



The pole/zero model helper

The constructor helper

The constructor helper

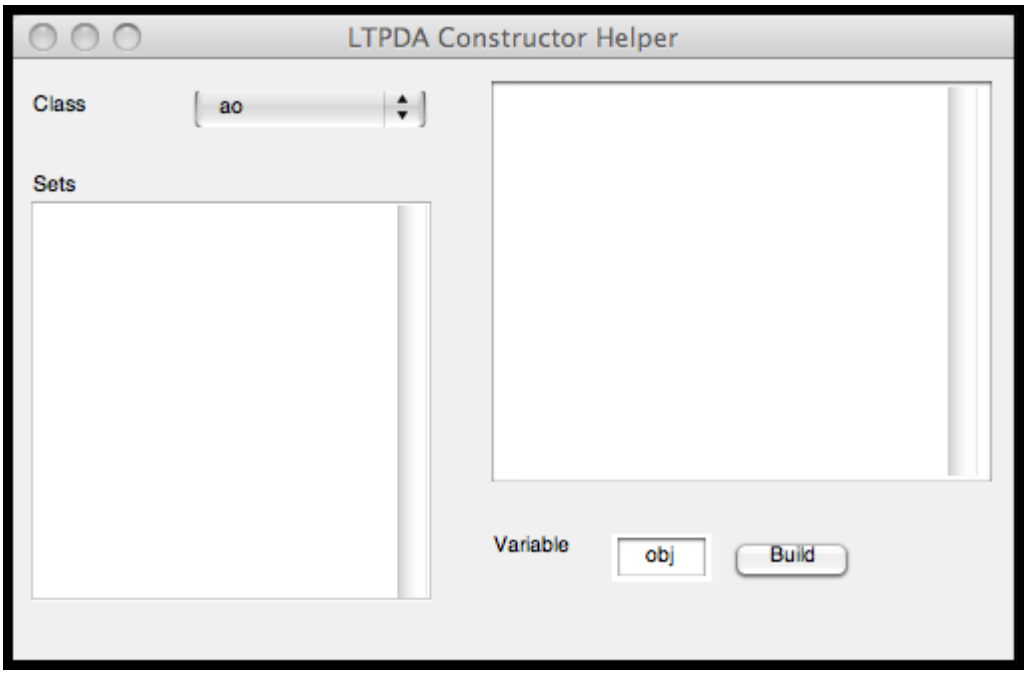
Since LTPDA is an object-oriented system, the user must create objects of different types using the appropriate constructors. The various constructor forms for each different LTPDA class can be explored using the constructor helper.

To start the constructor helper GUI:

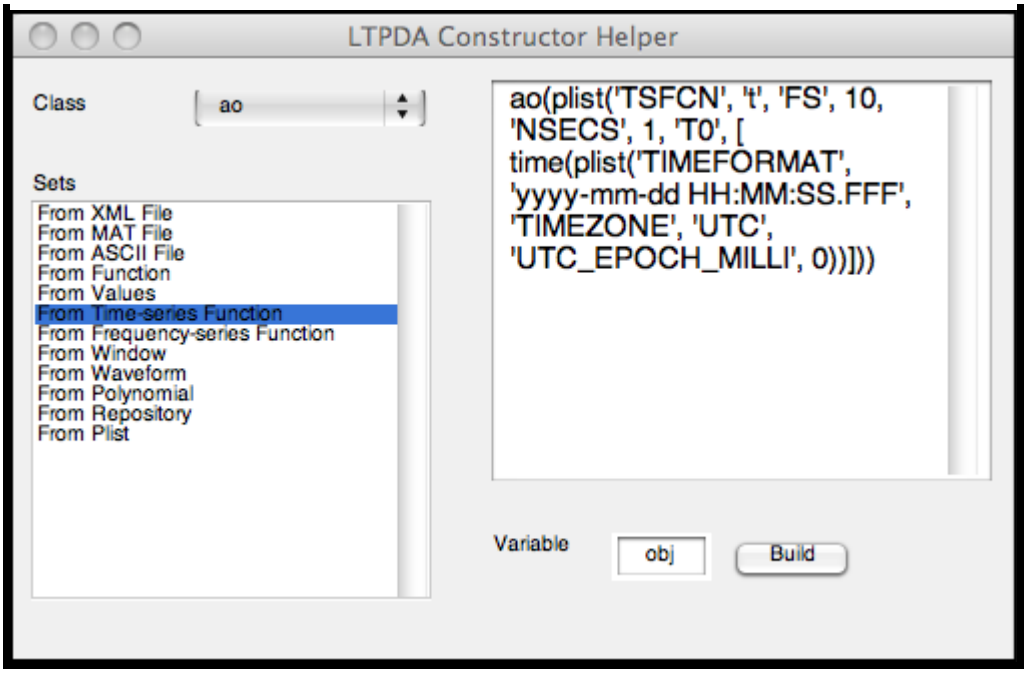
```
>> ltpda_constructor_helper
```

or click the appropriate button on the Launch Bay.

You should then be presented with the following figure:



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



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

The Spectral Window GUI | The LTPDA object explorer

The LTPDA object explorer

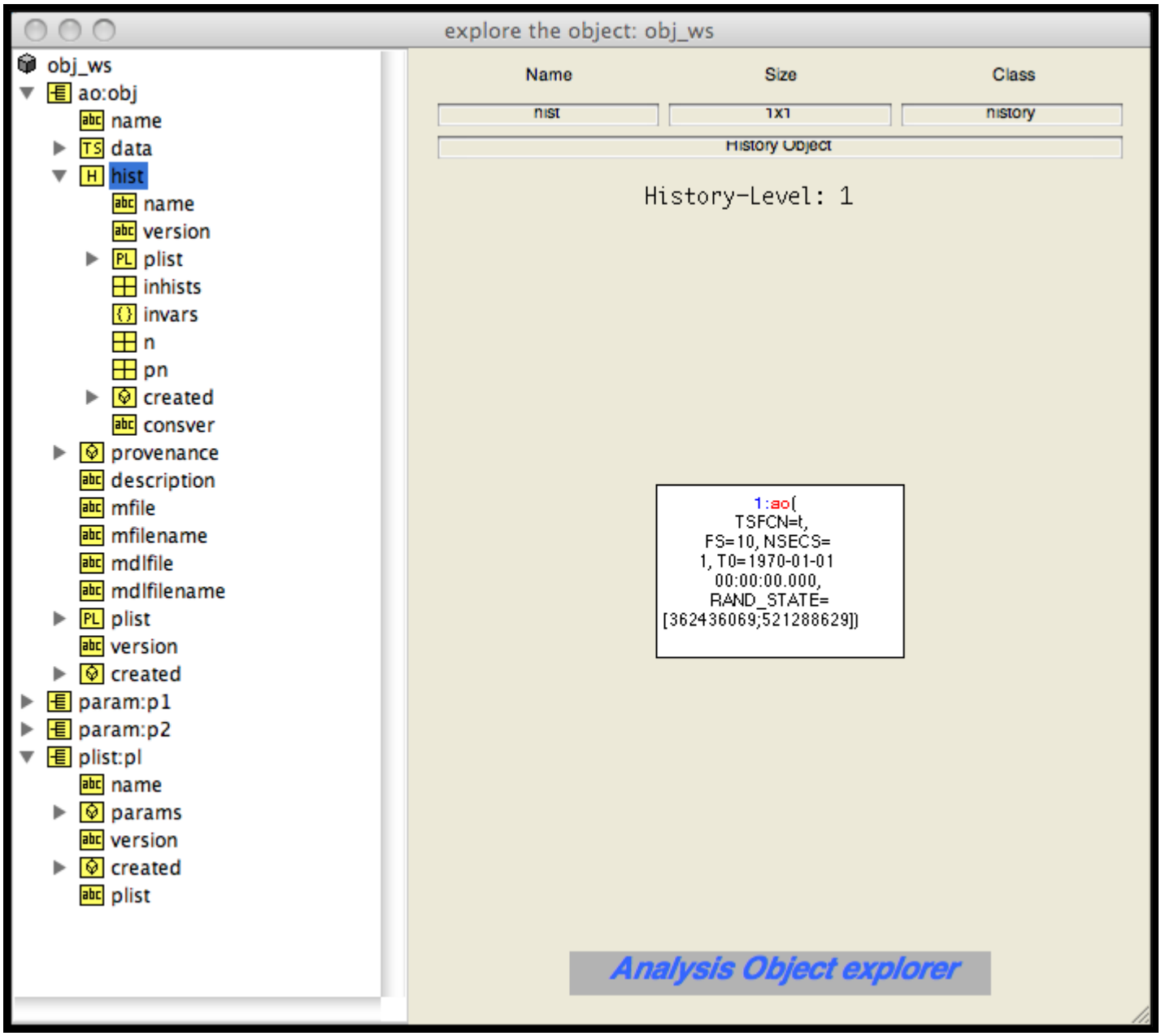
Since LTPDA works mainly with complex object types, it is often useful to explore the content of these objects graphically, particularly for Analysis Objects which may contain deep history trees. To do this, LTPDA offers the object explorer.

To start the object explorer:

```
>> explore_ao
```

or click the appropriate button on the Launch Bay.

The user is then presented with the following figure:



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

The constructor helper The quicklook GUI

The quicklook GUI

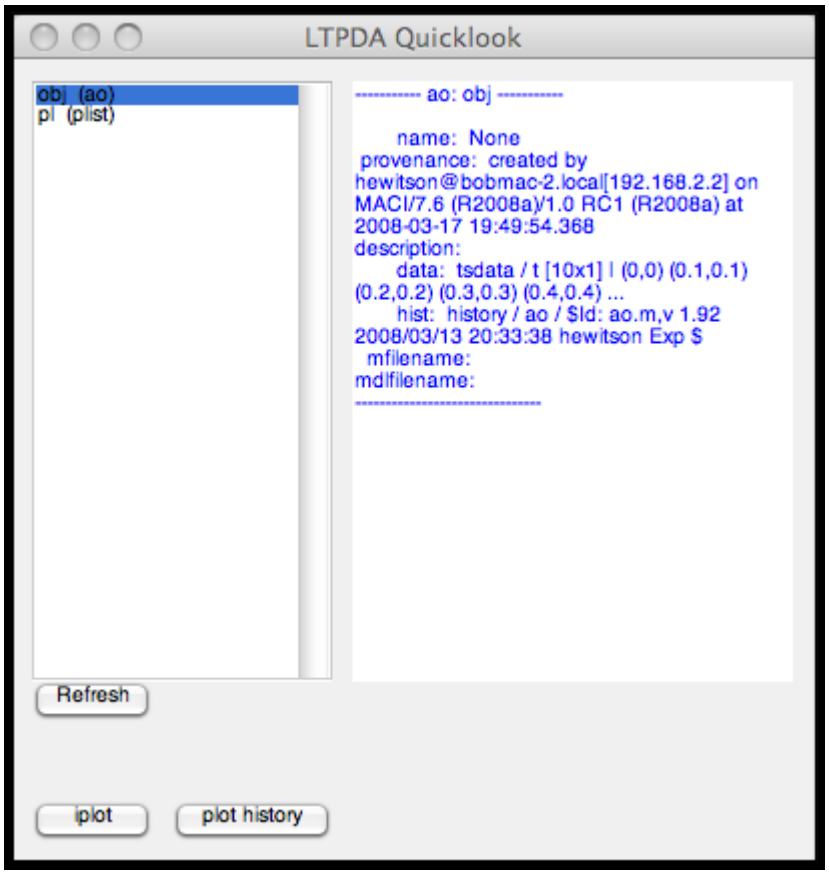
To quickly view all LTPDA objects currently in the MATLAB workspace, you can use the LTPDA Quicklook GUI.

To start the quicklook GUI:

```
>> ltpdaquicklook
```

or click the appropriate button on the Launch Bay.

The user will then be presented with the following figure:



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

Working with an LTPDA Repository

Content needs written...

The quicklook GUI What is an LTPDA 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 `ltpda_uo/retrieve` or the MATLAB LTPDA repository GUI (`repogui`).

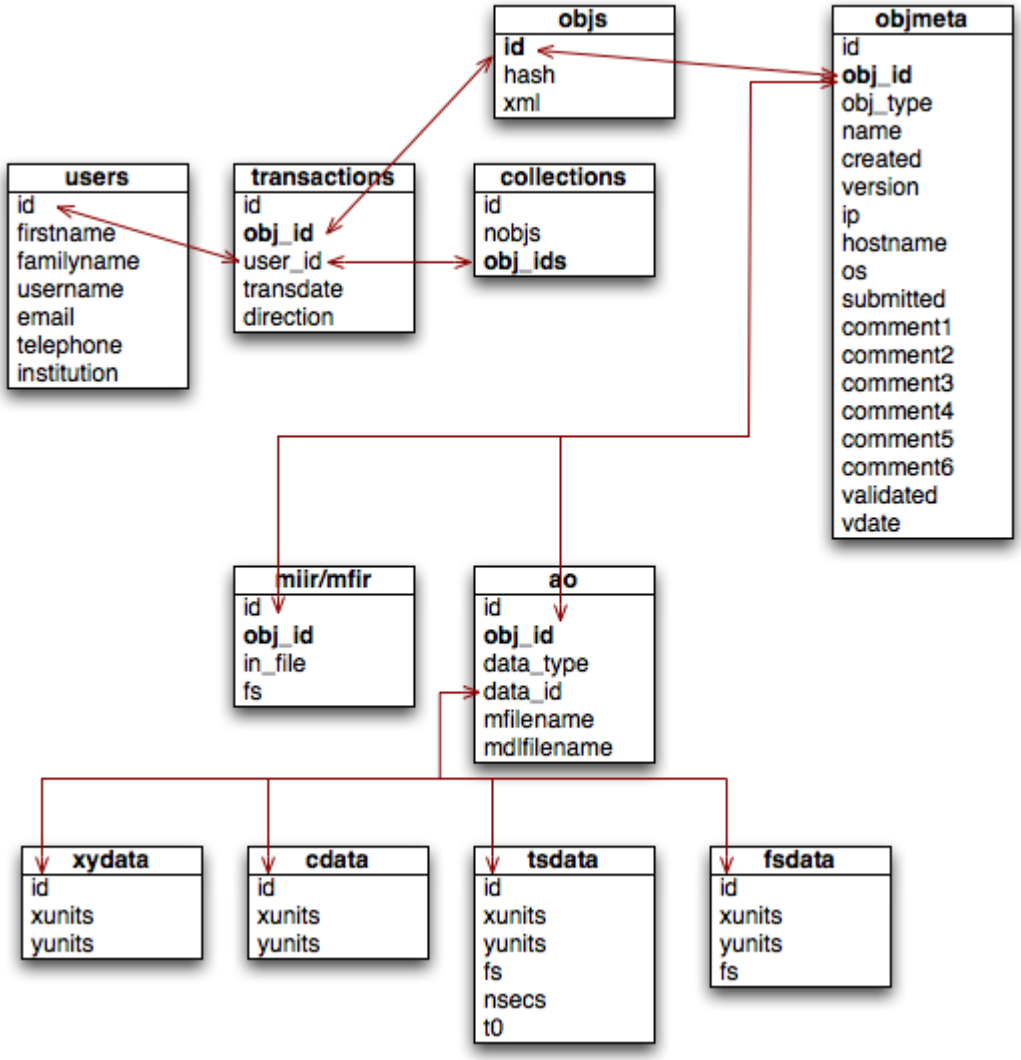
Any standard MySQL client can be used to query and search an LTPDA repository. For example, using a web-client or the standard MySQL command-line interface. In addition, the LTPDA Toolbox provides two ways to search the database: using the command `utils.mysql.dbquery` or using the LTPDA repository GUI (`repogui`). 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 indended 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.

Working with an LTPDA Repository Connecting to an LTPDA Repository

Connecting to an LTPDA Repository

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

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

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

```
>> conn = utils.mysql.connect('130.75.117.67', 'ltpda_test')
       ** Connecting to 130.75.117.67 as john.doe...
       ** Connection status:
                DatabaseProductName: 'MySQL'
             DatabaseProductVersion: '5.0.45'
                     JDBCDriverName: 'MySQL-AB JDBC Driver'
                  JDBCDriverVersion: [1x103 char]
             MaxDatabaseConnections: 0
                    CurrentUserName: 'john.doe@130.75.117.67'
                        DatabaseURL: 'jdbc:mysql://130.75.117.67/ltpda_test'
             AutoCommitTransactions: 'True'
```

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

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

What is an LTPDA Repository | Submitting LTPDA objects to a repository

Submitting LTPDA objects to a repository

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

- ao
- miir
- mfir
- filterbank
- pzmodel
- timespan
- ssm
- parfrac
- rational
- smodel
- matrix
- plist

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'

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. Tries to create a binary representation of the object. If this is possible and it is possible to store for this object type a binary, then insert the binary
   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 to submit
`sinfo`	An information structure (see below)

The information structure should have the following fields:

```
      'conn'                    - database connection object
      'experiment_title'        - a title for the submission (Mandatory, >4 characters)
      'experiment_description'  - a description of this submission (Mandatory, >10 characters)
      'analysis_description'    - a description of the analysis performed  (Mandatory, >10
characters));
      'quantity'                - the physical quantity represented by the data);
      'keywords'                - a comma-delimited list of keywords);
      'reference_ids'           - a string containing any reference object id numbers
      'additional_comments'     - any additional comments
      'additional_authors'      - any additional author names
```

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

```
      % Connect to a repository
      conn = utils.mysql.connect('130.75.117.67', 'ltpda_test');

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

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

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

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

The ID assigned to the submitted object is contained in the first output of the `submit` function:

```
      % Inspect the object ID
      disp(ids)
      212
```

Submitting collections

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

```
% Connect to a repository
conn = utils.mysql.connect('130.75.117.67', 'ltpda_test');

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

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

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

% Close connection
close(conn);

% END
```

If the verbosity level you set up on your LTPDA Toolbox preferences GUI is at lest 'PROC1', running this script yields an outut similar to the following:

```
** Connecting to 130.75.117.67 as john.doe...
** Connection status:
DatabaseProductName: 'MySQL'
DatabaseProductVersion: '5.0.45'
JDBCDriverName: 'MySQL-AB JDBC Driver'
JDBCDriverVersion: [1x56 char]
MaxDatabaseConnections: 0
CurrentUserName: 'hewitson@pixfirewall.aei.uni-hannover.de'
DatabaseURL: 'jdbc:mysql://130.75.117.67/ltpda_test'
AutoCommitTransactions: 'True'

M: running ao/ao
M:   constructing from plist
M: running ltpda_uo/submit
M:   sinfo structure is valid.
M:   submitting objects to repository.
M:   got user id 1 for user: john.doe
M:   submitting object: ao / sine wave
M: running ltpda_uoh/created
M: running ltpda_uoh/created
M:   uploading XML data...
M:  done.
M:   submitted object ao with id 234
M:   running query: insert into bobjs (obj_id, mat) values (?,?)
M:   running query INSERT INTO objmeta SET obj_id=234,obj_type='ao',name='sine
wave',created='2009-07-30 04:02:28',version='$Id: repo_submit_content.html,v 1.9 2009/08/03
12:29:51 mauro Exp $',ip='193.205.193.171',hostname='mauro-huellers-
macbook.local',os='MACI',submitted='2009-07-30 06:02:39',experiment_title='submit multiple
objects',experiment_desc='this is just a test of the whole
thing',reference_ids='',additional_comments='none',additional_authors='no
one',keywords='',quantity='',analysis_desc='no analysis this time';
M:   made meta-data entry
M:   running query INSERT INTO tsdata SET xunits=' [s] ',yunits=' []
',fs=100,nsecs=10,t0='1970-01-01 00:00:00';
M:   running query INSERT INTO ao SET
obj_id=234,data_type='tsdata',data_id=70,description='',mfilename='',mdlfilename='';
M:   running query INSERT INTO transactions SET obj_id=234,user_id=1,transdate='2009-07-
30 04:02:28',direction='in';
M:   updated transactions table
M:   submitting object: pzmodel / None
M: running ltpda_uoh/created
```

```
    M: running ltpda_uoh/created
    M:   uploading XML data...
    M:  done.
    M:   submitted object pzmodel with id 235
    M:   running query: insert into bobjs (obj_id, mat) values (?,?)
    M:   running query INSERT INTO objmeta SET
obj_id=235,obj_type='pzmodel',name='None',created='2009-07-30 04:02:28',version='$Id:
repo_submit_content.html,v 1.9 2009/08/03 12:29:51 mauro Exp
$',ip='193.205.193.171',hostname='mauro-huellers-macbook.local',os='MACI',submitted='2009-07-
30 06:02:42',experiment_title='submit multiple objects',experiment_desc='this is just a test
of the whole thing',reference_ids='',additional_comments='none',additional_authors='no
one',keywords='',quantity='',analysis_desc='no analysis this time';
    M:   made meta-data entry
    M:   running query INSERT INTO transactions SET obj_id=235,user_id=1,transdate='2009-07-
30 04:02:28',direction='in';
    M:   updated transactions table
    M:   submitting object: plist / none
    M:   uploading XML data...
    M:  done.
    M:   submitted object plist with id 236
    M:   running query: insert into bobjs (obj_id, mat) values (?,?)
    M:   running query INSERT INTO objmeta SET
obj_id=236,obj_type='plist',name='none',created='2009-07-30 04:02:28',version='$Id:
repo_submit_content.html,v 1.9 2009/08/03 12:29:51 mauro Exp
$',ip='193.205.193.171',hostname='mauro-huellers-macbook.local',os='MACI',submitted='2009-07-
30 06:02:43',experiment_title='submit multiple objects',experiment_desc='this is just a test
of the whole thing',reference_ids='',additional_comments='none',additional_authors='no
one',keywords='',quantity='',analysis_desc='no analysis this time';
    M:   made meta-data entry
    M:   running query INSERT INTO transactions SET obj_id=236,user_id=1,transdate='2009-07-
30 04:02:28',direction='in';
    M:   updated transactions table
    M:   running query INSERT INTO collections SET nobjs=3,obj_ids='234,235,236';
    M:   made collection entry
    M:   submission complete.
```

Connecting to an LTPDA Repository Exploring an LTPDA Repository

Exploring an LTPDA Repository

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

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

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

Examples of usage are:

Searching particular tables

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

Retrieving a list of tables

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

```
      >> info  = utils.mysql.dbquery(conn)
```

High-level queries

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

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

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

`ltpda_getAOsInTimeSpan`	Retrieve particular AOs in the given time-span

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 the function `ltpda_uo.retrieve` to retrieve objects. In additon, the constructors of each user class can be used to retrieve objects of that class.

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. The following script shows an example of retrieving a single object:

```
% Connect to a repository
[conn, username] = utils.mysql.connect('130.75.117.67', 'ltpda_test');

% Retrieve the object
q = ltpda_uo.retrieve(conn, 12);

% Close connection
close(conn);
```

Note that the retrieved object belongs, in general, to the class `ltpda_uo`, because the `retrieve` function is a method of that superclass.

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 = '130.75.117.67';
database = 'ltpda_test';

% Retrieve the object
q = ao(plist('hostname', hostname, 'database', dbname, 'ID', 12));
```

If you know the collection that contains the object, and the class, then you can directly call the class constructor method:

```
% Retrieve the object
```

```
q = ao(plist('hostname', hostname, 'database', dbname, 'CID', 2));
```

In this case, all AOs in the collection with CID 2 will be retrieved and stored in `q`.

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

```
q = ltpda_uo.retrieve(conn, 1,2,3);
```

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

Retrieving object collections

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

```
% Connect to a repository
[conn, username] = utils.mysql.connect('130.75.117.67', 'ltpda_test');

% Retrieve the collection
q = ltpda_uo.retrieve(conn, 'Collection', 1);

% Close connection
close(conn);
```

The output is a cell array 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 on the database.

```
% Connect to a repository
[conn, username] = utils.mysql.connect('130.75.117.67', 'ltpda_test');

% Retrieve the collection
q = ao(plist('hostname', hostname, 'database', dbname, 'ID', 12, 'binary', 'yes'));

% Close connection
close(conn);
```

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

Using the LTPDA Repository GUI

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

Starting the LTPDA Repository GUI

The GUI can be started using the command

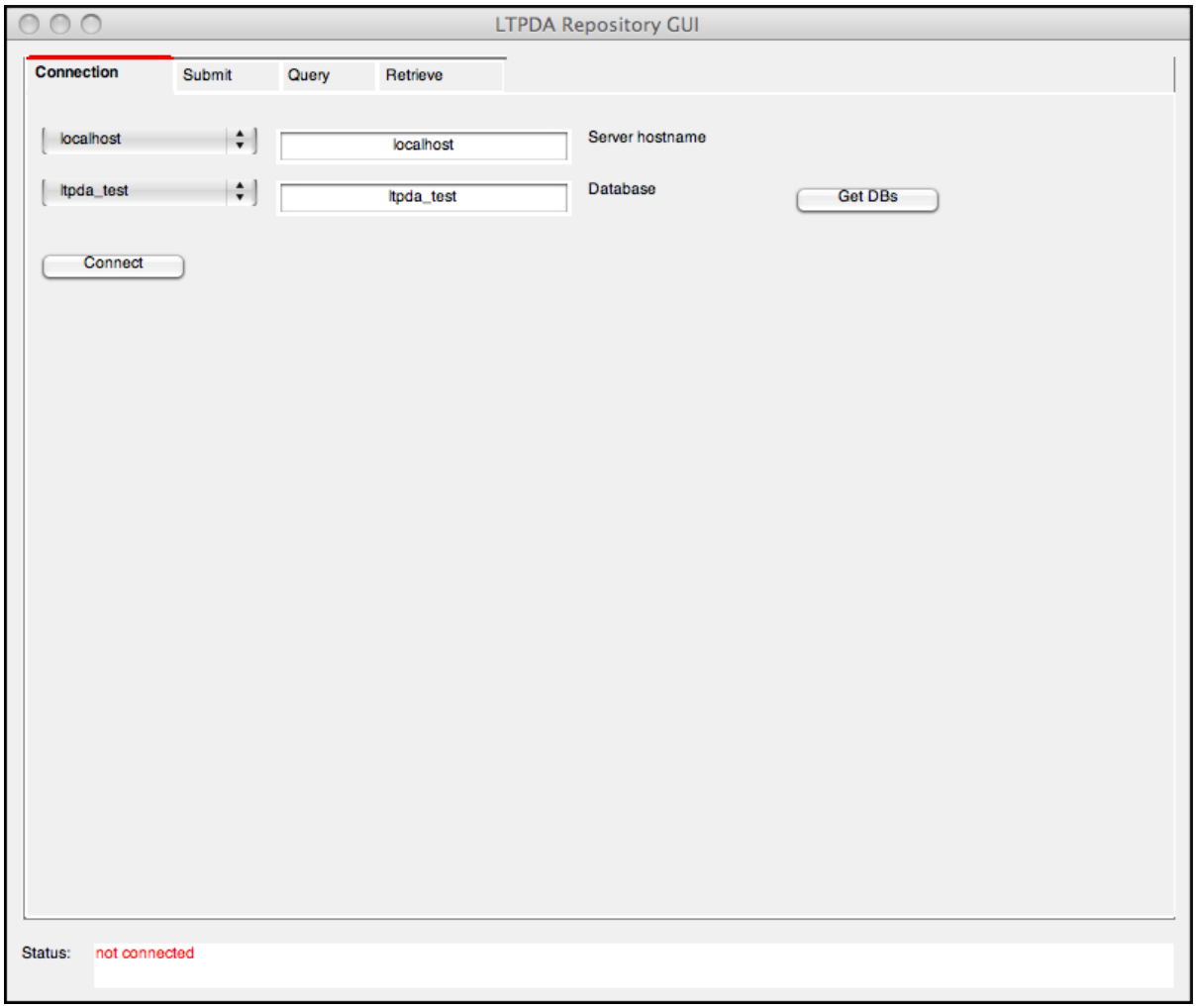
```
>> repogui
```

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

Retrieving LTPDA objects from a repository — Connecting to a repository

Connecting to a repository

The first tab pane is the connection panel.



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

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

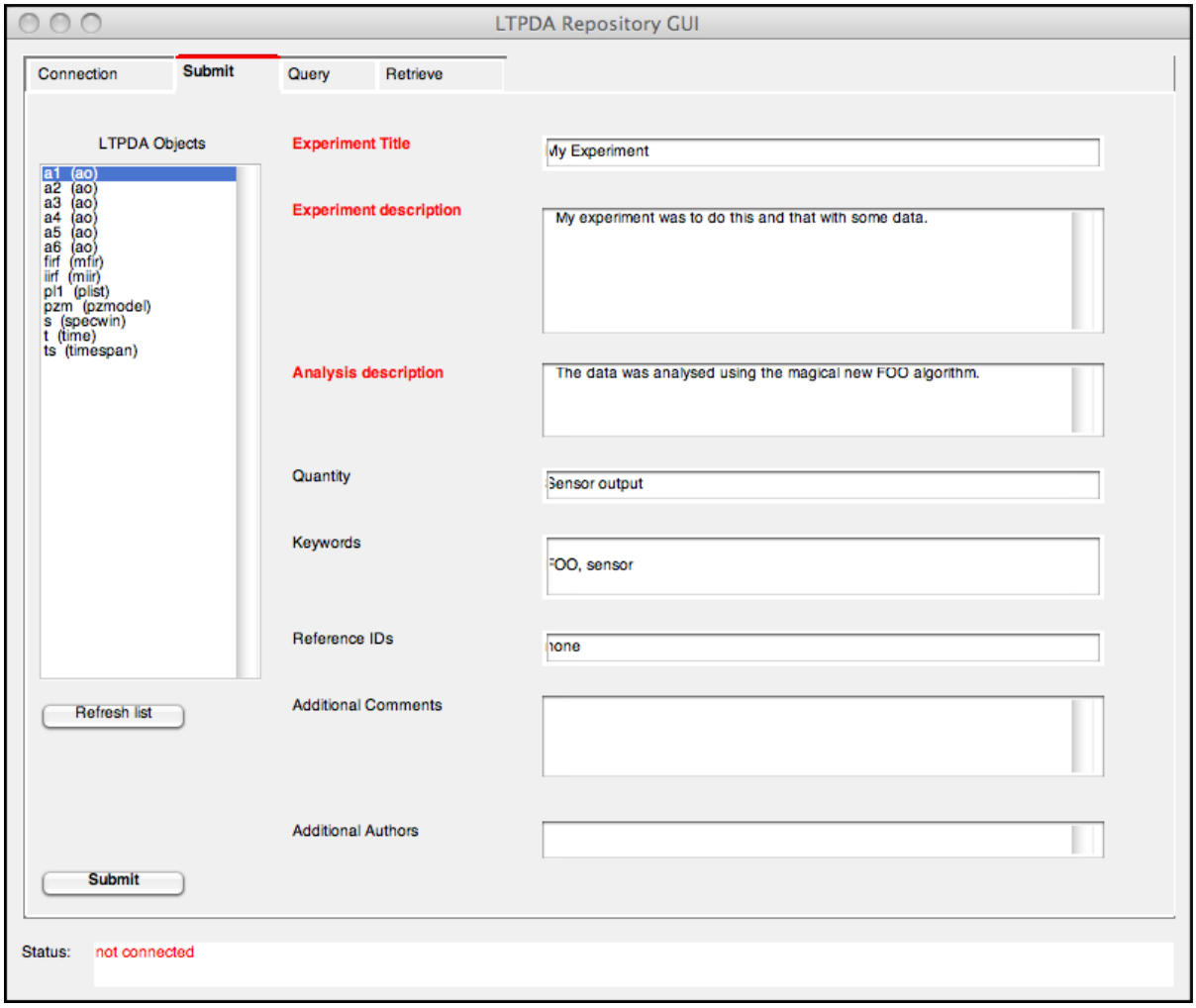
Disconnecting from a repository

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

Using the LTPDA Repository GUI Submitting objects to a repository

Submitting objects to a repository

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



Selecting the objects to submit

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

Completing the submission form

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

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

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

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

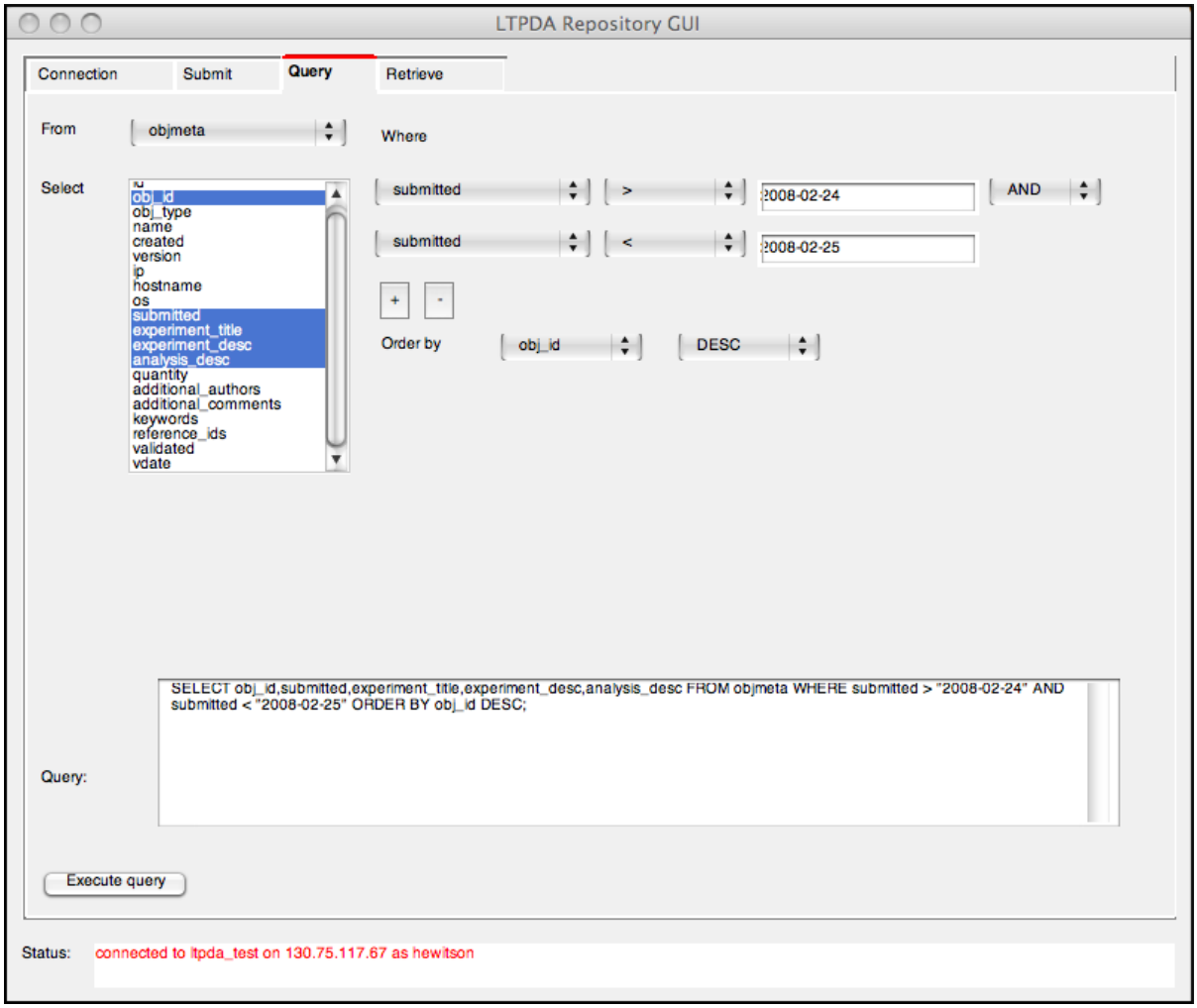
After inserting the useful information by filling the corresponding entries, the user can proceed with the submission by clicking the 'Submit' button. The Matlab window will show the response from the repository, including the IDs assigned to the submitted objects.

Connecting to a repository Querying the contents of a repository

Querying the contents of a repository

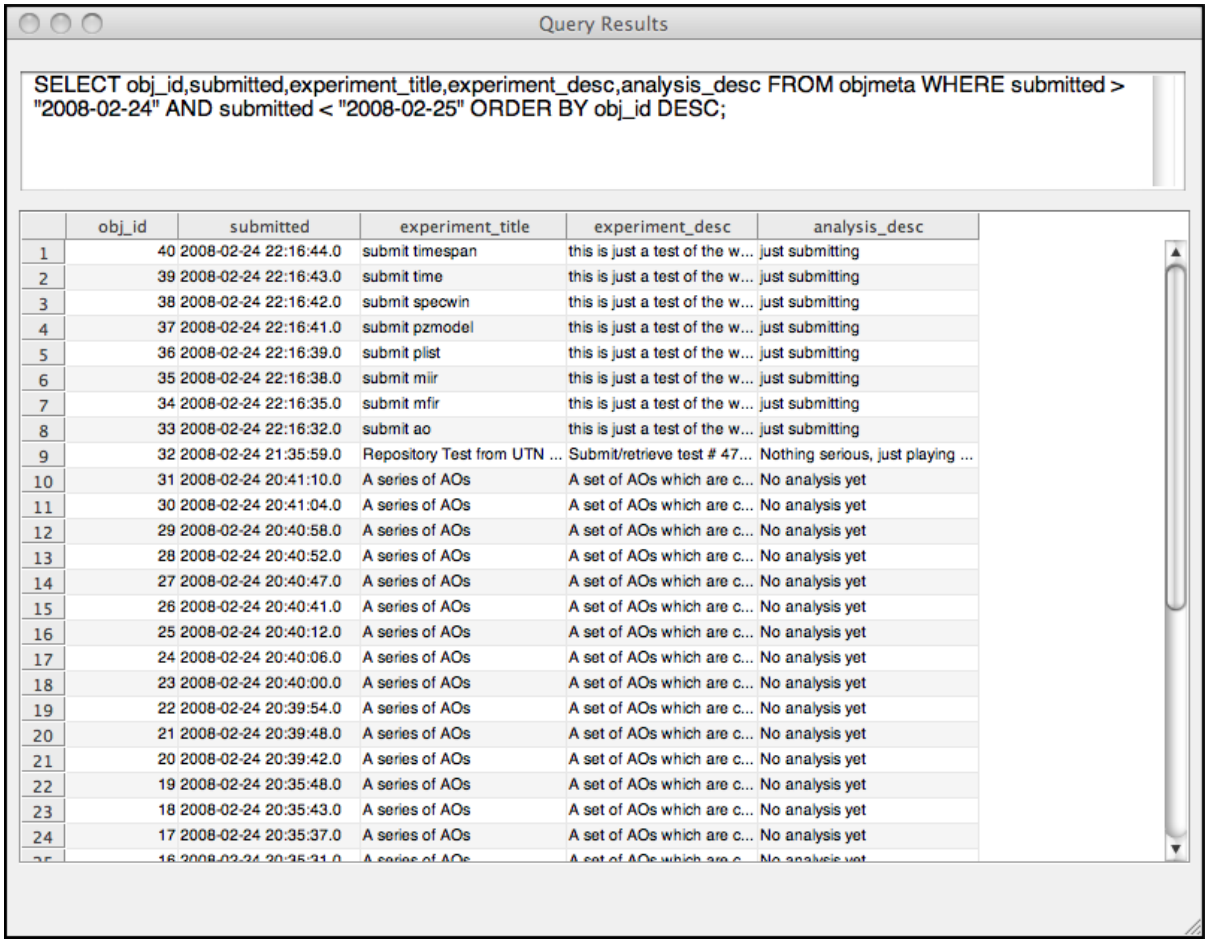
Querying an LTPDA repository is done using standard SQL statements. The repository GUI presents the user with the possibility to graphically build SQL statements which avoids learning SQL syntax. Currently, the SQL statements that can be built in this way are restricted to queries on a single database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

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



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

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



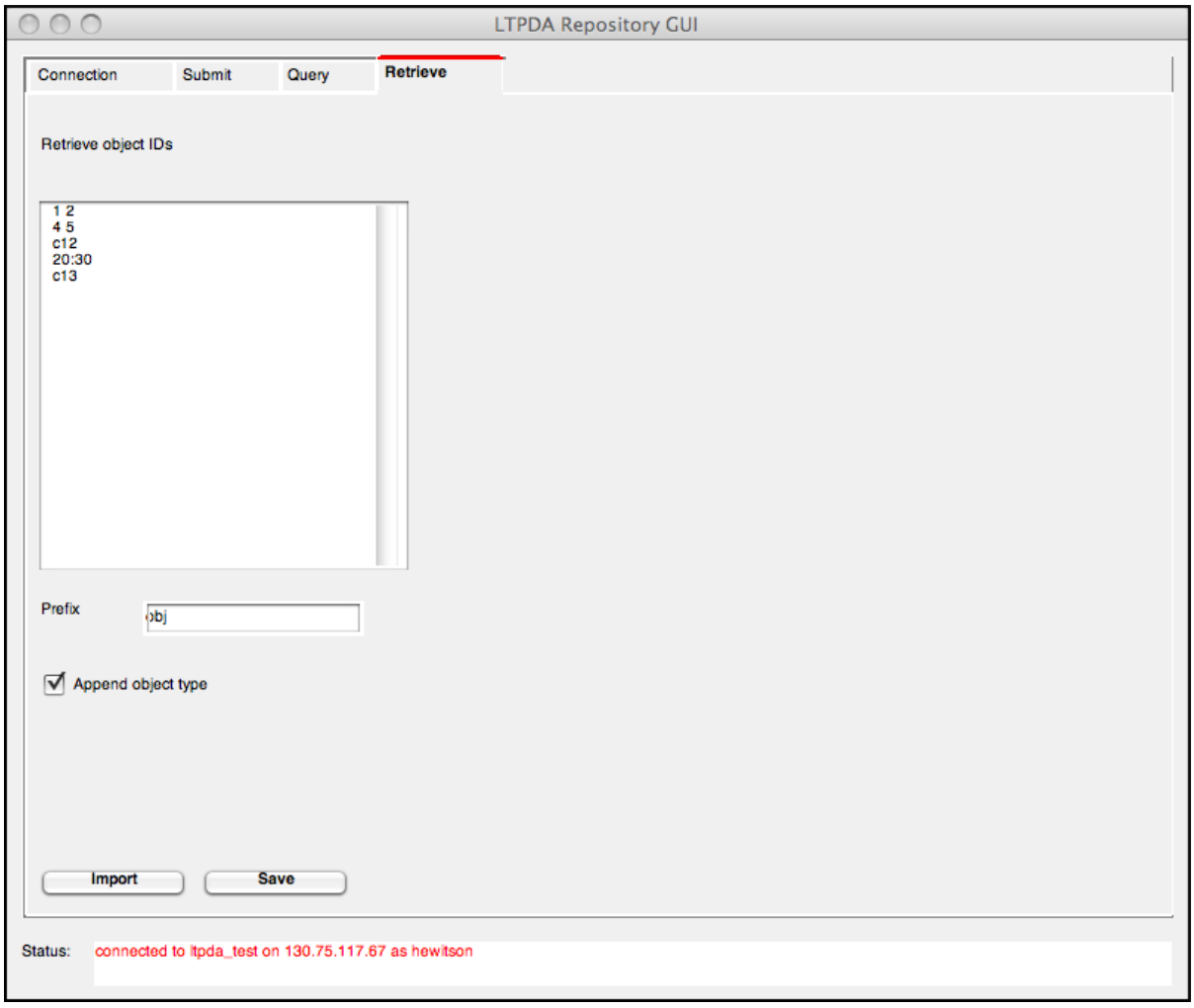
	obj_id	submitted	experiment_title	experiment_desc	analysis_desc
1	40	2008-02-24 22:16:44.0	submit timespan	this is just a test of the w...	just submitting
2	39	2008-02-24 22:16:43.0	submit time	this is just a test of the w...	just submitting
3	38	2008-02-24 22:16:42.0	submit specwin	this is just a test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

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

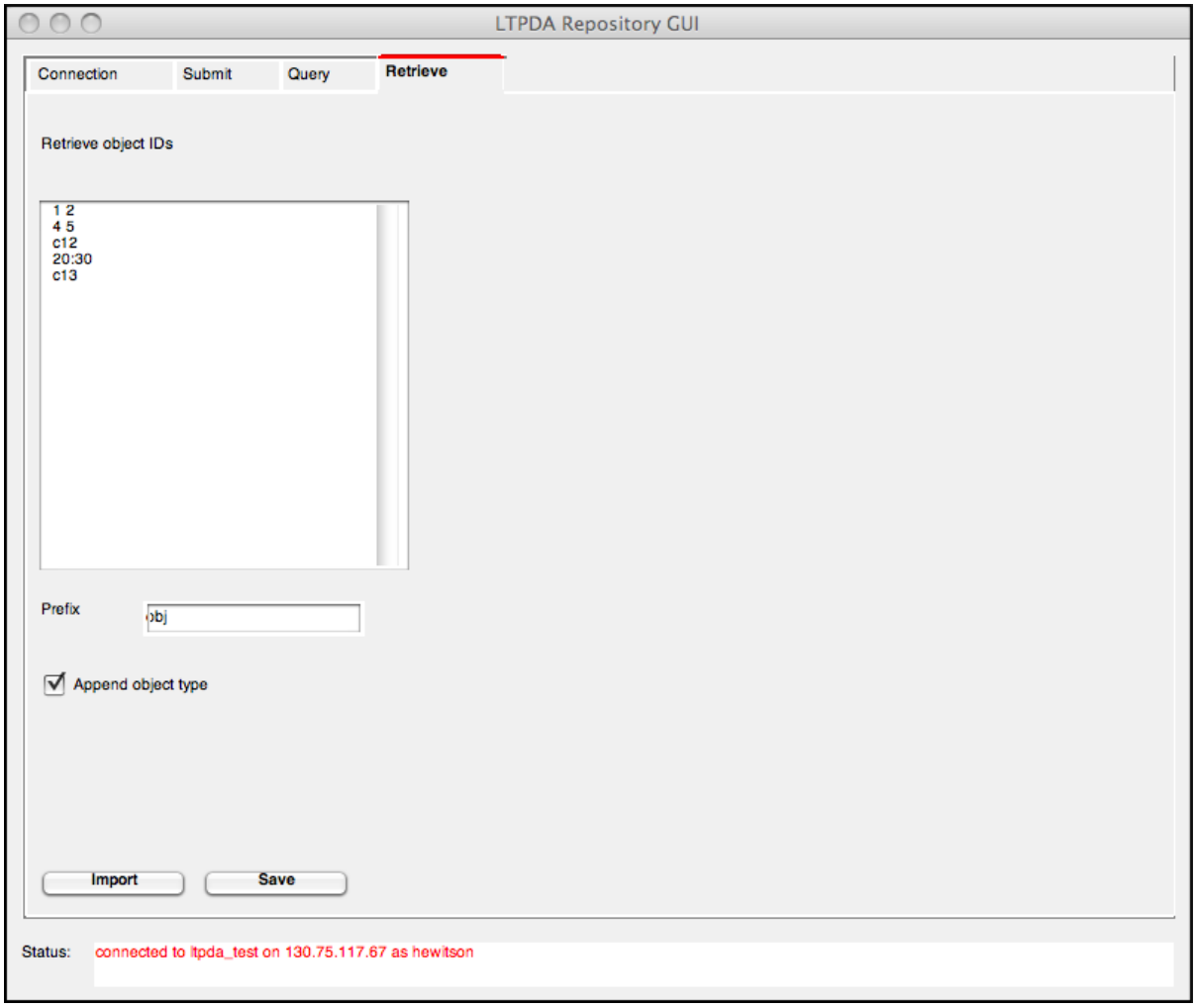
Submitting objects to a repository Retrieving objects and collections from a repository

Retrieving objects and collections from a repository

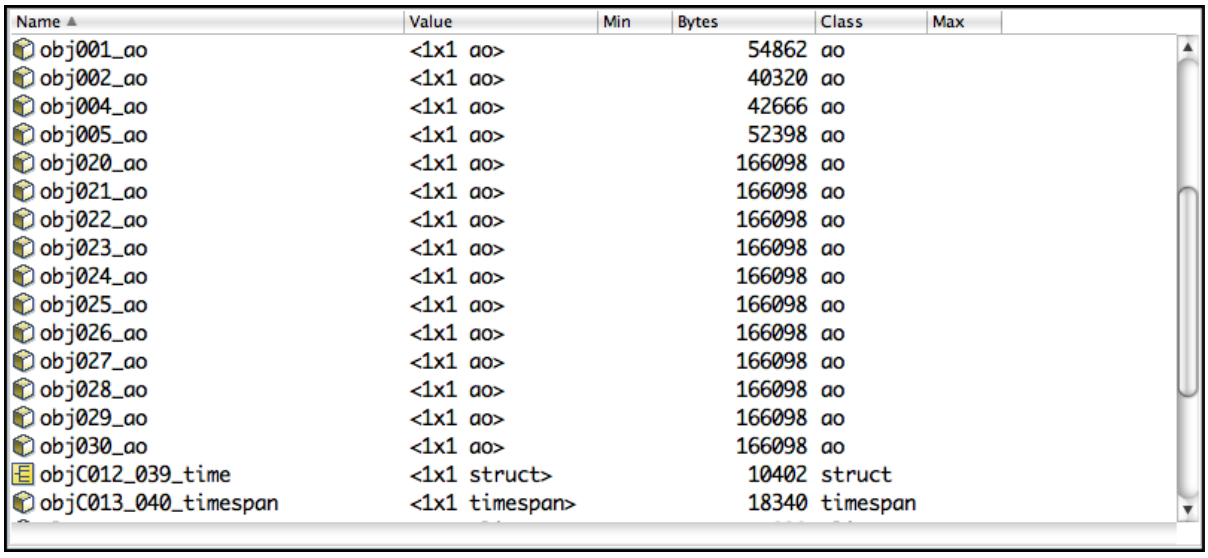
Retrieving objects from an LTPDA repository can be done using the retrieve panel shown below:



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



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



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

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

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

Querying the contents of a repository Class descriptions

Class descriptions

AO class	Implements analysis objects in the LTPDA toolbox
SSM class	Implements statespace model in the 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
SPECWIN class	Implements spectral window objects within LTPDA toolbox
TIME class	Implements time objects within LTPDA toolbox
PZ (POLE/ZERO) class	Implements pole/zero objects within LTPDA toolbox
MINFO class	Implements m-file info objects within LTPDA toolbox
HISTORY class	Implements history objects within LTPDA toolbox
PROVENANCE class	Implements provenance objects within LTPDA toolbox
PARAM class	Implements parameter objects within LTPDA toolbox
UNIT class	Implements unit objects within LTPDA toolbox
CDATA class	Implements constant data objects within LTPDA toolbox

FSDATA class	Implements frequency-series data objects within LTPDA toolbox
TSDATA class	Implements time-series data objects within LTPDA toolbox
XYDATA class	Implements x-y data objects within LTPDA toolbox
XYZDATA class	Implements x-y-z data objects within LTPDA toolbox

Retrieving objects and collections from a repository ao Class

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
mfile	Full text representation of the m-file that created this AO	ao
mfilename	The filename of the m-file that created this AO	ao
mdlfile	Full text representation of the mdl-file that created this AO	ao
mdlfilename	The filename of the mdl-file that created this AO	ao
procinfo	Contains extra processing information not contained in the main result of any method (plist-object).	ao
plotinfo	Plist-object which contains the	ao
version	CVS version string of the constructor	ao
hist	History object associated with this object	ltpda_uoh
description	Description of the object	ltpda_uoh

name	Name of the object	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
attachm	ATTACHM attach an m file to the analysis object.	ao
attachmdl	ATTACHMDL attach an mdl file to the analysis object.	ao
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
fromProcinfo	FROMPROCINFO returns for a given key-name the value of the procinfo-plist	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
md5	MD5 computes an MD5 checksum from an analysis objects.	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
setPlotinfo	SETPLOTINFO sets the 'plotinfo' 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
isprop	ISPROP tests if the given field is one of the object properties.	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
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
setName	SETNAME Set the property 'name'.	ltpda_uoh
setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh

Back to Top of Section

Internal

Methods	Description	Defined in class
ao2m	AO2M converts an analysis object to an '.m' file based on the history.	ao
plot	PLOT a simple plot of analysis objects.	ao
bsubmit	BSUBMIT bsubmits the given collection of objects 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

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 time-series to control forces.	ao
mdc1_x2acc	MDC1_X2ACC converts the input time-series to acceleration with a time-domain filter	ao

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 determinant function for Analysis objects.	ao
exp	EXP overloads the exp operator for Analysis objects. Exponential.	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	ao

	objects. Common (base 10) logarithm.	
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 togther	ao
svd	SVD overloads the determinant function for Analysis objects.	ao
transpose	TRANSPOSE 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 objects.	ao

var	VAR computes the variance of the data in the AO.	ao

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR overloads char() function for analysis objects.	ao
display	DISPLAY implement terminal display for analysis object.	ao
export	EXPORT export an analysis object to a text file.	ao
extractm	EXTRACTM extracts an m-file from an analysis object and saves it to disk.	ao
extractmdl	EXTRACTMDL extracts an mdl file from an analysis object and saves it to disk.	ao
iplot	IPLOT provides an intelligent plotting tool for LTPDA.	ao
iplotyy	IPLOT provides an intelligent plotting tool for LTPDA.	ao
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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
cohere	COHERE estimates the coherence between time-series objects	ao
compute	COMPUTE performs the given operations on the input AOs.	ao
consolidate	CONSOLIDATE resamples all input AOs onto the same time grid.	ao
conv	CONV vector convolution.	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	DROPDUPLICATES drops all duplicate samples in time-series AOs.	ao
dsmean	DSMEAN performs a simple downsampling by taking the mean of every N samples.	ao
evaluateModel	EVALUATEMODEL evaluate a curvefit model.	ao
fft	FFT overloads the fft method for Analysis objects.	ao
filter	FILTER overrides the filter function for analysis	ao

	objects.	
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
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
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 coherence estimation on a log frequency axis.	ao
lcpsd	LCPSD implement cross-power-spectral density estimation on a log frequency axis.	ao
lincom	LINCOM	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
noisegen1D	NOISEGEN1D generates colored noise from withe	ao

	noise.	
noisegen2D	NOISEGEN2D generates cross correleted colored noise from withe noise.	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
pwelch	PWELCH makes power spectral density estimates of the time-series objects	ao
resample	RESAMPLE overloads resample function for AOs.	ao
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
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
tfe	TFE estimates transfer function between time-series objects.	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	ao

	time-series	
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 operator for Analysis objects. Cosine of argument in radians.	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
isnumerical	This binary tells whether the system has numerical content only, or symbolic as well	ssm
timestep	Timestep of the difference equation. Zero means the representation is time continuous and A defines a differential equation.	ssm

nonlin		ssm
inputnames	Names corresponding to each input column-block in the B/D matrices. Cell array of strings	ssm
inputvarnames	Names corresponding to each input column in each column-block of the B/D matrices. Cell array of cell array of Strings	ssm
inputsizes	Width corresponding to each input column-block in the B/D matrices. It is a double vector	ssm
inputconn		ssm
Ninputs	Number of input column-blocks, it is a double	ssm
ssnames	Names corresponding to each input state-block in the A/B/C matrices. Cell array of strings	ssm
ssvarnames	Names corresponding to each variable in each input state-block in the A/B/C matrices. Cell array of cell array of Strings	ssm
sssizes	Size corresponding to each input state-block in the A/B/C matrices. It is a double vector	ssm
ssconn		ssm
Nss	Number of state-blocks, it is a double	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
outputsizes	Width corresponding to each output row-block in the C/D matrix. It is a double vector	ssm
outputconn		ssm
Noutputs	Number of output row-blocks, it is a double	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
paramsigmas	Expected variance of each parameter, stored as a double vector	ssm

Nparams	Total number of parameters	ssm
version	CVS version string of the constructor	ssm
hist	History object associated with this object	ltpda_uoh
description	Description of the object	ltpda_uoh
name	Name of the object	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
ssm	SSM statespace model class constructor.	ssm

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	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
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
setName	SETNAME Set the property 'name'.	ltpda_uoh
setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
findParams	FINDPARAMS returns parameter names matching the given pattern.	ssm

Back to Top of Section

Internal

Methods	Description	Defined in class
bsubmit	BSUBMIT bsubmits the given collection of objects 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

Back to Top of Section

Output

Methods	Description	Defined in class
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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
isstable	tells if ssm is numerically stable	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

Statespace

Methods	Description	Defined in class
assemble	assembles embedded subsytems, with exogenous inputs	ssm
bode	BODE makes a bode plot from the given inputs to outputs.	ssm
copy	COPY Make copy of ssm objects depending of the second input	ssm
double	Convert a statespace model object to double arrays for given i/o	ssm
getParamValues	GETPARAMVALUES returns parameter values for the given names.	ssm
kalman	kalman applies Kalman filtering to a discrete ssm with given i/o	ssm
minreal	minreal gives a minimal realization of a ssm object by deleting unreached states	ssm
modifparams	modifparams enables to modifyy and substitute parameters	ssm
modiftimestep	modiftime modifies the timestep of a ssm object	ssm
modify	modify allows to exectue a string to modify a ssm object	ssm
reduce	reduce enables to do model simplification	ssm
reduce_model	REDUCE_MODEL enables to do model simplification	ssm
resp	resp gives the timewise impulse response of a ssm	ssm

setparams	SETPARAMS enables to set parameters' value	ssm
simulate	simulate simulates a discrete ssm with given inputs	ssm
simulate2	simulate2 is the Old and slower version of simulate	ssm
sminreal	minreal gives a minimal realization of a ssm object by deleting unreached states	ssm
ssm2dot	SSM2DOT converts a statespace model object a DOT file.	ssm
ssm2iirpz	ssm2iirpz converts a statespace model object to an miir or a pzmodel	ssm
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
subsparams	subsparams enables to substitute symbollic patameters	ssm

Back to Top of Section

ao Class mfir Class

mfir Class

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
description	Description of the object	ltpda_uoh
name	Name of the object	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
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
setIunits	SETIUNITS sets the 'iunits' property of the ao.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the ao.	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
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

setName	SETNAME Set the property 'name'.	ltpda_uoh
setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
redesign	REDESIGN redesign the input filter to work for the given sample rate.	mfir

Back to Top of Section

Internal

Methods	Description	Defined in class
setHistout	SETHISTOUT Set the property 'histout'	ltpda_filter
bsubmit	BSUBMIT bsubmits the given collection of objects 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

Back to Top of Section

Output

Methods	Description	Defined in class
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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
resp	RESP Make a frequency response of the filter.	ltpda_filter

Back to Top of Section

ssm Class miir Class

miir Class

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

Back to Class descriptions

Properties

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
description	Description of the object	ltpda_uoh

name	Name of the object	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
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
setIunits	SETIUNITS sets the 'iunits' property of the ao.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the ao.	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
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	ltpda_uoh

	an ltpda_uoh object.	
setName	SETNAME Set the property 'name'.	ltpda_uoh
setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh
redesign	REDESIGN redesign the input filter to work for the given sample rate.	miir

Back to Top of Section

Internal

Methods	Description	Defined in class
setHistout	SETHISTOUT Set the property 'histout'	ltpda_filter
bsubmit	BSUBMIT bsubmits the given collection of objects 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
setHistin	SETHISTIN Set the property 'histin'	miir

Back to Top of Section

Output

Methods	Description	Defined in class
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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
resp	RESP Make a frequency response of the filter.	ltpda_filter

Back to Top of Section

mfir Class pzmodel Class

pzmodel Class

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

Back to Class descriptions

Properties

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
gain	Gain of the model	pzmodel
poles	Vector of poles (pz-objects)	pzmodel
zeros	Vector of zeros (pz-objects)	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
description	Description of the object	ltpda_uoh
name	Name of the object	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
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
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
setIunits	SETIUNITS sets the 'iunits' property of the ao.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the ao.	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
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
setName	SETNAME Set the property 'name'.	ltpda_uoh

setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	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
bsubmit	BSUBMIT bsubmits the given collection of objects 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
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

report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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.	pzmodel
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
simplify	SIMPLIFY simplifies pzmodels by cancelling like poles with like zeros.	pzmodel
times	TIMES overloads the multiplication operator for pzmodels.	pzmodel

Back to Top of Section

miir Class parfrac Class

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		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
description	Description of the object	ltpda_uoh
name	Name of the object	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
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
setIunits	SETIUNITS sets the 'iunits' property of the ao.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the ao.	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
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
setName	SETNAME Set the property 'name'.	ltpda_uoh

setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	ltpda_uoh

Back to Top of Section

Internal

Methods	Description	Defined in class
bsubmit	BSUBMIT bsubmits the given collection of objects 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
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 rational object.	parfrac

Back to Top of Section

Output

Methods	Description	Defined in class
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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

pzmodel Class rational Class

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
description	Description of the object	ltpda_uoh
name	Name of the object	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
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
setIunits	SETIUNITS sets the 'iunits' property of the ao.	ltpda_tf
setOunits	SETOUNITS sets the 'ounits' property of the ao.	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
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
setName	SETNAME Set the property 'name'.	ltpda_uoh
setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used	ltpda_uoh

	to recreate the input object(s).

Back to Top of Section

Internal

Methods	Description	Defined in class
bsubmit	BSUBMIT bsubmits the given collection of objects 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
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
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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
description	Description of the object	ltpda_uoh
name	Name of the object	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
isprop	ISPROP tests if the given field is one of the object properties.	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
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
setName	SETNAME Set the property 'name'.	ltpda_uoh
setProperties	SETPROPERTIES set different properties of an object.	ltpda_uoh
string	STRING writes a command string that can be used to recreate the input object(s).	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
bsubmit	BSUBMIT bsubmits the given collection of objects 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

Back to Top of Section

Output

Methods	Description	Defined in class
report	REPORT generates an HTML report about the input objects.	ltpda_uoh
save	SAVE overloads save operator for ltpda objects.	ltpda_uoh
type	TYPE converts the input objects to MATLAB functions.	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

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
plist	PLIST Plist class object constructor.	plist

Back to Top of Section

GUI function

Methods	Description	Defined in class
g_constructor	No description	plist

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
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
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 pairor a param-object into the parameter list.	plist
remove	REMOVE remove a parameter from the parameter list.	plist

setDesc	SETDESC Set the property 'desc'.	plist
setDescription	SETDESCRIPTION Set the property 'description'.	plist
setName	SETNAME Set the property 'name'.	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
bsubmit	BSUBMIT bsubmits the given collection of objects 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
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
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 shadows miir/iirResp and pzmodel/resp.	plist

Back to Top of Section

timespan Class specwin Class

specwin Class

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

Back to Class descriptions

Properties

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
type	Name of the spectral window object	specwin
alpha	Alpha parameter for various window functions	specwin
psll	Peak sidelobe level	specwin
rov	Recommended overlap	specwin
nenbw	Normalised equivalent noise bandwidth	specwin
w3db	dB bandwidth in bins	specwin
flatness	Window flatness	specwin
ws	Sum of window values	specwin
ws2	Sum of squares of window values	specwin
win	Window samples (column vector)	specwin
version	CVS version string of the constructor	specwin

Back to Top

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

Back to Top

Constructor

Methods	Description	Defined in class
specwin	% SPECWIN spectral window object class constructor.	specwin

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a specwin object into a string.	specwin
display	DISPLAY overloads display functionality for specwin objects.	specwin
plot	PLOT plots a specwin object.	specwin
string	STRING writes a command string that can be used to recreate the input window object.	specwin

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 time Class

time Class

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

Back to Class descriptions

Properties

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
utc_epoch_milli	Unix epoch time in milliseconds. The underlying timezone is UTC	time
timezone	Timezone of the current time	time
timeformat	Time format of the current time	time
time_str	Time string depending on the unix epoch time and time format	time
version	CVS version string of the constructor	time

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

Output	Output methods
Relational Operator	Relational operator methods

Back to Top

Arithmetic Operator

Methods	Description	Defined in class
minus	MINUS overloads - operator for time objects.	time
plus	PLUS overloads + operator for time objects.	time

Back to Top of Section

Constructor

Methods	Description	Defined in class
time	TIME time object class constructor.	time

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
string	STRING writes a command string that can be used to recreate the input time object.	time

Back to Top of Section

Internal

Methods	Description	Defined in class
setEpochtime	SETEPOCHTIME Set the property 'utc_epoch_milli'.	time
setTime_str	SETTIME_STR Set the property 'time_str'.	time
setTimeformat	SETTIMEFORMAT Set the property 'timeformat'.	time
setTimezone	SETTIMEZONE Set the property 'timezone'.	time

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a time object into a string.	time
display	DISPLAY overloads display functionality for time objects.	time
format	FORMAT Returns the time in specified format.	time

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

specwin Class pz (pole/zero) Class

pz (pole/zero) Class

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

Back to Class descriptions

Properties

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
f	Frequency of pole/zero	pz
q	Quality factor of pole/zero	pz
ri	Complex representation of pole/zero	pz
version	CVS version string of the constructor	pz

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
pz	PZ is the ltpda class that provides a common definition of poles and zeros.	pz

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
cp2iir	CP2IIR Return a,b IIR filter coefficients for a complex pole designed using the bilinear transform.	pz
cz2iir	CZ2IIR return a,b IIR filter coefficients for a complex zero designed using the bilinear transform.	pz
rp2iir	RP2IIR Return a,b coefficients for a real pole designed using the bilinear transform.	pz
rz2iir	RZ2IIR Return a,b IIR filter coefficients for a real zero designed using the bilinear transform.	pz

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a pz object into a string.	pz
display	DISPLAY display a pz object.	pz
string	STRING writes a command string that can be used to recreate the input pz object.	pz

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 the pz object.	pz

Back to Top of Section

time Class minfo Class

minfo 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
mname	Method name where the minfo object is defined	minfo
mclass	Class name where the minfo object is defined	minfo
mpackage	Package name where the minfo object is defined	minfo
mcategory	Category of the of the method where the minfo object is defined	minfo
mversion	Version string of the method where the minfo object is defined	minfo
sets	Possible set name for the different Default parameter lists	minfo
plists	All different default parameter lists	minfo
argsmin	Minimal input objects of the method	minfo
argsmax	Maximal input objects of the method	minfo
outmin	Minimal output objects of the method	minfo

outmax	Maximal input objects of the method	minfo
modifier	Defines if the method can be used as a modifier	minfo
version	CVS version string of the constructor	minfo

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
minfo	MINFO a helper class for LTPDA methods.	minfo

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
setArgsmax	SETARGSMAX Set the property 'argsmax'.	minfo
setArgsmin	SETARGSMIN Set the property 'argsmin'.	minfo
setModifier	SETMODIFIER Set the property 'modifier'.	minfo

setMversion	SETMVERSION Set the property 'mversion'.	minfo
setOutmax	SETOUTMAX Set the property 'outmax'.	minfo
setOutmin	SETOUTMIN Set the property 'outmin'.	minfo

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert an minfo object into a string.	minfo
display	DISPLAY display an minfo object.	minfo

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

pz (pole/zero) Class history Class

history Class

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

Back to Class descriptions

Properties

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
methodInfo	minfo-object which is created in the called method.	history
plistUsed	plist-object which is used in the called method.	history
methodInvars	Variable names which are used for the called method.	history
inhists	The older history-objects	history
proctime	Creation time of the history object (unix epoch time)	history
version	CVS version string of the constructor	history

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
history	HISTORY History object class constructor.	history
rebuild	REBUILD rebuilds the orignal object using the history.	history

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
getNodes	GETNODES converts a history object to a nodes structure suitable for plotting as a tree.	history

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a param object into a string.	history
display	DISPLAY implement terminal display for history object.	history
dotview	DOTVIEW view history of an object via the DOT interpreter.	history
hist2dot	HIST2DOT converts a history object to a 'DOT' file suitable for processing with graphviz	history
hist2m	HIST2M writes a new m-file that reproduces the analysis described in the history object.	history

plot	PLOT plots a history object as a tree diagram.	history
string	STRING writes a command string that can be used to recreate the input history object.	history

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

minfo Class provenance Class

provenance Class

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

Back to Class descriptions

Properties

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
creator	Current user of the LTPDA toolbox	provenance
ip	IP address of the creator	provenance
hostname	Hostname of the creator	provenance
os	Used system of the creator	provenance
matlab_version	MATLAB version	provenance
sigproc_version	Signal Processing Toolbox version	provenance
symbolic_math_version	Symbolic Math Toolbox version	provenance
optimization_version	Optimization Toolbox version	provenance
database_version	Database Toolbox version	provenance
control_version		provenance
ltpda_version	LTPDA Toolbox version	provenance
version	CVS version string of the constructor	provenance

Back to Top

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

Back to Top

Constructor

Methods	Description	Defined in class
provenance	PROVENANCE constructors for provenance class.	provenance

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a provenance object into a string.	provenance
display	DISPLAY overload terminal display for provenance objects.	provenance
string	STRING writes a command string that can be used to recreate the input provenance object.	provenance

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

history Class param Class

param Class

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

Back to Class descriptions

Properties

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
key	Key of the key/value pair	param
val	Val of the key/value pair	param
desc	Description of the key/value pair	param
version	CVS version string of the constructor	param

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
param	PARAM Parameter object class constructor.	param

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
mux	MUX concatenate params into a vector.	param

Back to Top of Section

Internal

Methods	Description	Defined in class
setDesc	SETDESC Set the property 'desc'.	param
setKey	SETKEY Set the property 'key'.	param
setKeyVal	SETKEYVAL Set the properties 'key' and 'val'	param
setVal	SETVAL Set the property 'val'.	param

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a param object into a string.	param
display	DISPLAY display a parameter	param
string	STRING writes a command string that can be used to recreate the input param object.	param

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

provenance Class unit Class

unit 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
strs	Unit character	unit
exps	Exponent of the unit	unit
vals	Number of the used unit prefix (mm -> 1e-3)	unit
version	CVS version string of the constructor	unit

Back to Top

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage
Output	Output methods
Relational Operator	Relational operator methods

Back to Top

Constructor

Methods	Description	Defined in class
unit	UNIT a helper class for implementing units in LTPDA.	unit

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj
string	STRING converts a unit object to a command string which will recreate the unit object.	unit
tolabel	TOLABEL converts a unit object to LaTeX string suitable for use as axis	unit

Back to Top of Section

Output

Methods	Description	Defined in class
display	DISPLAY display an unit object.	unit
factor	FACTOR factorises units in to numerator and denominator units.	unit

Back to Top of Section

Relational Operator

Methods	Description	Defined in class
eq	EQ overloads the == operator for ltpda unit objects.	unit
ne	NE overloads the ~= operator for ltpda unit objects.	unit

Back to Top of Section

cdata Class

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

Back to Class descriptions

Properties

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
yunits	Units of the y-axis	cdata
y	Data values of the y-axis	cdata
version	CVS version string of the constructor	cdata

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
cdata	CDATA is the constant data class.	cdata

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
applymethod	APPLYMETHOD applys the given method to the input 2D data.	cdata
applyoperator	APPLYOPERATOR applys the given operator to the two input data objects.	cdata
char	CHAR convert a cdata-object into a string.	cdata
getY	GETY Get the property 'y'.	cdata
setY	SETY Set the property 'y'.	cdata
setYunits	SETYUNITS Set the property 'yunits'.	cdata

Back to Top of Section

Output

Methods	Description	Defined in class
display	DISPLAY implement terminal display for cdata object.	cdata

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

unit Class fsdata Class

fsdata Class

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

Back to Class descriptions

Properties

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
t0	Time-stamp of the first data sample	fsdata
navs	Number of averages	fsdata
fs	Sample rate of data	fsdata
enbw	Equivalent noise bandwidth	fsdata
version	CVS version string of the constructor	fsdata
xunits	Units of the x-axis	data2D
yunits	Units of the y-axis	data2D
x	Data values of the x-axis	data2D
y	Data values of the y-axis	data2D
dx	Error on x values	data2D
dy	Error on y values	data2D

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
fsdata	FSDATA frequency-series object class constructor.	fsdata

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
applymethod	APPLYMETHOD applys the given method to the input 2D data.	data2D
applyoperator	APPLYOPERATOR applys the given operator to the two input data objects.	data2D
getDx	GETDX Get the property 'dx'.	data2D
getDy	GETDY Get the property 'dy'.	data2D
getX	GETX Get the property 'x'.	data2D
getY	GETY Get the property 'y'.	data2D

setDx	SETDX Set the property 'dx'.	data2D
setDy	SETDY Set the property 'dy'.	data2D
setX	SETX Set the property 'x'.	data2D
setXY	SETXY Set the property 'xy'.	data2D
setXunits	SETXUNITS Set the property 'xunits'.	data2D
setY	SETY Set the property 'y'.	data2D
setYunits	SETYUNITS Set the property 'yunits'.	data2D
setEnbw	SETENBW Set the property 'enbw'.	fsdata
setFs	SETFS Set the property 'fs'.	fsdata
setNavs	SETNAVS Set the property 'navs'.	fsdata
setT0	SETT0 Set the property 't0'.	fsdata

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a ltpda_data-object into a string.	data2D
display	DISPLAY implement terminal display for fsdata object.	fsdata

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

cdata Class tsdata Class

tsdata Class

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

Back to Class descriptions

Properties

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
t0	Time-stamp of the first data sample	tsdata
fs	Sample rate of data	tsdata
nsecs	The length of this time-series in seconds	tsdata
version	CVS version string of the constructor	tsdata
xunits	Units of the x-axis	data2D
yunits	Units of the y-axis	data2D
x	Data values of the x-axis	data2D
y	Data values of the y-axis	data2D
dx	Error on x values	data2D
dy	Error on y values	data2D

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
tsdata	TSDATA time-series object class constructor.	tsdata

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
applymethod	APPLYMETHOD applys the given method to the input 2D data.	data2D
applyoperator	APPLYOPERATOR applys the given operator to the two input data objects.	data2D
getDx	GETDX Get the property 'dx'.	data2D
getDy	GETDY Get the property 'dy'.	data2D
getY	GETY Get the property 'y'.	data2D
setDx	SETDX Set the property 'dx'.	data2D
setDy	SETDY Set the property 'dy'.	data2D

setX	SETX Set the property 'x'.	data2D
setXY	SETXY Set the property 'xy'.	data2D
setXunits	SETXUNITS Set the property 'xunits'.	data2D
setY	SETY Set the property 'y'.	data2D
setYunits	SETYUNITS Set the property 'yunits'.	data2D
collapseX	COLLAPSEX Checks whether the x vector is evenly sampled and then removes it	tsdata
fixNsecs	FIXNSECS fixes the numer of seconds.	tsdata
getX	GETX Get the property 'x'.	tsdata
growT	GROWT grows the time (x) vector if it is empty.	tsdata
setFs	SETFS Set the property 'fs'.	tsdata
setNsecs	SETNSECS Set the property 'nsecs'.	tsdata
setT0	SETT0 Set the property 't0'.	tsdata

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a ltpda_data-object into a string.	data2D
display	DISPLAY overloads display functionality for tsdata objects.	tsdata

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

fsdata Class xydata Class

xydata Class

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

Back to Class descriptions

Properties

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
version	CVS version string of the constructor	xydata
xunits	Units of the x-axis	data2D
yunits	Units of the y-axis	data2D
x	Data values of the x-axis	data2D
y	Data values of the y-axis	data2D
dx	Error on x values	data2D
dy	Error on y values	data2D

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
xydata	XYDATA X-Y data object class constructor.	xydata

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
applymethod	APPLYMETHOD applys the given method to the input 2D data.	data2D
applyoperator	APPLYOPERATOR applys the given operator to the two input data objects.	data2D
getDx	GETDX Get the property 'dx'.	data2D
getDy	GETDY Get the property 'dy'.	data2D
getX	GETX Get the property 'x'.	data2D
getY	GETY Get the property 'y'.	data2D
setDx	SETDX Set the property 'dx'.	data2D
setDy	SETDY Set the property 'dy'.	data2D
setX	SETX Set the property 'x'.	data2D
setXY	SETXY Set the property 'xy'.	data2D

setXunits	SETXUNITS Set the property 'xunits'.	data2D
setY	SETY Set the property 'y'.	data2D
setYunits	SETYUNITS Set the property 'yunits'.	data2D

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a ltpda_data-object into a string.	data2D
display	DISPLAY overloads display functionality for xydata objects.	xydata

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

tsdata Class xyzdata Class

xyzdata Class

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

Back to Class descriptions

Properties

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
version	CVS version string of the constructor	xyzdata
zunits	Units of the z-axis	data3D
z	Data values of the z-axis	data3D
xunits	Units of the x-axis	data2D
yunits	Units of the y-axis	data2D
x	Data values of the x-axis	data2D
y	Data values of the y-axis	data2D
dx	Error on x values	data2D
dy	Error on y values	data2D

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

Back to Top of Section

Helper

Methods	Description	Defined in class
get	GET get a property of a object.	ltpda_obj
isprop	ISPROP tests if the given field is one of the object properties.	ltpda_obj

Back to Top of Section

Internal

Methods	Description	Defined in class
getDx	GETDX Get the property 'dx'.	data2D
getDy	GETDY Get the property 'dy'.	data2D
getX	GETX Get the property 'x'.	data2D
getY	GETY Get the property 'y'.	data2D
setDx	SETDX Set the property 'dx'.	data2D
setDy	SETDY Set the property 'dy'.	data2D
setX	SETX Set the property 'x'.	data2D
setXY	SETXY Set the property 'xy'.	data2D
setXunits	SETXUNITS Set the property 'xunits'.	data2D

setY	SETY Set the property 'y'.	data2D
setYunits	SETYUNITS Set the property 'yunits'.	data2D
getZ	GETZ Get the property 'z'.	data3D
setZ	SETZ Set the property 'z'.	data3D
setZunits	SETZUNITS Set the property 'zunits'.	data3D

Back to Top of Section

Output

Methods	Description	Defined in class
char	CHAR convert a ltpda_data-object into a string.	xyzdata
display	DISPLAY overloads display functionality for xyzdata objects.	xyzdata

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

xydata Class Constructor Examples

Constructor Examples

Constructor examples

Constructor examples of the AO class

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']
'use_fs'	If this value is set, the x-axes is computed by the fs value. [default: empty]
'columns'	[1 2 1 4] Each pair represents the x- and y-axes (each column pair creates an analysis object). If the value 'use_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.
% 'use_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 (`use_fs`) and each column of data is converted in to a single AO.

```
    % 'use_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',     ...
    'use_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'

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 gauspuls)
'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 Constructor examples of the MFIR class

Constructor examples of the MFIR class

Copy an FIR filter object

The following example creates a copy of an FIR filter object (blue command).

```
>> fir1 = mfir(plist('type', 'lowpass'));
>> fir2 = mfir(fir2)
------ mfir/1 -------
     gd: 32.5
version: $Id: mfir.m,v 1.84 2009/02/24 17:02:44 ingo Exp
  ntaps: 65
     fs: 1
 infile:
      a: [0.0007 0.0004 -6.9093e-19 -0.0006 -0.0012 -0.0014 -0.0011 1.4147e-18 ...
histout: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  ...
 iunits: [] [1x1 unit]
 ounits: [] [1x1 unit]
   hist: mfir.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.

```
>> fir1 = mfir()
------ 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: none
----------------------
>> fir2 = fir1;
>> fir2.setName('my new name')
------ 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
---------------------
```

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);
iplot(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]

The following example creates a mfir object from an analysis object.

```
a1  = ao(plist('fsfcn', '1./(50+f)', 'fs', 1000, 'f', linspace(0, 500, 1000)));
pl  = plist('ao', a1);
fir = mfir(pl)
```

Construct an FIR filter object from a pole/zero model

The following example creates a new FIR 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)
--------------------
>> fir = mfir(pzm) % Use the default sample rate fs=8 * frequency of the highest pole or
zero in the model
>> fir = mfir(pzm, plist('fs', 100))
------ mfir/1 -------
     gd: 257
version: $Id: mfir.m,v 1.84 2009/02/24 17:02:44 ingo Exp
  ntaps: 513
     fs: 100
 infile:
      a: [-0 -3.013e-10 -1.2486e-09 -2.8506e-09 -5.1166e-09 -7.8604e-09 -1.1133e-08 ...
histout: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
 iunits: [] [1x1 unit]
 ounits: [] [1x1 unit]
   hist: mfir.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)
>> fir = mfir(pl)
```

Construct a MFIR object from a standard type

Construct an FIR 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 = 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:

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

```
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) + ... + a(na+1)*x(n-na).

Constructor examples of the AO class Constructor examples of the MIIR class

Constructor examples of the MIIR class

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(iir2)
------ 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
  ntaps: 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:

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

```
a  = [0.5 -0.01];
b  = [1 0.1]
fs = 1;

f = miir(a,b,fs)
```

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) + ... + a(na+1)*x(n-na) - b(2)*y(n-1) - ... - b(nb+1)*y(n-nb)

Constructor examples of the MFIR class Constructor examples of the PZMODEL class

Constructor examples of the PZMODEL class

General information about pole/zero models

Construct empty PZMODEL object

The following example creates an empty pzmodel object

```
pzm = pzmodel()
---- pzmodel 1 ----
model:    None
gain :    0
pole 001: pole(NaN)
zero 001: zero(NaN)
--------------------
```

Construct a PZMODEL object by loading the object from a file

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

```
p = pzmodel('pzmodel.mat')
p = pzmodel('pzmodel.xml')
```

Construct a PZMODEL object from gain, poles and zeros

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

```
gain  = 10;
poles = [pole(1,2) pole(40)];
zeros = [zero(10,3) zero(100)];

pzm = pzmodel(gain, poles, zeros)
---- pzmodel 1 ----
model:    None
gain :    10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
```

```
zero 002: zero(100)
------------------
```

It is possible to give the model direct a name.

```
gain  = 10;
poles = [pole(1,2) pole(40)];
zeros = [zero(10,3) zero(100)];

pzm = pzmodel(gain, poles, zeros, 'my model name')
---- pzmodel 1 ----
model:    my model name
gain :    10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
zero 002: zero(100)
------------------
```

Construct a PZMODEL object from an existing model

The pzmodel constructor also accepts as an input existing models in a LISO file format

```
pzm = pzmodel('foo.fil')
```

Construct a PZMODEL object from a parameter list (PLIST) object

Construct a PZMODEL from its definion.

'gain'	Model gain [default: 1]
'poles'	Vector of pole objects [default: empty pole]
'zeros'	Vector of zero objects [default: empty zero]
'name'	Name of model [default: 'None']

```
poles = [pole(0.1) pole(1,100)];
zeros = [zero(10,3) zero(100)];
pl = plist('name', 'my filter', 'poles', poles, 'zeros', zeros, 'gain', 10);

pzm = pzmodel(pl)
---- pzmodel 1 ----
model:    my filter
gain :    10
pole 001: pole(0.1)
pole 002: pole(1,100)
zero 001: zero(10,3)
zero 002: zero(100)
------------------
```

Constructor examples of the MIIR class Constructor examples of the PARFRAC class

Constructor examples of the PARFRAC class

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:      [0.999999999999999;-6.00000000000001;5.99999999999999]
    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 PZMODEL class Constructor examples of the RATIONAL class

Constructor examples of the RATIONAL class

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:   []
    --------------------
```

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'

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 PARFRAC class Constructor examples of the TIMESPAN class

Constructor examples of the TIMESPAN class

Construct empty TIMESPAN object

The following example creates an empty timespan object

```
ts = timespan()
---------- timespan 01 ----------

name       : None
start      : 2008-03-30 20:00:00.000
end        : 2008-03-30 20:00:00.000
timeformat: yyyy-mm-dd HH:MM:SS.FFF
interval   :
timezone   : UTC

created    : 2008-03-30 20:00:00.000
version    : $Id: timespan.m,v 1.23 2008/03/25 10:57:49 mauro Exp
plist      : plist class
---------------------------------
```

Construct a TIMESPAN object by loading the object from a file

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

```
t = timespan('timespan.mat')
t = timespan('timespan.xml')
```

Construct a TIMESPAN object with a start and end time

It is possible to specify the start-/end- time either with a time-object or with a time string.

```
t1_obj = time('14:00:00');
t2_obj = time('15:00:00');
t1_str = '14:00:00';
t2_str = '15:00:00';

ts1 = timespan(t1_obj, t2_obj)    % two time-objects
ts2 = timespan(t1_obj, t2_obj)    % two strings
ts3 = timespan(t1_str, t2_obj)    % combination of time-object and string
ts4 = timespan(t1_str, t2_str)    % combination of time-object and string
```

Construct a TIMESPAN object from a parameter list (PLIST) object

Construct an TIMESPAN by its properties definition

'start' The starting time [default: '1970-01-01 00:30:00.000']

'end' The ending time [default: '1980-01-01 12:00:00.010']

Additional parameters:

'timezone' Timezone (string or java object) [default: 'UTC']

'timeformat' Time format (string) [default: 'yyyy-mm-dd HH:MM:SS.FFF']

```
t1 = time('14:00:00');
t2 = time('15:00:00');

pl1 = plist('start', t1, ...
            'end',   t2);
pl2 = plist('start',       t1, ...
            'end',         t2, ...
            'timeformat', 'yyyy-mm-dd HH:MM:SS', ...
            'timezone',   'CET');

ts1 = timespan(pl1)
ts2 = timespan(pl2)
```

Constructor examples of the RATIONAL class Constructor examples of the PLIST class

Constructor examples of the PLIST class

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

Creating parameter lists from parameters.

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

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

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

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


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

Creating parameter lists directly.

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

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

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

Appending parameters to a parameter list.

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

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

Finding parameters in a parameter list.

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

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

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

Removing parameters from a parameter list.

You can also remove parameters from a parameter list:

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

Setting parameters in a parameter list.

You can also set parameters contained in a parameter list:

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

Combining multiple parameter lists.

Parameter lists can be combined:

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

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

Constructor examples of the TIMESPAN class Constructor examples of the SPECWIN class

Constructor examples of the SPECWIN class

Construct empty SPECWIN object

The following example creates an empty specwin object

```
w = specwin()
-------- None ------------

   alpha: -1
    psll: -1
     rov: -1
   nenbw: -1
    w3db: -1
flatness: -1
      ws: -1
     ws2: -1
     win: 1

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

Construct a SPECWIN object by loading the object from a file

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

```
p = specwin('specwin.mat')
p = specwin('specwin.xml')
```

Construct a SPECWIN of a particular window type an length

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

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

   alpha: 0
    psll: 31.5
     rov: 50
   nenbw: 1.5
    w3db: 1.4382
```

```
      flatness: -1.4236
            ws: 50
           ws2: 37.5
           win: 100

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

List of available window functions

Construct a SPECWIN Kaiser window with the prescribed psll

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

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

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

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

Construct a SPECWIN object from a parameter list (PLIST) object

Construct a SPECWIN from its definion.

'Name'	Spectral window name [default: 'Kaiser']
'N'	Spectral window length [default: 10]
'PSLL'	Peak sidelobe length (only for Kaiser type) [default: 150]

```
pl = plist('Name', 'Kaiser', 'N', 1000, 'PSLL', 75);

w = specwin(pl)
-------- Kaiser ------------

   alpha: 3.21394
    psll: 75
     rov: 63.4095
   nenbw: 1.85055
    w3db: 1.75392
flatness: -0.963765
      ws: 389.305
     ws2: 280.892
     win: 1000
```

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

```
    pl = plist('Name', 'Hanning', 'N', 1000);

    w =specwin(pl)
    -------- Hanning ------------

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

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

Constructor examples of the PLIST class

Functions - By Category

Functions - By Category

Categories

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.

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.
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	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.
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 arbitrarly 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	%%%
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	%%%
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.
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 lscov 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.
fromProcinfo	ao	% FROMPROCINFO returns for a given key-name the value of the procinfo-plist
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_uo	% 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.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an object.
setT0	ao	% SETT0 sets the 't0' property of the ao.
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 pairor 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'
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	
setIunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the ao.
setMdlfile	ltpda_uo	% 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 ao.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an 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	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	
setIunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the ao.
setMdlfile	ltpda_uo	% 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 ao.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an 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.

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.
setIunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the ao.
setMdlfile	ltpda_uo	% 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 ao.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an 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.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setIunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the ao.
setMdlfile	ltpda_uo	% 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 ao.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an 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.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setIunits	ltpda_tf	% SETIUNITS sets the 'iunits' property of the ao.
setMdlfile	ltpda_uo	% 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 ao.
setPlotinfo	ltpda_uoh	% SETPLOTINFO sets the 'plotinfo' property of an ltpda_uoh object.
setProcinfo	ltpda_uoh	% SETPROCINFO sets the 'procinfo' property of the ao.

setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an 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.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setEndT	timespan	% SETENDT Set the property 'endT'.
setMdlfile	ltpda_uo	% 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.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an object.
setStartT	timespan	% SETSTARTT Set the property 'startT'.
setTimeformat	timespan	% SETTIMEFORMAT Set the property 'timeformat'.
setTimezone	timespan	% SETTIMEZONE Set the property 'timezone'.
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.
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_uo	% 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.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an object.

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.
inv	smodel	% INV makes the inverse of a matrix of smodels
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_uo	% 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.

setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an object.
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_uo	% 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.

setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an 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.
ncols	matrix	% NCOLS Retruns the number of columns of the inner objects.
nrows	matrix	% NROWS Retruns the number of rows of the inner objects.
osize	matrix	% OSIZE Retruns the size of columns of the inner objects.
setDescription	ltpda_uoh	% SETDESCRIPTION sets the 'description' property of an ltpda_uoh object.
setMdlfile	ltpda_uo	% 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.
setProperties	ltpda_uoh	% SETPROPERTIES set different properties of an object.

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 togther
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 subsytems, with exogenous inputs
kalman	ssm	% KALMAN applies Kalman filtering to a discrete ssm with given i/o
keepParameters	ssm	% KEEPPARAMETERS enables to substitute symbollic patameters
modifTimeStep	ssm	% MODFTIMESTEP modifies the timestep of a ssm object
sMinReal	ssm	% SMINREAL gives a minimal realization of a ssm object by deleting unreached states
simplify	simplify	
simulate	ssm	% SIMULATE simulates a discrete ssm with given inputs
subsParameters	ssm	% SUBSPARAMETERS enables to substitute symbolic patameters
convol_integral	smodel	% CONVOL_INTEGRAL implements the convolution integral for smodel objects.
det	smodel	% DET evaluates the determinant for smodel objects matrices.
diff	smodel	% DIFF implements differentiation operator for smodel objects.

inv_laplace_transform	smodel	% INV_LAPLACE_TRANSFORM implements continuous s-domain inverse Laplace transform for smodel objects.
simplify	smodel	% SIMPLIFY implements simplify operator for smodel objects.
sum	smodel	% SUM adds all the elements of smodel objects arrays.
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.
iplotyy	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.
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.
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.
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.
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.
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.
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.
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.
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.
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.
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.

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.

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
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
g_constructor	plist	% This is the generic constructor for LTPDA package, to be used in the GUI

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

Constructor examples of the SPECWIN class LTPDA Training Session 1

LTPDA Training Session 1

This series of help pages consitute the first training session of LTPDA. The various data-packs used throughout the tutorials are available for download on the LTPDA 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 evem 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

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:

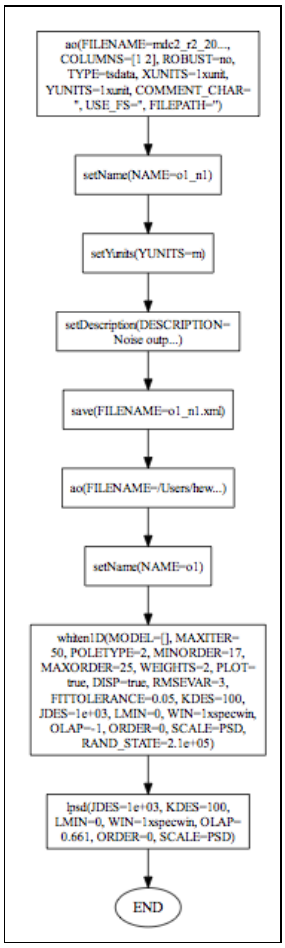
- constructing Analysis Objects (AOs) from simulated data
- constructing AOs from data files
- setting the properties of an AO
- basic operations on AOs, for example, adding and multiplying AOs together
- plotting the data in AOs
- viewing the history of an AO

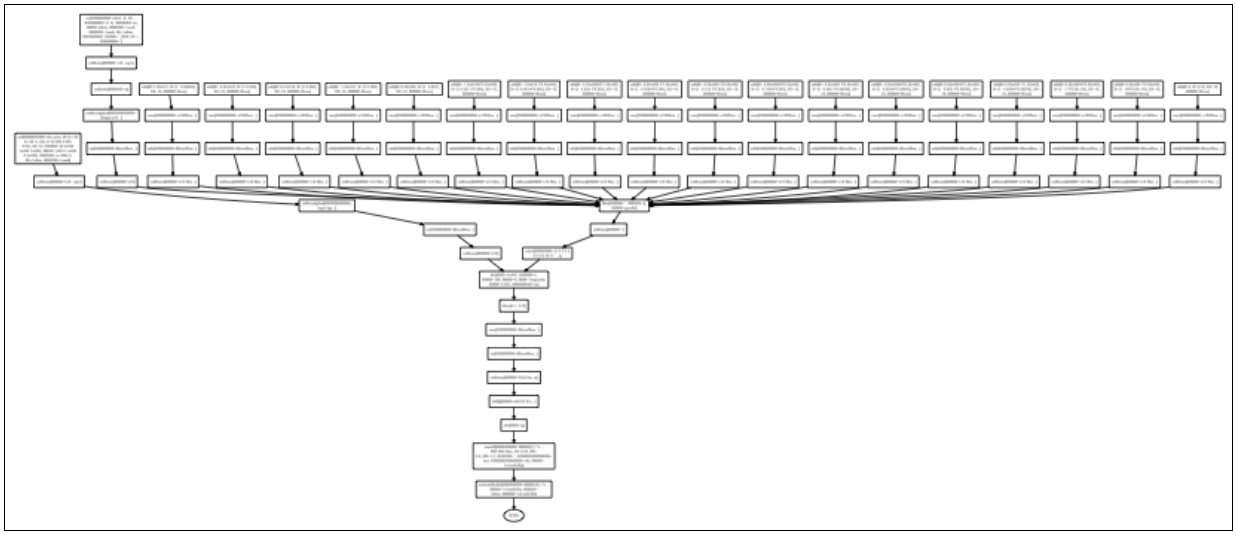
LTPDA Training Session 1 Introducing Analysis Objects

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:





Topic 1 - The basics of LTPDA Making AOs

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 output like the following:

```
>> a = ao
M: running ao/ao
M: running ao/display
----------- ao 01: a -----------

       name:  none
description:
       data:  None
       hist:  ao / ao / $Id: ltpda_training_topic_1_2_content.html,v 1.4 2009/03/08 11:46:00 hewitson Exp $
  mfilename:
mdlfilename:
--------------------------------
```

You have just made your first AO. It's not a very exciting AO since it contains no data, but it is an AO nontheless. 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
description:
       data: 1
             -------- cdata 01 ------------
                   y:  [1x1], double
             yunits:  []
             -------------------------------

       hist:  ao / ao / $Id: ltpda_training_topic_1_2_content.html,v 1.4 2009/03/08 11:46:00 hewitson Exp $-->$Id: ltpda_training_topic_1_2_content.html,v 1.4
2009/03/08 11:46:00 hewitson Exp $
  mfilename:
mdlfilename:
--------------------------------
```

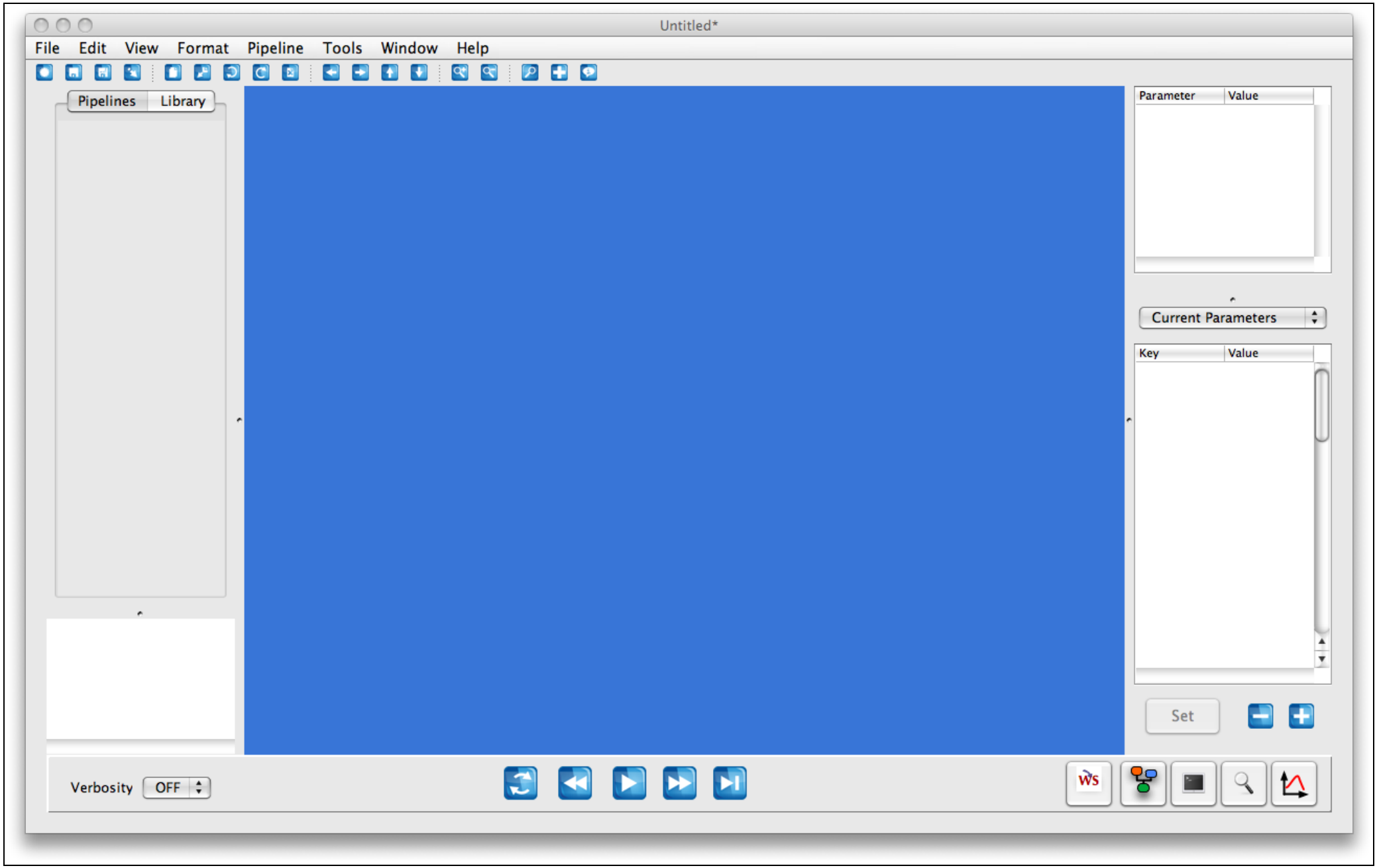
Now you can see that your AO has some data. The data is of type cdata (more on that later), it has no Y units, and it contains a single value, 1.

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
```

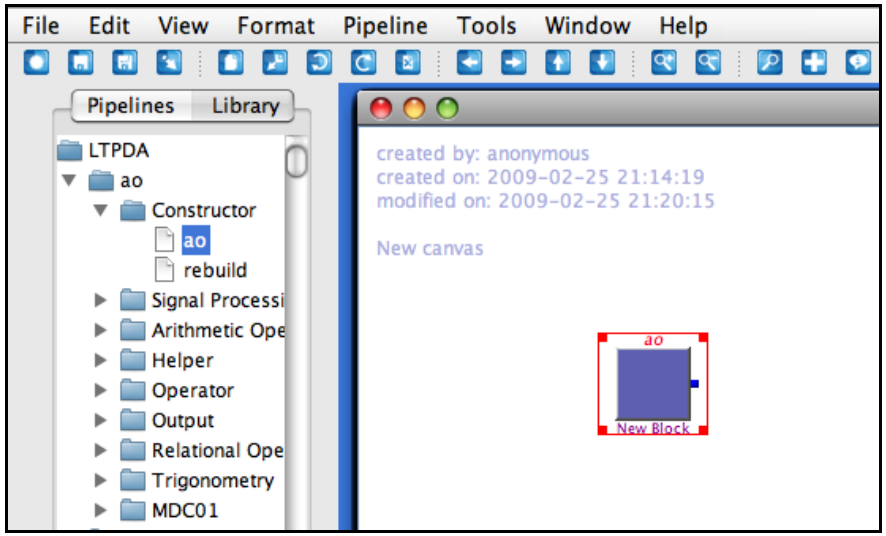
You should see a window like the one below:



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.

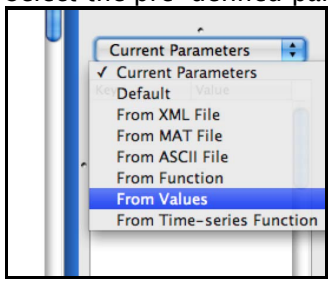
To construct the simple AO as we did above on the terminal, first create an empty pipeline in the workbench by going to "Pipeline->New Pipeline" or hit `ctrl-n` (`cmd-n` on OS X). The you can drag an AO constructor block from the Library on the left. You can also double click on the block in the library to add it to the canvas.



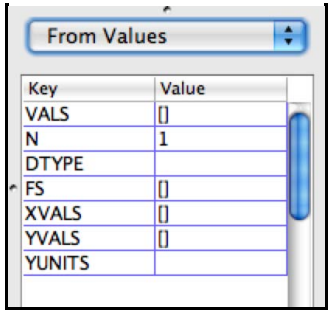
Blocks can be added to the canvas using the "Quick Block" dialog. Hit `ctrl-b` (`cmd-b` on OS X) on the canvas to open the quick block dialog. Begin typing to find the block you want (e.g. 'a' 'o') the hit `enter` to add that block to the canvas. Hit `enter` to add multiple blocks the same. Hit `escape` 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

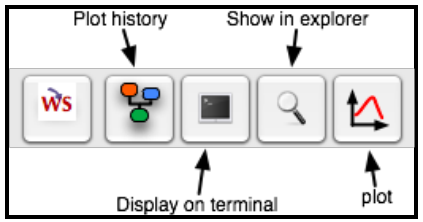


3. You should see the parameter list like



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. Enter a new value (any MATLAB expression) in the dialog box and click the OK button
8. 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
description:
       data: 1
             -------- cdata 01 ------------
                  y:  [1x1], double
             yunits:  []
             -----------------------------
```

```
       hist:  ltpda_uoh / setName / $Id: ltpda_training_topic_1_2_content.html,v 1.4 2009/03/08 11:46:00 hewitson Exp $
  mfilename:
mdlfilename:
---------------------------------
```

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

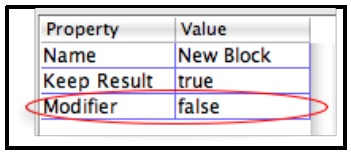
```
b = setName(a, 'Bob')
```

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.

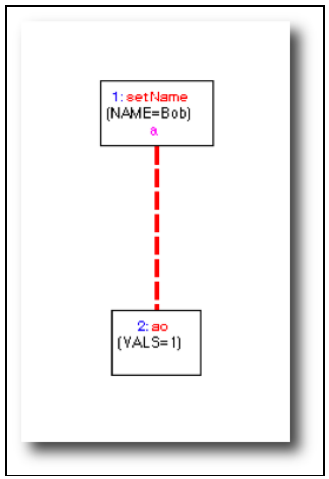
You can do the same on the workbench by using a `setName` block. To use the 'modifier' behaviour, you set the block to be a modifier in the 'block properties' table:



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

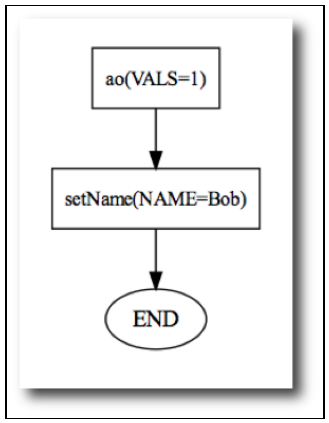
We can now look at the history of this object (in case our memory is really short). The history can be viewed in different ways. For short history trees, the easiest is to use the MATLAB-based history plotter built in to LTPDA. In the MATLAB terminal, type plot(a.hist) and hit return. You should get 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.



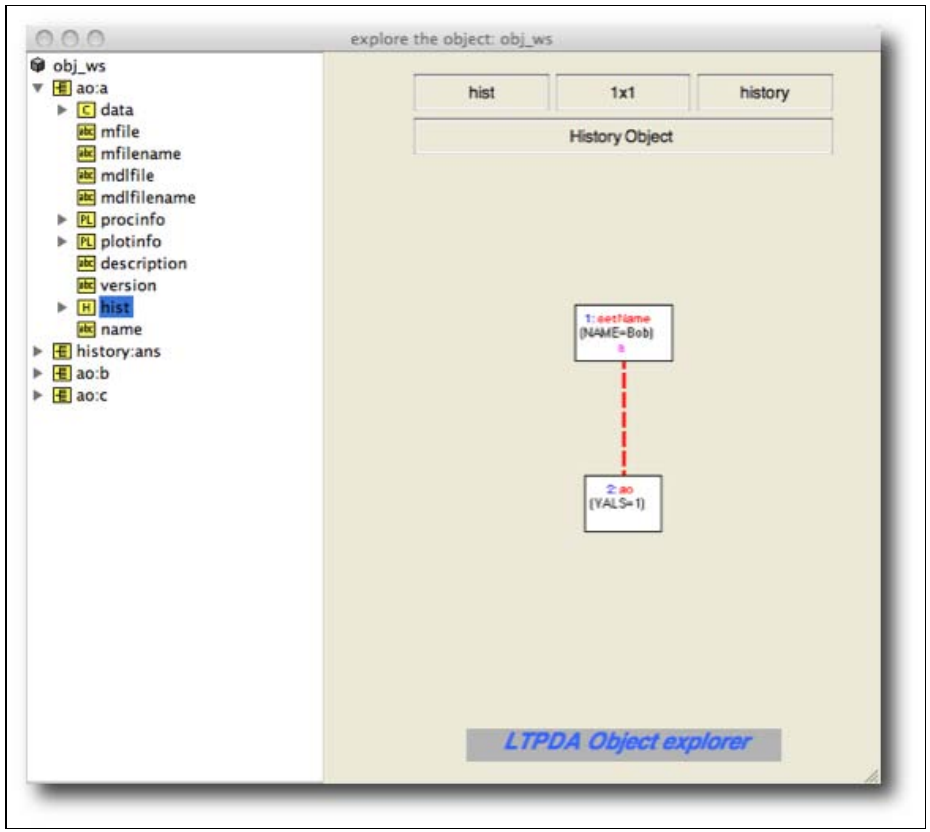
When you click on the 'display history' button on the workbench, a filename created from combining the pipeline name and the block name is used in the call to `dotview`. You will find the PDF file in the current MATLAB working directory.

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 `y`, and the units of the data in `yunits`.
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
description:
       data: 1
             -------- cdata 01 ------------
                  y:  [1x1], double
             yunits:  [km]
             -----------------------------

       hist:  ao / setYunits / $Id: ltpda_training_topic_1_2_content.html,v 1.4 2009/03/08 11:46:00 hewitson Exp $
  mfilename:
mdlfilename:
---------------------------------
```

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`.)

Introducing Analysis Objects

Making a time-series AO

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 va0lues and fs
M: running ao/display
----------- ao 01: a -----------

       name:  none
description:
       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
             xunits:  [s]
             yunits:  []
              nsecs:  2.5
                 t0:  1970-01-01 00:00:00.000
             -------------------------------

       hist:  ao / ao / $Id: ltpda_training_topic_1_3_content.html,v 1.5 2009/03/08 11:46:00
hewitson Exp $-->$Id: ltpda_training_topic_1_3_content.html,v 1.5 2009/03/08 11:46:00
hewitson Exp $
  mfilename:
mdlfilename:
--------------------------------
```

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')
```

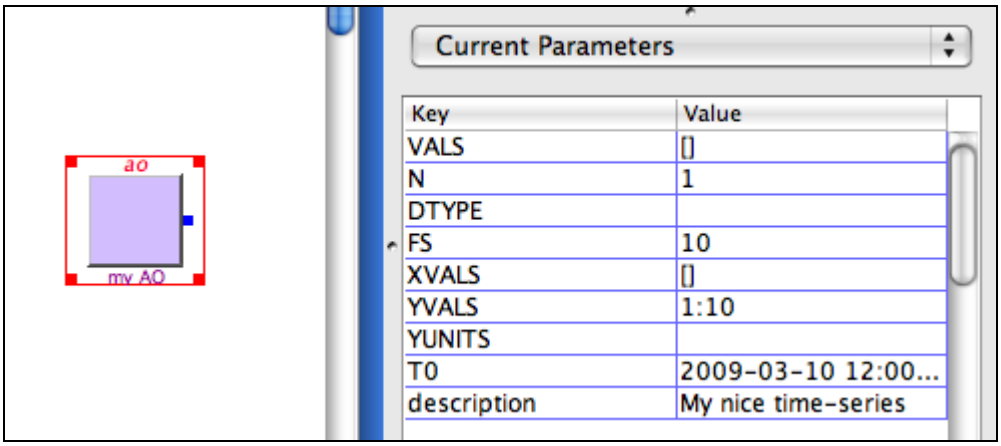
You can do all of this in one block on the workbench. To do that:

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 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., `10`. 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.
8. 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.)
9. 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
description:
       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
             xunits:  [s]
             yunits:  []
              nsecs:  1000
                 t0:  1970-01-01 00:00:00.000
             -------------------------------

       hist:  ao / ao / $Id: ltpda_training_topic_1_3_content.html,v 1.5 2009/03/08 11:46:00
hewitson Exp $-->$Id: ltpda_training_topic_1_3_content.html,v 1.5 2009/03/08 11:46:00
hewitson Exp $
  mfilename:
mdlfilename:
-------------------------------
```

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)
```

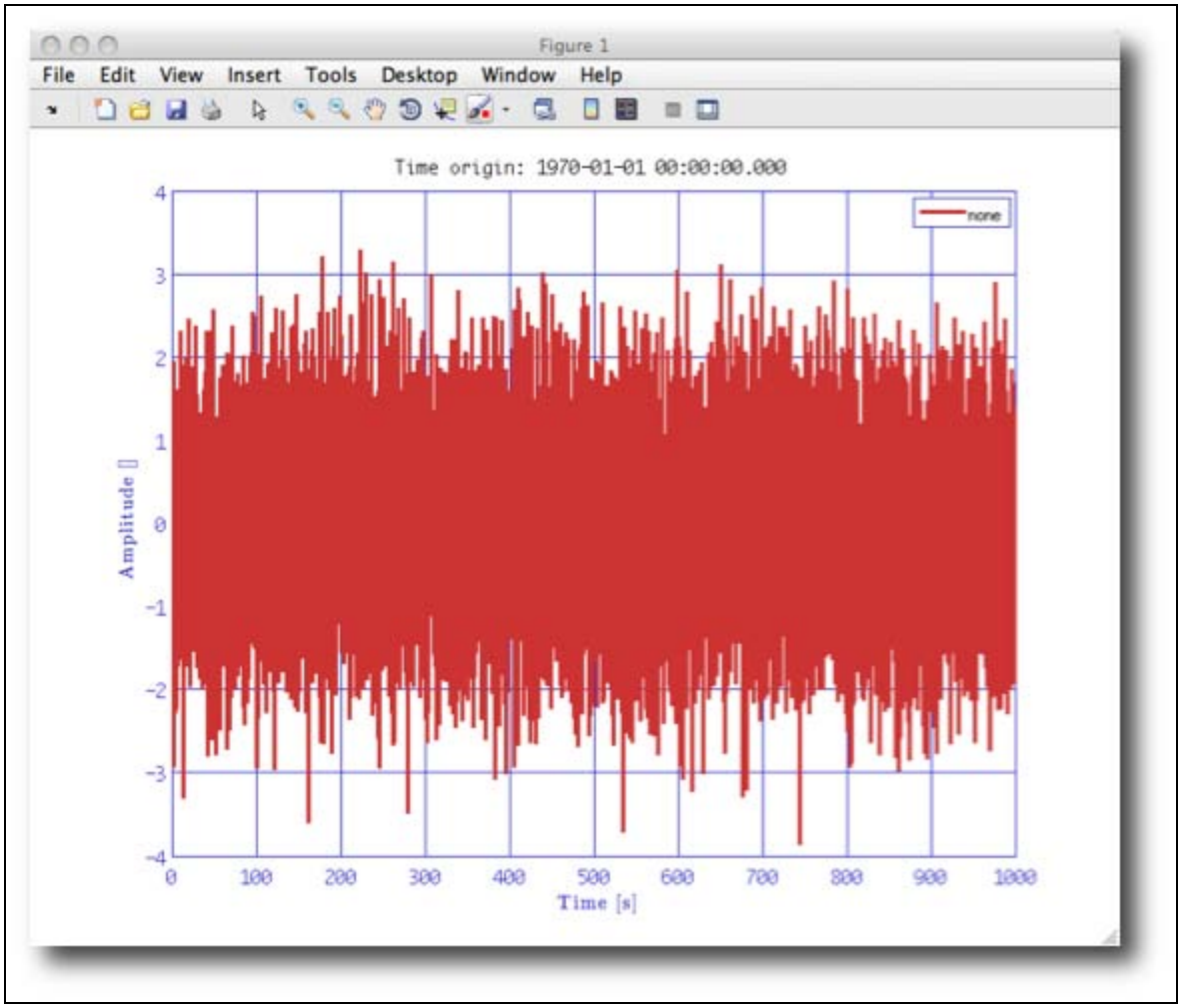
Here we have made two AOs with different random white-noise data vectors.

Digression: plotting the data

To plot the data in the AO, you can use the intelligent plotting method, `iplot`. For example, type in the MATLAB terminal:

```
a1.iplot
```

and you should see a plot like the one below.



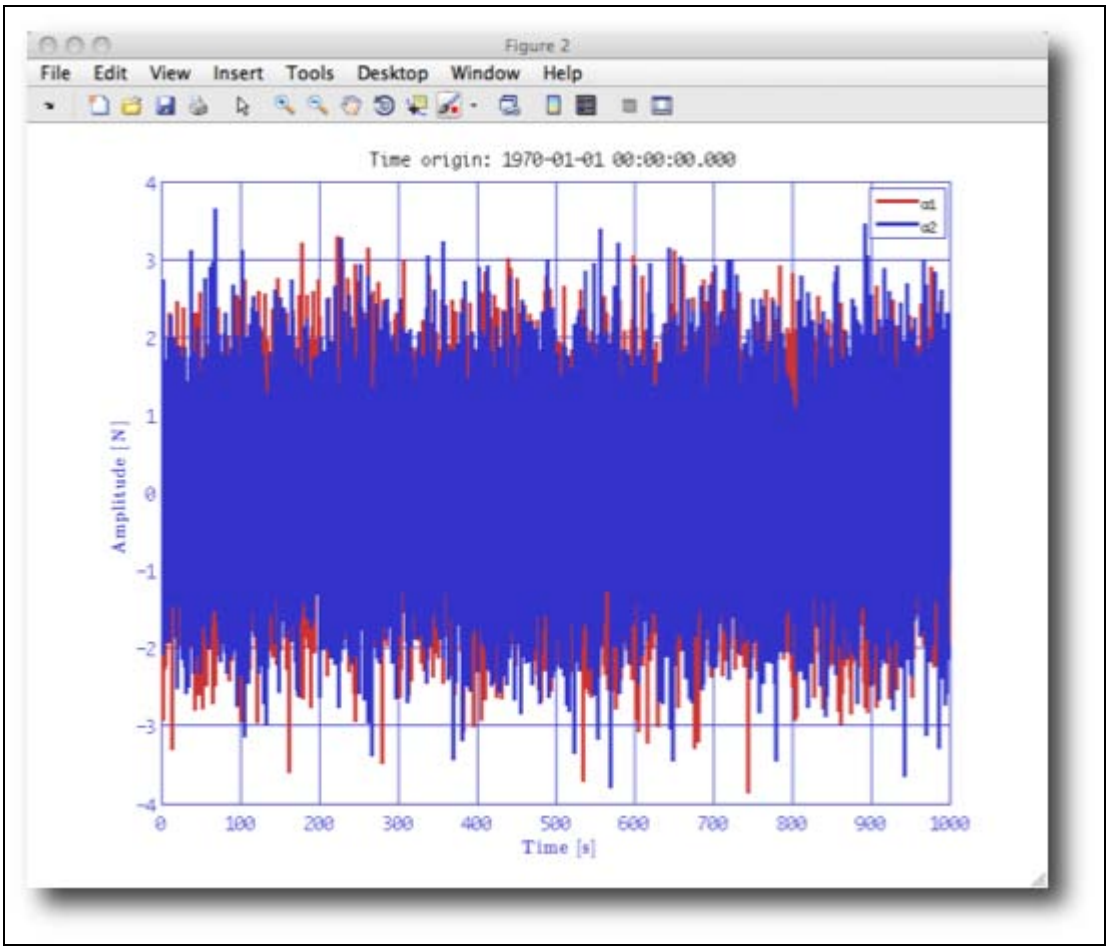
We can make a more interesting plot if we first specify some of the properties of the AOs. For example, type the following commands to set the names and Y units of the two AOs we made earlier:

```
a1.setName
a2.setName
setYunits(a1,a2,'N')
```

Now plot both time-series together with:

```
iplot(a1,a2)
```

and you shoud see a plot like the following:



Calling the `setName` method with no input argument causes the AO to be named with the variable name.

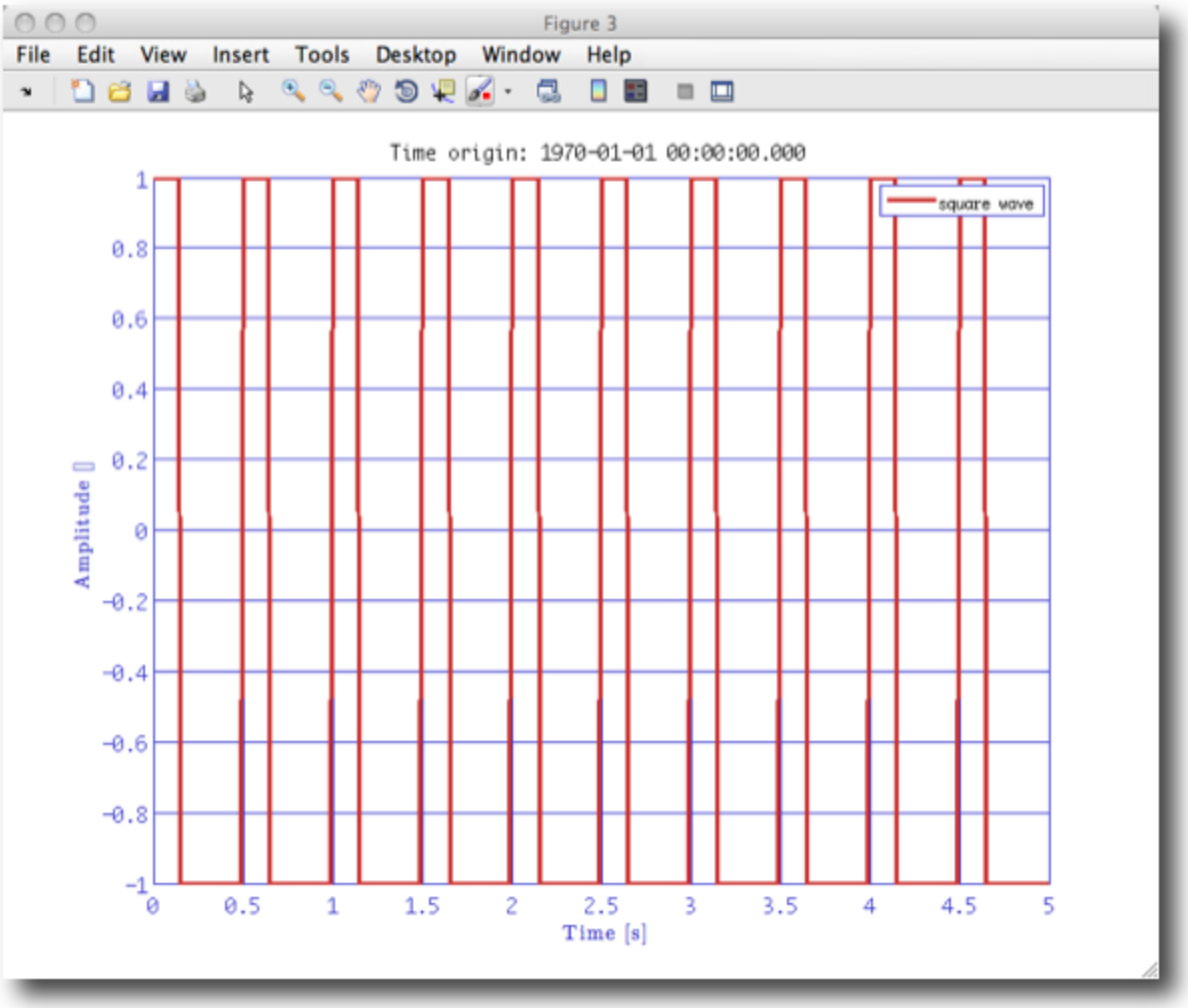
`iplot` has many configurable parameters which are (mostly) documented in the help.

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 eave
                    '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 gauspuls)
             '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.
```

Making AOs

Basic math with AOs

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,

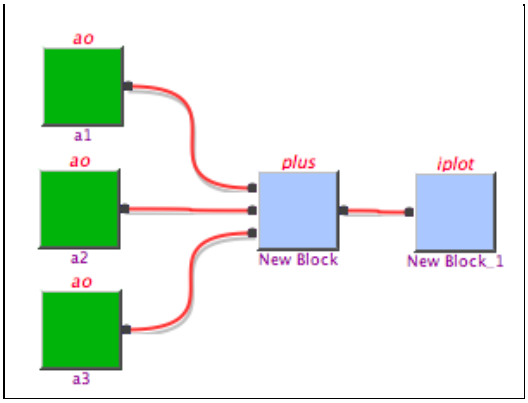
```
a = [ao(1) ao(2) ao(3)]
b = [ao(4) ao(5)]
c = a + b
```

will give an error.

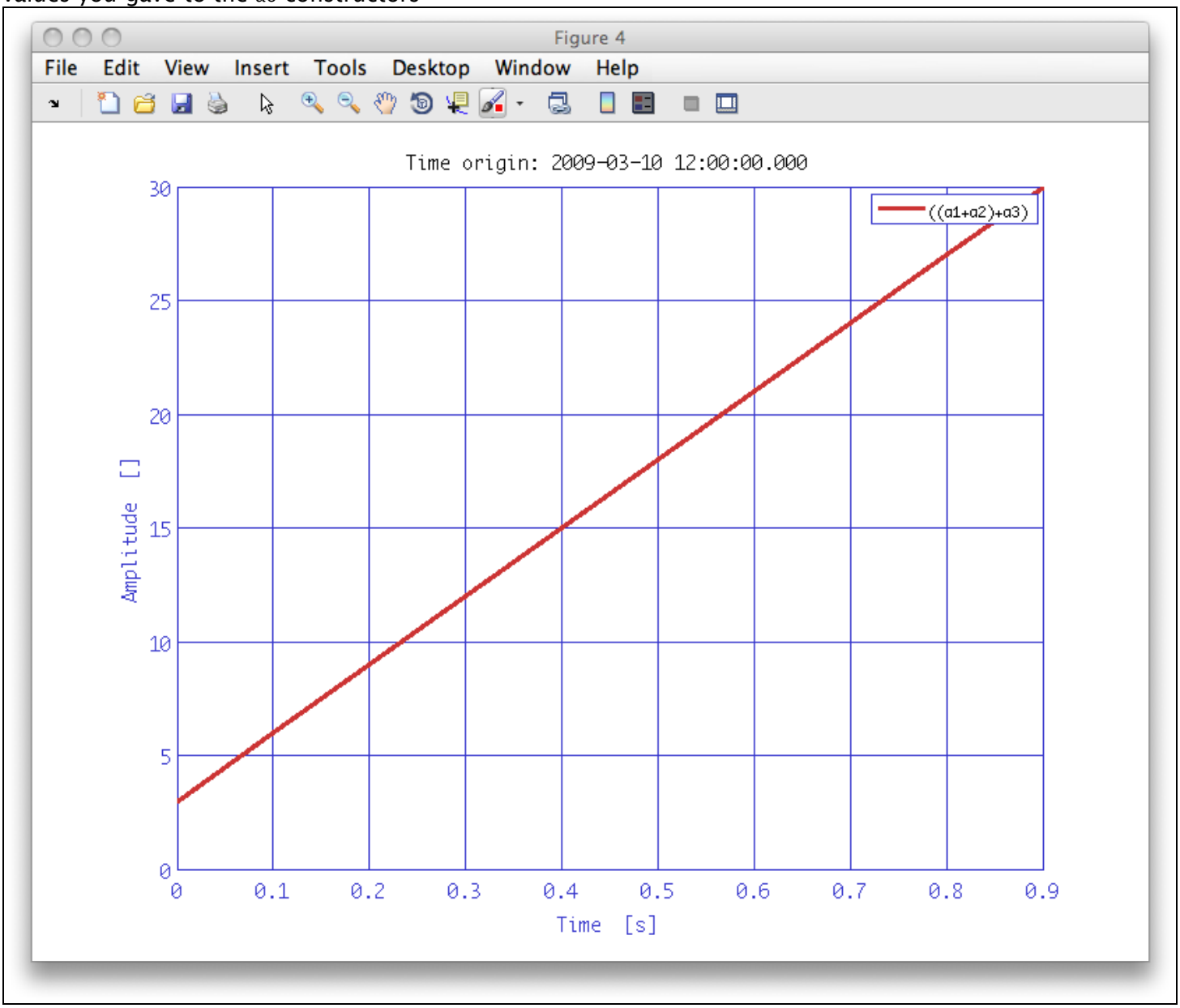
You can do all of this on the workbench as well, of course.

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 Values"
4. Duplicate the block two more times (`ctrl/cmd-d`)
5. Enter a single number for each, or a vector the same length for each
6. Drag a `plus` block to the canvas
7. 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
8. 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` 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.
9. 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:



10. 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



Making a time-series AO

Saving and loading AOs

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:

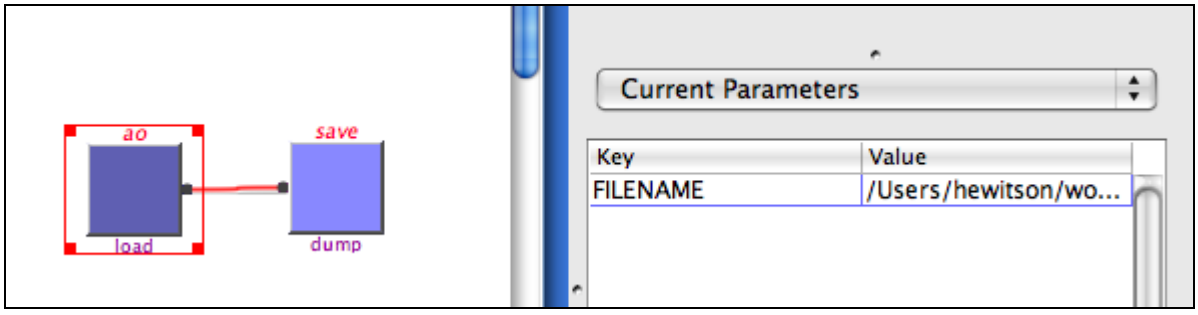
```
a1 = ao('foo.xml');
a2 = ao('foo.mat');
```

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
description:
       data: (0,0.454992359600435) (0.1,0.307272609435985) (0.2,0.691315196578838) (0.3,-
0.713845553902851) (0.4,0.0136718022334676) ...
             -------- tsdata 01 ------------

                 fs:  10
                  x:  [1000 1], double
                  y:  [1000 1], double
             xunits:  [s]
             yunits:  []
              nsecs:  100
                 t0:  1970-01-01 00:00:00.000
             -------------------------------

       hist:  ao / ao / $Id: ltpda_training_topic_1_6_content.html,v 1.3 2009/03/08 11:46:00
hewitson Exp $-->$Id: ltpda_training_topic_1_6_content.html,v 1.3 2009/03/08 11:46:00
hewitson Exp $
  mfilename:
mdlfilename:
------------------------------------------------------------
```

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 randon 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.iplot(plist('XRanges', [0 2]))
```

Saving and loading AOs Writing LTPDA scripts

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 with multiple outputs

[t11, t21, t12, t22] = tfe(a1,a2);
Axx = t12 ./ t21;

%% Make TFE then index with ()

txx = tfe(a1,a2);
Axx = txx(1,2) ./ txx(2,1); % <-- BREAKS THE HISTORY

%% Use index method

txx = tfe(a1,a2);
Axx = txx.index(1,2) ./ txx.index(2,1);
```

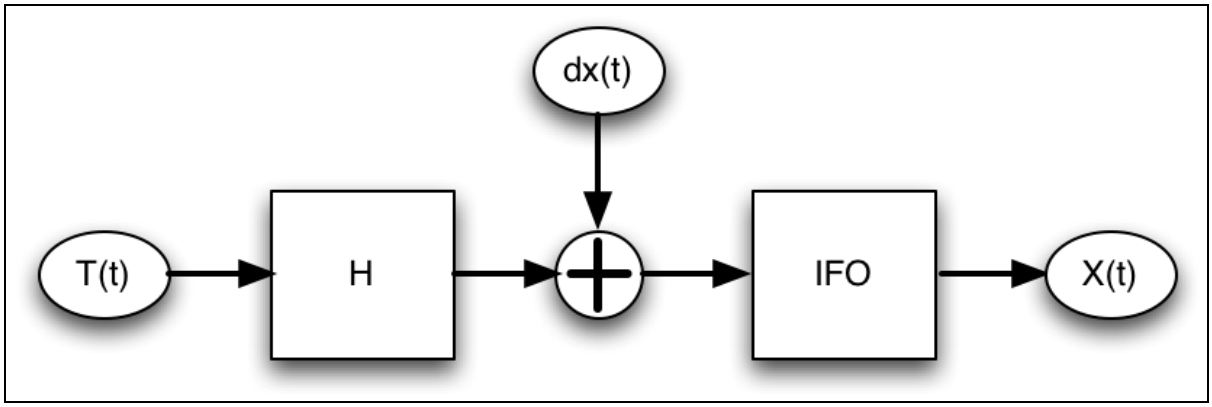
You can explore the commands used to rebuild the object by using the `type` method. Or you can plot the history as we did earlier. For example, try this command for each of the three cases above:

```
type(Axx)
```

You should see that in the second case, the history doesn't include the indexing of the output of the `tfe` method when doing the division. As a result, the output of the division will be a 4 element vector of AOs, not the single AO we were expecting.

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, `T(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 constuctor, 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

		this simple file format.
DESCRIPTION	'Interferometer data'	Set some text to the 'description' field of the AO.

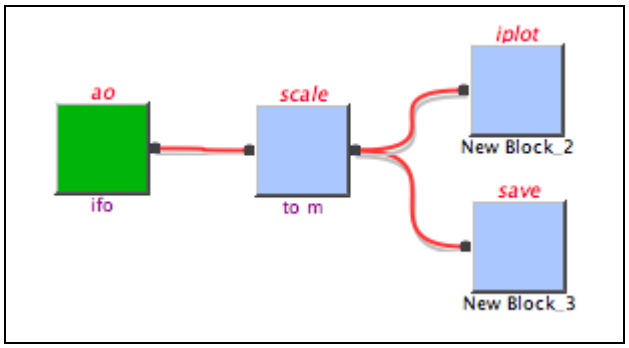
Once we've loaded the data we can calibrate it to displacement using the following equation:

$$Y(t)[\mathrm{m}] = X(t)\frac{\lambda}{2\pi}[\mathrm{rad}]$$

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 constuctor, 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.

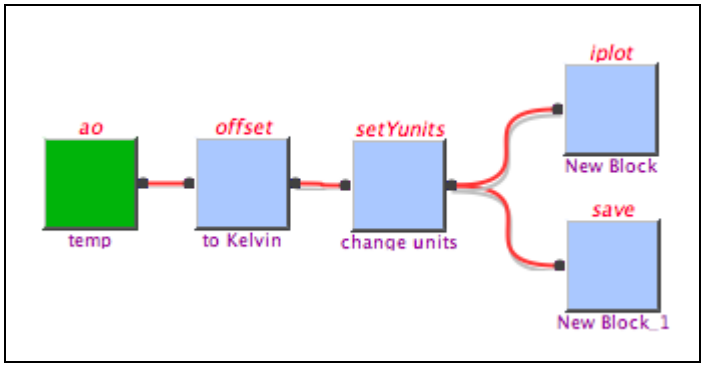
ROBUST	'no'	Use fast data reading for this simple file format.
DESCRIPTION	'Temperature data'	Set some text to the 'description' field of the AO.

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'))
```

Writing LTPDA scripts Topic 2 - Pre-processing of data

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.

IFO/Temperature Example - Introduction Downsampling a time-series AO

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.

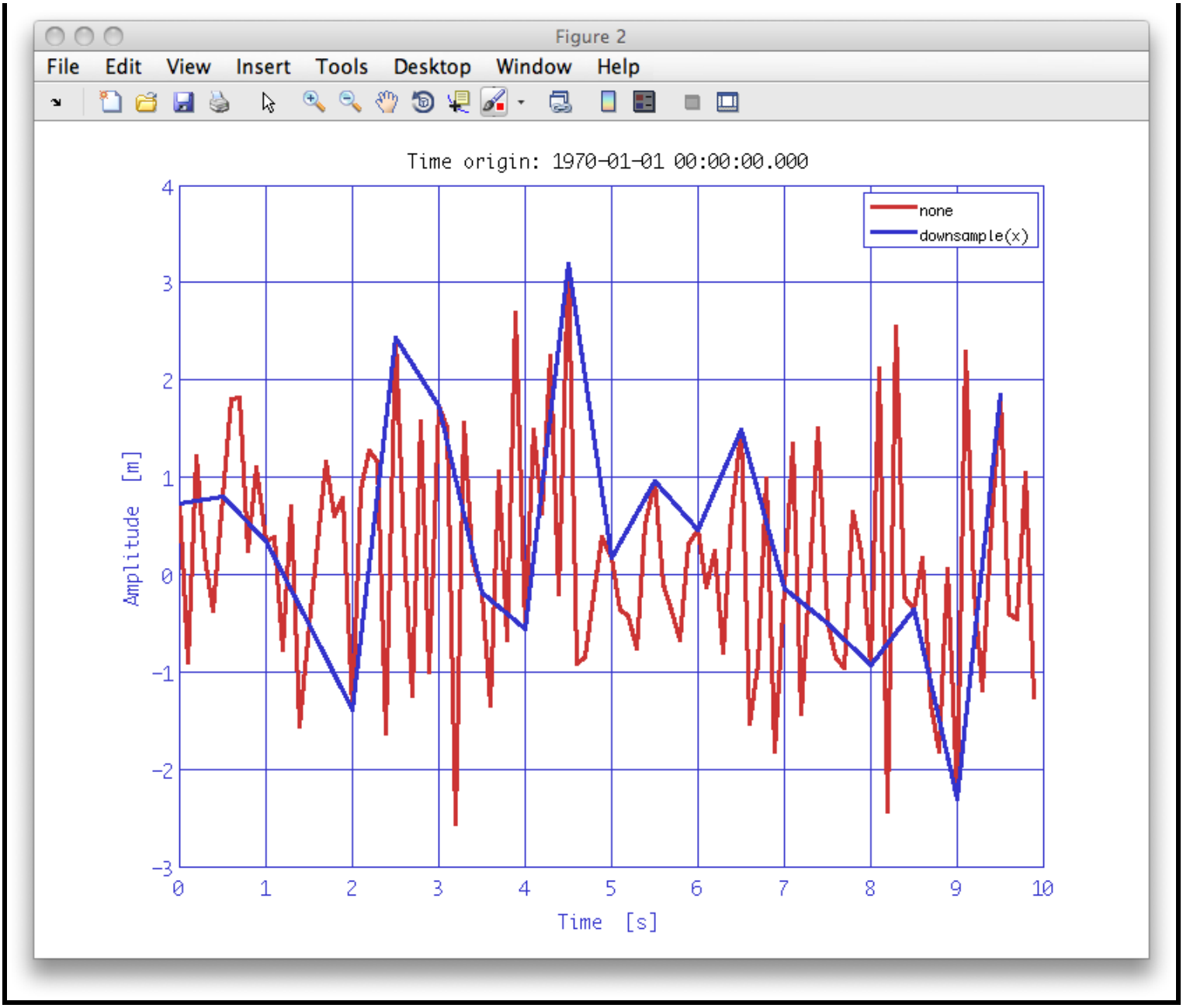
Example 1

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.

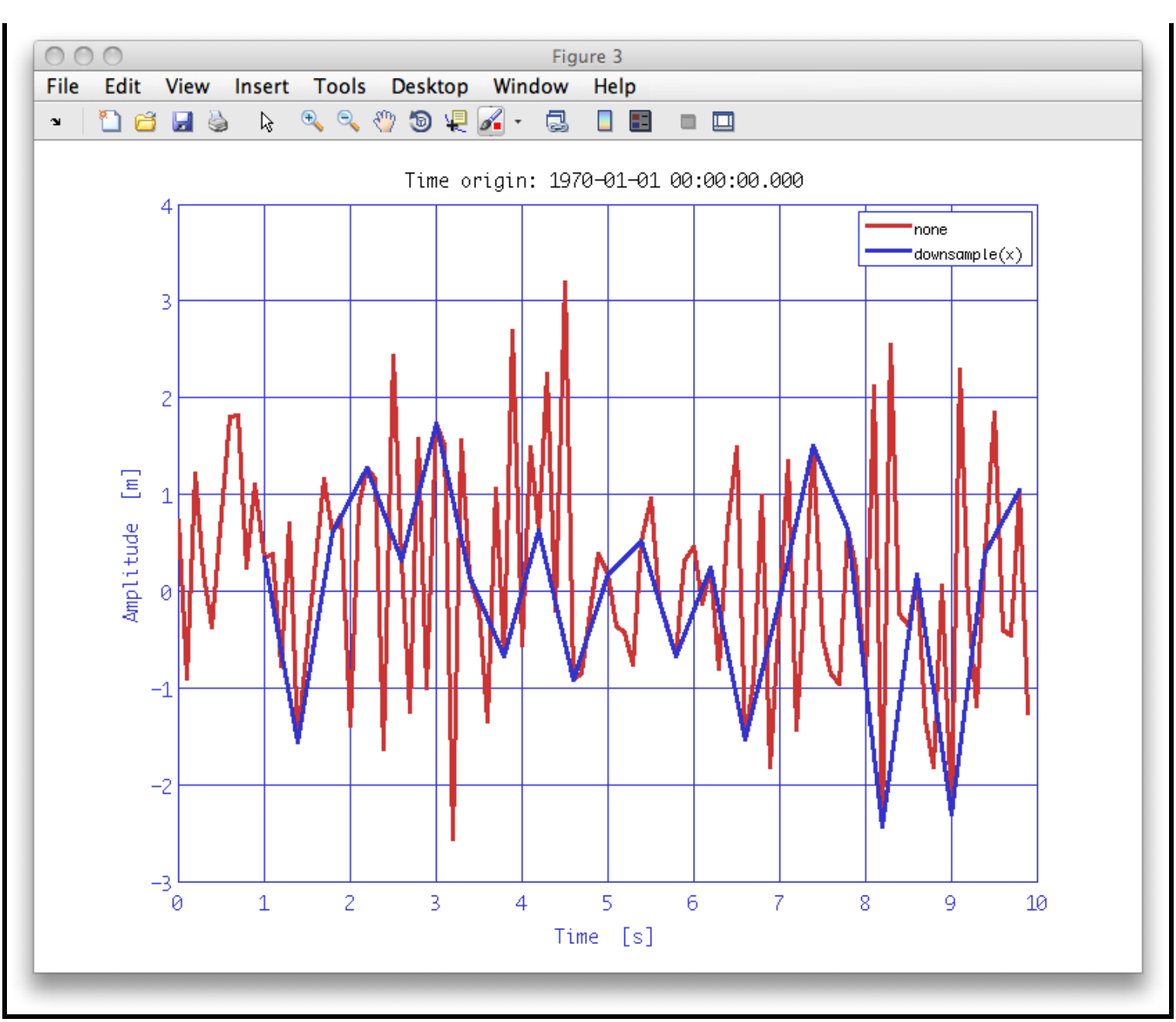
```
pl_down = plist('factor', 5); % add the decimation factor
x_down  = downsample(x, pl_down); % downsample the input AO, x
iplot(x, x_down) % plot original,x, and decimated,x_down, AOs
```



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
iplot(x, x_downoff) % plot original,x, and decimated,x_downoff, AOs
```



Topic 2 - Pre-processing of data

Upsampling a time-series AO

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 offet. The value must be between 0 and N-1.

Example 1

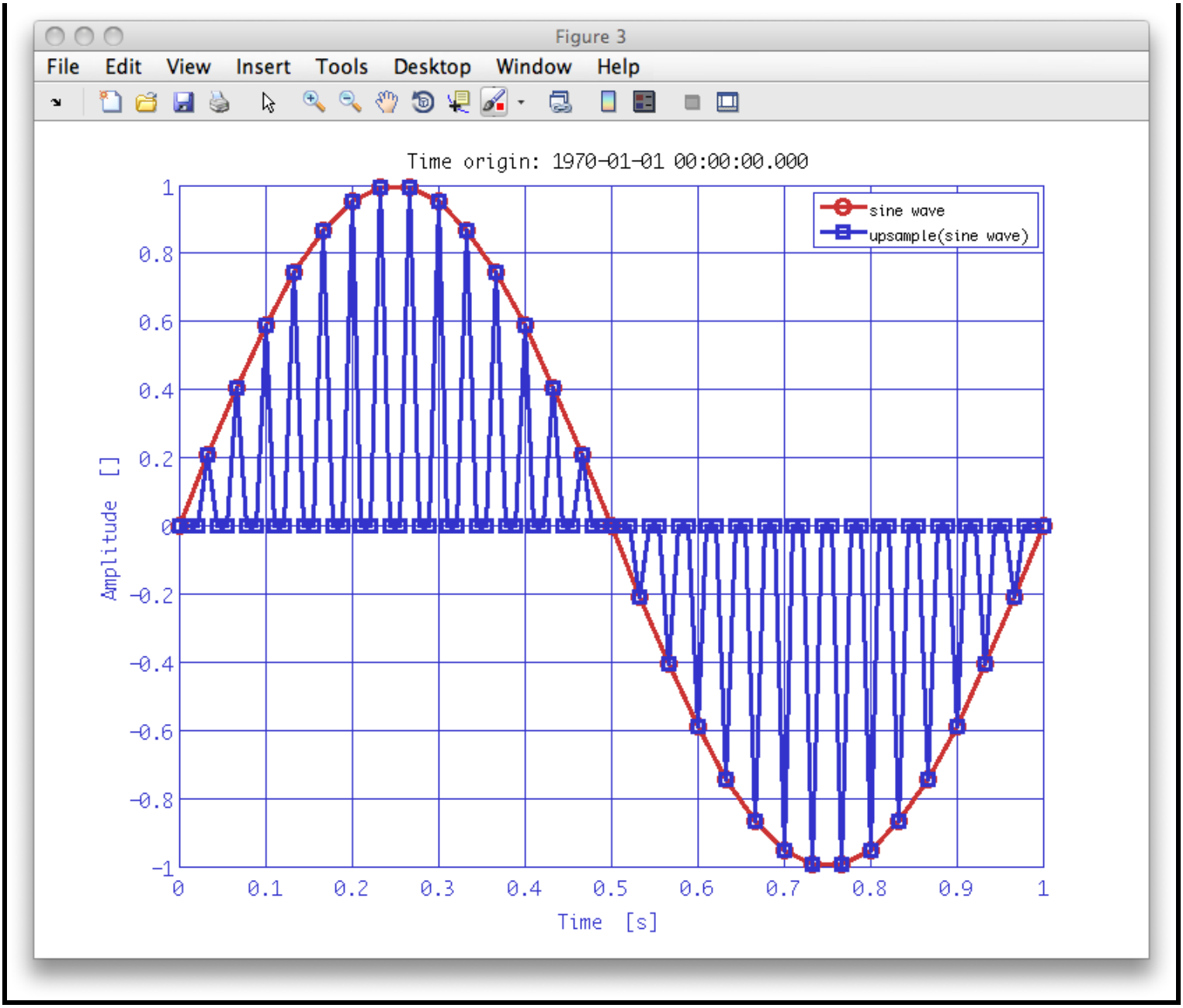
We will upsample a sine-wave by a factor of 3 with no initial phase offset.

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)
iplot(x, x_up, plist('XRanges', [0 1], 'Markers', {'o', 's'})) % plot original and upsampled data
```

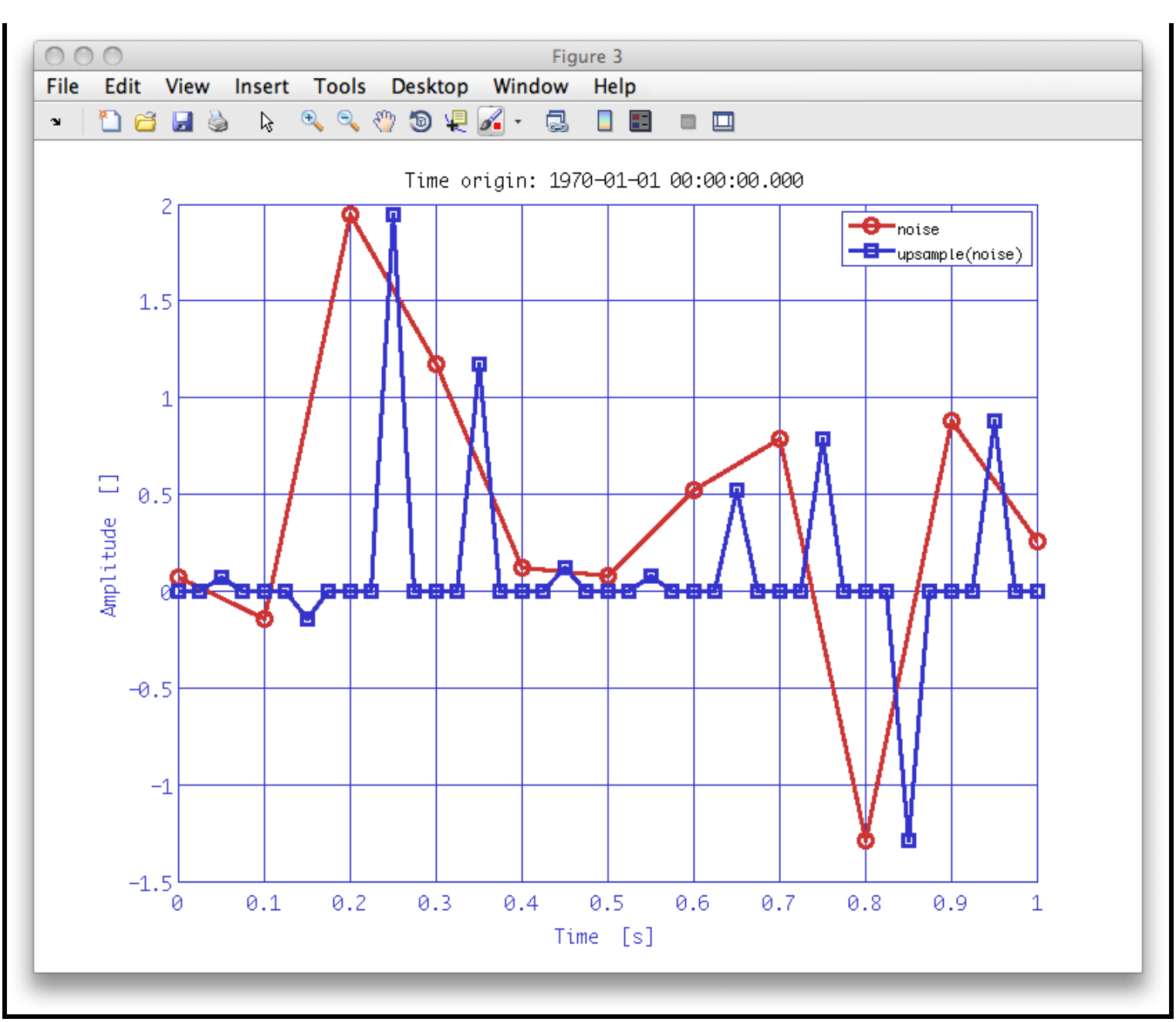


Example 2

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)
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

Resampling a time-series AO

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 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.

In order to use this method,

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

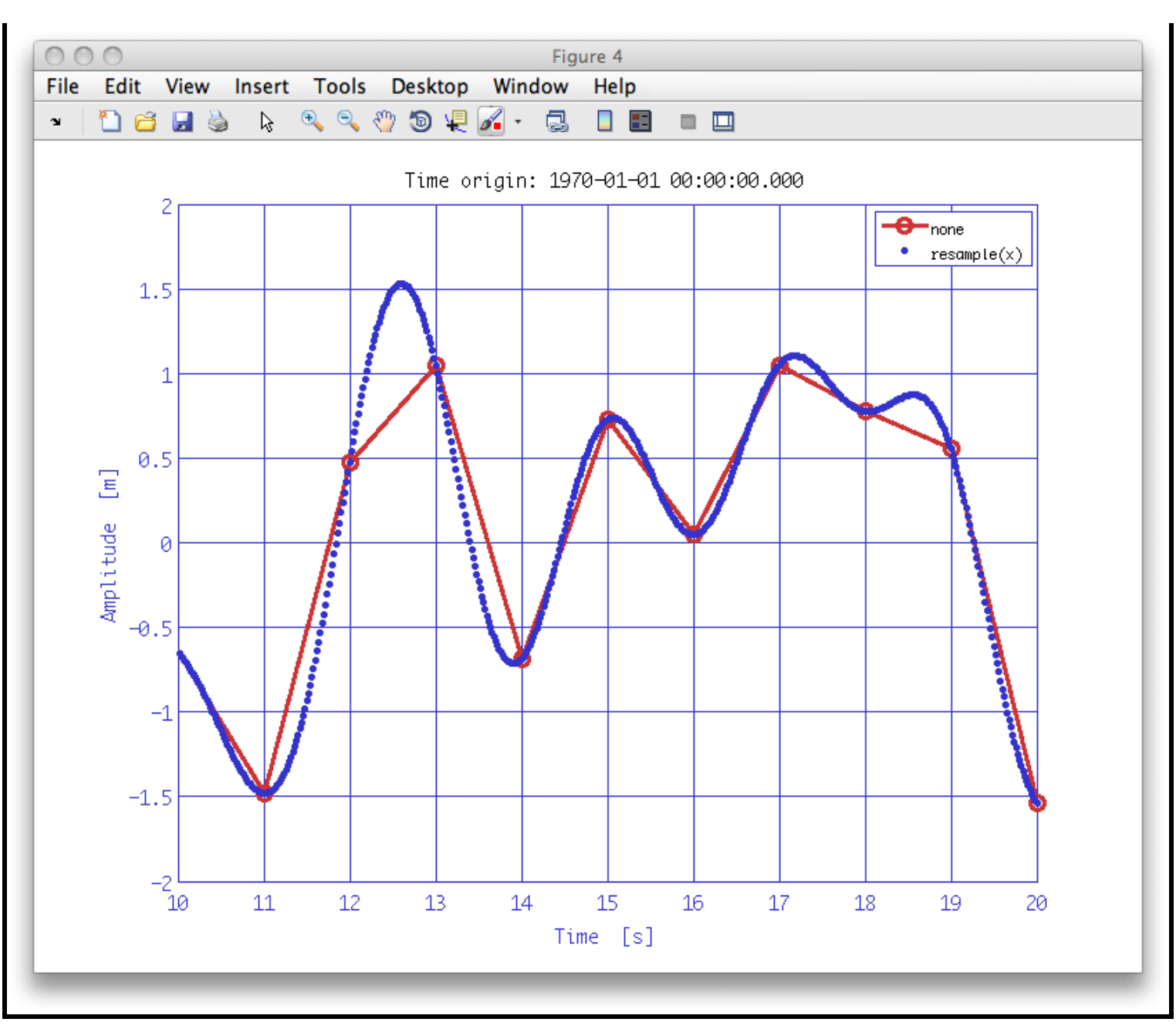
With the following parameter:

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)
iplot(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

Interpolation of a time-series AO

The ao has a method for interpolating data using different forms of interpolation. This method is called ao/interp.

To configure interp, use the following parameters:

Key	Description
VERTICES	A new set of vertices (relative to the t0) 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 singal on to a new time-grid. The result will be to increase the sample rate by a factor 2.

First we create an AO:

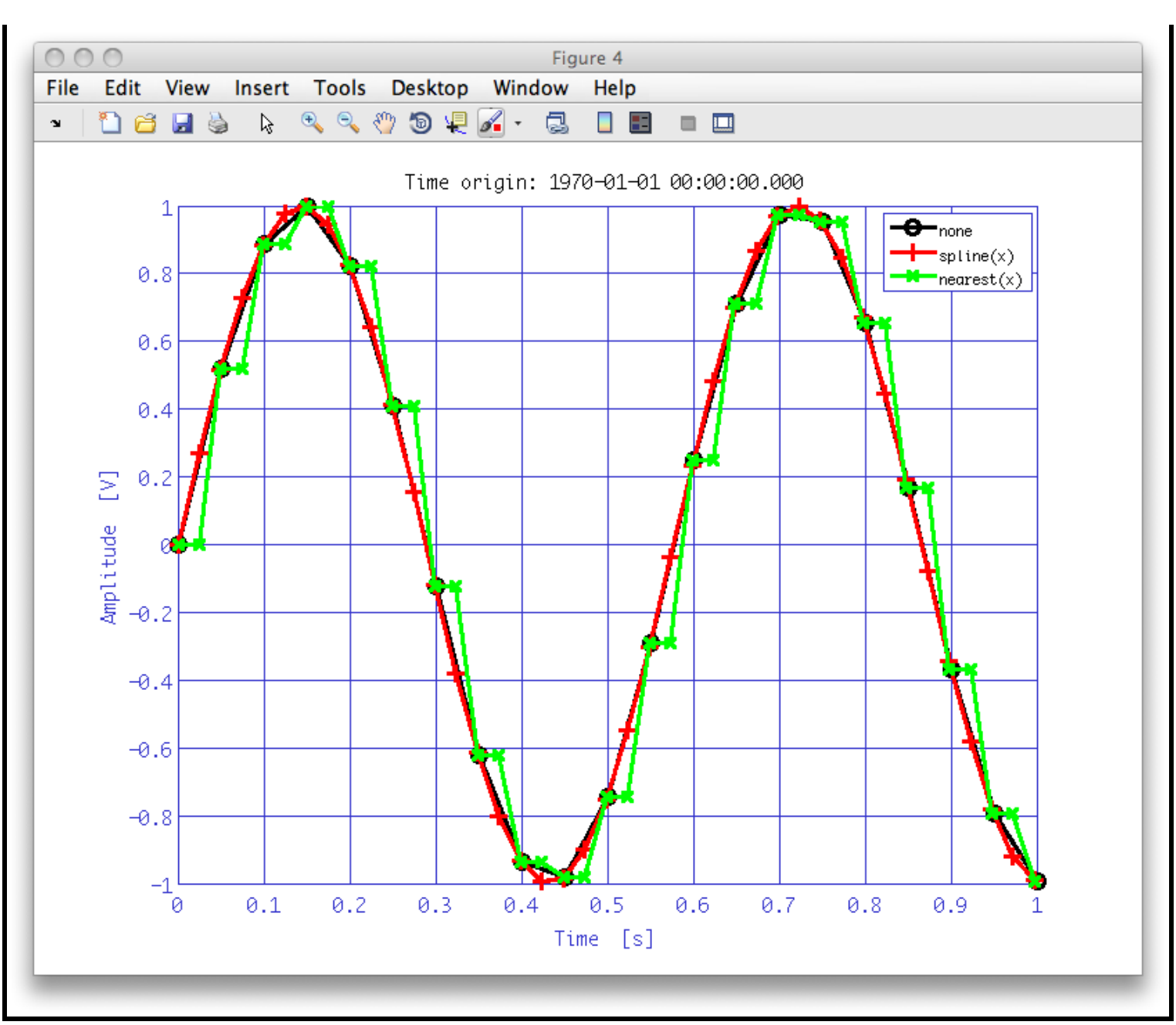
```
pl = plist('tsfcn','sin(2*pi*1.733*t)','fs',20,'nsecs',10,'yunits','V');
x  = ao(pl);
```

Then we create the new time-grid we want to resample on to.

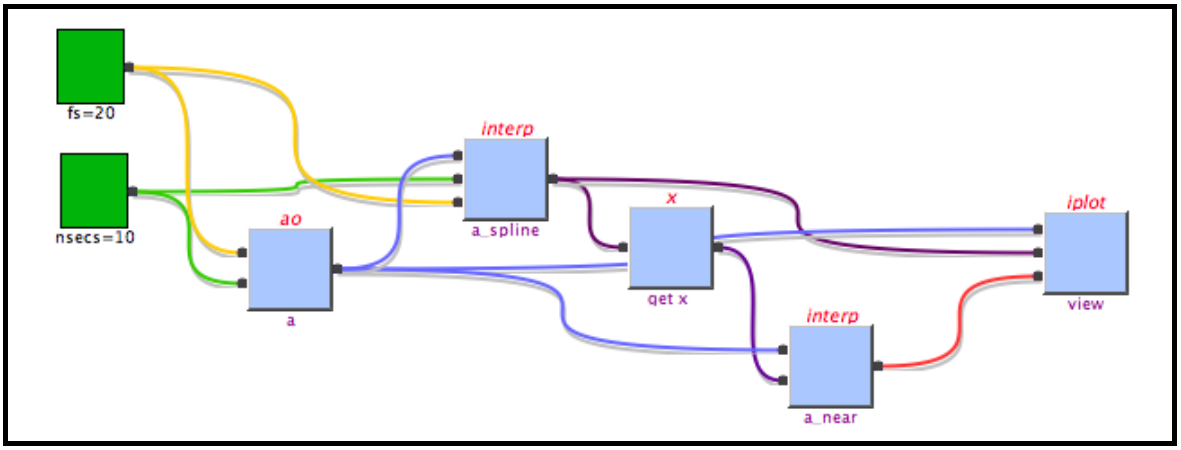
```
t    = linspace(0, x.data.nsecs - 1/x.data.fs, 2*len(x));
```

And finally we can apply our new time-grid to the data using interp:

```
pl_spline  = plist('vertices',t);
pl_nearest = plist('vertices',t,'method','nearest');
x_spline   = interp(x,pl_spline);
x_nearest  = interp(x,pl_nearest);
iplot(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'
    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 eminating 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 then 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 method `ao/detrend` 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 `polyfit` and `polyval` are used to construct the polynomial which is then subtracted from the data.

Example 1

In this example we will construct a time-series that consists of noise plus a known quadratic trend. We will then remove that trend using `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.

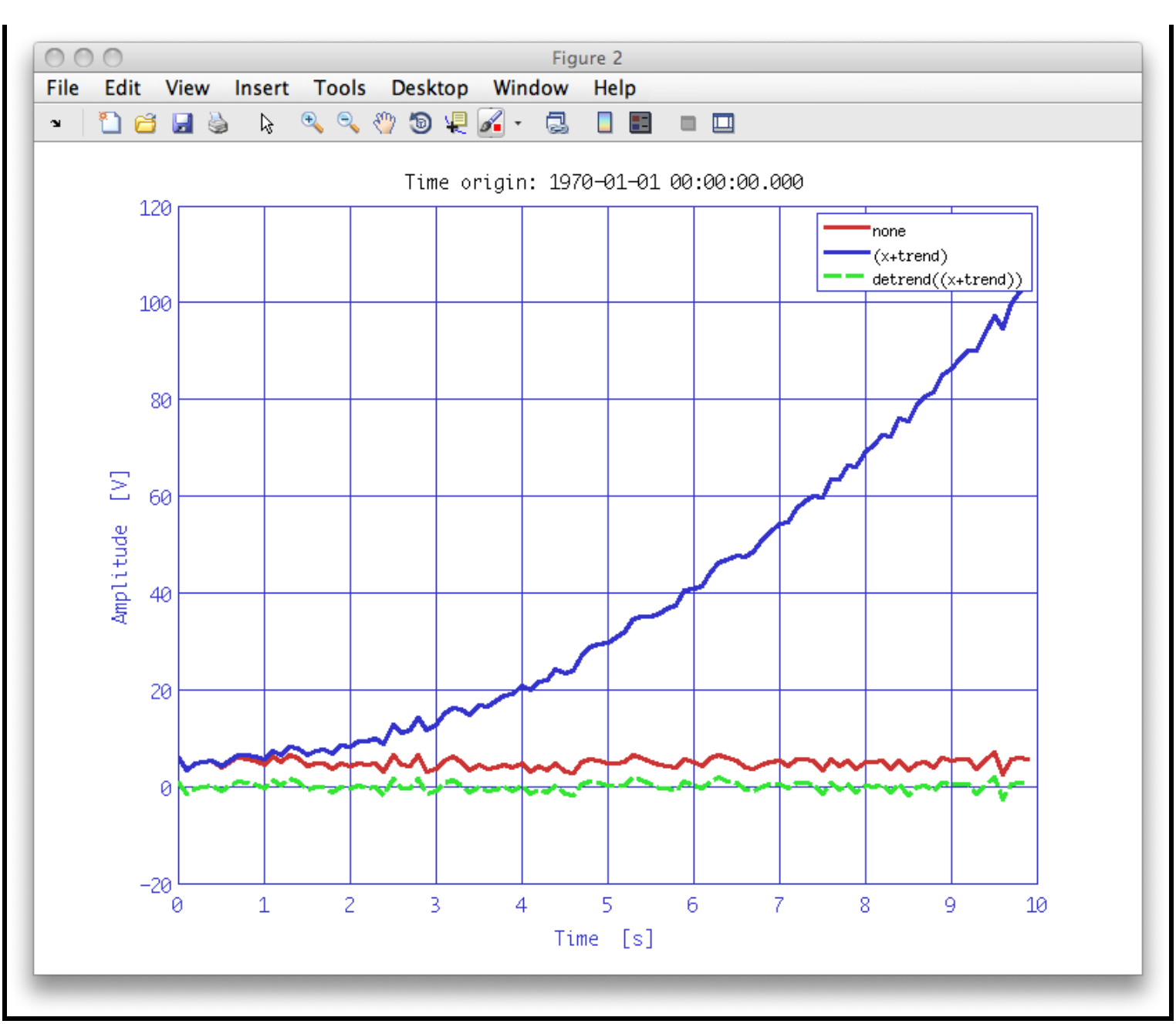
```
% Construct noise data stream
pl = plist('tsfcn','5+randn(size(t))','fs',10,'nsecs',10,'yunits','V');
x  = ao(pl);
% Construct a quadratic data series
pl_trend = plist('tsfcn','t.^2','fs', 10,'nsecs',10);
trend = ao(pl_trend);
% Add them together
fcn_trend = x+trend;
```

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);
iplot(x,fcn_trend,detr, plist('LineStyles', {'', '', '--'}));
```

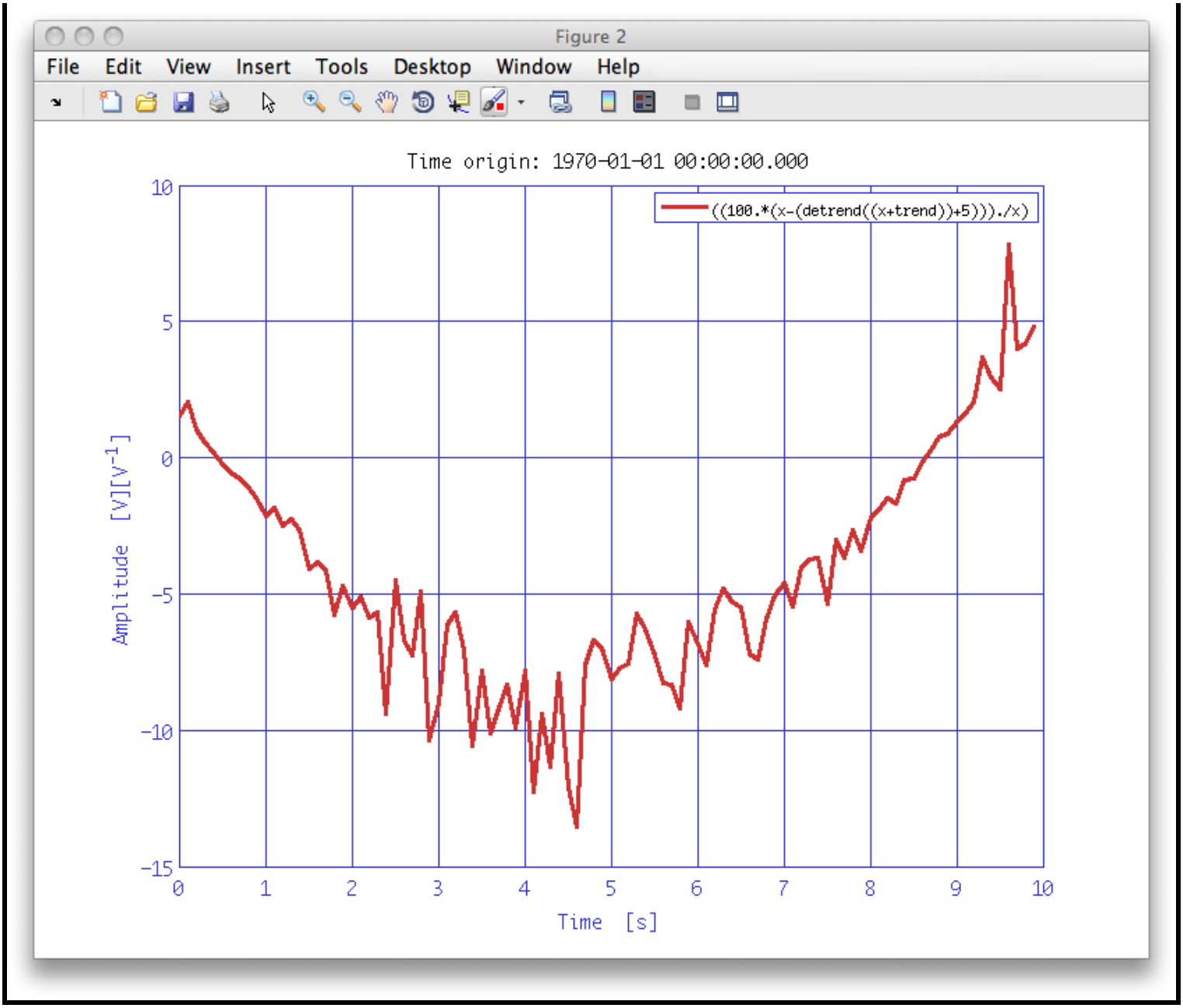
In the `plist` we specified 'LineStyles' as empty strings. These just serve as place holders and can be interpreted as "just to the default". If you want a data-series plotted with no line, then specify 'none', for example, `{'none', '-', '--'}`.



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.

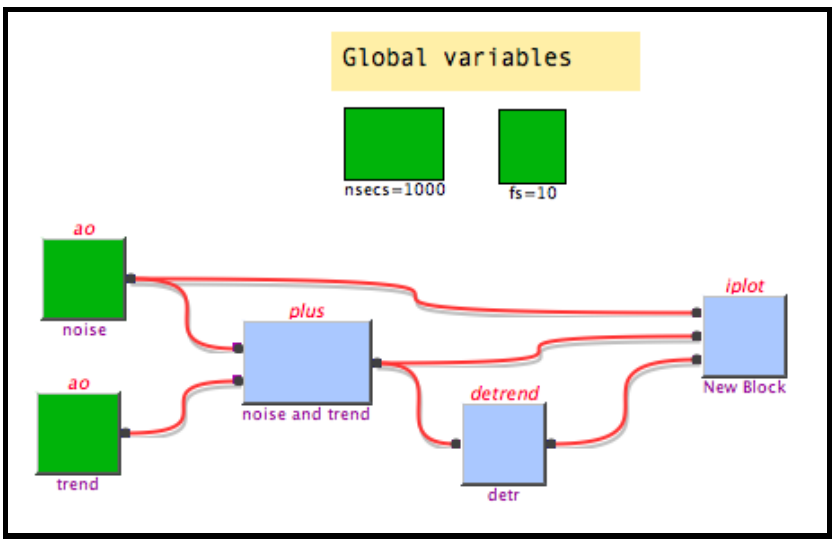
```
detr = detr+5;
diff = 100.*(x-detr)./x;
iplot(diff);
```

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.

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.

Interpolation of a time-series AO Whitening noise

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 b 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.

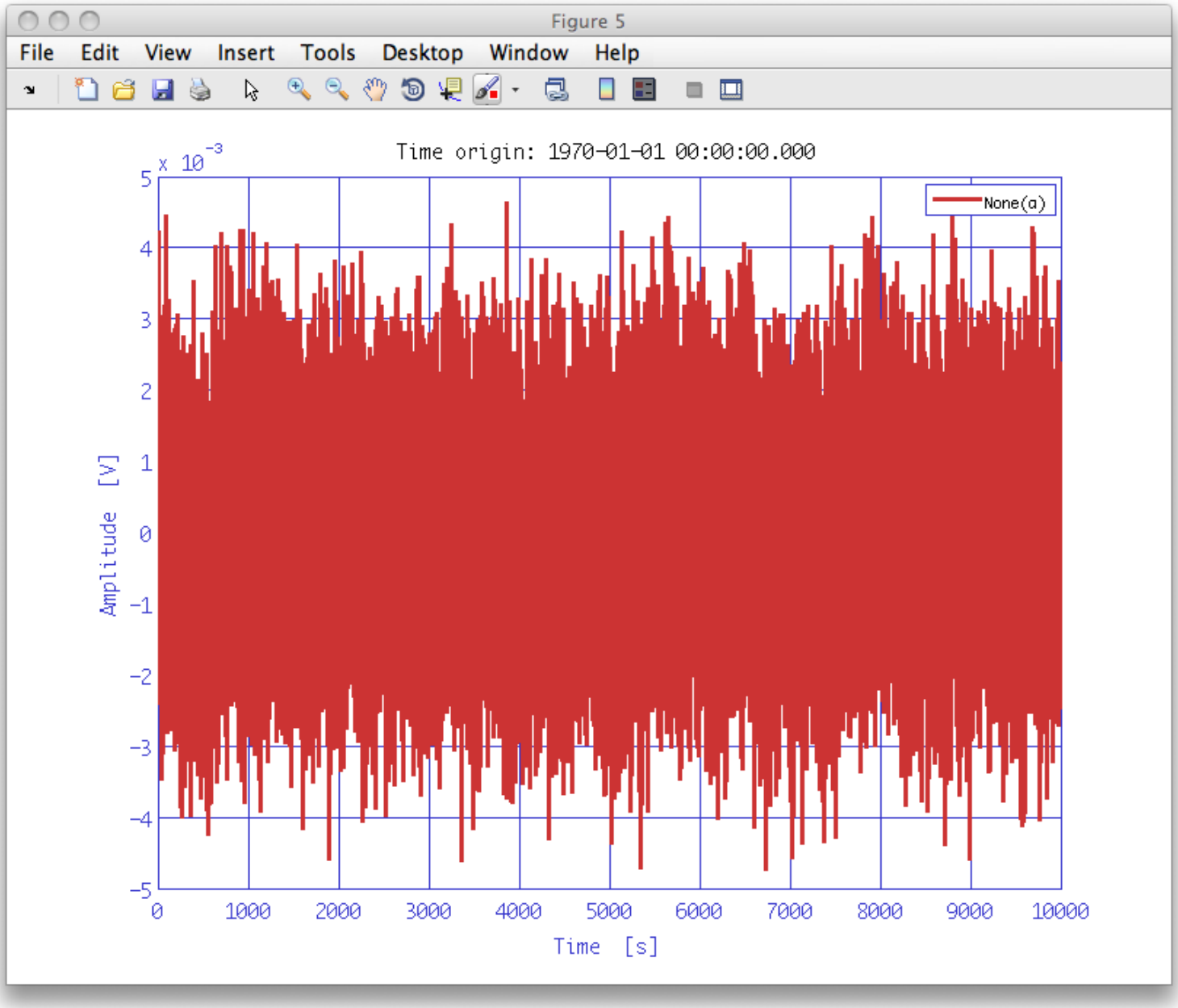
The data we will whiten can be found in your data packet in the 'topic2' sub-directory.

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 `iplot`.

```
>> iplot(a);
```



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);
```

No we can call the whitening function `whiten1D` with our input AO, `a` and the parameter list pl:

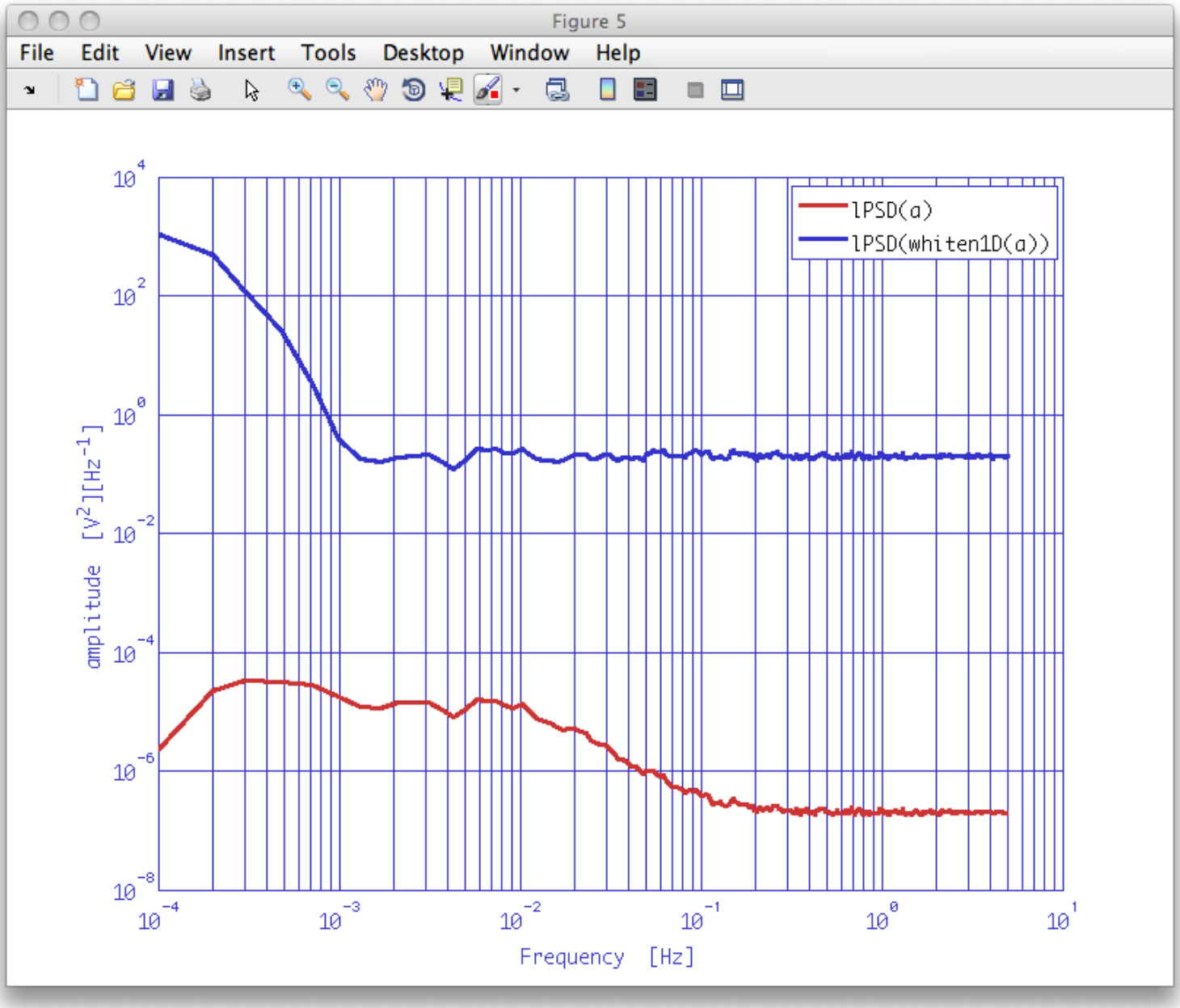
```
>> aw = whiten1D(a,pl);
```

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:
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

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','fs',100,'f',1,'yunits','V');
a  = ao(pl)
```

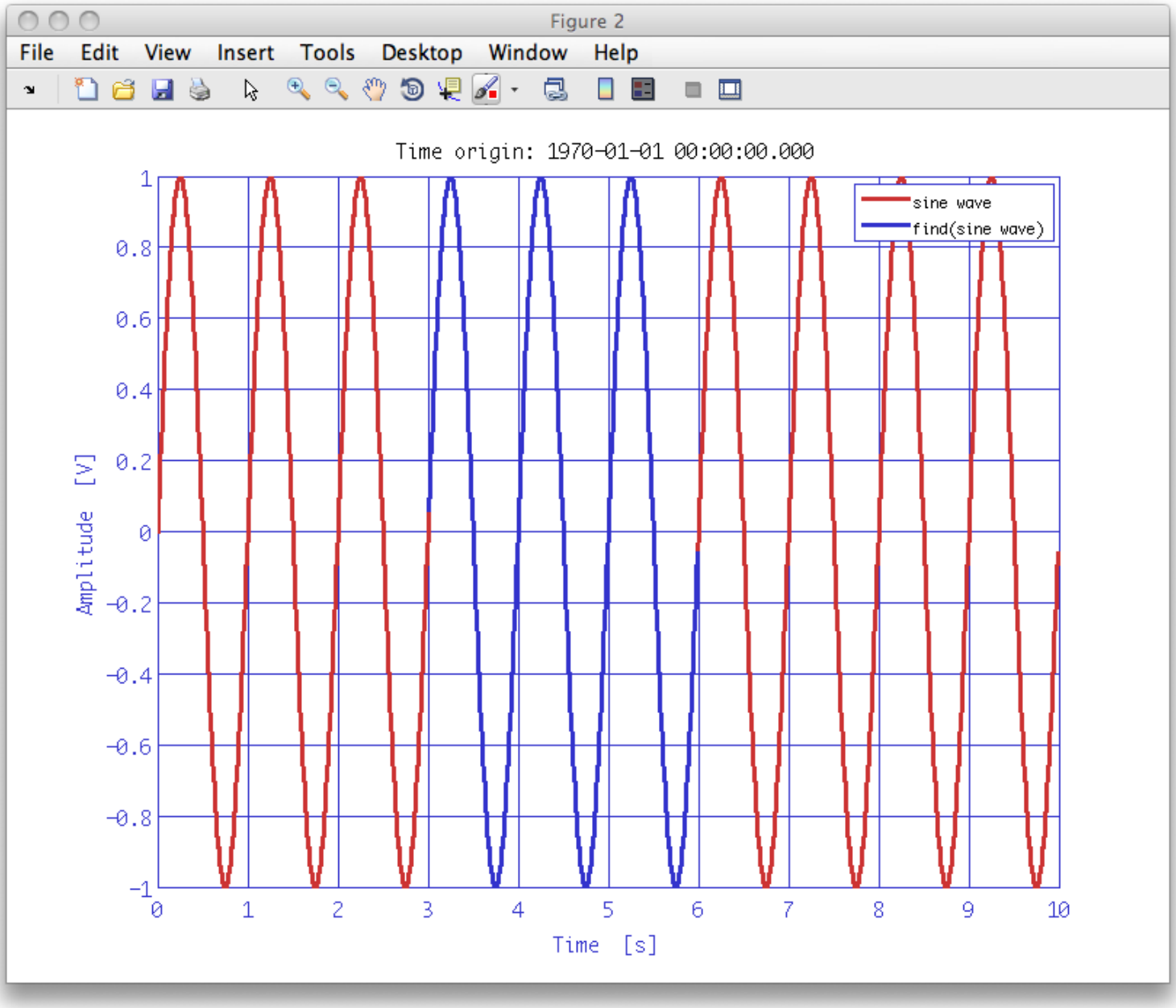
Example 1 - using `find`

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 -0.5 and 0.5. The following code does just that

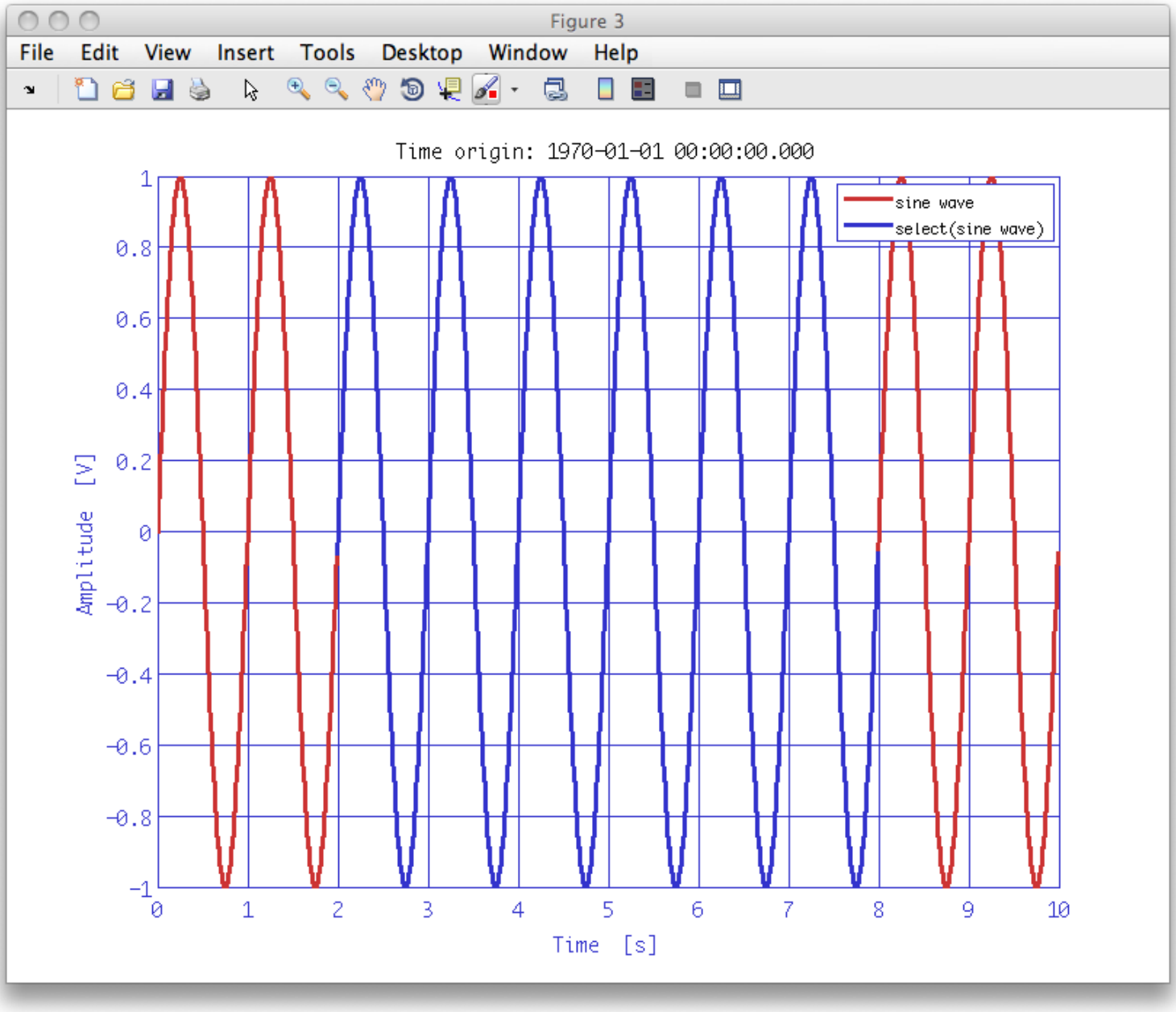
```
a_find = find(a, plist('query', 'x>3 & x<6'));
iplot(a, a_find)
```



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 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.

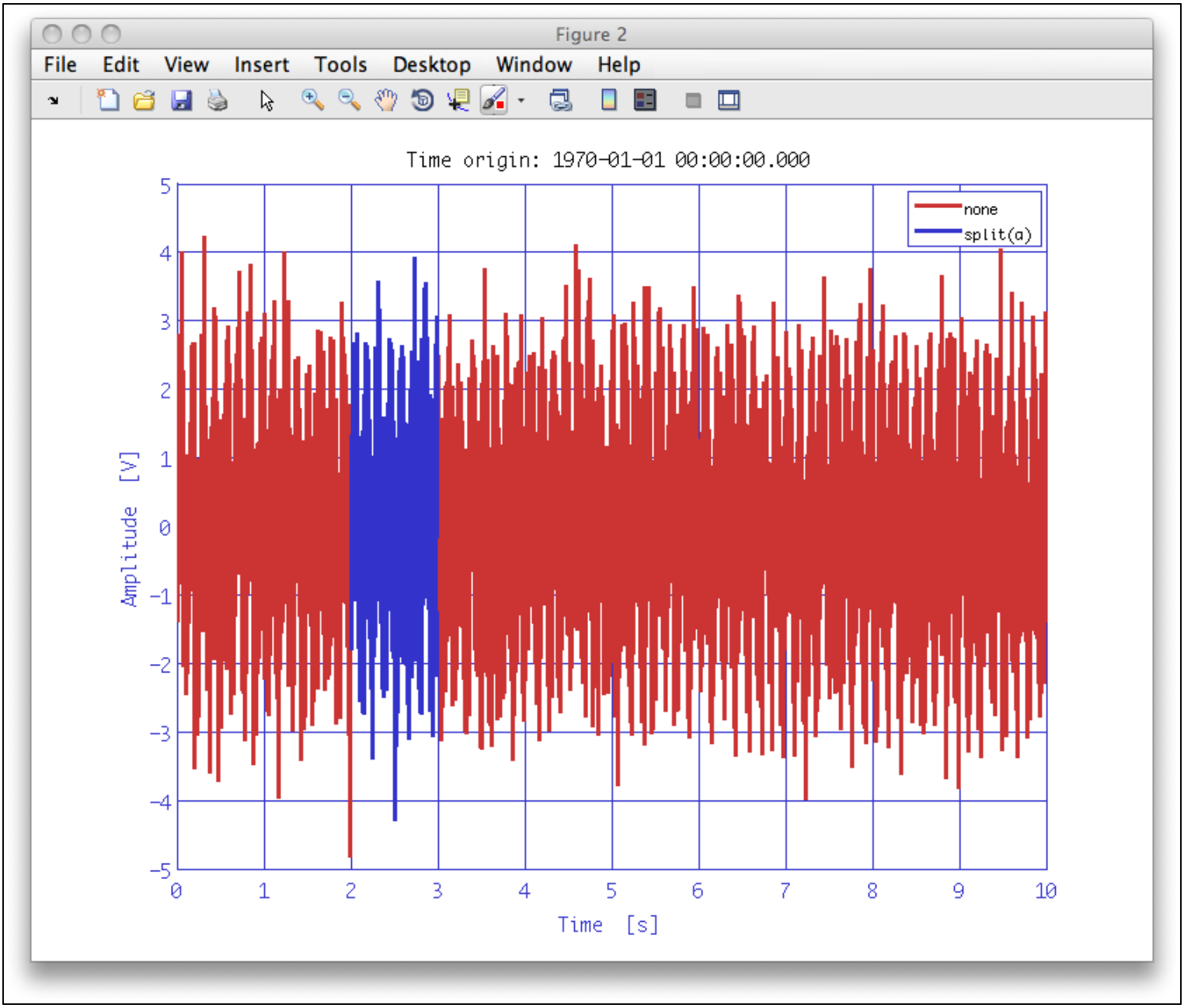
Split by times

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 `select`

```
pl_time = plist('times', [2 3]);
a_time  = split(a,pl_time);
iplot(a, a_time)
```



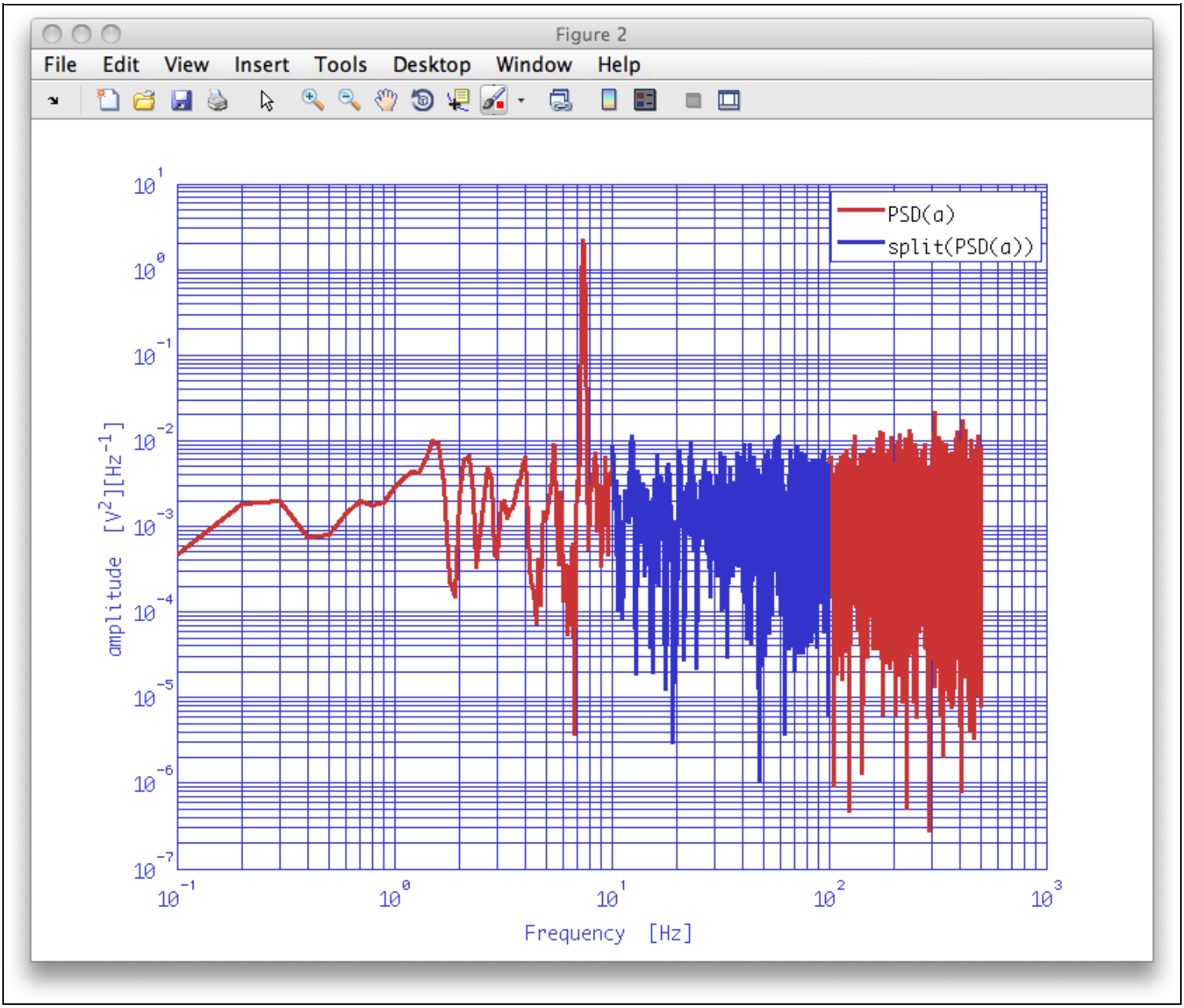
Split by frequencies

For this we need a frequency data AO. One easy way to get this is by computing the power spectrum using `psd`.

```
axx = a.psd;
```

Again we need a vector for the parameter list and pass it to `split`

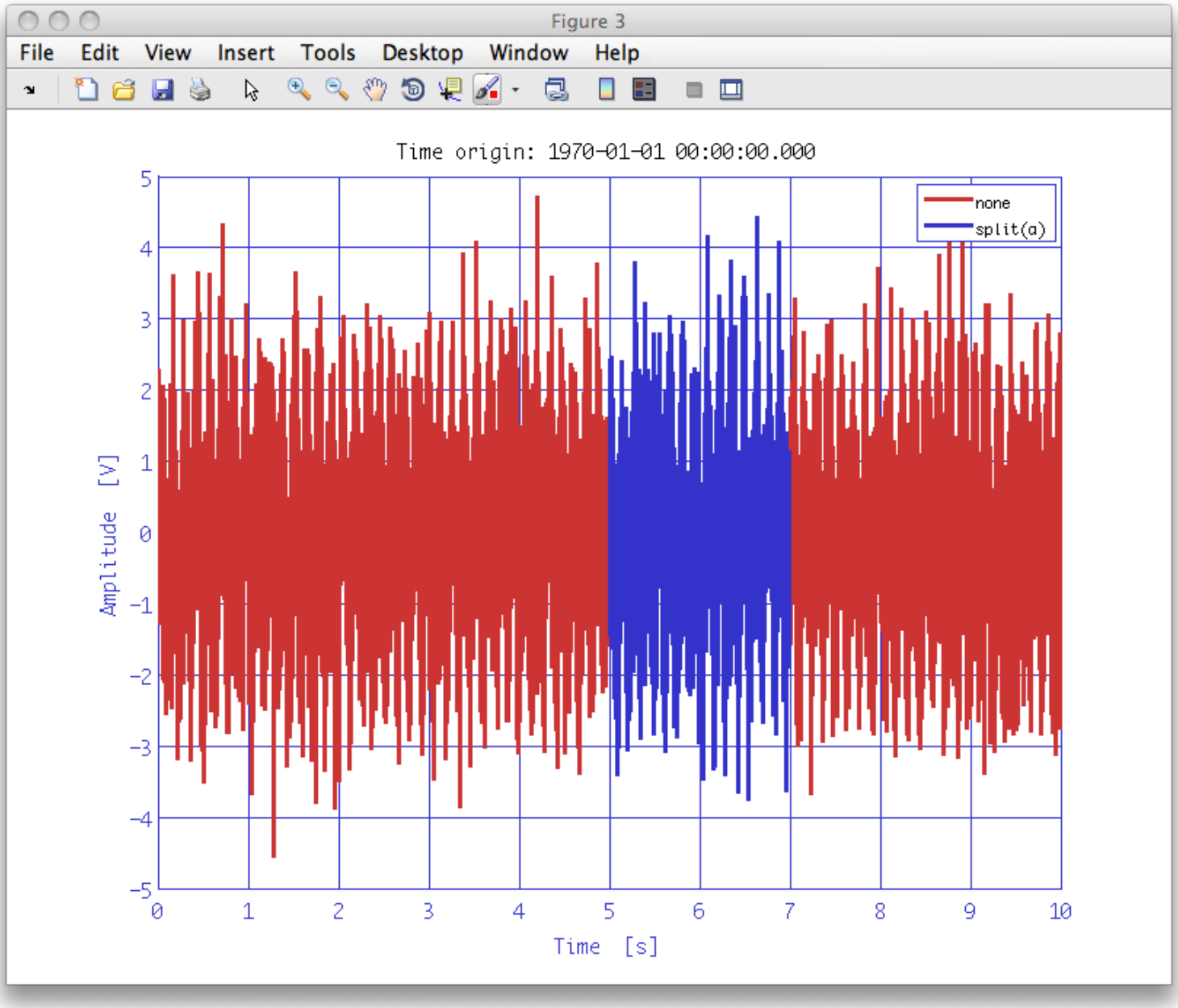
```
pl_freq  = plist('frequencies',[10 100]);
axx_freq = split(axx,pl_freq);
iplot(axx, axx_freq)
```



Split by intervals

We can also split the AO by passing a time interval to the `split` function.

```
pl_interv = plist('start_time', 5000,'end_time', 7000);
a_interv= split(a,pl_interv);
iplot(a,a_interv)
```

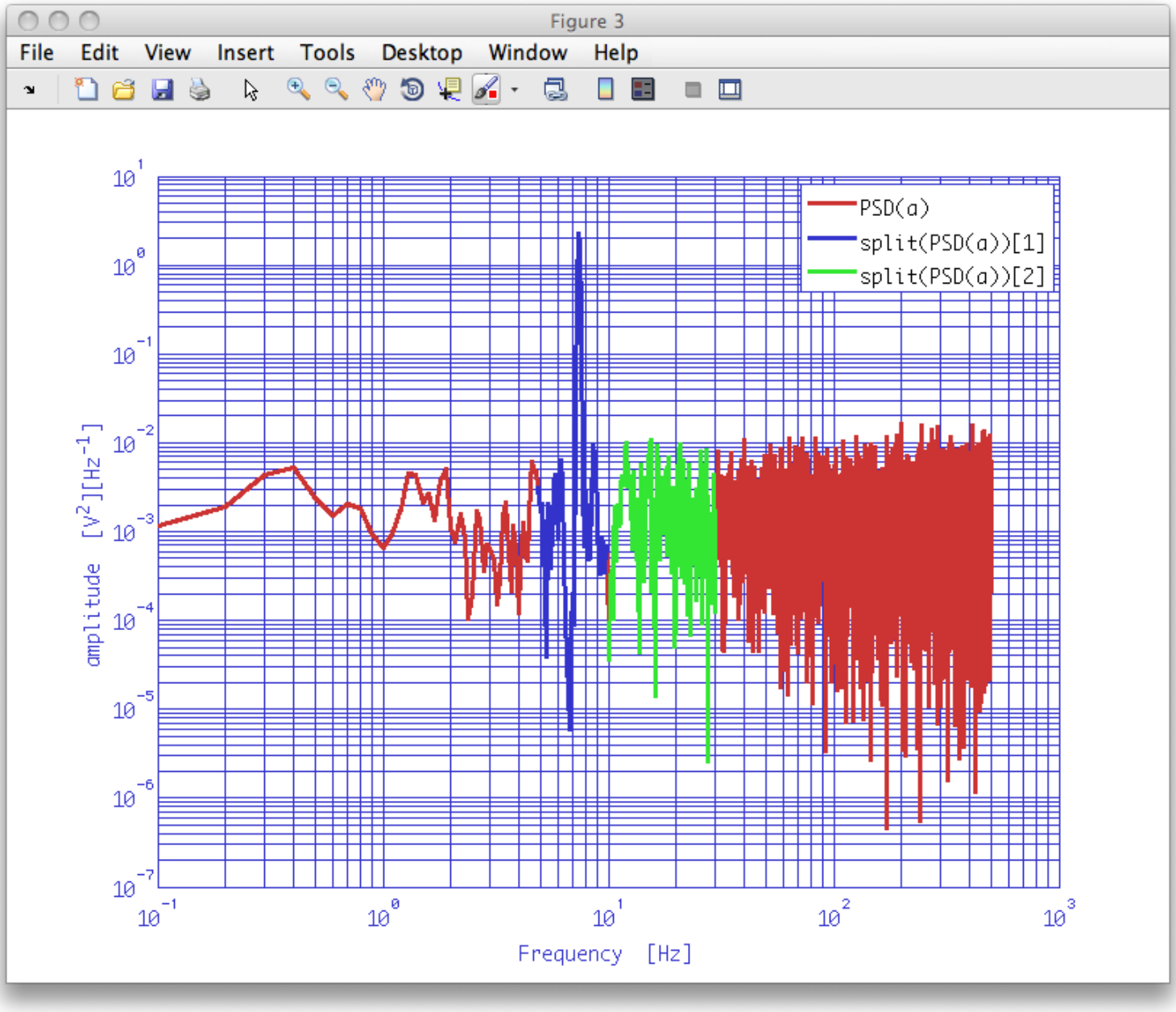


Split by samples

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 `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)
```

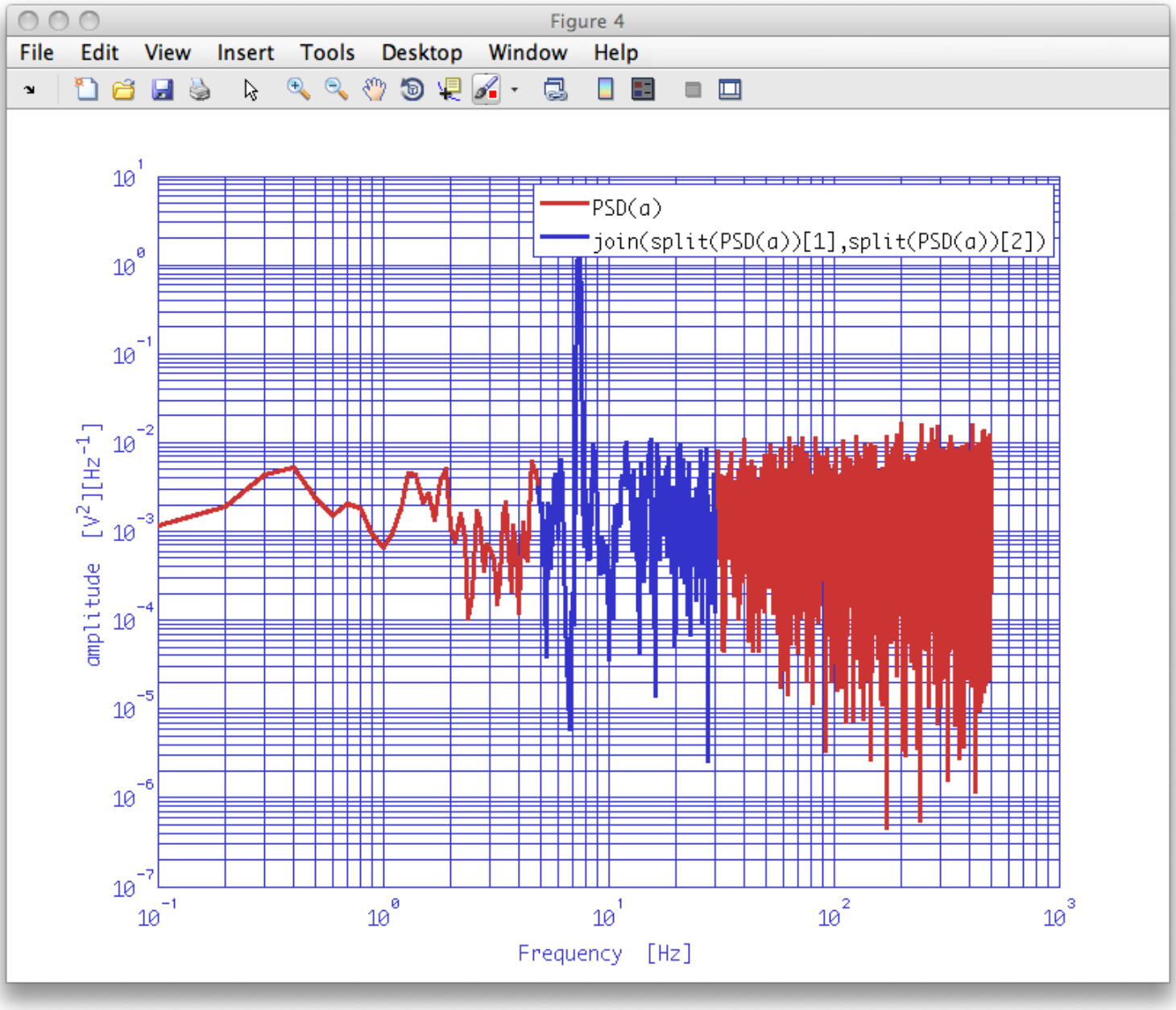


Although in this example the two resulting AOs are contiguous, they need not be.

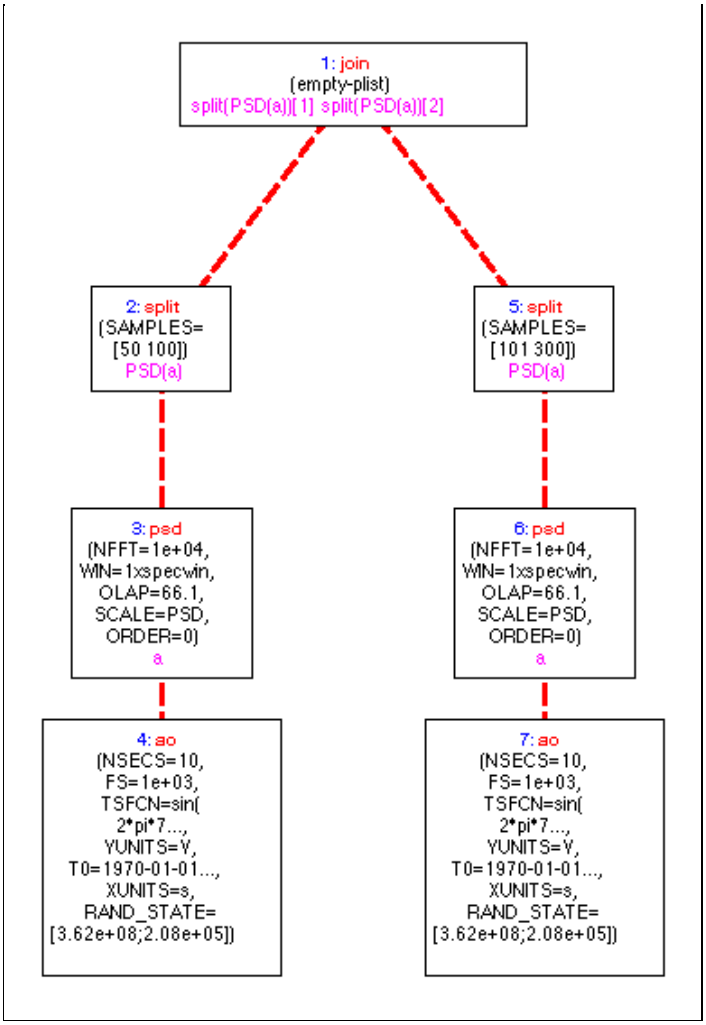
Join AOs

We can join our two AOs back together using `join`

```
axx_join = join(axx_samp1, axx_samp2);
iplot(axx, axx_join)
```



If we look at the history for `axx_join` (`plot(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.

Select and find data from an AO IFO/Temperature Example - Pre-processing

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:

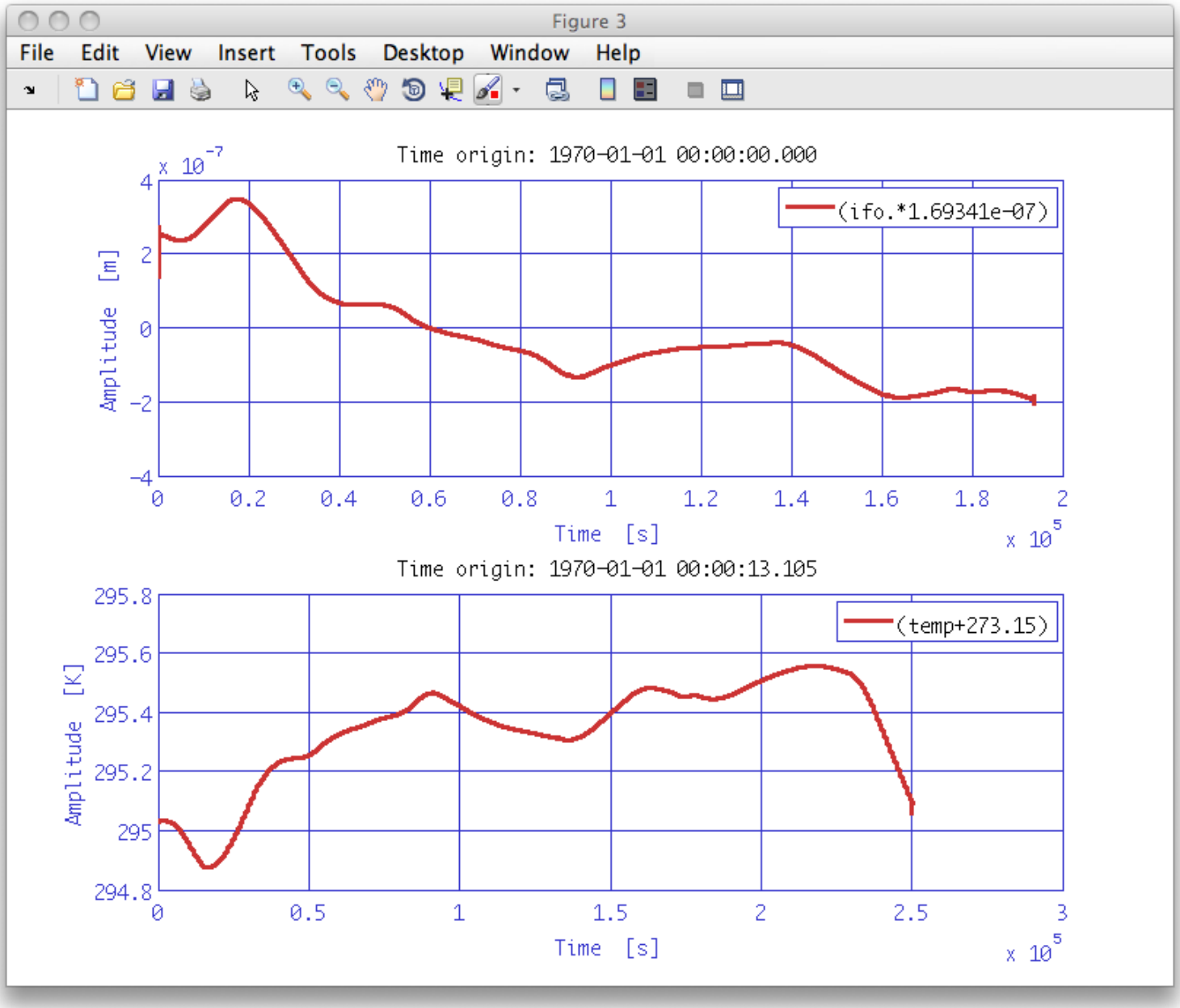
```
ifo  = ao('ifo_temp_example/ifo_disp.xml');
temp = ao('ifo_temp_example/temp_kelvin.xml');
```

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. 1) The two data streams:
   - 1) do not have the same sampling frequency.
   - 2) are not of the same length (nsecs).
2. The interferometer data has a small transient at the start

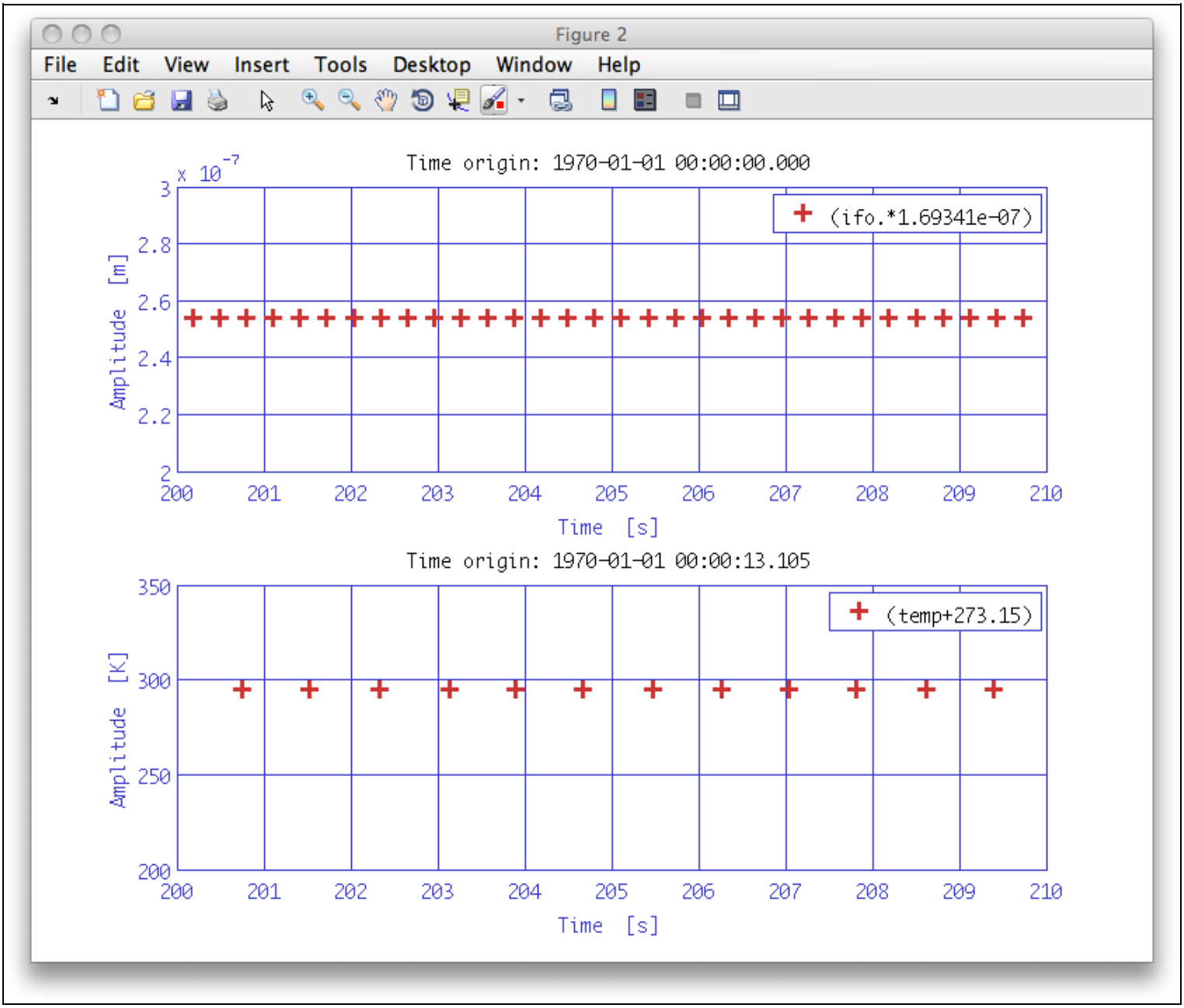
To have a closer look at the samples we plot markers at each sample and only plot a zoomed-in section. You can do this by creating a parameter list with the following key/value pairs:

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 insteaad 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 is not evenly sampled.

Before we proceed with the later analysis of this data, we need to

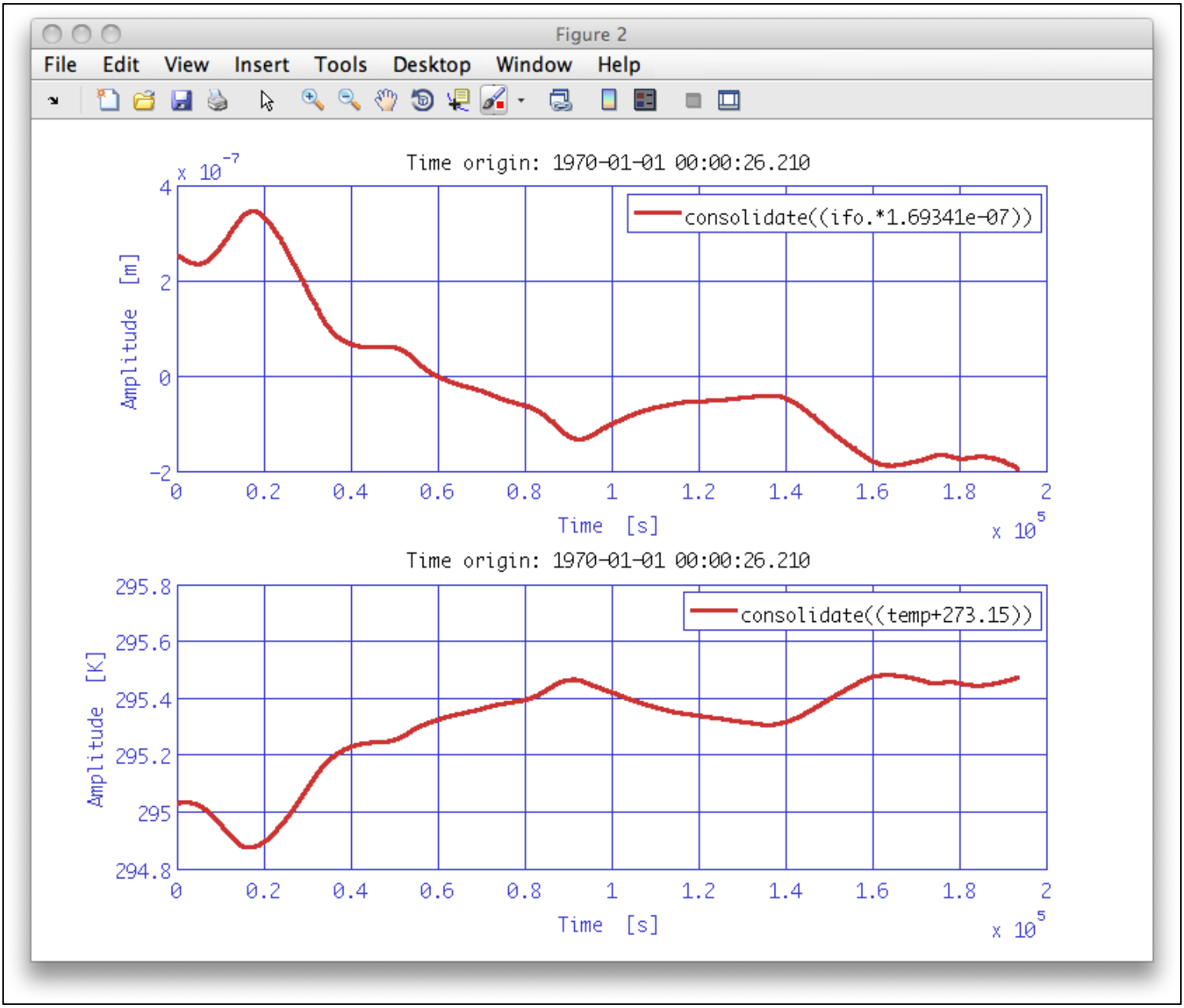
- Fix the uneven sampling of the temperature data
- Resample both data streams to the same rate
- Resample both data streams on to the same timing grid
- Select the segment of interferometer data that matches the temperature data

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));
```

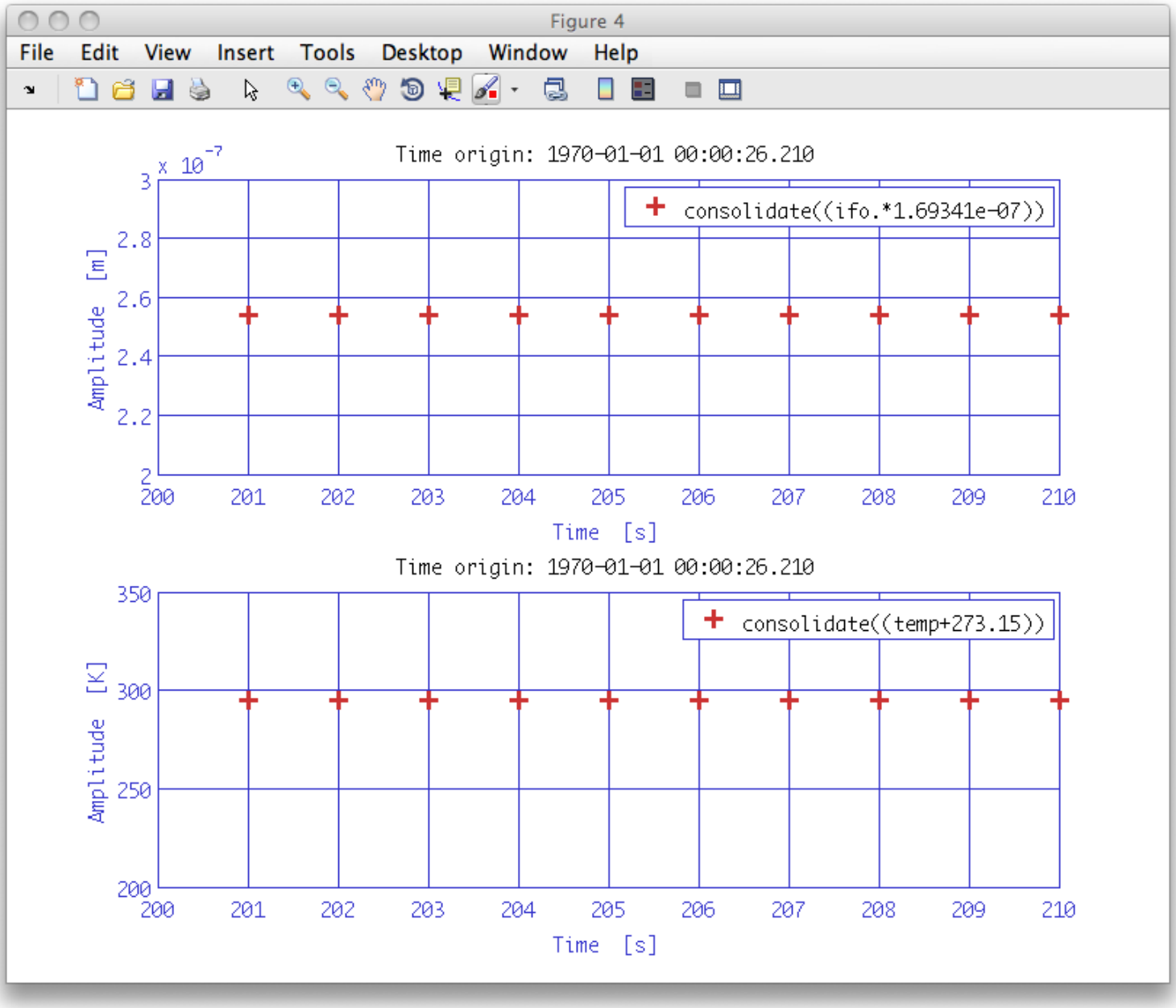
We tell `consolidate` that we want to have our data resampled to 1 Hz by specifying the parameter key 'fs'.

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');
```

Split and join AOs

Topic 3 - Spectral Analysis

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:

IFO/Temperature Example - Pre-processing Introducing Spectral Analysis

Introducing Spectral Analysis

Spectral analysis

For the sake of brevity, a very simple collection of the basis of spectral analysis, copied directly from Matlab documentation, is included in this user manual section.

More detailed description, including the whole document referred before, can be found by typing:

```
>> doc signal
```

in the Matlab terminal.

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/windwing/detrending
3. Log-scale PSD on MDC1 data: a "realistic" example from the first Mock Data Challenge, employing the log-scale PSD estimation method

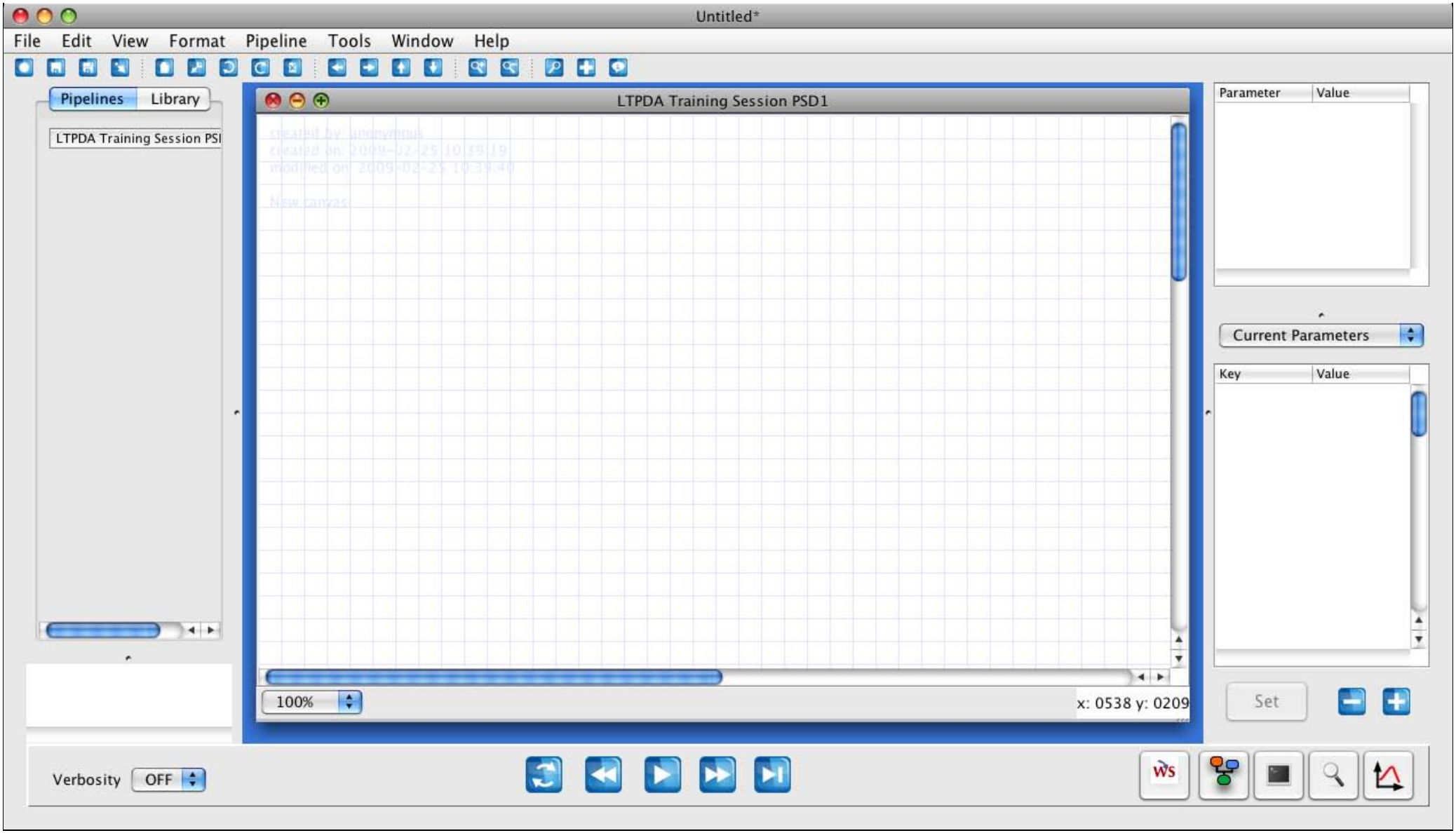
Example 1: Simply PSD

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. Anyway, 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 refert to the appropriate section of the user manual.)

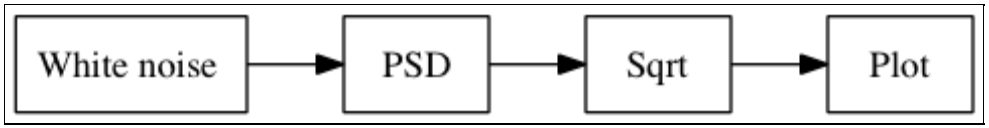
Let's use the command "Pipeline -> Rename Pipeline" to give this diagram a more significant name, such as, for instance, "LTPDA Training Session PSD1". You should see a window like the one below:



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:

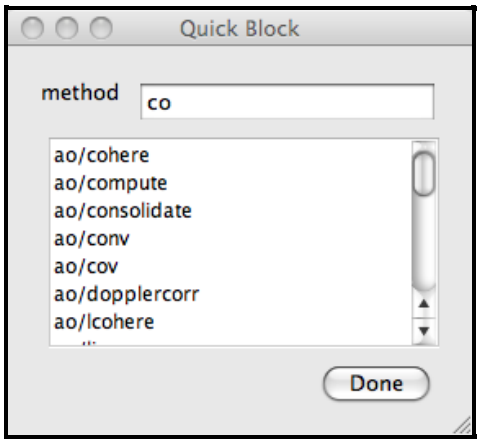


Simulate a time-series of white noise data

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 first parameter line (`WAVEFORM`), so we can choose "noise" from the drop-down list that will appear. If needed, more help can be found here.

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:

- `Scale` allows to choose the quantity to be sent in output (ASD, PSD, AS, PS)
- `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
- `Nfft` allows to choose the number of samples in each periodogram evaluation
- `Order` allows to choose the degree of detrending applied to each segment prior to windowing

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:

- `Scale` = 'PSD' so we evaluate the Power Spectral Density [output units] will be [input units]^2 / Hz
- `Win` = The value set by the user in the preferences
- `Olap` = -1 so the overlap will be chosen based on the windows properties
- `Nfft` -1 so to estimate the PSD only on one window, including the whole data set
- `Order` = 0 so to remove the mean value from the data before applying the window

Extract the square root of the calculated power spectrum

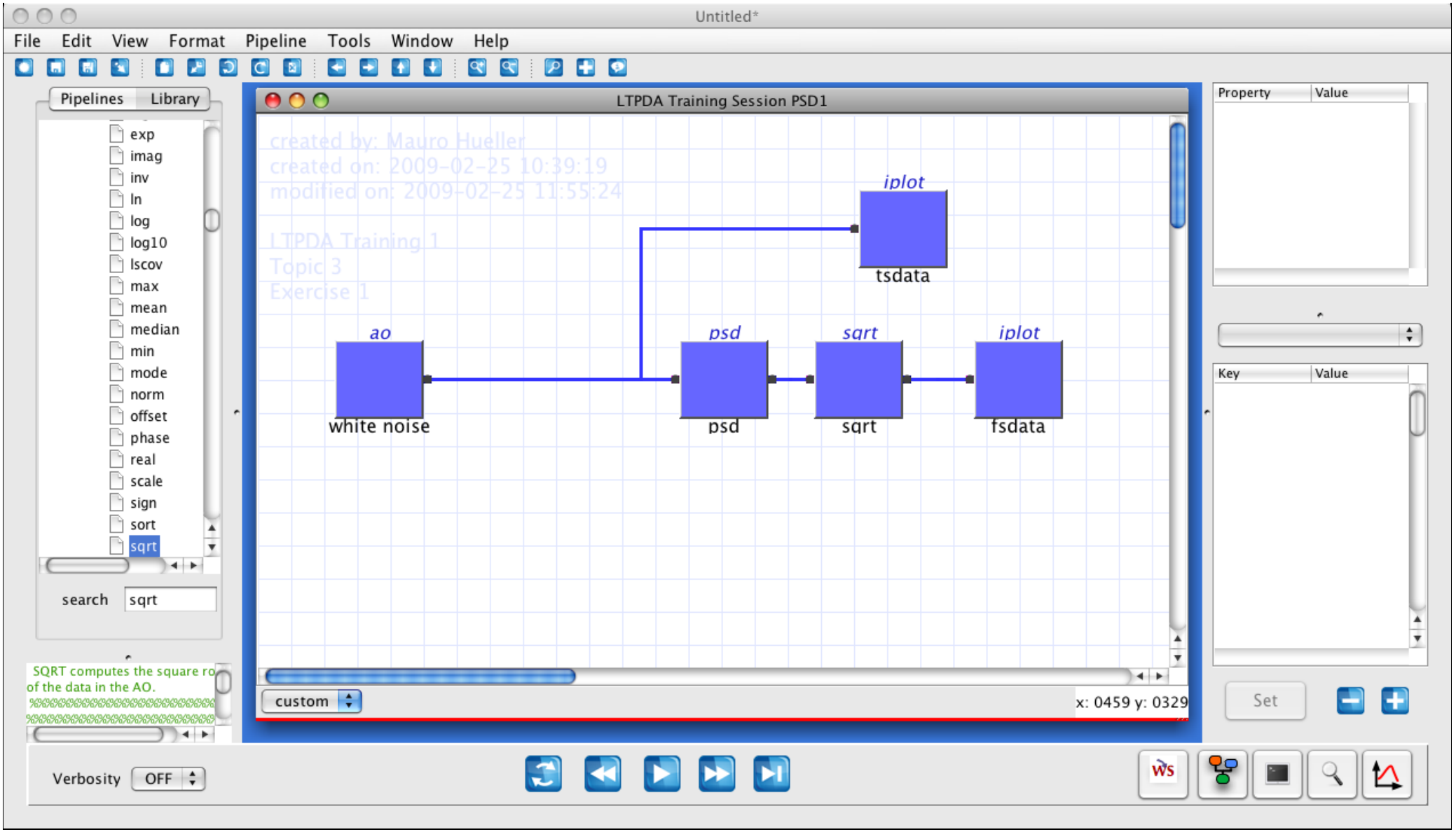
For this exercise, we proceed by computing the square root of the calculated spectrum. Notice: by default, as described in its help, `ao/sqrt` applied to `fsdata` and `tsdata` objects acts only on the *y* (dependent) variable. We'll see later that is possible to choose different outputs so as to avoid 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's remains is to just 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 `iplot` to look at the original time-series signal.

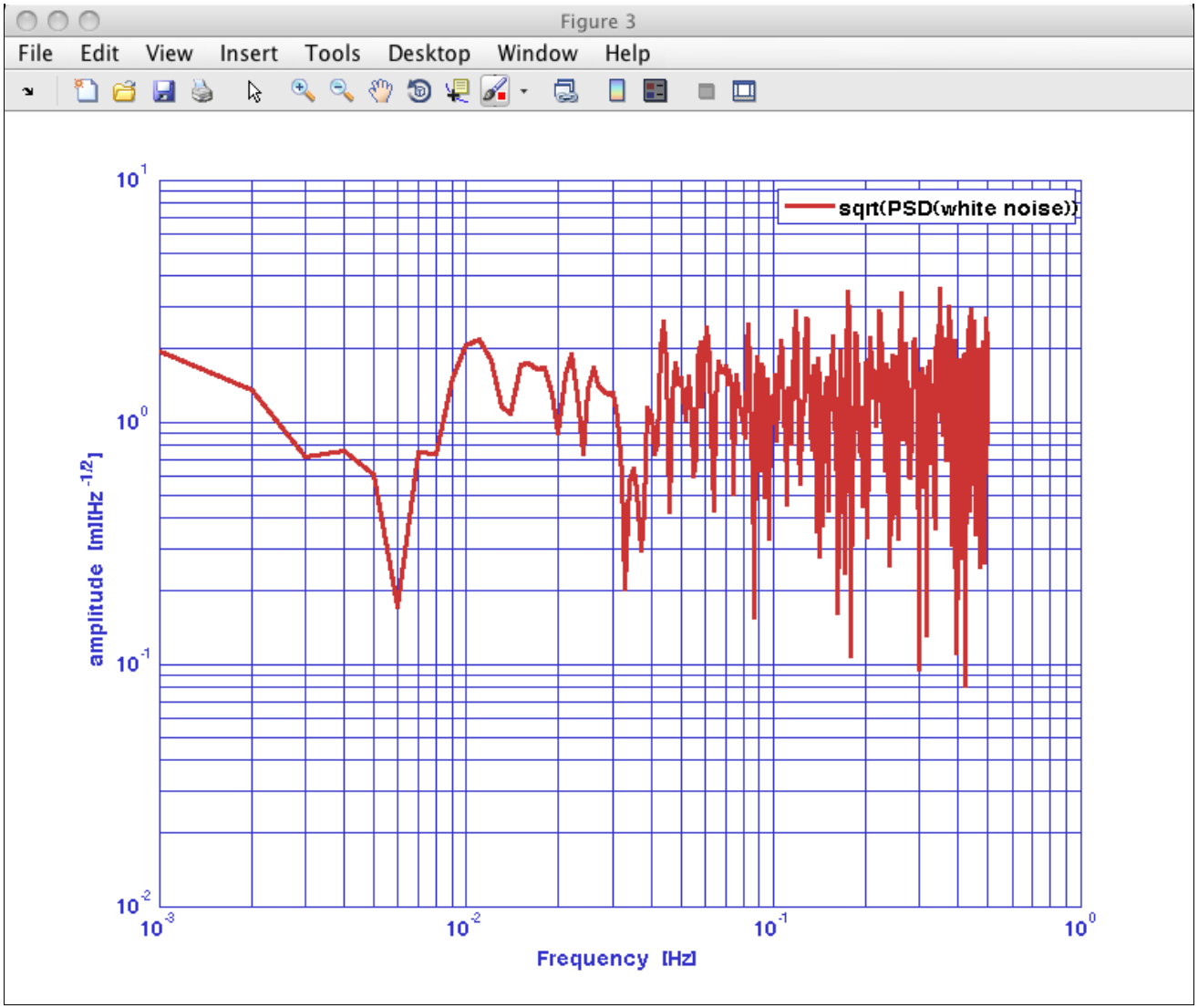
So at the end we should have a situation like the following:



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

Power Spectral Density estimation

Example 2: Windowing data

Example 2: Windowing data

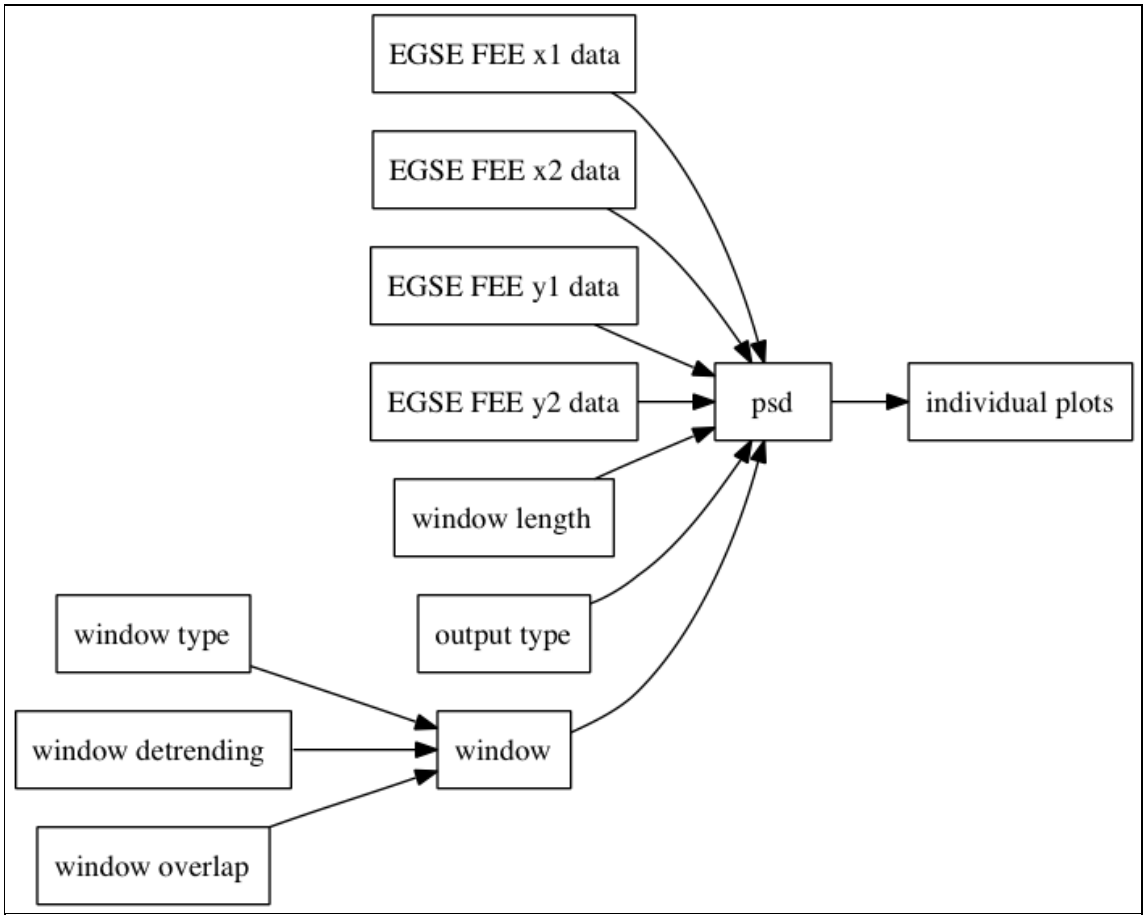
In the second exercise we will use again the graphical programming environment called LTPDA Workbench. After checking that the workbench is loaded, let's go ahead and create a new pipeline, or analysis diagram.

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.
USE_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:

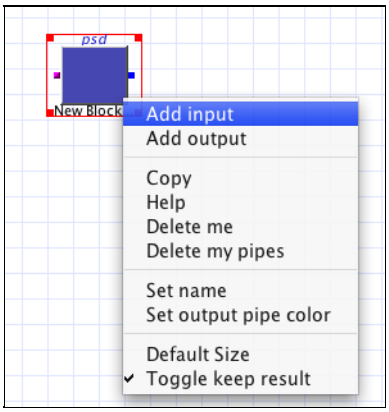
Key	Value
FILENAME	'topic3/EGSE_FEE_x1.dat'
FILENAME	'topic3/EGSE_FEE_x2.dat'
FILENAME	'topic3/EGSE_FEE_y1.dat'
FILENAME	'topic3/EGSE_FEE_y2.dat'

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":

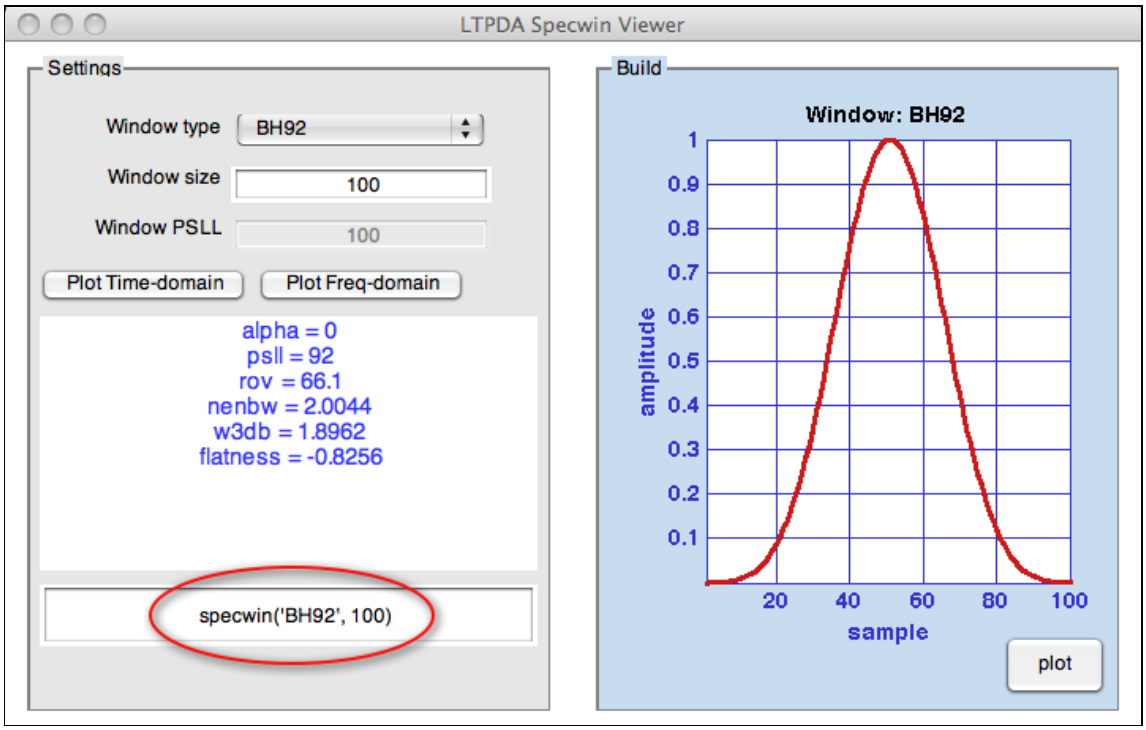


Then obviously we need to connect each of the `ao` constructor blocks to an input of the `psd` block.

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 alreay selected, based on the user preferences. Here I chosed 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 reccomended 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).

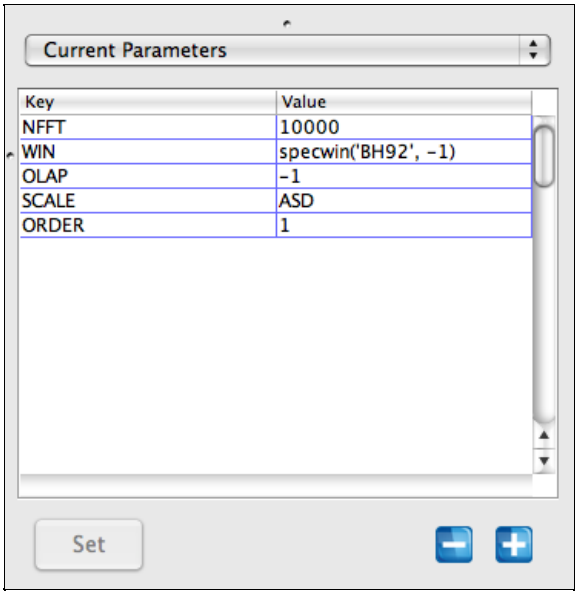
Here I choose to use ASD, so I double-click on the "Value" corresponding to the "Scale" entry and go ahead and enter the string, 'ASD'.

Choosing the detrending

Detrending here refers to an additional detrending performed for each individual segment, prior to applying the window.

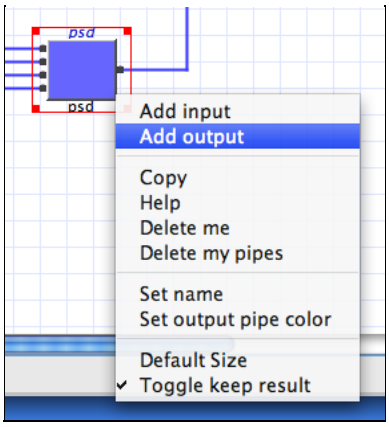
In this case we want to subtract a linear drift, so just enter 1.

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":

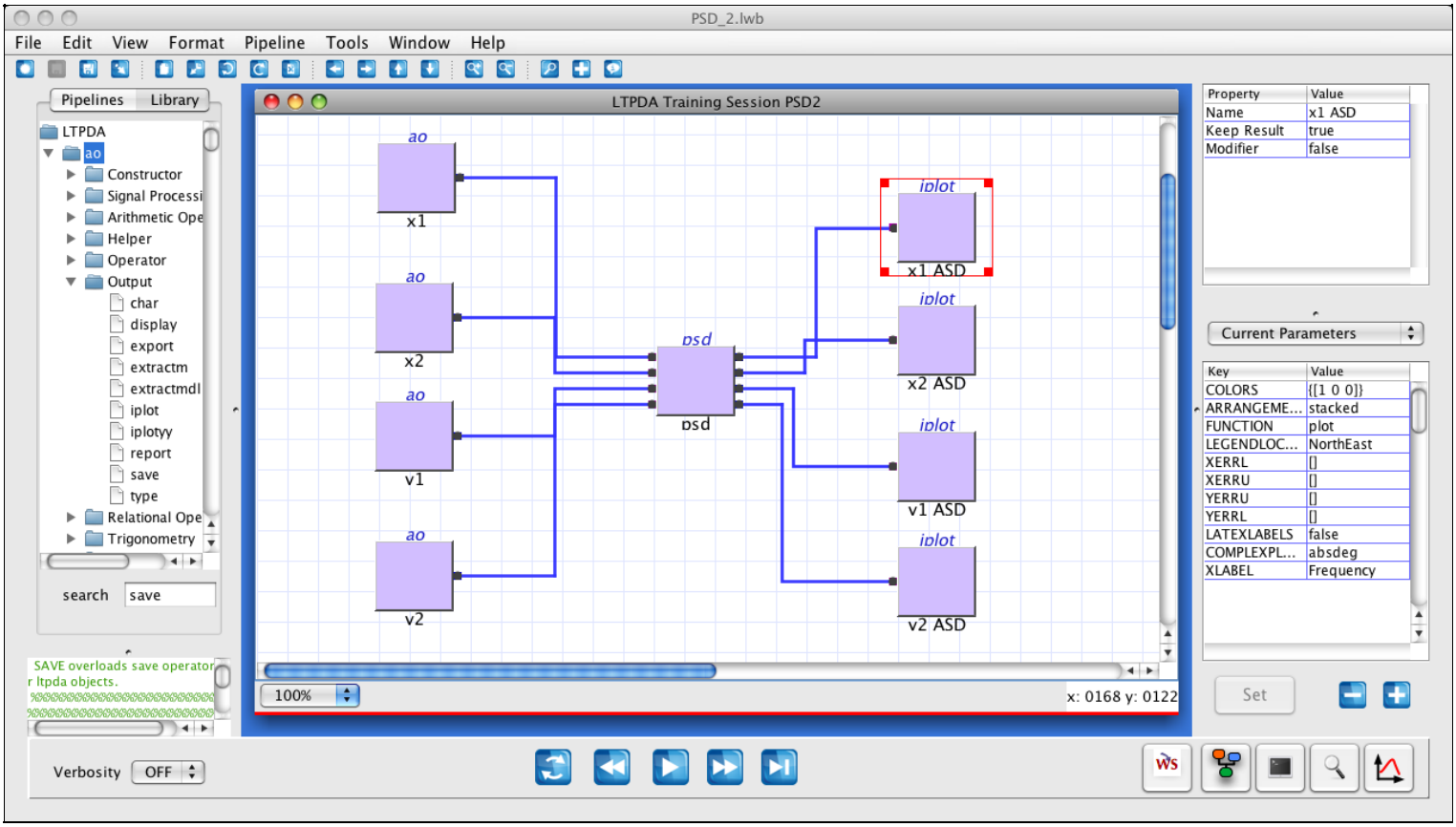


We can now go ahead and plot the individual `ao`s in many plots, just by adding `iplot` blocks and connecting them to each output port of the `psd` block.

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:

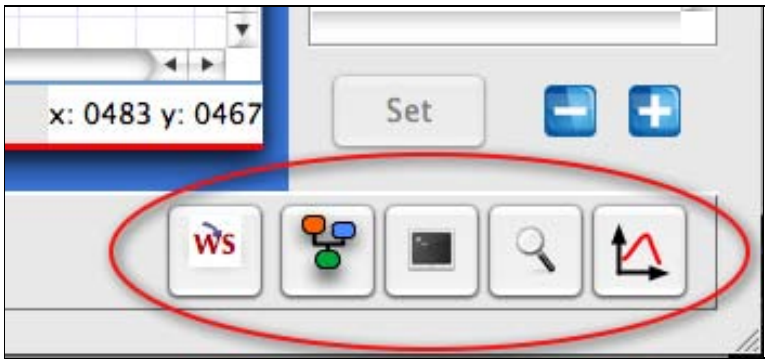


We can go ahead and execute it ...

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-bottm side of the GUI:



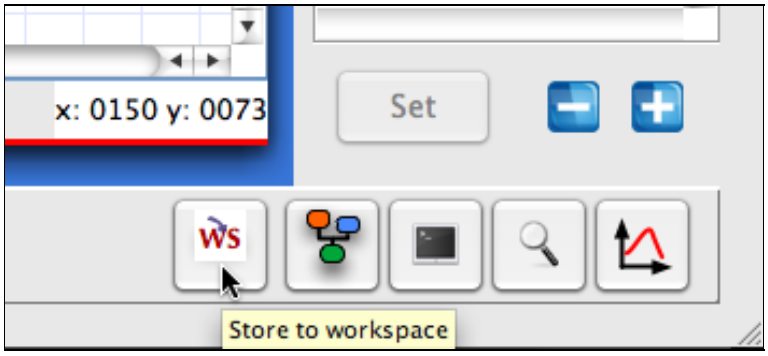
They allow us to:

- Display the history of the selected object
- Display the selected object on the Matlab terminal
- Explore the selected object
- Plot the selected object

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 worskpace

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 beteween 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:

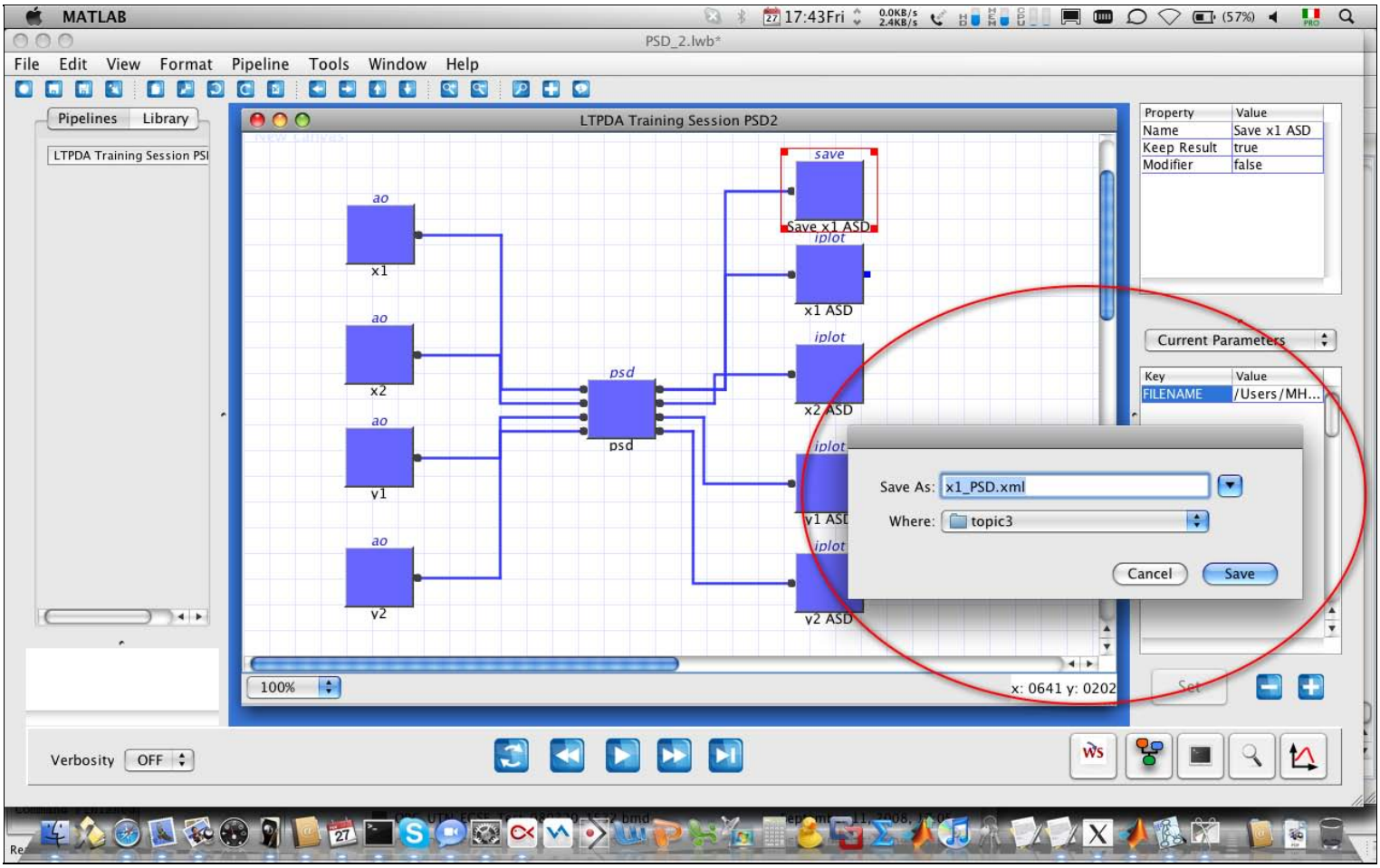
```
>> psd_PORT0
```

```
----------- ao 01: ASD(x1) -----------

       name:  ASD(x1)
description:  EGSE FEE x1 data
       data: (0,2.53724220837033e-18) (0.00051203275,4.68581398941711e-18)
(0.0010240655,5.7415265802551e-18) (0.00153609825,5.76606096674634e-18)
(0.002048131,5.64506157755643e-18) ...
             ----------- fsdata 01 -----------

                 fs:  10.2407
                  x:  [10001 1], double
                  y:  [10001 1], double
             xunits:  [Hz]
             yunits:  [F Hz^(-1/2)]
                 t0:  1970-01-01 01:00:00.000
               navs:  31
             --------------------------------

       hist:  ao / ao / $Id: ltpda_training_topic_3_2_2_content.html,v 1.9 2009/08/03 13:38:27
mauro Exp $
  mfilename:
mdlfilename:
-------------------------------------


>>
```

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.



Example 1: Simply PSD Example 3: Log-scale PSD on MDC1 data

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
```

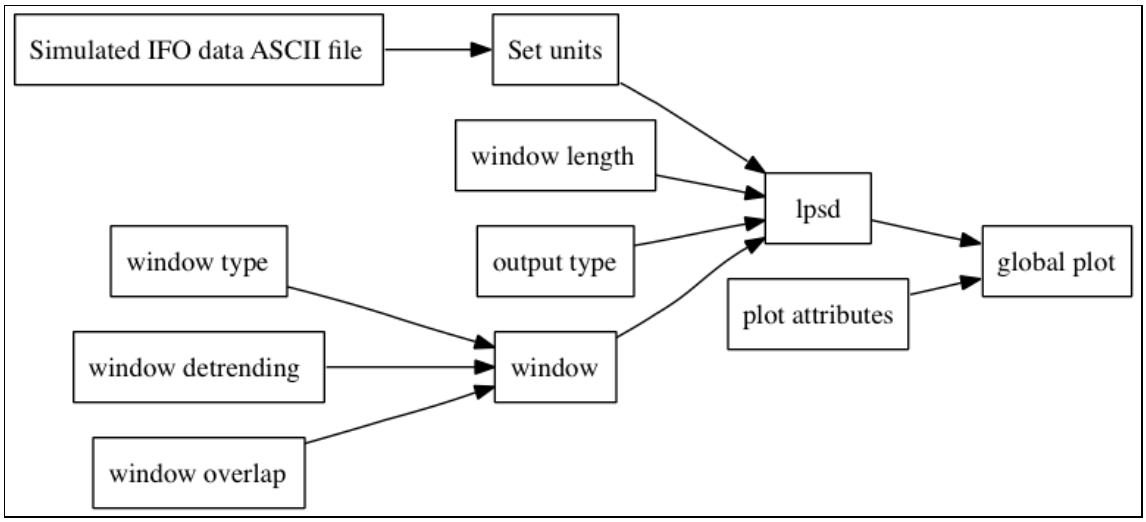
As we did before, let's go ahead and create a new pipeline, or analysis diagram.

Again, use the command "Pipeline -> Rename Pipeline" to give this diagram a more significant name, such for instance "LTPDA Training Session PSD1".

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" colum, 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.
USE_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	'specwin('BH92', 0)'	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.

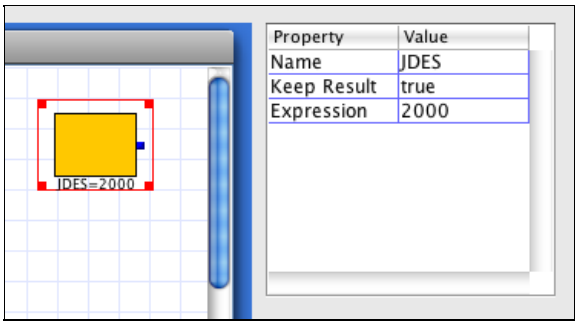
- `'Jdes'` - the number of spectral frequencies to compute
- `'Kdes'` - the desired number of averages
- `'Lmin'` - the minimum segment length [default: 0]

We will act on the first one, so to decide how many PSD bins to estimate, levaing to the algorithm the choice of their location and the length of the windows on each bin.

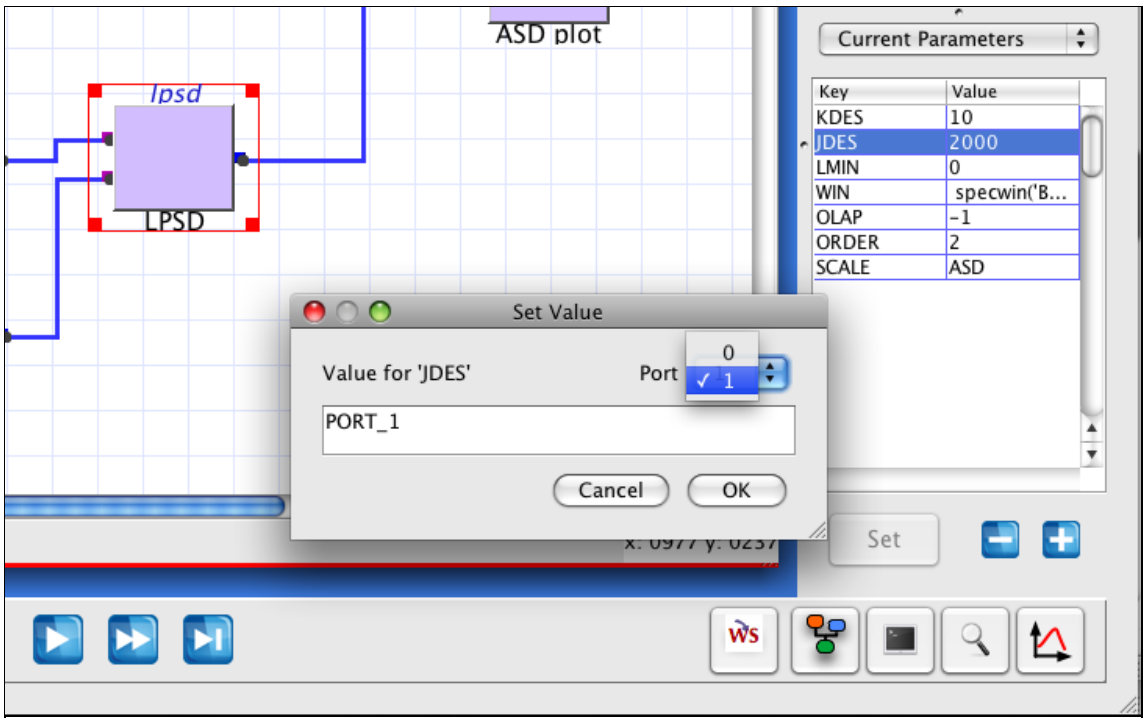
Key	Value	Description
JDES	2000	Compromising execution time and resolution
KDES	10	Slightly less than the default value
LMIN	0	The default value

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 "Additional blocks->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", then move to the Parameter Set table, double-click on the "Value" entry corresponding to the "JDES" key, and select the now available "PORT_1". As an alternative to the last step, we can also type "PORT_1" in the text box.



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.

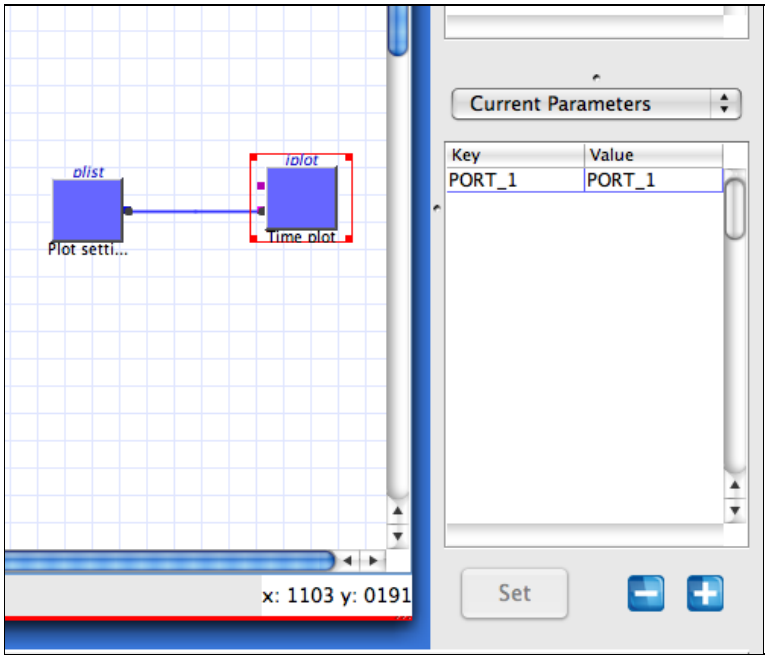
We make a little additional exercise: let's add one block of the type

`plist`

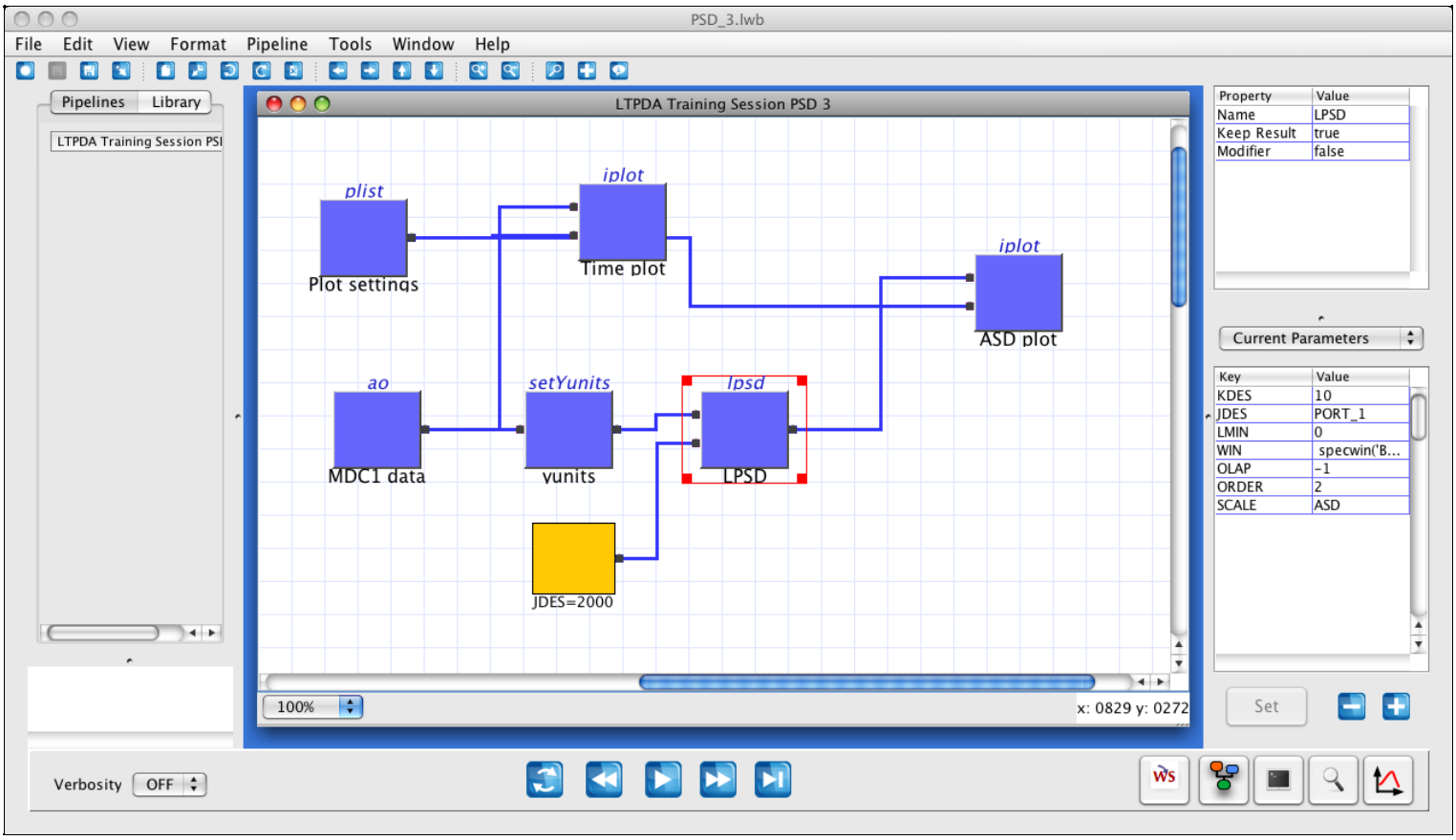
As usual, to add parameters, just hit he "+" sign, so we can define some "key" and "value" properties:

Key	Value	Description
COLORS	{[0 0 1],[1 0 0]}	Setting *x1* to blue and *x12* to red

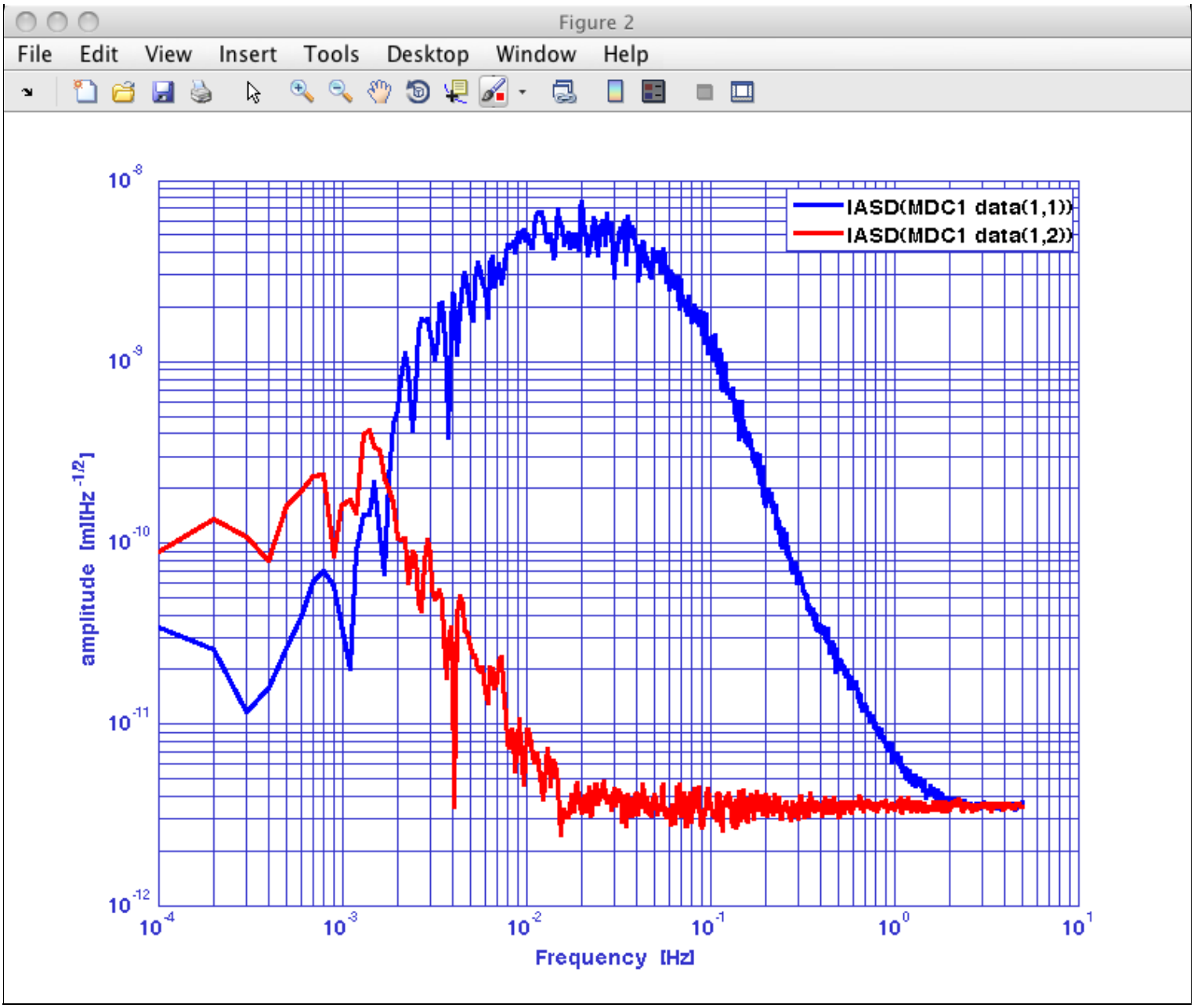
Now we can input this `plists` to the `iplot` blocks, by adding an input to them, then going into the Parameter Set area, hitting the "+" botton, and selecting "PORT_1" for both the "key" and "value" entries. Like:



Ready for execution then ... and that's how the workbench should look like:



The execution takes some while ... and here are the results:



Example 2: Windowing data
Empirical Transfer Function estimation

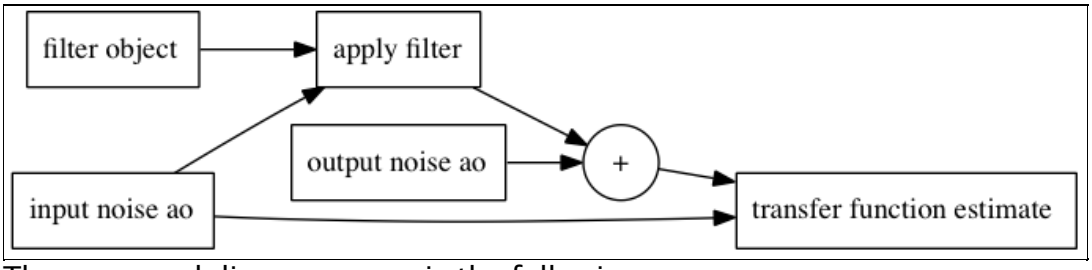
Empirical Transfer Function estimation

Let's run this exercise on empirical estimation of Transfer Functions on the Matlab terminal.

The idea of the exercise is the following:

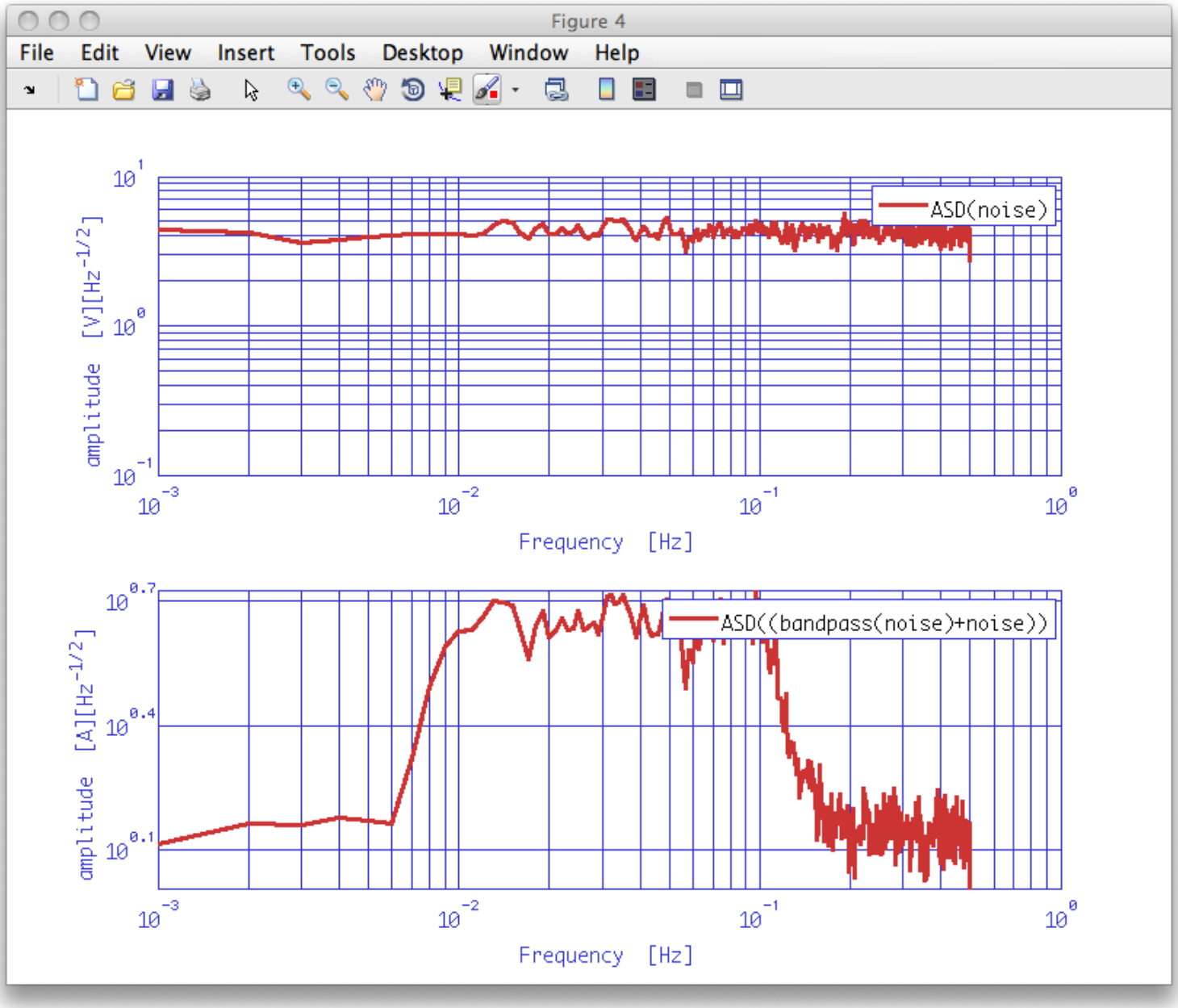
1. simulate some white noise *x(t)*
2. build a band-pass filter *F*
3. pass the input noise *x(t)* through the filter and add some more noise *yn(t)* at the output so to have *y = F*x(t) + yn(t)*
4. evaluate and plot the transfer function *x -> y*

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 lenght 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
tf = tfe(x, y, plist('nfft', 1000, 'win', 'BH92', 'olap', -1, 'order', 0));
tfxy = index(tf, 1, 2);
```

Please note the usage of the

`ao/index`
method to access the elements of an `ao`s matrix, of size [2x2] in this case, without breaking the history of the objects.

We also would like to evaluate the expected transfer function x->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

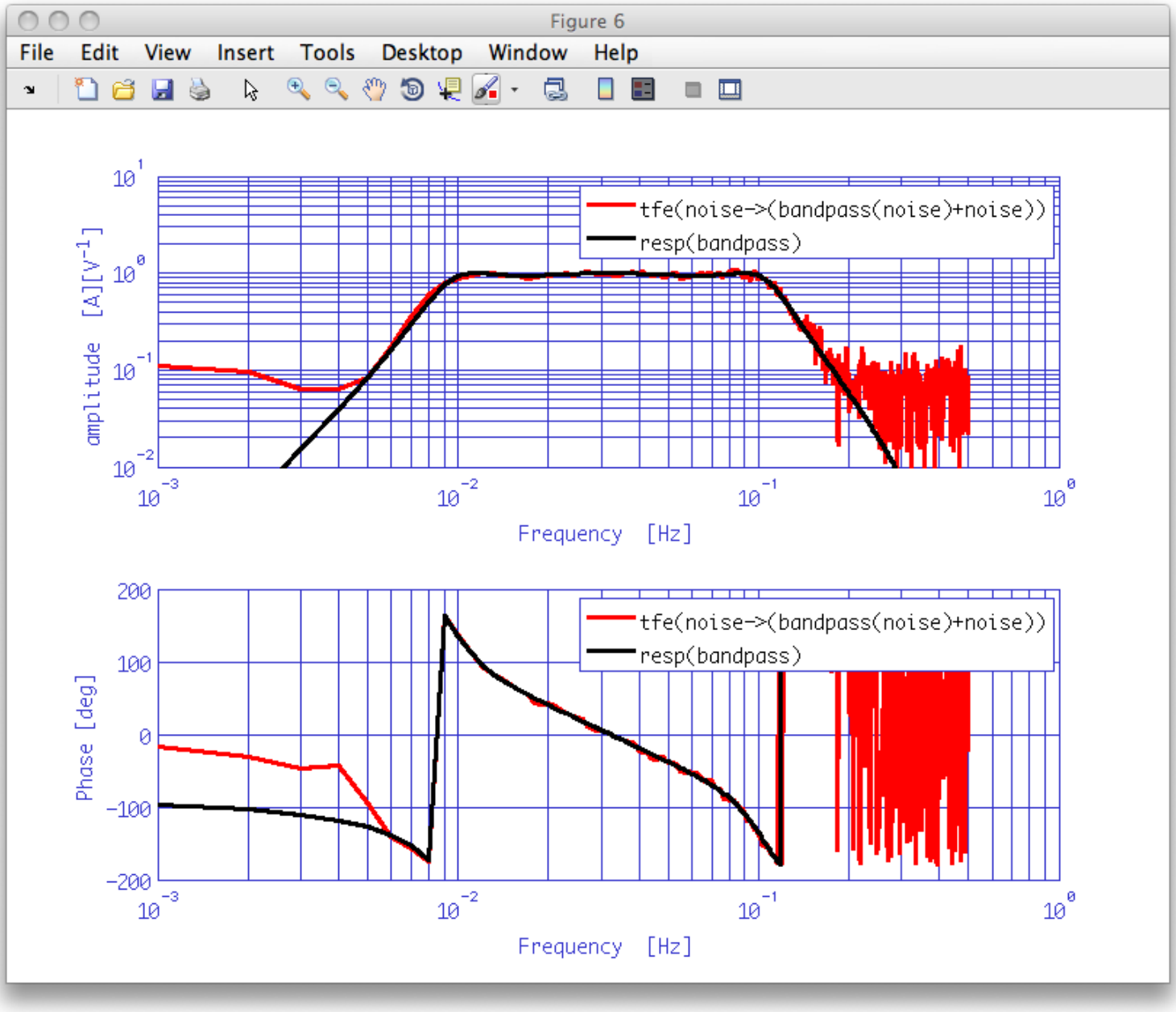
So we can just pass the x field of the fsdata `ao` containing the transfer function estimate. However, we can also just pass the AO itself. In which case, the `resp` function will take the X values from the AO.

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
iplot(tfxy, rf, plist('colors',{[1 0 0],[0 0 0]},'YRanges', {[1e-1 1e2], [-200 200]}))
```



Example 3: Log-scale PSD on MDC1 data IFO/Temperature Example - Spectral Analysis

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` constuctor, 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 ao
START_TIME	x.t0 + 40800	A time-object to start at
END_TIME	x.t0 + 193500	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 possibly to directly add a numer (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);
```

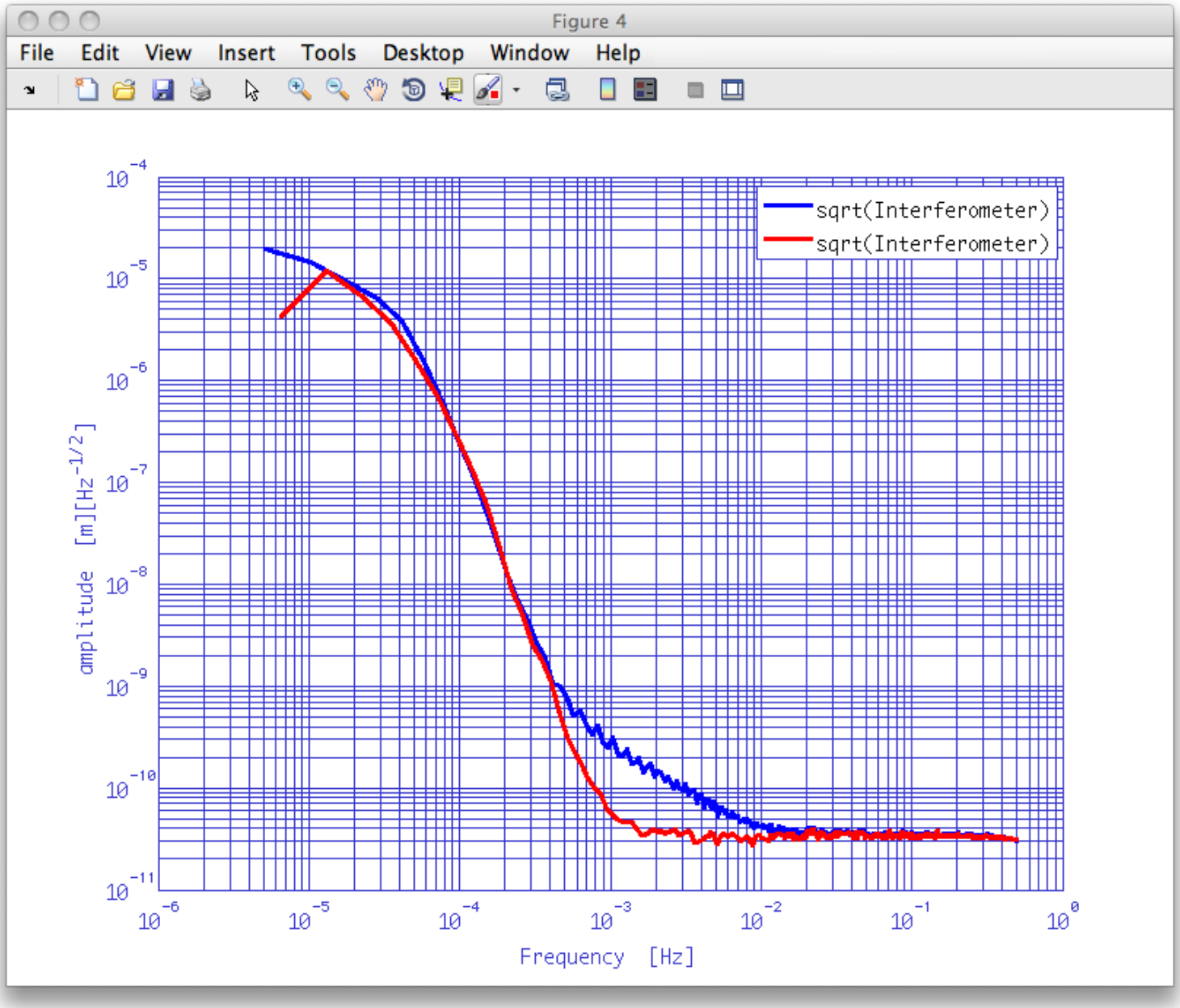
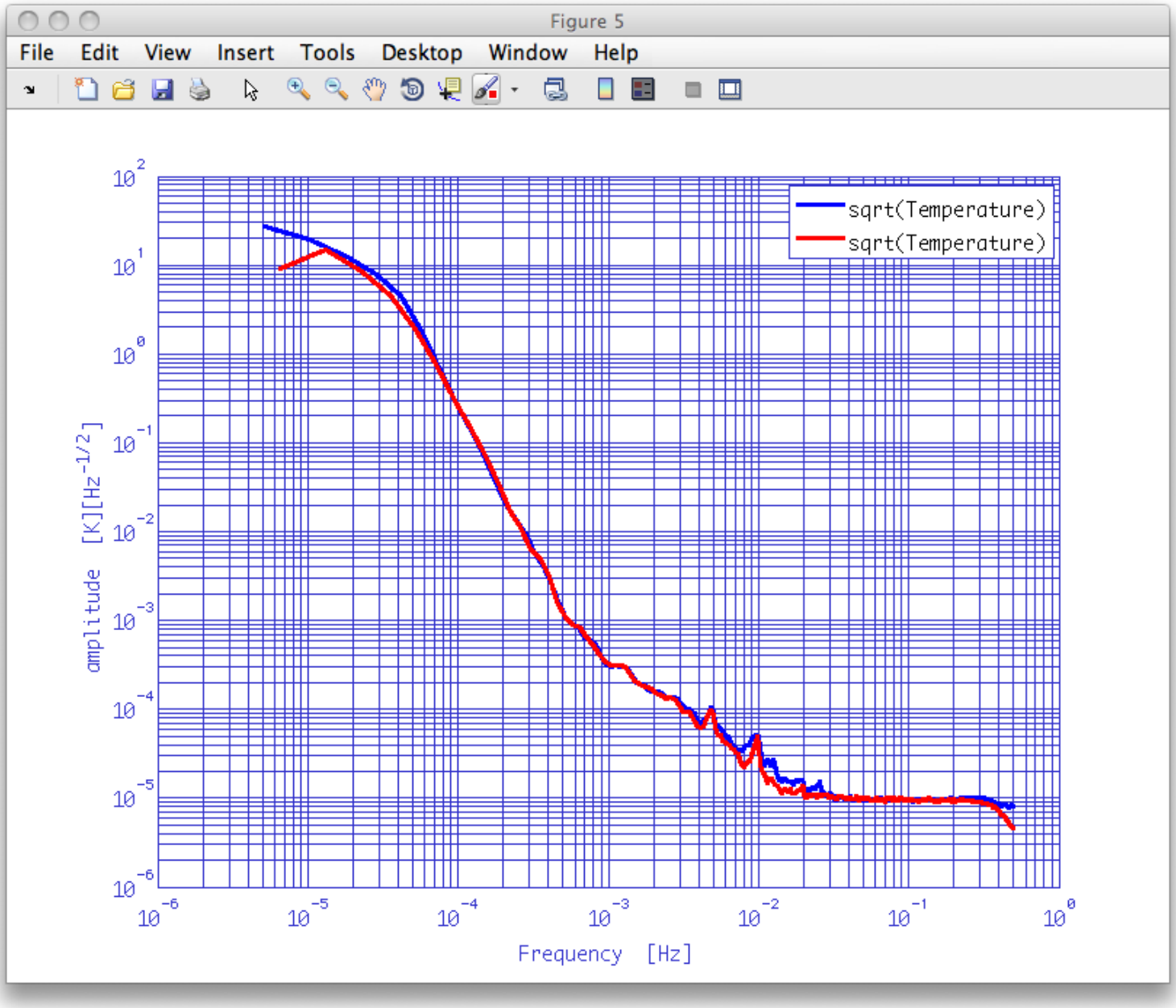


Figure 4
File Edit View Insert Tools Desktop Window Help
sqrt(Interferometer)
sqrt(Interferometer)
amplitude [m][Hz^-1/2]
Frequency [Hz]



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. The output will be a [2x2] matrix of aos, and we want to look at the off-diagonal terms, by using the

ao/index

method.

Hint:

```
%% CPSD estimate
C = lcpsd(T_red, x_red)
CTx = index(C,1,2);
CxT = index(C,2,1);
% Plot estimated CPSD
iplot(CTx);
iplot(CxT);
```

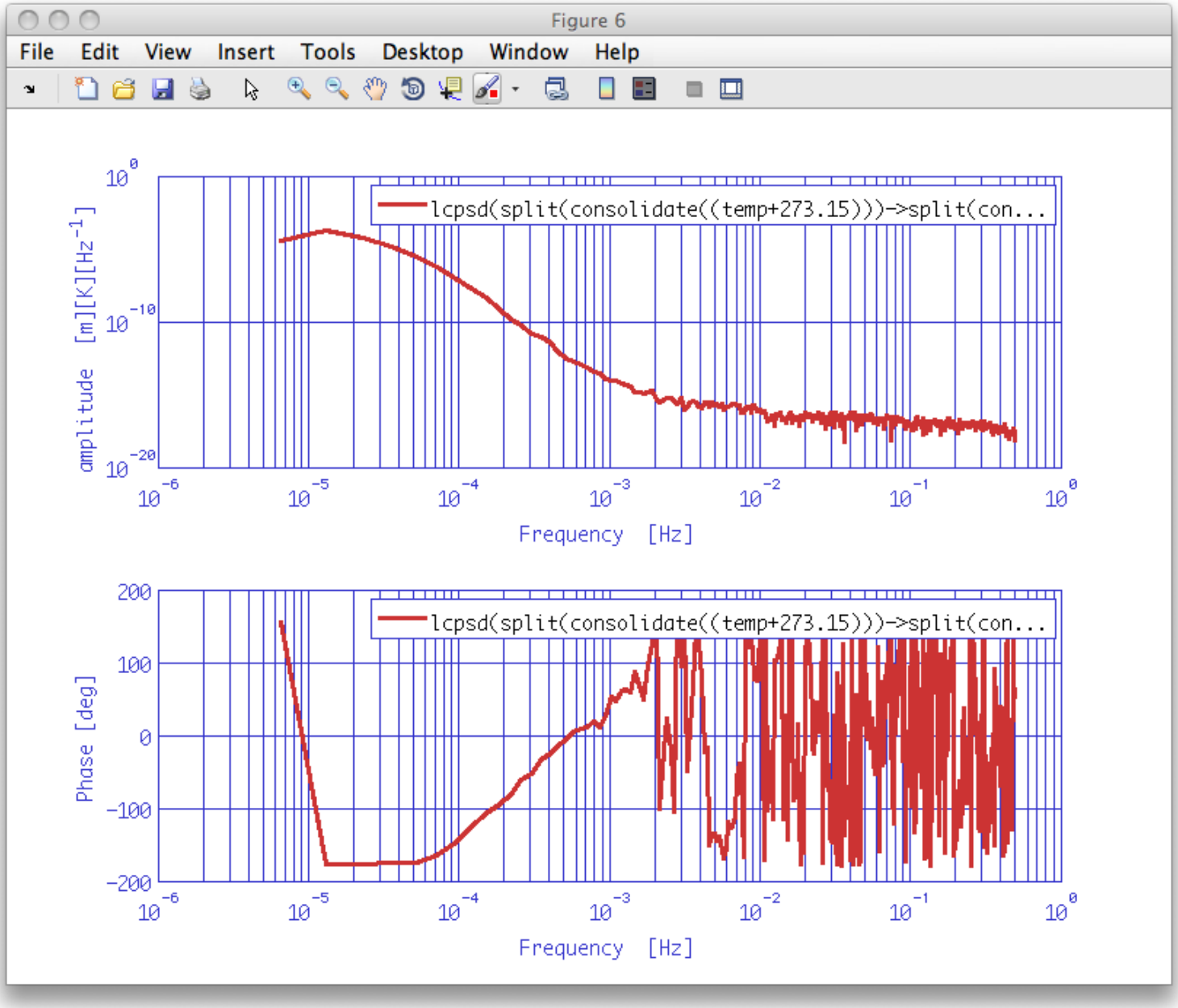
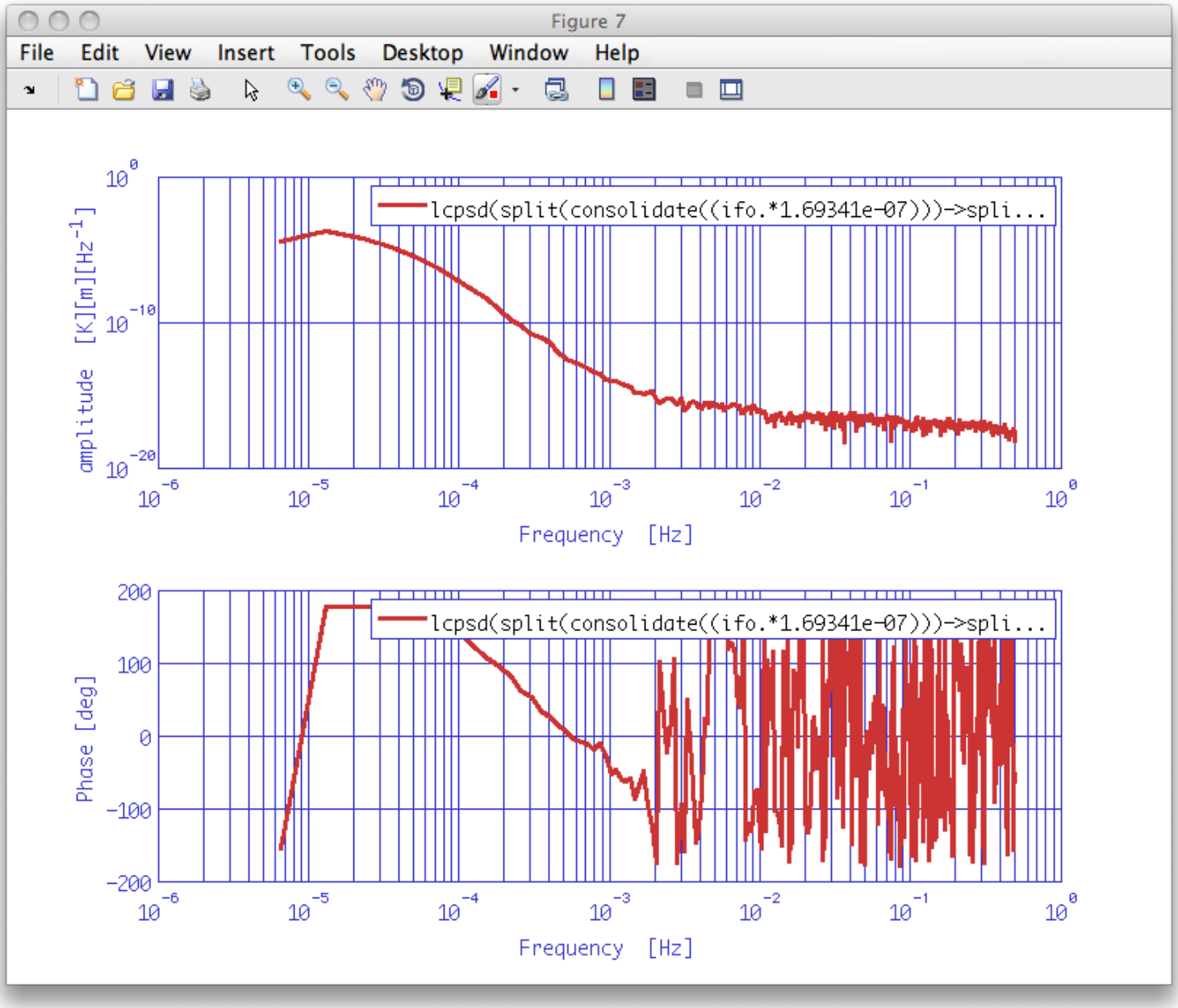


Figure 6
File Edit View Insert Tools Desktop Window Help
lcpsd(split(consolidate((temp+273.15)))->split(con...
amplitude [m][K][Hz^-1]
Frequency [Hz]
lcpsd(split(consolidate((temp+273.15)))->split(con...
Phase [deg]
Frequency [Hz]



As expected, there is a strong low-frequency correlation between the IFO data `x` and the temperature data `T`.

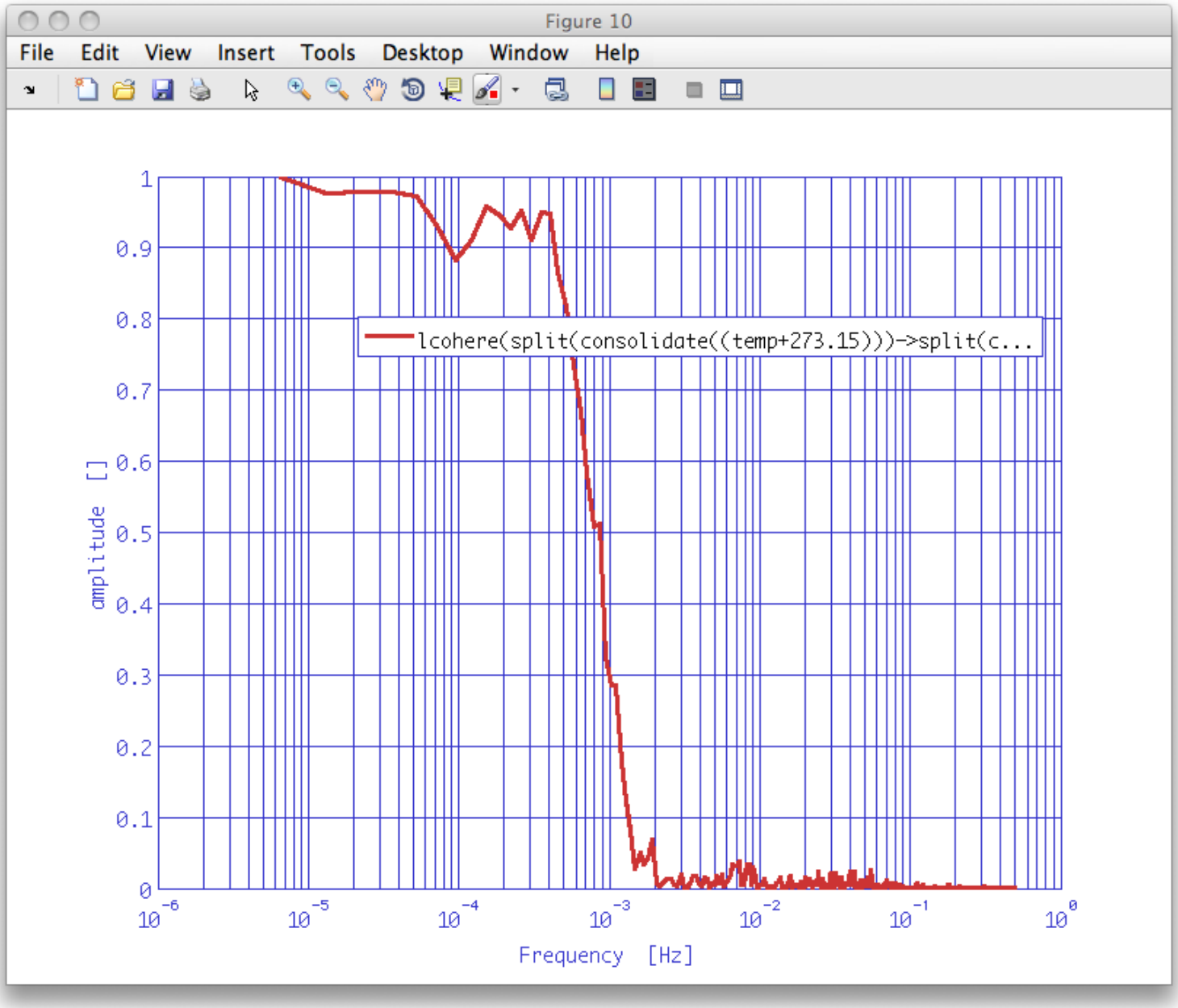
Estimating the cross-coherence

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 `ao`s, and we want to look at one of the off-diagonal terms:

```
%% Coherence estimate
coh = lcohere(T_red, x_red);
% Plot estimated cross-coherence
iplot(index(coh,1,2), 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/ltfe`
method.

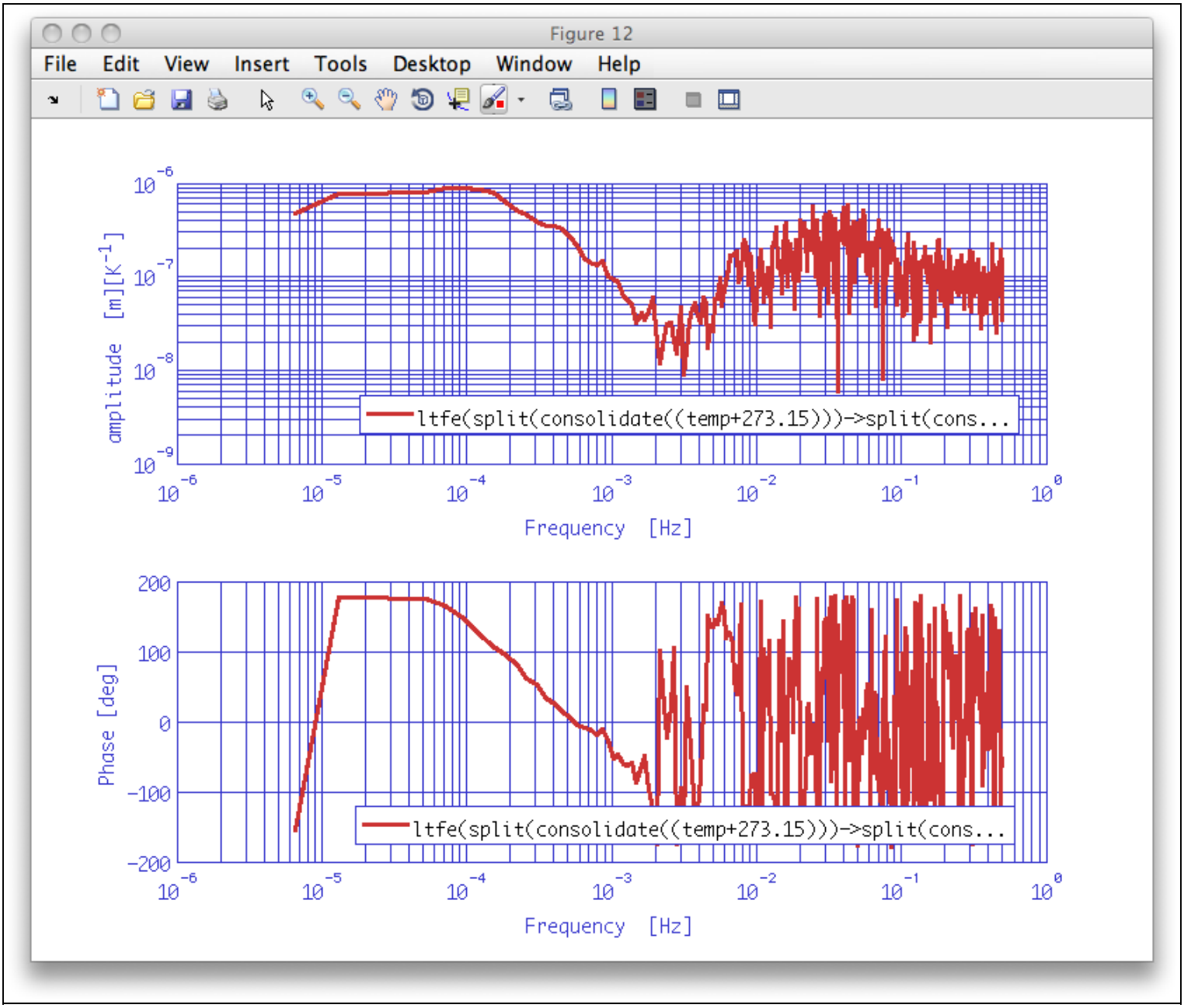
Let's also extract from the [2x2] matrix of `ao`s the result for the transfer function from T to IFO, using the

`ao/index`
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)
tfTx = index(tf,1,2);

% Plot estimated TF
```

```
iplot(tfTx);
```



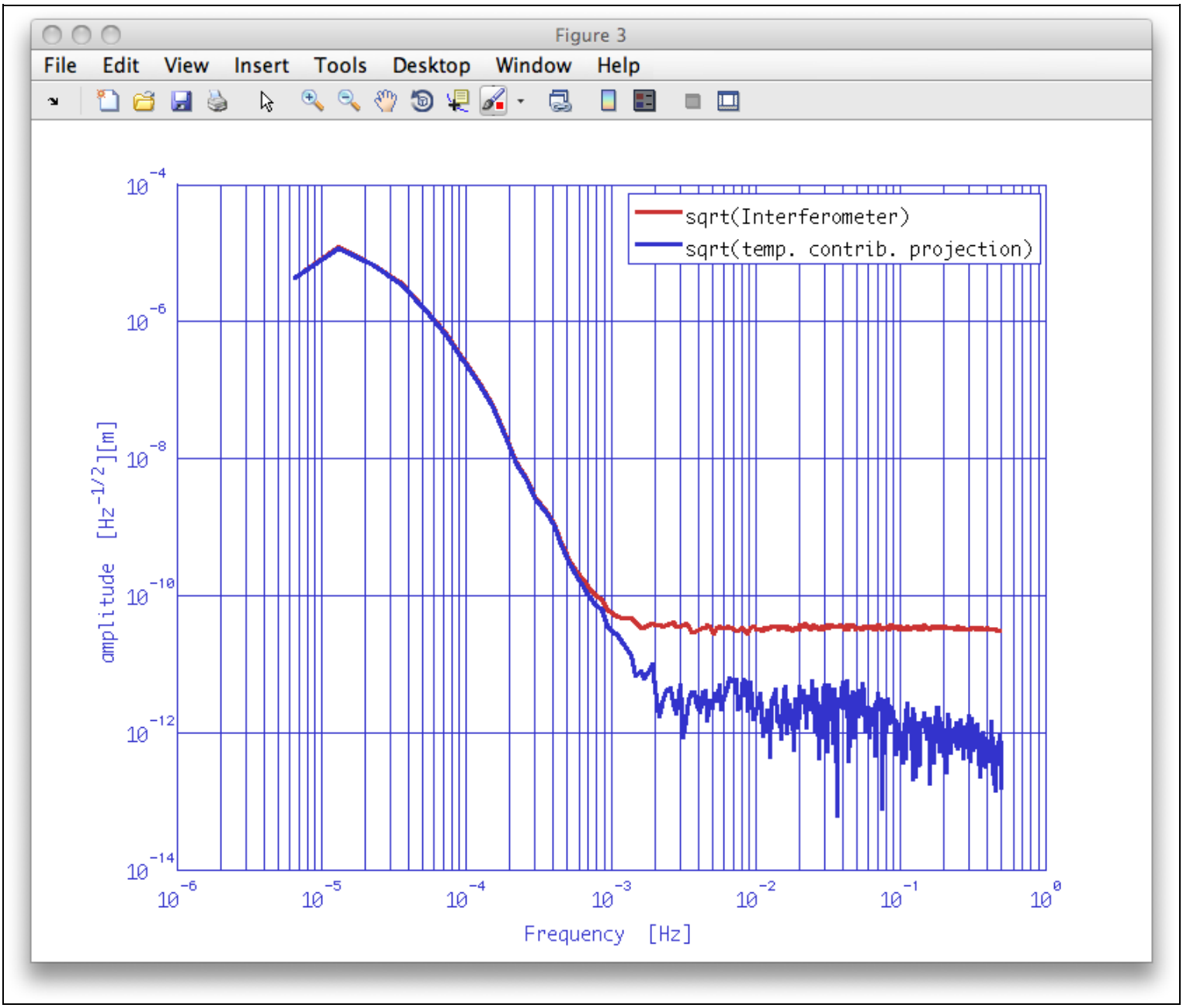
As expected, the transfer function T -> IFO is well measured at low frequencies.

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(tfTx)).^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 fininsh 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);
tfTx.save(pl_save_xT_TFE);
```

Empirical Transfer Function estimation Topic 4 - Transfer function models and digital filtering

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

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 amd 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

Topic 4 - Transfer function models and digital filtering Pole zero model representation

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)

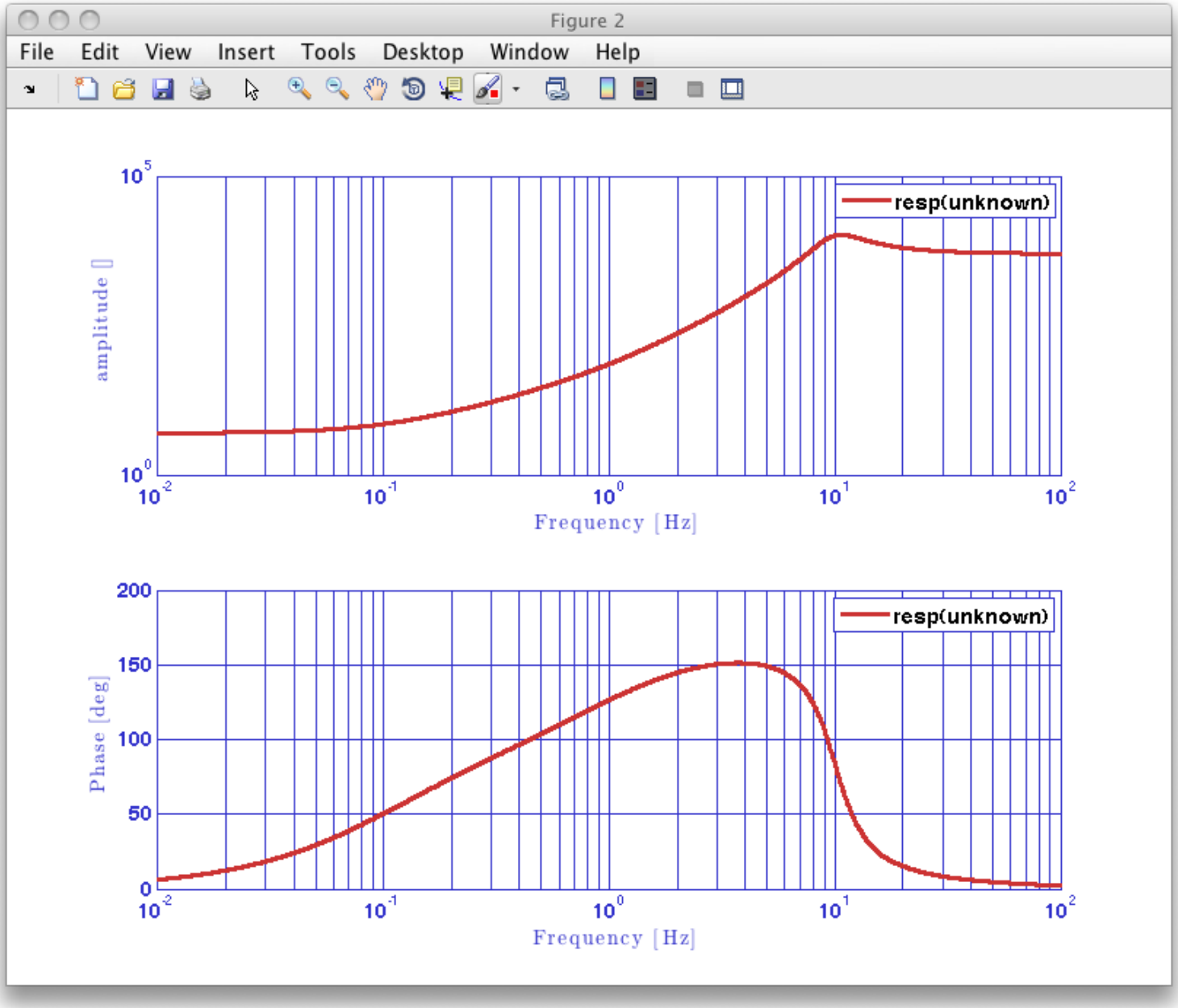
Our pole zero model has a pole at 10Hz with a quality factor of Q=2, and two zeros, one at 1Hz, one at 0.1Hz.

```
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 reals poles. The response is a exponential decay.

Create transfer function models in s domain | Partial fraction representation

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:

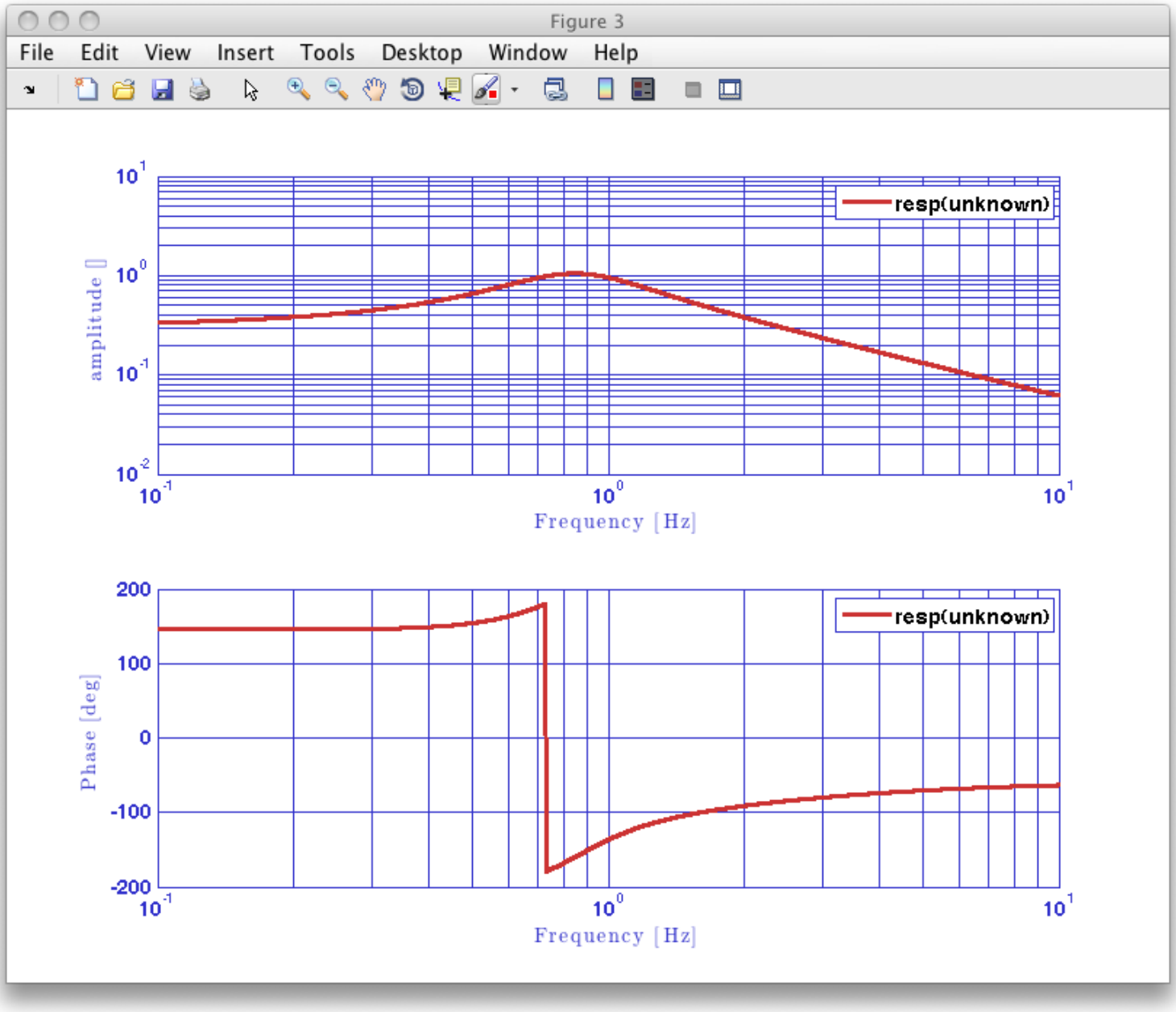
- R(1) = 1+2i, R(2) = 2
- P(1) = 10+i, P(2) = 2+5*i
- K = 0

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 a 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:

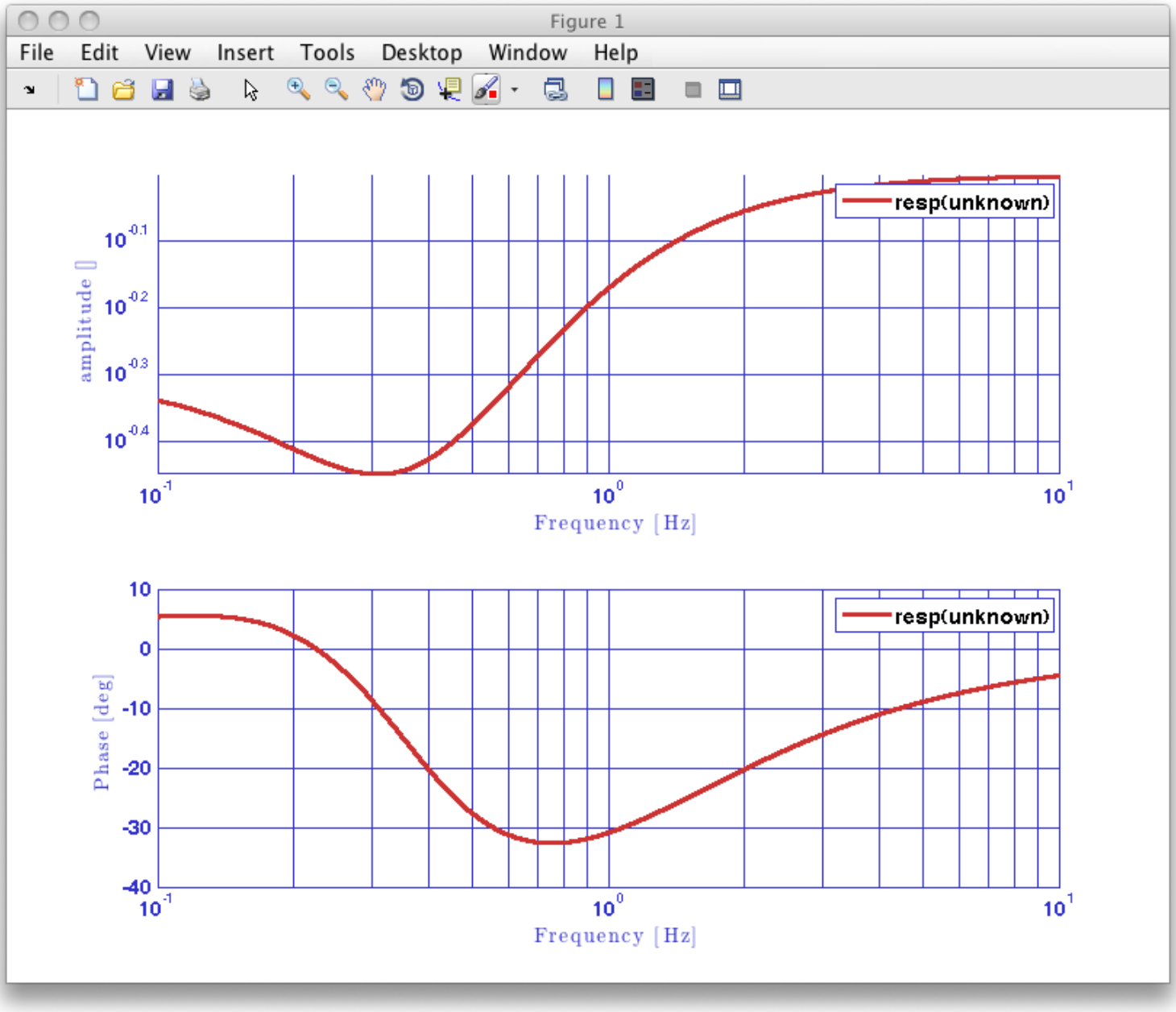
- num = [1 3 5]
- den = [1 8 10]

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))
```

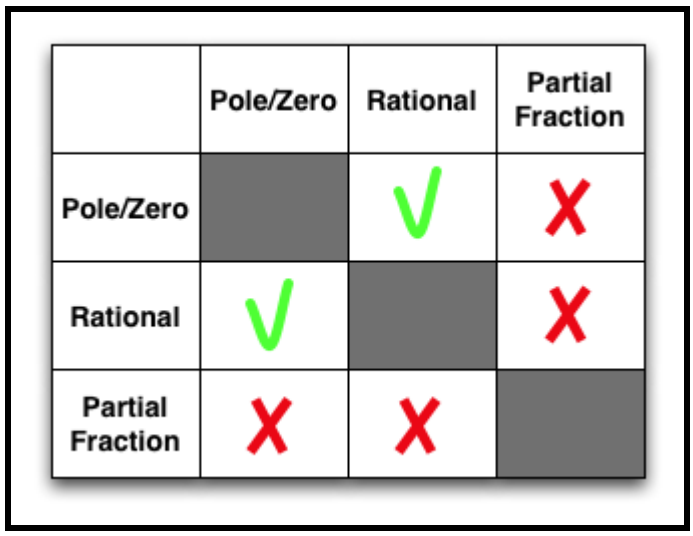


Partial fraction representation Transforming models between representations

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 (v2.0), only the transformations shown in the following table are allowed



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: []
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:   []
--------------------
```

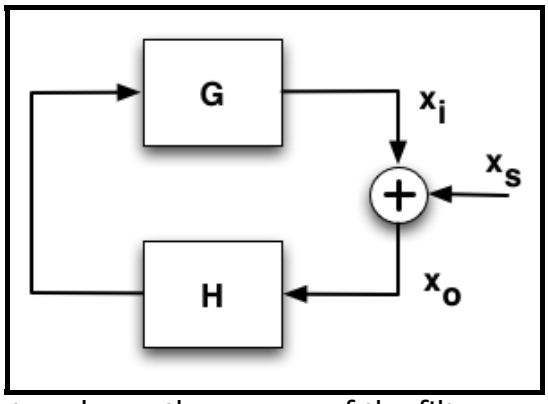
The transformation return again the first `pzmodel` however the name property allows us to follow the operations that we've been performing to this object.

```
pzm2 = pzmodel(rat)
---- pzmodel 1 ----
name: pzmodel(rational(mymodel))
gain: 5
delay: 0
iunits: []
ounits: []
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

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

$$\frac{x_o}{x_s} = \mathrm{CLG} = \frac{1}{1-\mathrm{OLG}} = \frac{1}{1-\mathrm{H}\cdot\mathrm{G}}$$

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: []
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 contructor accepts this files as inputs. We can then type

```
H = pzmodel('topic4/LISOFile.fil')
M:   load file: LISOfile.fil
---- pzmodel 1 ----
name: LISOfile
gain: 1000000000
delay: 0
iunits: []
ounits: []
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: []
    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)
    ------------------
```

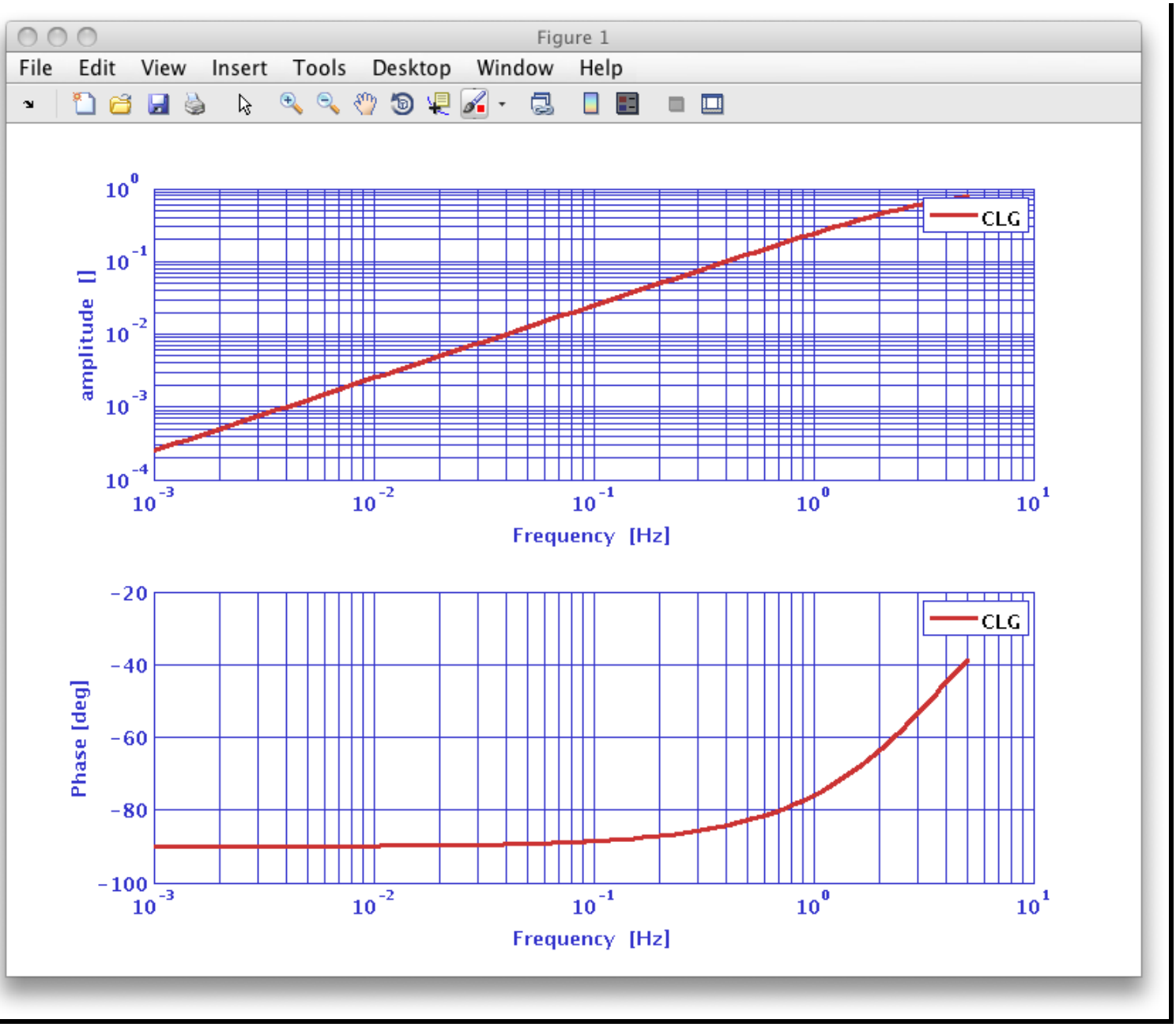
which we need to simplify to get rid of cancelling poles and zeros. We also set the model name here.

```
    G.setName;
    G.simplify
    M:   cancelling pole (f=1e-06 Hz,Q=NaN) and zero (f=1e-06 Hz,Q=NaN)
    ---- pzmodel 1 ----
    name: simplify(G)
    gain: 0.004
    delay: 0
    iunits: []
    ounits: []
    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 H model with a delay from the LISO file 'LISOFileDelay.fil'

```
>> HDel = pzmodel('topic4/LISOFileDelay.fil')
M:   load file: LISOFileDelay.fil
---- pzmodel 1 ----
    name: LISOFileDelay
    gain: 1000000000
   delay: 0.125
  iunits: []
  ounits: []
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 substracted if the models are divided.

Transforming models between representations

How to filter data

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 system By discretizing transfer function models

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) 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)
```

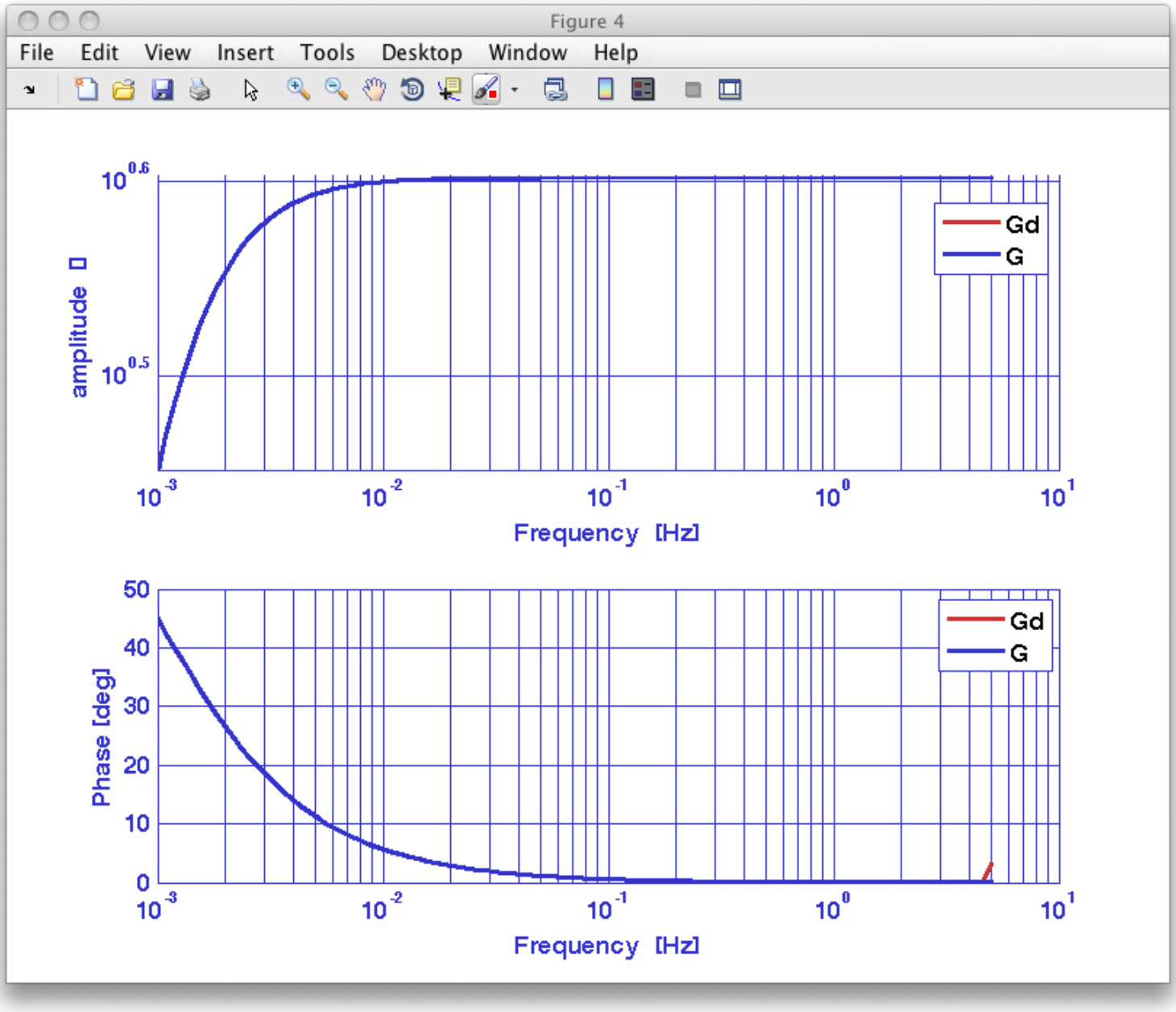


Figure 4
File Edit View Insert Tools Desktop Window Help
amplitude
Gd
G
Frequency [Hz]
Phase [deg]
Gd
G
Frequency [Hz]

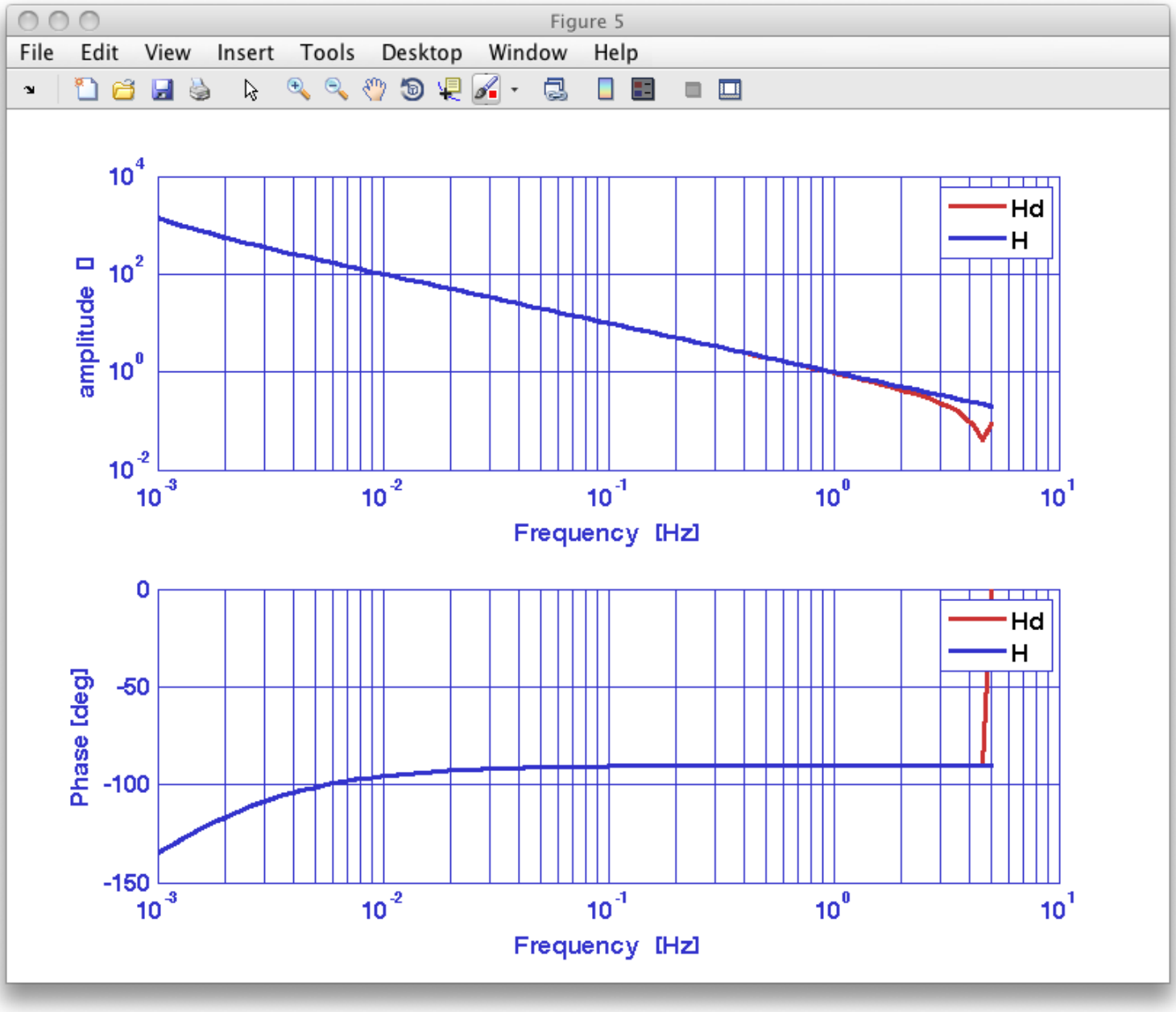
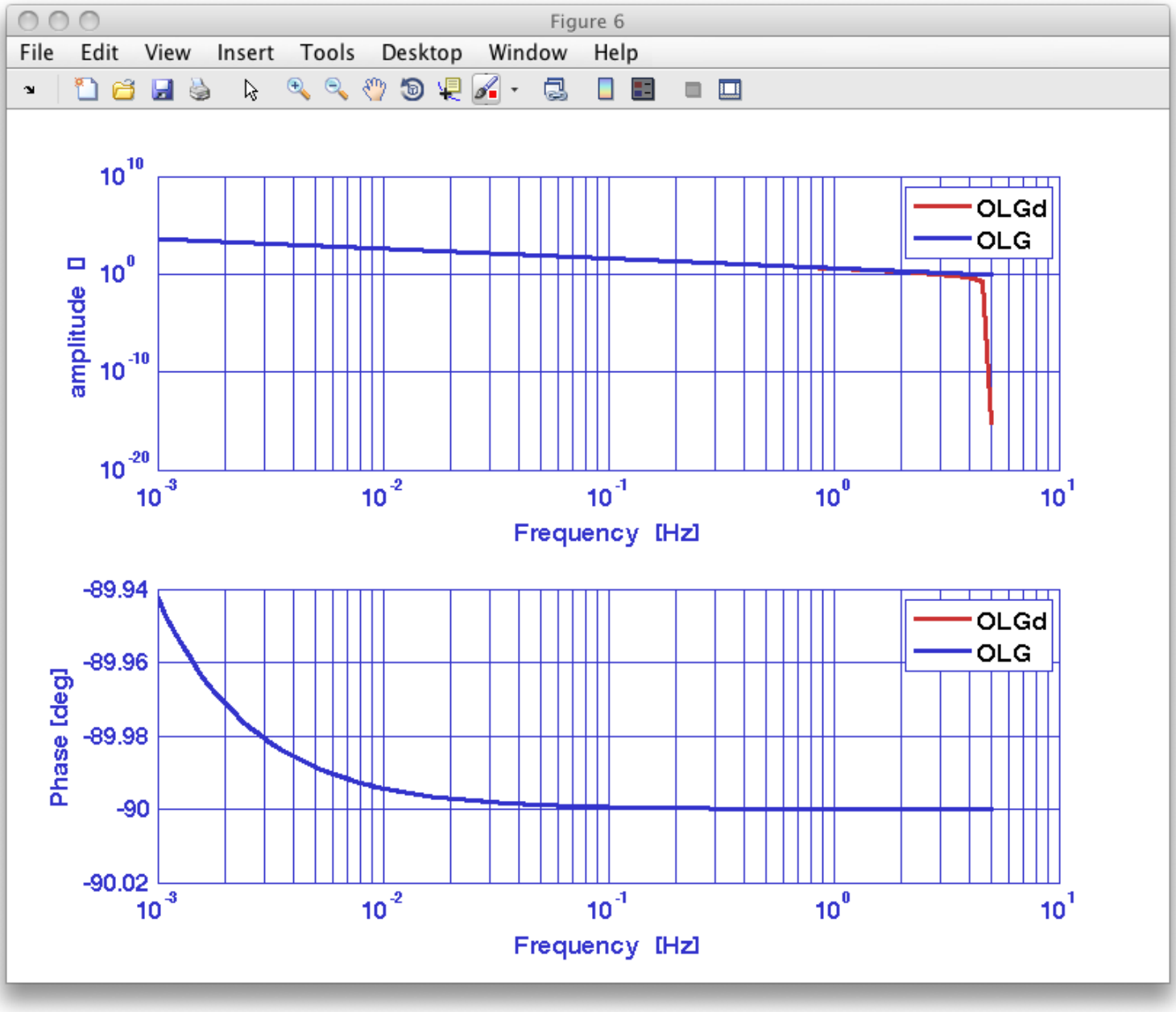


Figure 5
File Edit View Insert Tools Desktop Window Help
amplitude
Hd
H
Frequency [Hz]
Phase [deg]
Hd
H
Frequency [Hz]



Once the comparison is done, one is usually interested in the coefficients of the digital implemetation. These are directly accesible 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 (v2.0) 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

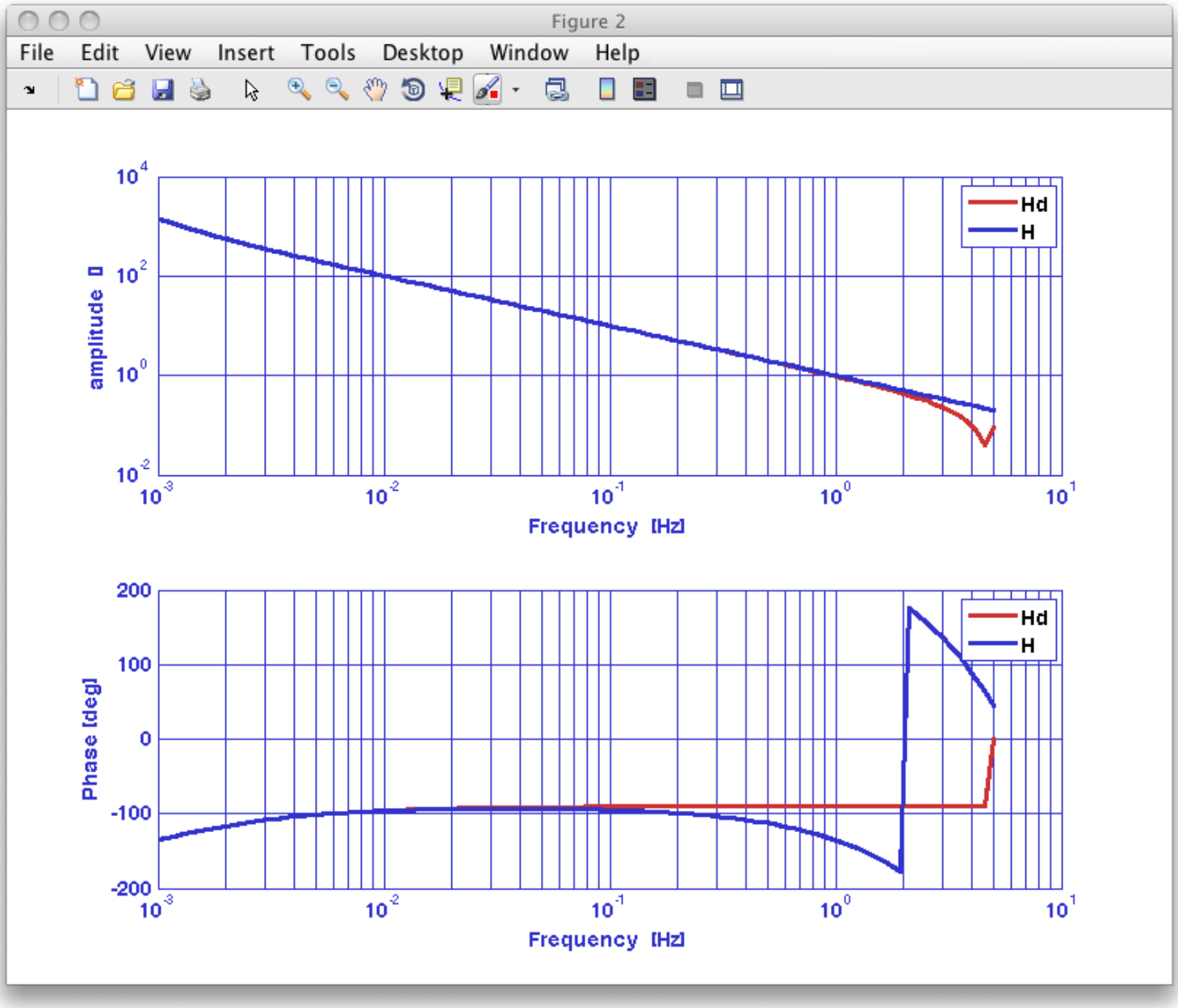


Figure 2
File Edit View Insert Tools Desktop Window Help
amplitude []
Frequency [Hz]
Hd
H
Phase [deg]
Frequency [Hz]
Hd
H

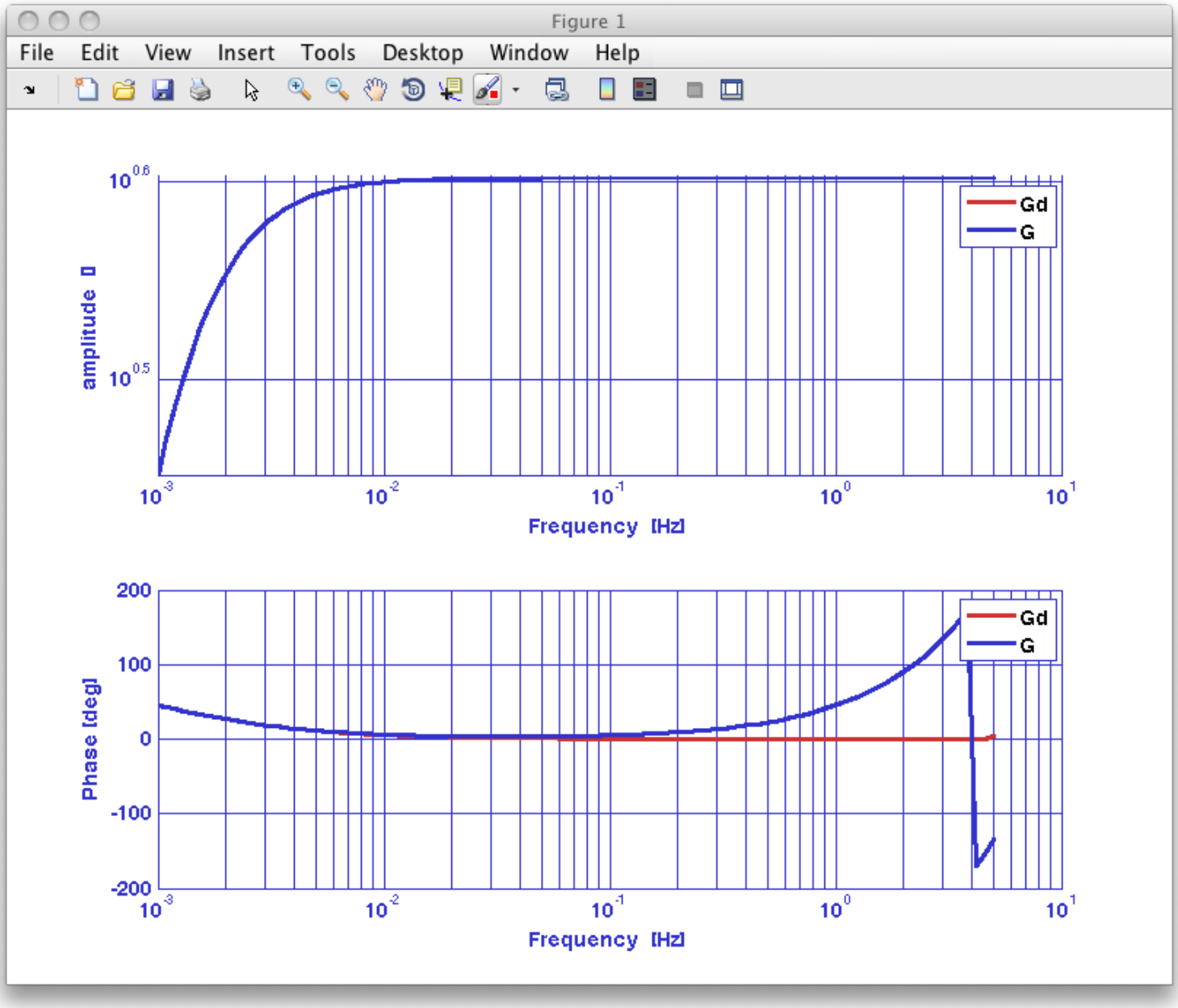
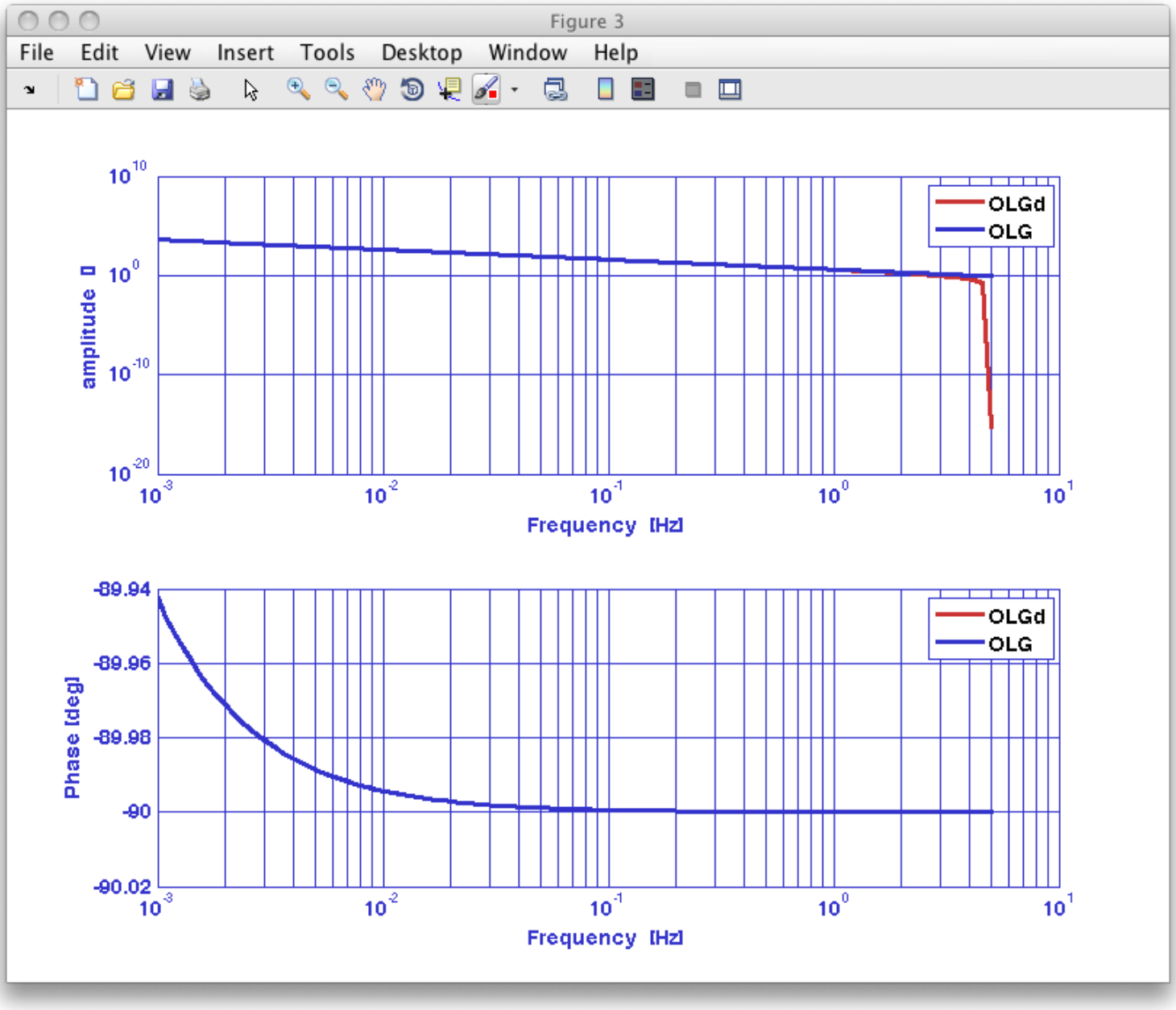


Figure 1
File Edit View Insert Tools Desktop Window Help
amplitude []
Frequency [Hz]
Gd
G
Phase [deg]
Frequency [Hz]
Gd
G



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.

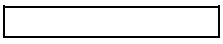
Estimating power spectrum in a limited band

In this example we want to estimate the power spectrum of an interferometer time series in the LTP bandwidth, [1e-3,30e-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

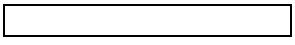
Key	Value
TYPE	'bandpass'
ORDER	2
FS	3.247
FC	[5e-4 5e-2]

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. Athough 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 frequecy 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(AO,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

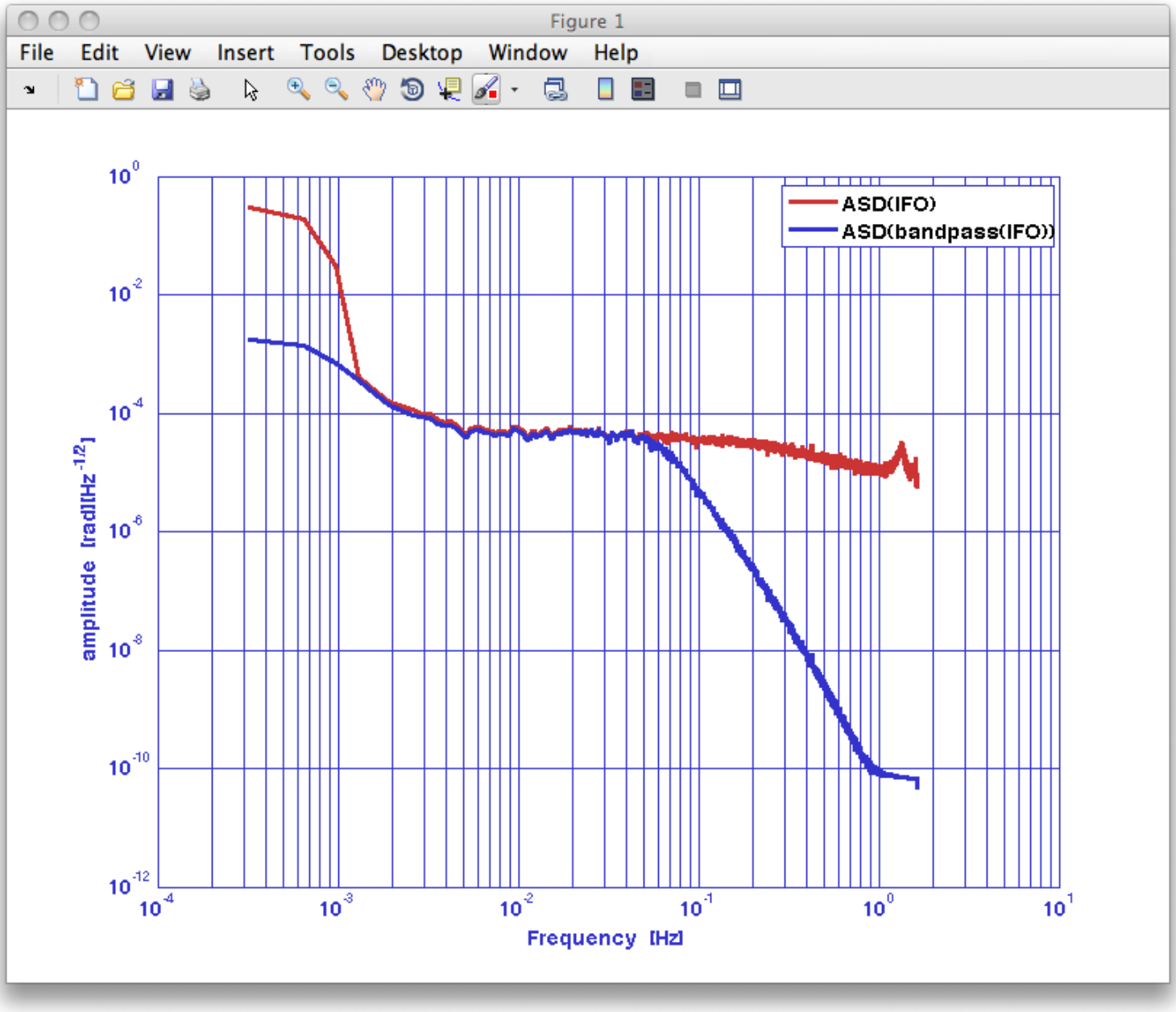
Key	Value
NFFT	1e4
SCALE	'ASD'

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 `iplot`

```
p.iplot
```

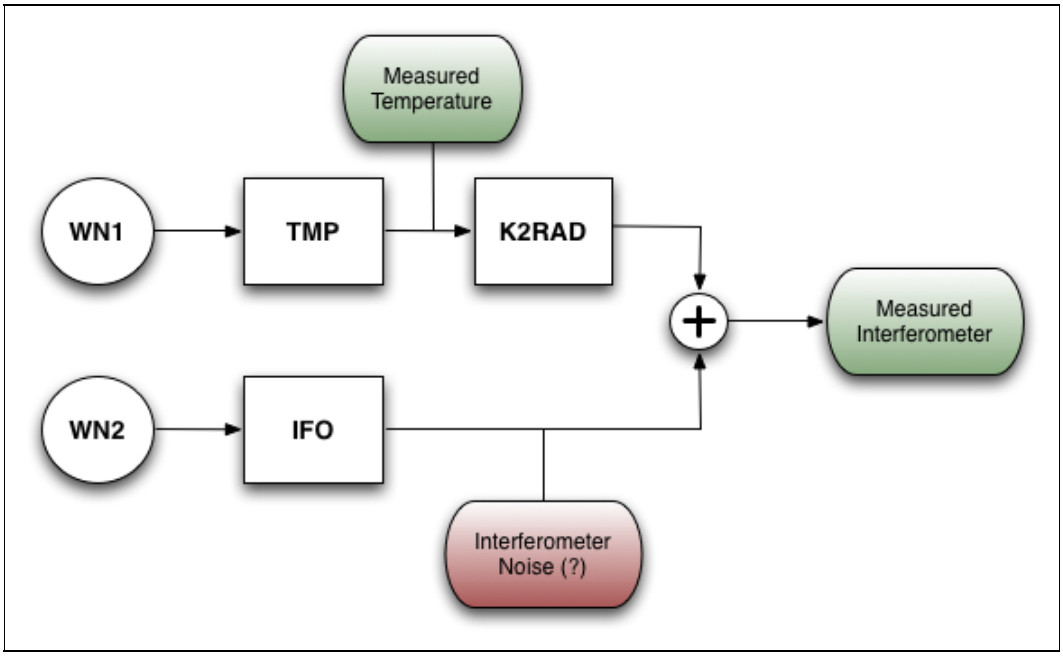


By discretizing transfer function models
IFO/Temperature Example - Simulation

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.setOunits('K')
TMP.setName('TMP')
```

STEP 2: Build a interferometer noise PZMODEL with the following properties

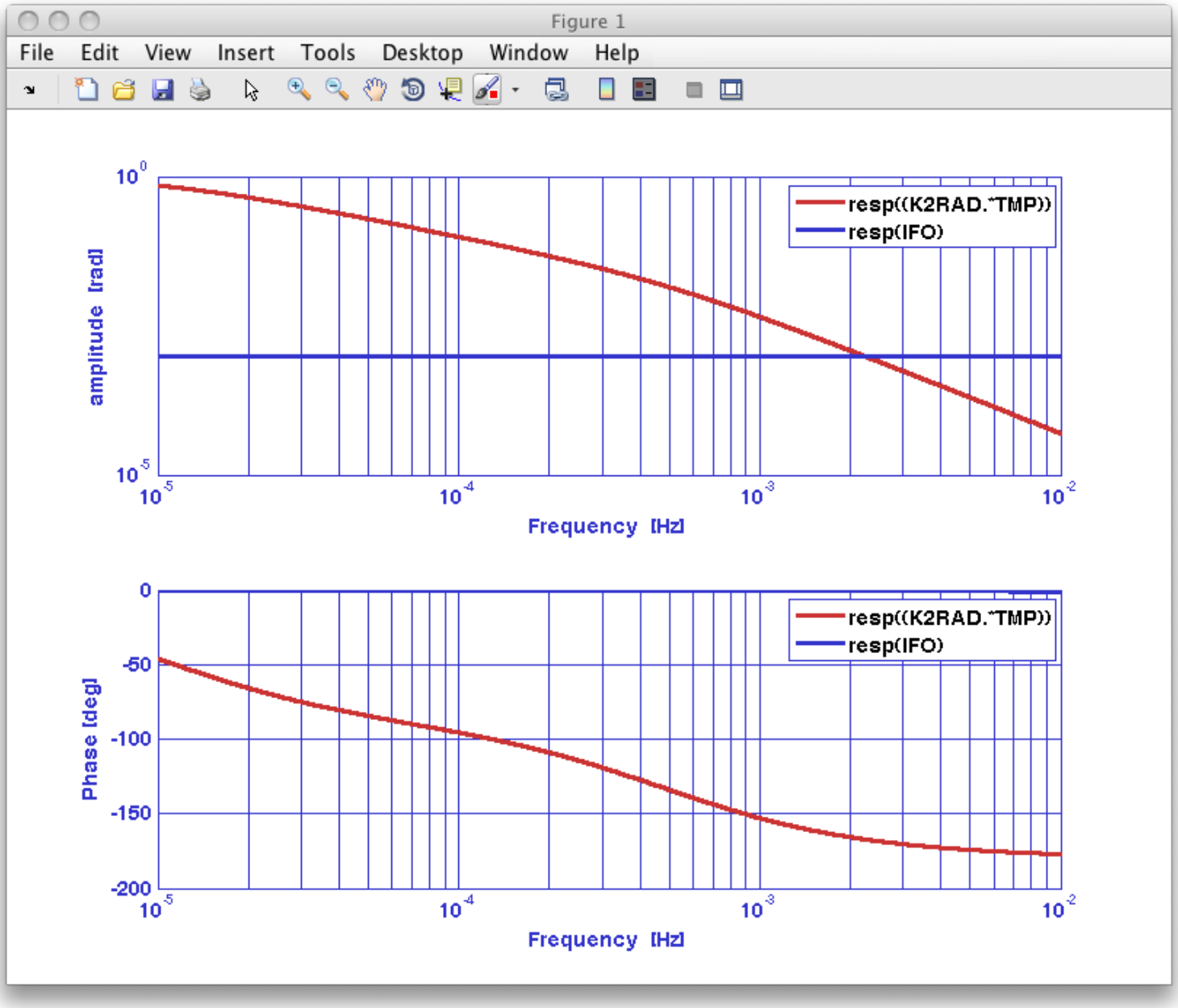
Key	Value
'name'	'IFO'
'ounits'	'rad'
GAIN	1e-3
POLE 1	0.4

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 fs = 1Hz 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**

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,[EMPd K2Rd],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.

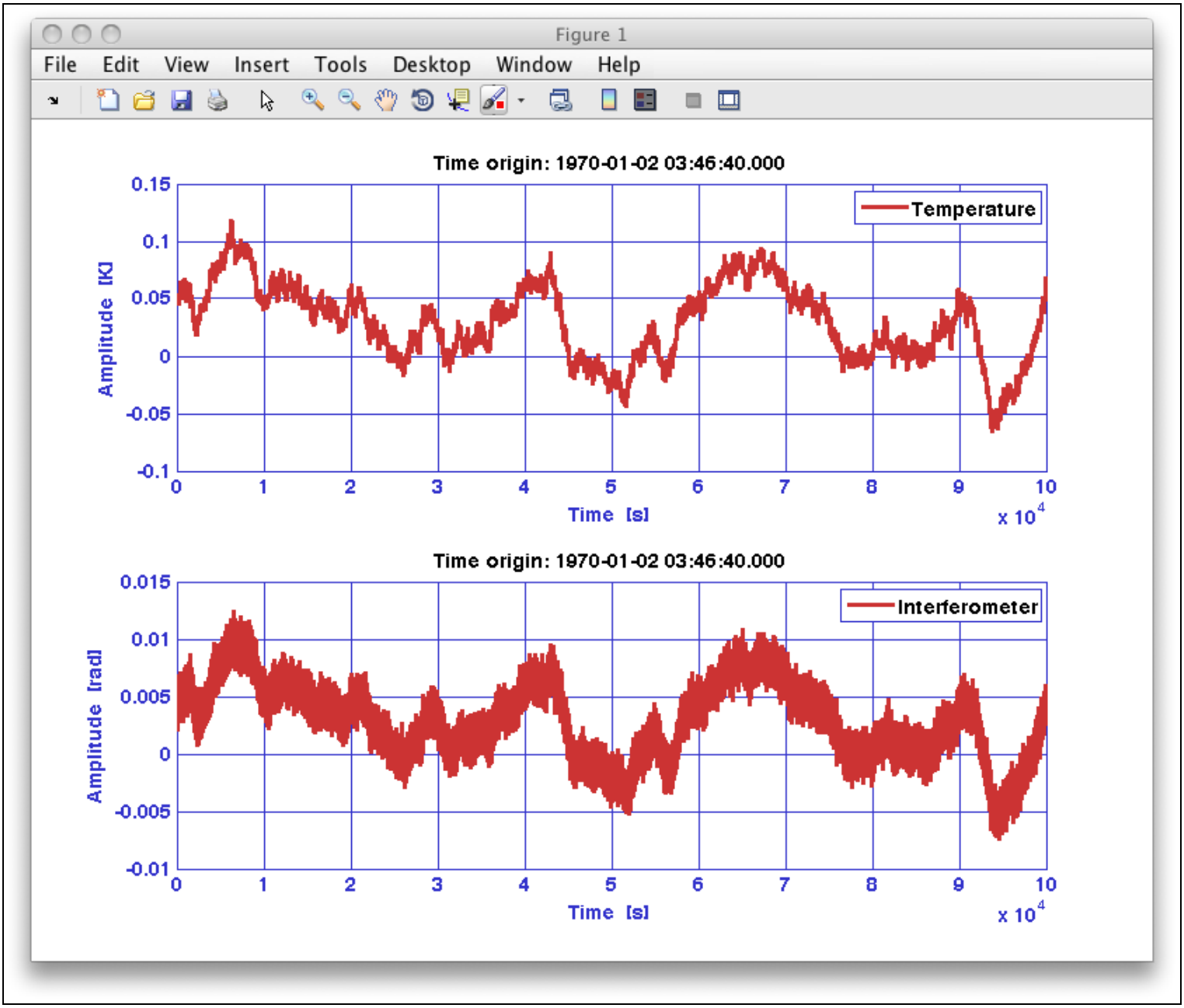
STEP 10: Split the data streams

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'))
```

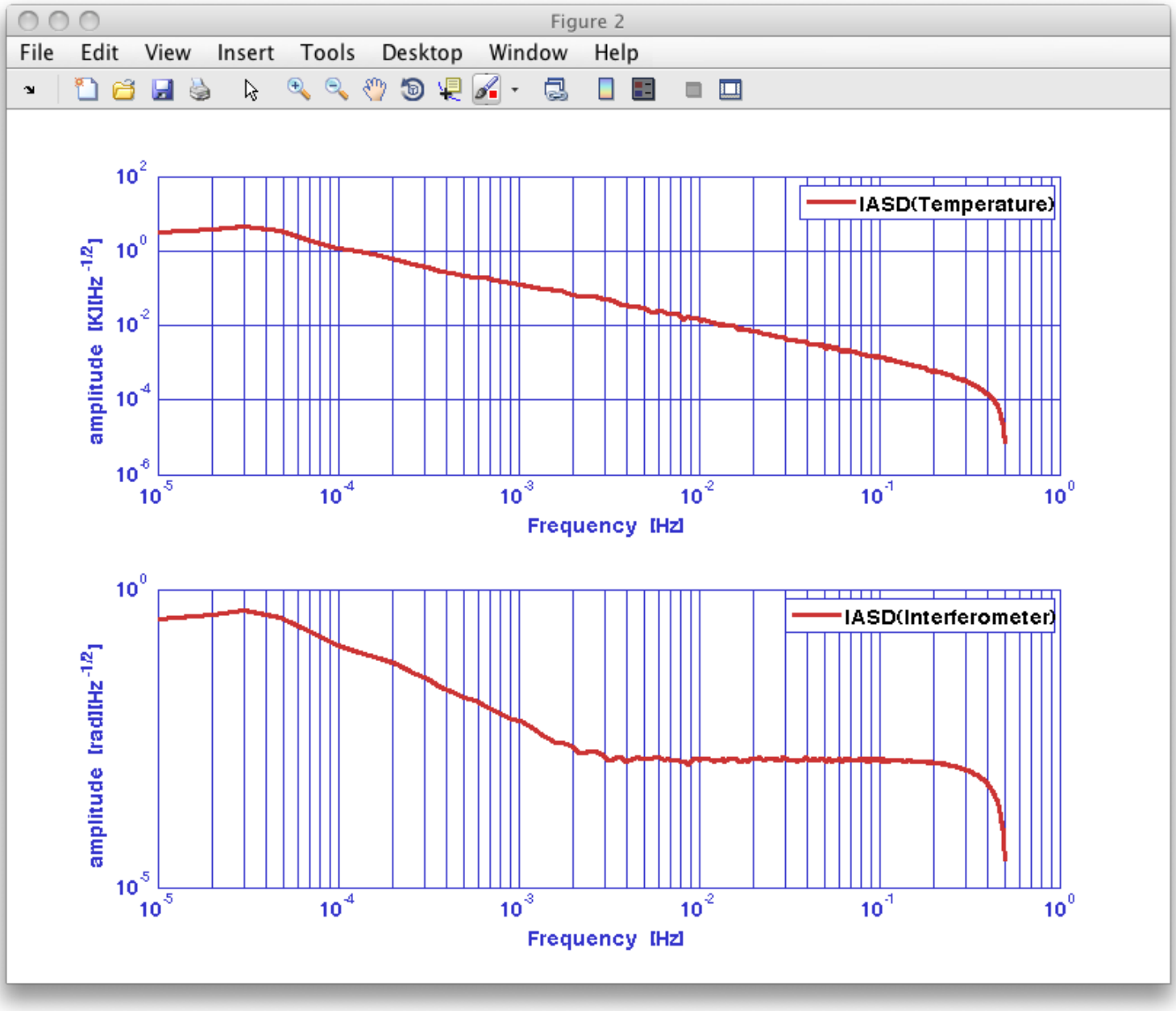


STEP 11: Compute power spectral estimates for the temperature and interferometric data

Here you need to apply `lpsd` or `psd` methods. For example:

```
pl = plist('order',1,'scale','ASD')
psd_T = lpsd(T,pl)
psd_ifo = lpsd(ifo,pl)
```

The resulting spectrum should look like this



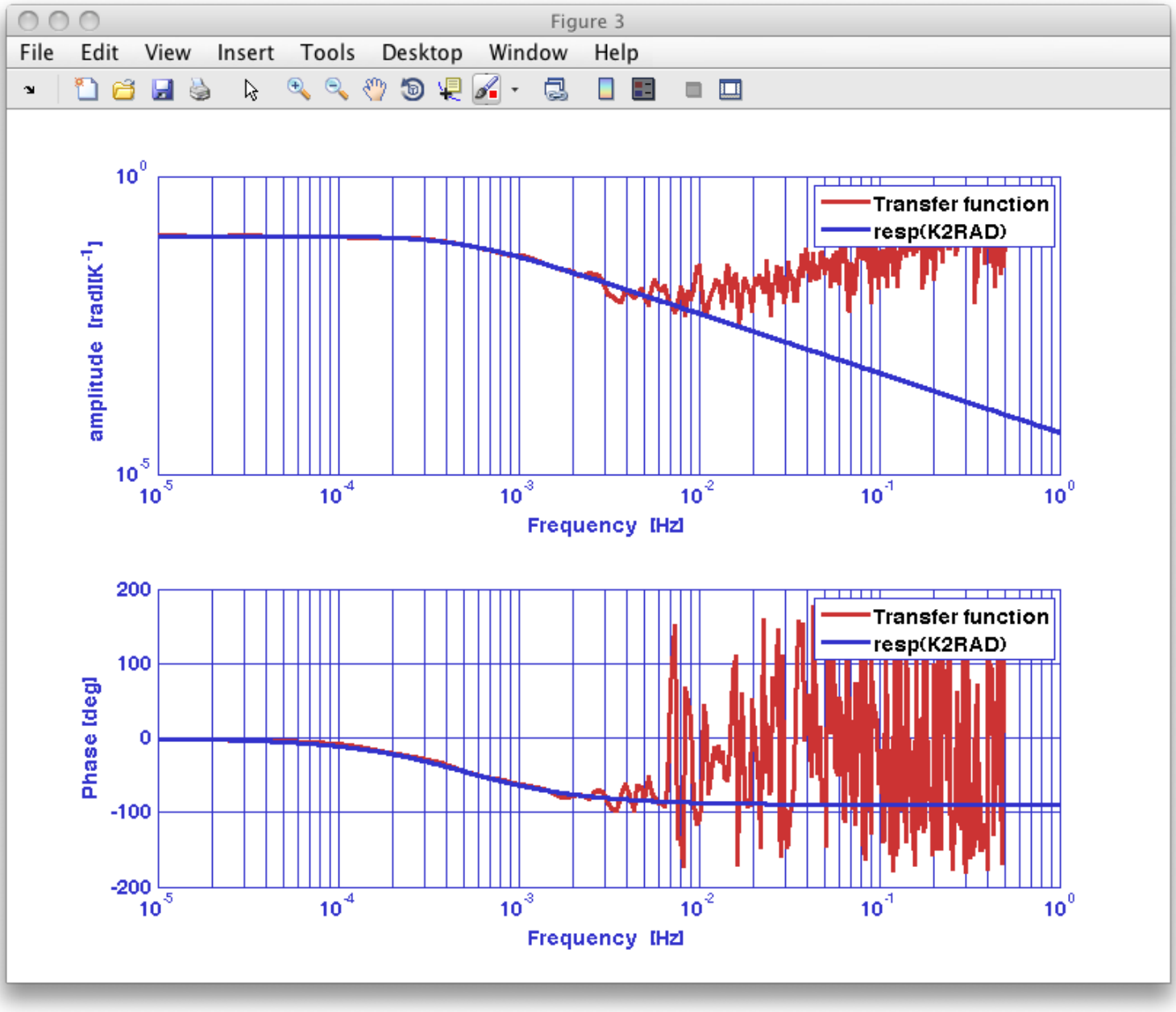
STEP 12: Compute transfer function estimate for the temperature and interferometric data

Here you need to apply `ltfe` or `tfe` methods. For example:

```
T2ifo = ltfe(T,ifo)
T2ifo.setName('Transfer function')
```

You can now compare the transfer function model with the estimation obtained from the data:

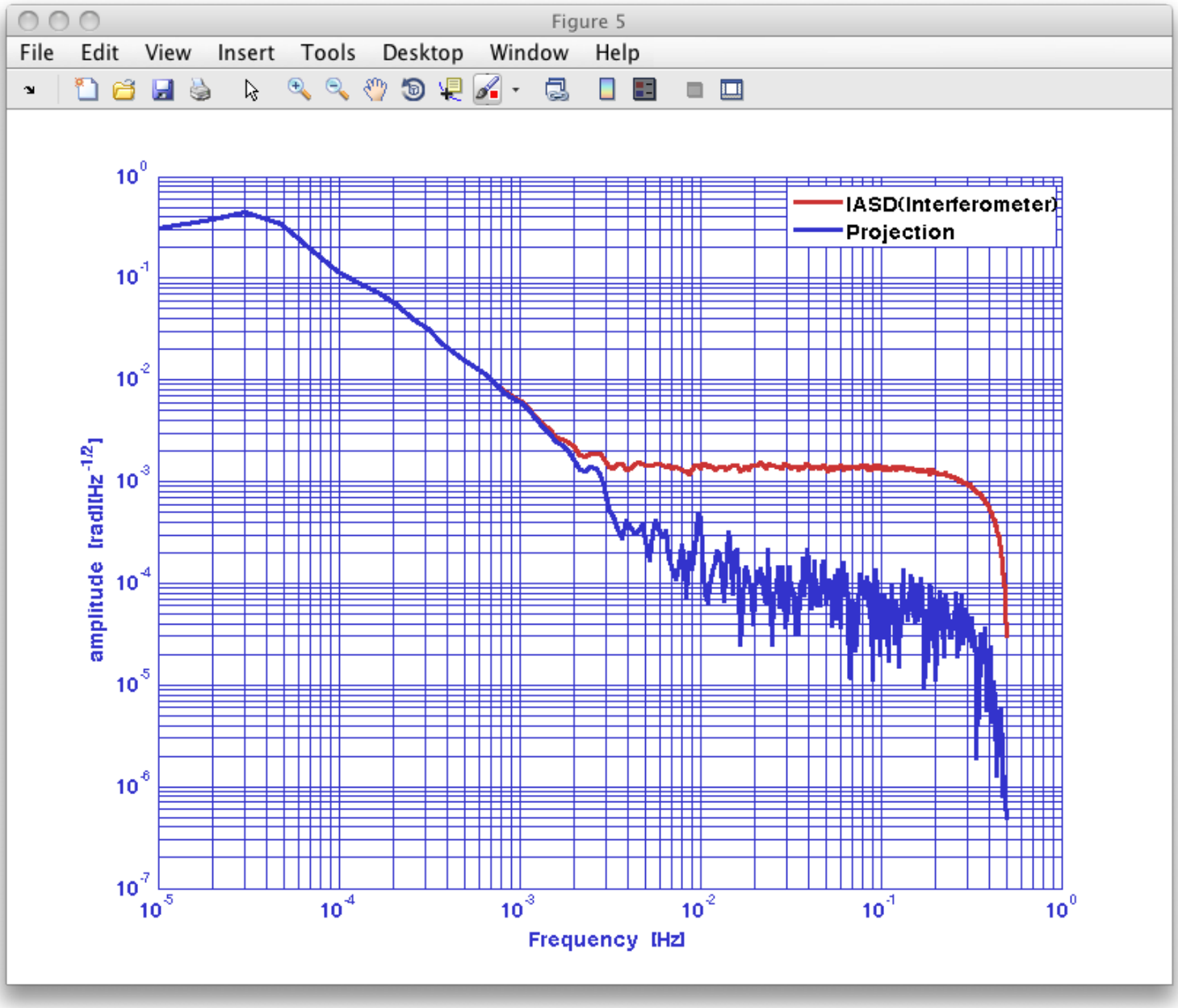
```
pl = plist('f1',1e-5,'f2',1)
iplot(T2ifo(1,2),resp(K2RAD,pl))
```



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(T2ifo(1,2)).*psd_T;
Projection.simplifyYunits
Projection.setName;
% Plot against interferometer noise
iplot(psd_ifo,Projection)
```



By defining filter properties

Topic 5 - Model fitting

Topic 5 - Model fitting

Topic 5 of the training session aims to biefly 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 polynimials
- 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

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 agorithm can be found in the Z-domain fit help 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 metres ('m').

Now we are ready to move on the second step where we will:

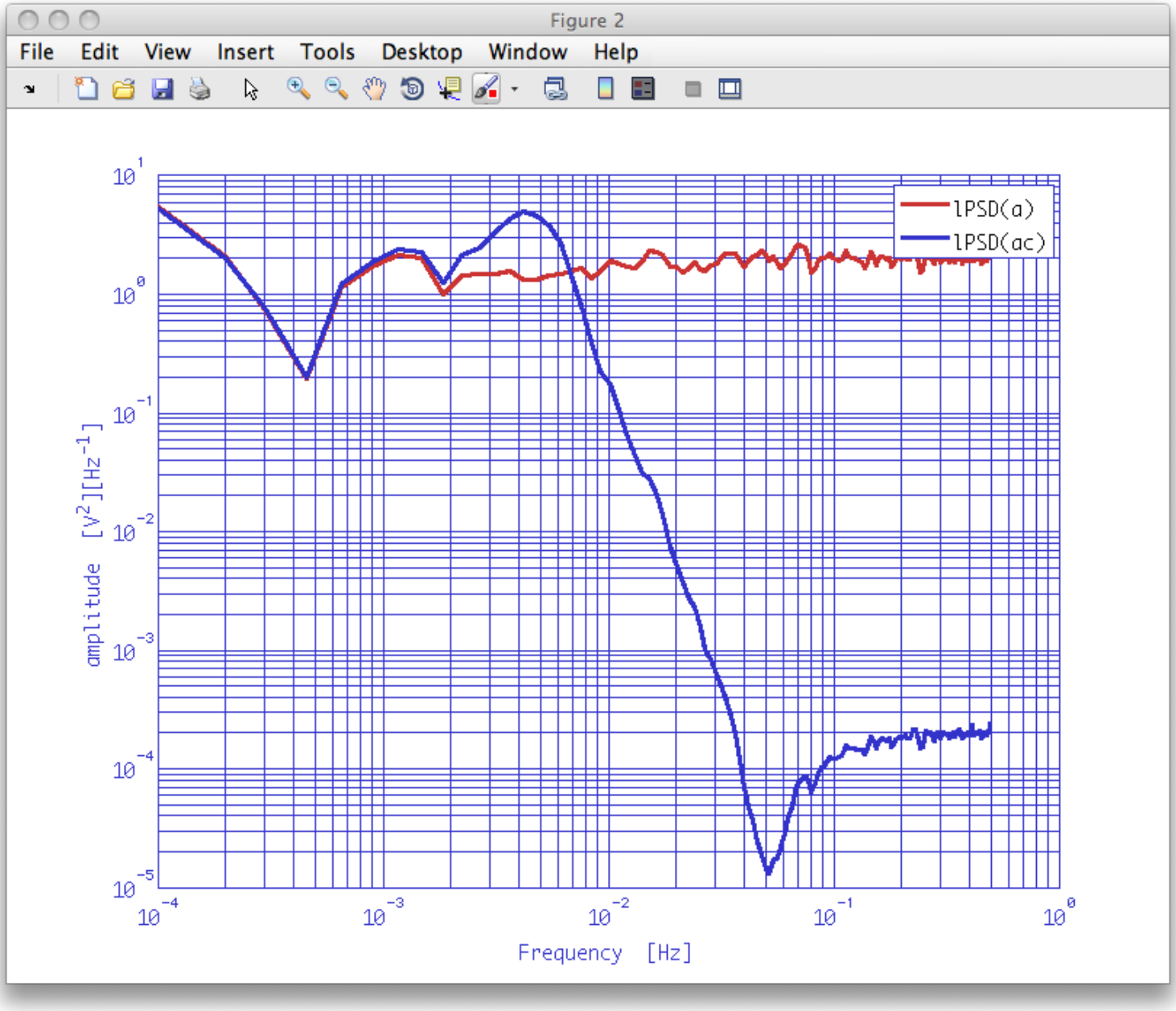
- Build a pole-zero model (`pzmodel`)
- Construct a `miir` filter from the `pzmodel`
- filter white noise data with the `miir` filter in order to obtain a colored noise time series.

```
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 calcualte PSD of the data in order to check the difference between the coloured noise and the white noise.

```
axx  = lpsd(a);
acxx = lpsd(ac);
iplot(axx,acxx)
```

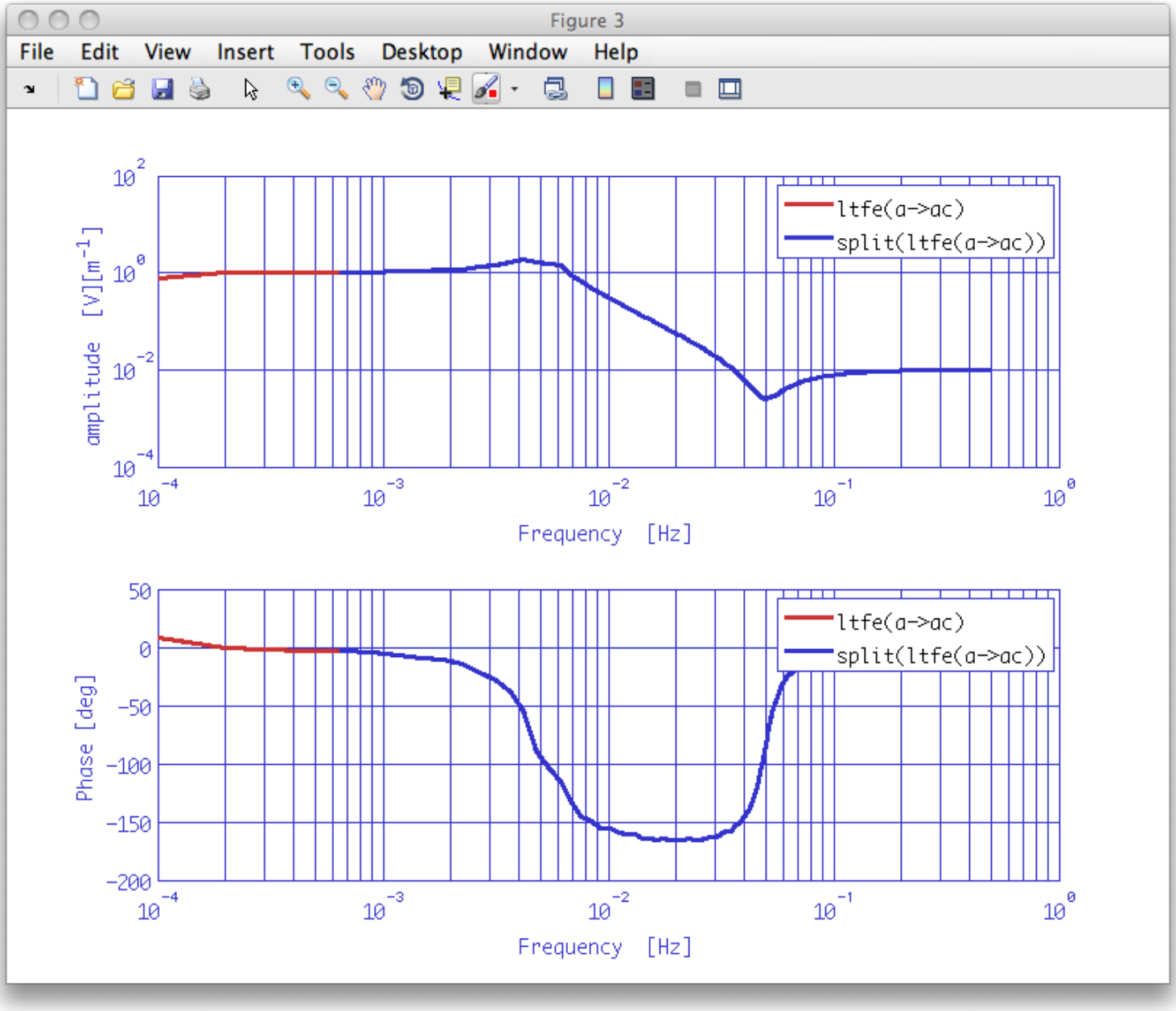
You should see something similar to this picture.



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);
tf = tf.index(1,2);
tfsp = split(tf,plist('frequencies', [5e-4 5e-1]));

iplot(tf,tfsp)
```

The plot looks like the following:



It is now the moment to start fitting with `zDomainFit`. As reported in the function help 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:

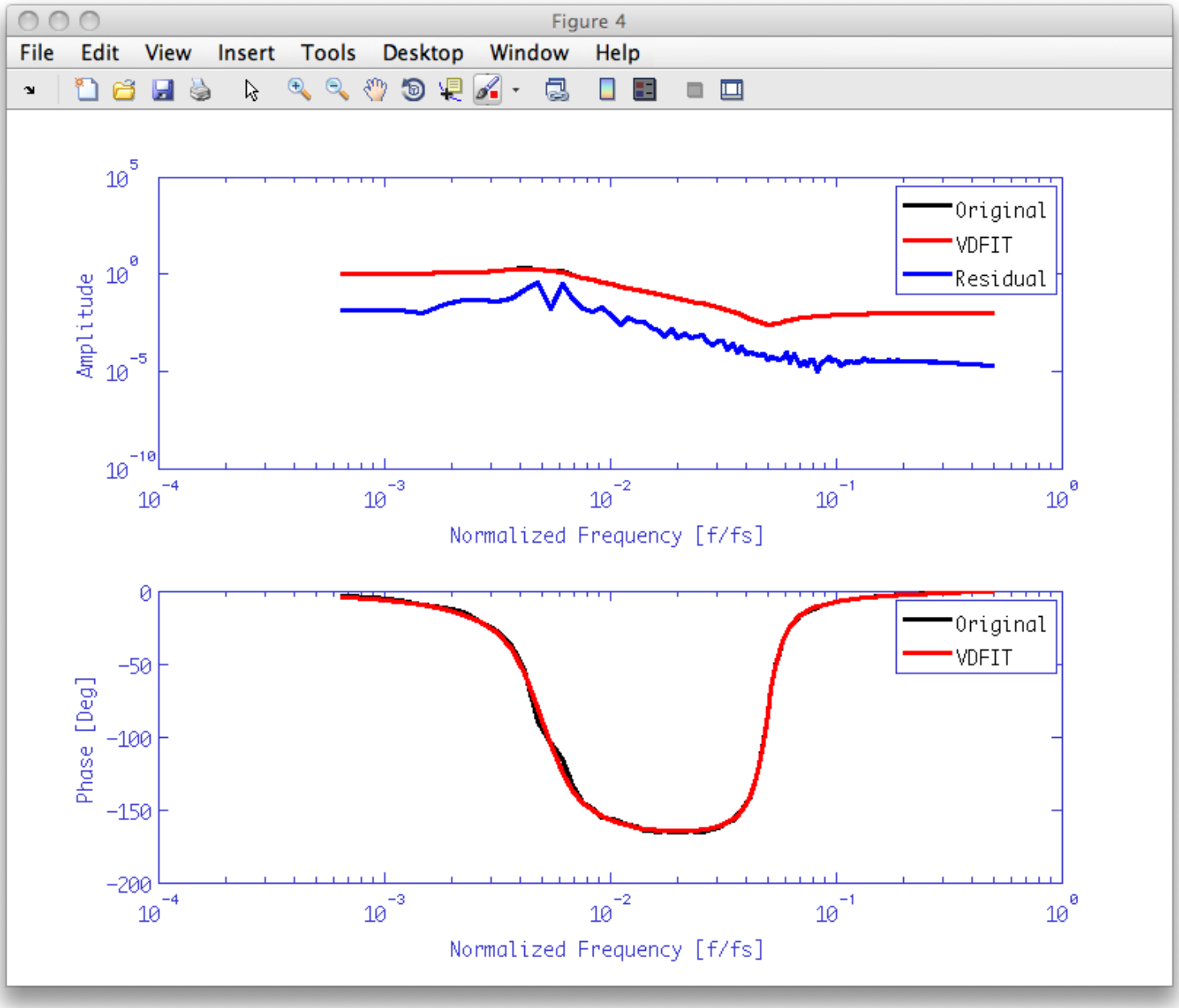
Key	Description
`ResLogDiff`	This makes `zDomainFit` check that the normalized difference between the data and the fit residuals (in log scale) is larger than the value specified. If `d` is the input value for the parameter, then the algorithm checks that `log10(data) - log10(res) > d`.
`RMSE`	If `r` 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','c1',...   % 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.

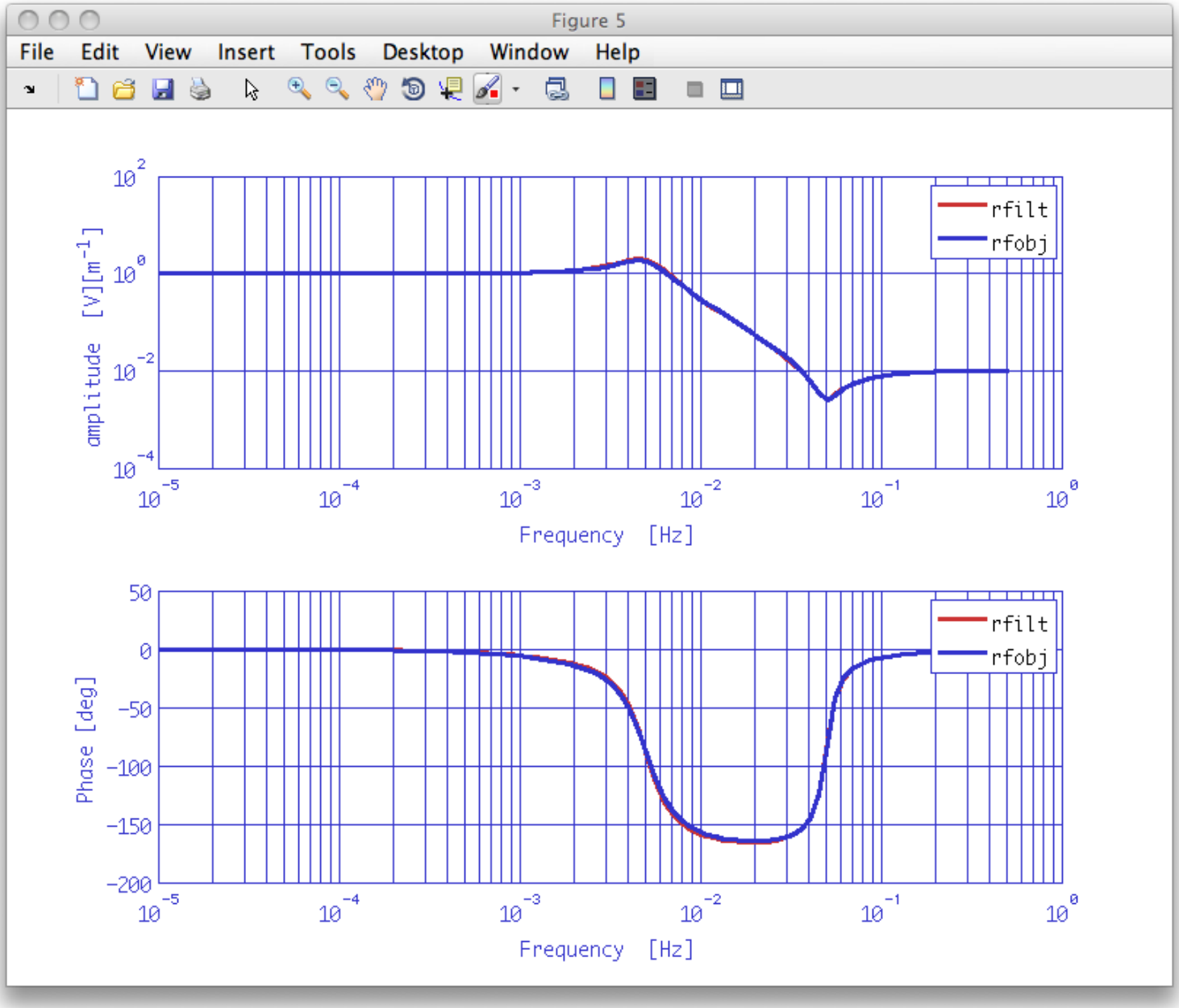


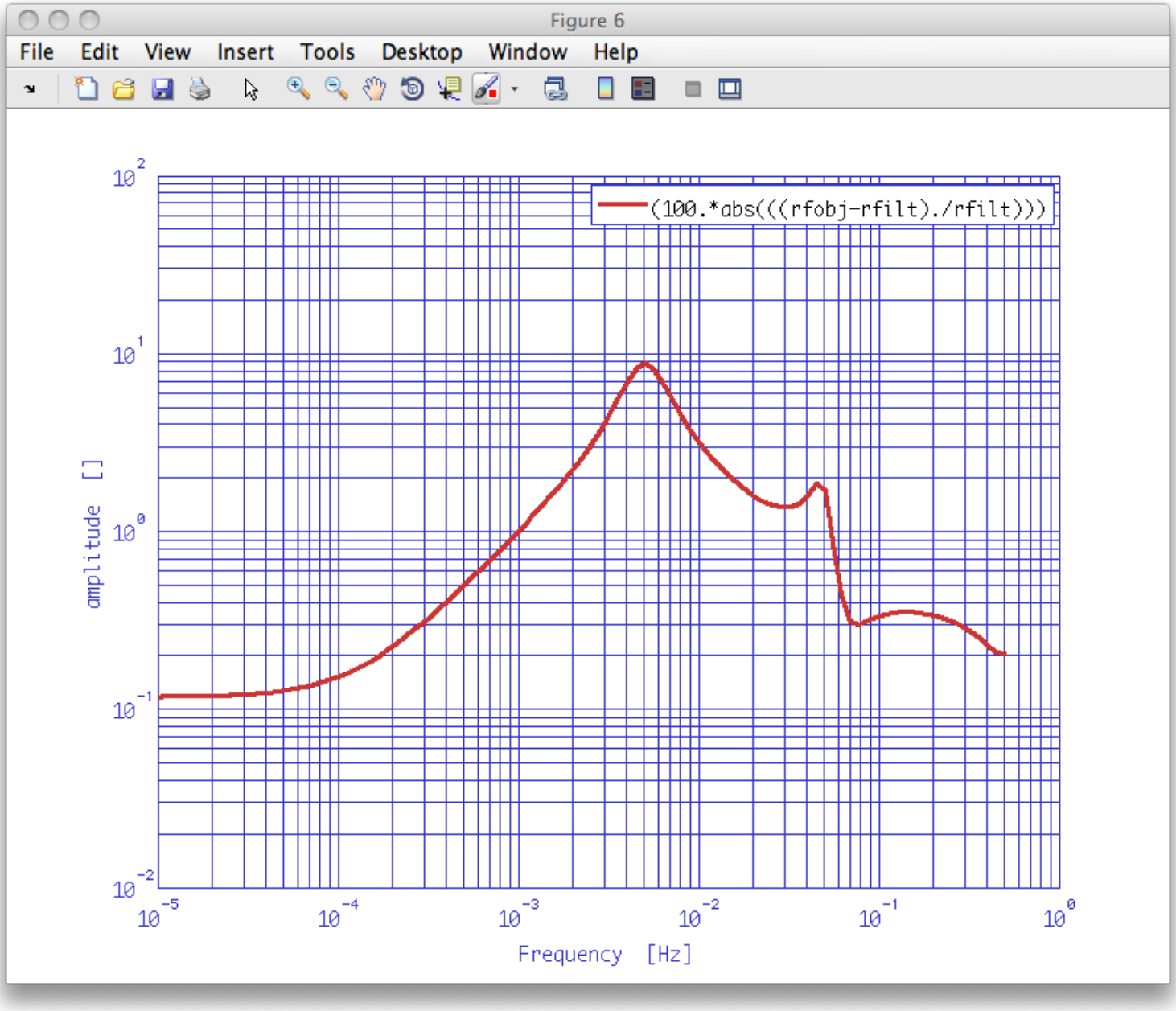
We can now check the result of our fitting procedures. We calculate frequency response of the fitted models (filters) and compare them with the starting IIR filter response, then we will plot the percentage error on the filters magnitudes.

Note that the result of the fitting procedure is a parallel bank of 3 IIR filters.

```
% 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,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%.





Topic 5 - Model fitting
Generation of noise with given psd

Generation of noise with given psd

Generation of model noise is performed with the function ao/noisegen1D. Details on the agorithm 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`.

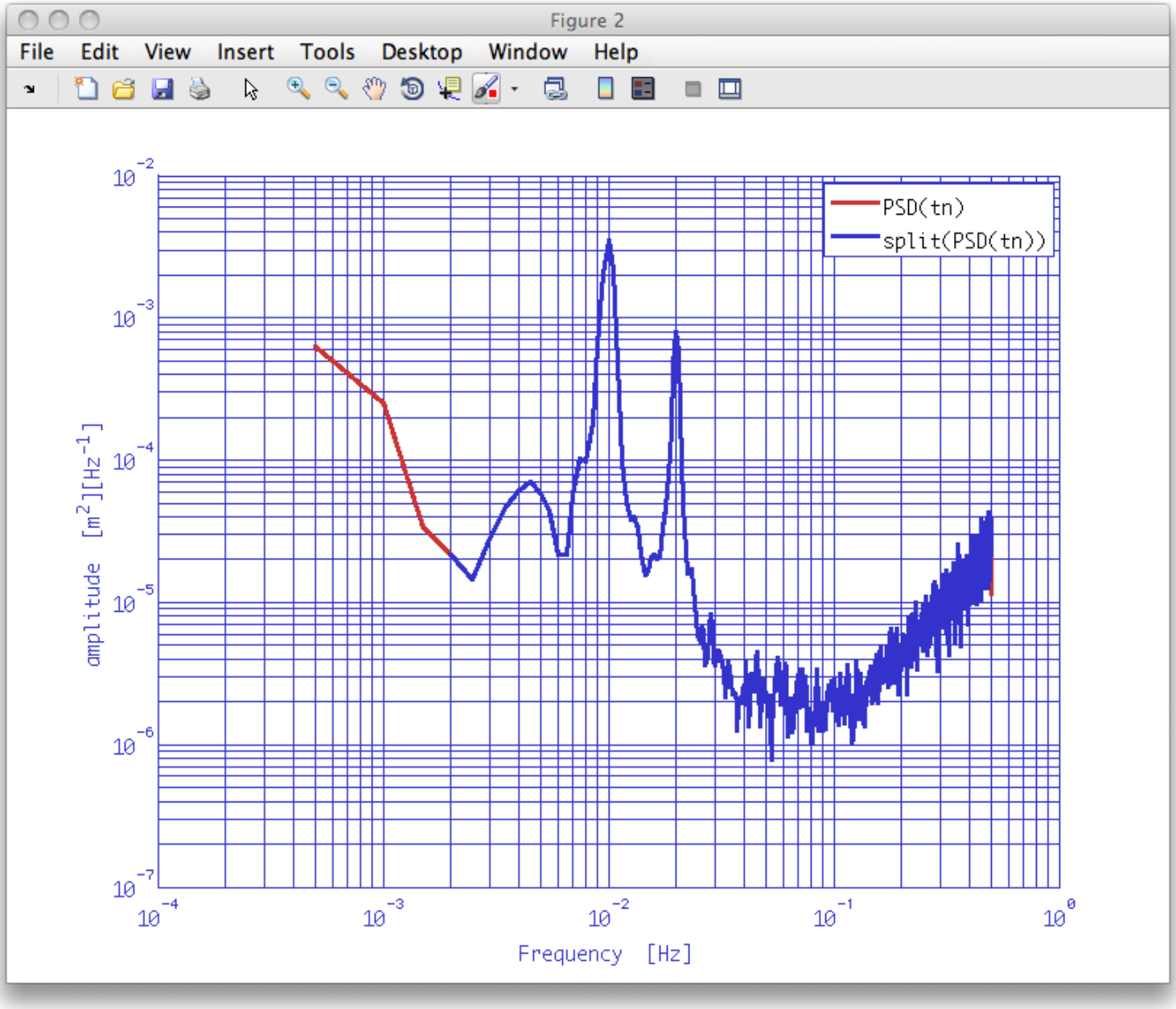
Now do the psd of our data. we need some average otherwise the fitting algorithm is not able to run correctly.

```
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]));
iplot(tnxx,tnxxr)
```

The result should look like

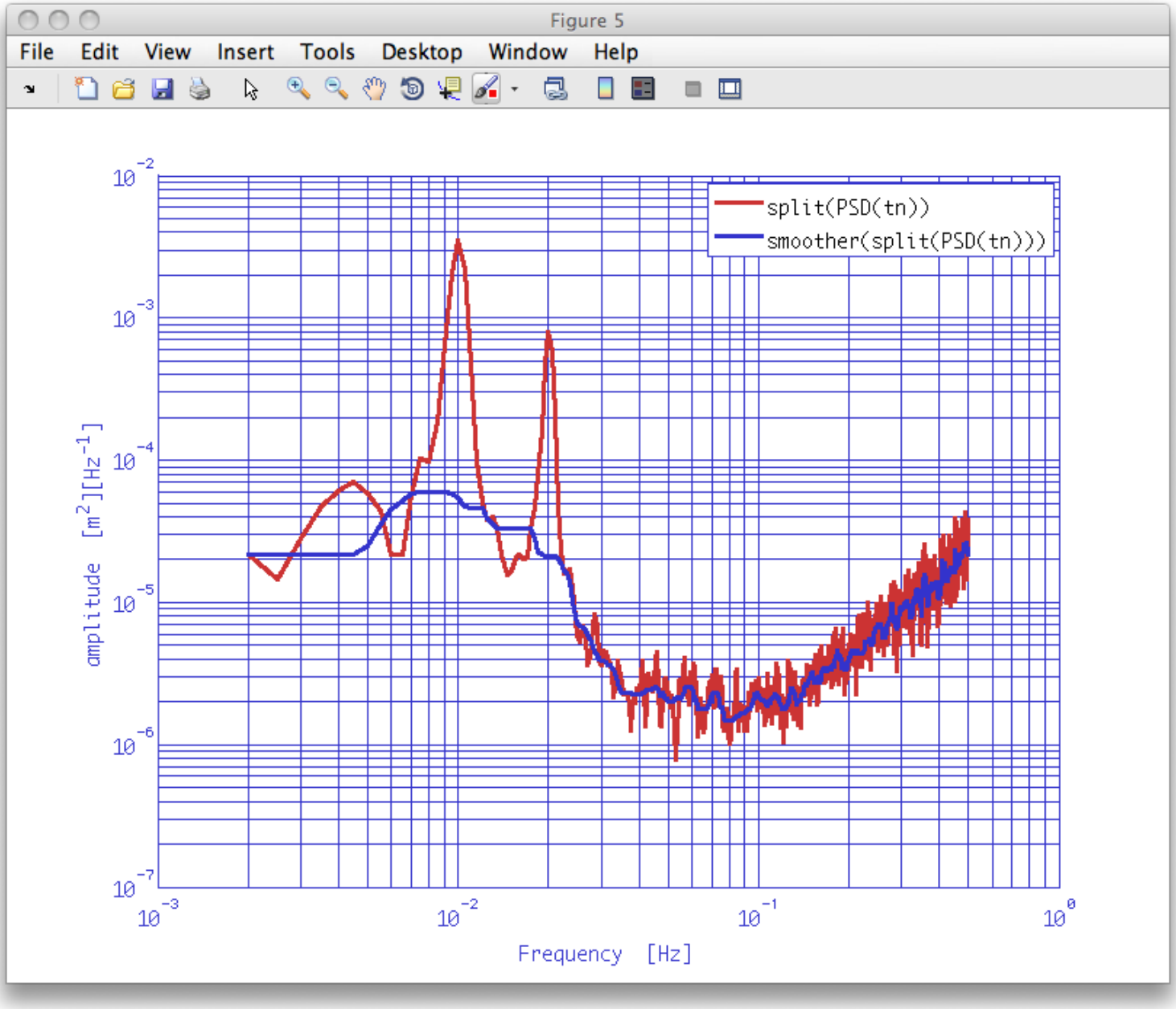


Now it is the moment to fit our PSD to extract a smooth model to pass to the noise generator. First of all we could define a set of proper weights for our fit process. We smooth our PSD data and then define the weights as the inverse of the absolute value of smoothed PSD. This should help the fit function 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);
iplot(tnxxr, stnxx)
wgh = 1./abs(stnxx);
```

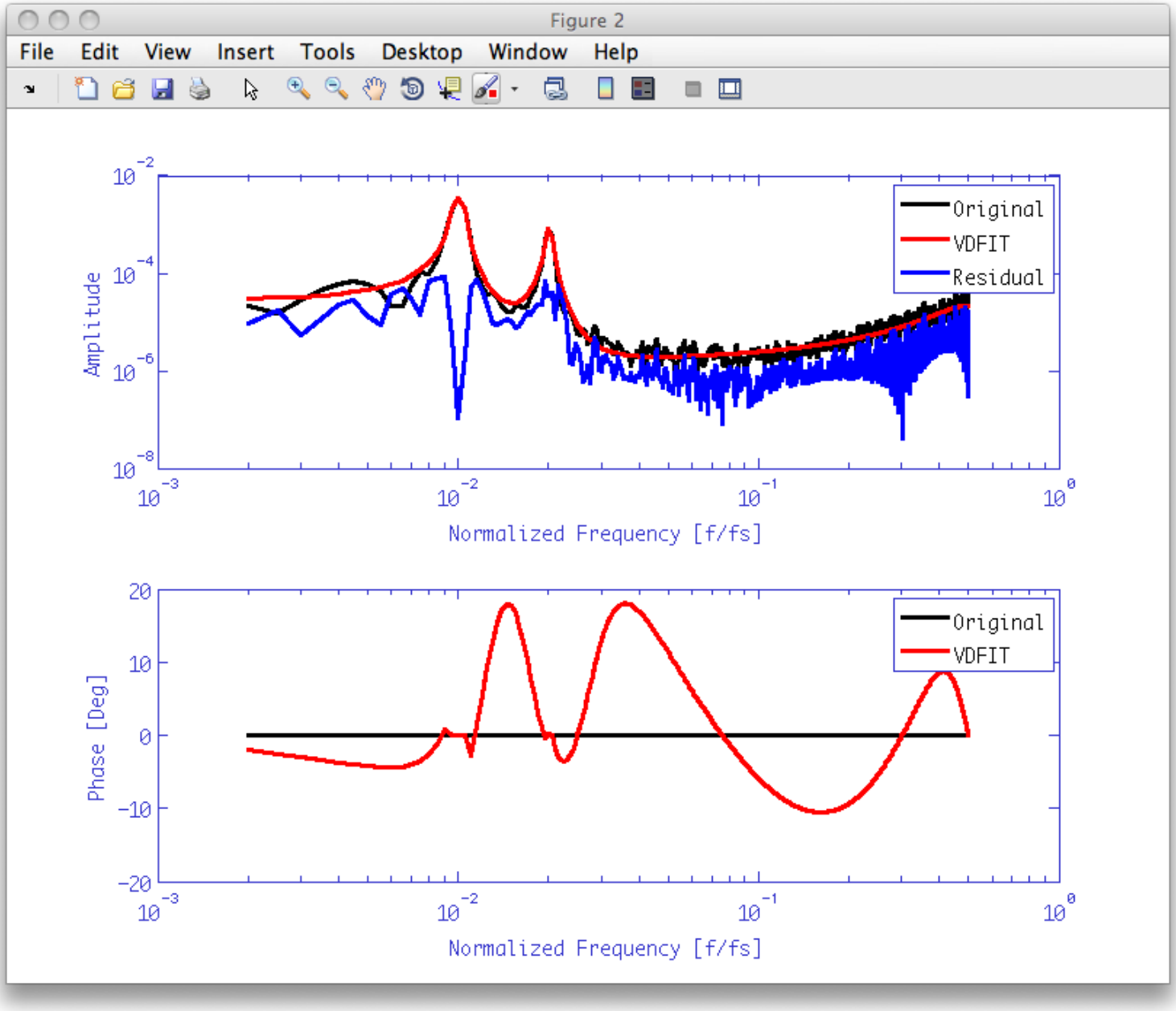
The result of the `smoother` method is shown in the plot below:

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','c1',...
              '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'));
```

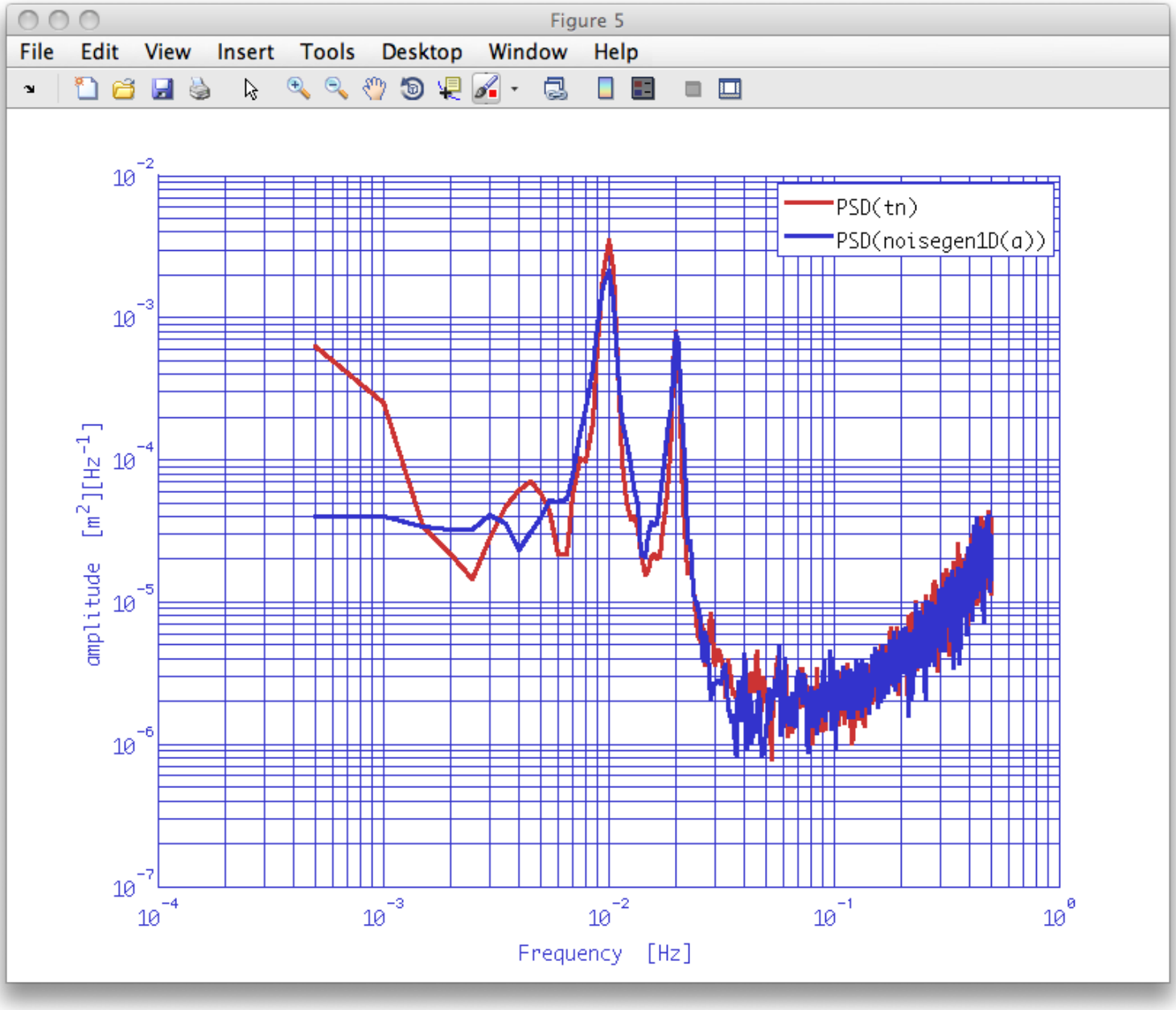
Then start noise coloring process calling `noisegen1D`

```
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 polynimials

Fitting time series with polynimials

Fitting time series with polynimials exploits the function `ao/polyfit`. Details on the agorithm can be found in the appropriate help page.

Fitting time series with polynimials

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.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);
```

The variable `p` is a `cdata` analysis object containing the coefficients of the fitted polynimial.

```
----------- ao 01: polyfit(a) -----------

name:  polyfit(a)
description:
data: -3.36766610680378e-014 1.00459257400442e-010 -9.19003973844817e-008 3.98578358994058e-005 -
0.0123010606401583 1.02090831777042 -87.6426134396757
-------- cdata 01 ------------
y:  [1x7], double
yunits:  []
------------------------------

hist:  ao / polyfit / $Id: polyfit.m,v 1.23 2009/02/19 16:57:47 anneke Exp
mfilename:
mdlfilename:
----------------------------------------
```

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.

```
iplot(a,b)
iplot(a-b)
```

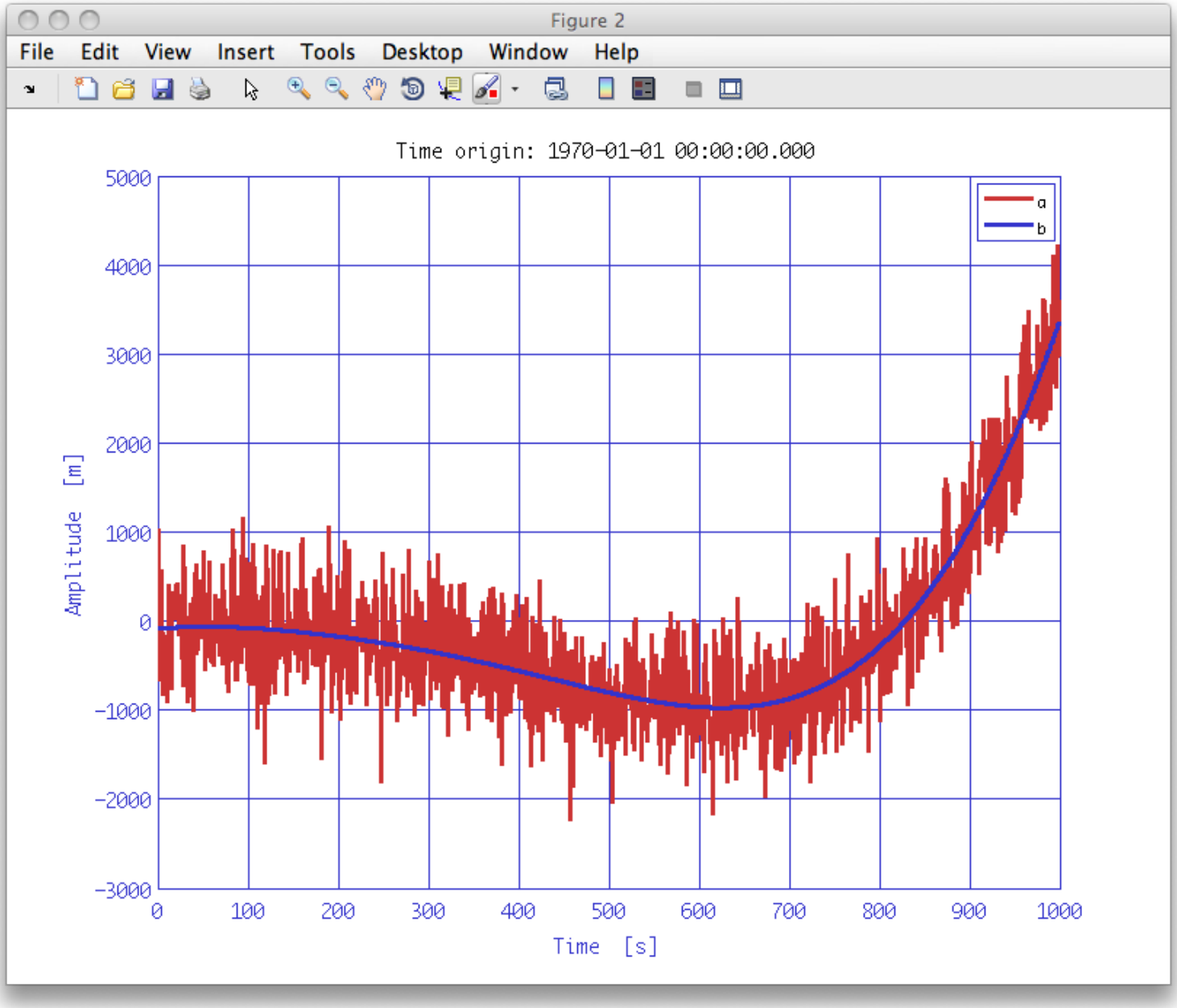
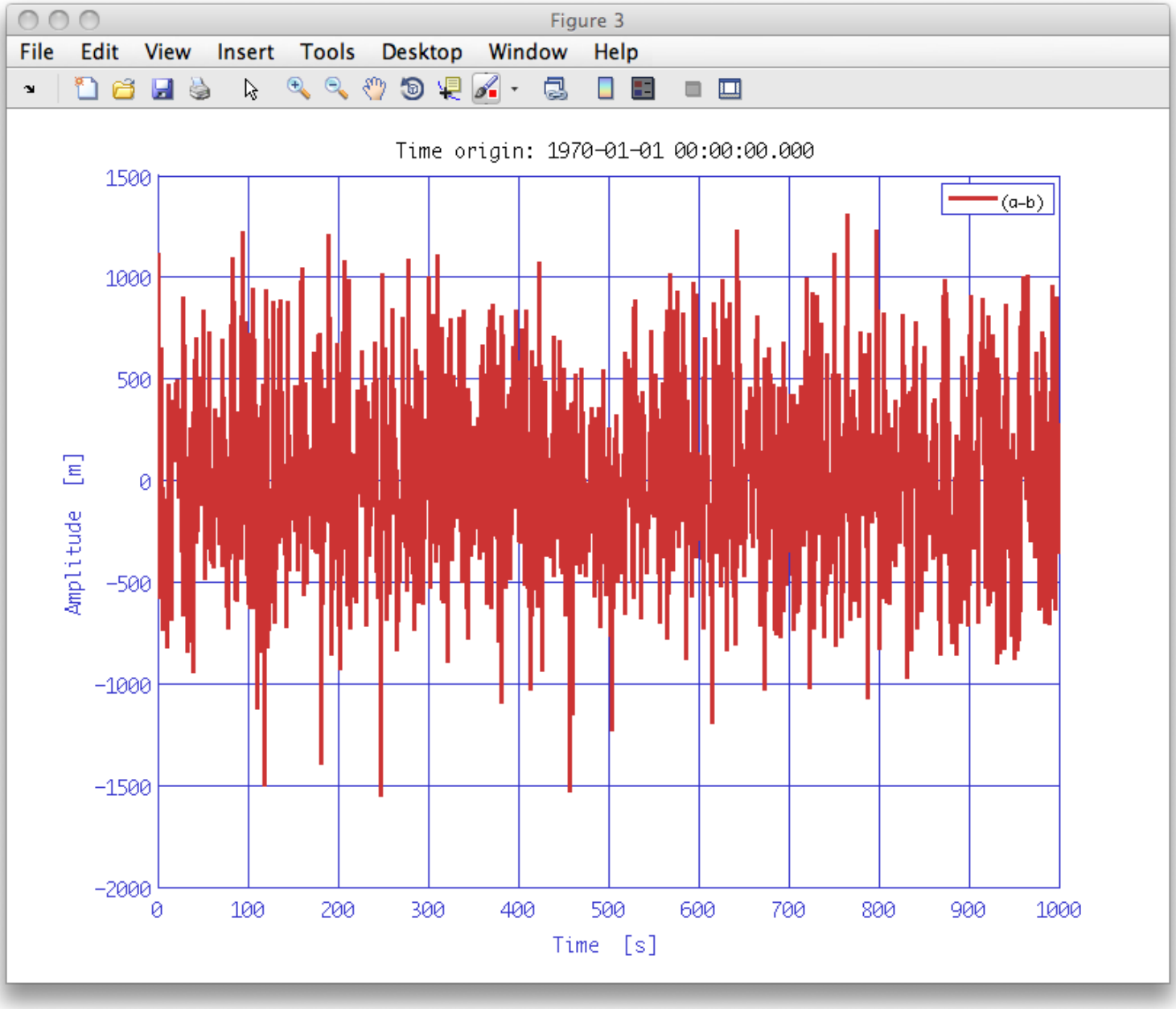


Figure 2
File Edit View Insert Tools Desktop Window Help
Time origin: 1970-01-01 00:00:00.000
a
b
Amplitude [m]
5000
4000
3000
2000
1000
0
-1000
-2000
-3000
0
100
200
300
400
500
600
700
800
900
1000
Time [s]



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.

Generation of noise with given psd

Non-linear least square fitting of time series

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

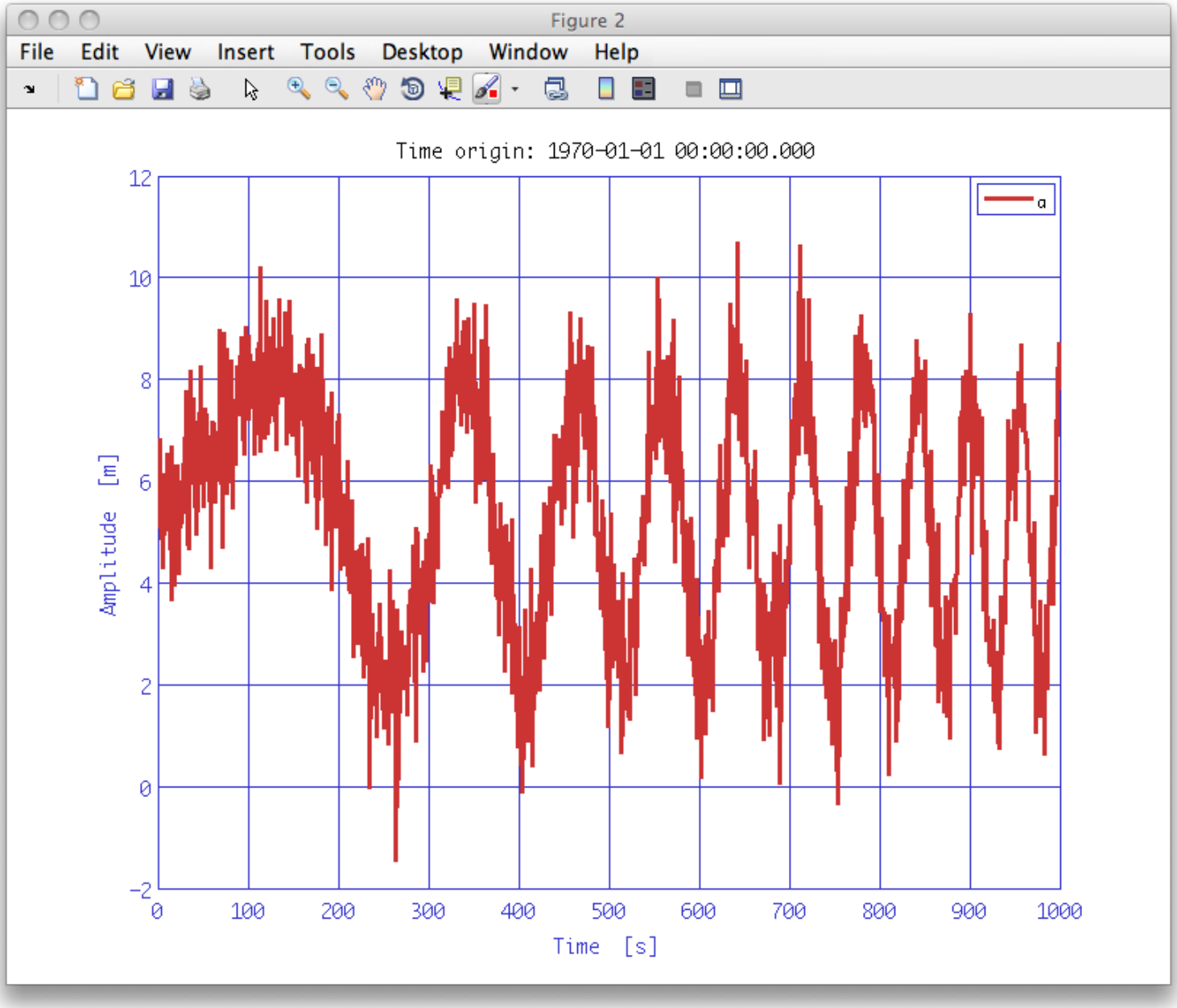
During this exercise we will:

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:

$$F(t) = A\sin\left[2\pi\left(f_0 + kt\right)t + \varphi\right]$$

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.

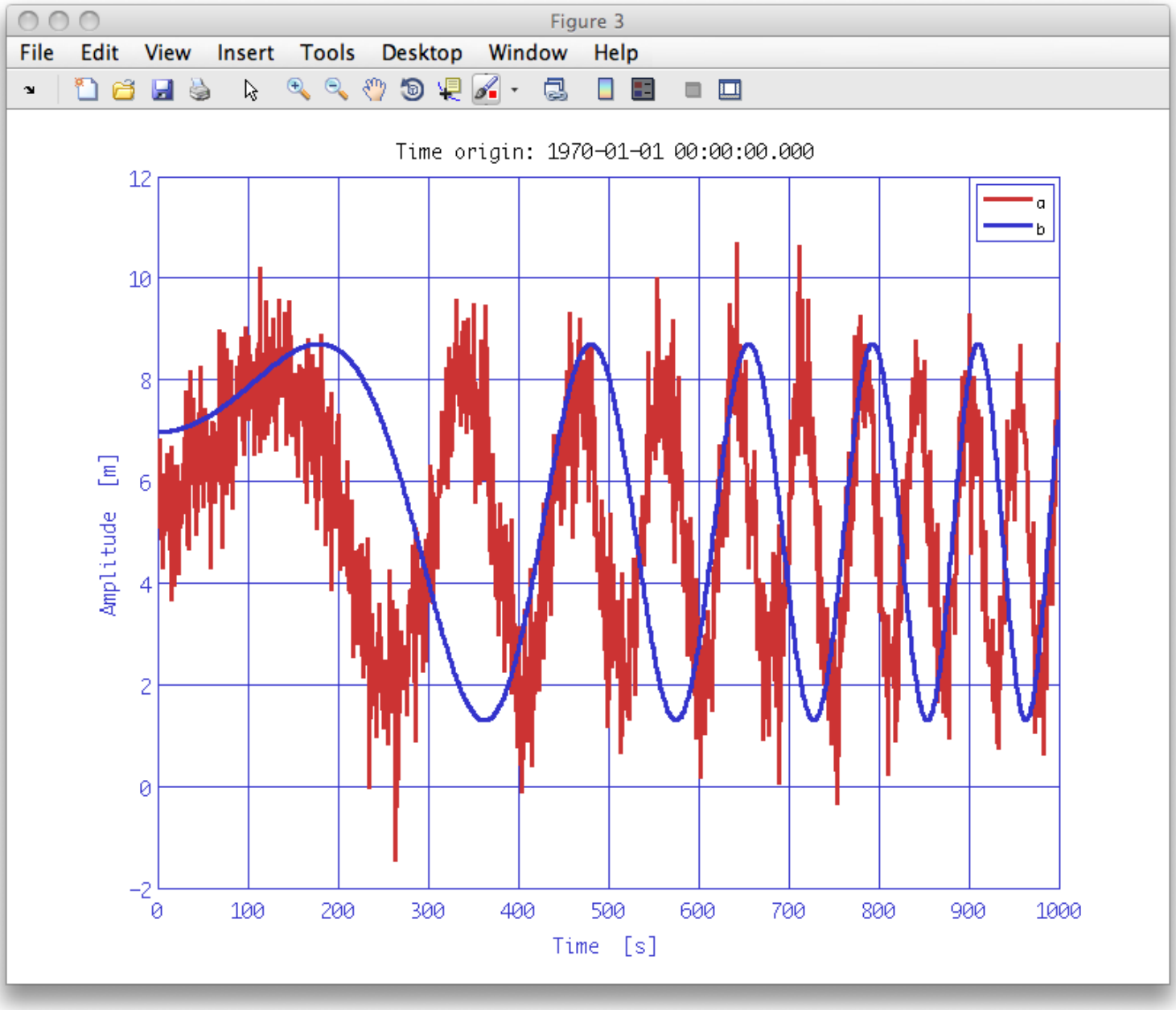
```
plfit = plist('Function', 'ADDP{1} + P(1).*sin(2.*pi.*(P(2) + P(3).*Xdata).*Xdata + P(4))', ...
              'P0', [4 3e-5 5e-6 0.5], ...
              'LB', [1 0 0 -pi], ...
              'UB', [5 1 1 pi],...
              'ADDP', {5});
params = curvefit(a, plfit);
```

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, ...
               'dtype', 'tsdata', ...
               'ADDP', {5});
b = evaluateModel(params, pleval);
```

```
b.setYunits(a.yunits);
b.setXunits(a.xunits);
b.setName;
iplot(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.

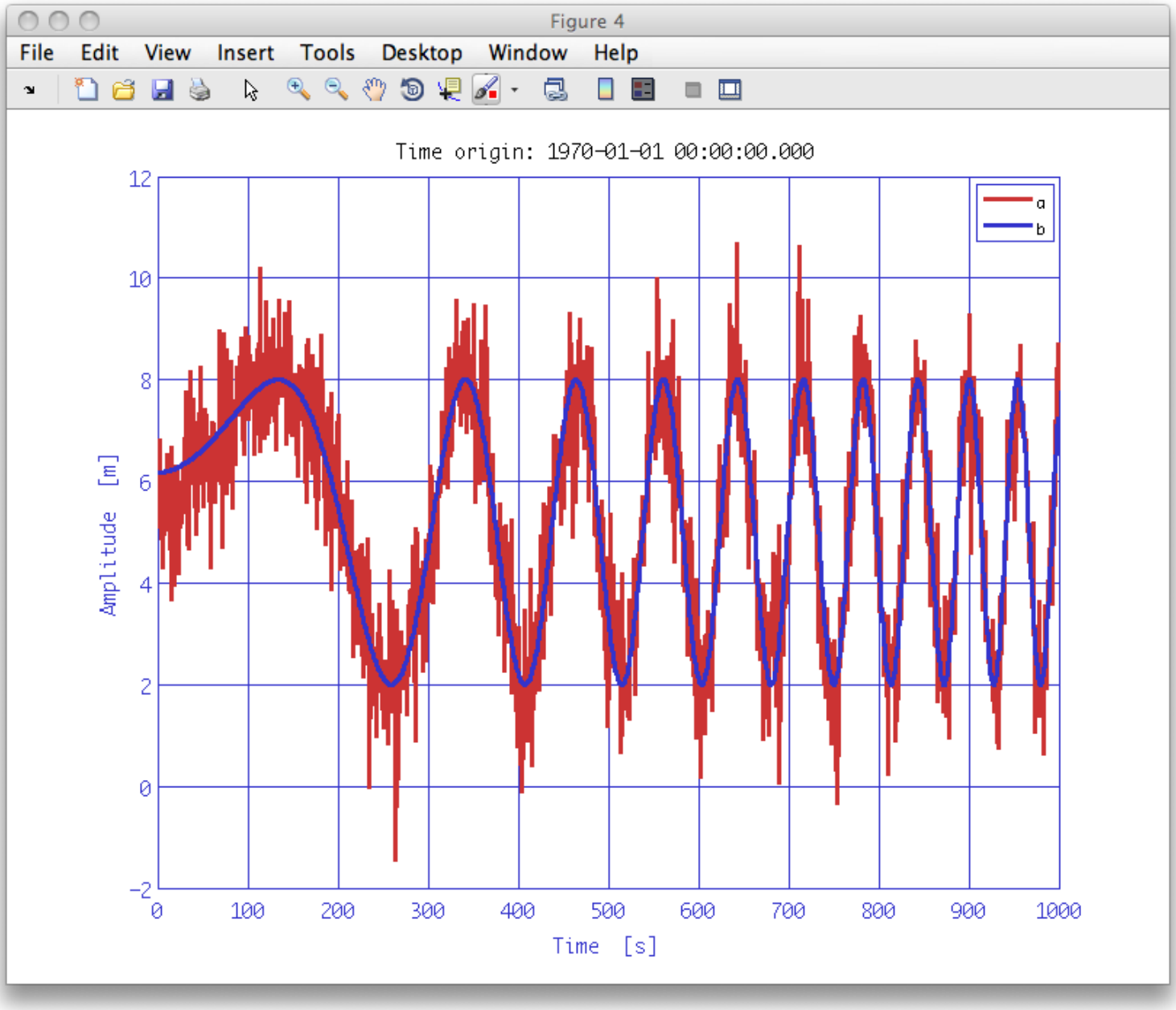


Looking at data we could try to fix some parameters like the amplitude and phase. The y starting level is near 0.4 + bias corresponding to a phase angle around 0.4 rad; the amplitude is close to 3. Now run the fit again with a reduced set of parameters.

```
% Do the fit again
plfit = plist('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 = plist('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);
```

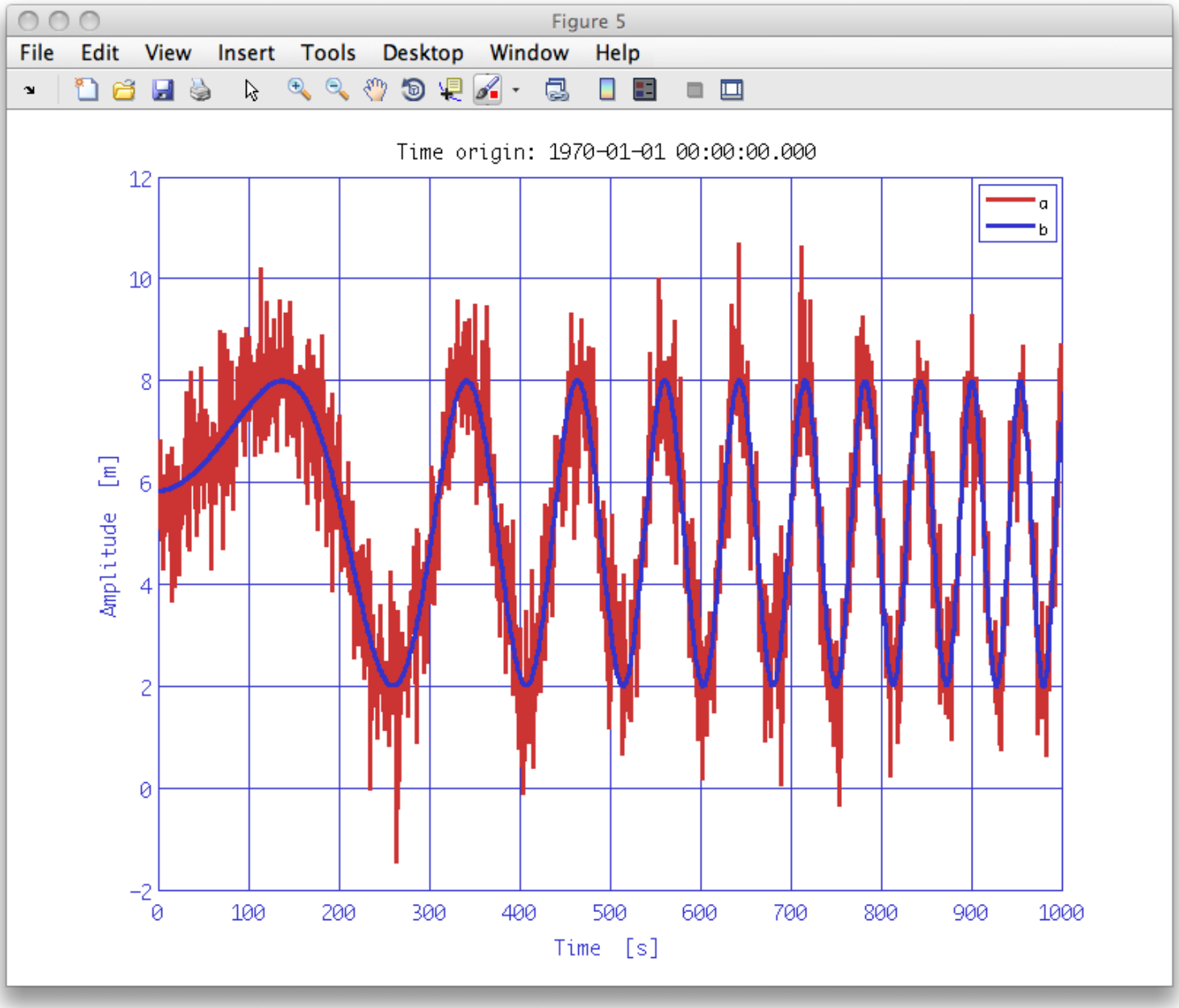
```
b.setName;

iplot(a,b)
```

The fit now looks like better...



Actual parameters contained in `params` are now a good starting guess for the wave frequancy 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;

iplot(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!

Fitting time series with polynimials IFO/Temperature Example - signal subtraction

IFO/Temperature Example - signal subtraction

During this exercise we will:

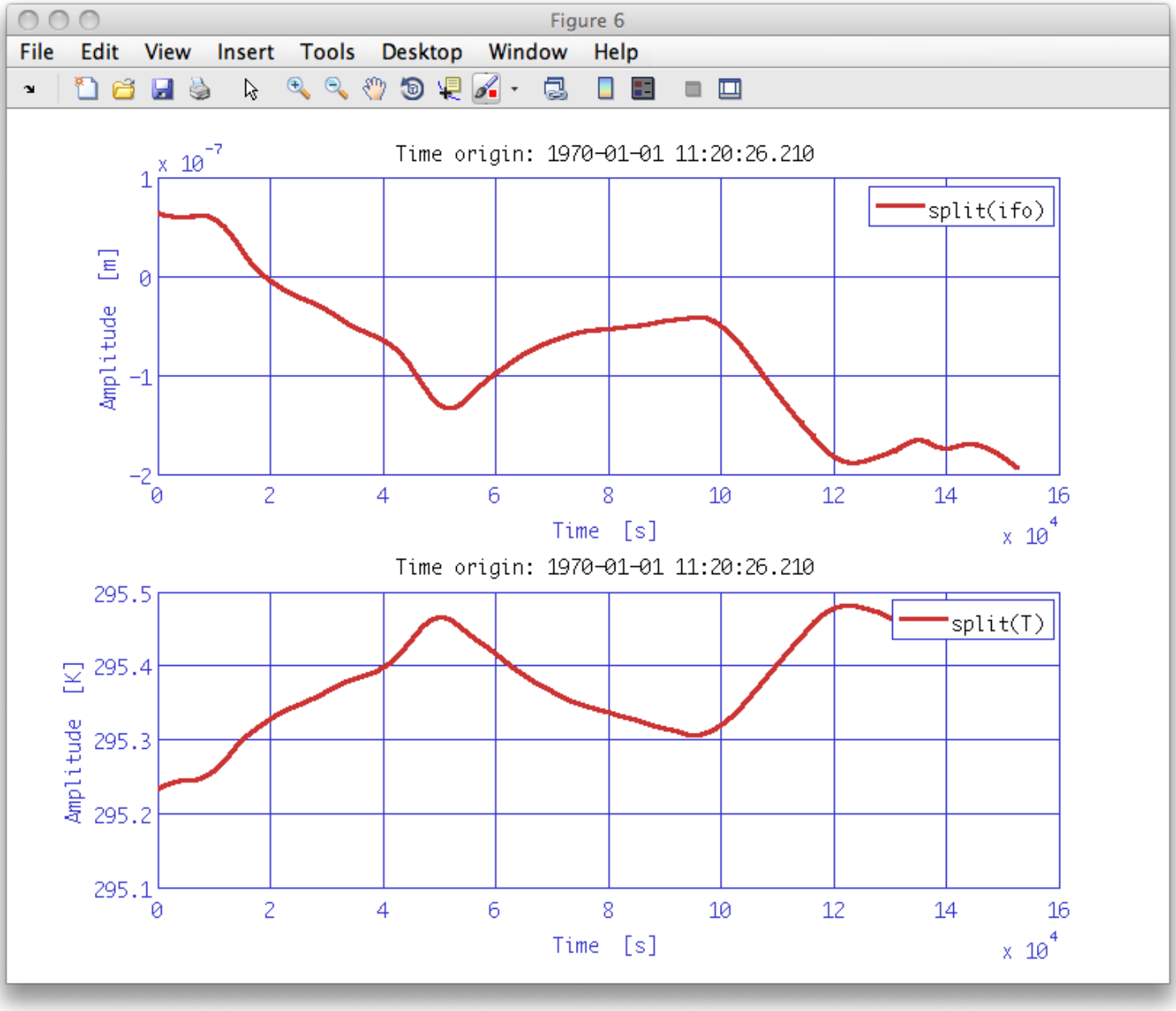
1. Load the AOs with the IFO and Temperature data
2. Load the transfer function of the data
3. Split the TF to select the meaningful region only
4. Fit the TF with `zDomainFit`
5. Subtract the temperature contribution from the IFO signal

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'))
```



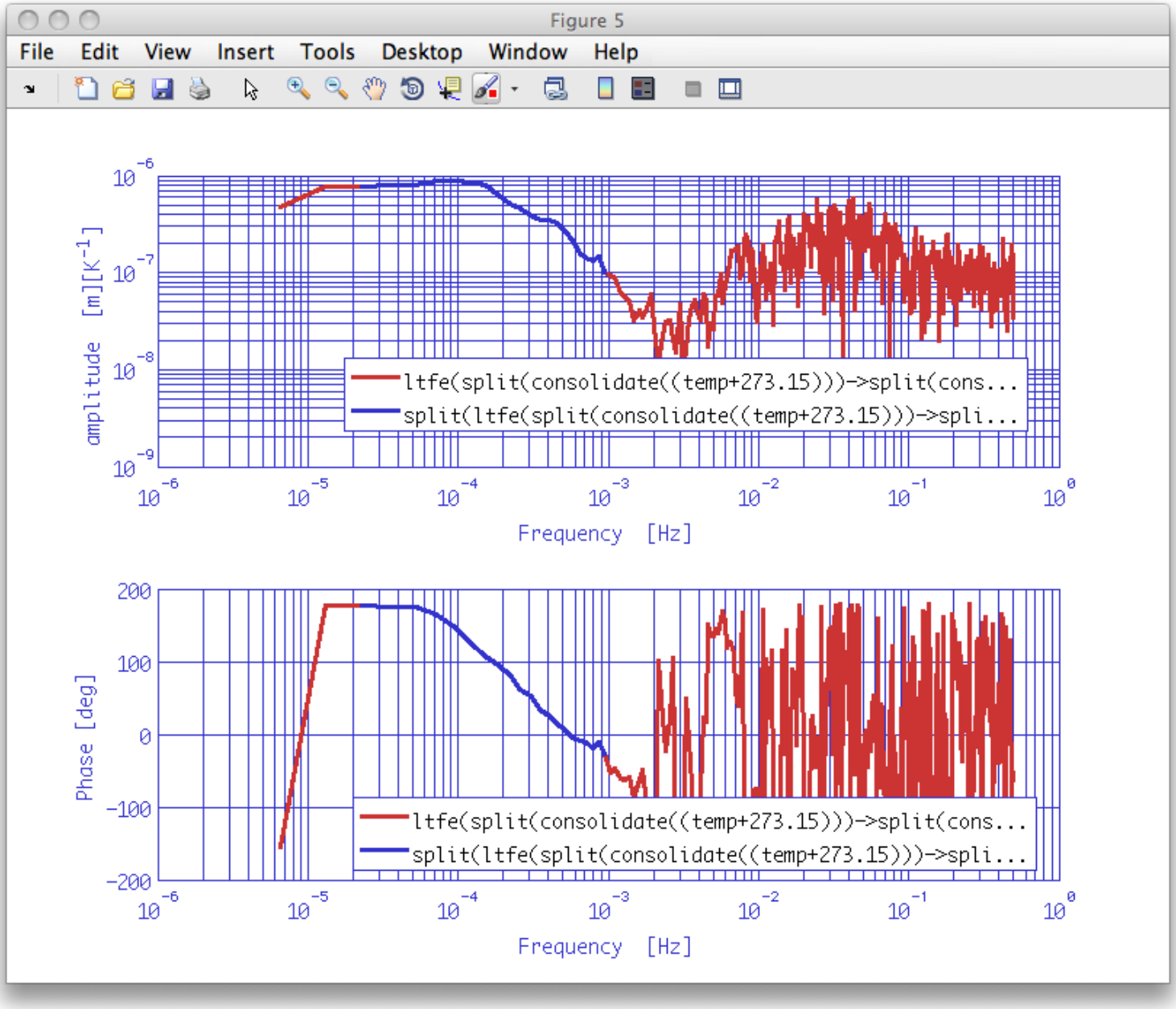
Let us load the transfer function estimate we made in Topic 3.

```
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.

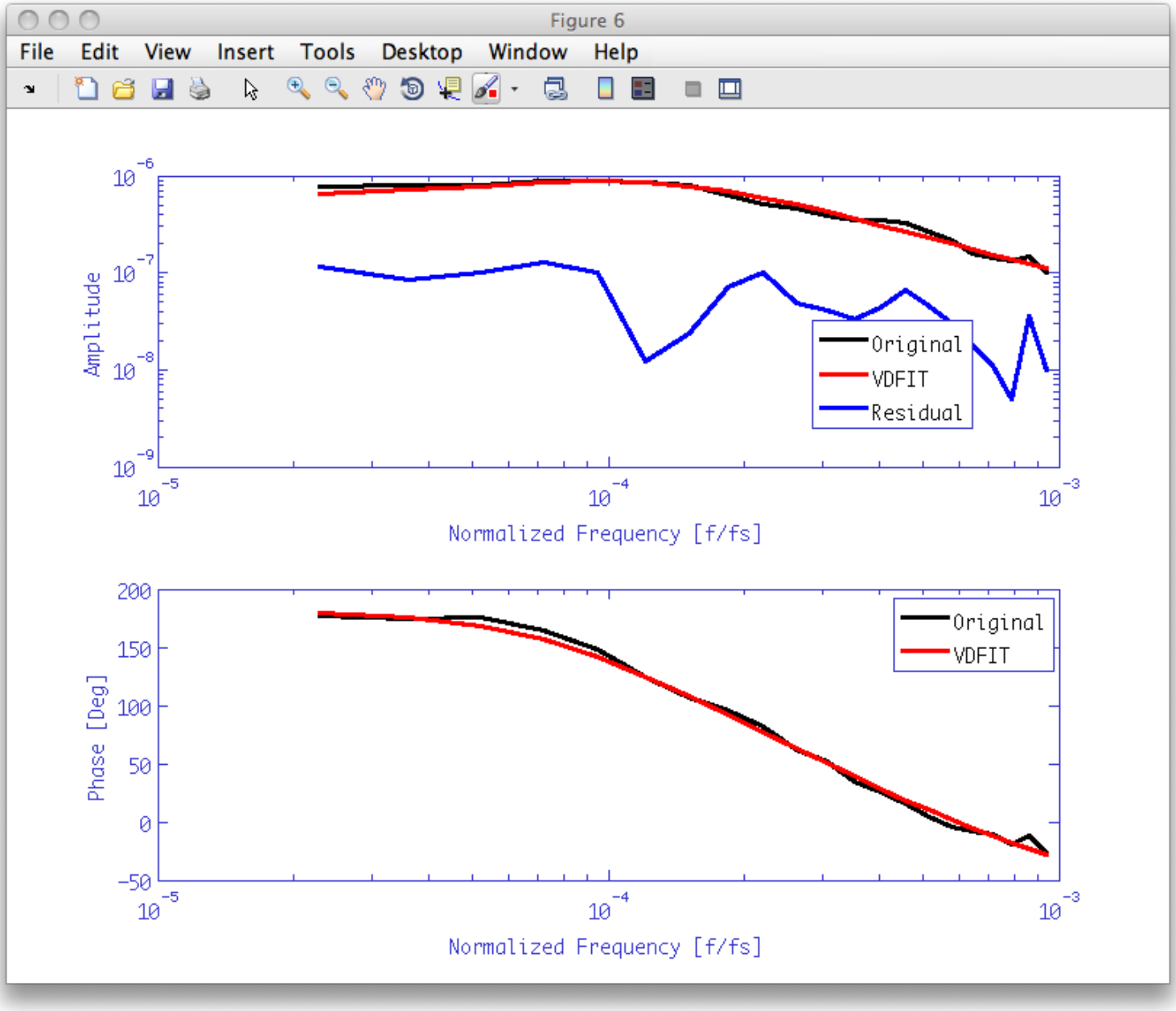
```
tfsp = split(tf,plist('frequencies', [2e-5 1e-3]));
iplot(tf,tfsp)
```

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 automathic 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','c1',...
    'maxiter',20,...
    'minorder',3,...
    'maxorder',3,...
    'weightparam','abs',...
    'Plot','on',...
    'ForceStability','on',...
    'CheckProgress','off');

fobj = zDomainFit(tfsp,plfit);
fobj.setIunits('K');
fobj.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).

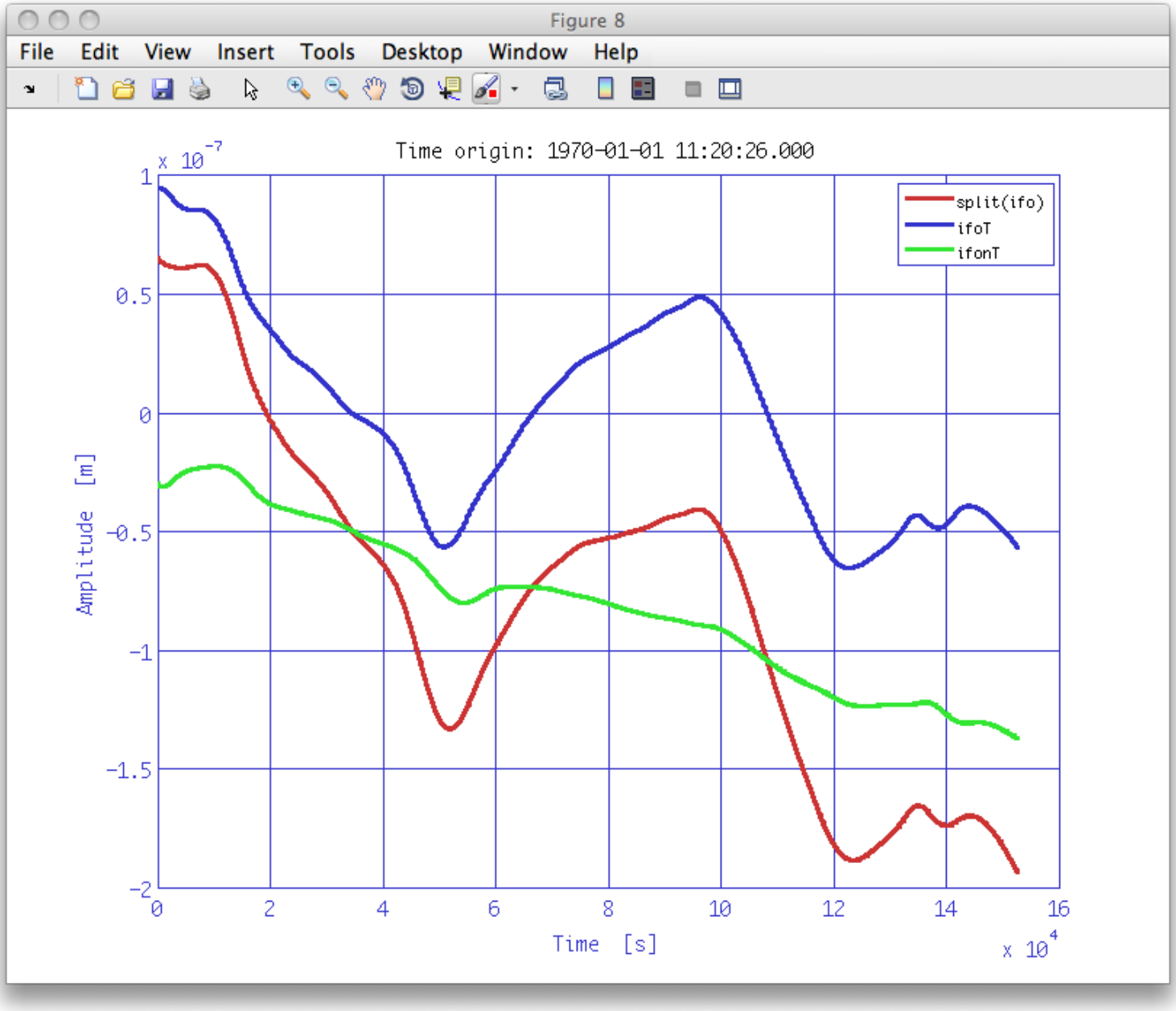
```
ifoT = filter(T_red,fobj,plist('bank','parallel'));
ifoT.detrend(plist('order',0));
ifoT.simplifyYunits;
ifoT.setName;
```

Then we subtract temperature contribution from measured interferometer data

```
ifonT = ifo_red - ifoT;
ifonT.setName;
```

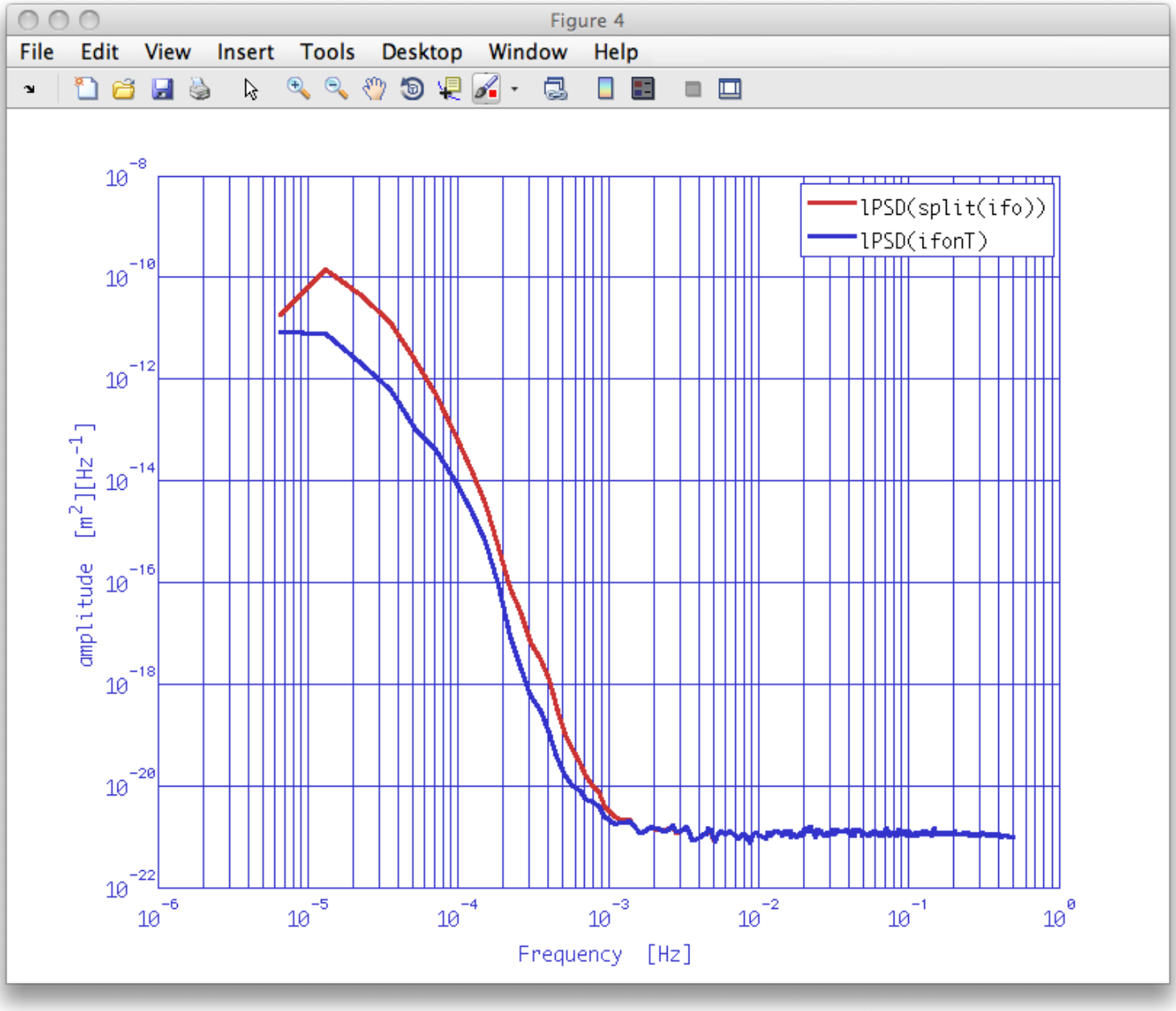
The figure reports measured interferometer data, temperature contribution to interferometer output and interferometer output without thermal drifts.

```
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

Starting the LTPDA Toolbox

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

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

Additional 3rd-party software Trouble-shooting

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Training Session 1
Documents
Bugs and features
Troubleshooting

Training session one takes place at the AEI in Hannover on the 10th and 11th of March, 2009. More details can be found at the home page.

Tutorials can be read on-line here

NOTE: The tutorial topics are designed to run with version 2.0 of the toolbox. Later versions may require some changes.

DataPack.zip
The data needed for the training session tutorials.

LTPDA_training_S1_v3.pdf
The planned agenda.

ltpda_intro.pdf
Introduction slides

ltpda_topic_1.pdf
Topic 1 slides

ltpda_topic2.pdf
Topic 2 slides

ltpda_topic3.pdf
Topic 3 slides

ltpda_topic4.pdf
Topic 4 slides

ltpda_topic5.pdf
Topic 5 slides

home >> Training Sessions >> Training Session 1 >>

LTPDA Training Session 1

Date: 10th - 11th March 2009
Location: Hannover, Seminar room 103
web-site: http://www.aei.mpg.de/~minofr/Home.html

Introduction

This is the first in a planned series of training sessions on the use of LTPDA. The training session is organised as a set of topics which aim to introduce the participant to the use of LTPDA for standard signal processing. Each topic features an introductory presentation (~15 minutes) followed by a working session in which the participant can work through the associated examples and tasks. The intention is to have LTPDA experts on-hand to help with the worked examples. For this first training session, the aim is to have a mixture of examples and tasks to be carried out using both script and graphical programming.

Requirements

In order to participate fully in the training session, it is advisable to come equipped with a laptop with MATLAB installed. The LTPDA toolbox has the following requirements

MATLAB version: 7.7 (R2008b)
Signal Processing Toolbox version: 6.10 (R2008b)
Symbolic Math Toolbox: version: 5.1 (R2008b) *****
Optimization Toolbox version: 4.1 (R2008b) *****
Database Toolbox version: 3.5 (R2008b) *****
Control System Toolbox version: 8.2 (R2008b) *****

The (*****) items are for advanced features, but many, if not all, of the activities will be possible without those toolboxes.

In addition, for those without their own laptop, we plan to have a small number of machines configured with LTPDA (~10). It may be that people will have to work in groups, depending on the number of attendees.

Agenda

Day 1

11:00 Introduction to and installation of LTPDA [Martin]
11:30 Topic 1 [Martin]
13:00 Lunch break
14:00 Topic 1 continued
15:00 Coffee break
15:15 Topic 2 [Anneke]
17:00 Question and answer session
19:00 Dinner

Day 2

09:00 Topic 3 [Mauro]
10:45 Coffee break
11:00 Topic 4 [Miquel]
12:30 Lunch break
13:30 Topic 4 cont
15:00 Coffee break
15:30 Topic 5 [Luigi]
17:00 Q/A session
18:00 Close

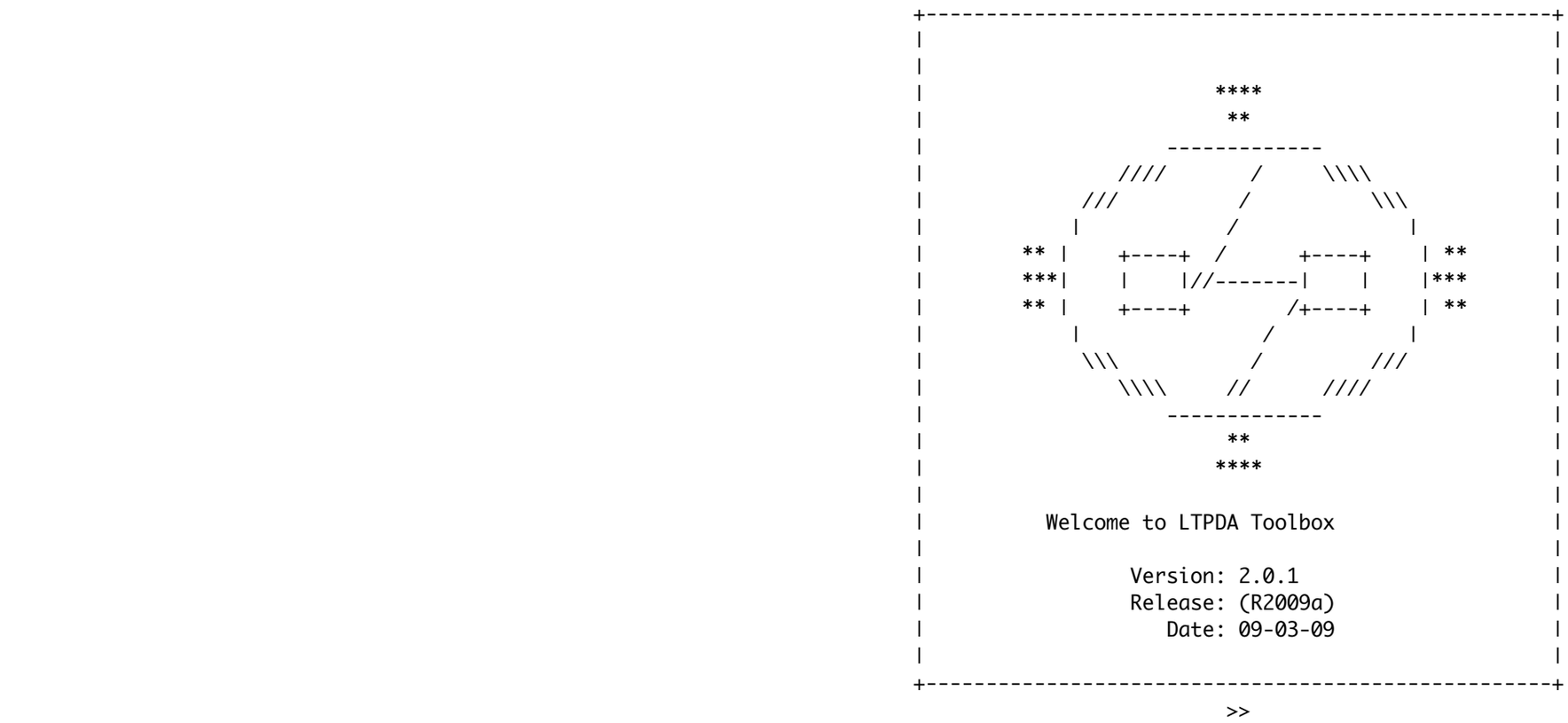
Topics

#	Topic	Description
1	Creating AOs and setting their properties. Viewing the data and the history. Basic ao manipulation.	This topic aims to introduce the various ways of creating Analysis Objects. In particular, the participant will learn how to create: 1. AOs containing simulated data 2. AOs from existing data files
2	Pre-processing of data	Various common signal processing tasks are addressed here: 1. Resampling time-series data 2. Interpolation 3. De-trending 4. Whitening 5. Data selection (split, select, find)

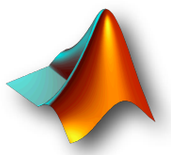
#	Topic	Description
3	Spectral Analysis	LTPDA contains various spectral analysis algorithms for computing: 1. Power Spectral Density estimates 2. Cross-power spectra 3. Coherence estimates 4. Transfer function estimates
4	Transfer function models and digital filtering	Create transfer function models in various forms: 1) As pole/zero models 2) As partial fraction expansions 3) As rational polynomials in s Build loop models, plot responses. Building digital filters from the models. Building standard digital filters. Filtering data.
5	Model fitting	Fit models to data: 1) Polynomial fitting 2) linear least-squares fitting 3) Non-linear least-squares fitting 4) Vector fitting (fitting pole/zero models)

The First LTPDA Training Session

M Hewitson



Welcome



Formalities

- WiFi connection:
  - SSID: UHEvent
  - WPA key: dgk2009nalw

- Websites
  - LTPDA:
    - http://www.lisa.aei-hannover.de/ltpda/
  - Training session:
    - http://www.aei.mpg.de/~minofr/Home.html

- Lunch (both days)
  - in Atrium downstairs
- Dinner tonight
  - Himalaya - see handout

Aims

- Introduce concepts of LTPDA
- How to do ‘standard’ lab-experiment data analysis

Bugs and mails

- e-mail to martin.hewitson@aei.mpg.de
  - soon to have a public bug tracking system

- Mailing lists: please subscribe to:

  - http://lists.aei.mpg.de/cgi-bin/mailman/listinfo/ltpda_releases

  - http://lists.aei.mpg.de/cgi-bin/mailman/listinfo/ltpda_users

The development team (your helpers)

1. Mauro Hueller
2. Nicola Tateo
3. Luigi Ferraioli
4. Martin Hewitson
5. Ingo Diepholz
6. Anneke Monsky
7. Miquel Nofrarias
8. Gerrit Visscher
9. Adrien Grynagier
10. Josep Sanjuan
11. Adolfo Villafiorita
12. Andrea Mattioli
13. Andrea Manica

UTN

AEI

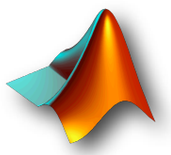
Stuttgart

IEEC

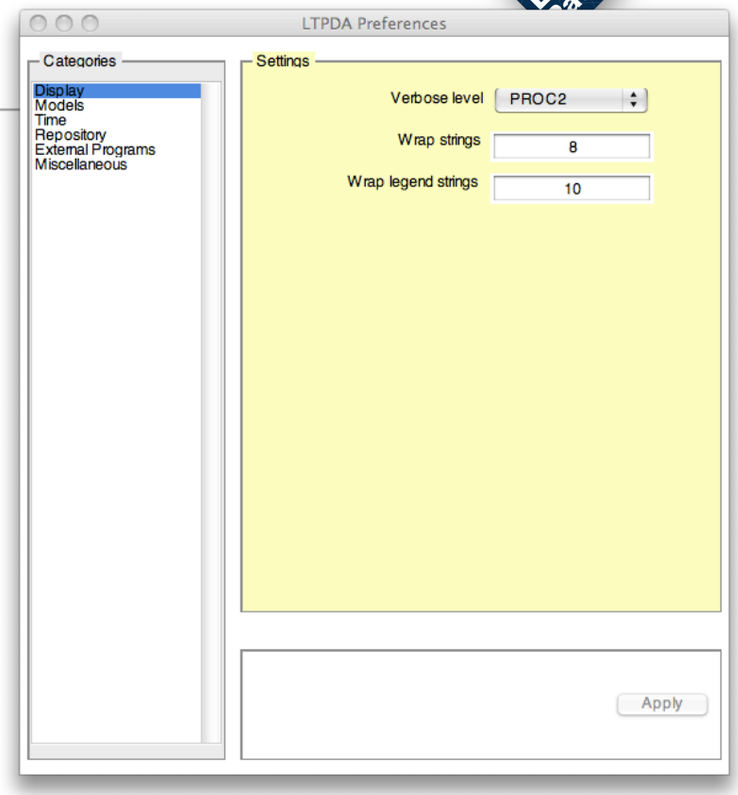
FBK

Thanks

- IT Staff
- Secretariat
- LOC



Installation - let the fun begin

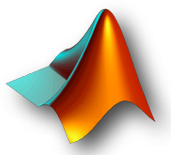


- Download latest version from
  - http://www.lisa.aei-hannover.de/ltpda/
- Unzip to somewhere
- Start MATLAB
- File->Set Path...
- Click ‘Add with Subfolders...’
- Navigate to the `ltpda_toolbox` folder you unzipped
- On MATLAB terminal
  - `>> ltpda_startup`
    - (you can add this command to you normal `startup.m` file)
- Set preferences
  - for now just click ‘Apply’ and close the GUI

Additional steps...

- Test installation
  - >> run_tests

- Install graphviz
  - see LTPDA user manual (>> doc)
    - LTPDA Toolbox
      - Getting Started with the LTPDA Toolbox
        - Additional 3rd-party software



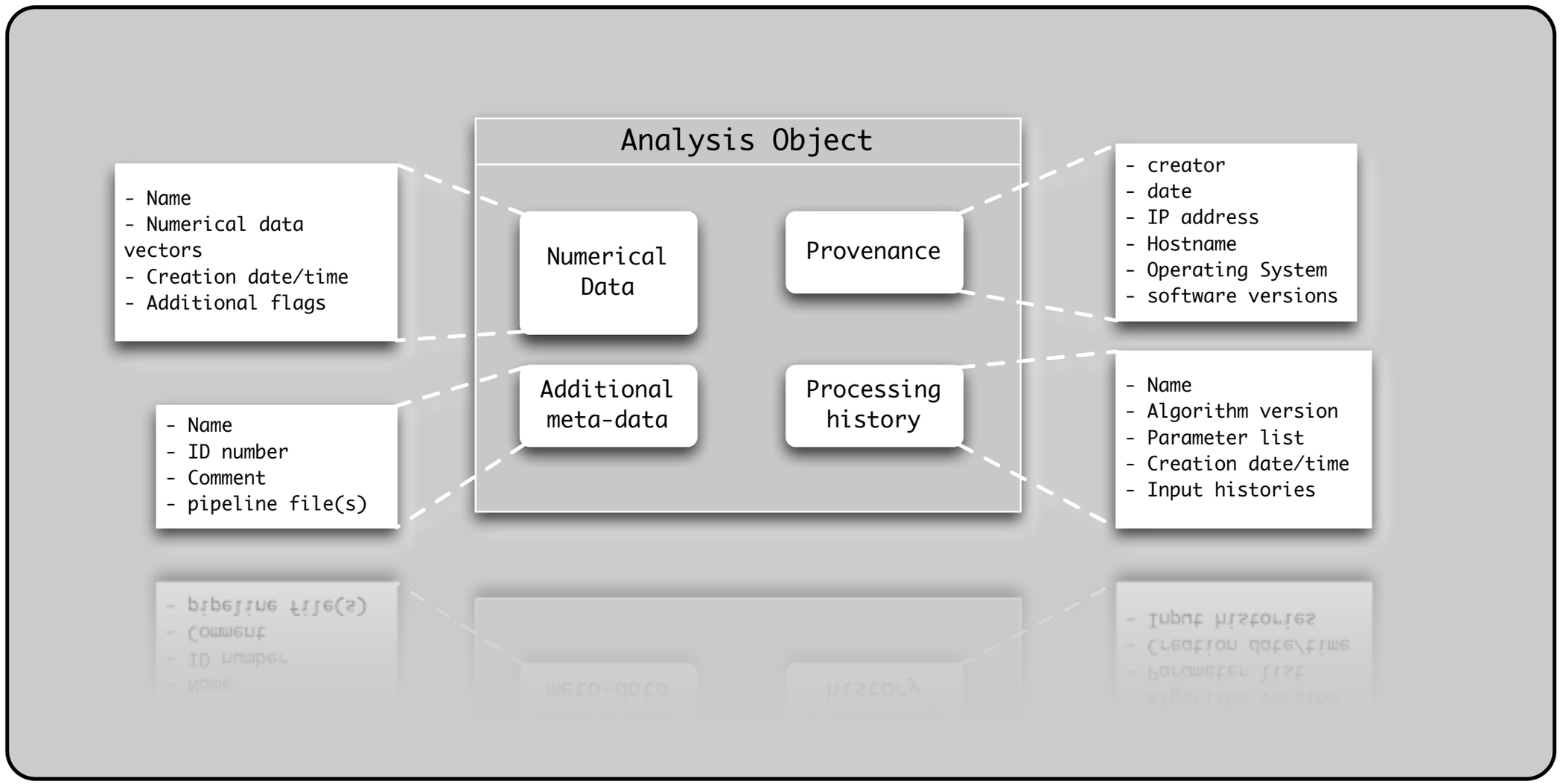
The First LTPDA Training Session - Topic 1

M Hewitson

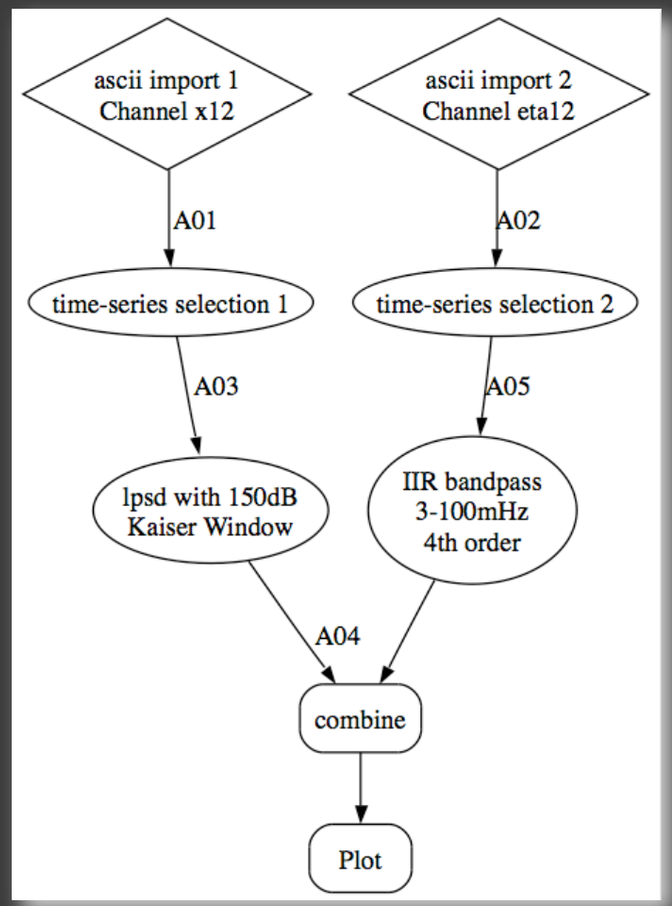
```
+-------------------------------------------------+
|                                                 |
|                                                 |
|                     ****                        |
|                      **                         |
|                  -------------                  |
|               ////       /     \\\\              |
|             ///          /         \\\           |
|           |             /             |         |
|       ** |    +----+  /      +----+    | **     |
|       ***|    |    |//-------|    |    |***     |
|       ** |    +----+        /+----+    | **     |
|           |               /             |       |
|            \\\           /         ///          |
|               \\\\     //      ////              |
|                   -------------                 |
|                       **                        |
|                      ****                       |
|                                                 |
|          Welcome to LTPDA Toolbox               |
|                                                 |
|                 Version: 1.9.3                  |
|                 Release: (R2008b)               |
|                    Date: 19-10-08               |
|                                                 |
+-------------------------------------------------+
```

Introducing Analysis Objects (AOs)

- An AO contains more than just data

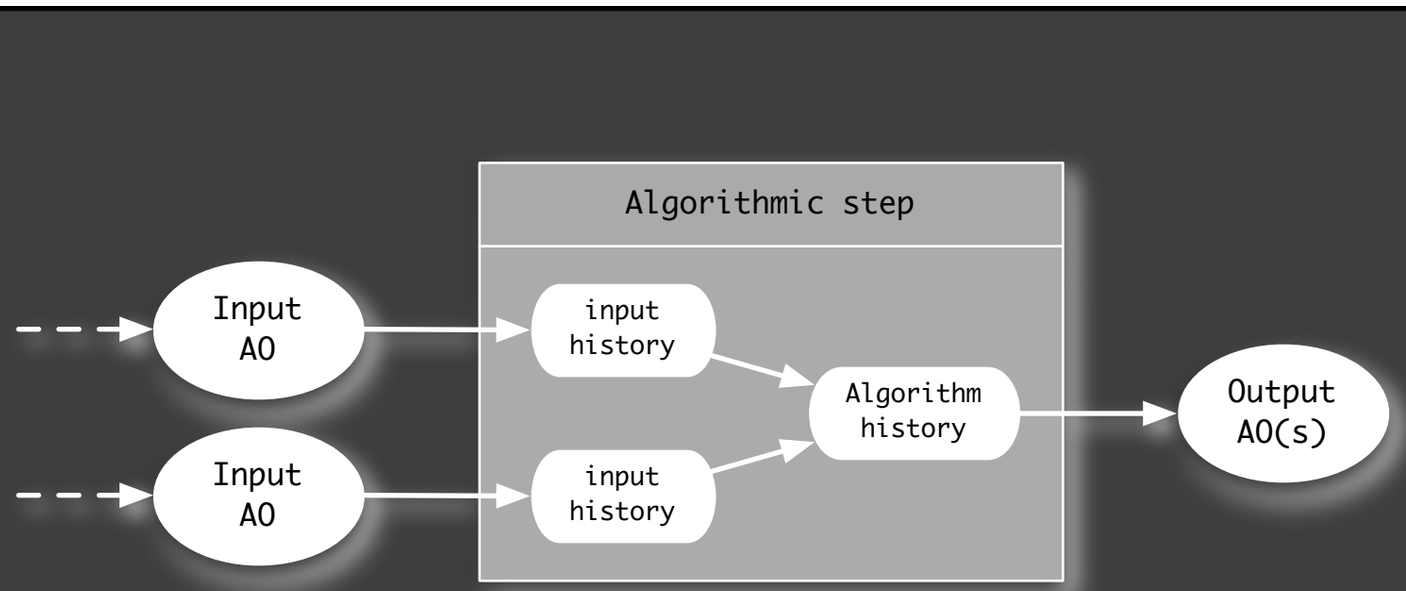


History



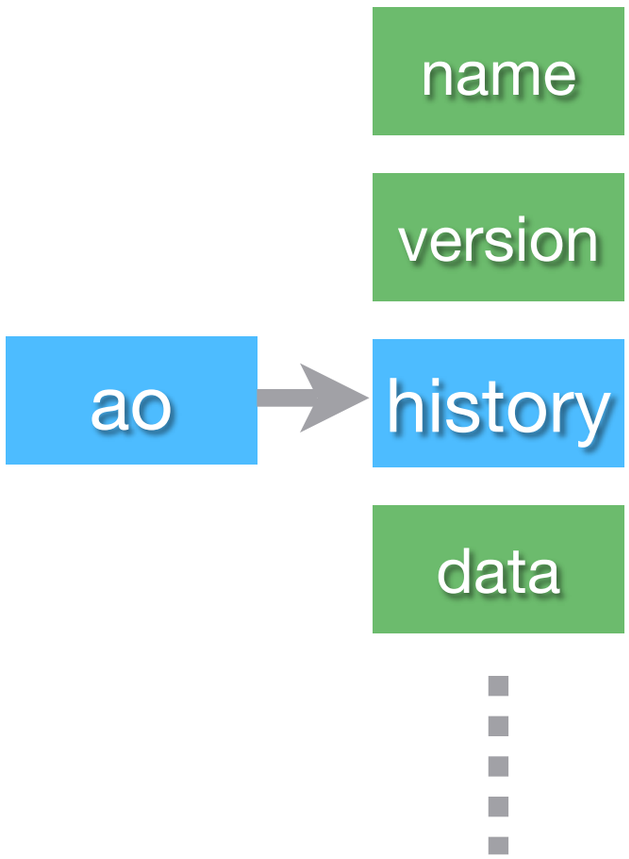
traceable results

Intelligent algorithms

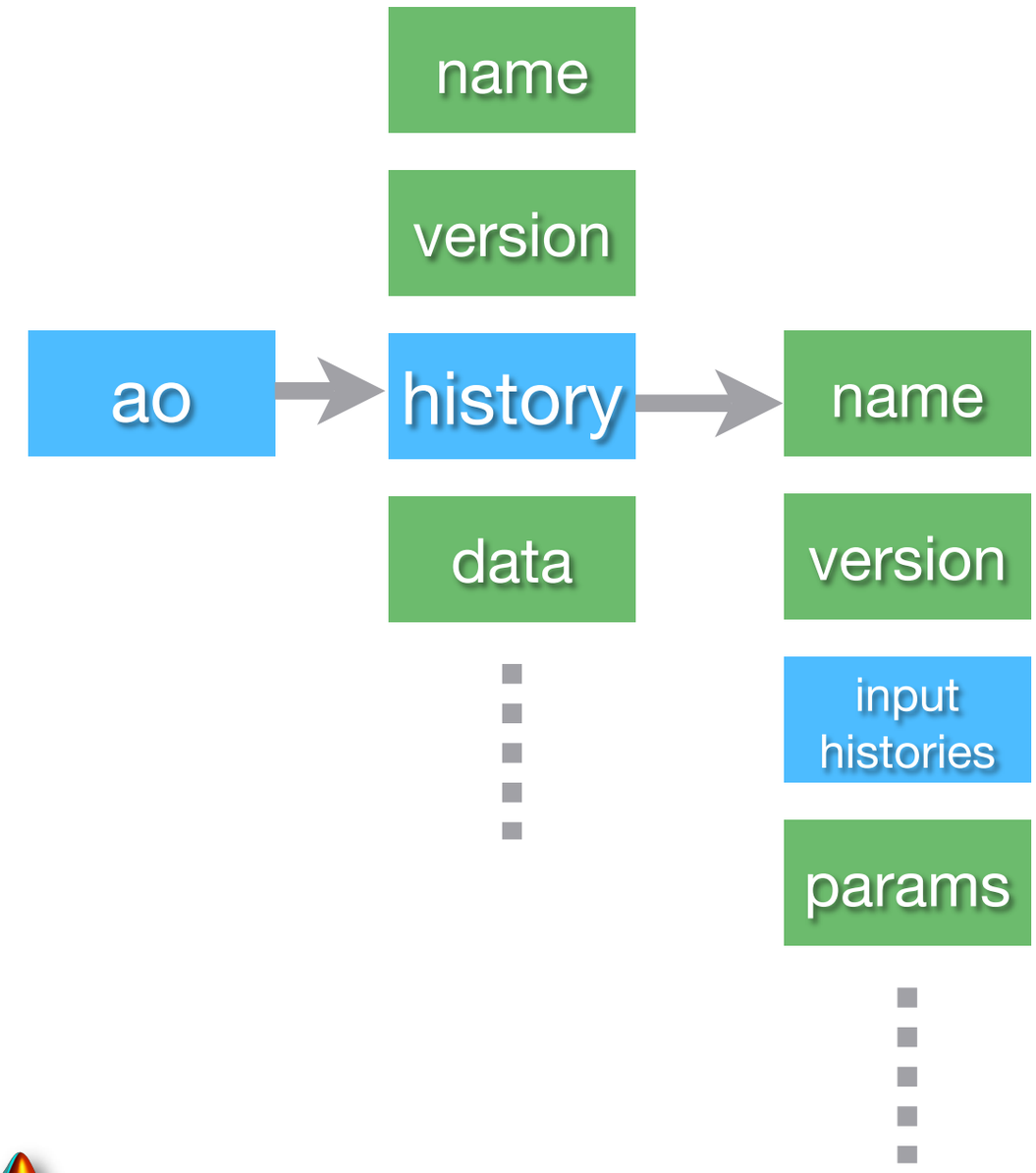


History tree

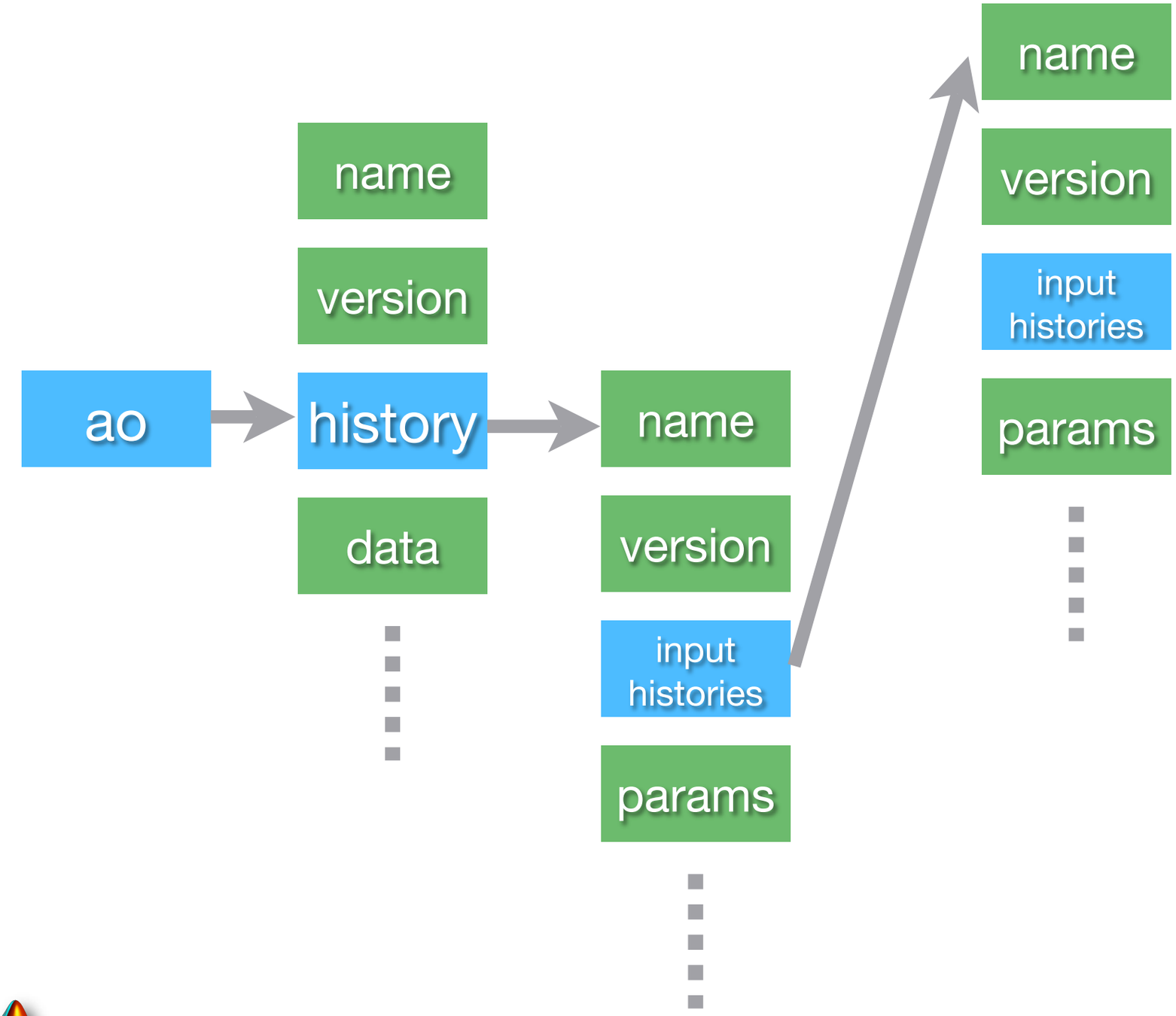
History tree



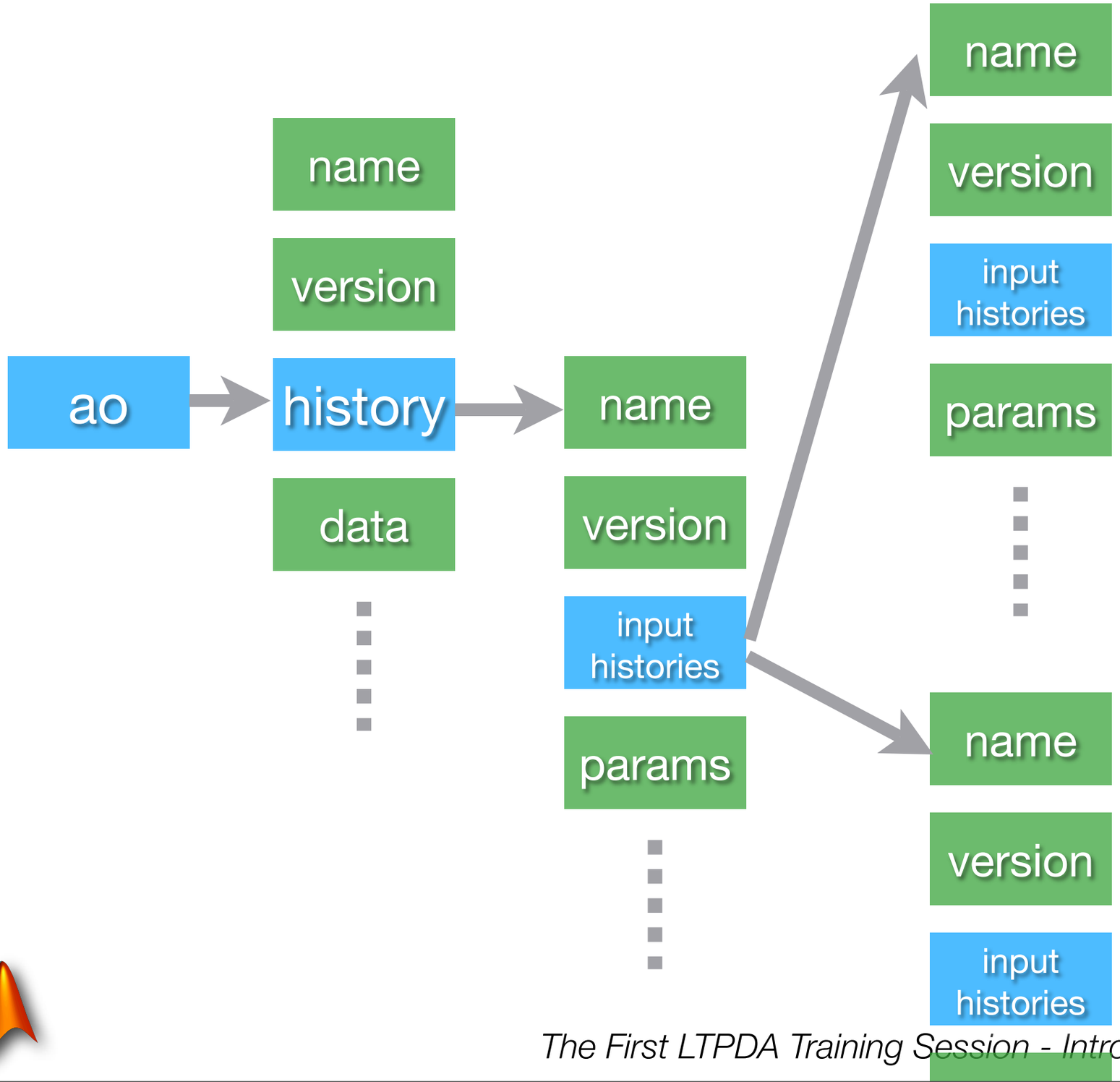
History tree



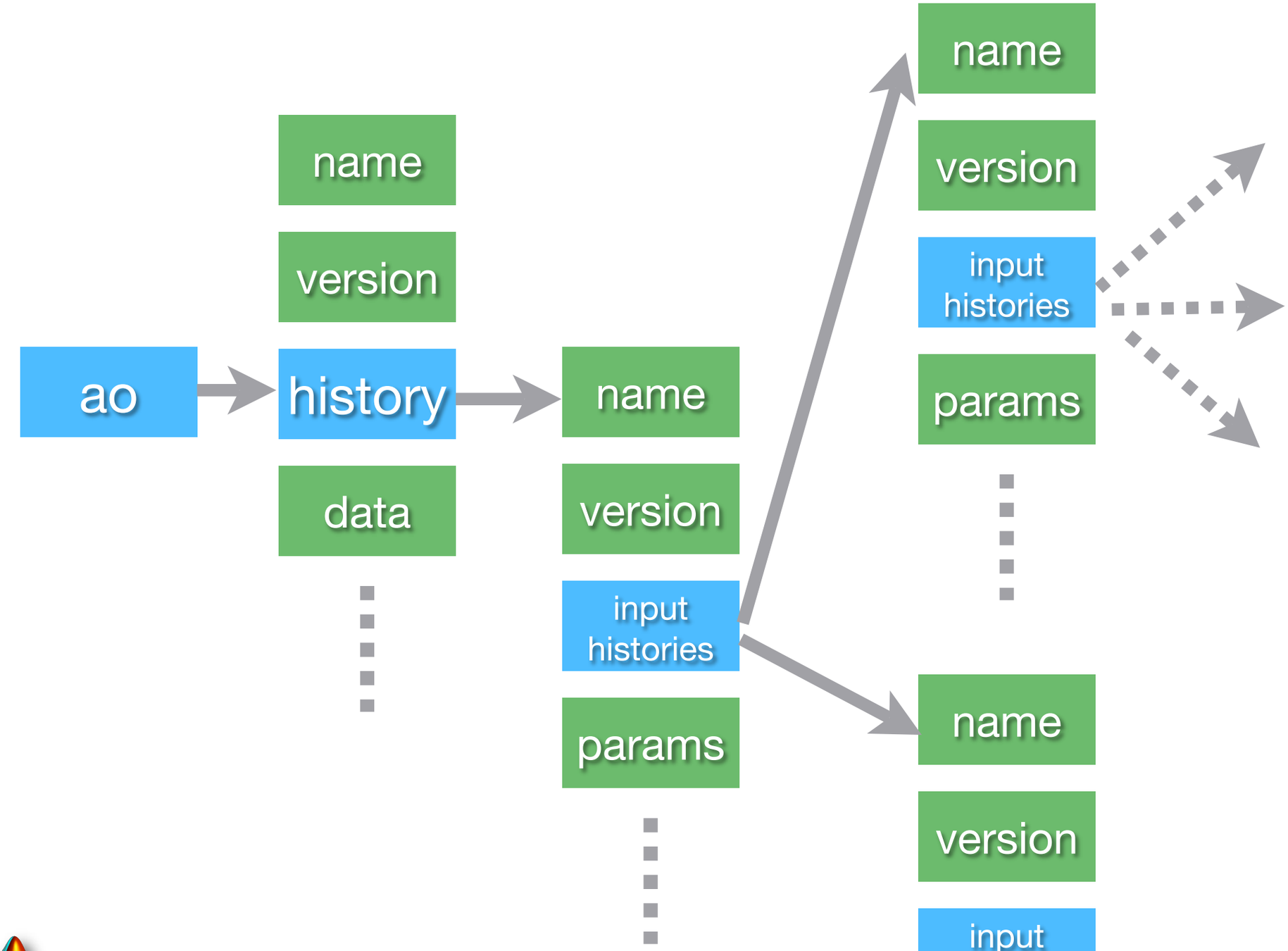
History tree



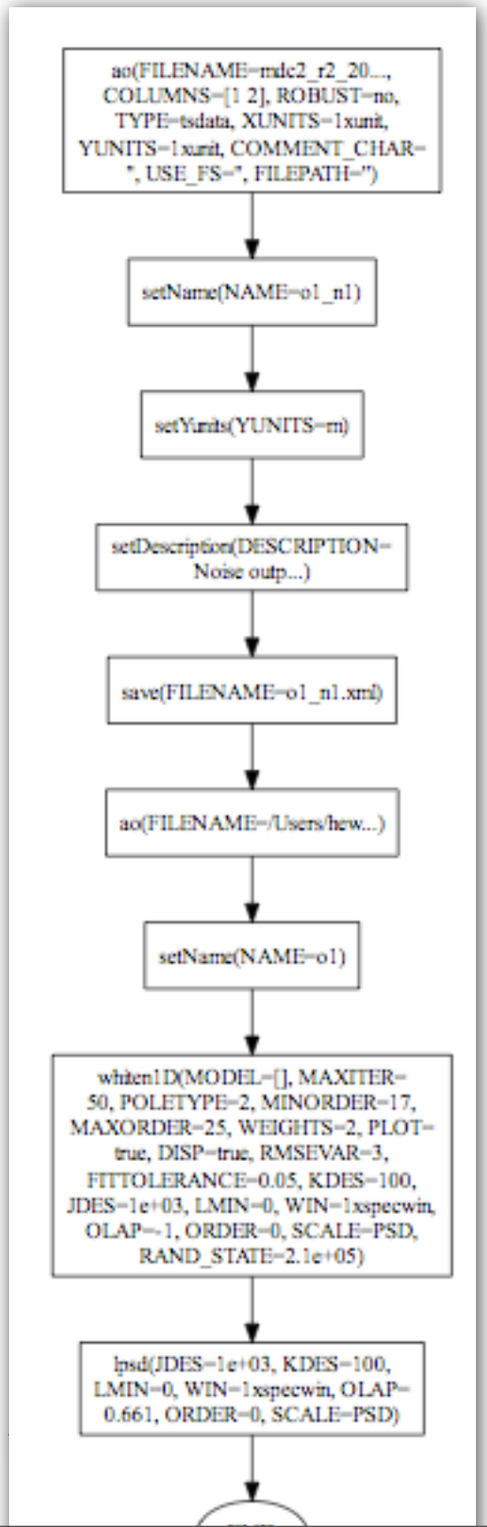
History tree

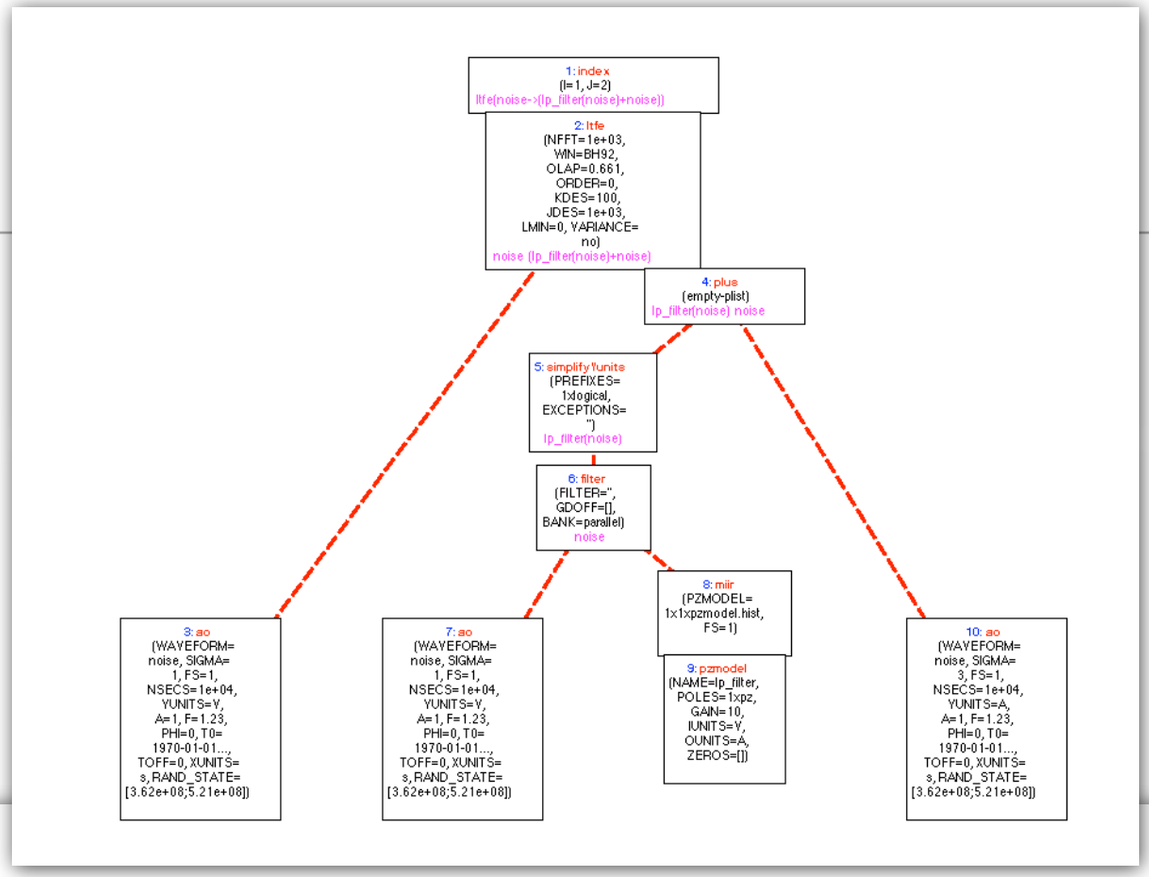


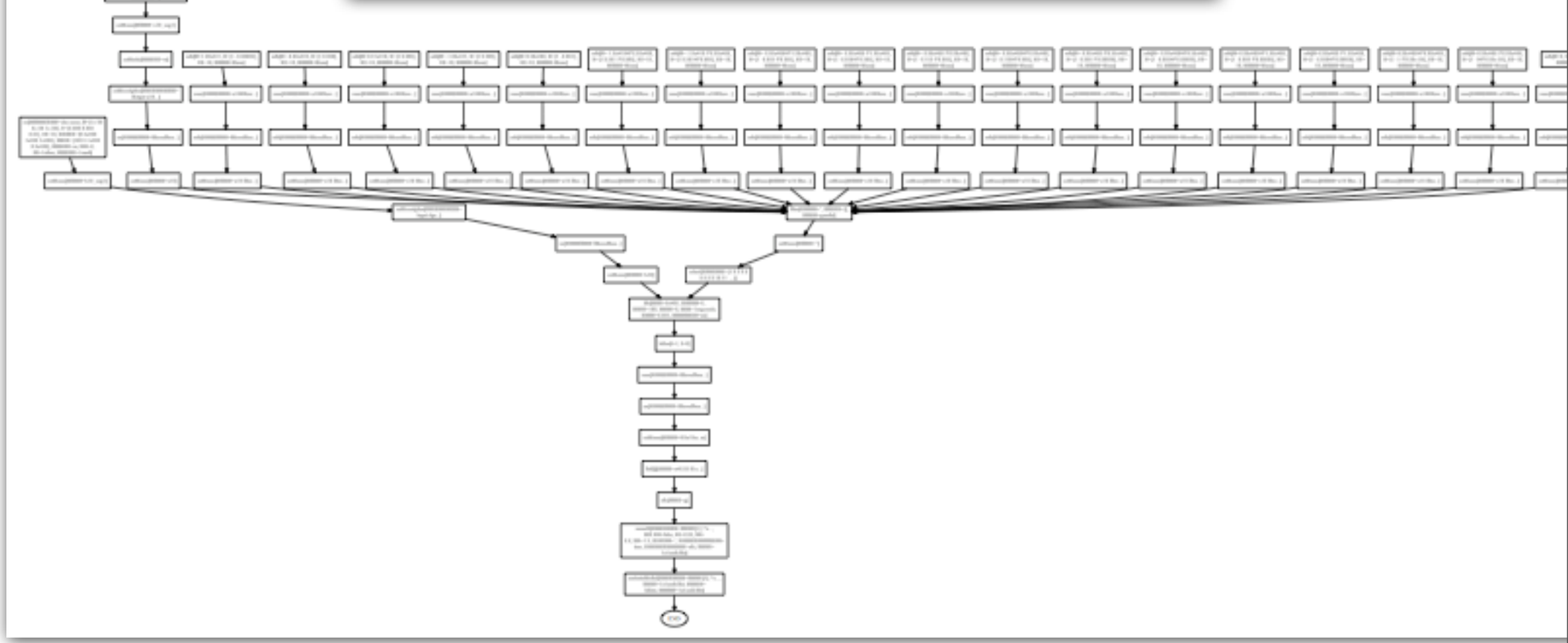
History tree



Viewing history



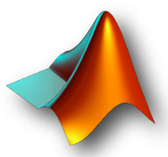




Reliving history

```
obj.type(<file>)
```
output commands needed to rebuild this object

```
robj = obj.rebuild
```
rebuild this object



LTPDA User Objects

- Not only AOs behave in this way:

pzmodel

miir

timespan

parfrac

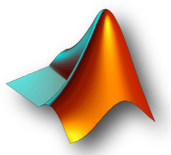
mfir

rational

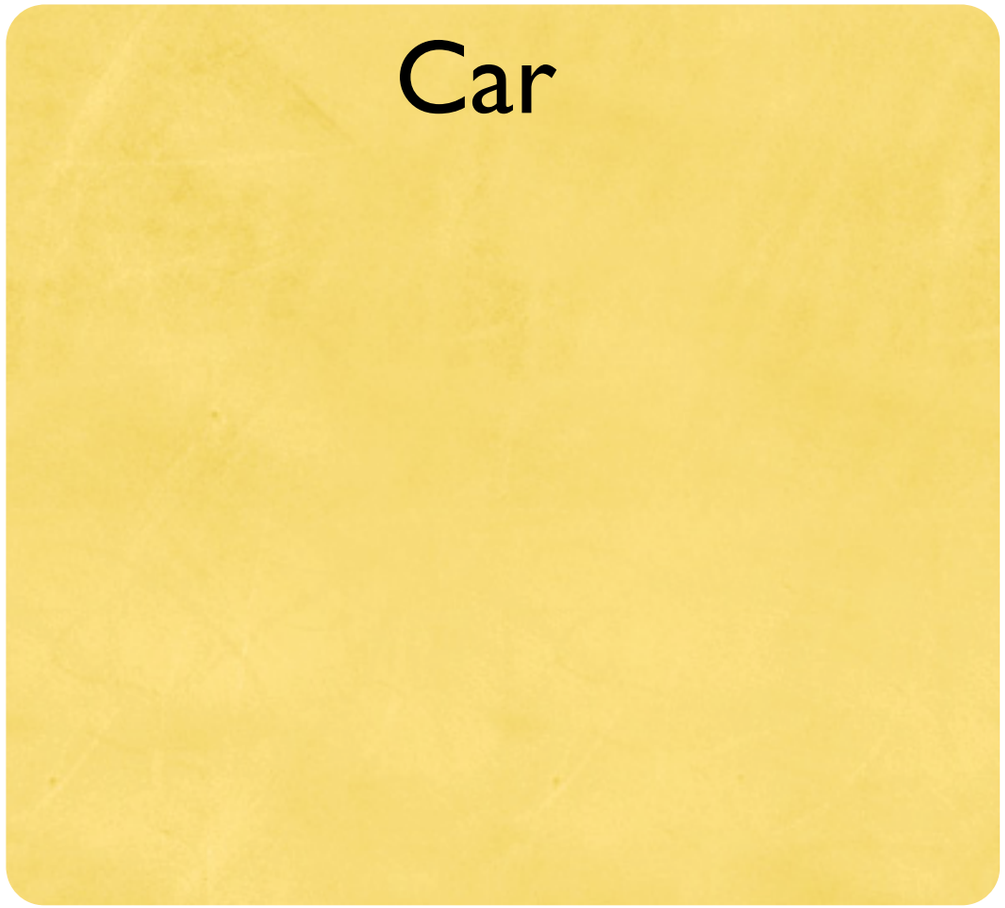
ssm

plist

Object-oriented programming for beginners



Object-oriented programming for beginners



Object-oriented programming for beginners

Car

Properties:

Object-oriented programming for beginners

Car

Properties: color

Object-oriented programming for beginners

Car

Properties: color

engine size

Object-oriented programming for beginners

Car

Properties: color

engine size

cost

Object-oriented programming for beginners

Car

Properties: color

engine size

cost

Methods:

Object-oriented programming for beginners

Car

Properties: color
engine size
cost

Methods: start

Object-oriented programming for beginners

Car

Properties: color
engine size
cost

Methods: start
stop

Object-oriented programming for beginners

Car

Properties: color
engine size
cost

Methods: start
stop
drive

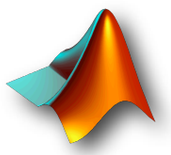
Object-oriented programming for beginners

Car

Properties: color
engine size
cost

Methods: start
stop
drive

object = instance of a class



Object-oriented programming for beginners

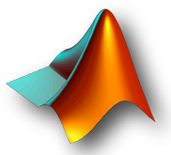
Car

Properties: color
engine size
cost

Methods: start
stop
drive

object = instance of a class

```
car1 = Car('blue')
car2 = Car('red')
```



Object-oriented programming for beginners

Car

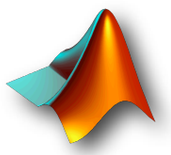
Properties: color
engine size
cost

Methods: start
stop
drive

object = instance of a class

```
car1 = Car('blue')
car2 = Car('red')
```

objects



Object-oriented programming for beginners

Car

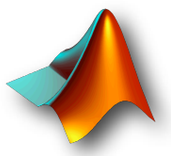
Properties: color
engine size
cost

Methods: start
stop
drive

object = instance of a class

```
car1 = Car('blue')
car2 = Car('red')
```

objects — constructors



Object-oriented programming for beginners

Car

Properties: color
engine size
cost

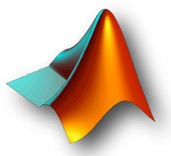
Methods: start
stop
drive

object = instance of a class

```
car1 = Car('blue')
car2 = Car('red')
```

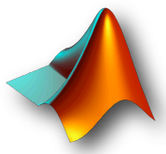
objects — constructors

```
car1.start
car1.drive
stop(car1)
```



Exercise 1

- Now it's your turn
- Open the MATLAB documentation
  - In the MATLAB terminal
    - >> doc
  - "Help -> Product Help>"
- Go to section:
  - LTPDA Toolbox
    - LTPDA Training Session 1
      - Topic 1
        - Making AOs
- And work your way through
  - "Exercise 1 - Your First Analysis Object"



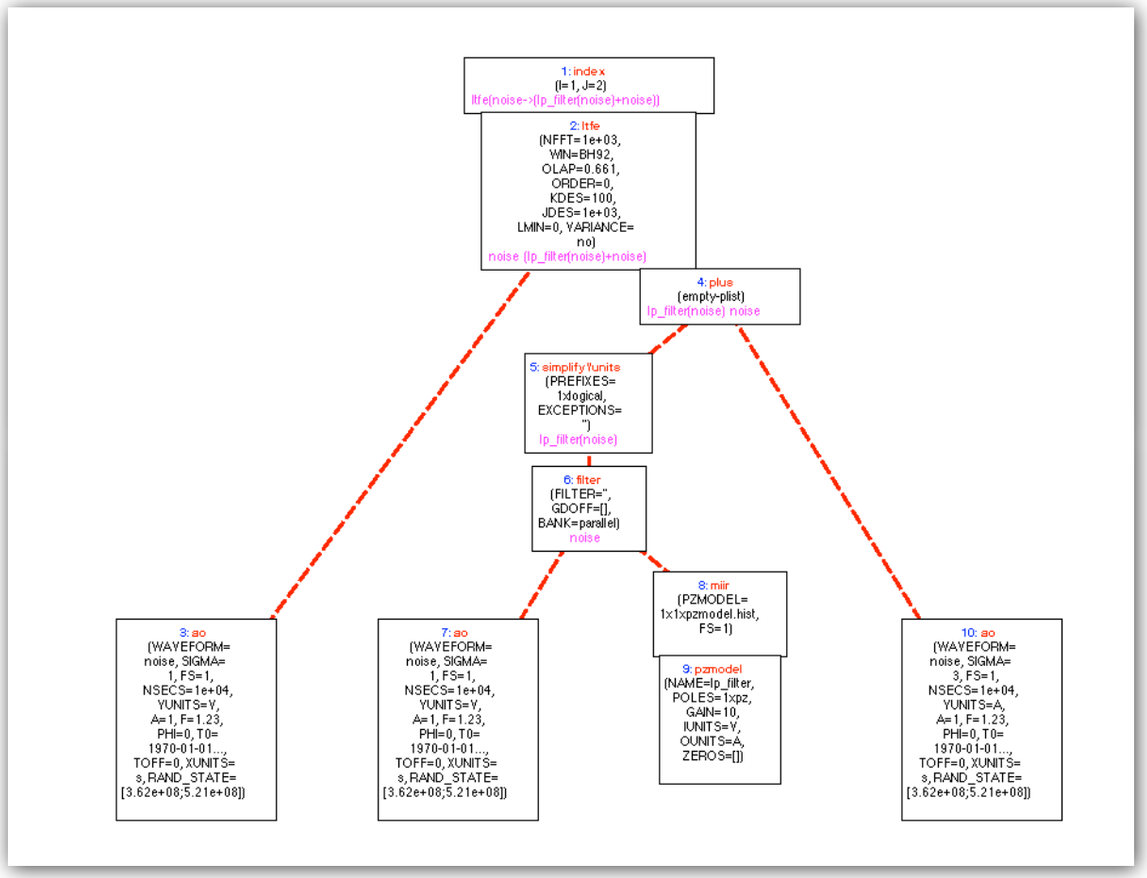
Exercise 2 - setting object properties

- AOs have properties, e.g.,
  - name
  - description
- Work through:
  - Exercise 2 - Setting properties of AOs

```
>> a.setName('Bob')
```

Exercise 3 - Viewing the history

- We saw earlier that AOs store their processing history.
- Here we learn how to view that history
- Work through the section
  - Exercise 3 - Viewing the history

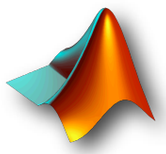


Exercise 4 - Making Time-series AOs

- AOs can contain different types of data
  - Time-series data are stored in a `tsdata` object
  - They also have properties:

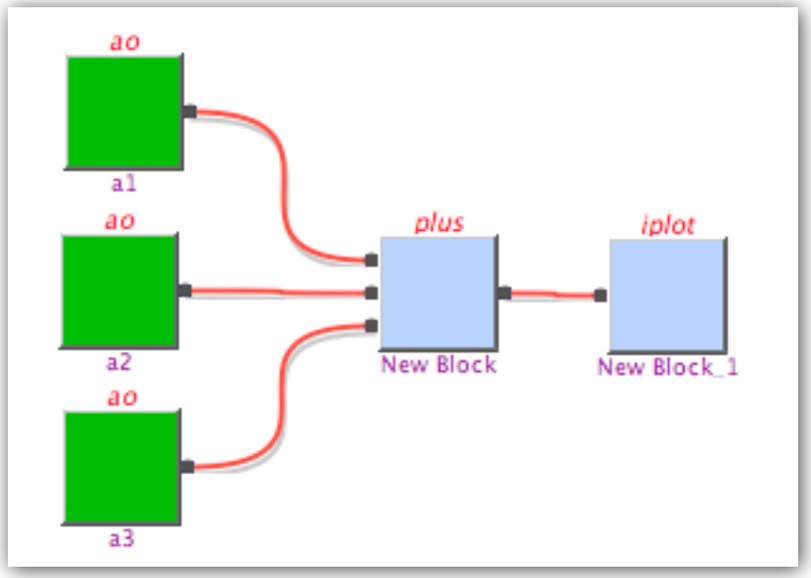
tsdata	
t0	Absolute time-stamp of first sample
xunits	X-axis units
yunits	Y-axis units

- Work through section:
  - Topic 1
    - Making a time-series AO



Basic Math

- You can operate on AOs using a large set of methods
  - In particular, many typical Math operations are available (overloaded)
    - Further details at: http://www.lisa.aei-hannover.de/ltpda/documents/files/operator_rules.pdf
- Work through the help section
  - Topic 1
    - Basic Math with AOs



Saving and loading AOs

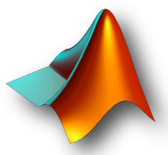
- All LTPDA User Objects can be saved to (and loaded from), file in
  - XML format
  - binary MAT format
- Work through help section
  - Topic 1
    - Saving and loading AOs

```
<?xml version="1.0" encoding="utf-8"?>
<ltpda_object ltpda_version="2.0 (R2008b)">
   <object shape="1x1" type="ao">
      <property prop_name="data" shape="1x1" type="fsdata">
         <object shape="1x1" type="fsdata">
            <property prop_name="t0" shape="1x1" type="time"
               <object shape="1x1" type="time">
                  <property prop_name="utc_epoch_milli" shap
                  <property prop_name="timezone" shape="1x1"
                  <property prop_name="timeformat" shape="1x
                  <property prop_name="time_str" shape="0x0"
                  <property prop_name="version" shape="1x53"
               </object>
            </property>
            <property prop_name="navs" shape="1x1" type="dou
            <property prop_name="fs" shape="1x1" type="doubl
            <property prop_name="enbw" shape="1x1" type="dou
            <property prop_name="version" shape="1x55" type=
            <property prop_name="xunits" shape="1x1" type="u
               <object shape="1x1" type="unit">
                  <property prop_name="strs" shape="1x1" typ
```

Reading data files

- You can construct AOs from existing ASCII (raw) data files
- Work through help section
  - Topic 1
    - Constructing AOs from data files

```
a = ao('topic1/simpleASCII.txt')
```



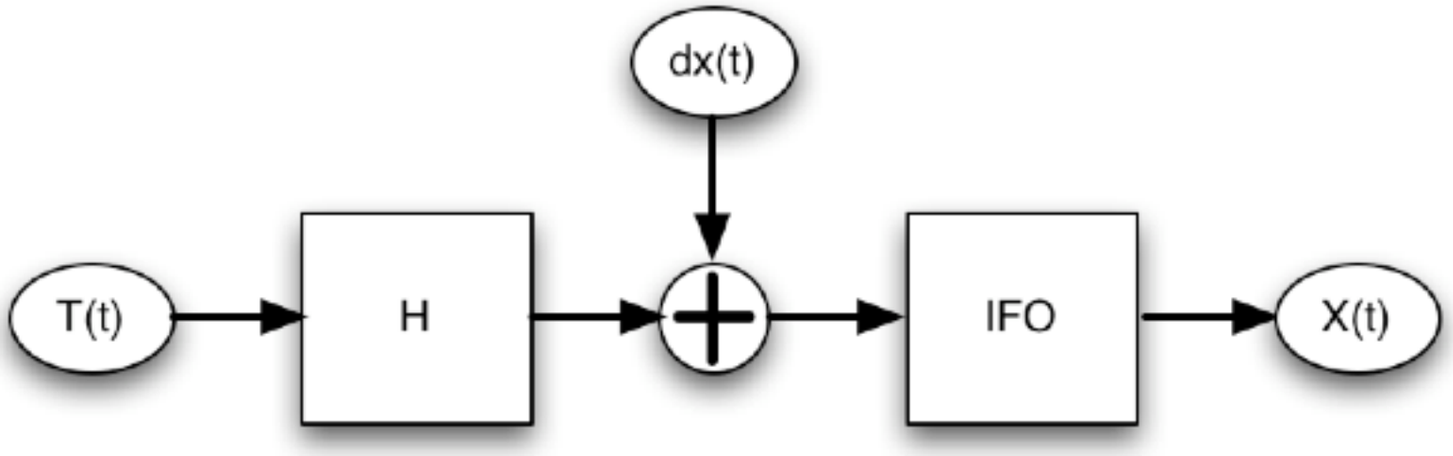
Writing LTPDA scripts

- So far we've done everything on the command-line or on the workbench
- Now we look at writing LTPDA scripts
  - There are some subtle differences to standard MATLAB scripts and some recommended practices
- Read through help section
  - Topic 1
    - Writing LTPDA scripts
- ** More details on the `tfe` method later

```
%% Make two test AOs

a1 = ao(plist('tsfcn', 'randn(size(t)
a2 = ao(plist('tsfcn', 'randn(size(t)

%% Make TFE with multiple outputs

[t11, t21, t12, t22] = tfe(a1,a2);
Axx = t12 ./ t21;

%% Make TFE then index with ()

txx = tfe(a1,a2);
Axx = txx(1,2) ./ txx(2,1); % <-- BRE

%% Use index method

txx = tfe(a1,a2);
```

IFO/Temperature example

- We have a data analysis exercise which will develop fully over the course of the training session
- This is the first part: reading and preparing the data
- Work through help section
  - Topic 1
    - IFO/Temperature Example - Introduction

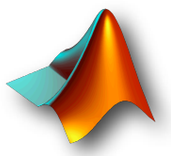


Topic 2
Signal Pre-Processing

Anneke Monsky

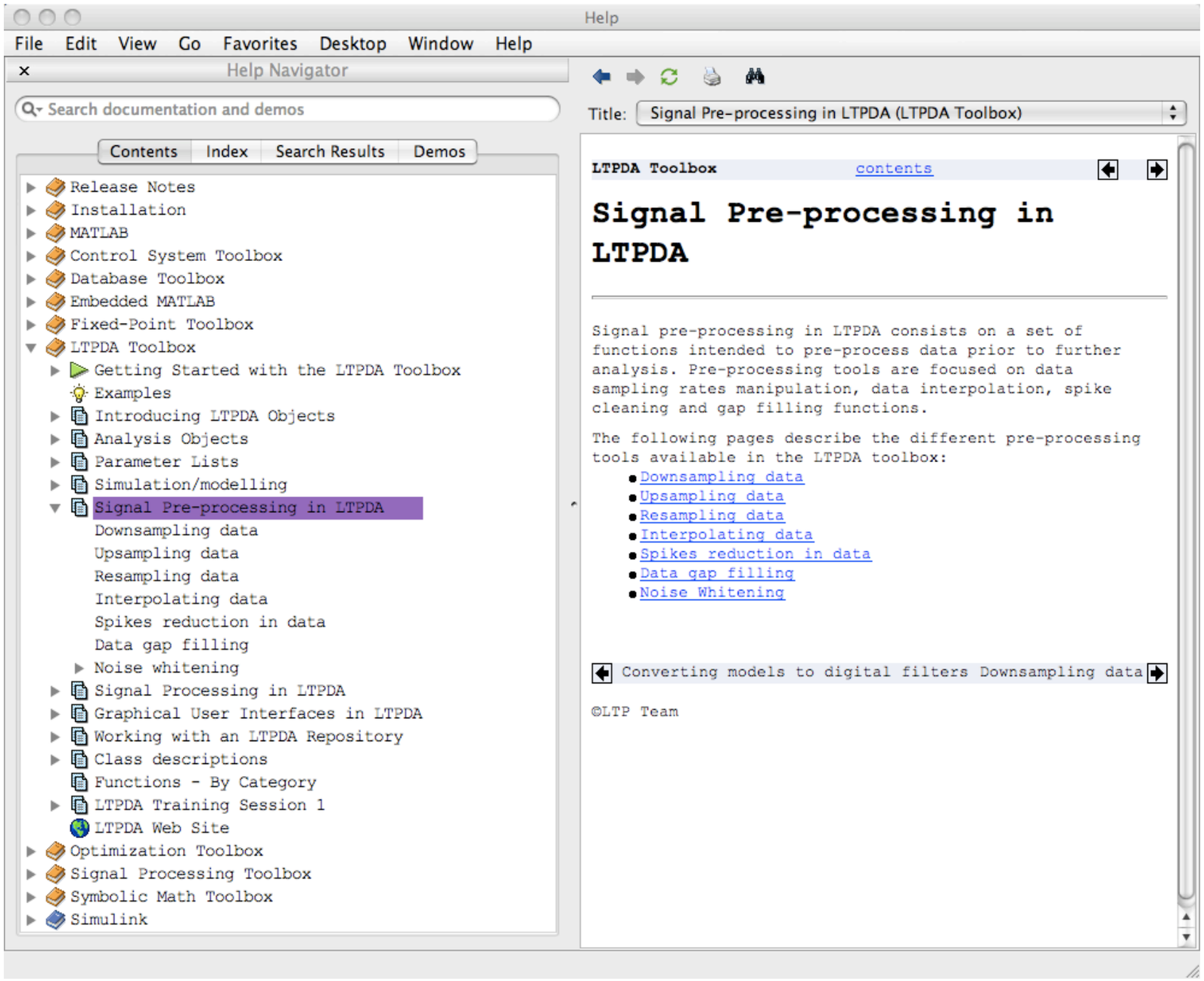
Signal

- data preparation for further analysis
- toolbox preserves a bunch of function for
  - resampling
  - interpolation
  - basic fitting routines (detrend)
  - noise whitening
  - selecting methods



Documentation

LTPDA Toolbox - Signal Pre-processing in LTPDA



Changing the sample rate

- Integer factor
  - Down-sample - to reduce data load
  - Up-sample - to match sample rates
- Re-sample
  - fs_out = P/Q * fs_in (P and Q are integers)

Changing the sample rate

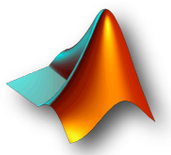
- Integer factor
  - Down-sample - to reduce data load
  - Up-sample - to match sample rates
- Re-sample
  - fs_out = P/Q * fs_in (P and Q are integers)

Parameters

Downsample	Upsample	Resample
offset	delay	filter

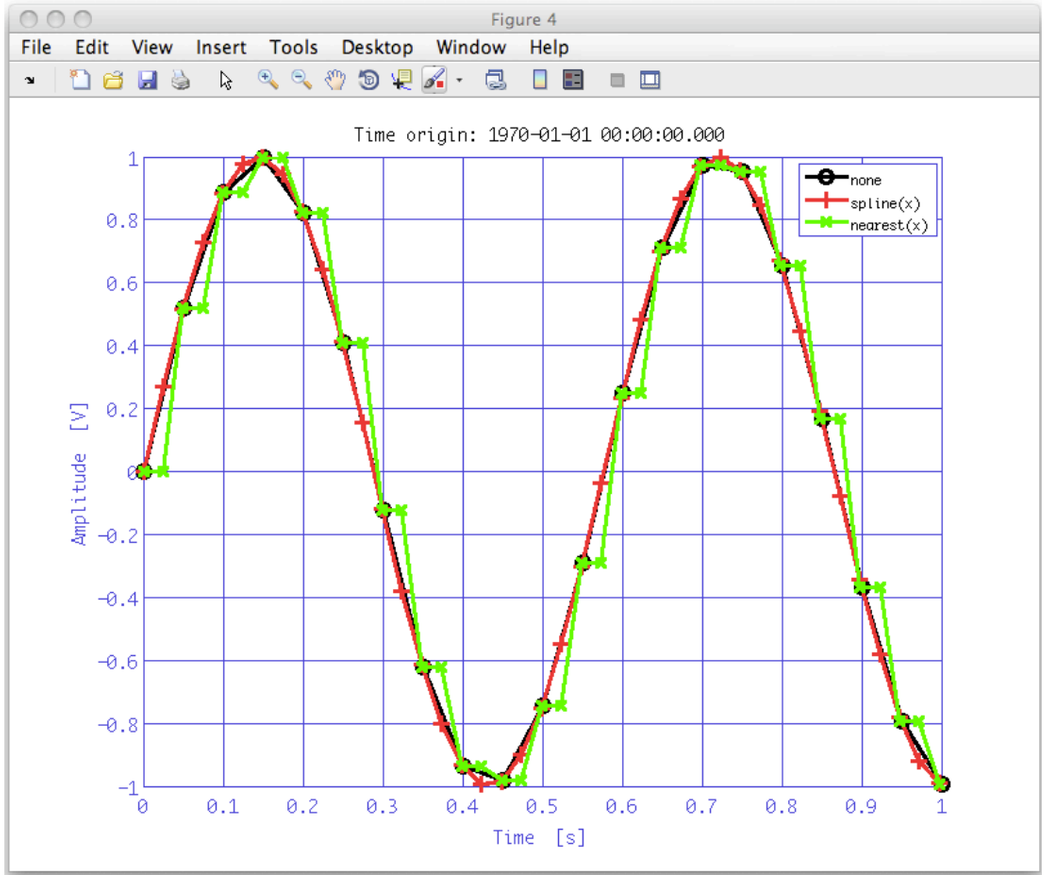
Topic 2
Exercises 1,2,3

- Open MATLAB documentation
  - In the MATLAB terminal
    - >> doc
  - “Help -> Product Help>”
- work through
  - LTPDA Toolbox - LTPDA Training Session 1 Topic 2
    - Downsampling
    - Upsampling
    - Resampling



Interpolation

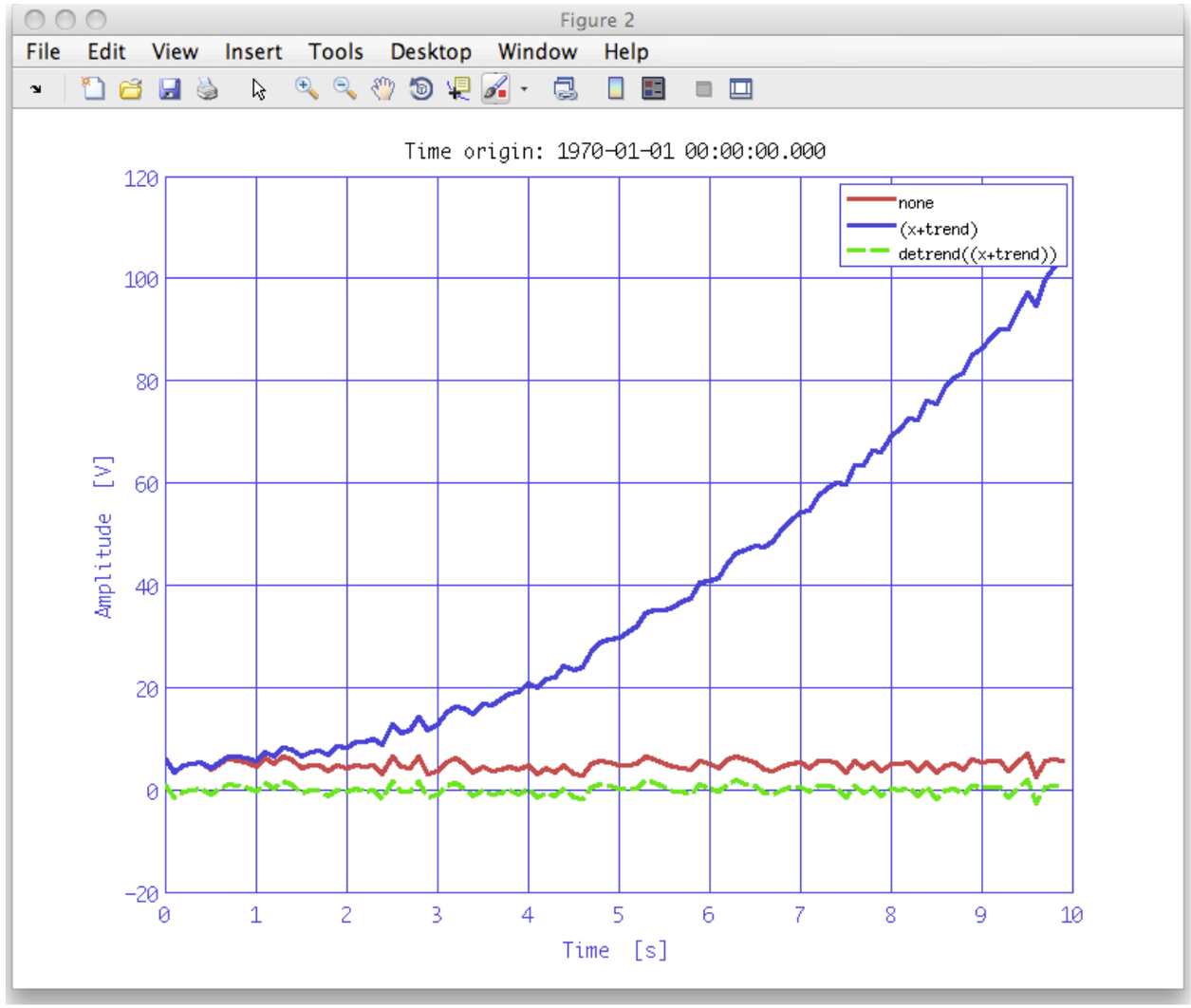
‘vertices’	new time grid
interpolation methods	
‘linear’	linear
‘spline’	spline
‘cubic’	cubic
‘nearest’	nearest



- work through
  - LTPDA Toolbox - LTPDA Training Session 1 Topic 2
    - Interpolation

Detrending

- Remove trends by
  - subtracting polynomial fit from data
- ao/detrend calls MATLABs polyfit
- work through
  - LTPDA Toolbox - LTPDA Training Session 1 Topic 2
    - detrending



Whitening

- The LTDA Toolbox offers various ways to white your data
  - with a known filter
    - build filter and apply it to your data
  - with a known model of spectral content
    - use whiten1D
      - for single, uncorrelated data streams
    - whiten2D
      - for a pair of correlated data streams
  - without model (Exercise)
    - let whiten1D fit a model to the spectrum of your data

- work through
  - LTPDA Toolbox - LTPDA Training Session 1 Topic 2
    - whitening

Select & find/ split & join

- Chose the samples you want to analyse
  - find/select data samples by their properties
    - sample numbers - ‘select’
    - query for x and y values - ‘find’
  - split data by
    - intervals, times, frequencies, samples
- Group of functions helps you to
  - for find and select exactly the data you want, split your data into pieces and eventually
  - join them back together

- work through
  - LTPDA Toolbox - LTPDA Training Session 1 Topic 2
    - select and find
    - split and join

Pre-processing the IFO/Temp example

- One function that combines all necessary operations
  - consolidate
- consolidate fixes our two data streams such that
  - they start at the same time
  - they have the same sampling rate
  - are evenly sampled on the same grid

- work through
  - LTPDA Toolbox - LTPDA Training Session 1 Topic 2
    - IFO/Temp example

LTPDA Training session

Topic 3: spectral analysis

Mauro Hueller

AEI Hannover, 09-10/03/2009

Power Spectral Density Estimation (1)

Definition:

$$P_{xx}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{+\infty} R_{xx}(m) \exp(-2\pi i \cdot f \cdot m / f_s)$$

Estimates the *one-sided* PSD:

$$\tilde{P}_{xx}(f) = \left\{ \begin{array}{ll} 0, & -\frac{f_s}{2} \le f \le 0 \\ 2P_{xx}(f), & 0 \le f \le \frac{f_s}{2} \end{array} \right\}$$

Power Spectral Density Estimation (2)

The PSD at each frequency is estimated via the Welch method:

Given a discretized signal $x[n]$ of length *N*

- Data are divided into segments of length *L* and multiplied by a window:

$$w(n)$$

this also reduces the edge-effects (simulating a periodic sequence)

The PSD at each frequency *f* is estimated as:

$$\hat{P}_{xx}(f) = \frac{|X_L|^2}{f_s \cdot L \cdot U} \qquad \text{where}$$

$$x_L[n] = x[n]w[n]$$

$$x_L(f) = \sum_{n=0}^{L-1} x_L[n] \exp\left(-2\pi i \cdot \frac{f \cdot n}{f_s}\right) \qquad U = \frac{1}{L}\sum_{n=0}^{L-1} \left| w(n)^2 \right|$$

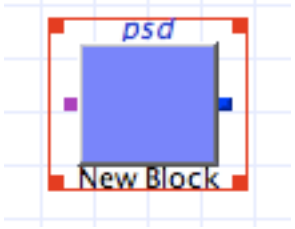
Power Spectral Density Estimation (3)

- Methods:
  - `ao/psd`: linear frequency scale
  - `ao/lpsd`: log-frequency scale
  - `specwin`: implements spectral windows

Power Spectral Density Estimation (4)

```
a = [ao with time-series data]
S = a.psd(plist('win',win,…
  'nfft',nfft,'olap',olap,…
  'order',order,'scale',scale))
```

Or add a block on a workbench:



Power Spectral Density Parameters

Name	Description	Values	Default
scale	the output quantity	'PSD' gives Power Spectral Density [m^2 Hz^-1] 'ASD' gives Amplitude Spectral Density [m Hz^-1/2] 'PS' gives Power Spectrum [m^2] 'ASD' gives Amplitude Spectrum [m]	'PSD'
win	A spectral window to multiply the data by	'BH92' or 'Rectangular' or … (name or object)	Taken from user preferences
nfft	Window length	-1: one window, length = ao data set Or: length (number of points)	-1
order	Order of segment detrending (prior to windowing)	-1: no detrending 0: mean subtraction N: order N polynomial trend subtraction	0
olap	Percentage overlap between adjacent segments	-1: taken from window parameters 0: no overlap 100: total overlap	-1

Power Spectral Density Estimation (5)

Features:

- Multiple inputs:

```
S = psd(a1,a2,a3,plist())
```

- Matrix inputs:

```
S = psd([a1 a2;a3 a4],plist())
```

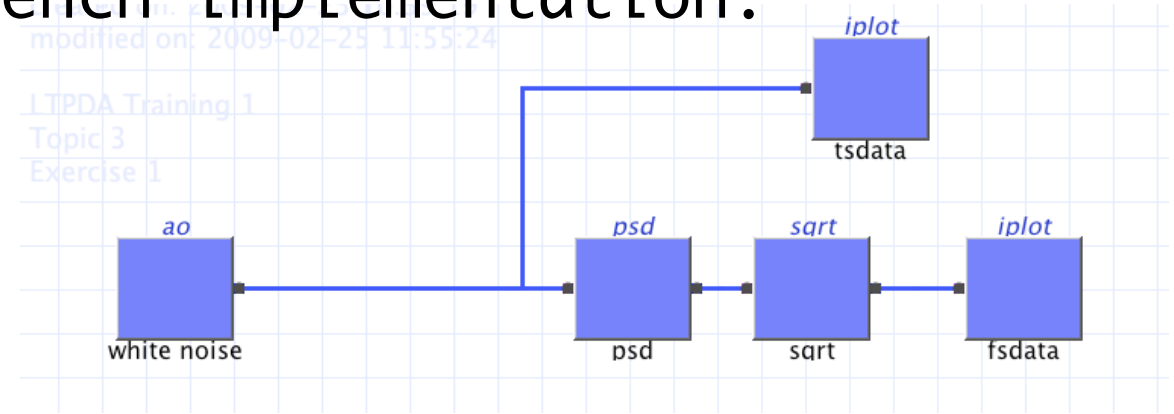
PSD Exercise 1

Very simple idea:

Parameters for psd:
all default

PSD Exercise 1

- Workbench implementation:



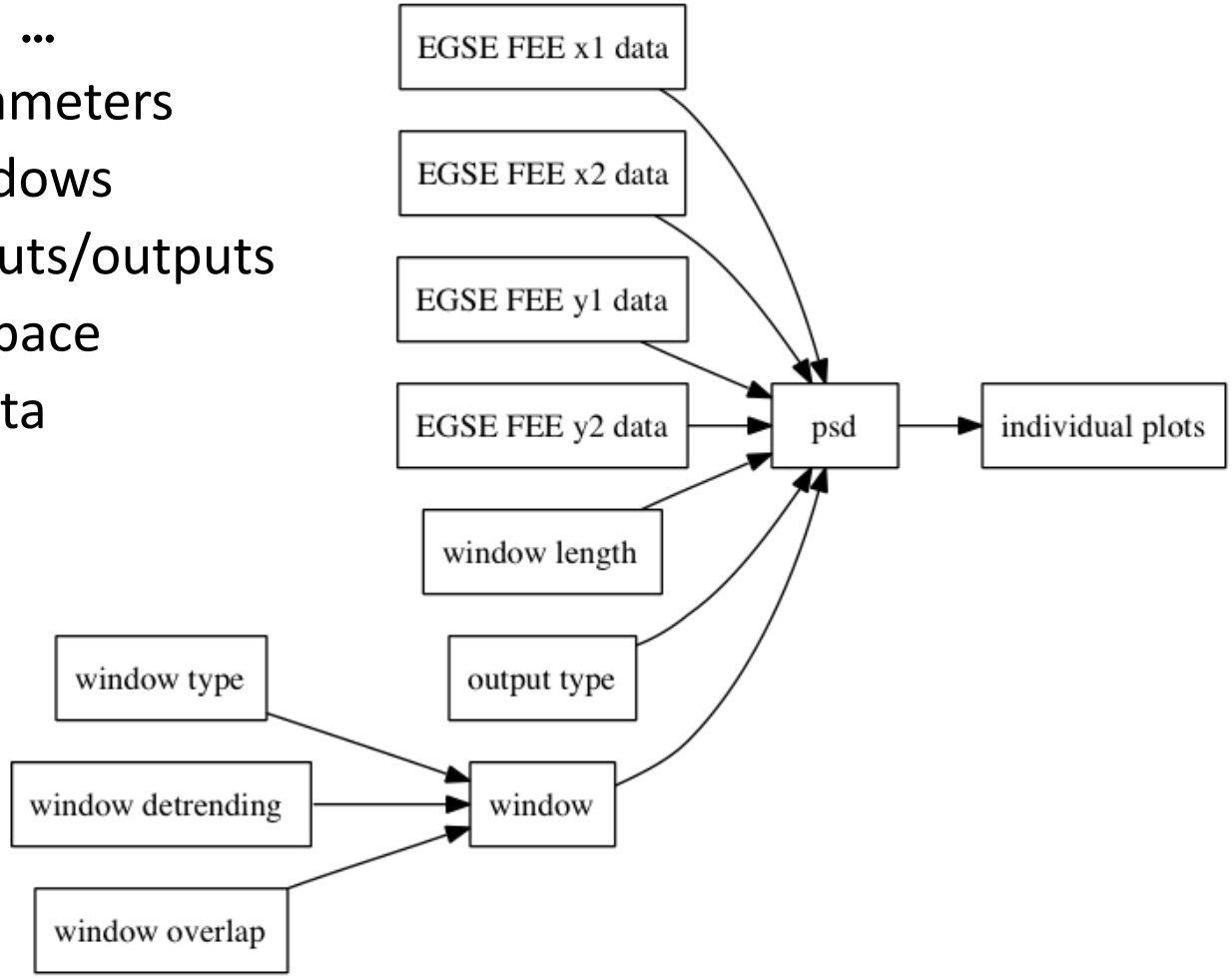
- Matlab terminal implementation:

```
>> a1 =
   ao(plist('waveform','noise','type','normal','nsecs',
   1000,'sigma',1.0,'yunits','m'))
>> iplot(a1)
>> S1 = a1.psd(plist('win','BH92'))
>> P1 = sqrt(S1)
>> iplot(P1)
```

PSD Exercise 2

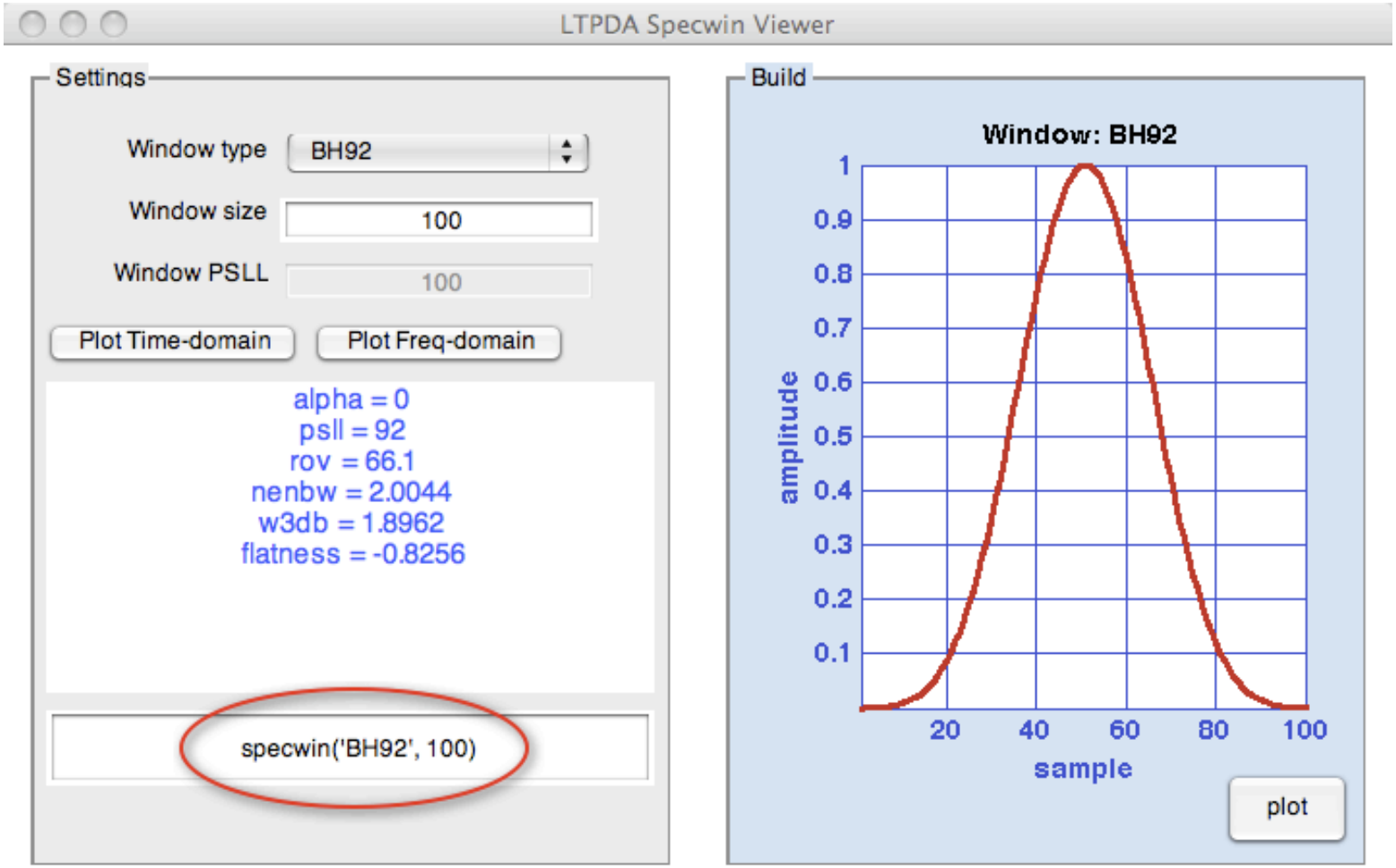
More involved …

- Playing with parameters
- Playing with windows
- Adding block inputs/outputs
- Output to workspace
- Saving output data



PSD Exercise 2

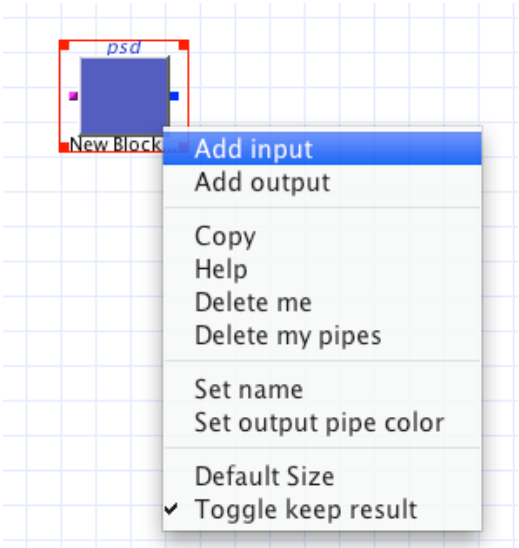
Spectral windows

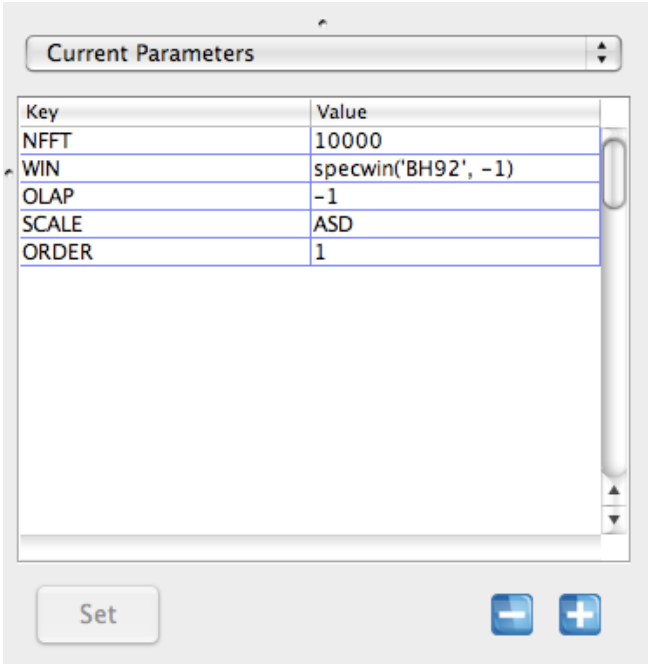


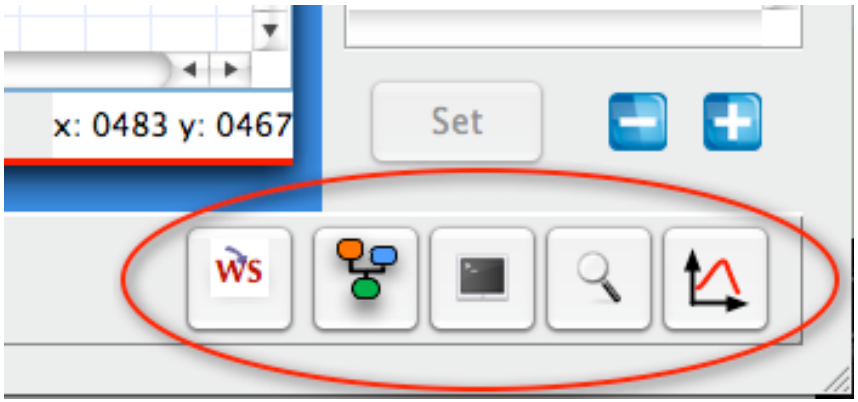
PSD Exercise 2

More involved …

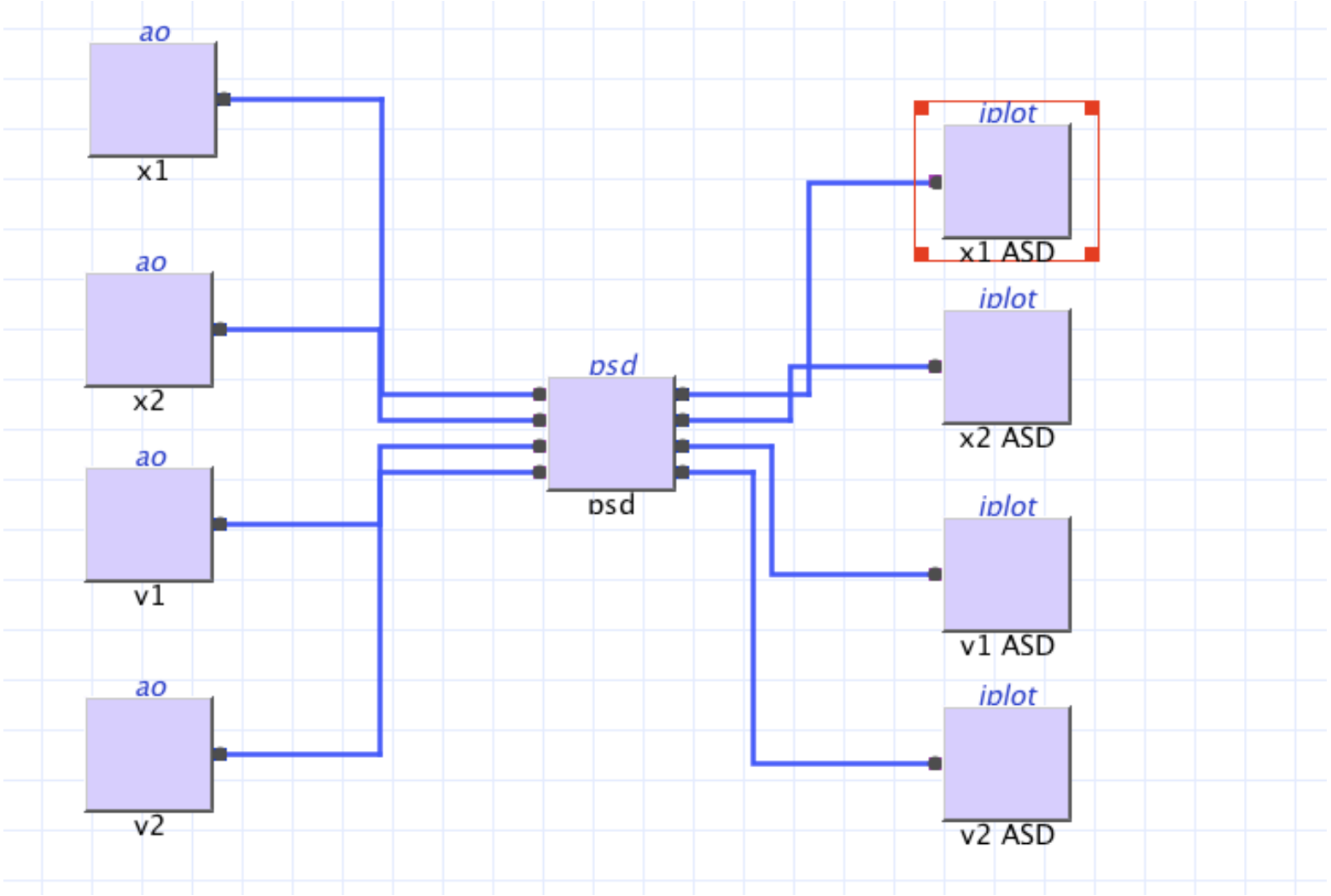
- Playing with parameters
- Adding inputs/outputs to blocks
- Output to workspace
- Saving output data







PSD Exercise 2



Log-Scale Power Spectral Density Estimation

Implementation of the algorithm described in
"Measurement 39 (2006) 120-129"

The same as `psd` but:

- Reduces individual point variance by adjusting the window length at each frequency
- Frequency bins and number of averages are calculated automatically
- Slower because of the much higher number of DFT evaluation
- Energy content of the spectrum is preserved
- Reduced resolution due to shorter window length
- Lower uncertainty

Log-Scale Power Spectral Density Parameters

The same as `psd` but with:

Name	Description	Values	Default
Kdes	Desired number of averages	An integer number	100
Jdes	Number of spectral frequencies to calculate	An integer number	fs/2
Lmin	Minimum segment length	An integer number	0

And `nfft` has no meaning, that is calculated for each frequency

Log-scale Power Spectral Density Estimation

Features:

- Multiple inputs:

```
S = lpsd(a1,a2,a3,plist())
```

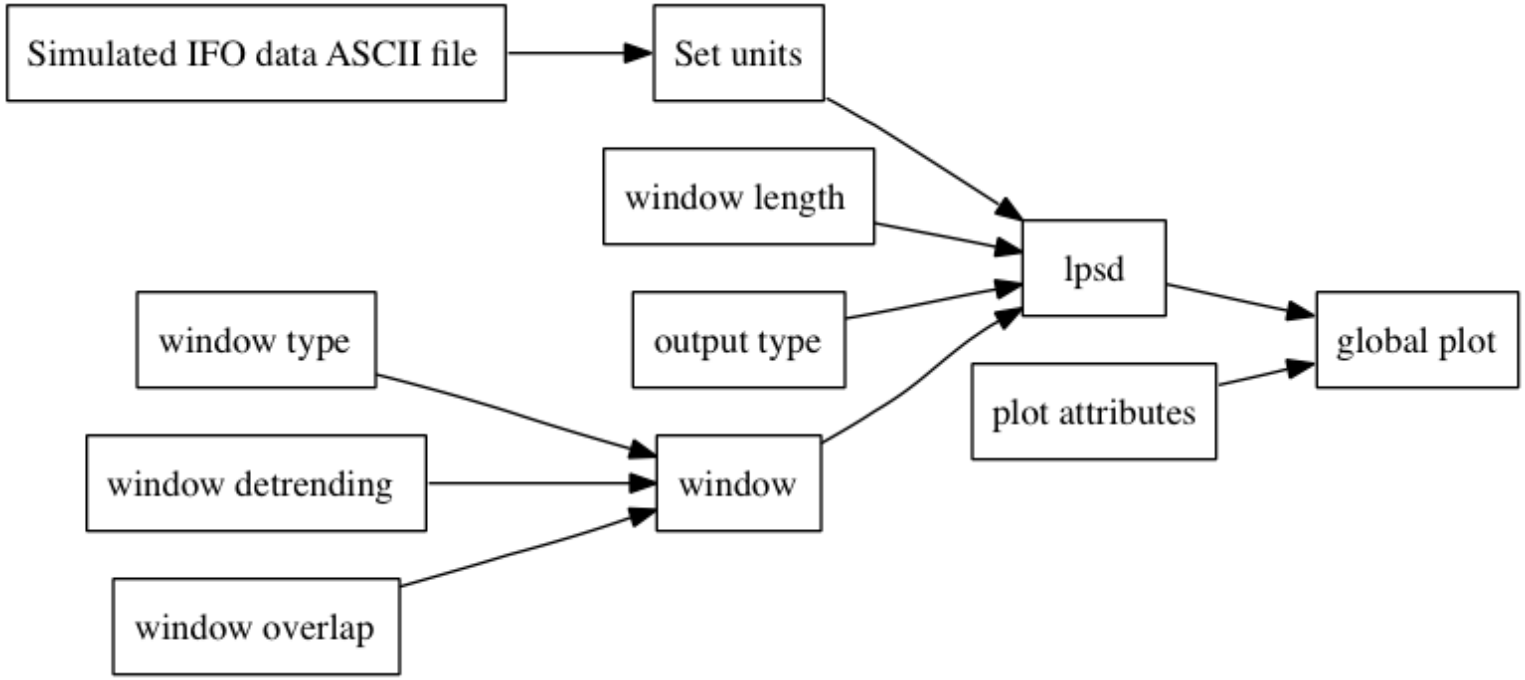
- Matrix inputs:

```
S = lpsd([a1 a2;a3 a4],plist())
```

PSD Exercise 3

`Using lpsd`

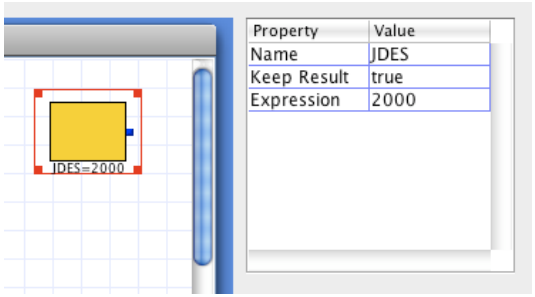
- Log-scale psd calculation to reduce variance
- Using MDC1 IFO data
- Setting units
- Setting plot features

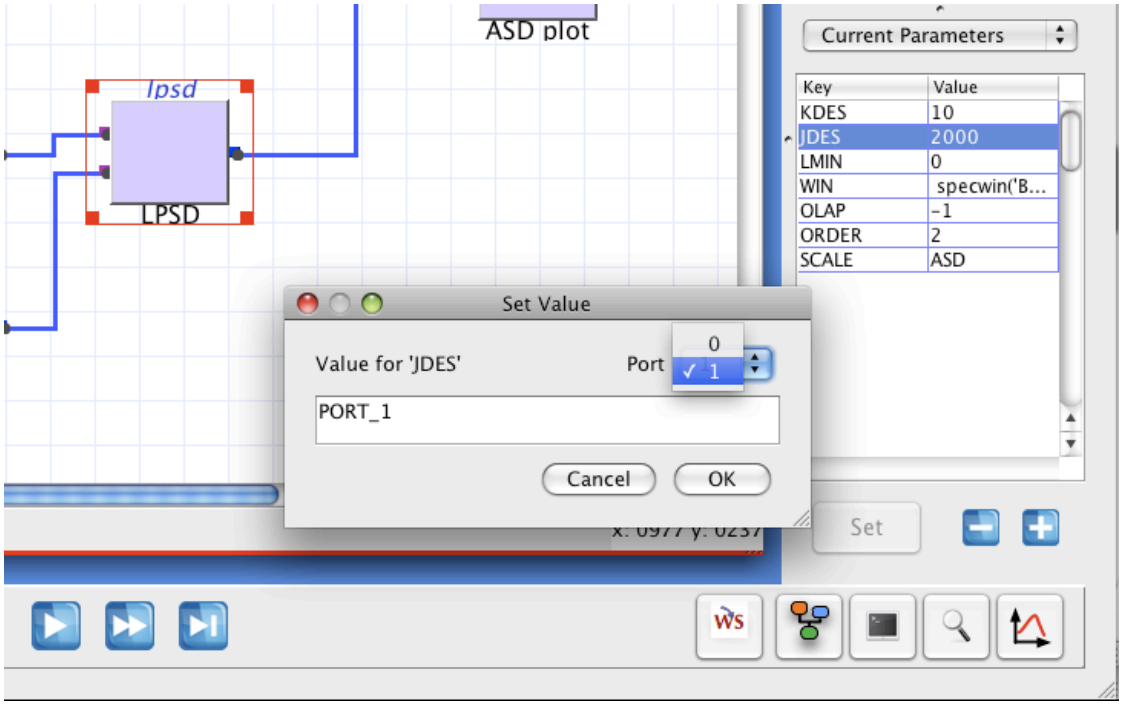


PSD Exercise 3

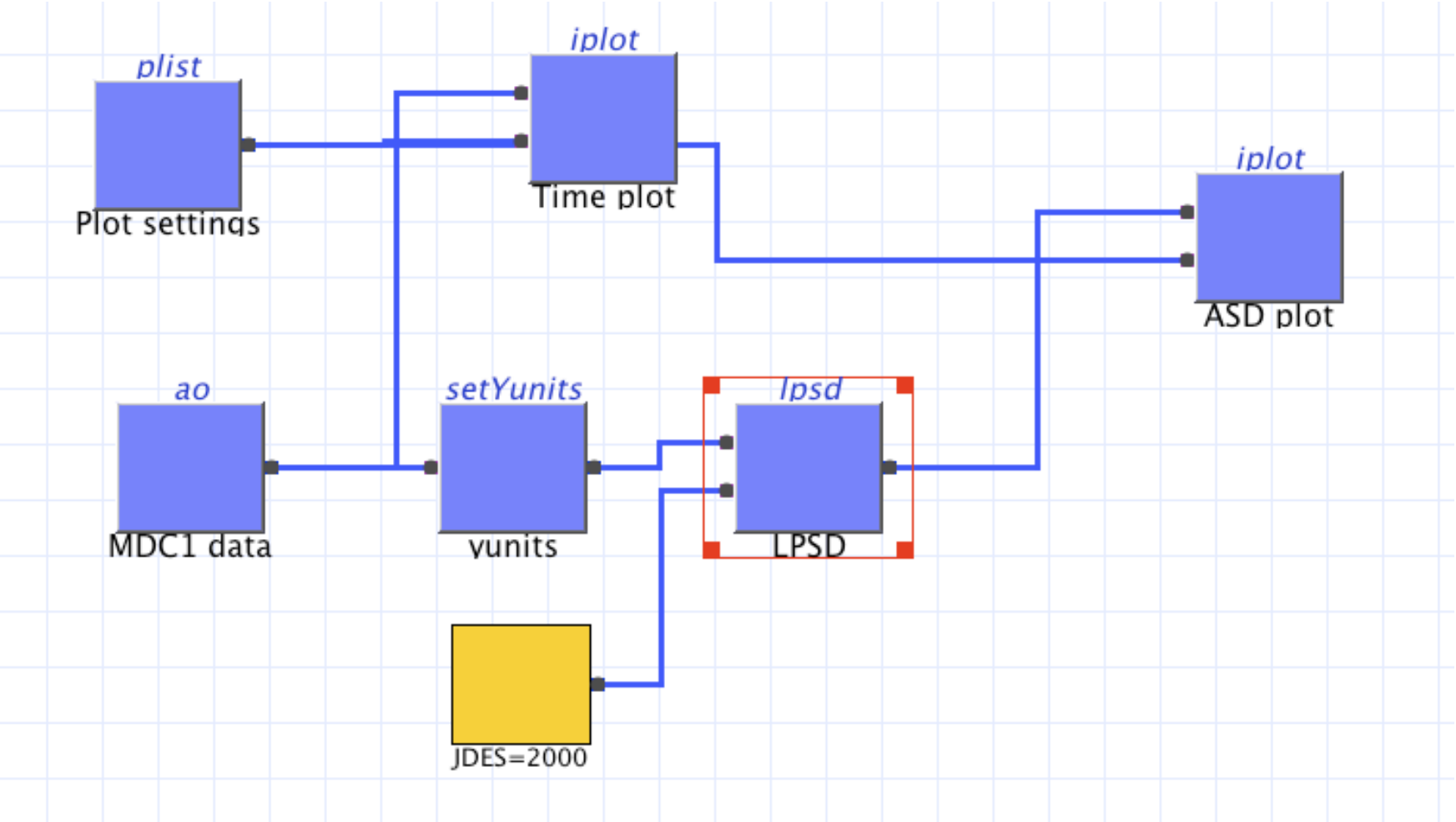
Using `lpsd`

- Log-scale psd calculation to reduce variance
- Using MDC1 IFO data
- Setting units
- Setting plot features
- Adding constant blocks





PSD Exercise 3



Cross- Power Spectral Density Estimation

Definition:

$$C_{xy}(f) = \frac{1}{f_s} \sum_{m=-\infty}^{+\infty} R_{xy}(m) \exp(-2\pi i \cdot f \cdot m / f_s)$$

Estimates the *one-sided* CPSD:

$$\tilde{C}_{xy}(f) = \left\{ \begin{array}{ll} 0, & -\frac{f_s}{2} \le f \le 0 \\ 2P_{xy}(f), & 0 \le f \le \frac{f_s}{2} \end{array} \right\}$$

Cross- Power Spectral Density Estimation (2)

The CPSD at each frequency is estimated via the Welch method:

Given two discretized signals $x[n]\ y[n]$ of length *N*

Data are divided into segments of length *L* and multiplied by a window:

$$w(n)$$

this also reduces the edge-effects (simulating a periodic sequence)

The CPSD at each frequency *f* is estimated as:

$$\hat{C}_{xy}(f) = \frac{X_L Y_L^*}{f_s \cdot L \cdot U}$$

where

$$x_L(f) = \sum_{n=0}^{L-1} x_L[n] \exp\left(-2\pi i \cdot \frac{f \cdot n}{f_s}\right) \qquad U = \frac{1}{L}\sum_{n=0}^{L-1} \left| w(n)^2 \right|$$

Cross- Power Spectral Density Estimation (3)

- Methods:
  - ao/cpsd: linear frequency scale
  - ao/lcpsd: log-frequency scale

Similarly, we can evaluate *coherence*:

$$Coh_{xy}(f) = \frac{\left|C_{xy}(f)\right|^2}{P_{xx}(f)P_{yy}(f)}$$

- Methods:
  - ao/cohere: linear frequency scale
  - ao/lcohere: log-frequency scale

Cross- Power Spectral Density Estimation (4)

```
C = cpsd(x,y,plist('win',win,…
  'nfft',nfft,'olap',olap,…
  'order',order,'scale',scale))
Cxy = index(C,1,2)
Cyx = index(C,2,1)
Cxx = index(C,1,1)
```

Cross- Power Spectral Density Parameters

Name	Description	Values	Default
win	A spectral window to multiply the data by	'BH92' or 'Rectangular' or … name or object	Taken from user preferences
nfft	Length of the window	-1: one window, length = ao data set length Or: length (number of points)	-1
order	Order of segment detrending (prior to windowing)	-1: no detrending 0: mean subtraction N: order N polynomial trend subtraction	0
olap	Percentage overlap between adjacent segments	-1: taken from window parameters 0: no overlap 100: total overlap	-1

And similarly for `lcpsd` …

Transfer Function Estimation

Definition:

$$T_{xy}(f) = \frac{C_{xy}(f)}{P_{yy}(f)}$$

The TFE at each frequency is estimated via the Welch method

- Methods:
  - ao/`tfe`: linear frequency scale
  - ao/`ltfe`: log-frequency scale

Transfer Function Estimation (2)

```
T = tfe(x,y,plist('win',win,…
  'nfft',nfft,'olap',olap,…
  'order',order,'scale',scale))
Txy = index(C,1,2)
Tyx = index(C,2,1)
Txx = index(C,1,1)
```

Transfer Function Estimation Parameters

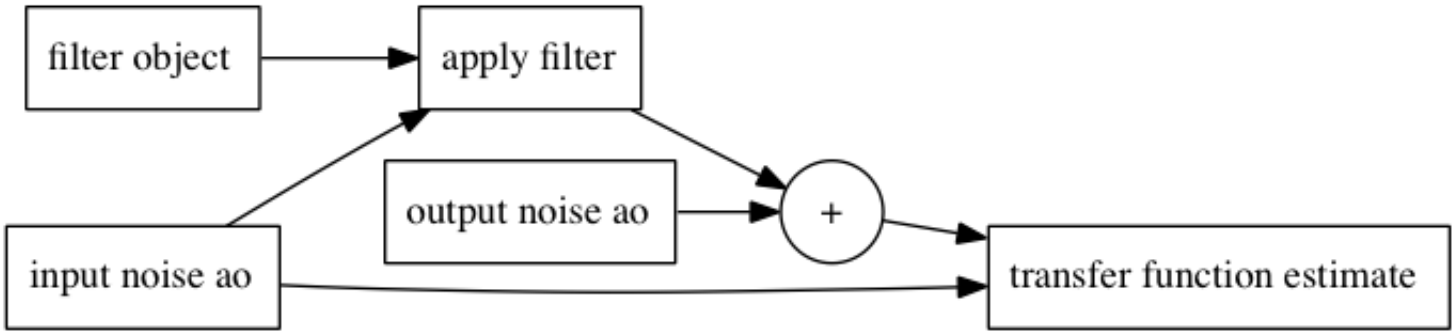
Name	Description	Values	Default
win	A spectral window to multiply the data by	'BH92' or 'Rectangular' or … name or object	Taken from user preferences
nfft	Length of the window	-1: one window, length = ao data set length Or: length (number of points)	-1
order	Order of segment detrending (prior to windowing)	-1: no detrending 0: mean subtraction N: order N polynomial trend subtraction	0
olap	Percentage overlap between adjacent segments	-1: taken from window parameters 0: no overlap 100: total overlap	-1
Variance	Computes transfer function variance	'yes' or 'no'	'no'

And similarly for `ltfe` …

TFE Exercise 1

Using `tfe`

- Simulated data input: white noise
- Band-pass filter object
- Filtering the input noise
- Adding output white noise
- Estimate the transfer function

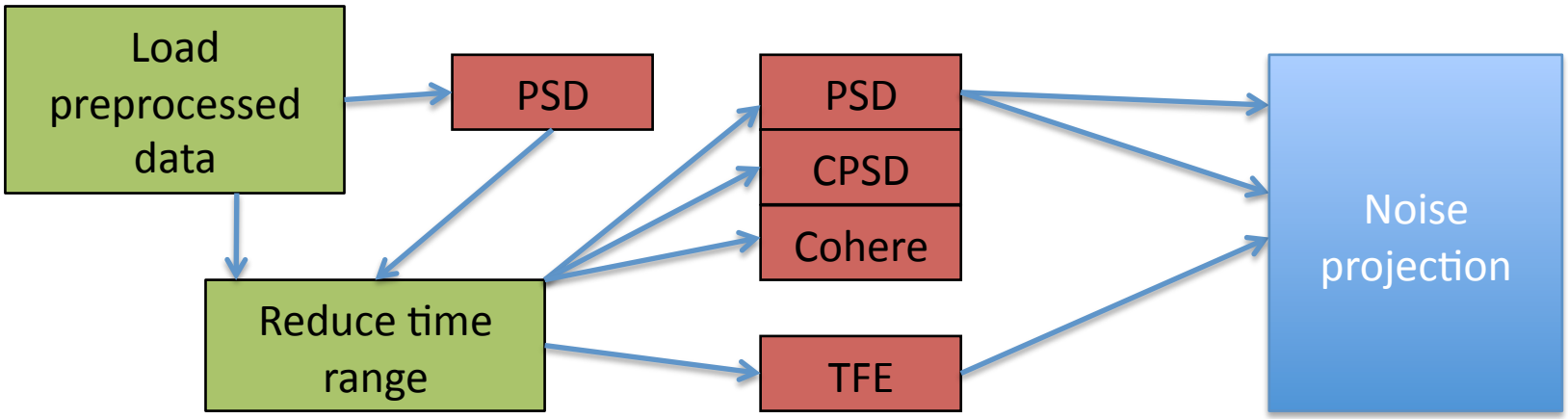


IFO/Temperature Example

Estimating the empirical transfer function

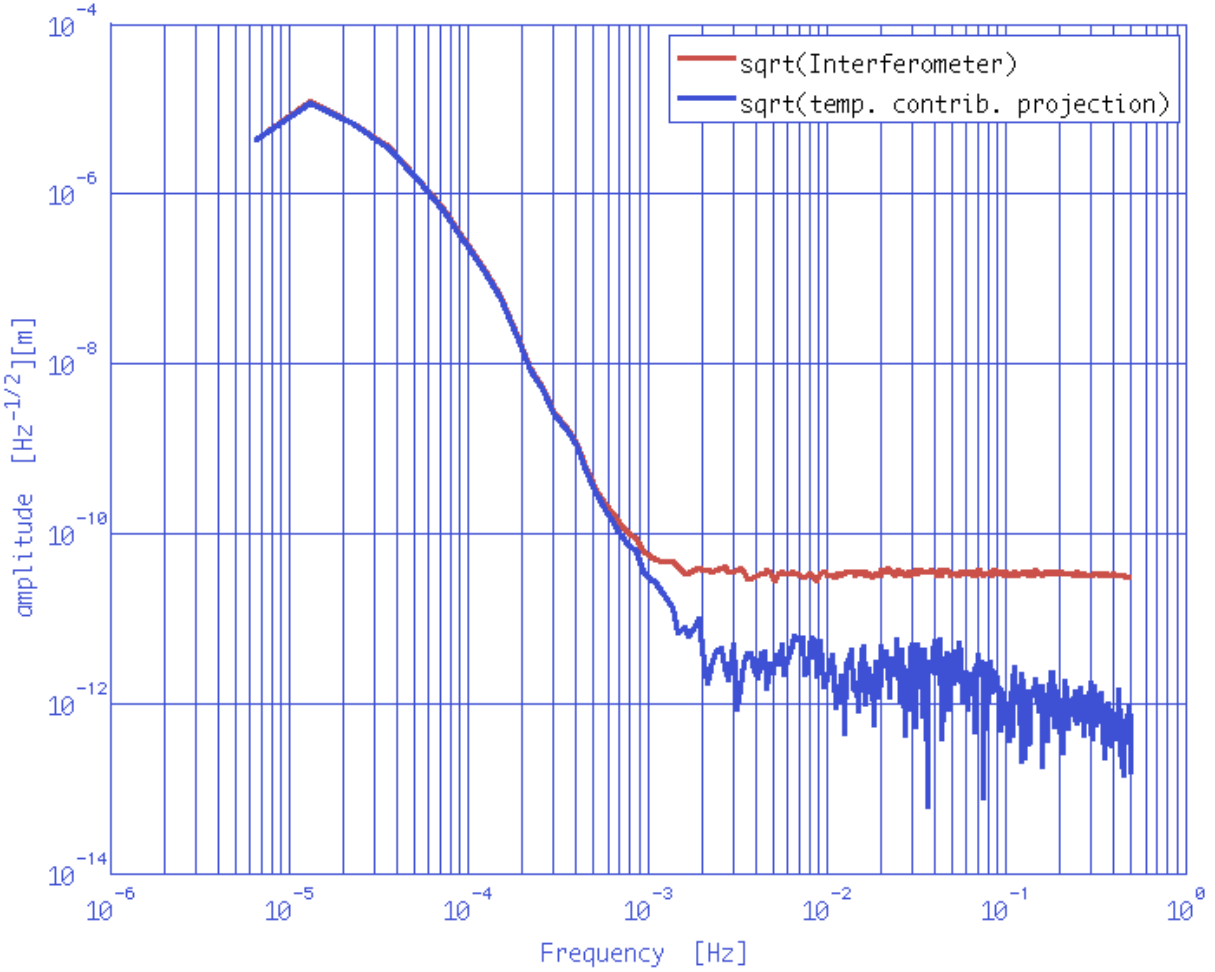
temperature -> position

- Load preprocessed data
- Evaluate PSD of *T* and *x*
- Reduce the time range
- Evaluate CPSD and cross-coherence of *T* and *x*
- Estimate the transfer function of *T* into *x*
- Perform the noise projection



IFO/Temperature Example

- We are aiming to obtain:



Changes after R2.0

- cpsd, lcpsd, cohere, lcohere, tfe,ltfe will output a *single* object

```
Cxy = cpsd(x,y,plist('nfft', 1000));
```

- All methods will include variance (if number of windows > 1) in the *procinfo* field

Topic 4: Transfer function models and digital filters

1. Transfer function models in s domain
   - 1.1. Pole zero representation
   - 1.2. Rational representation
   - 1.3. Partial fraction representation

1. Transformation between representations

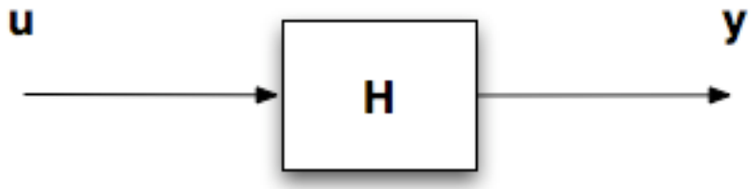
2. Modelling a system

3. Filtering data
   - 3.1. discretizing a model
   - 3.2. setting filter properties

4. IFO/Temperature example

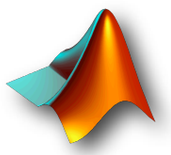
Overview

- The general scheme: input, output and a transfer function



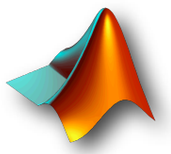
- Aim of this topic:
  - How to model the transfer function H in continuous domain, H = H(s)
  - How to discretize our model H(s) -> H(z)
  - How to filter data with H(z)
  - How to define H(z) from filter properties

Tools used here



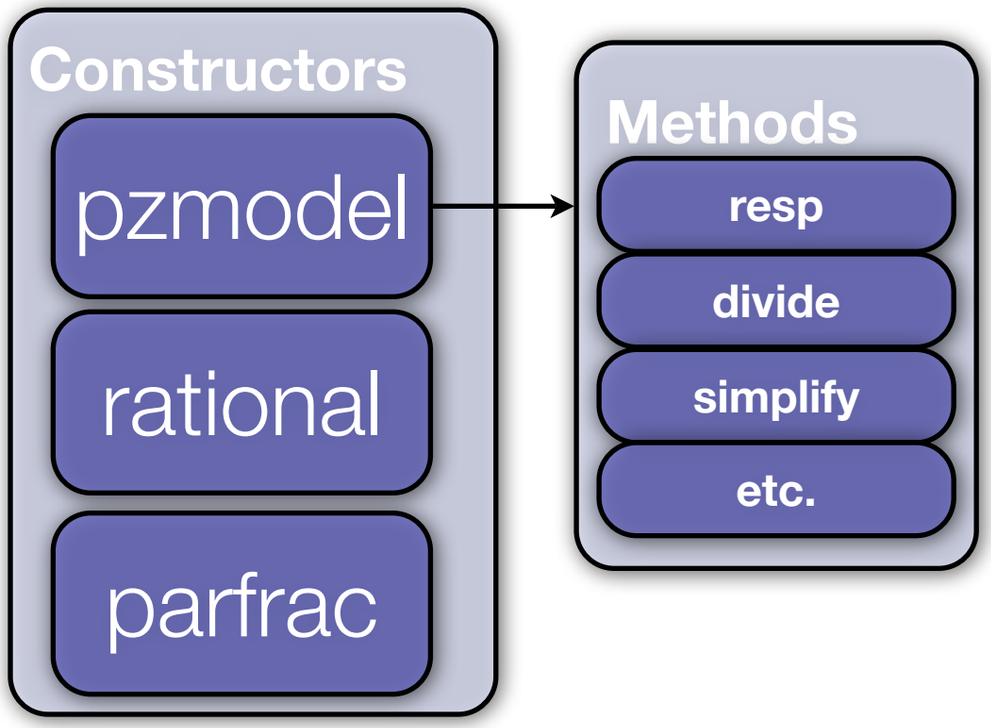
Tools used here

1. Continuous domain



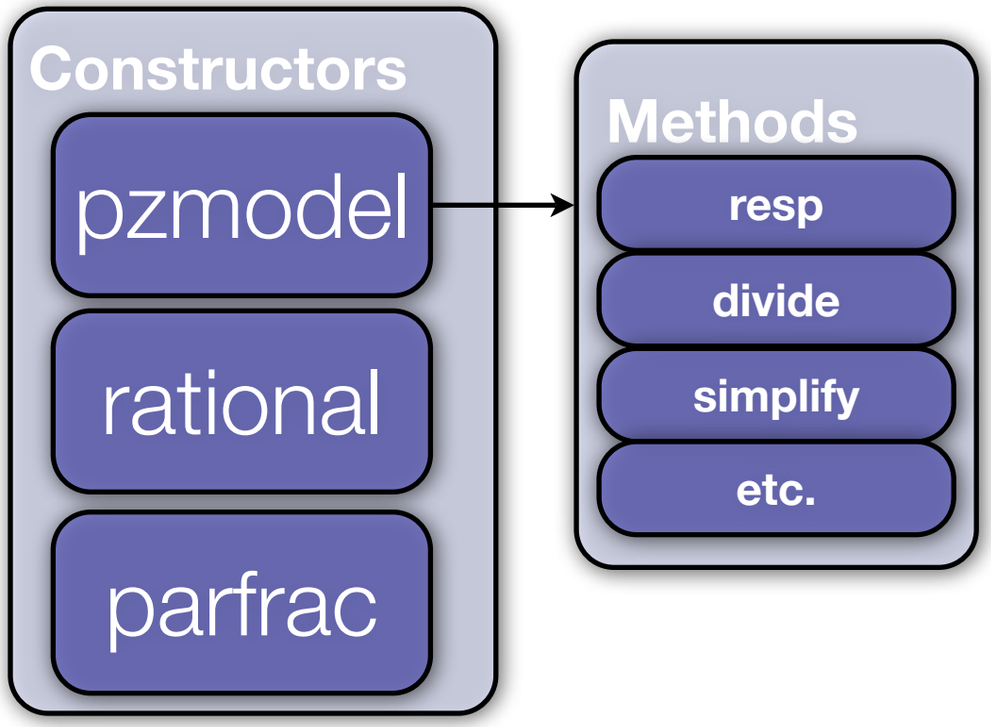
Tools used here

1. Continuous domain



Tools used here

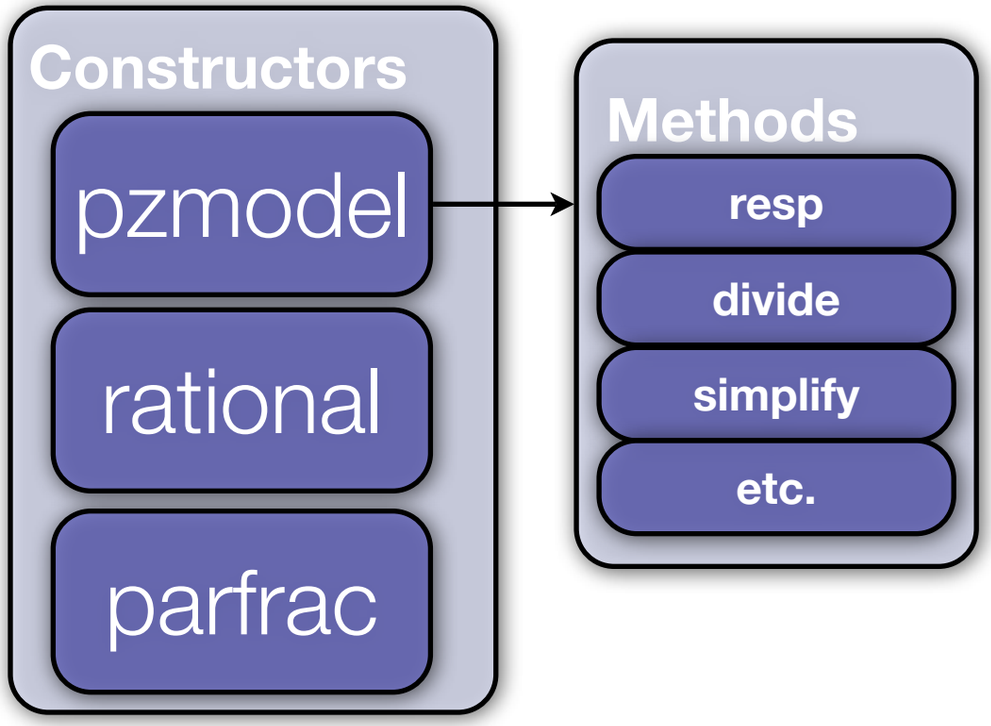
1. Continuous domain



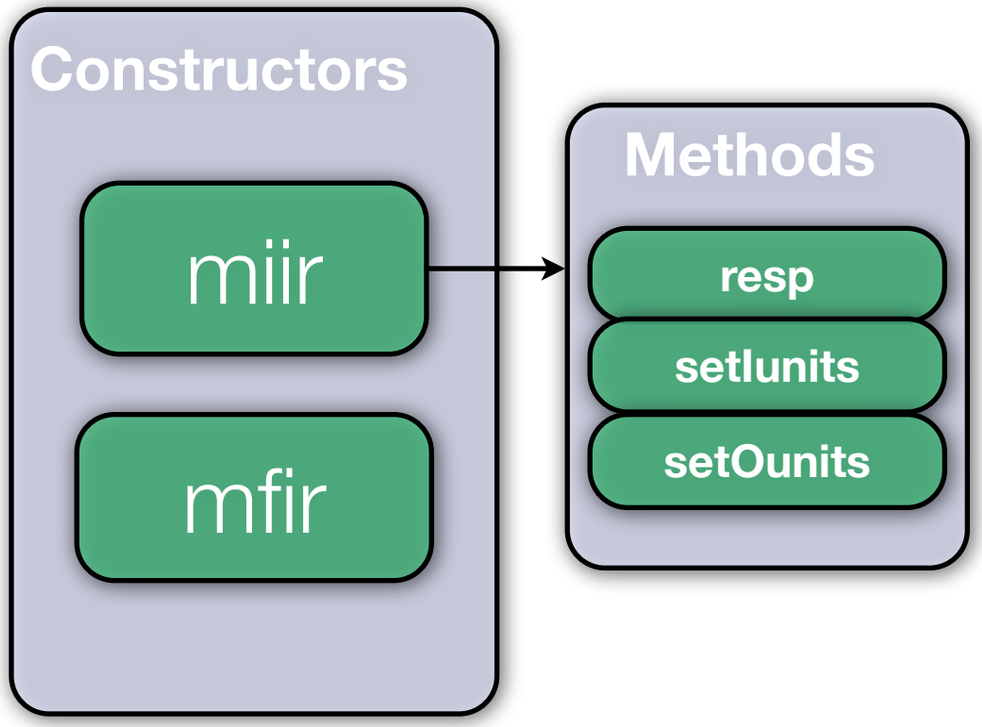
2. Discrete domain

Tools used here

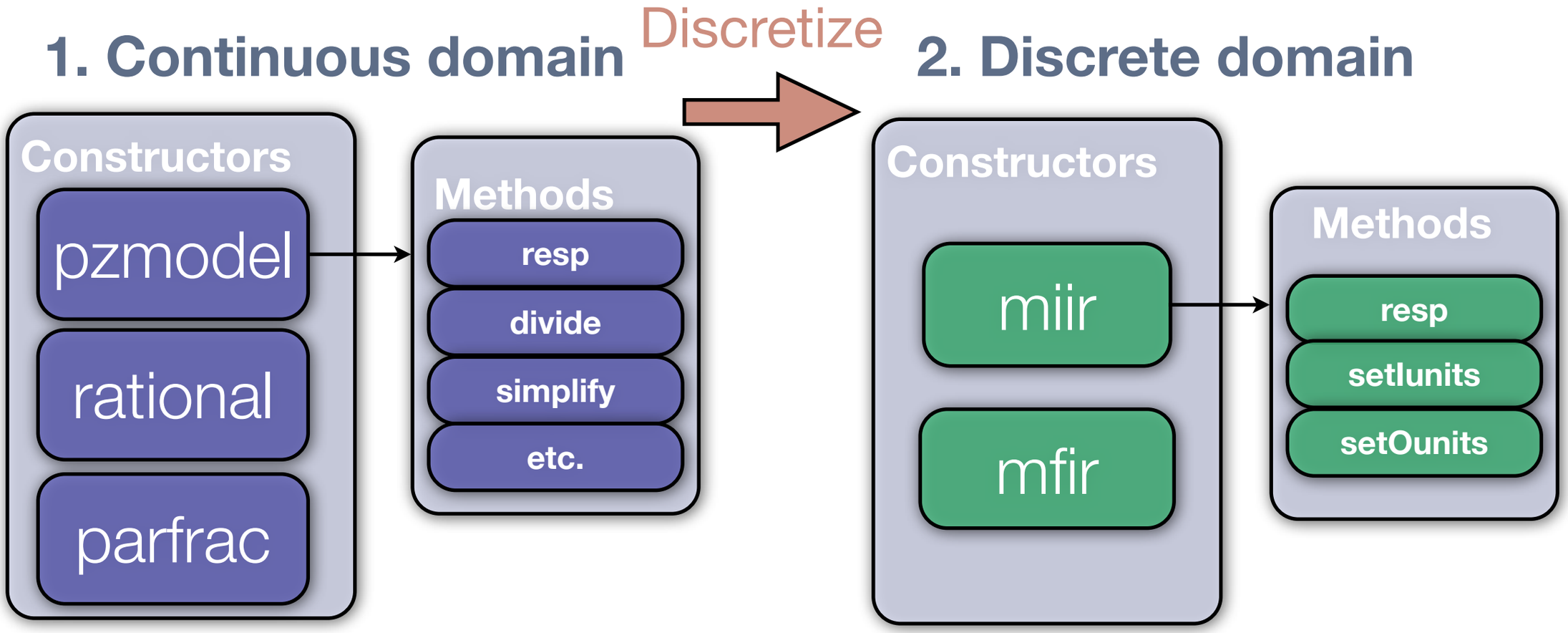
1. Continuous domain



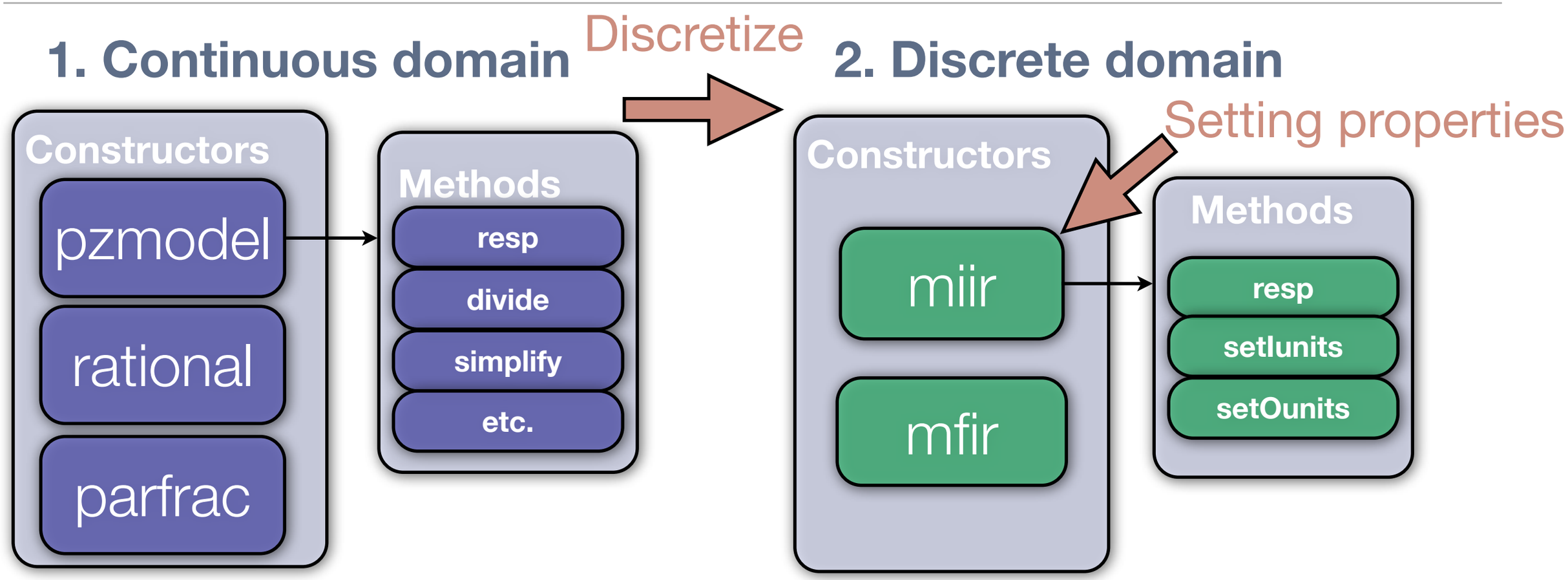
2. Discrete domain



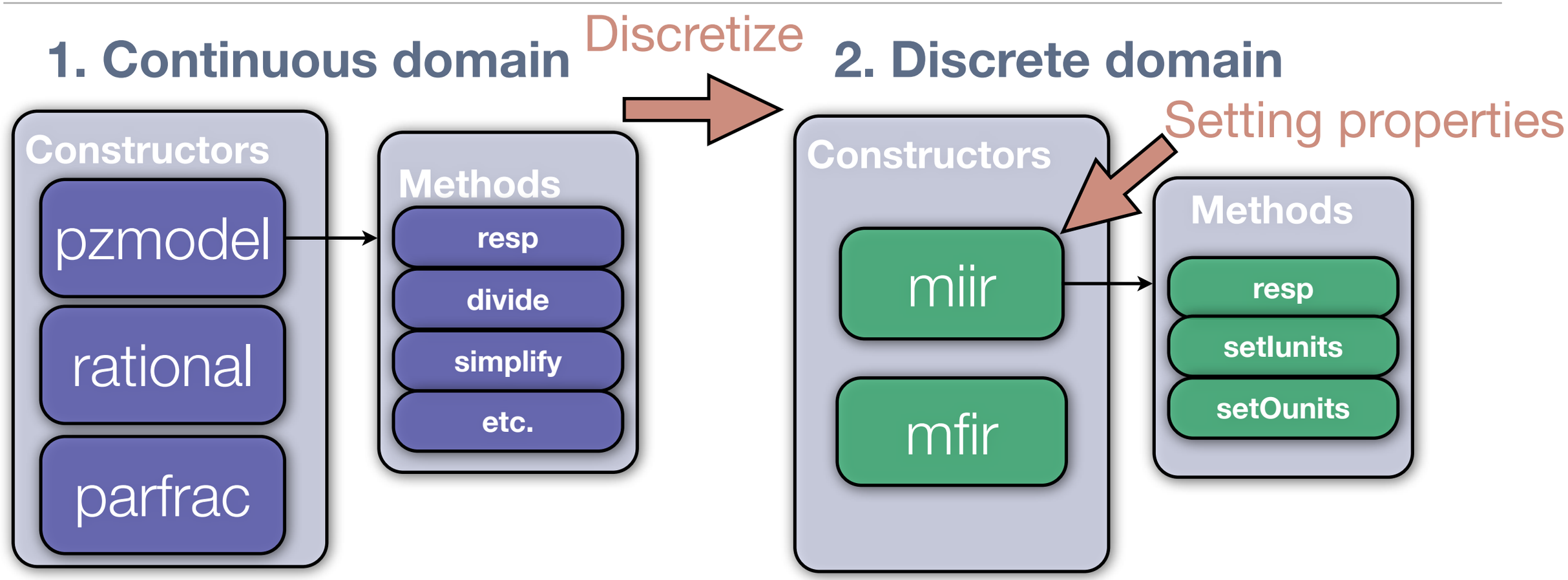
Tools used here



Tools used here

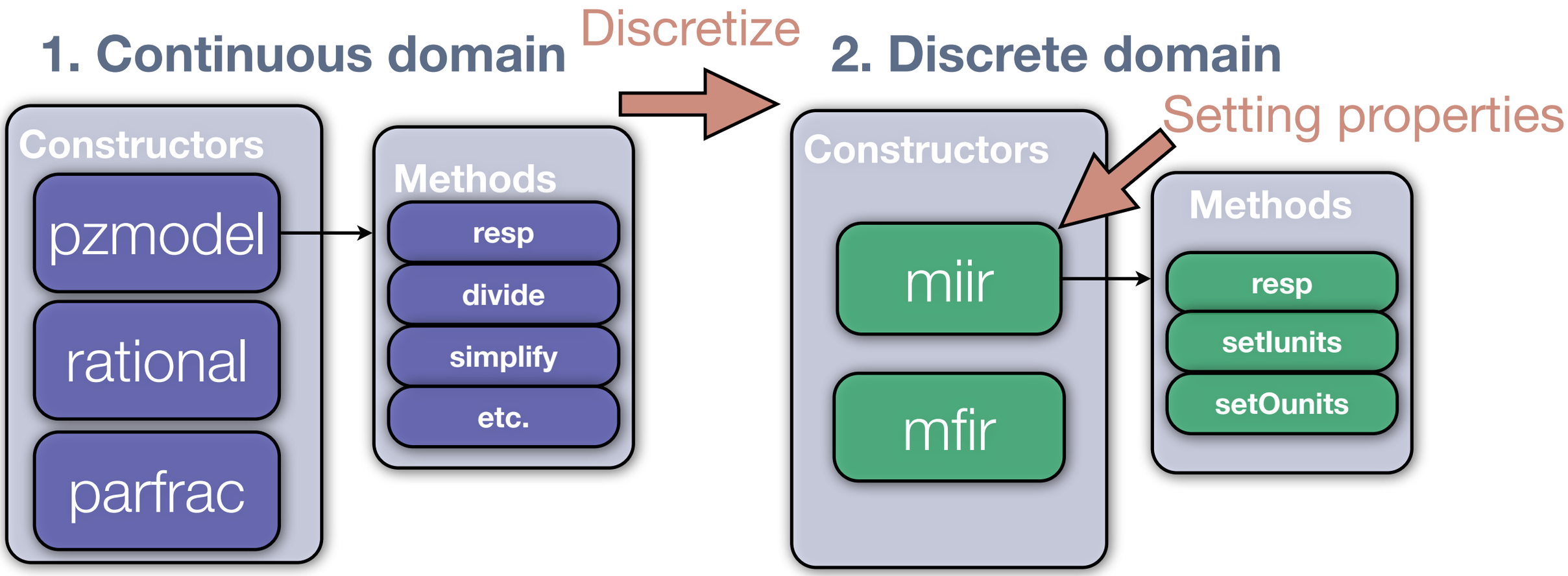


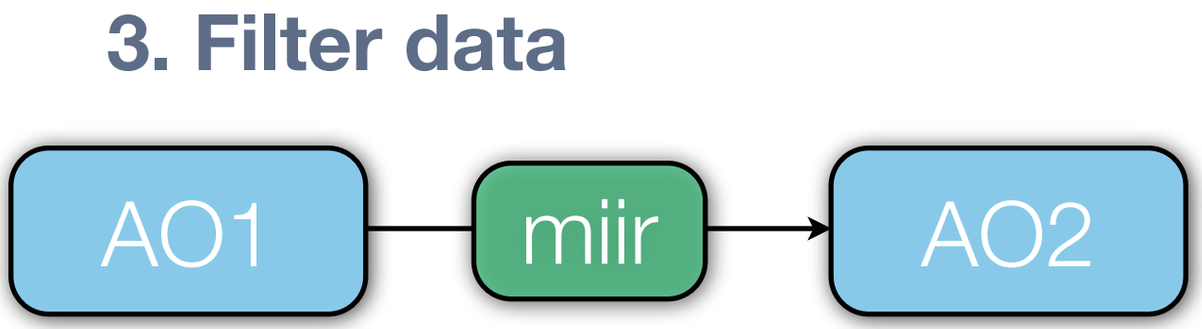
Tools used here



3. Filter data

Tools used here





1.1 Pole zero model

- A pole zero model is defined by:
  - Gain, poles, zeros, delay

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

- LTPDA constructor: PZMODEL
  - PZMODELs can be multiplied and divided
  - Delay is added or subtracted in such a case
  - Can read LISO files

About poles (and zeros) notation

About poles (and zeros) notation

- Pole pairs: (f = 1 Hz, Q)

About poles (and zeros) notation

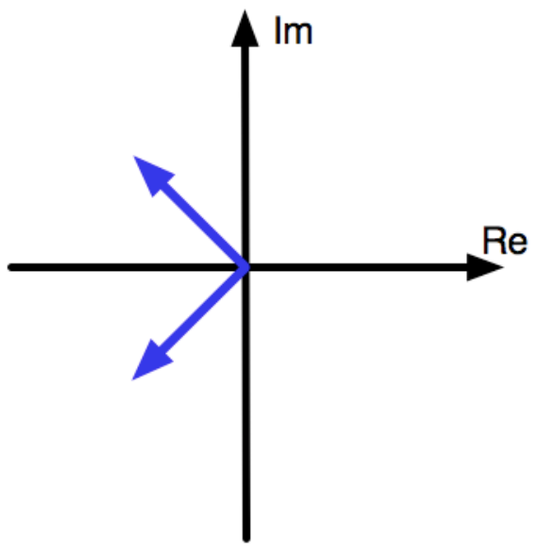
- Pole pairs: (f = 1 Hz, Q)

  Q > 0.5
  (underdamped)

About poles (and zeros) notation

- Pole pairs: (f = 1 Hz, Q)

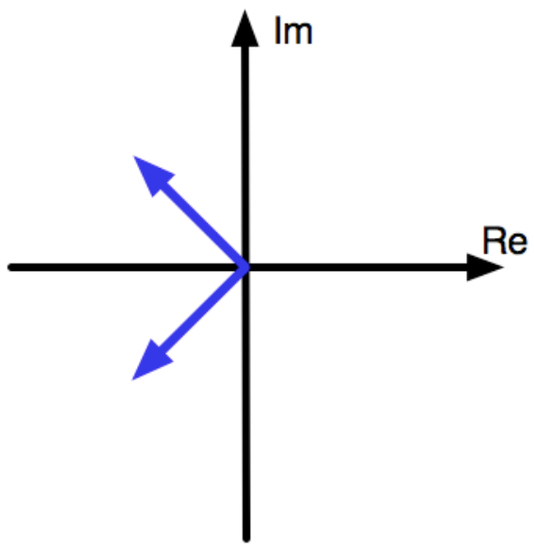
Q > 0.5
(underdamped)



About poles (and zeros) notation

- Pole pairs: (f = 1 Hz, Q)

Q > 0.5
(underdamped)

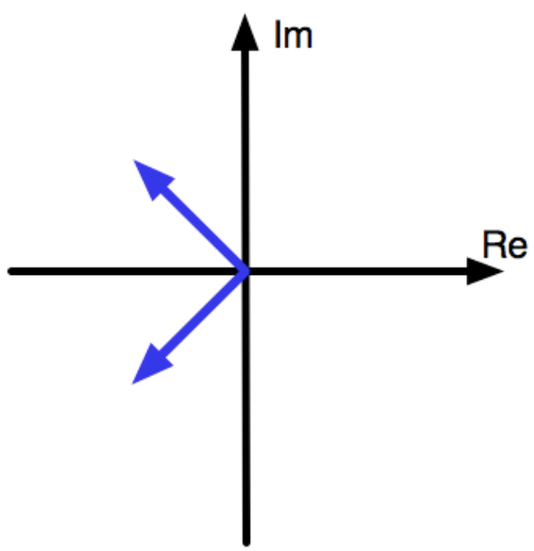


Q = 0.5
(critically damped)

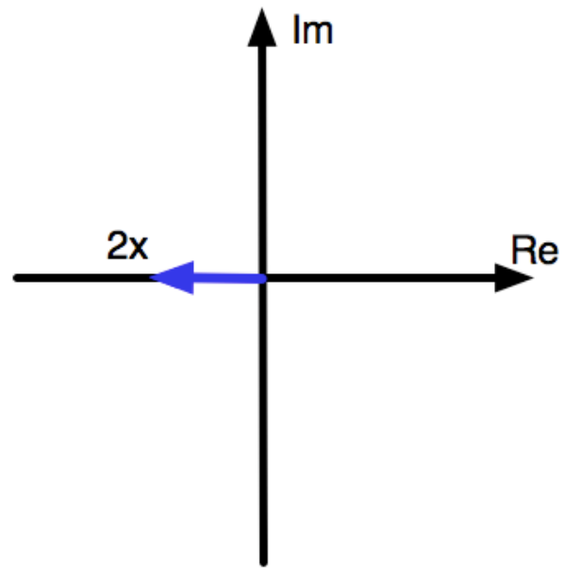
About poles (and zeros) notation

- Pole pairs: (f = 1 Hz, Q)

Q > 0.5
(underdamped)



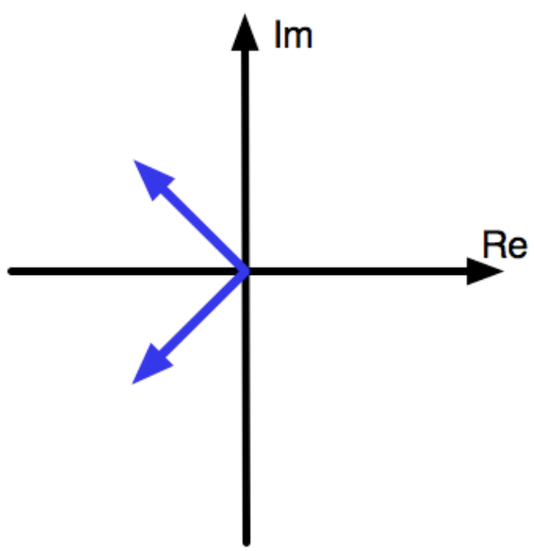
Q = 0.5
(critically damped)



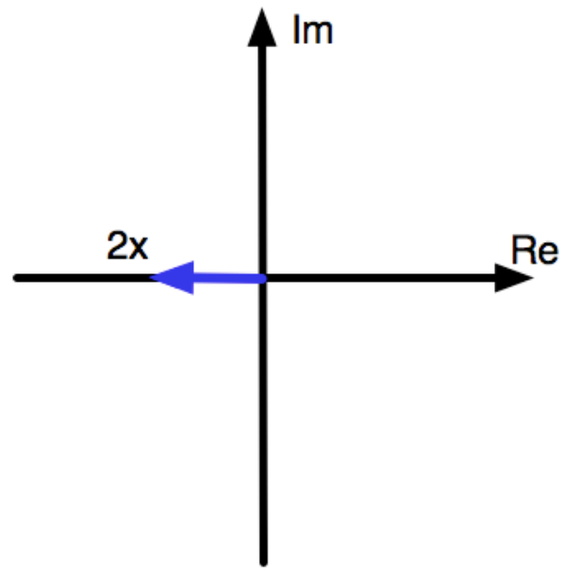
About poles (and zeros) notation

- Pole pairs: (f = 1 Hz, Q)

Q > 0.5
(underdamped)



Q = 0.5
(critically damped)

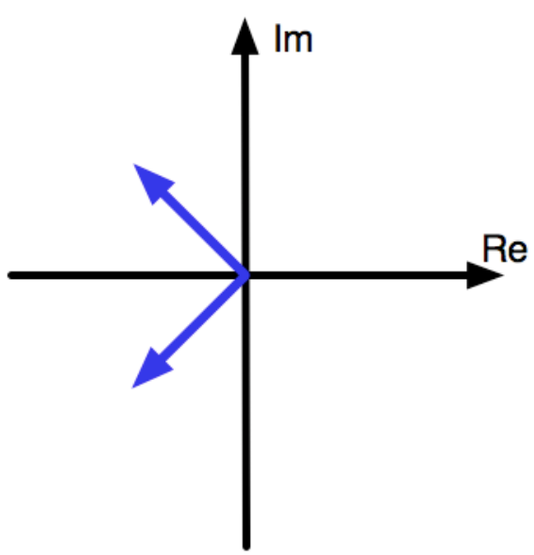


Q < 0.5
(overdamped)

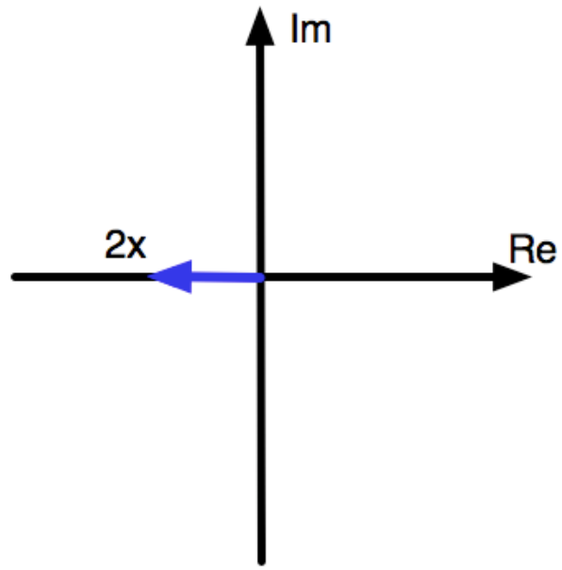
About poles (and zeros) notation

- Pole pairs: (f = 1 Hz, Q)

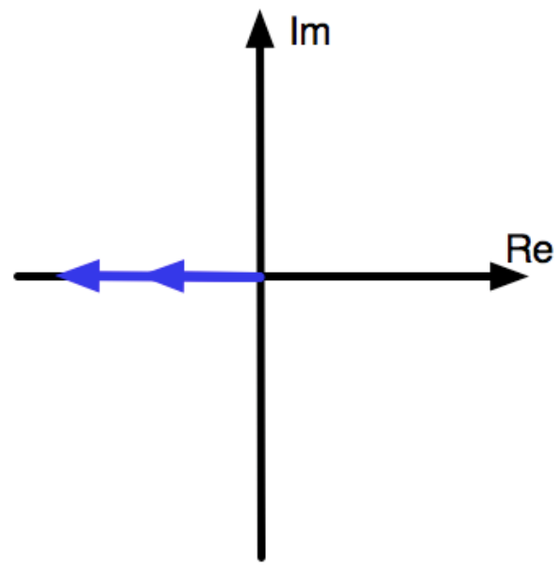
Q > 0.5
(underdamped)



Q = 0.5
(critically damped)



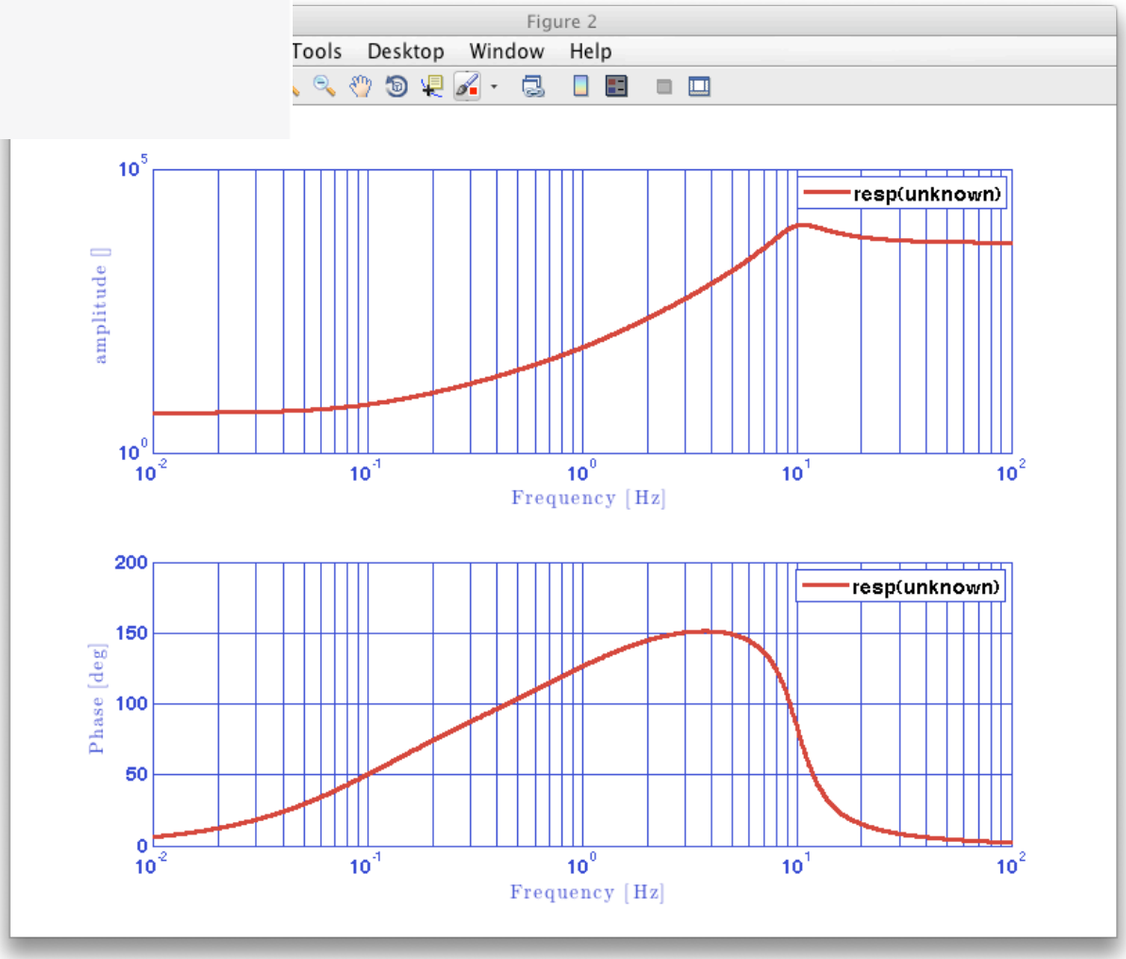
Q < 0.5
(overdamped)



1.1 Pole zero model

- Working example: Compute pole zero response
  - Topic 4 > Create transfer func... > Create pole zero model

Key	Value
GAIN	5
POLES	(f = 1 Hz, Q = 2)
ZEROS	(f = 1 Hz), (f = 0.1 Hz)



1.2 Rational model

- A rational model is defined by:
  - Num. and den. coefficients

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

- LTPDA constructor: RATIONAL
  - RATIONALs can NOT be multiplied and divided

- Working example: Compute rational response
  - Topic 4 > Create transfer func... > Create rational model

1.3 Partial fraction model

- A partial fraction model is defined by:
  - Poles, residues and direct terms

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

- LTPDA constructor: PARFRAC
  - PARFRACs can NOT be multiplied and divided

- Working example: Compute par. frac. response
  - Topic 4 > Create transfer func... > Create par. frac. model

1.4 Transforming models

- Only rational <-> pzmodel translation is implemented in v2.0
  - Works inserting object into constructor, e.g. rat = rational(pzm)

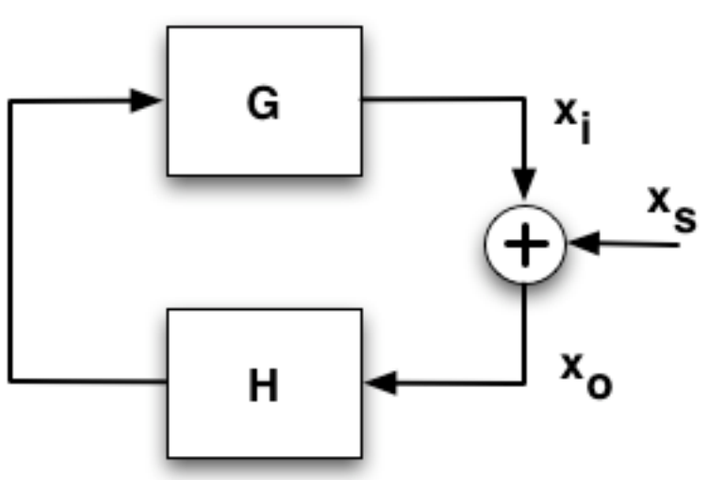
	Pole/Zero	Rational	Partial Fraction
Pole/Zero		✓	✗
Rational	✓		✗
Partial Fraction	✗	✗	

- Working example: pzmodel -> rational -> pzmodel
  - Topic 4 > Transforming models between representations

2. Modelling a system

- Modelling a closed loop system with pzmodels
  - Basic pzmodel operations

- Our system:

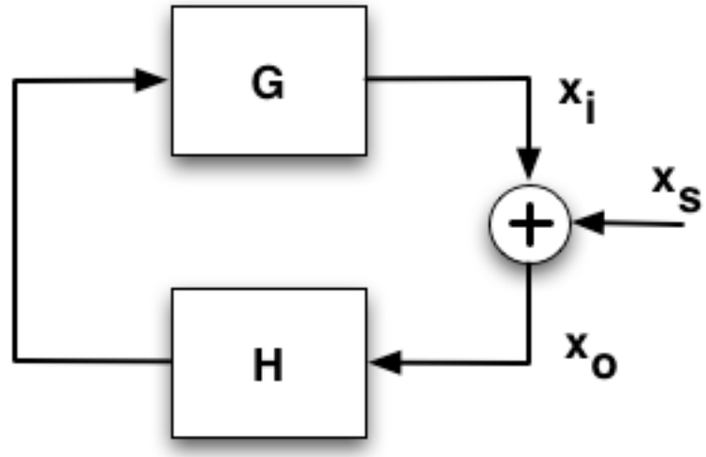


$$\frac{x_o}{x_s} = CLG = \frac{1}{1 - OLG} = \frac{1}{1 - H \cdot G}$$

- Stating the problem
  - Assuming OLG and H known, determine H and CLG

2. Modelling a system

- Step-by-step



1. G = OLG/H
   (G is a pzmodel)
2. Operate on G: setName, simplify ...
3. CLG = 1/(1-OLG)
   (CLG is NOT a pzmodel)
4. Repeat loading H with delay

$$\frac{x_o}{x_s} = \mathrm{CLG} = \frac{1}{1-\mathrm{OLG}} = \frac{1}{1-\mathrm{H}\cdot\mathrm{G}}$$

- Working example: Modelling a system
  - Topic 4 > Modelling a system

3. Entering the discrete domain

- The LTPDA toolbox allows you to build digital filters...
  - Discretizing your model
    - Example: find the filters for H,G, OLG in our closed loop
  - Defining filter properties
    - Example: Design a bandpass filter to evaluate power spectrum in a bandwidth

- Filter constructors in LTPDA
  - MIIR (IIR filters)

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

  - MFIR (FIR filters)

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

3.1 By discretizing a transfer function

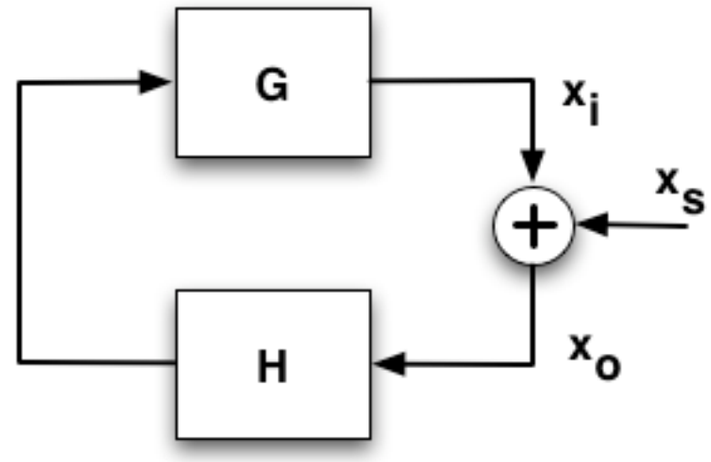
- Syntax: insert pzmodel into constructor

```
Gd = miir(G,plist('fs',10));
```

Constructor

pzmodel obj.

filter obj.

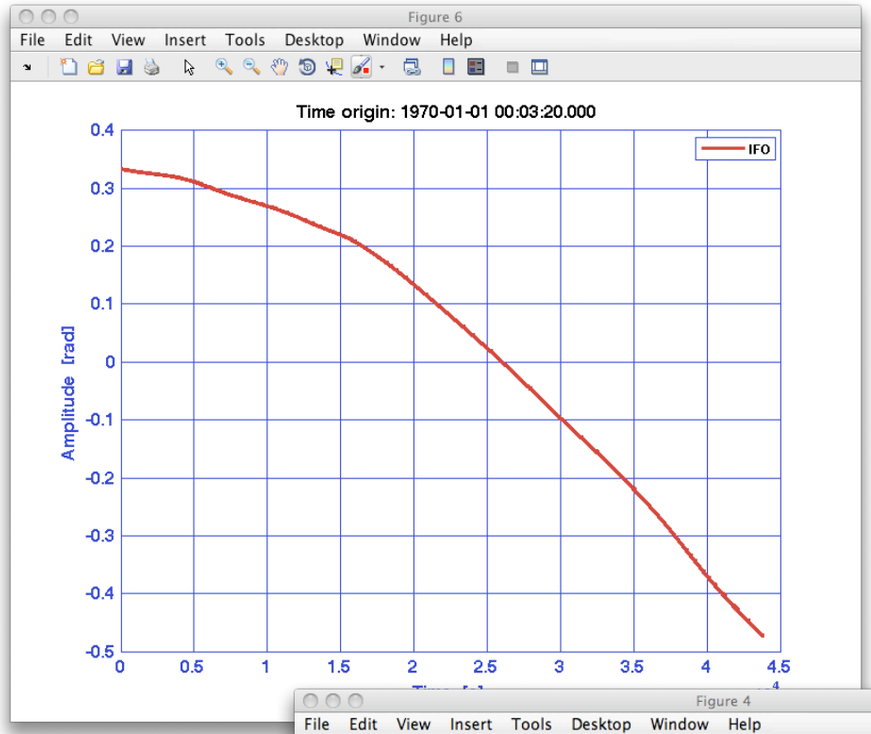


- Step-by-step
  1. Discretize G,H,OLG
  2. Compare continuous and digital response
  3. Get filter coefficients
  4. Delay is NOT used in the discretization!!

- Working example: Get filters for closed loop pzmodels
  - Topic 4 > How to filter data > By discretizing...

3.2 By defining filter properties

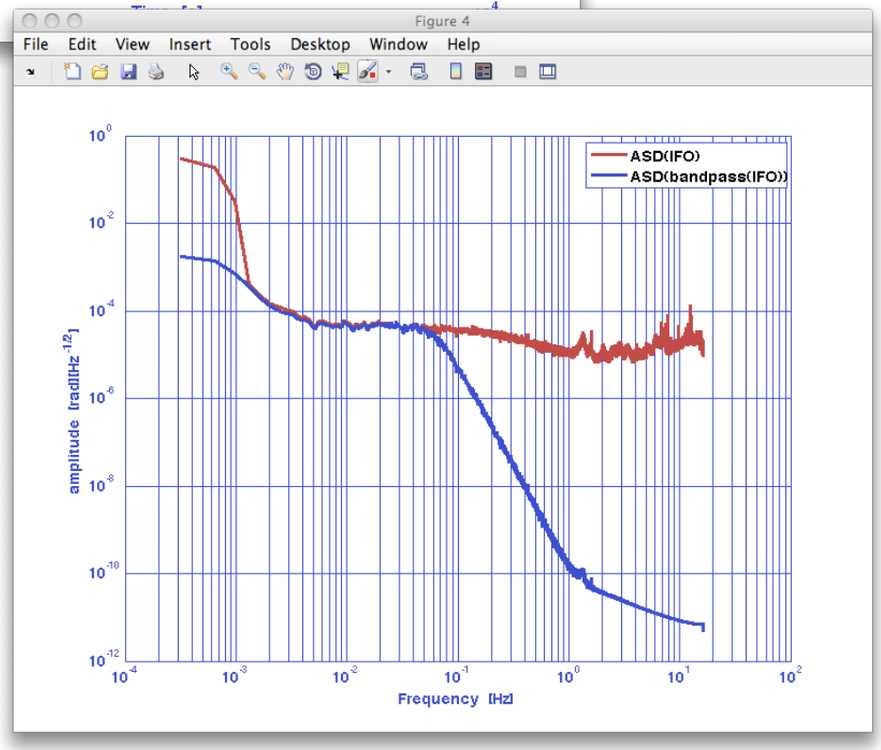
- Design a bandpass filter
  - Standard pre-processing step used in LTP lab
  - Alternative to detrending



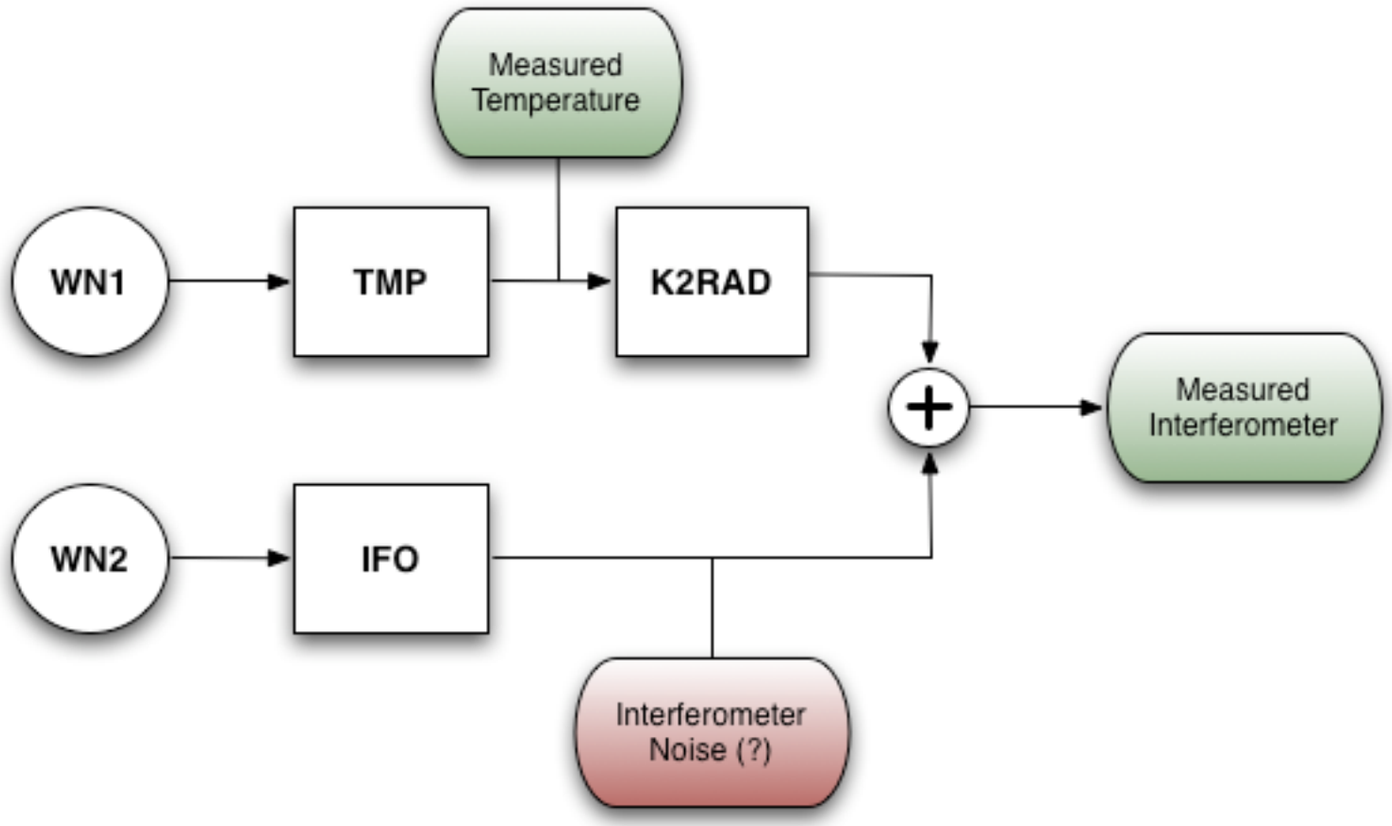
- Syntax:

```
Gd = miir(plist('fs',32.47, 'order',...));
```

- Working example: Band pass



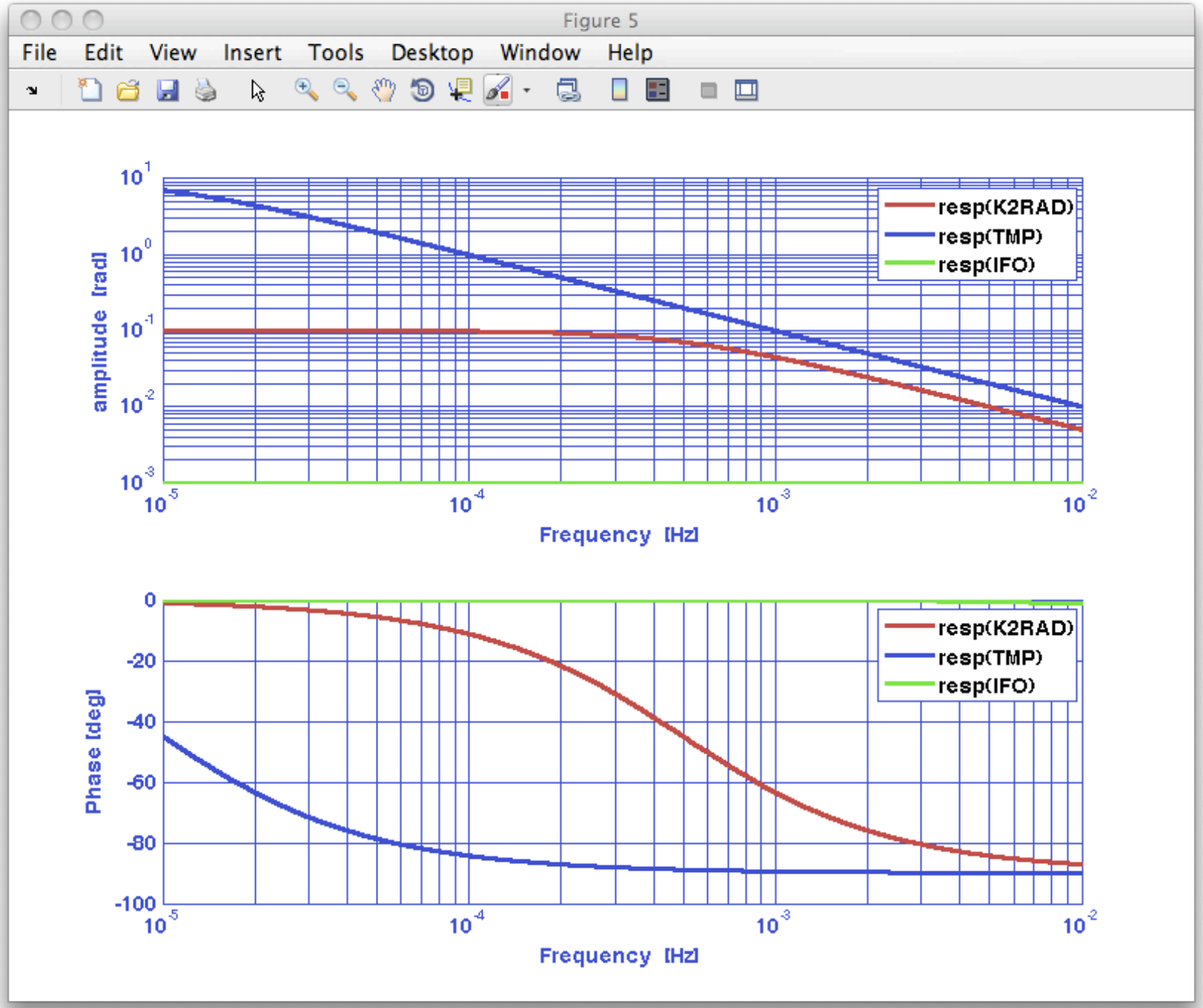
4. IFO/Temperature Example



- Aim: perform the analysis with a toy model
  - Create transfer function models: TMP, IFO,K2RAD
  - Discretize
  - Filter (white noise) data
  - Estimate transfer function (topic 3) with synthetic data

4. IFO/Temperature Example

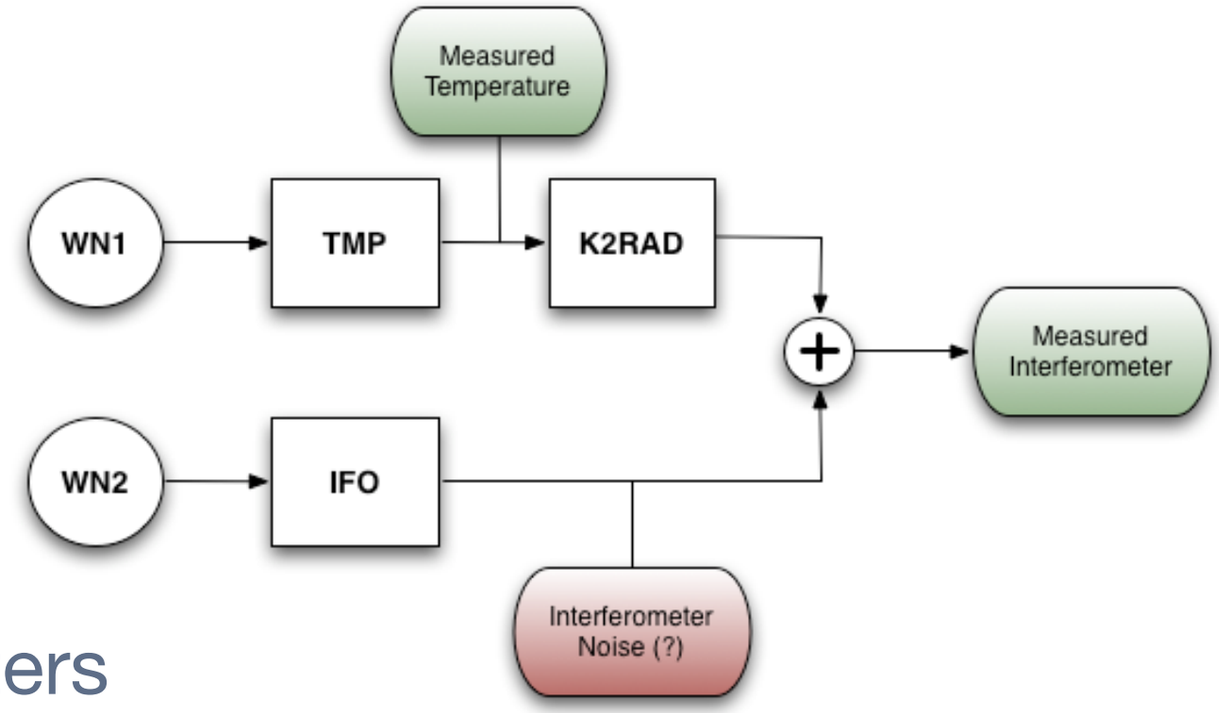
- Our toy models

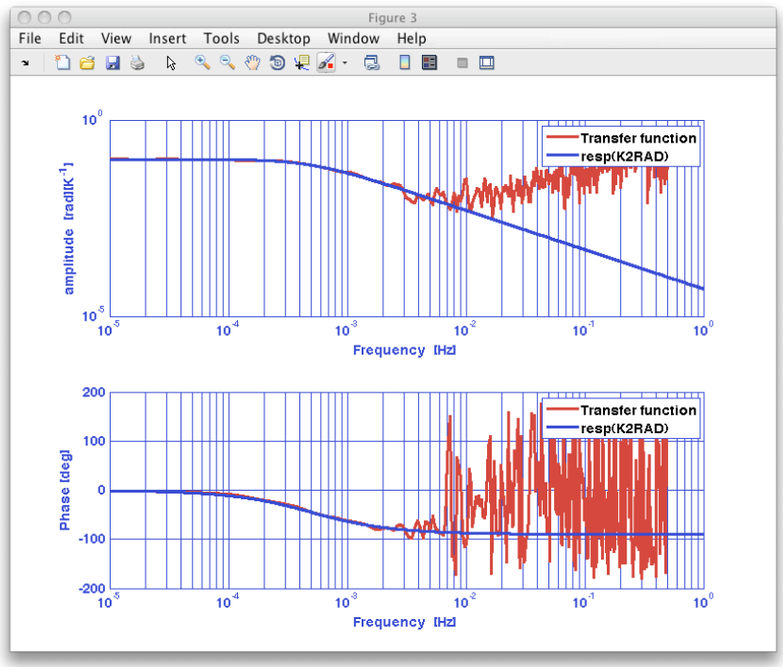


4. IFO/Temperature Example

- Step-by-step

1. Generate models: TMP, IFO, K2RAD
2. Discretize
3. Build two white noise time series
4. Filter with the digital filters
5. Estimate transfer function
6. Project temperature noise

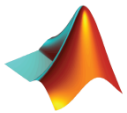




- Working example: IFO/Temperature

LTPDA Training Session Topic 5

Luigi Ferraioli



Data Fitting in LTPDA

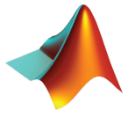
Function	Description
curvefit	Non-linear least square fit to data
lisovfit	LISO to fit a pole/zero model to the input frequency-series
lscov	Overloads lscov function of MATLAB for Analysis Objects
polyfit	Overloads polyfit function of MATLAB for Analysis Objects
sDomainFit	Fit a partial fraction model to frequency series data
zDomainFit	Fit a partial fraction z-domain model to frequency series data

Correlated functions

Function	Description
noisegen1D	Generate colored noise with given power spectral density
noisegen2D	Generate cross correlated colored noise with given cross spectral density
whiten1D	Noise whitening tool
whiten2D	Noise whitening tool for two cross-correlated time series

Scheduled Changes for the next release

Function	Status	Actions
curvefit		
lisovfit		
lscov		
polyfit		
sDomainFit		• Fit Objects with a delay • Fit correctly real objects
zDomainFit		
noisegenND		Multichannel noise generator
whitenND		Multichannel noise whitening tool

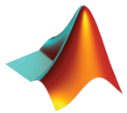


Inspected Functions

Function	Description
curvefit	Non-linear least square fit to data
polyfit	Overloads polyfit function of MATLAB for Analysis Objects
zDomainFit	Fit a partial fraction z-domain model to frequency series data

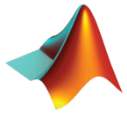
Correlated functions

Function	Description
noisegen1D	Generate colored noise with given power spectral density



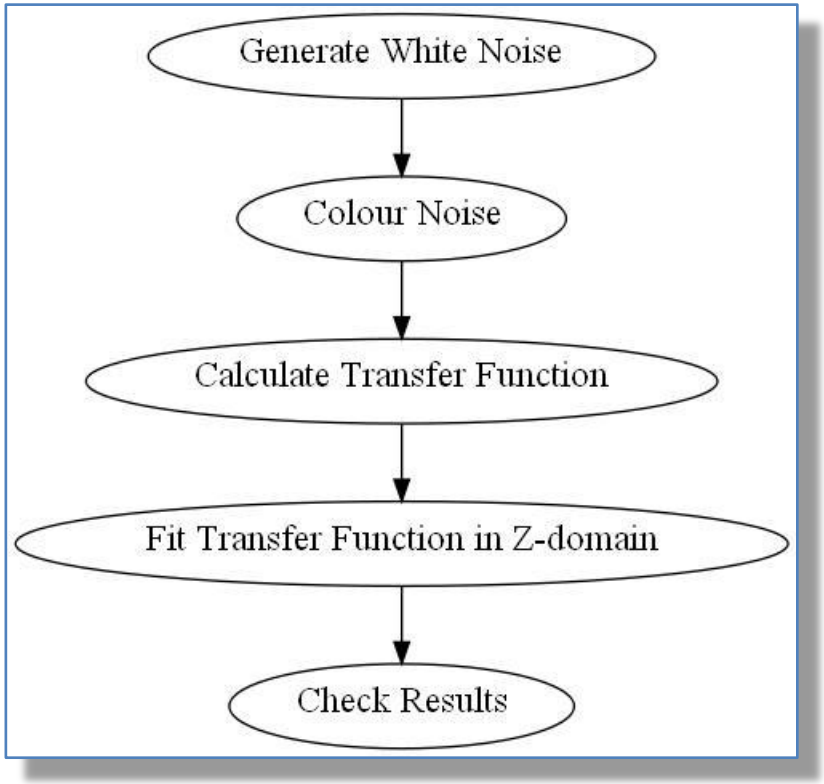
Topic 5 – Exercise 1

- Go to help section
  - LTPDA Toolbox
    - LTPDA Training Session 1
      - Topic 5 - Model fitting
      - Open the page of the first exercise
        - » System identification in z-domain
      - Open a new editor window
        - » In Matlab command window type » edit



Topic 5 – Exercise 1
System Identification in Z-domain

Relevant functions
zDomainFit
It is used to run a system identification in z-domain on a transfer function calculated from data. Model order is automatically selected on the basis of input tolerance settings



Topic 5 – Exercise 1

Generate random noise

```
a = ao(plist('tsfcn', 'randn(size(t))', 'fs', 1, 'nsecs', 10000,'yunits','m'));
```

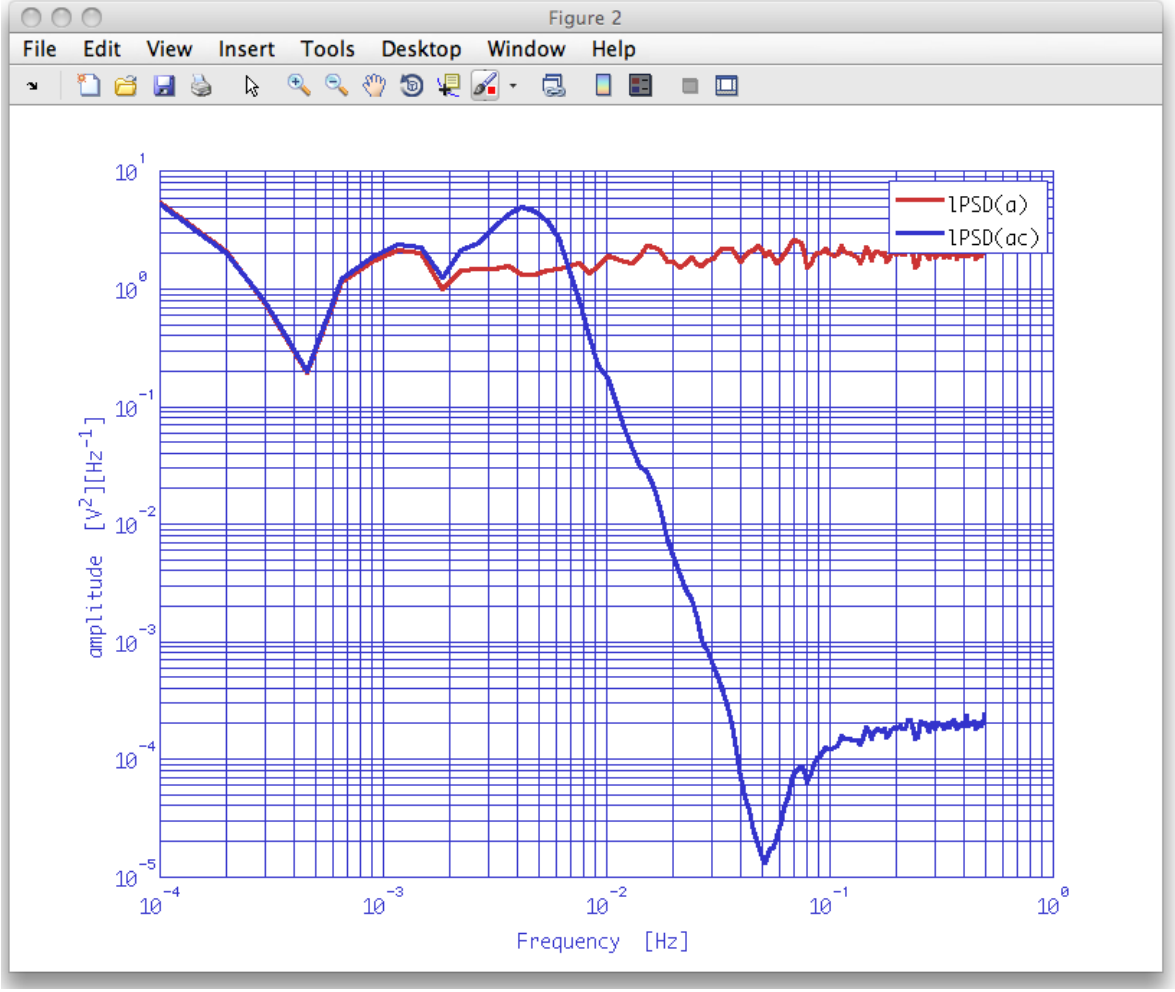
- Build a pzmodel
- Convert to a miir
- Color white noise

```
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;
```

Make PSD and inspect results

```
axx  = lpsd(a);
acxx = lpsd(ac);
iplot(axx,acxx)
```

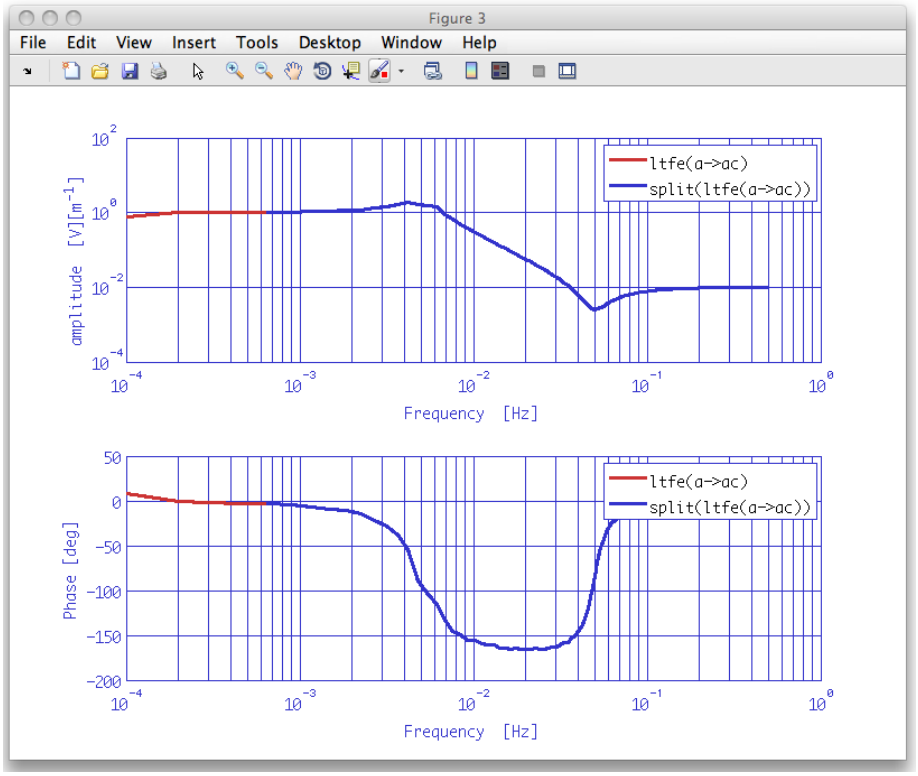
Topic 5 – Exercise 1



Topic 5 – Exercise 1

Calculate transfer function and cut away first bins

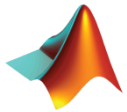
```
tf = ltfe(a,ac);
tf = tf.index(1,2);
tfsp = split(tf,plist('frequencies', [5e-4 5e-1]));

iplot(tf,tfsp)
```



Topic 5 – Exercise 1

Fit Transfer Function

```
% Set up the parameters
plfit = plist('FS',1,...     % Sampling frequency for the model filters
  'AutoSearch','on',...      % Automatically search for a good model
  'StartPolesOpt','c1',...   % Define the properties of the starting poles - complex
  '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');
```



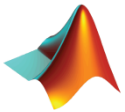
Topic 5 – Exercise 1

Fit Transfer Function

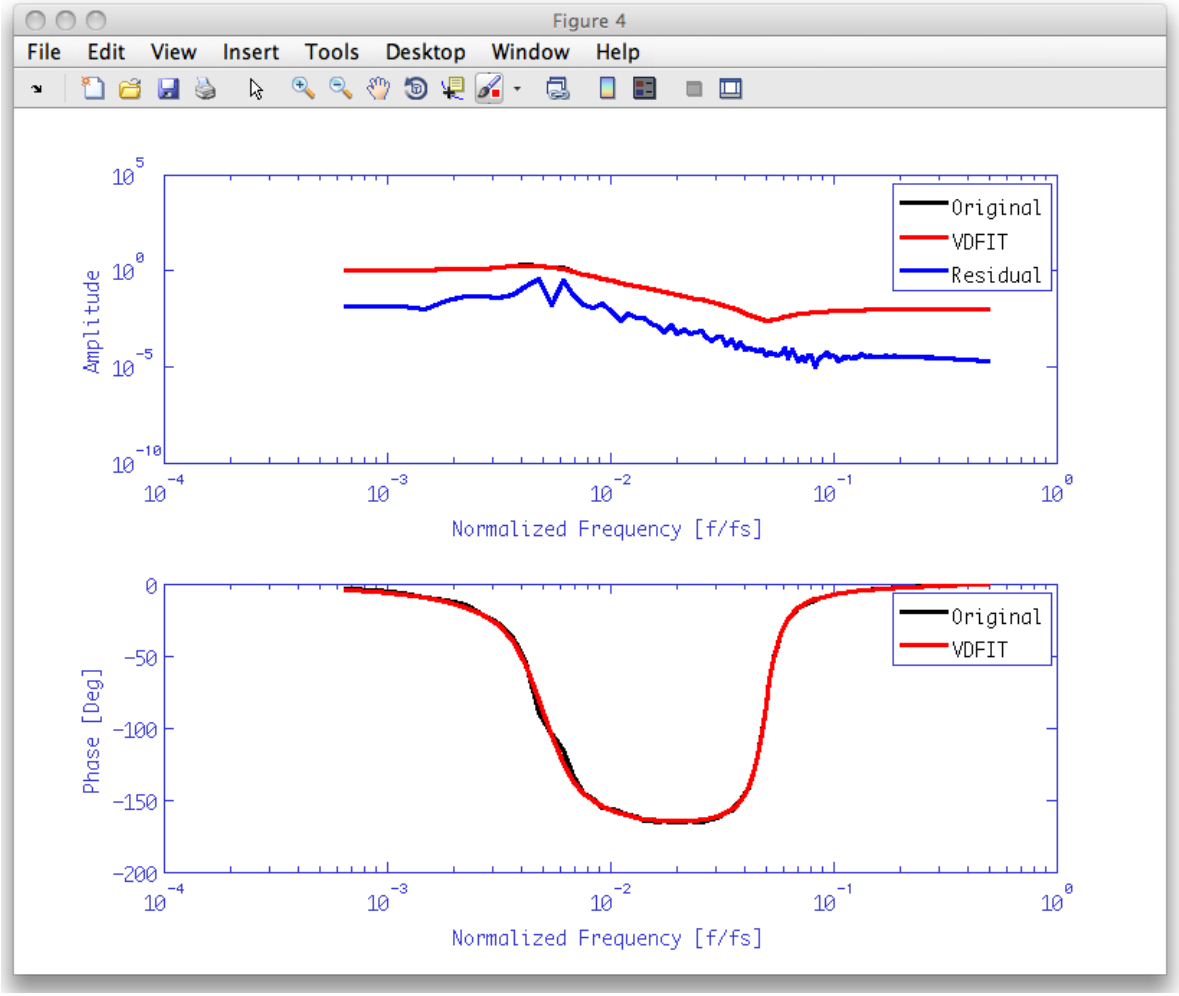
Check if the log-scale difference between data and fit residuals is larger than the assigned value

Check if the step-by-step root mean square error variation is lower than 10^{-d} (d is the assigned value)

```
% Set up the parameters
plfit = plist('FS',1,...     % Sampling frequency for the model filters
  'AutoSearch','on',...      % Automatically search for a good model
  'StartPolesOpt','c1',...   % Define the properties of the starting poles - complex
  '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');
```



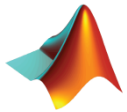
Topic 5 – Exercise 1



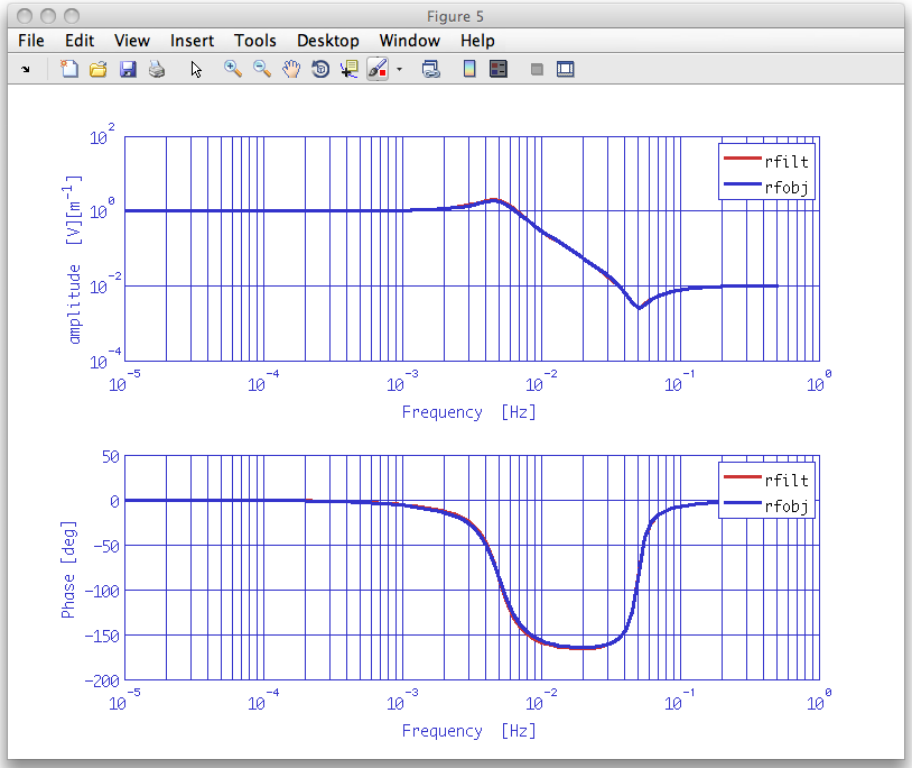
Topic 5 – Exercise 1

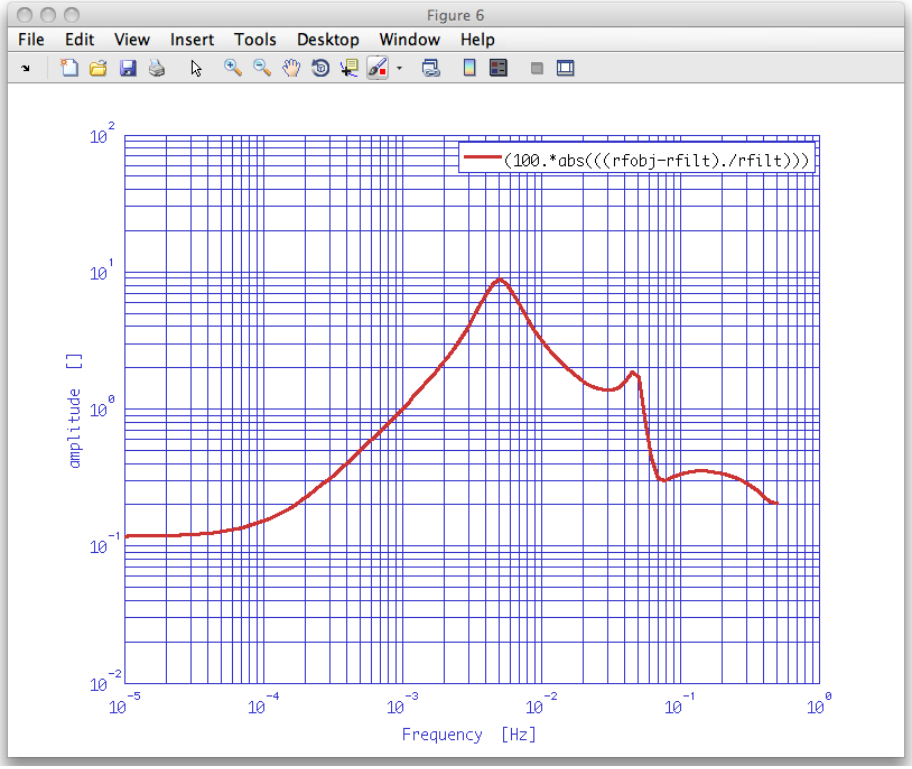
Calculate filters response and check fit results

```
% 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,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]))
```



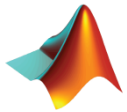
Topic 5 – Exercise 1





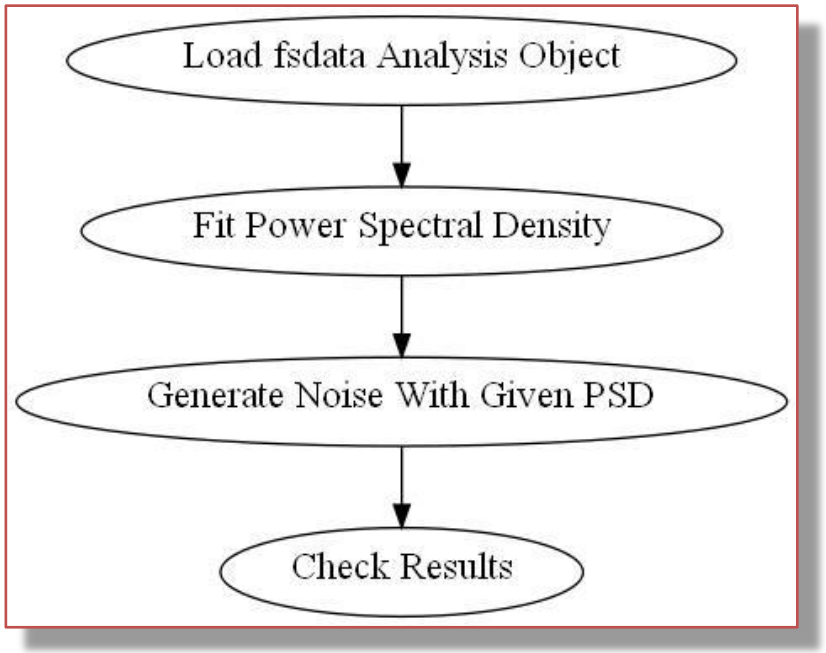
Topic 5 – Exercise 2

- Go to help section
  - LTPDA Toolbox
    - LTPDA Training Session 1
      - Topic 5 - Model fitting
      - Open the page of the second exercise
        - » Generation of noise with given psd
      - Open a new editor window
        - » In Matlab command window type » edit



Topic 5 – Exercise 2
Generation of Noise with Given PSD

Relevant functions	
zDomainFit	It is used to get a smooth model for the calculated psd.
noisegen1D	Generate noise with the given power spectral density



Topic 5 – Exercise 2

Load test noise

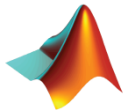
```
tn = ao(plist('filename', 'topic5/T5_Ex03_TestNoise.xml'));
tn.setName;
```

Calculate power spectral density

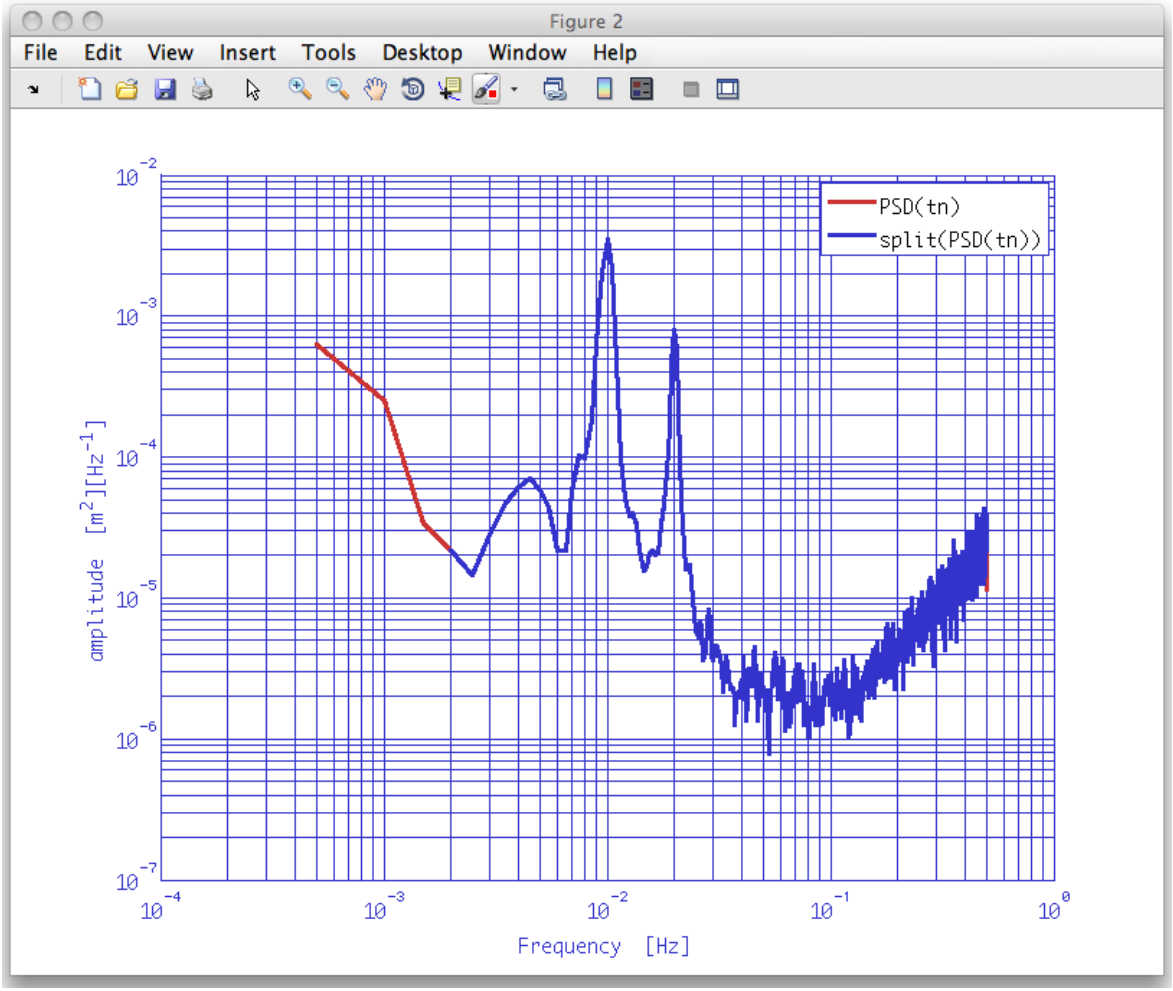
```
tnxx = tn.psd(plist('Nfft',2000));
```

Cut away first bins and plot

```
tnxxr = split(tnxx,plist('frequencies', [2e-3 5e-1]));
iplot(tnxx,tnxxr)
```



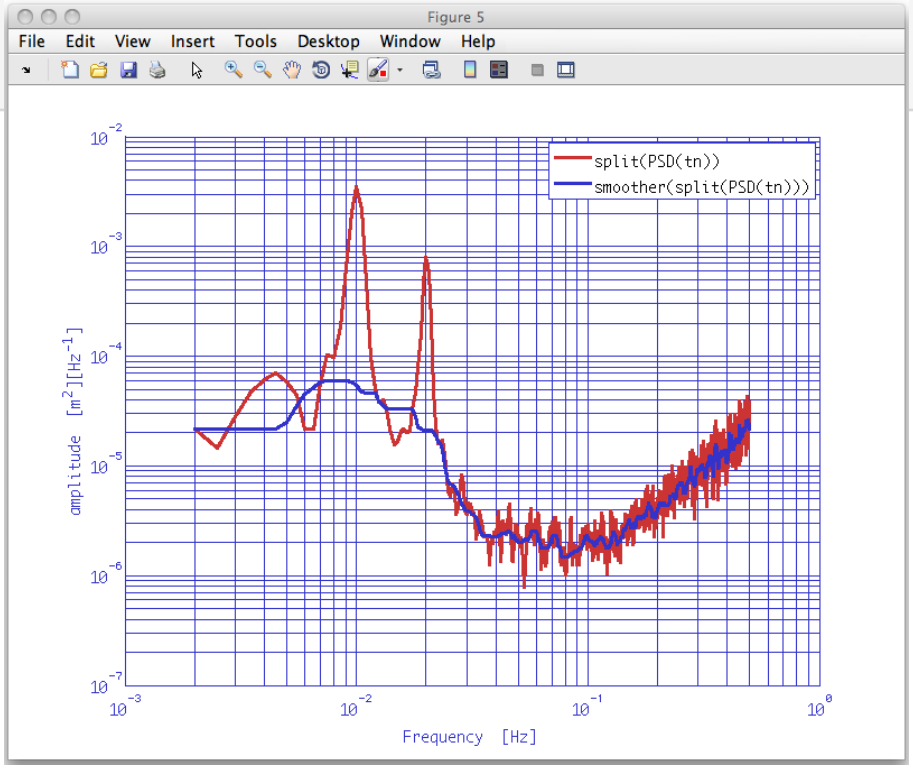
Topic 5 – Exercise 2



Topic 5 – Exercise 2

We smooth 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

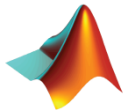
```
stnxx = smoother(tnxxr);
iplot(tnxxr, stnxx)
wgh = 1./abs(stnxx);
```



Topic 5 – Exercise 2

Fit PSD

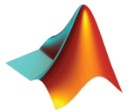
```
plfit = plist('FS',1,...
              'AutoSearch','on',...
              'StartPolesOpt','c1',...
              'maxiter',50,...
              'minorder',10,...
              'maxorder',45,...
              'weights',wgh,... % assign externally calculated weights
              'ResLogDiff',[],...
              'ResFlat',0.77,...
              'RMSE',5,...
              'Plot','on',...
              'ForceStability','off',...
              'CheckProgress','off');

% Do the fit
[param,fmod] = zDomainFit(tnxxr,plfit);
```



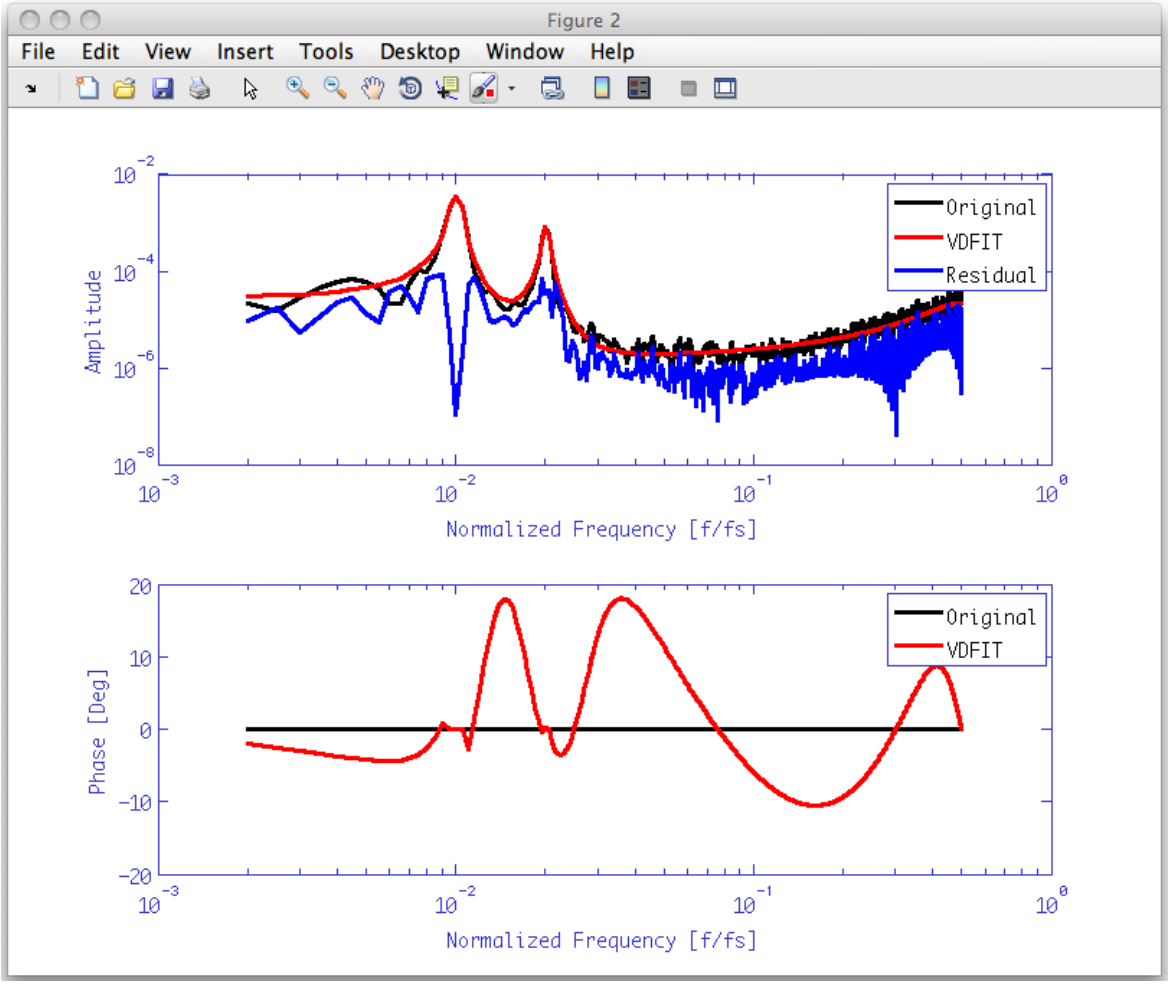
Topic 5 – Exercise 2

Check if residuals spectral flatness is larger than the assigned value

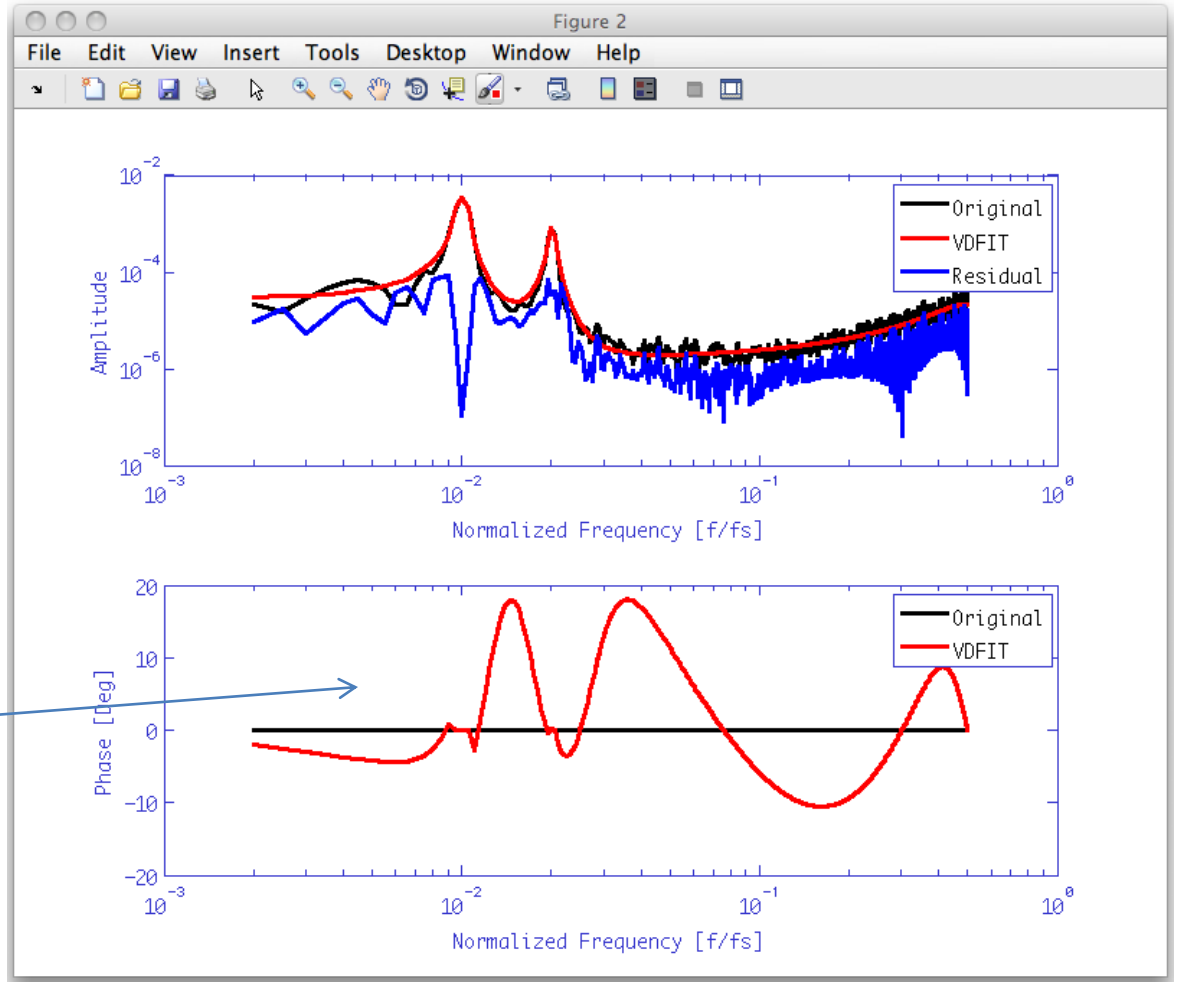
```
plfit = plist('FS',1,...
              'AutoSearch','on',...
              'StartPolesOpt','c1',...
              'maxiter',50,...
              'minorder',10,...
              'maxorder',45,...
              'weights',wgh,... % assign externally calculated weights
              'ResLogDiff',[],...
              'ResFlat',0.77,...
              'RMSE',5,...
              'Plot','on',...
              'ForceStability','off',...
              'CheckProgress','off');

% Do the fit
[param,fmod] = zDomainFit(tnxxr,plfit);
```



Topic 5 – Exercise 2



Topic 5 – Exercise 2



Phase is not perfectly fitted – this will be solved with the next release

Topic 5 – Exercise 2

Generate white noise

```
a = ao(plist('tsfcn', 'randn(size(t))', 'fs', 1, 'nsecs', 10000,'yunits','m'));
```

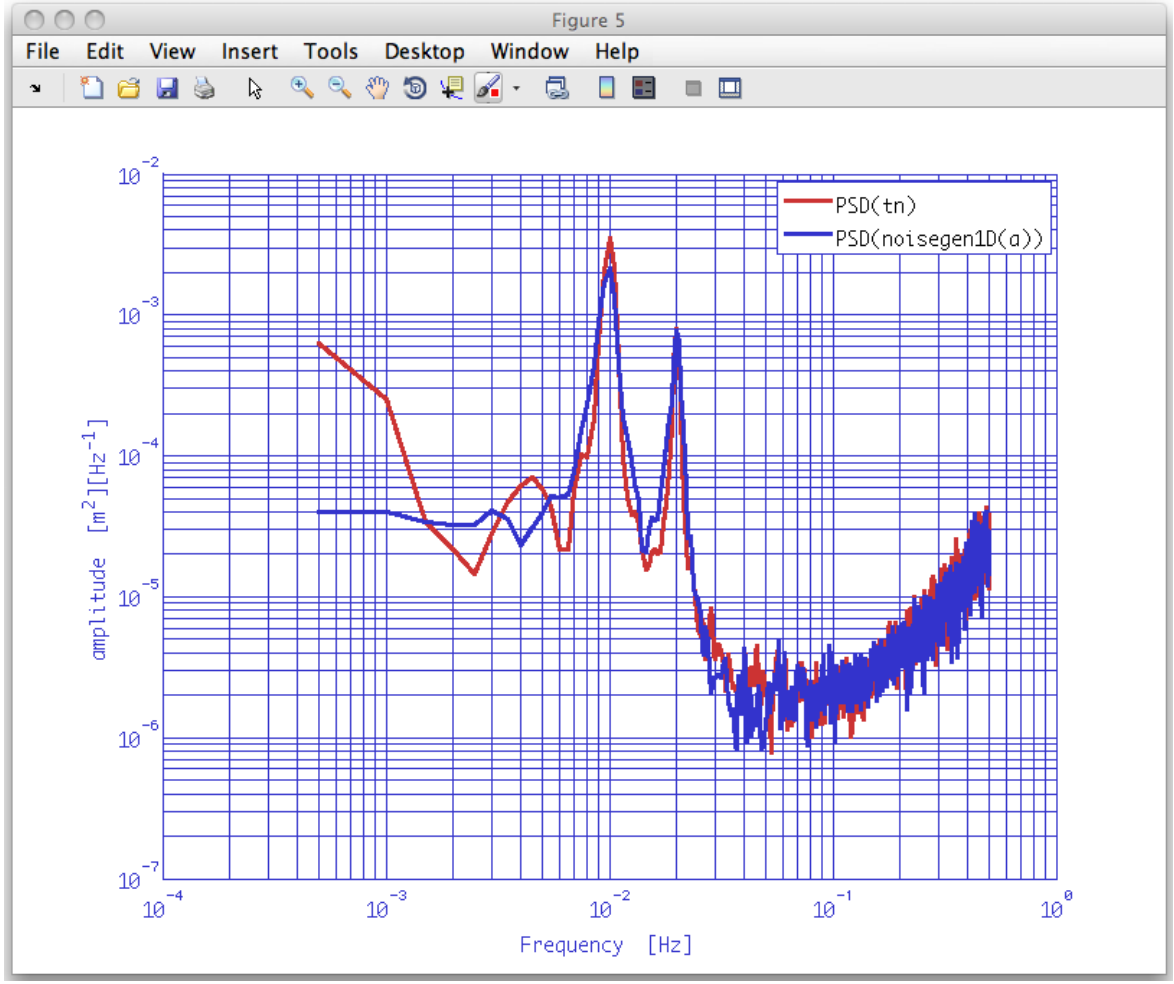
Color noise with given psd

```
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);
```

Calculate psd and plot

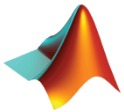
```
acxx = ac.psd(plist('Nfft',2000));
iplot(tnxx,acxx)
```

Topic 5 – Exercise 2



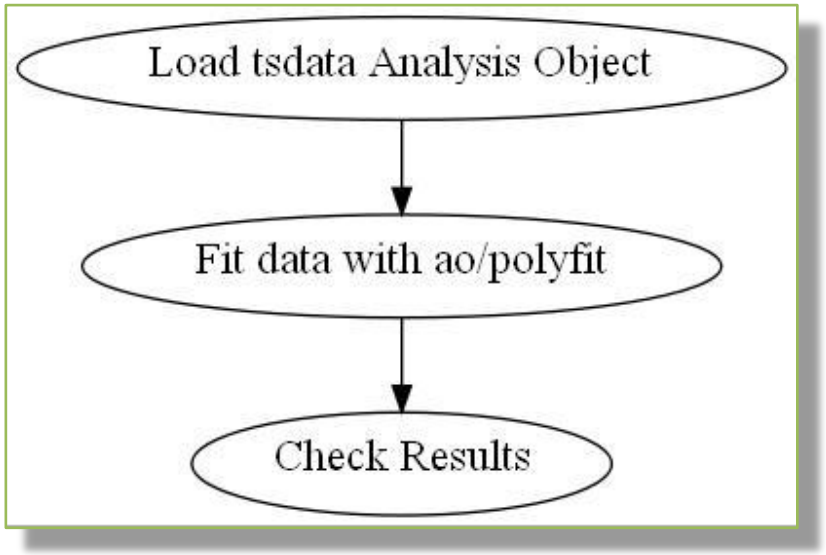
Topic 5 – Exercise 3

- Go to help section
  - LTPDA Toolbox
    - LTPDA Training Session 1
      - Topic 5 - Model fitting
      - Open the page of the first exercise
        - » Fitting time series with polynomials
      - Open a new editor window
        - » In Matlab command window type » edit



Topic 5 – Exercise 3
Fit Time Series with Polynomials

Relevant functions	
polyfit	Fit a time series with a 6 order polynomial



Topic 5 – Exercise 3

Load test data

```
a = ao(plist('filename', 'topic5/T5_Ex04_TestNoise.xml'));
a.setName;
```

Fit data with a polynomial

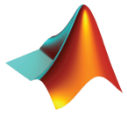
```
plfit = plist('N', 6);
p     = polyfit(a, plfit);
```

Evaluate fitted model

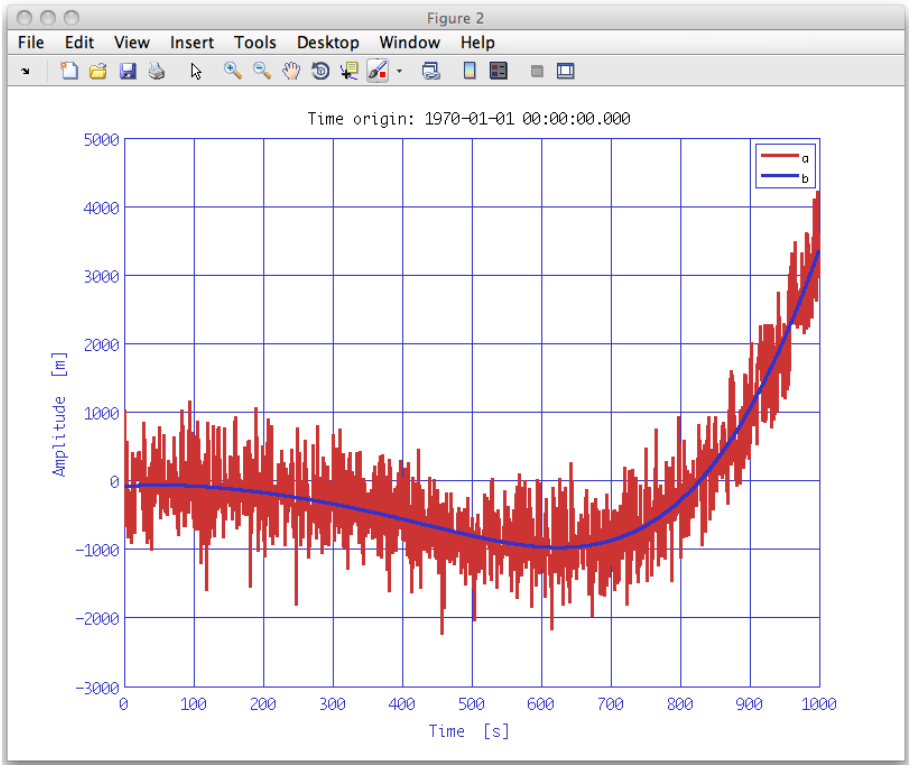
```
b = ao(plist('polyval', p, 't', a));
b.setYunits(a.yunits);
b.setName;
```

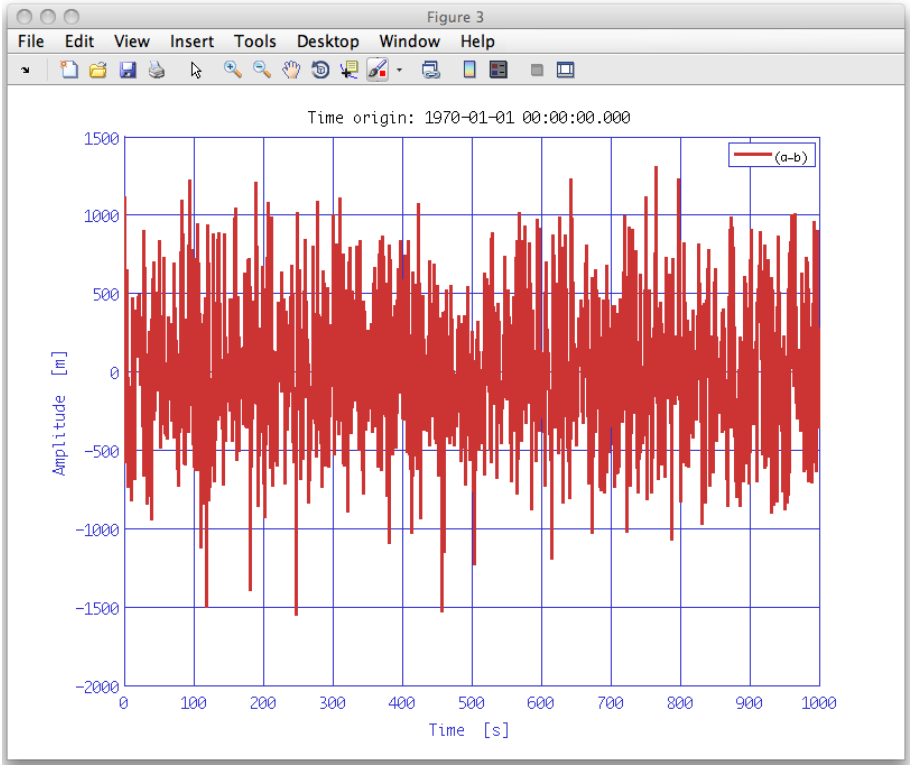
Check results - plot data, model and fit residuals

```
iplot(a,b)
iplot(a-b)
```



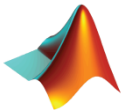
Topic 5 – Exercise 3





Topic 5 – Exercise 4

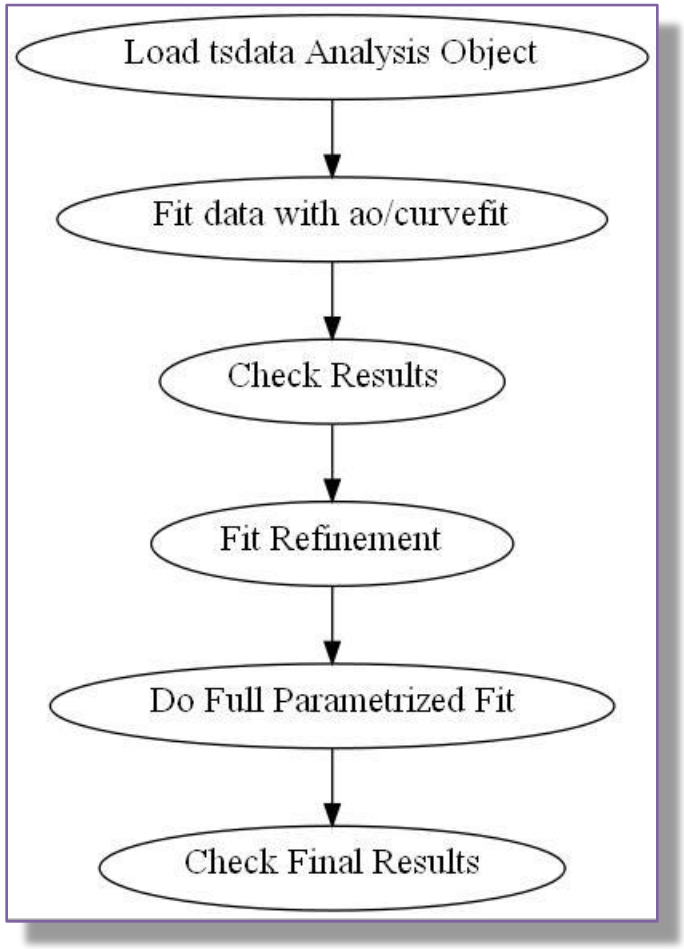
- Go to help section
  - LTPDA Toolbox
    - LTPDA Training Session 1
      - Topic 5 - Model fitting
      - Open the page of the first exercise
        - » Non-linear least square fitting of time series
      - Open a new editor window
        - » In Matlab command window type » edit



Topic 5 – Exercise 4
Non-linear least square fit of time series

Relevant functions	
curvefit	Fit a time series with a non-linear model (linearly chirped sine wave)

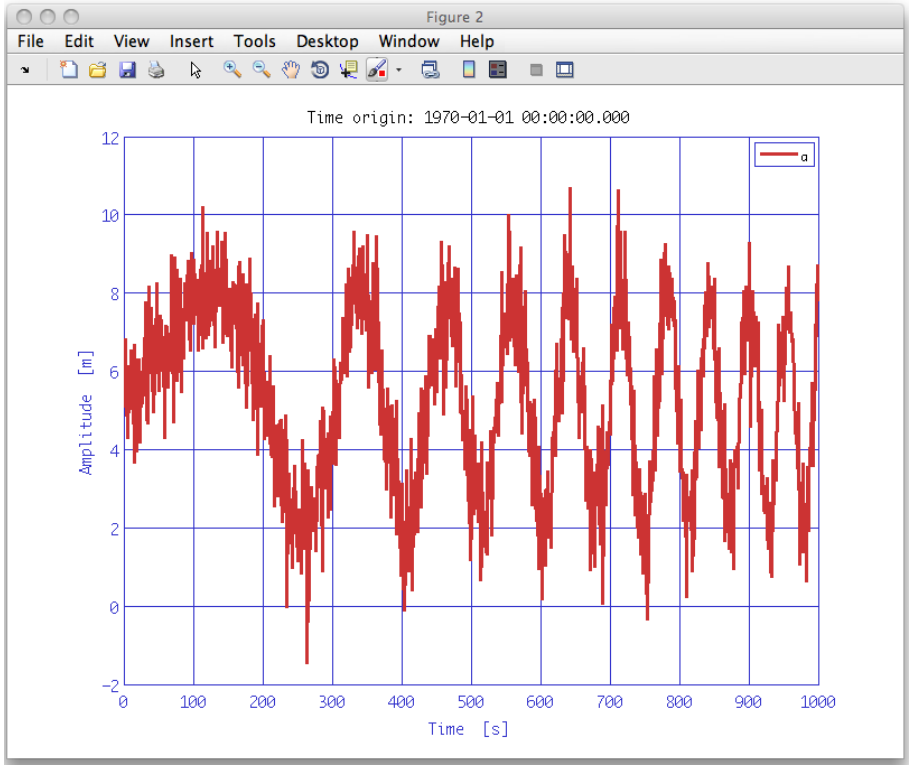
$$A\sin\left[2\pi \quad f_0 + kt \quad t + \varphi\right]$$



Topic 5 – Exercise 4

Load data and plot

```
a = ao(plist('filename', 'topic5/T5_Ex05_TestNoise.xml'));
a.setName;
iplot(a)
```



Topic 5 – Exercise 4

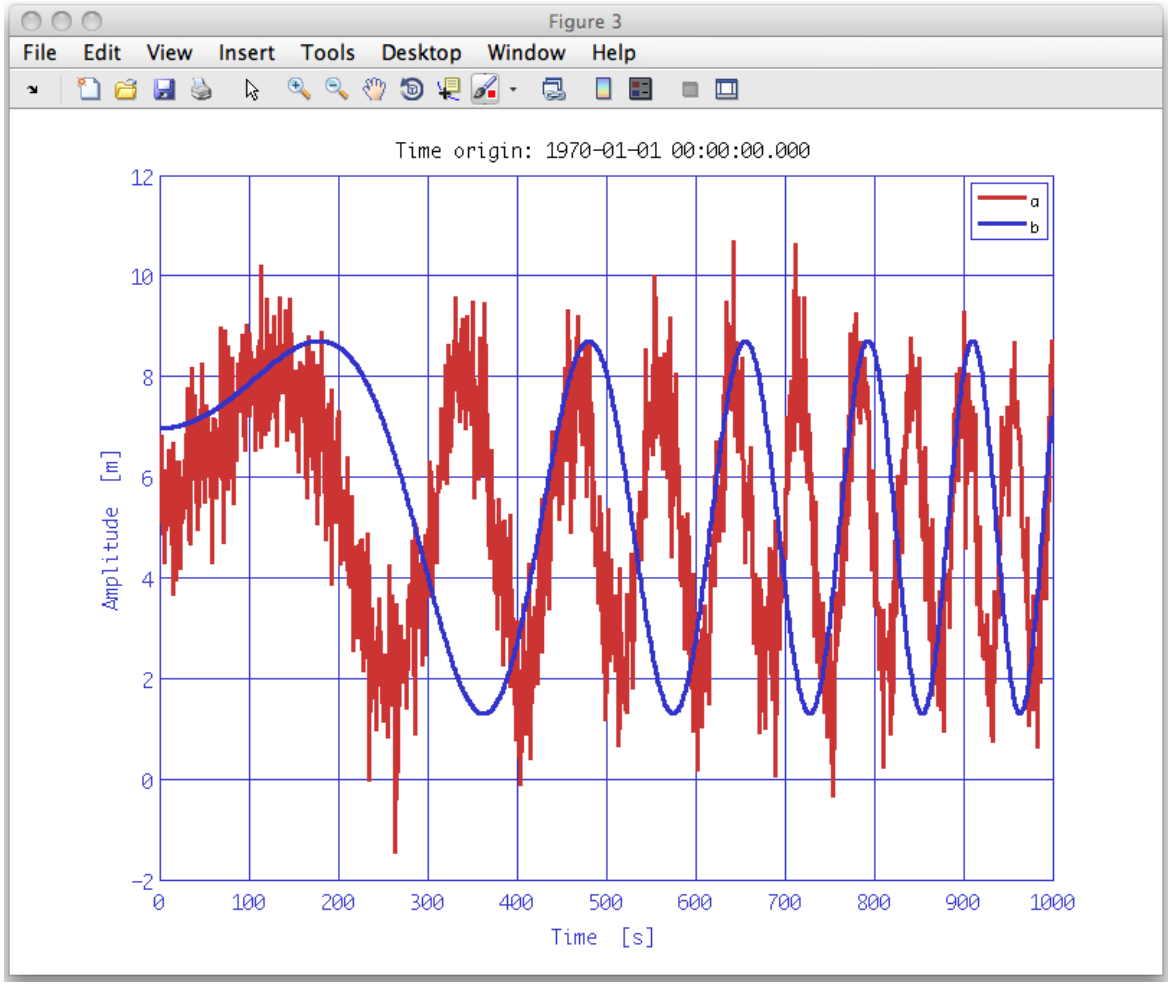
Fit data – Try fitting Amplitude, Frequency, Chirp Parameter and Phase

```
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);
```

Evaluate fit results

```
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;
iplot(a,b)
```

Topic 5 – Exercise 4

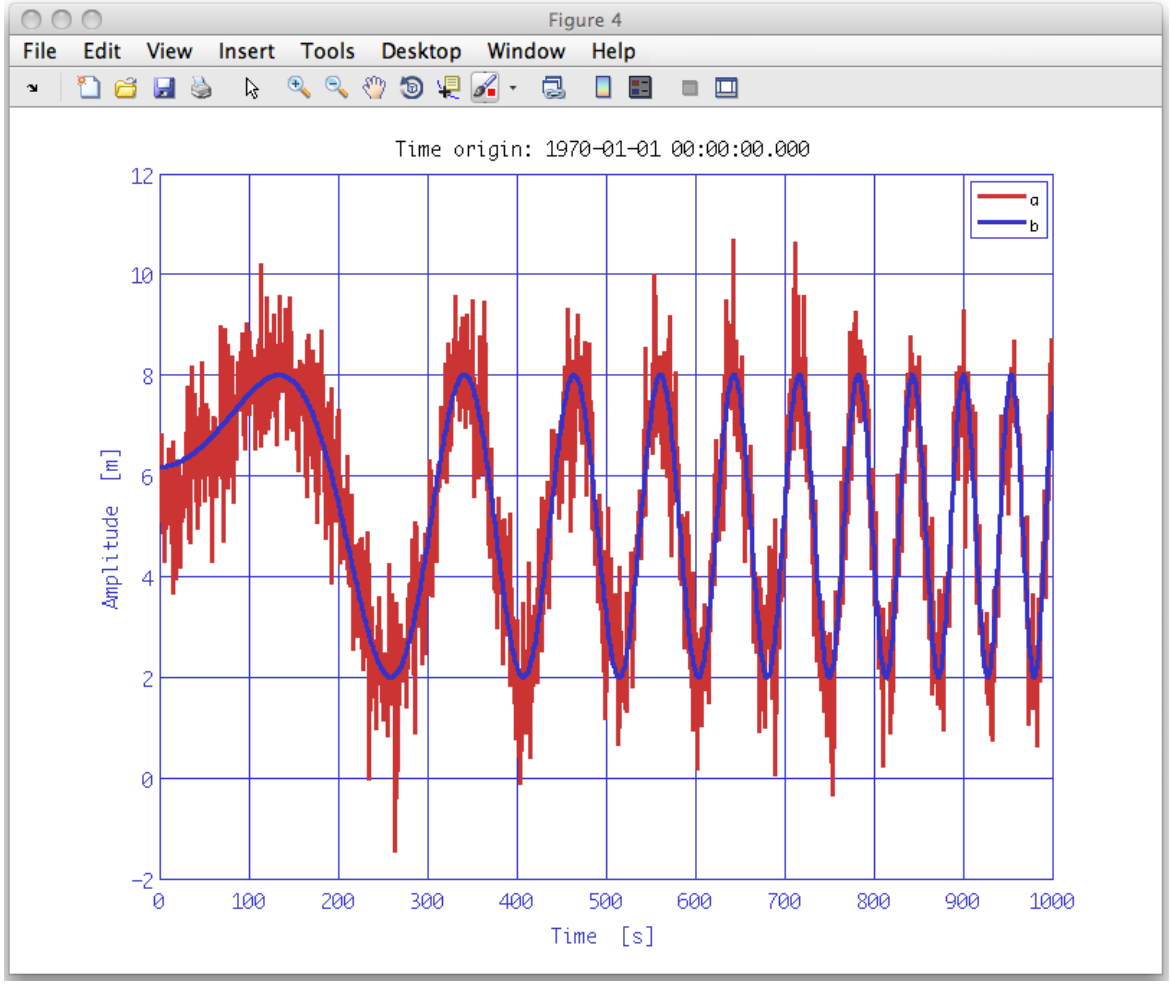


Topic 5 – Exercise 4

Run a fit with a reduced set of parameters – fix amplitude and phase

```
% Do the fit again
plfit = plist('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 = plist('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;

iplot(a,b)
```

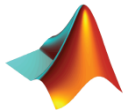
Topic 5 – Exercise 4



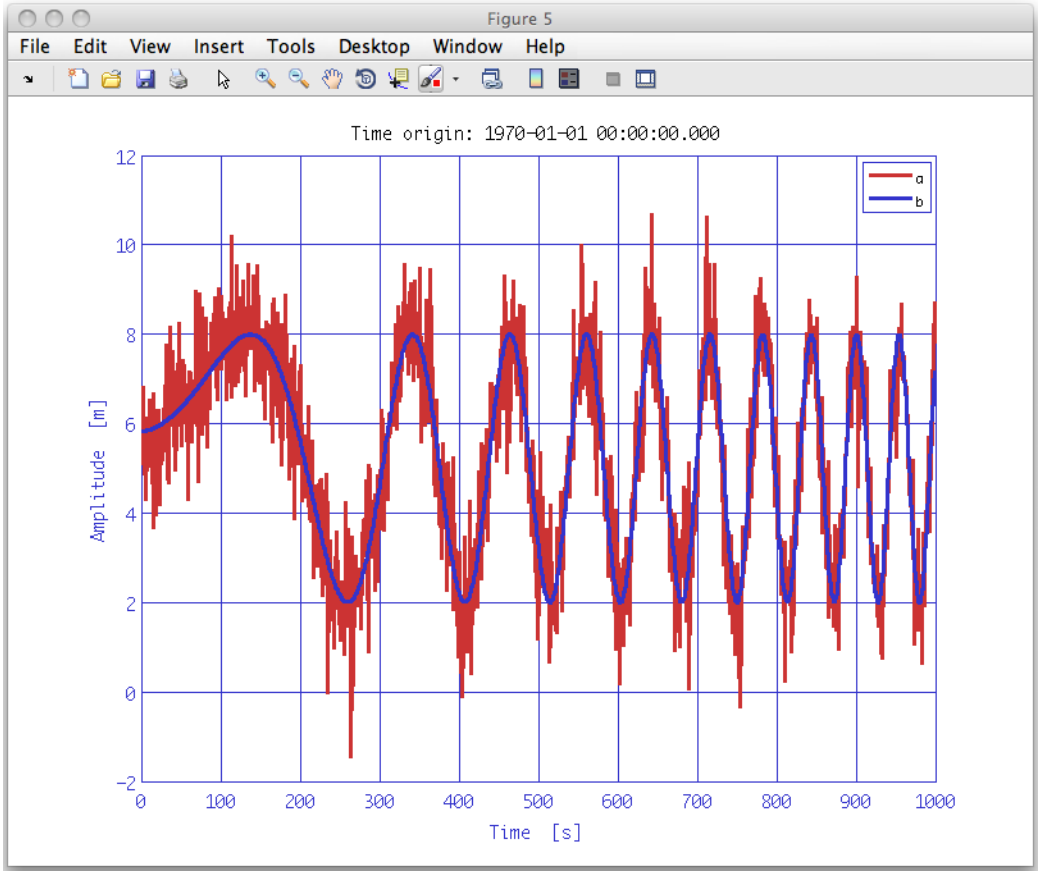
Topic 5 – Exercise 4

Run a fit with the full set of parameters – use former results as starting guess for frequency and chirp parameter

```
% 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;

iplot(a,b)
```



Topic 5 – Exercise 4



True parameters

```
ADDP = 5
P(1) = 3
P(2) = 1e-4
P(3) = 1e-5
P(4) = 0.3
```

Fitted parameters

```
ADDP = 5
P(1) = 2.993
P(2) = 0.000121
P(3) = 9.983e-006
P(4) = 0.278
```

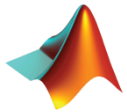
Topic 5 – Exercise 4

We could look at the covariance matrix in the process info
» In the command window of Matlab »

```
    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
```

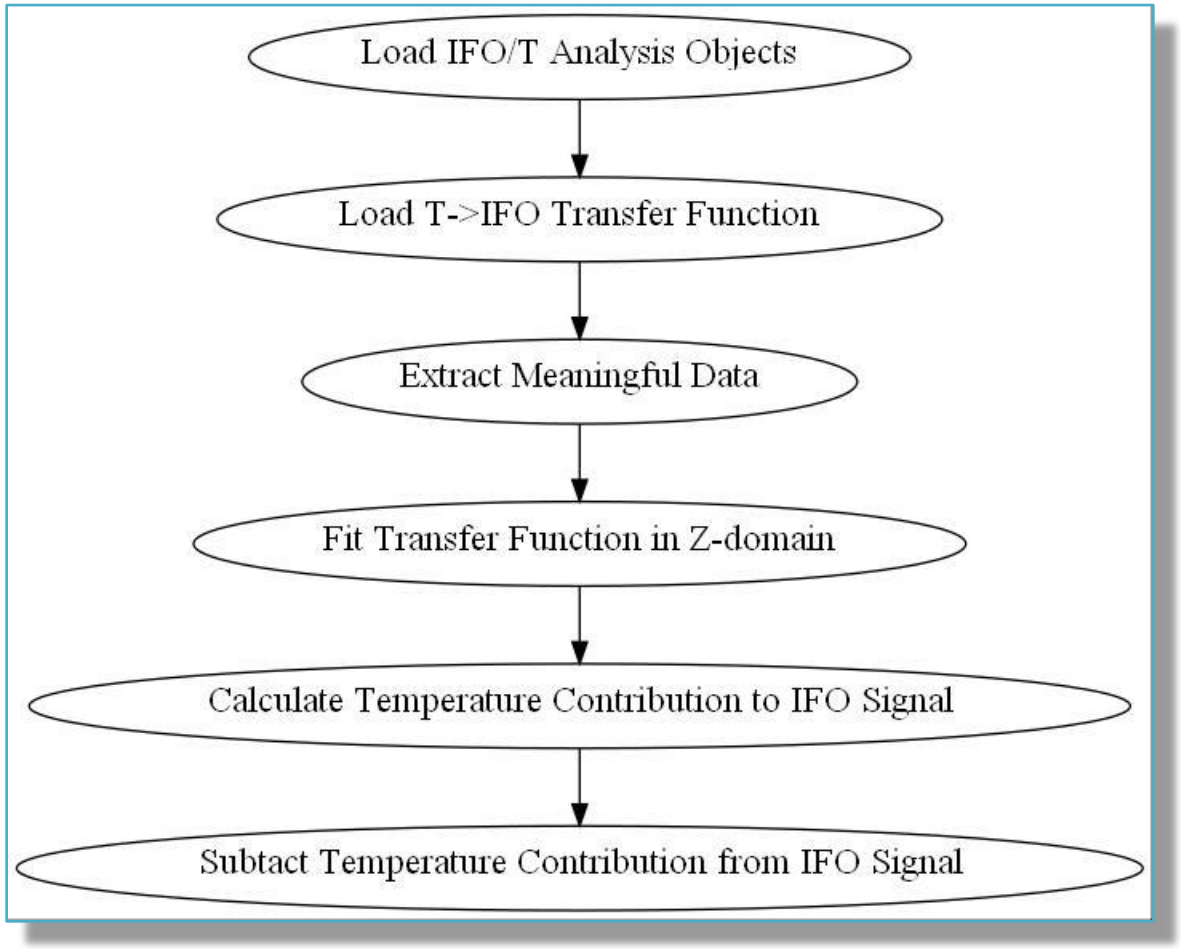
Topic 5 – Exercise 5

- Go to help section
  - LTPDA Toolbox
    - LTPDA Training Session 1
      - Topic 5 - Model fitting
      - Open the page of the first exercise
        - » Time-domain subtraction of temperature contribution to interferometer signal
      - Open a new editor window
        - » In Matlab command window type » edit



Topic 5 – Exercise 5
IFO/Temperature Example

Relevant functions
zDomainFit
Fit a z domain model to measured transfer function. The model is used to filter data so as isolating temperature contribution to IFO signal



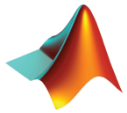
Topic 5 – Exercise 5

Load data from exercise 2 and split to extract the good part

```
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);
```

Plot to inspect data

```
iplot(ifo_red,T_red,plist('arrangement', 'subplots'))
```



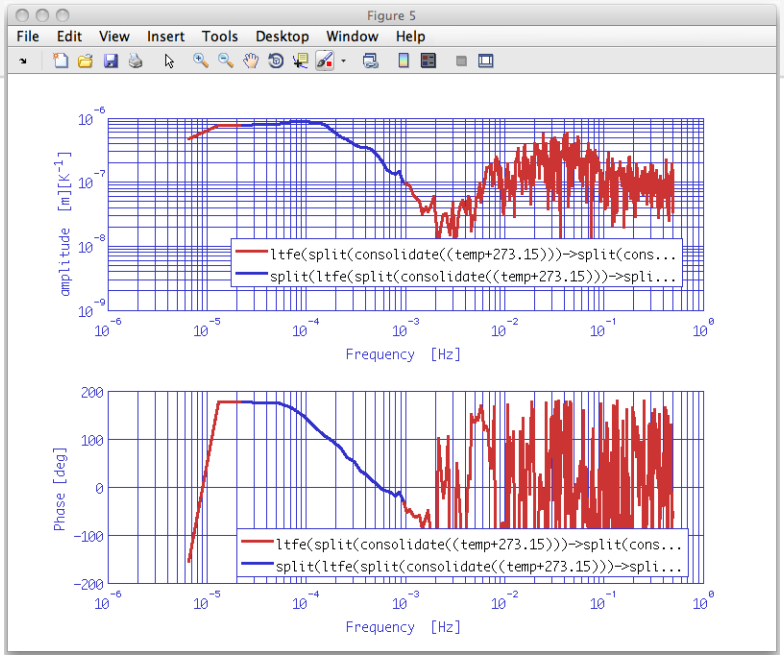
Topic 5 – Exercise 5

Load transfer function data from exercise 4

```
tf = ao('ifo_temp_example/T_ifo_tf.xml');
```

Split to extract meaningful data

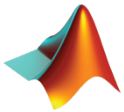
```
tfsp = split(tf,plist('frequencies', [2e-5 1e-3]));
iplot(tf,tfsp)
```



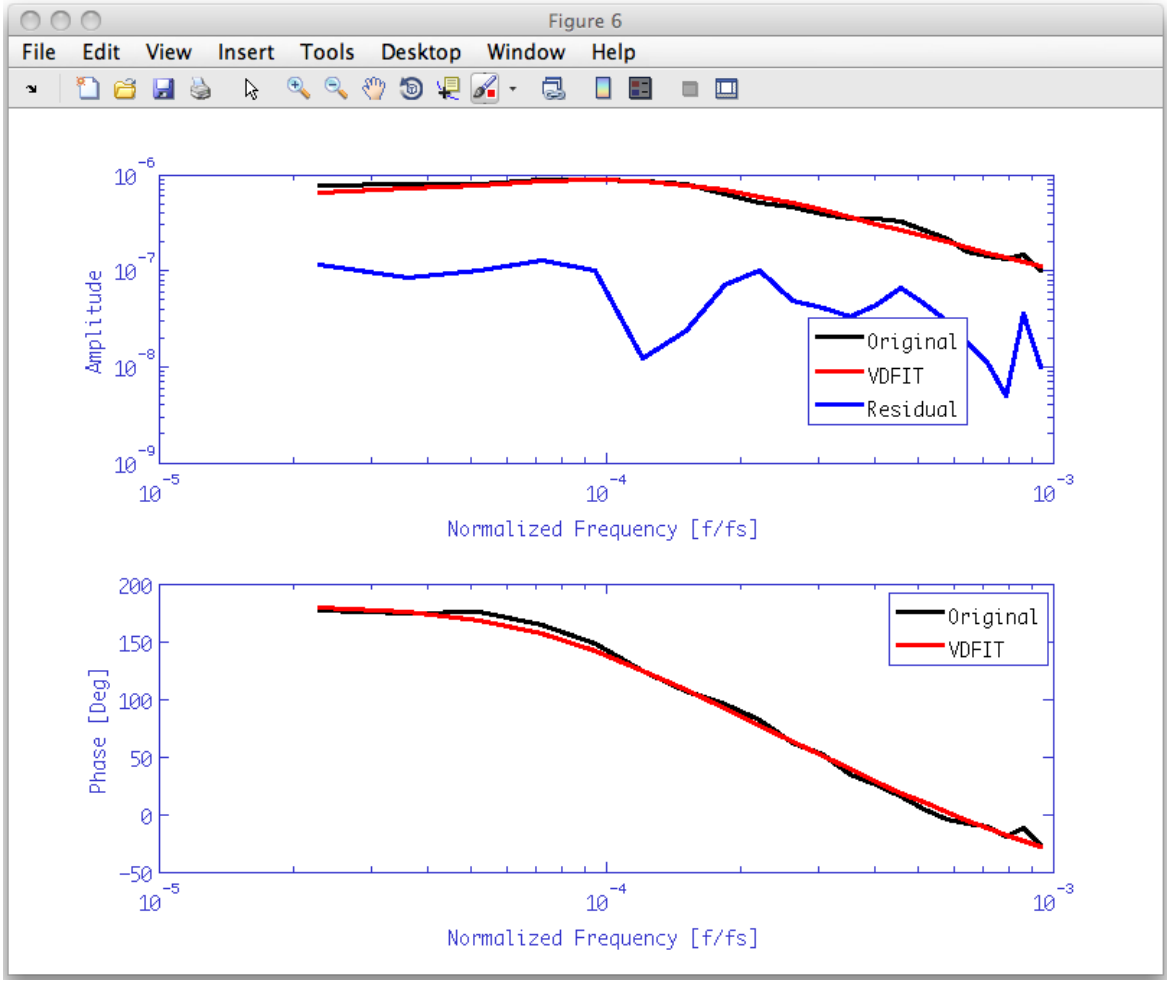
Topic 5 – Exercise 5

Fit transfer function – fit with a fixed order

```
plfit = plist('FS',1,...
    'AutoSearch','off',...
    'StartPolesOpt','c1',...
    'maxiter',20,...
    'minorder',3,...
    'maxorder',3,...
    'weightparam','abs',...
    'Plot','on',...
    'ForceStability','on',...
    'CheckProgress','off');

fobj = zDomainFit(tfsp,plfit);
fobj.setIunits('K');
fobj.setOunits('m');
```



Topic 5 – Exercise 5



Topic 5 – Exercise 5

Filter Temperature data with the fitted model in order to extract temperature contribution to interferometer signal

```
ifoT = filter(T_red,fobj,plist('bank','parallel'));
ifoT.detrend(plist('order',0));
ifoT.simplifyYunits;
ifoT.setName;
```

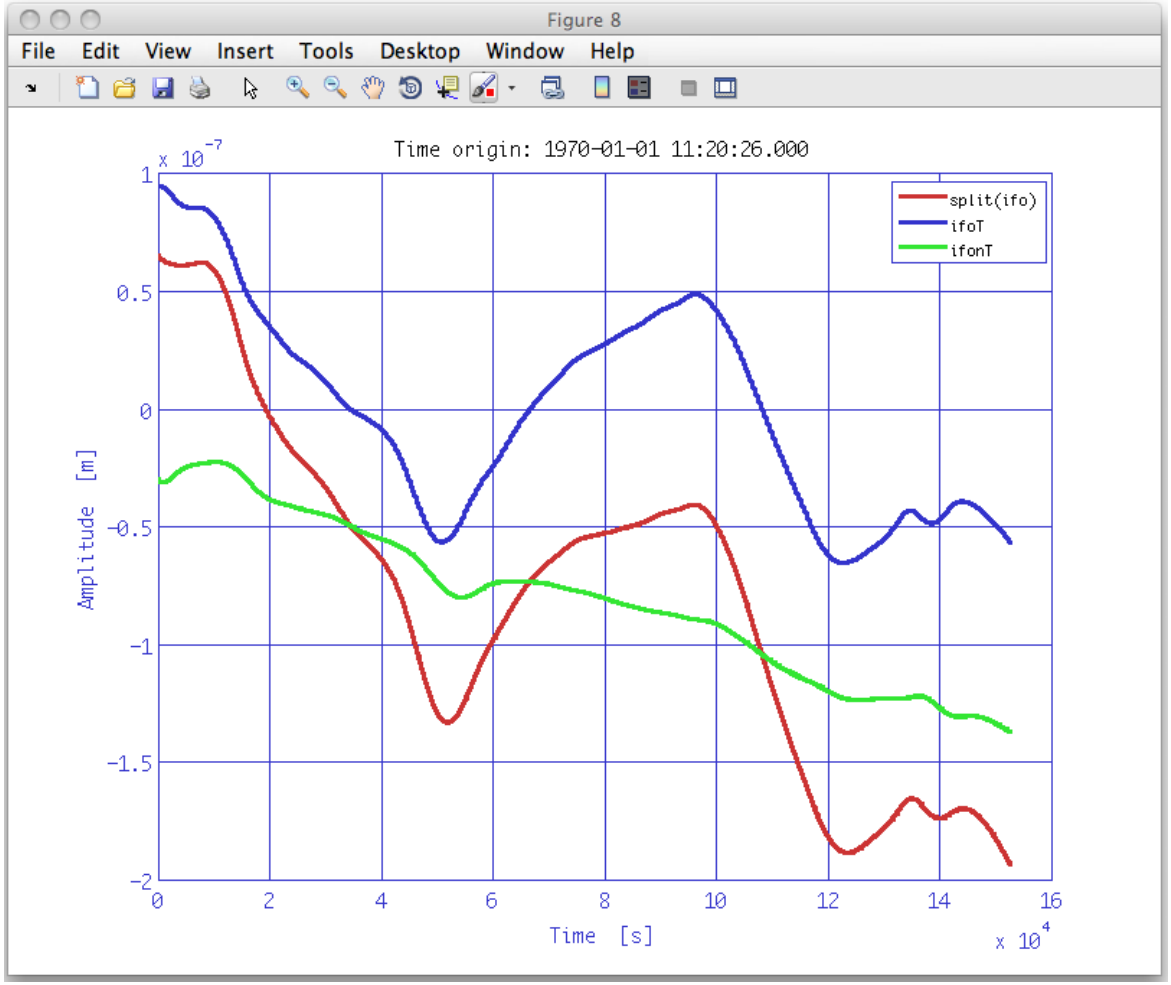
Subtract temperature contribution

```
ifonT = ifo_red - ifoT;
ifonT.setName;
```

Plot to check results

```
iplot(ifo_red,ifoT,ifonT)
```

Topic 5 – Exercise 5

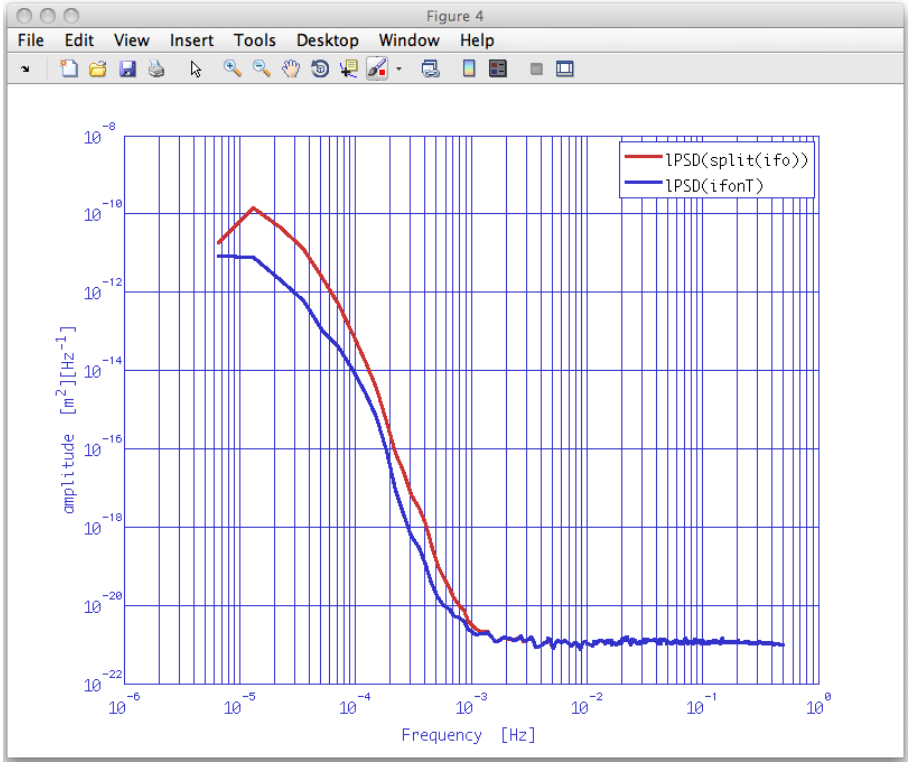


Topic 5 – Exercise 5

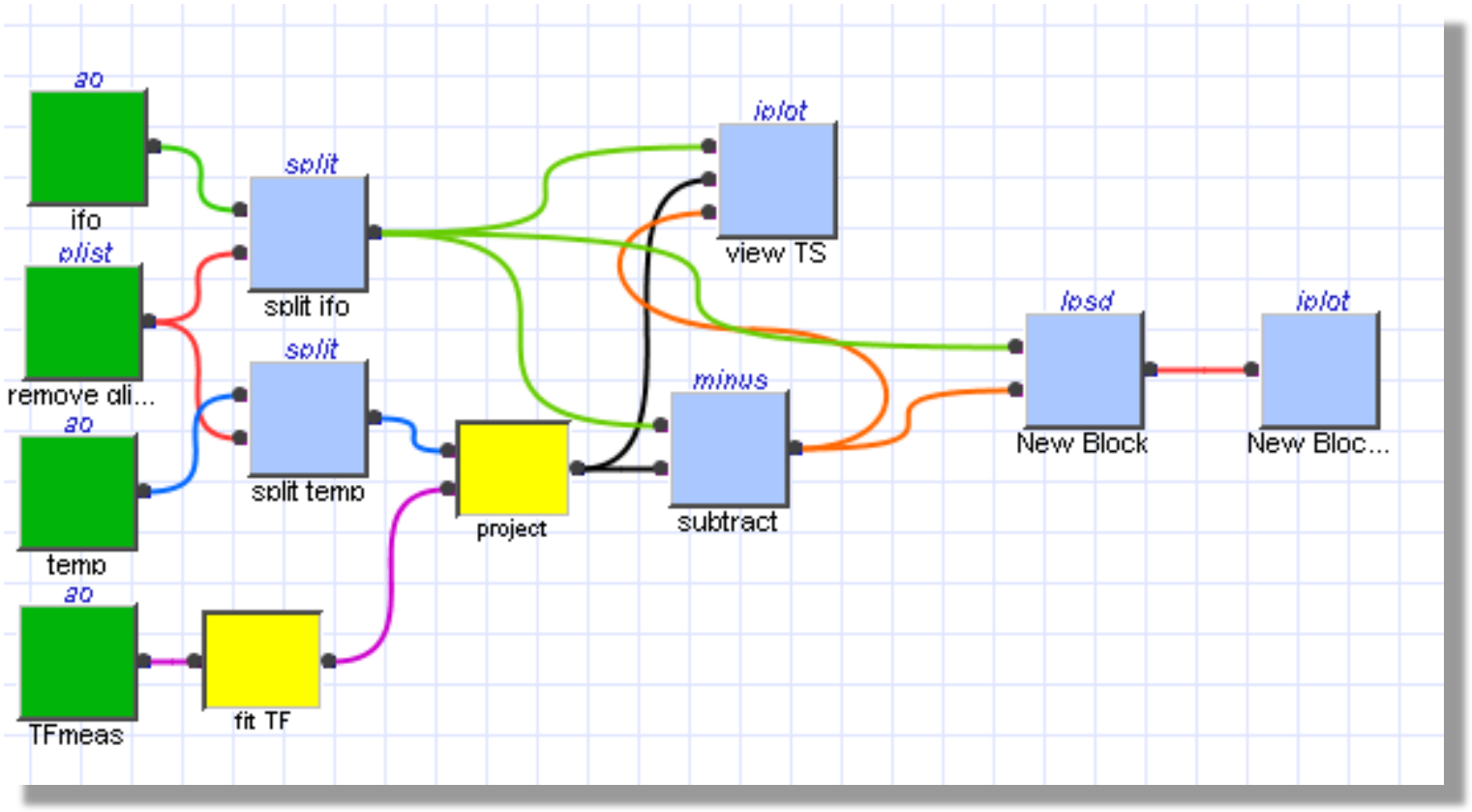
Compare power spectral density of IFO and IFO-Temp signals

```
ifoxx = ifo_red.lpsd;
ifonTxx = ifonT.lpsd;
iplot(ifoxx,ifonTxx)
```

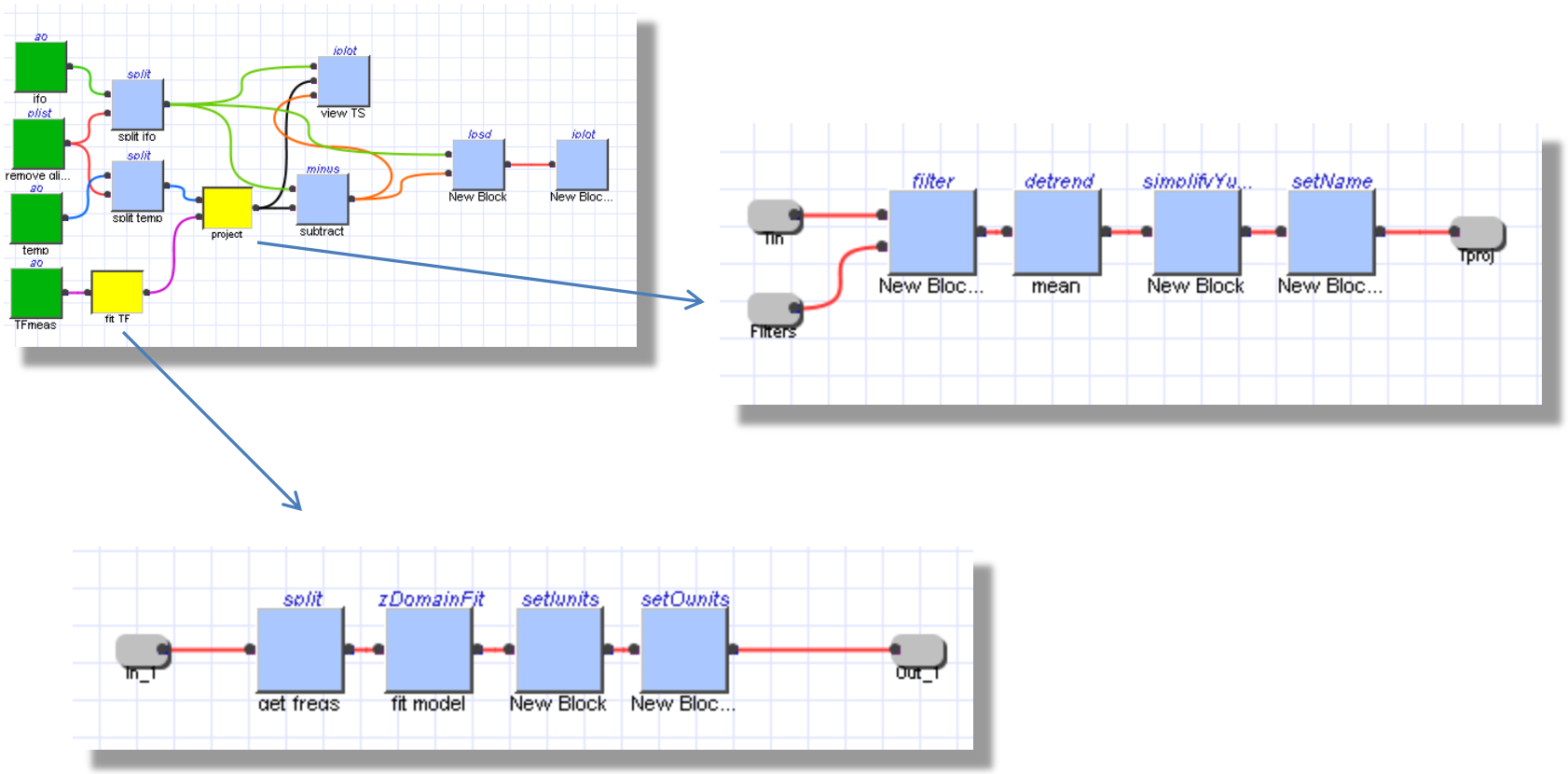
Temperature subtraction changes low frequencies power content



Topic 5 – Exercise 5



Topic 5 – Exercise 5



LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

- home
- Installation
- System requirements
- Downloads
- File repository
- Release Schedule
- User manual
- Training Sessions
- Documents
- Bugs and features
- Troubleshooting

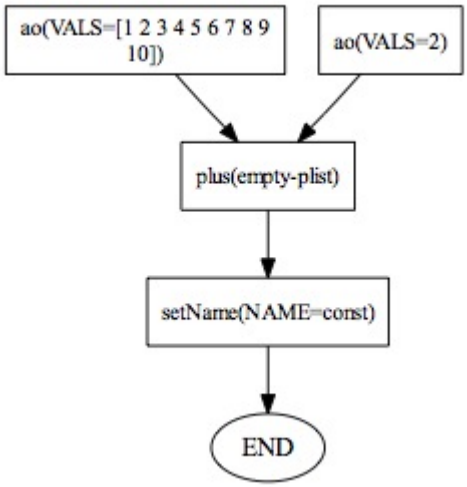
LTPDA is a MATLAB toolbox that uses an object-oriented approach to data analysis. LTPDA Objects are processed through a data analysis pipeline. At each analysis step, a record is kept of exactly what algorithm was applied to which object and with which parameters. In this way, the result of a particular data analysis is one or more objects, each containing the final result as numerical data together with a full processing history of how the result was achieved.

Latest version: V2.0.1

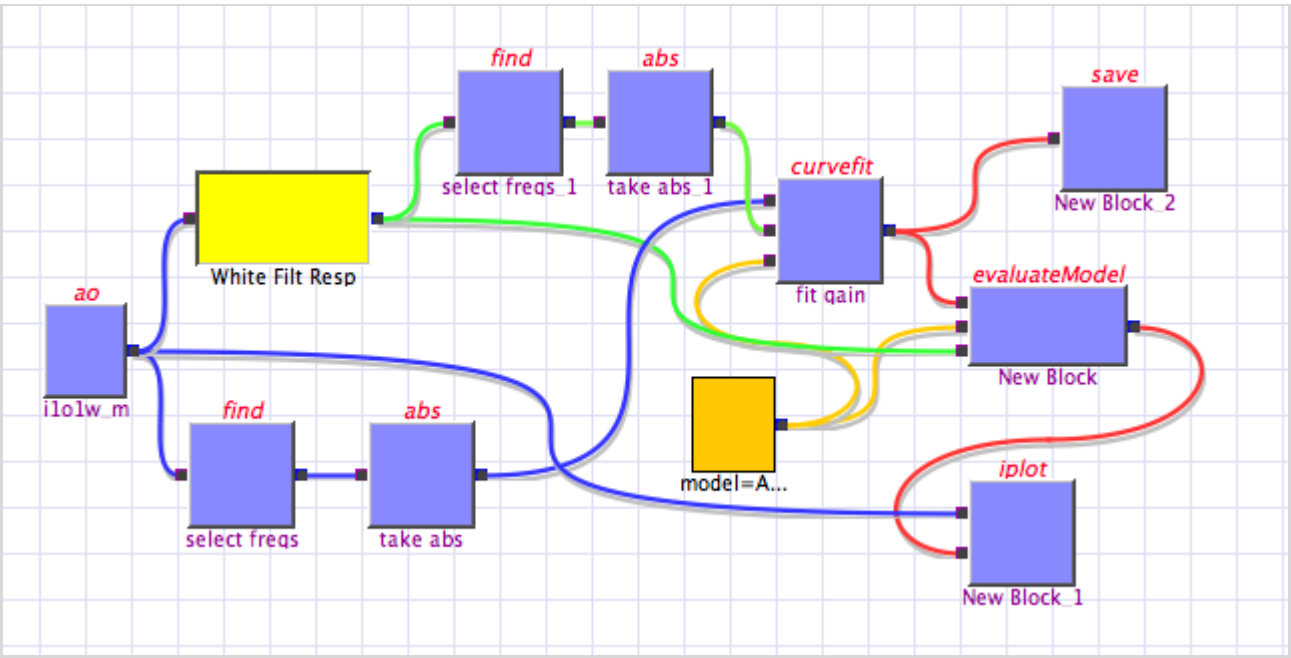
LTPDA includes algorithms and objects for

1. pre-processing of time-series data
2. performing spectral analysis of various kinds
3. performing digital filtering via IIR and FIR filters
4. constructing pole/zero models
5. constructing state-space models
6. and much more

```
% Create two AOs
a = ao(1:10);
b = ao(2);

% Add them together
c = a+b;

% Set the name
c.setName('const')

% Plot the result
c.iplot
```



In addition, there is a graphical design interface which allows data analysis pipelines to be built using a diagram editor. The resulting diagram is then executed via LTPDA commands.



This toolbox is being developed for use in the data analysis of the LISA Pathfinder Mission.

home >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Here you will find details of LTPDA training sessions.

1. Training session 1

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Training Session 1
Documents
Bugs and features
Troubleshooting

home >> Training Sessions >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

Installation

See the appropriate section of the LTPDA User Manual for installation instructions.

home >> Installation >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

System Requirements

The minimum requirements for the latest version of the LTPDA Toolbox are:

Software	Version
MATLAB	7.6 (R2009a)
Signal Processing Toolbox	6.11
Database Toolbox*	3.5.1
Symbolic Math Toolbox	5.2
Optimisation Toolbox	4.2

* Needed for interacting with an LTPDA Repository. 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.

home >> System requirements >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Select the version you want to download.

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

Training Sessions
Documents
Bugs and features
Troubleshooting

home >> Downloads >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Select which type of files you want to download.

- home
- Installation
- System requirements
- Downloads
- **File repository**
  - mfiles
  - pipelines
  - objects
  - Movies
- Release Schedule
- User manual
- Training Sessions
- Documents
- Bugs and features
- Troubleshooting

home >> File repository >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version	Release Date
0.1	1st May 2007
0.2	1st June 2007
0.3	1st July 2007
0.4	1st August 2007
0.5	1st September 2007
0.6	1st October 2007
0.7	1st November 2007
0.8	1st December 2007
0.9	1st January 2008
1.0	1st February 2008
2.0	1st March 2009
2.1	October 2009
3.0	April 2010

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

home >> Release Schedule >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

LTPDA User Manual

You can view the user manual on-line here.

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

home >> User manual >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

AO Design Document
1st December, 2006

operator_rules.pdf
The design document for the rules governing the behaviour of the basic math operators of the AO class.

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

home >> Documents >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

Bugs and feature requests can be submitted here. You can self-subscribe to be a reporter.

home >> Bugs and features >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

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 by Mauro Hueller in case the above doesn't work: What happens with Matlab2007b on WinXP is that after you create the java.opts file, Matlab actually doesn't start (it crashes after the splashscreen). The workaround I found is setting an environment variable MATLAB_RESERVE_LO=0.
   HOWTO:

   1. Select Start->Settings->Control Panel->System
   2. Select the "Advanced" Tab
   3. On the bottom, center, click on "Environment variables"
   4. Click "New" (I choose the one under "User variables for CurrentUser")

      Variable Name: MATLAB_RESERVE_LO
      Variable Value: 0
   5. Click OK as many times it needs

   [As far as I remember, it shouldn't ask for reboot.]
   Then edit/reate the java.opts file. You can also specifying the units (for instance -Xmx512m or -Xmx524288k or -Xmx536870912 will all give you 512 Mb).

home >> Troubleshooting >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

- home
- Installation
- System requirements
- Downloads
- File repository
- Release Schedule
- User manual
- Training Sessions
- Documents
- Bugs and features
- Troubleshooting

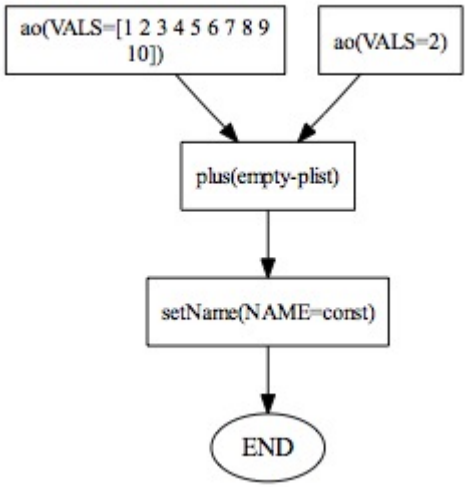
LTPDA is a MATLAB toolbox that uses an object-oriented approach to data analysis. LTPDA Objects are processed through a data analysis pipeline. At each analysis step, a record is kept of exactly what algorithm was applied to which object and with which parameters. In this way, the result of a particular data analysis is one or more objects, each containing the final result as numerical data together with a full processing history of how the result was achieved.

Latest version: V2.0.1

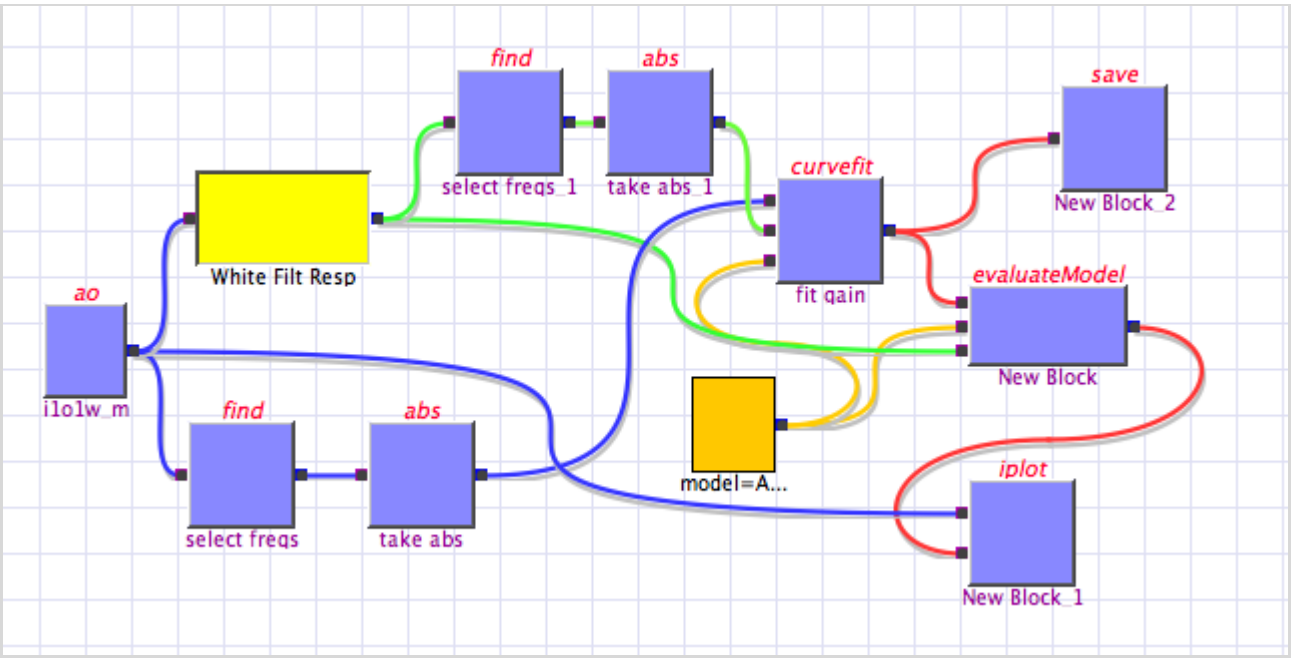
LTPDA includes algorithms and objects for

1. pre-processing of time-series data
2. performing spectral analysis of various kinds
3. performing digital filtering via IIR and FIR filters
4. constructing pole/zero models
5. constructing state-space models
6. and much more

```
% Create two AOs
a = ao(1:10);
b = ao(2);

% Add them together
c = a+b;

% Set the name
c.setName('const')

% Plot the result
c.iplot
```



In addition, there is a graphical design interface which allows data analysis pipelines to be built using a diagram editor. The resulting diagram is then executed via LTPDA commands.



This toolbox is being developed for use in the data analysis of the LISA Pathfinder Mission.

home >>

Max-Planck-Institute for Gravitational Physics (Albert-Einstein-Institute)

UNIVERSITÀ DEGLI STUDI DI TRENTO

Arithmetic Operators in LTPDA

S2-AEI-TN-3059

prepared by	Martin Hewitson
checked by	
Issue	1
Revision	1
Status	Release
Number of pages	12
date of issue	February 18, 2009
approved by	

Part I.
Distribution List

Name	Company/ Institute
M. Hewitson	AEI Hannover

Contents

List of Tables

Operator	Symbol	description
`plus`	`+`	The addition operator.
`minus`	−	The subtraction operator.
`times`	`.*`	Element-wise multiplication operator.
`rdivide`	`./`	Element-wise division operator.

Table 1: Element-wise operators.

1. Introduction

This document aims to outline the desired behaviour for the various arithmetic operators for the AO class of LTPDA.

The aim is to describe how each operator should behave with regards:

- the inputs to the operator,
- the names of the input objects,
- the units of the input objects,
- and the allowed data types.

The data contained in the AO is always handled according to MATLAB's rules. Here we are only concerned with how the AO containers are handled. So, for example, if we want to do `v+b` where `v` is a vector of time-series AOs and `b` is a single time-series AO, we only discuss here how to handle the elements of `v`; after that, MATLAB handles how the data in `v(i)` and `b` are actually added together and will throw errors if MATLAB rules are violated. The rules are defined as a set of examples which should be exhaustive.

In this version of the document, data type `xyzdata` is not considered.

2. Element-wise operators

This section is concerned with those operators typically associated with operating element-wise on the data, for example, the `times` (`.*`) operator. The operators are detailed in Table 1.

2.1. Container rules

The following AO variables are assumed:

`a` a single AO

`b` a single AO

`c` a single AO

`V_N` a vector of N AOs

`U_N` a vector of N AOs

`M_NP` a Matrix of $N \times P$ AOs

`H_PQ` a Matrix of $P \times Q$ AOs

The following rules use `plus` as an example, but the use of the other operators is interchangeable.

2.1.1. Rule 1

```
o = plus(a,b)
```

Here we just add the data in `a` to the data in `b`, assuming the data conforms to MATLAB rules.

2.1.2. Rule 2

```
o = plus(V_N,a)
```

The output `o` will be equivalent to

```
[V(1)+a, V(2)+a, .. , V(N)+a]
```

so the size of `o` will be N elements.

2.1.3. Rule 3

```
o = plus(V_N,V_M)
```

will throw an error.

2.1.4. Rule 4

```
o = plus(V_N,U_N)
```

The output will be the vector:

```
[V(1)+U(1), V(2)+U(2), .. , V(N)+U(N)]
```

such that the size of ○ will be N elements.

2.1.5. Rule 5

```
○ = plus(M_NP,a)
```

will give an output

$$o = \begin{bmatrix} M(1,1)+a & M(1,2)+a & \cdots \\ M(2,1)+a & \ddots & \cdots \\ \vdots & \vdots & M(N,P)+a \end{bmatrix} \tag{1}$$

so ○ will be of size $N \times P$.

2.1.6. Rule 6

```
○ = plus(M_NP,V_N1)
```

will give an output

$$o = \begin{bmatrix} M(1,1)+V(1) & M(1,2)+V(1) & \cdots \\ M(2,1)+V(2) & \ddots & \cdots \\ \vdots & \vdots & M(N,P)+V(N) \end{bmatrix} \tag{2}$$

so ○ will be of size $N \times P$.

2.1.7. Rule 7

```
○ = plus(M_NP,V_1P)
```

will give an output

$$o = \begin{bmatrix} M(1,1)+V(1) & M(1,2)+V(2) & \cdots \\ M(2,1)+V(1) & \ddots & \cdots \\ \vdots & \vdots & M(N,P)+V(P) \end{bmatrix} \tag{3}$$

so o will be of size $N \times P$.

2.1.8. Rule 8

```
o = plus(M_NP,V_Q)
```

will give an error.

2.1.9. Rule 9

```
o = plus(M_NP,H_PQ)
```

will give an error.

2.1.10. Rule 10

```
o = plus(M_NP,H_NP)
```

will give an output

$$o = \begin{bmatrix} M(1,1)+H(1,1) & M(1,2)+H(1,2) & \cdots \\ M(2,1)+H(2,1) & \ddots & \cdots \\ \vdots & \vdots & M(N,P)+H(N,P) \end{bmatrix} \tag{4}$$

so o will be of size $N \times P$.

2.1.11. Rule 11

For more than two inputs, the result will be as if the commands are nested. For example:

```
o = plus(a,b,c)
```

will be handled the same as

```
o = plus(plus(a,b),c)
```

or for

```
o = plus(V_N,b,c)
```

	tsdata	fsdata	xydata	cdata
tsdata	y	n	y	y
fsdata	n	y	y	y
xydata	y	y	y	y
cdata	y	y	y	y

Table 2: Data type compatibility.

will be handled the same as

```
o = plus(plus(V_N,b),c)
```

At each stage, the above rules must apply.

2.2. Data types

The data compatibility matrix for these operators is given in Table 4.

2.3. Object names

The name of the output object will be based on the names of the input objects as

```
op = (in1.name op in2.name)
```

The name will be extended at each stage of a nested operation. For example, suppose we have three AOs, `a`, `b`, and `c`, each having a name the same as the variable name, then

```
o = plus(a,b,c)
```

will yield a name:

```
o.name = ((a+b)+c)
```

2.4. Units

2.4.1. `plus`, `minus`

For the addition operator, the units of the two inputs must be the same, or one (or both) must be empty (dimensionless). The output data will have units of the either input which is dimensioned.

Operator	Symbol	description
`mtimes`	*	The matrix multiplication operator.
`mrdivide`	/	The matrix division operator.

Table 3: Matrix operators.

2.4.2. `times`, `rdivide`

There are no restrictions for these operators. The units of the output data will be the product (or division) of the input units.

3. Matrix operators

This section is concerned with those operators typically associated with matrix manipulation. The operators are detailed in Table 3.

3.1. Container rules

The following AO variables are assumed:

`a` a single AO

`b` a single AO

`V_N` a vector of N AOs

`U_N` a vector of N AOs

`M_NP` a Matrix of $N \times P$ AOs

`H_PQ` a Matrix of $P \times Q$ AOs

The following rules use `mtimes` as an example, but the use of the other operators is interchangeable.

3.1.1. Rule 1

```
o = mtimes(a,b)
```

will yield an single output AO.

3.1.2. Rule 2

`o = mtimes(V_N1,a)` or `o = mtimes(V_1N,a)`

will behave the same as

```
o = times(V_N1,b)
```

for the AO containers, but the data will be processed by MATLAB according to the `mtimes` operator. So the output will be

```
o = [V(1)*b, V(2)*b, .. , V(N)*b]
```

3.1.3. Rule 3

```
o = mtimes(V_1N,U_N1)
```

will yield an output

```
o = [V(1)*U(1) + V(2)*U(2) + .. + V(N)*U(N)]
```

3.1.4. Rule 4

```
o = mtimes(V_1N,U_1N)
```

will throw an error.

3.1.5. Rule 5

```
o = mtimes(V_N1,U_1M)
```

will yield an output equivalent to

$$o = \begin{bmatrix} V(1)*U(1) & V(1)*U(2) & \cdots \\ V(2)*U(1) & \ddots & \cdots \\ \vdots & \vdots & V(N)*U(M) \end{bmatrix} \tag{5}$$

so of size $N \times M$.

3.1.6. Rule 6

```
o = mtimes(V_N1,U_N1)
```

will throw an error.

3.1.7. Rule 7

```
o = mtimes(M_NP,V_N1)
```

will throw an error.

3.1.8. Rule 8

```
o = mtimes(M_NP,V_1P)
```

will throw an error.

3.1.9. Rule 9

```
o = mtimes(M_NP,H_NP)
```

will throw an error.

3.1.10. Rule 10

```
o = mtimes(M_NP,H_PQ)
```

will yield an output

$$o = \begin{bmatrix} M(1,:) * H(:,1) & M(1,:) * H(:,2) & \cdots \\ M(2,:) * H(:,1) & \ddots & \cdots \\ \vdots & \vdots & M(N,:) * H(:,Q) \end{bmatrix} \tag{6}$$

so of size $N \times Q$.

	tsdata	fsdata	xydata	cdata
tsdata	y	n	y	y
fsdata	n	y	y	y
xydata	y	y	y	y
cdata	y	y	y	y

Table 4: Data type compatibility.

3.2. Data types

The data compatibility matrix for these operators is given in Table 4.

3.3. Object names

The name of the output object will be based on the names of the input objects as

```
out.name = [( in1.name op in2.name )]
```

The name will be extended at each stage of a nested operation. For example, suppose we have three AOs, `a`, `b`, and `c`, each having a name the same as the variable name, then

```
o = mtimes(a,b,c)
```

will yield a name:

```
o.name = ((a*b)*c)
```

3.4. Units

3.4.1. `mtimes`, `mrdivide`

There are no restrictions for these operators. The units of the output data will be the product (or division) of the input units.

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

The V2.0.1 release of the toolbox.

ltpda_toolbox_2.0.1.zip
The V2.0.1 release of the toolbox.

change_log_v2.0.1.pdf
Change Log

ltpda_client_um_201_090309.pdf
User manual

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

The V2.0.1 release of the toolbox.

ltpda_toolbox_2.1rc1.zip
The first release candidate for V2.1 of the toolbox.

change_log_v2.1rc1.pdf
Change Log

ltpda_client_um_201_090309.pdf
User manual

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

The V2 release of the toolbox.

ltpda_toolbox_2.0.zip
The V2 release of the toolbox.

change_log_v2.0.pdf
Change Log

ltpda_client_um_20_040309.pdf
User manual

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - **version_2.0**
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

The third release candidate for V2 of the toolbox.

ltpda_toolbox_2.0rc3.zip
Third release candidate of version 2.0 of the LTPDA Toolbox

change_log_v2.0rc3.pdf
Change Log

ltpda_client_um_20rc1_030109.pdf
User manual

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - **version_2.0rc3**
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

The second release candidate for V2 of the toolbox.

ltpda_toolbox_2.0rc2.zip
Second release candidate of version 2.0 of the LTPDA Toolbox

change_log_v2.0rc2.pdf
Change Log

ltpda_client_um_20rc1_030109.pdf
User manual

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

Training Sessions
Documents
Bugs and features
Troubleshooting

home >> Downloads >> version_2.0rc2 >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

The first release candidate for V2 of the toolbox.

ltpda_toolbox_2.0rc1.zip
First release candidate of version 2.0 of the LTPDA Toolbox

change_log_v2.0rc1.pdf
Change Log

ltpda_client_um_20rc1_030109.pdf
User manual

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 1.9.3 of the LTPDA toolbox.

ltpda_toolbox_1.9.3.zip
Version 1.9.3 of the LTPDA Toolbox

change_log_v1.9.3.pdf
Change Log

ltpda_client_um_193_141108.pdf
User manual (pdf)

- home
- Installation
- System requirements
- Downloads
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - **version_1.9.3**
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 1.9.2 of the LTPDA toolbox.

ltpda_toolbox_1.9.2.zip
Version 1.9.2 of the LTPDA Toolbox

change_log_v1.9.2.pdf
Change Log

ltpda_client_um_192_080908.pdf
User manual (pdf)

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 1.9.1 of the LTPDA toolbox. Requires MATLAB R2008a or later.

ltpda_toolbox_1.9.1.zip
Version 1.9.1 of the LTPDA Toolbox

change_log
Change Log

ltpda_client_um_191_260808.pdf
User manual (pdf)

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - **version_1.9.1**
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

ltpda_toolbox_1.9.1.zip
Version 1.9.1b of the LTPDA Toolbox

change_log.pdf
Change Log

ltpda_client_um_10_080401.pdf
User manual (pdf)

- home
- Installation
- System requirements
- Downloads
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

ltpda_toolbox_10_080401.zip
Version 1.0 of the LTPDA Toolbox

change_log.txt
Change Log

ltpda_client_um_10_080401.pdf.zip
User manual (pdf)

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - **version_1.0**
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

Training Sessions
Documents
Bugs and features
Troubleshooting

home >> Downloads >> version_1.0 >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

ltpda_toolbox_1.0rc3.zip
Version 1.0rc3 of the LTPDA Toolbox

change_log.txt
Change Log

- home
- Installation
- System requirements
- Downloads
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - **version_1.0rc3**
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

ltpda_toolbox_1.0rc2.zip
Version 1.0rc2 of the LTPDA Toolbox

ltpda_user_manual_v1rc2.pdf
User manual (pdf)

change_log.txt
Change Log

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

ltpda_toolbox_1.0rc1.zip
Version 1.0rc1 of the LTPDA Toolbox

ltpda_manual_v1rc1.pdf
User manual (pdf)

change_log.txt
Change Log

- home
- Installation
- System requirements
- Downloads
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - **version_1.0rc1**
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

ltpda_toolbox_0.99.zip
Version 0.99 of the LTPDA Toolbox

change_log.txt
Change log

ltpda_v0.99_user_manual.pdf
User manual (pdf)

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - **version_0.99**
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.5 of the LTPDA toolbox.

ltpda_toolbox_0.9.zip
Version 0.9 of the toolbox

change_log.txt

ltpda_user_manual_v0.9.pdf
User Manual (pdf)

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - **version 0.9**
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.5 of the LTPDA toolbox.

ltpda_toolbox_0.7.zip
Version 0.7 of the toolbox

change_log.txt

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.5 of the LTPDA toolbox.

ltpda_toolbox_0.5.zip
Version 0.5 of the toolbox

change_log.txt

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.4.1 of the LTPDA toolbox.

ltpda_toolbox_0.4.1.zip
Version 0.4.1 of the toolbox

change_log.txt

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.3 of the LTPDA toolbox.

ltpda_toolbox_0.3.zip
Version 0.3 of the toolbox

change_log.txt

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - **version 0.3**
  - version 0.2a
  - version 0.2
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.2a of the LTPDA toolbox. This is an intermediate release for the Hannover tutorial.

ltpda_toolbox_0.2a.zip
Version 0.2a of the toolbox

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

Version 0.2 of the LTPDA toolbox. This version includes new classes and functionality, together with a more robust underlying framework. In addition, a prototype client/server system for storing and retrieving AOs from a central repository is included.

Most notably, there is only one version now; all mex files have been re-coded in pure MATLAB so this version will run on all platforms.

change_log.txt

ltpda_toolbox_0.2.zip
Version 0.2 of the toolbox

- home
- Installation
- System requirements
- **Downloads**
  - version_2.1rc1
  - version_2.0.1
  - version_2.0
  - version_2.0rc3
  - version_2.0rc2
  - version_2.0rc1
  - version_1.9.3
  - version_1.9.2
  - version_1.9.1
  - version_1.9.1 beta
  - version_1.0
  - version_1.0rc3
  - version_1.0rc2
  - version_1.0rc1
  - version_0.99
  - version 0.9
  - version 0.7
  - version 0.5
  - version 0.4.1
  - version 0.3
  - version 0.2a
  - **version 0.2**
  - version 0.1
- File repository
- Release Schedule
- User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

This is very much a pre-alpha release just to let people see how things are developing. A lot of functionality already exists and the parsing SIMULINK interface is working with about 70% of the functionality of the underlying toolbox. The direct SIMULINK interface is currently not part of this distribution.

ltpda_macintel_0.1.zip
LTPDA Toolbox for Mac Intel. Compiled on OS X 10.4 with MATLAB R20007a.

ltpda_linux_0.1.zip
LTPDA Toolbox for Linux. Compiled on FC6 with MATLAB R2007a.

ltpda_windows_0.1.zip
LTPDA Toolbox for Windows. Compiled on Windows XP with MATLAB R2007a and Visual C++ Studio Express.

home
Installation
System requirements
Downloads
version_2.1rc1
version_2.0.1
version_2.0
version_2.0rc3
version_2.0rc2
version_2.0rc1
version_1.9.3
version_1.9.2
version_1.9.1
version_1.9.1 beta
version_1.0
version_1.0rc3
version_1.0rc2
version_1.0rc1
version_0.99
version 0.9
version 0.7
version 0.5
version 0.4.1
version 0.3
version 0.2a
version 0.2
version 0.1
File repository
Release Schedule
User manual

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
mfiles
pipelines
objects
Movies
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

LTPDA example script files.

example_1.m
Example 1

example_2.m
Example 2

test_abs.m
Use the absolute value method of the AO class.

test_ao_cpsd.m
Estimate the cross-spectral density of two AOs.

test_ao_downsample.m
Downsample a time-series AO.

test_ao_interp.m
Interpolate a time-series AO.

test_ao_join_ts.m
Join multiple time-series AOs.

test_ao_polyfit.m
Fit a polynomial to a data series.

test_ao_pwelch.m
Make a power-spectral density estimate of a time-series.

test_ao_select.m
Select particular samples of an AO.

test_ao_cohere.m
Compute the coherence between two AOs.

test_ao_curvefit.m
Do non-linear fitting of data.

test_ao_upsample.m
Update time-series data.

test_ao_waveform.m
Construct AOs from waveforms.

test_ao_split_frequency.m
Split an AO by frequency.

test_ltpda_cpsd.m
Compute the cross-spectral density of two AOs.

test_ltpda_lincom.m
Make a linear combination of AOs.

test_iir_filtering.m
Do some digital filtering with IIR filters.

test_ltpda_tfe.m
Compute transfer function estimates between AOs.

home >> File repository >> mfiles >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
mfiles
pipelines
objects
Movies
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

LTPDA Workbench pipelines

- test_ao_fft.lwb
- test_ao_xcorr.lwb
- test_ao_tfe.lwb
- test_abs.lwb
- test_ao_mean.lwb
- test_iir_filtering.lwb
- test_fir_filtering.lwb
- test_ao_conj.lwb
- test_ao_downsample.lwb
- test_ao_split_freq.lwb
- test_ao_split.lwb
- test_lpsd.lwb
- test_lincom_cdata.lwb
- test_ao_resample.lwb
- test_ao_find.lwb
- test_ao_ctranspose.lwb
- test_ao_polyfit.lwb
- example_1.lwb
- test_ao_psd.lwb
- example_2.lwb
- test_ao_interp.lwb
- test_ao_detrend.lwb

home >> File repository >> pipelines >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
mfiles
pipelines
objects
Movies
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

home >> File repository >> objects >>

LTPDA

a MATLAB© toolbox for accountable and reproducible data analysis

home
Installation
System requirements
Downloads
File repository
mfiles
pipelines
objects
Movies
Release Schedule
User manual
Training Sessions
Documents
Bugs and features
Troubleshooting

home >> File repository >> Movies >>

Some thoughts on data analysis
S2-AEI-TN-3037
DRAFT 0.2 2006/11/30

G. Heinzel, M. Hewitson, A. Monsky
AEI Hannover

December 1, 2006

1 Introduction

In the context of the recently started 'LTP data analysis meetings', we discussed some issues of implementing data structures and algorithms for LTP. Some of the thoughts are presented here. They are clearly not final but merely an input for discussion and refinement.

Figure 1 shows a typical result of the data analysis that is done now at AEI Hannover for data from the LTP EM. Although this does not include all possibilities and complications forseen for the real LTP data, it will nevertheless be used as an example here to show some typical properties.

The core algorithms that were used to produce Figure 1 are working already, but need to be re-implemented in an easily accessible and well-documented way.

The most important difference between our present way of analysing data and what is needed for LTP is what we call 'traceability' and 'reproduceability'. What we mean by that is that any member of the LTP team should be able to do the following when she or he receives an analysis result like Figure 1:

- Find out all steps that were involved in the production of the final graph in human-readable form. This includes:
  - which raw data was involved (date, channel, time segment, time of retrieval if data can be changed later by new downlinks)
  - all operations on the data
  - The above for all channels of a multi-channel plot (in the example, one older 'reference' spectrum).
- Re-do all operations provided he has access to the raw data and standard tools. One point to be discussed is how 'automatic' this needs to be.
- Re-plot the results with different plot properties (colors etc)

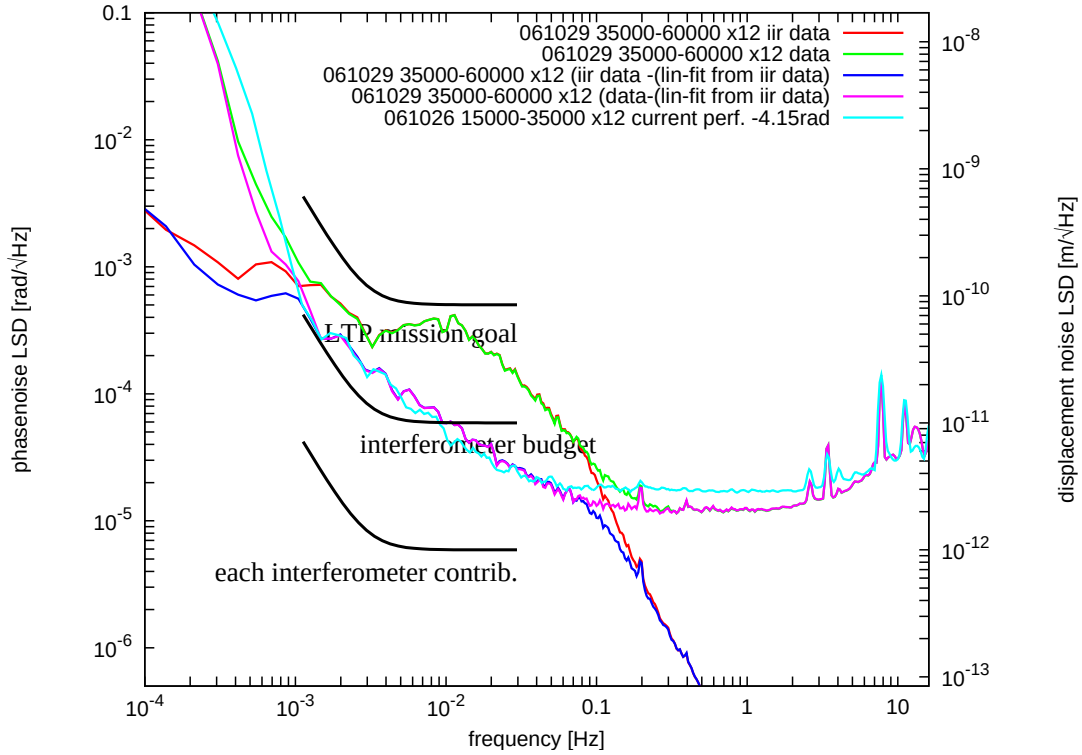


Figure 1: Typical data analysis result of the ongoing EM experiments.

One idea coming up in our discussions was the concept of an 'analysis object (AO)'. An AO would be a single file that holds all data to fulfill the above requirements. In particular, for the case of the example Figure 1, it would hold:

- The numerical data of the 5 different spectra that are plotted
- A standardized description of each of the 5 spectra (such as "spectrum with log f-axis", frequency range, unit of y axis etc.)
- The full processing history.

All algorithms would operate on AOs only, taking one or more AOs as input and producing as output one (or in exceptional cases more than one) new AOs.

In doing so, the algorithm in question operates on the data and writes the results in the new AO. The input data is not copied in the new AO. The history of the input AOs is copied to the output AO, and the present algorithms appends an appropriate section describing its own action and parameters.

Exceptions to the rule that all operations have AOs both as input and as output are:

- The initial creation of an AO from raw data
- the final plotting operation of an AO

We want to avoid the situation where plots float around without the information precisely what was done to which data. That is why we propose to have as units to be sent around between members of various teams the final AOs and not plots. The recepient can then plot the object but also study its history or redo the analysis, possibly with different parameters.

One open question is how to include the history information into the plot itself (a PDF file, for example) which will invariably also be produced and sent around.

2 An example

Figure 2 gives an overview and Figure 3 shows in detail the various processing steps that were performed to arrive at Figure 1. These are only representative examples for the purpose of this discussion. The details are neither complete nor necessarily correct.

The various numerical experiments that were necessary to arrive at this particular procedure and set of parameters (e.g. a time domain plot to select the appropriate time segment) are *not* recorded in this structure. If necessary, it must be recorded by another mechanism that records all activities of a user.

Also shown are the intermediate results which are also AOs. These need not necessarily be saved.

What needs to be saved in each AO is, however, its 'history', i.e. a description of all steps above it that led to its creation and are detailed enough to allow reproducing the same AO including all its data from only raw data and the history information in the AO.

Various types of AO can be seen in Figure 3:

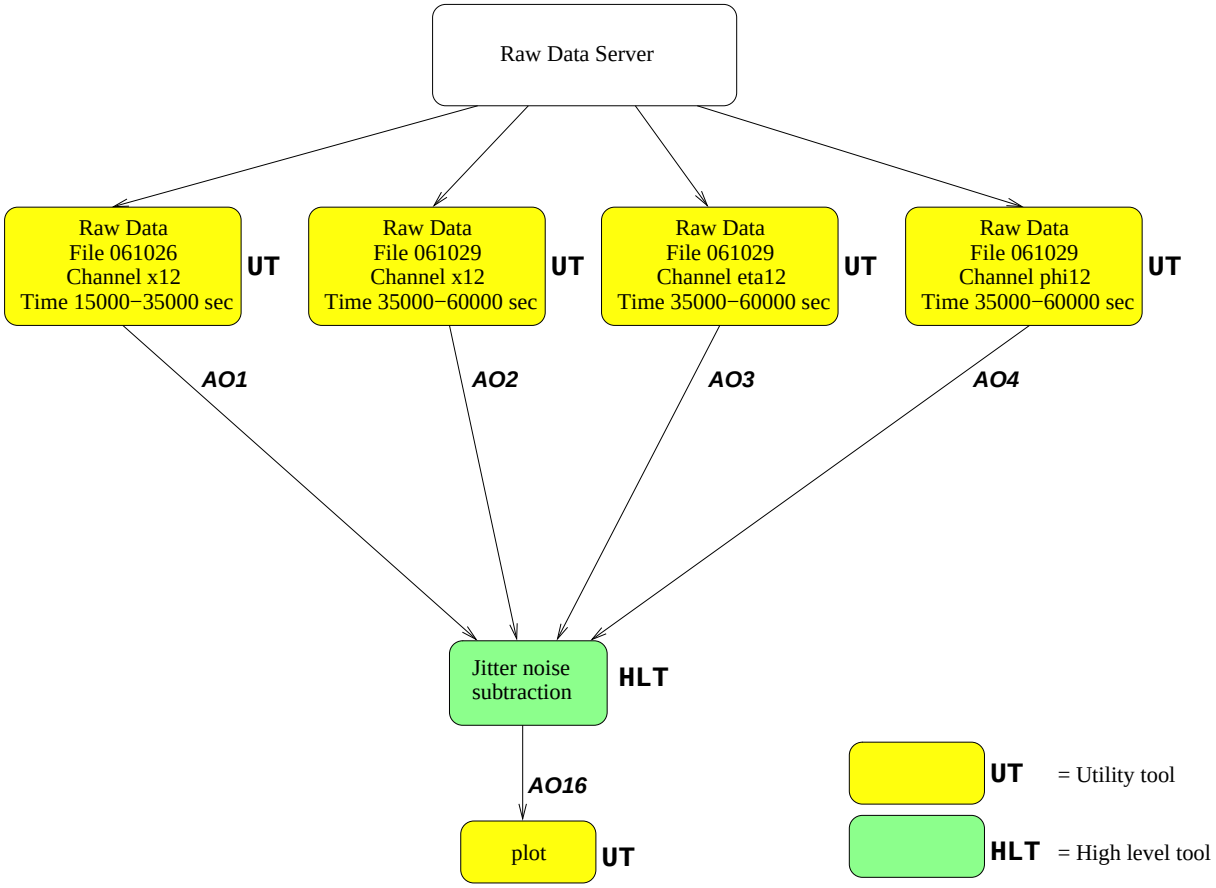


Figure 2: Simplified processing steps for the example of Figure 1.

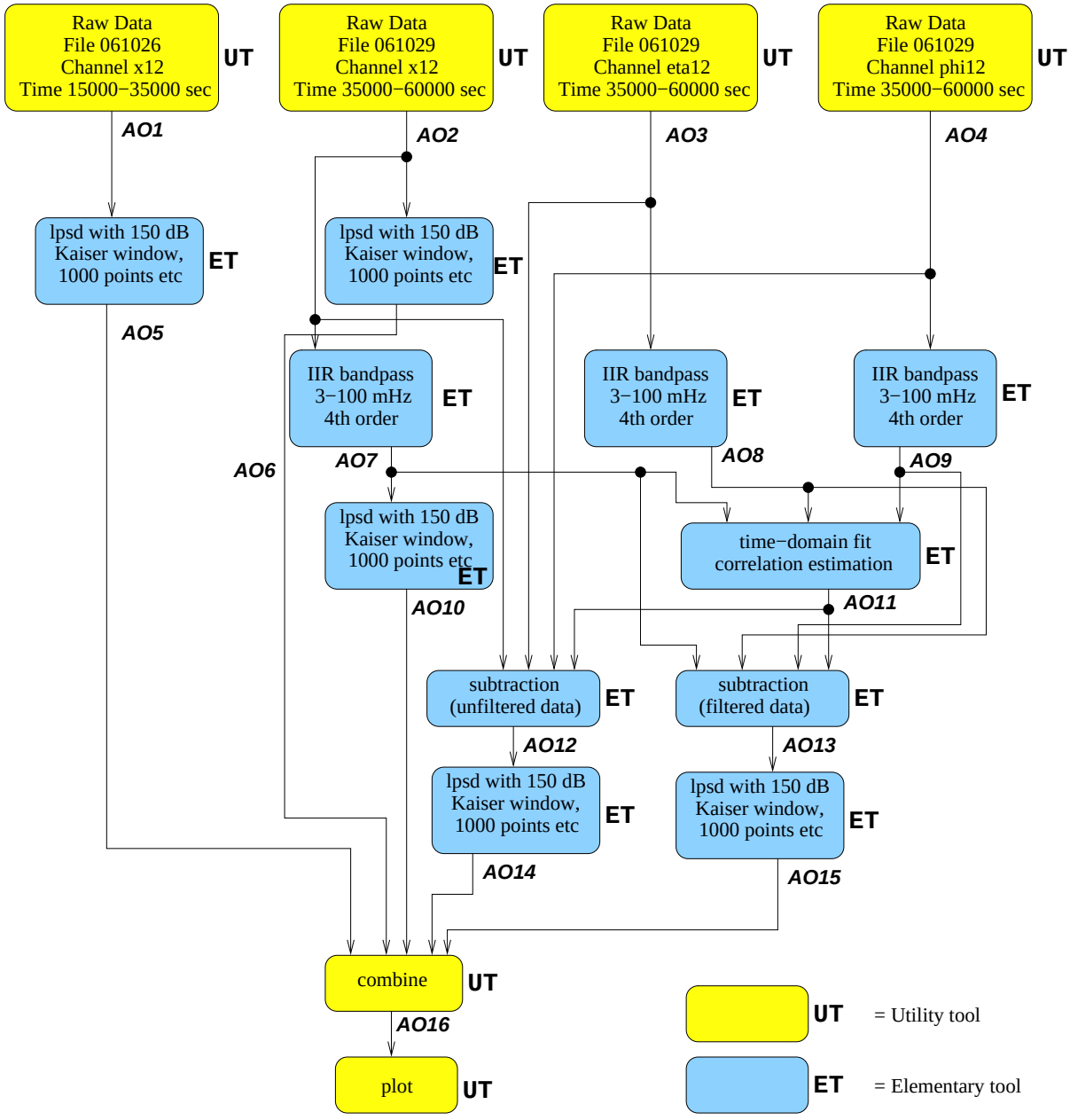


Figure 3: Detailed processing steps for the example of Figure 1.

- AO1, AO2, AO3 and AO4 are single-channel time series directly created from raw data. Their 'history' is very short and contains only the details of their creation (channel number, time segment etc.).
- AO5, AO6, AO10, AO14, AO15 are single-channel spectra (from lpsd in this case).
- AO7, AO8, AO9, AO12, AO13 are processed single-channel time series. Their standardized description must reflect the fact that they are single-channel time series and also properties such as sampling rate, length, absolute time, y unit etc. In some cases of processing it is, however, not possible any more to attach a channel name to such a processed time series. In addition to this standardized description, each of these AO must, of course, also contain its history.
- AO11 is unusual. It is neither a time series nor a spectrum but a set of correlation coefficients for the purpose of a subsequent time-domain noise subtraction algorithm. It is not clear how such information should be stored; for the time being I assume it is also an AO.
- AO16 is a multi-channel object containing 5 spectra

3 Analysis objects

The first task will be to arrive at a useful definition and implementation of AOs. Many possibilities exist that have not yet been analyzed in their consequences. In the example above, one AO can contain multiple channels. Such channels could be called 'data objects', for example.

Questions:

- Does it make sense to combine un-comparable data in one AO? Examples are time series from different times (probably yes), time series and spectra (probably no).
- How to save an AO? Some possibilities are MATLAB binary format, ASCII format or our own binary format. A preliminary promising candidate for saving AOs on disk is the XML format, which is ASCII based and hence human-readable, well-defined and flexible.

Apart from the algorithms themselves (which read and create AOs), at least the following special procedures must be available:

- Creating an AO from raw data
- printing the history of an AO in human-readable form
- dumping a complete AO in ASCII format for transmission to people without access to our special software, other plotting programs etc.
- our generic plotting tool that should ideally be a one-click operation on an AO.

Plotting may not make sense for all types of AO (i.e. very long unfiltered time series, correlation coefficients).

4 Algorithms

Algorithms work on AOs or parts thereof and produce new AOs. They may have multiple inputs (e.g. two time series to compute a correlation). They must know which type of input they require and be able to verify that an input AO has the appropriate type and properties (e.g. two time series must be from the same time for a correlation to be sensible).

If the input is a multi-channel AO, the user must select one channel out of the possibilities. Instead of adding the "select an input channel" functionality to each elementary algorithm, a better possibility would be to have an 'extract' utility algorithm which takes one channel out of a multi-channel AO. Then the algorithms could be simpler in so far as an algorithm with a single-channel input only accepts single-channel AOs as input. Extraction of one channel out of a multi-channel AO would then be another algorithm instead of the first step of all algorithms.

It is probably best to have algorithms as simple and elementary as possible.

Combinations of elementary algorithms that are executed many times can be combined into and saved as a procedure, which we call 'high level algorithm'. An example would be 'standardized spectral density' which includes some preprocessing and the parameters (Blackman-Harris window, 50 % overlap etc.) hardwired.

The toolbox to be developed thus contains three classes of algorithms:

Elementary algorithms such as IIR filters, arithmetic functions, FFT etc.

Utility algorithms such as create an AO from the raw data server, plot an AO, save or load AOs, combine multiple AOs into a single AO or extract parts of an AO.

High-level algorithms which are composed of elementary and utility algorithms to implements standard procedures such as 'standardized spectrum', 'noise subtraction' etc.

One possibility to handle the algorithms in a user-friendly way would be to use the SIMULINK graphical user interface, with the underlying functions being programmed in MATLAB.

5 History

From Figure 3 it is clear that in general the history of an AO may have a complicated tree structure. In multi-channel AOs it is probably best to keep the history of each channel tightly connected to that channel, to allow easy extraction of one channel from the file.

Each algorithm operating on an AO appends a description of its action to the history. This needs to contain

- Name of algorithm, Version number, time of running, etc.
- all variable parameters and options of the algorithm, preferably as the same set the the user sees and can change
- User comment?

6 Conclusion

Clearly there are many possiblities and options. If the structure of an AO is correctly chosen, the implementation of algorithms will be easy, but the opposite is also true.

Therefore it is important to look at typical analysis situations and see which type of AO structure would be best suited.

Change Log for LTPDA Toolbox v2.0.1

M Hewitson 08-03-09

Introduction

This version of LTPDA is V2.0.1.

This is a very minor update on V2.0. Mostly changes to the documentation, particularly those sections for the training session.

Detailed changes since v2.0

2009-03-08 12:46 hewitson

* m-toolbox/html_help/help/ug/:
  ltpda_training_topic_1_2_content.html,
  ltpda_training_topic_1_3_content.html,
  ltpda_training_topic_1_6_content.html,
  ltpda_training_topic_2_3_content.html,
  ltpda_training_topic_3_2_3_content.html,
  ltpda_training_topic_3_5_content.html,
  ltpda_training_topic_3_6_content.html,
  images/ltpda_training_1/topic3/TFE_in_out.png,
  images/ltpda_training_1/topic3/TFE_result_1.png: Small edits and
  fixes.

2009-03-08 12:17 miquel

* m-toolbox/html_help/help/ug/:
  ltpda_training_topic_4_4_2_content.html,
  images/ltpda_training_1/topic4/BandpassEx4.png: corrections

2009-03-08 11:52 hewitson

* m-toolbox/test/LTPDA_training/topic2/: mk_testdata.m, whiten.mat:
  Slight change of the model and output a MAT file because
  otherwise we hit the java heap problem with big XML files.

2009-03-08 09:48 hewitson

* m-toolbox/classes/@ao/iplot.m: Bug fix in xyz plot. Don't
  understand why this didn't show up earlier!

2009-03-07 21:57 miquel

* m-toolbox/html_help/help/ug/ltpda_training_topic_4_1_1_content.html: more corrections

2009-03-07 16:41 hewitson

* m-toolbox/test/LTPDA_training/ifo_temp.lwb: A full workbench for the IFO/Temp examples. Topics 1-5.

2009-03-07 11:18 hewitson

* m-toolbox/html_help/help/ug/: ltpda_training_topic_3_1_content.html, ltpda_training_topic_3_2_1_content.html, ltpda_training_topic_3_2_2_content.html, ltpda_training_topic_3_2_3_content.html: A bunch of small edits and typo fixes.

2009-03-07 11:02 hewitson

* m-toolbox/html_help/help/ug/ltpda_training_topic_2_3_content.html: Some fixes to the text describing resample.

2009-03-06 19:11 nicola

* m-toolbox/test/aorepo_proto_test/repo_test_rand_objs.m: Now reporting more data.

2009-03-06 19:10 nicola

* m-toolbox/test/aorepo_proto_test/repo_test_func.m: Update to work with parfrac objects.

2009-03-06 16:58 luigi

* m-toolbox/html_help/help/ug/: images/ltpda_training_1/topic5/ex4_chirpsine.gif, ltpda_training_topic_5_4_content.html: added an equation for topic 5 ex4

2009-03-06 15:36 mauro

* m-toolbox/html_help/help/ug/: ltpda_training_topic_3_5_content.html, ltpda_training_topic_3_6_content.html: Typo fixed

2009-03-06 15:36 mauro

* m-toolbox/html_help/help/ug/sigproc_intro_content.html: Updated moving relevant definitions at the top

2009-03-06 14:59 miquel

* m-toolbox/html_help/help/ug/ltpda_training_topic_4_5_content.html: correction

2009-03-06 14:14 miquel

* m-toolbox/html_help/help/ug/: images/ltpda_training_1/topic4/lfo2TempEx_T4_1.png, images/ltpda_training_1/topic4/lfo2TempEx_T4_3.png, images/ltpda_training_1/topic4/lfo2TempEx_T4_5.png, images/ltpda_training_1/topic4/lfo2TempEx_T4_6.png, ltpda_training_topic_4_5_content.html, images/ltpda_training_1/topic4/lfo2TempEx_T4_4.png: corrections and expanded explanations in the ifo/T example. Set names in figures

2009-03-06 13:44 amattioli

* testing/stp/SYSTEM_TESTS_V2/DEVEL/test/: datasetDemo.pl, functionDemo.pl, inputDemo.pl: Prolog engine test file for single execution.

2009-03-06 13:40 amattioli

* testing/stp/SYSTEM_TESTS_V2/DEVEL/singleRun.sh: BASH script for a single Prolog execution.

2009-03-06 12:42 mauro

* testing/utp_1.1/utps/ao/utp_ao_lcpsd.m: UTP updated with tests on: - CPSD(x,x) == PSD(x) - CPSD(x,y) == conj(CPSD(y,x)) So far this last test is failing, investigations are in progress

2009-03-06 12:37 mauro

* testing/utp_1.1/utps/ao/utp_ao_cpsd.m: Minor fix

2009-03-06 11:41 mauro

* m-toolbox/classes/@ssm/ssm.m: Fixed help for 'built-in' option

2009-03-06 11:20 miquel

* m-toolbox/html_help/help/ug/:
  ltpda_training_topic_4_3_content.html,
  ltpda_training_topic_4_4_1_content.html,
  images/ltpda_training_1/topic4/ClosedLoop_G_delay.png,
  images/ltpda_training_1/topic4/ClosedLoop_Gd_delay.png,
  images/ltpda_training_1/topic4/ClosedLoop_H_delay.png,
  images/ltpda_training_1/topic4/ClosedLoop_Hd_delay.png,
  images/ltpda_training_1/topic4/ClosedLoop_OLG_delay.png,
  images/ltpda_training_1/topic4/ClosedLoop_OLGd_delay.png:
  corrections to topic 4

2009-03-06 10:53 amattioli

* testing/stp/SYSTEM_TESTS_V2/DEVEL/m_scripts/createPrologDB.m:
  Matlab entry point for the bash 'run' script.

2009-03-06 10:46 amattioli

* testing/stp/SYSTEM_TESTS_V2/DEVEL/postProcessing.perl: PERL
  postprocessing file: converts an intermediate file generated with
  the Prolog engine to a Matlab executable test suite with profile
  information.

2009-03-06 10:26 mauro

* m-toolbox/test/test_specwin_nenbw_definition.m: Test script
  comparing definitions of nenbw for all specwins

2009-03-06 10:24 amattioli

* testing/stp/SYSTEM_TESTS_V2/DEVEL/run.sh: BASH run script:
  cleans, calls Matlab functions extractor, runs prolog, calls perl
  postprocessing function.

2009-03-06 10:16 amattioli

* testing/stp/SYSTEM_TESTS_V2/DEVEL/pl_engine/engine-main.pl:
  Prolog engine for functions' composition used during system tests
  v2.

2009-03-06 10:11 anneke

*
  m-toolbox/html_help/help/ug/ltpda_training_topic_2_2_content.html:
  typos

2009-03-06 09:10 mauro

* testing/utp_1.1/utps/ao/utp_ao_cpsd.m: UTP updated with tests on:
  - CPSD(x,x) == PSD(x) - CPSD(x,y) == conj(CPSD(y,x))

2009-03-06 09:09 mauro

* m-toolbox/classes/@ao/psd.m: Changed to set the Equivalent Noise
  Bandwidth from the value calculated from the window properties,
  rather than taking it from the default. - The numerical
  difference is well below 0.1% - The set value is now the one
  actually used by the PS and AS estimators

2009-03-05 19:23 miquel

* m-toolbox/test/LTPDA_training/topic4/LISOfile.fil: LISO file
  corrected

2009-03-05 19:04 hewitson

* m-toolbox/m/gui/ltpdv/callbacks/ltpdv_server_get_data.m: Fixed to
  use the new ao/fromGEOserver constructor. The released version
  doesn't work because it relied on addHistory being public and we
  didn't test this GUI after that change :(

2009-03-05 19:03 hewitson

* m-toolbox/m/ao_models/ao_model_mdc1_fd_i1o12.m: Typo in help.

2009-03-05 19:02 hewitson

* m-toolbox/classes/@plist/parse.m: Typo.

2009-03-05 19:02 hewitson

* m-toolbox/classes/@ao/fromGEOserver.m: Bug fix to make the 'get
  latest' feature work properly

2009-03-05 17:53 nicola

* m-toolbox/test/aorepo_proto_test/: repo_test_func.m,
  repo_test_rand_objs.m: Mostly bug fixes.

2009-03-05 17:31 luigi

* m-toolbox/html_help/help/ug/: ng2D_content.html,
  whiten2D_content.html: typo correction

2009-03-05 17:24 miquel

* m-toolbox/html_help/help/ug/:
  ltpda_training_topic_4_4_1_content.html,
  ltpda_training_topic_4_4_2_content.html,
  ltpda_training_topic_4_5_content.html,
  images/ltpda_training_1/topic4/IFO2TSimulation.png: typos and an
  explanatory scheme added

2009-03-05 17:20 luigi

* m-toolbox/html_help/help/ug/:
  ltpda_training_topic_5_1_content.html,
  ltpda_training_topic_5_2_content.html,
  ltpda_training_topic_5_3_content.html,
  ltpda_training_topic_5_4_content.html,
  ltpda_training_topic_5_5_content.html: typo corrections for topic
  5

2009-03-05 16:38 nicola

* m-toolbox/test/aorepo_proto_test/repo_test_func.m: Uodate to
  handle also parfrac and rational objects.

2009-03-05 16:31 miquel

* m-toolbox/html_help/help/ug/:
  ltpda_training_topic_4_1_1_content.html,
  ltpda_training_topic_4_1_2_content.html,
  ltpda_training_topic_4_1_content.html,
  ltpda_training_topic_4_2_content.html,
  ltpda_training_topic_4_3_content.html: typos and corrections

2009-03-05 14:49 mauro

* m-toolbox/html_help/help/ug/sigproc_intro_content.html: Some
  clarifications

2009-03-05 14:48 mauro

*
  m-toolbox/html_help/help/ug/ltpda_training_topic_1_2_content.html:
  Corrected typo

2009-03-05 14:48 mauro

* m-toolbox/html_help/help/ug/:

ltpda_training_topic_3_1_content.html,
ltpda_training_topic_3_2_1_content.html,
ltpda_training_topic_3_2_2_content.html,
ltpda_training_topic_3_2_3_content.html,
ltpda_training_topic_3_2_content.html,
ltpda_training_topic_3_content.html: Updated: - typos - some clarifications here and there

2009-03-05 13:50 ingo

* m-toolbox/classes/: @plist/plist.m, @ssm/ssm.m: MANTIS bug 0000309: Resolve constructor commands to get an object from the repository. Prpblem in plist: plist with key-word 'hostname' (UTPs used always 'conn') ssm: didn't use the inherited fromRepository method.

2009-03-05 12:46 ingo

* m-toolbox/classes/@unit/: mpower.m, power.m: Bug fix: The power method should only raise the exponent and not the value

2009-03-05 12:10 mauro

* m-toolbox/test/high_level_queries/: make_test_data.m, test_high_level_query.m: Updated for the new toolbox Testing just resumed

2009-03-05 11:51 ingo

* m-toolbox/classes/@ao/: ge.m, gt.m, le.m, lt.m: Bug fix: This method accepts now a plist as an input (but do nothing with it) because the GUI uses at least one empty plist as an input.

2009-03-05 11:48 ingo

* m-toolbox/classes/@pz/pz.m: Bug fix: The command pz({1 2 3}) created twice as much of pz-objects. The first three objects were empty.

2009-03-05 10:05 mauro

* testing/utp_1.1/utps/ao/: utp_ao_cohere.m, utp_ao_cpsd.m, utp_ao_lcohere.m, utp_ao_lcpsd.m, utp_ao_ltfe.m, utp_ao_tfe.m: UTP update: added test on units

2009-03-04 17:10 mauro

* testing/utp_1.1/utps/ao/utp_ao_lpsd.m: Added test on units

2009-03-04 17:02 luigi

* m-toolbox/html_help/help/ug/: sdomainfit_content.html, zdomainfit_content.html: content help updated for zDomainFit and sDomainFit

2009-03-04 13:43 luigi

* m-toolbox/html_help/help/ug/images/: sigproc_diff_algo01.gif, sigproc_diff_algo02.gif, sigproc_diff_algo03.gif, sigproc_diff_algo04.gif: changed figures background color

2009-03-04 13:24 luigi

* m-toolbox/html_help/help/ug/sigproc_diff_content.html: corrected backgroud color of first table

2009-03-04 13:21 luigi

* m-toolbox/html_help/help/ug/sigproc_diff_content.html: removed a repetition of the ORDER2SMOOTH field

2009-03-04 13:07 luigi

* m-toolbox/html_help/help/ug/sigproc_diff_content.html: corrected second figure number

2009-03-04 13:07 gerrit

* repository/sync.inc.php: without conflicts

2009-03-04 13:03 luigi

* m-toolbox/html_help/help/ug/sigproc_diff_content.html: some corrections

2009-03-04 12:13 luigi

* m-toolbox/html_help/help/helptoc.xml: helptoc.xml updated

2009-03-04 12:10 luigi

* m-toolbox/html_help/help/ug/sigproc_diff_content.html: corrected a bug, pyton script needs some tex at the beginning of the document

2009-03-03 18:44 luigi

* m-toolbox/html_help/help/helptoc.xml: helptoc.xml updated

2009-03-03 18:43 luigi

* m-toolbox/html_help/help/ug/sigproc_content.html: updated

2009-03-03 18:42 luigi

* m-toolbox/html_help/help/ug/sigproc_diff_content.html:
  sigproc_diff.html help page

2009-03-03 18:41 luigi

* m-toolbox/html_help/help/ug/images/: sigproc_diff_algo01.gif,
  sigproc_diff_algo02.gif, sigproc_diff_algo03.gif,
  sigproc_diff_algo04.gif, sigproc_diff_algo05.png,
  sigproc_diff_algo06.png: images for sigproc_diff help page

2009-03-03 17:13 nicola

* m-toolbox/classes/+utils/@math/randelement.m: Updated function
  help.

2009-03-02 18:43 ingo

* m-toolbox/test/LTPDA_training/TrainingSessionAll.m: Update topic
  4

2009-03-02 18:35 mauro

* m-toolbox/test/aorepo_proto_test/repo_test_rand_objs.m: Not used
  in Training session, nor in the documentation