LTPDA Toolbox

LTPDA Toolbox

LTPDA Toolbox

LTPDA Toolbox™

Functions:
- **By Category**

What's New

- **LTPDA Release Notes**
  Summarizes new features, bug fixes, upgrade issues, etc.

- **General Release Notes for LTPDA 2.6**
  For all products, highlights new features, installation notes, bug fixes, and compatibility issues

---

Documentation Set

- **Getting Started**
  Introduces the LTPDA Toolbox and gets you started using it

- **LTPDA Objects**
  Introduces LTPDA objects and related concepts

- **Constructor examples**
  Gives examples of how to construct different LTPDA objects

- **Using the LTPDA Workbench**
  Introduces the use of the LTPDA Workbench to do graphical design of data analysis pipelines

---

Product Examples and Demos

- **LTPDA Training Session 1**
  Presents a series of help pages consitute the first training session of LTPDA.

- **Constructor Examples**
  Presents a collection how to construct different LTPDA objects

- **LTPDA Demos**
  Presents a collection of demos that you can run from the Help browser to help you learn LTPDA

LTPDA Web Site Resources

- **LTPDA Homepage**
- **System requirements**
- **User manual**
- **Bug Tracking Tool**
- **Troubleshooting**
- **Extension Modules**

LTPDA Toolbox™   [contents](#)   ◀ ▶

Getting Started with the 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.

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

©LTP Team

LTPDA Toolbox™   [contents](#)

What is the LTPDA Toolbox

This section covers the following topics:

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

Overview

The LTPDA Toolbox is a MATLAB$^®$ Toolbox designed for the analysis of data from the LISA Technology Package (part of the LISA Pathfinder Mission). However, the toolbox can be used for any general purpose signal processing, and is particularly useful for analysis of typical lab-based experiments.

The Toolbox implements accountable and reproducible data analysis within MATLAB$^®$.

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

Features of the LTPDA Toolbox

The LTPDA Toolbox has the following features:

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

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

Expected Background for Users

MATLAB

This documentation assumes you have a basic working understanding of MATLAB. You need to know basic MATLAB syntax for writing your own m-files.

Getting Started with the LTPDA Toolbox   System Requirements

©LTP Team

System Requirements

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

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

Platforms

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

MATLAB and Related Products

The LTPDA Toolbox requires MATLAB. In addition, the following MathWorks Toolboxes are required for some features:

Component	Version	Comment
MATLAB	7.10 (R2010a)	
Signal Processing Toolbox	6.13	
Symbolic Math Toolbox	5.4	
Optimization Toolbox*	5.0	The optimisation toolbox is only used for a small number of methods.

©LTP Team

System Requirements (LTPDA Toolbox)

Setting-up MATLAB

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

- [Add the LTPDA Toolbox to the MATLAB path](#)
- [Starting LTPDA Toolbox](#)
- [Edit the LTPDA Preferences](#)

Add the LTPDA Toolbox to the MATLAB path

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

- In MATLAB versions earlier than **R2012b**
  File -> Set Path...
  With MATLAB version **R2012b**
  HOME-tab -> 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
```

You should see the following GUI:
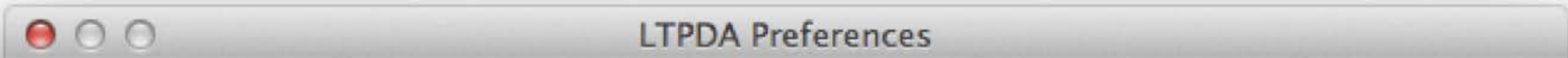
| Display | Plot | Extensions | Time | Repository | External | Misc |

Verbose Level: **PROC4** ⇕

Wrap Strings: 8 ⇕

Wrap Legend Strings: 10 ⇕

Done

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.

LTPDA Preferences: Display

LTPDA Preferences: Plot

LTPDA Preferences: Plot Styles

LTPDA Preferences: Extensions

LTPDA Preferences: Time

LTPDA Preferences: Repository

LTPDA Preferences: External

LTPDA Preferences: Misc

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"
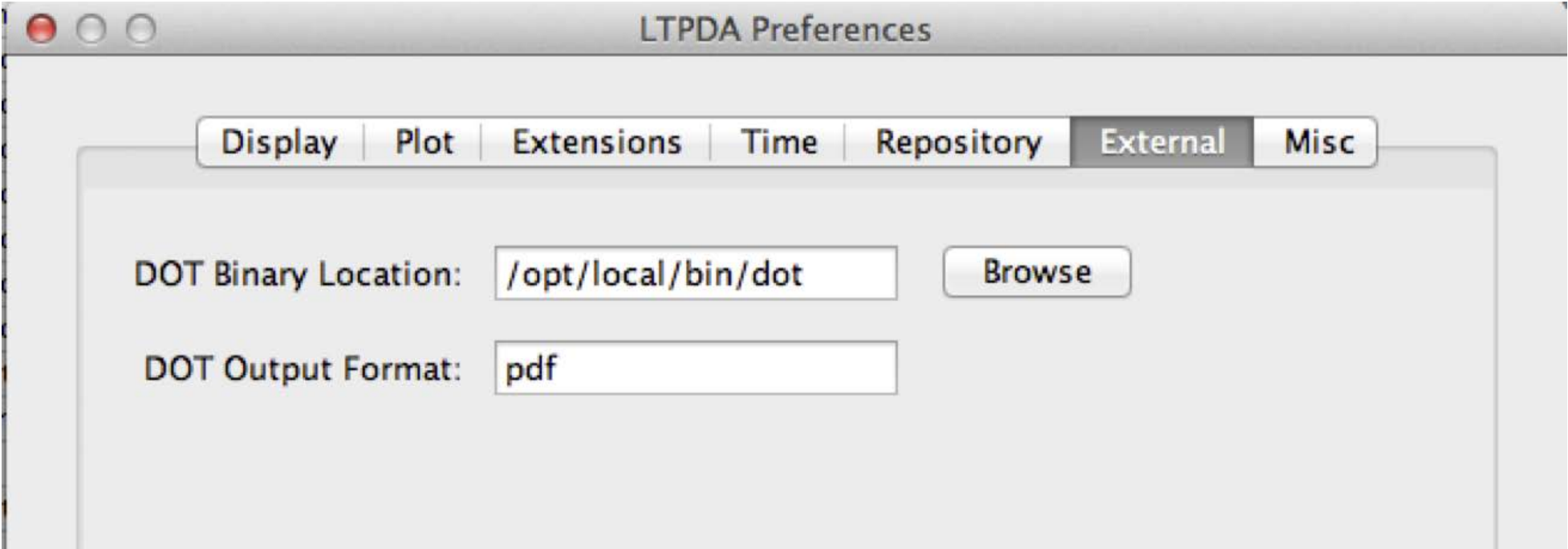
4. Set the path to the 'dot.exe' binary in the editable text field "DOT binary". If you perform the default installation, this should be something like: 'c:\Program Files\Graphviz2.28\bin\dot.exe';
5. Define in the editable text field "DOT format" the graphics format to output. See formats for available 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, choose '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 installation on Windows.
   2. Set the path to the 'dot' binary in the editable text field "DOT binary".
      - If you performed the default installation from fink, this should be something like: '/sw/bin/dot'
      - If you performed the default installation from the web page, this should be something like: '/usr/local/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, choose '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 installation on Windows.
   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, choose '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 ▶

©LTP Team

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](#)

4. **Problems to use external links in LTPDA help**

User with a Ubuntu 12.04 LTS OS may have problems to open external links because Firefox needs a much newer libstdc++ library as MATLAB delivers. You can check if this workaround apply to you by typing "**!firefox &**" in the command window. The should you get the following error massage:

```
>> !firefox &
XPCOMGlueLoad error for file /usr/lib/firefox/libxpcom.so:
libxul.so: cannot open shared object file: No such file or directory
Couldn't load XPCOM.
firefox: Signal 127
```

If so then should you use the system libstdc++ library instead of MATLAB's libstdc++ library.

This can be done by performing the following steps:
1. Identify the location of MATLAB's libstdc++ library path.
   You can do this with the linux command "**locate -i libstdc++.so.6**". MATLAB 2012a habe two libstdc++ libraries (older MATLAB versions have only one libstdc++ library) and the path are:
   $MATLABROOT/sys/os/$ARCH
   $MATLABROOT/bin/$ARCH
   where $MATLABROOT is the installation path of your MATLAB (e.g. /usr/local/MATLAB/R2012a) and $ARCH is the architecture corresponds to yours OS (e.g. glnxa64).
2. Move MATLAB's libstdc++ library to a backup folder.
   Use the following command in a ubuntu terminal.

```
cd $MATLABROOT/sys/os/$ARCH
sudo mkdir old
sudo mv libstdc++* old/
```

You should do the steps above for all libstdc++ libraries you find on MATLAB installation path.

◀ Additional 3rd-party software   Introducing LTPDA Objects ▶

©LTP Team

Introducing LTPDA Objects

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

For full details of objects in MATLAB, refer to [MATLAB Classes and Object–Oriented Programming](#).

LTPDA Classes

Various classes make up the object–oriented infrastructure of LTPDA. The figure below shows all the classes in LTPDA. All classes are derived from the base class, `ltpda_obj`. The classes then fall into two main types deriving from the classes `ltpda_nuo` and `ltpda_uo`.
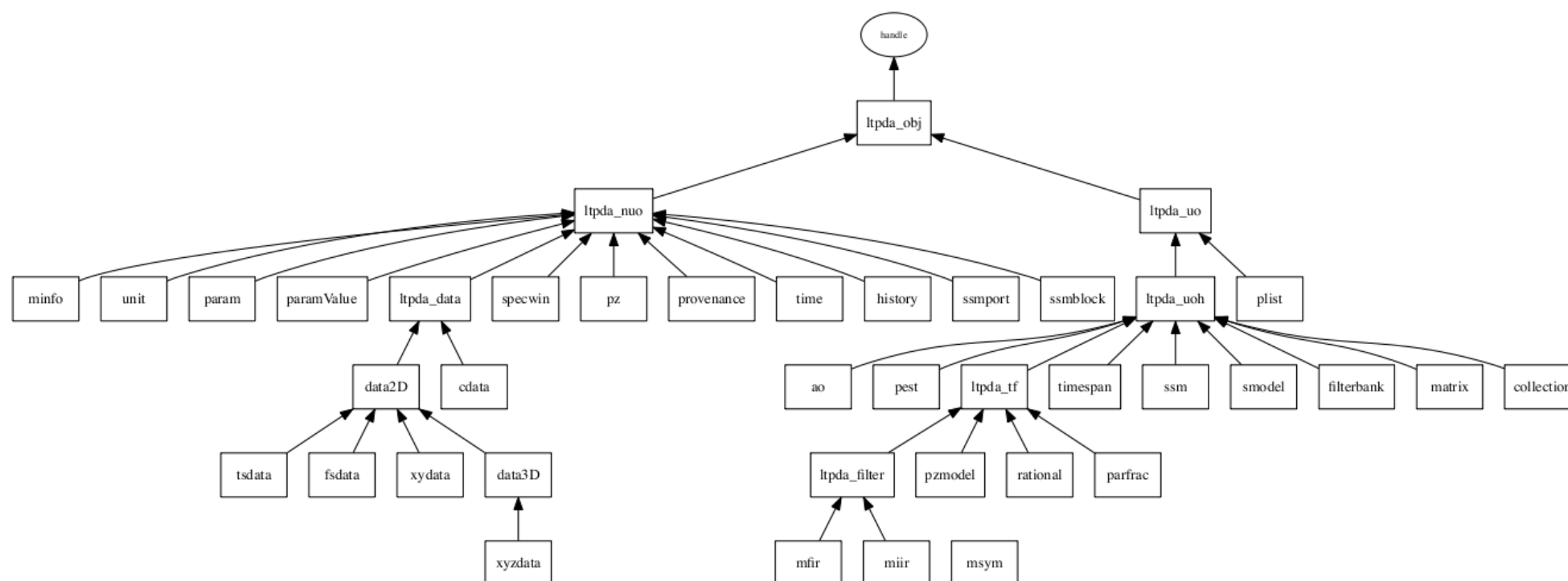
The left branch, `ltpda_nuo`, are termed 'non–user objects'. These objects are not typically accessed or created by users. The right branch, `ltpda_uo`, are termed 'user objects'. These objects have a 'name' and a 'history' property which means that their processing history is tracked through all LTPDA algorithms. In addition, these 'user objects' can be saved to disk or to an LTPDA repository.

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

Details of each class are given in:

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

©LTP Team

Types of User Objects

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

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

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

◀ Introducing LTPDA Objects   Creating LTPDA Objects ▶

©LTP Team

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.

◀ Types of User Objects	Working with LTPDA objects ➡

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Working with LTPDA objects

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

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

- [Calling object methods](#)
- [Setting object properties](#)
- [Copying objects](#)
- [Exploring objects](#)

Calling object methods

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

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

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

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

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

or

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

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

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

or

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

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

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

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

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

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

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

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

Then call the `psd` method:

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

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

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

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

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

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

Setting object properties

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

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

Reading the value of a property is achieved by:

```
>> ii.name

ans =

My Filter
```

Copying objects

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

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

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

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

ans =

My Name
```

Copying the object can be achieved using the copy constructor:

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

ans =

none
```

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

Exploring objects

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

See the LTPDA Objects Explorer GUI documentation.

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

Creating AOs from text files.

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

```
>> a = ao('data.txt')
----------- ao 01: data.txt_01_02 -----------

     name:  data.txt_01_02
     data: (0,-1.06421341288933) (0.1,1.60345729812004) (0.2,1.23467914689078) ...
              -------- tsdata 01 ------------

                    fs:  10
                     x:  [100 1], double
                     y:  [100 1], double
                    dx:  [0 0], double
                    dy:  [0 0], double
                xunits:  [s]
                yunits:  []
                 nsecs:  10
                    t0:  1970-01-01 00:00:00.000
              ------------------------------

          hist:  ao / ao / SId: fromDatafile ... $-->$Id: ao ... S
       mdlfile:  empty
   description:
          UUID:  e6ccfcb6-da49-4f4c-8c2c-3054fe5d2762
      --------------------------------------------
```

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

```
>> a = ao(plist('filename', 'data.txt'))
```
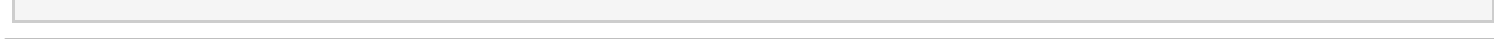
Creating AOs from XML or .mat files

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

```
>> a = ao('a.xml')
----------- ao 01: a -----------

        name:  None
        data: (0,-0.493009815316451) (0.1,-0.180739356415037) (0.2,0.045841105713705)
...
                -------- tsdata 01 ------------

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

        hist:  ao / ao / SId: fromVals ... $-->$Id: ao ... S
     mdlfile:  empty
 description:
        UUID:  2fed6155-6468-4533-88f6-e4b27bc6e1aa
        -----------------------------
```

Creating AOs from MATLAB functions

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

```
>> a = ao(plist('fcn', 'randn(1000,1)'))
----------- ao 01: a -----------

        name:  None
        data: -1.28325610460477 -2.32895451628334 0.901931466951714 -1.83563868373519
0.06675 ...
                -------- cdata 01 ------------
                     y:  [1000x1], double
                    dy:  [0x0], double
                yunits:  []
                -----------------------------

        hist:  ao / ao / SId: fromFcn ... $-->$Id ... $
     mdlfile:  empty
 description:
        UUID:  0072f8d0-f804-472b-a4aa-e9ec6a8de803
        -----------------------------
```

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

Creating AOs from functions of time

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

```
        pl = plist();
        pl = append(pl, 'nsecs', 100);
        pl = append(pl, 'fs', 10);
        pl = append(pl, 'tsfcn', 'sin(2*pi*1*t)+randn(size(t))');
        a = ao(pl)
        ----------- ao 01: a -----------

            name:  None
            data: (0,1.37694916561229) (0.1,-0.820427237640771) (0.2,1.09228819960292) ...
                  -------- tsdata 01 ------------

                      fs:  10
                       x:  [1000 1], double
                       y:  [1000 1], double
                      dx:  [0 0], double
                      dy:  [0 0], double
                  xunits:  [s]
                  yunits:  []
                   nsecs:  100
                      t0:  1970-01-01 00:00:00.000
                  ----------------------------

            hist:  ao / ao / SId: fromTSfcn ... $-->$Id: ao ... S
         mdlfile:  empty
     description:
            UUID:  c0f481cf-4bdd-4a91-bc78-6d34f8222313
        ------------------------------
```

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

Creating AOs from window functions

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

```
>> w = specwin('Hanning', 1000)
------ specwin/1 -------
    type: Hanning
   alpha: 0
    psll: 31.5
     rov: 50
   nenbw: 1.5
    w3db: 1.4382
flatness: -1.4236
      ws: 500
     ws2: 375.000000000001
     win: [0 9.86957193144233e-06 3.94778980919441e-05 8.88238095955174e-05 0.0001579
...
 version: SId: specwin.m,v 1.67 2009/09/01 09:25:24 ingo Exp S
 -----------------------

>> a = ao(w)
---------- ao 01: ao(Hanning) -----------

        name:  ao(Hanning)
        data: 0 9.86957193144233e-06 3.94778980919441e-05 8.88238095955174e-05
0.0001579 ...
              -------- cdata 01 ------------
                   y:  [1x1000], double
                  dy:  [0x0], double
              yunits:  []
              ----------------------------

        hist:  ao / ao / SId: fromSpecWin ... $-->$Id: ao ... S
     mdlfile:  empty
 description:
        UUID:  ea1a9036-b9f5-4bdb-b3a3-211e9d697060
    ----------------------------------------
```

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

```
>> ao(plist('win', 'Hanning', 'length', 1000))
----------- ao 01: ao(Hanning) -----------

       name:  ao(Hanning)
       data: 0 9.86957193144233e-06 3.94778980919441e-05 8.88238095955174e-05
0.0001579 ...

                -------- cdata 01 ------------
                    y:  [1x1000], double
                   dy:  [0x0], double
                yunits:  []
                ----------------------------

       hist:  ao / ao / SId: fromSpecWin ... -->$Id: ao ... S
    mdlfile:  empty
description:
       UUID:  5b81f67a-45b9-43f8-a74a-bc5161fd718f
-----------------------------------------
```

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

Creating AOs from waveform descriptions

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

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

asaw = ao(pl)
----------- ao 01: Sawtooth -----------

       name:  Sawtooth
       data: (0,-1) (0.01,-0.96) (0.02,-0.92) (0.03,-0.88) (0.04,-0.84) ...
                -------- tsdata 01 ------------

                   fs:  100
                    x:  [500 1], double
                    y:  [500 1], double
                   dx:  [0 0], double
                   dy:  [0 0], double
                xunits:  [s]
                yunits:  []
                 nsecs:  5
                   t0:  1970-01-01 00:00:00.000
                ----------------------------

       hist:  ao / ao / SId: fromWaveform ... $-->$Id: ao ... S
    mdlfile:  empty
description:
       UUID:  cb76c866-ee3f-47e6-bb29-290074666e43
-------------------------------------
```

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

```
iplot(asaw);
```
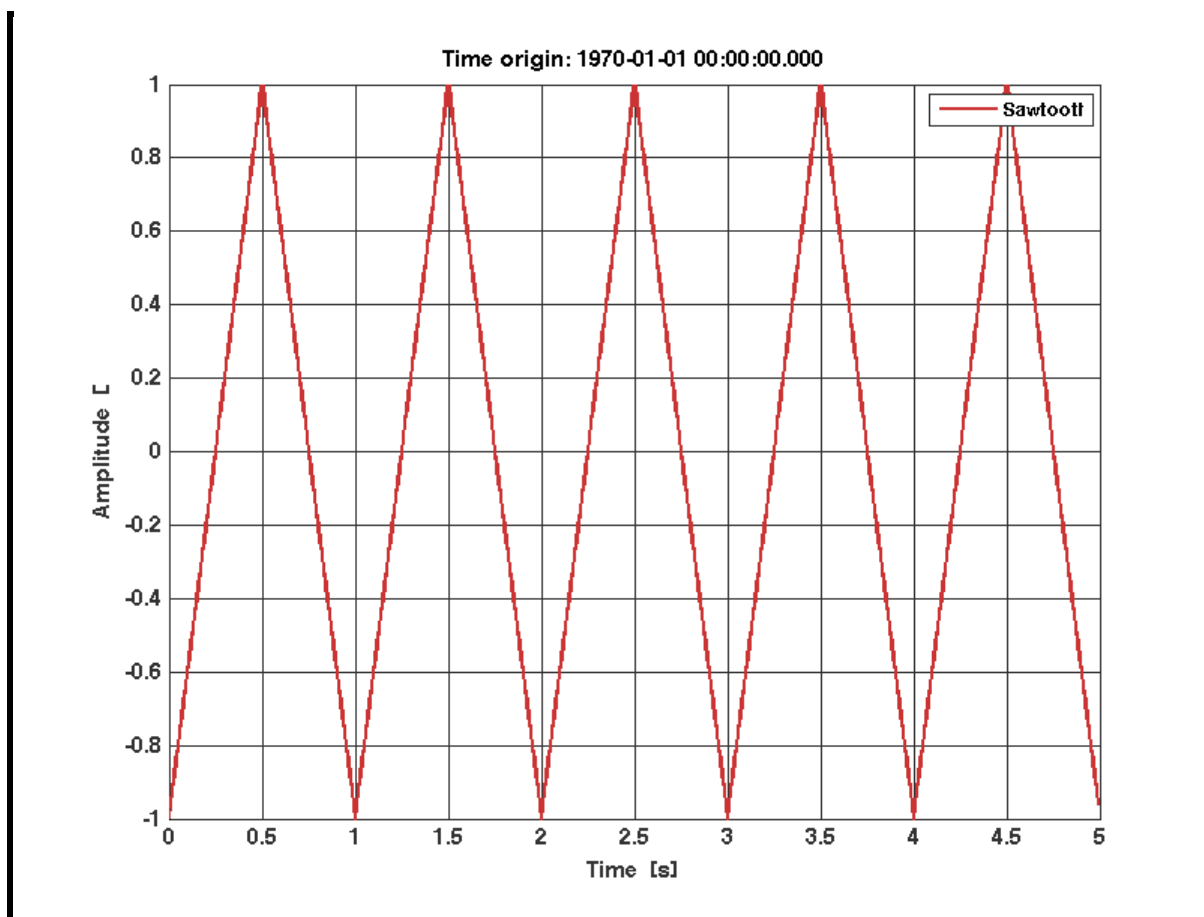
Creating AOs from pole zero models

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

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

The Document *Generation of Random time series with prescribed spectra* by Gerhard Heinzel (S2-AEI-TN-3034)
corrects a mistake in the aforesaid paper and describes the practical implementation. The following code creates an AO with a time series having a prescribed spectral density, defined by the input pole zero model:

```
        f1    = 5;
        f2    = 10;
        f3    = 1;
        gain  = 1;
        fs    = 10;   %sampling 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)
        ----------- ao 01: noisegen(None) -----------

             name:  noisegen(None)
             data: (0,9.20287001568168) (0.1,-3.88425345108961) (0.2,6.31042718242658) ...
                   -------- tsdata 01 ------------

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

```
                      dy:  [0 0], double
                  xunits:  [s]
                  yunits:  []
                   nsecs:  100
                      t0:  1970-01-01 00:00:00.000
              -----------------------------

           hist:  ao / ao / SId: fromPzmodel ... $-->$Id: ao ... S
        mdlfile:  empty
    description:
           UUID:  4a89d910-8672-475f-91cd-4fcc4b52a6b4
        --------------------------------------------
```

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

```
        iplot(a);
```

©LTP Team
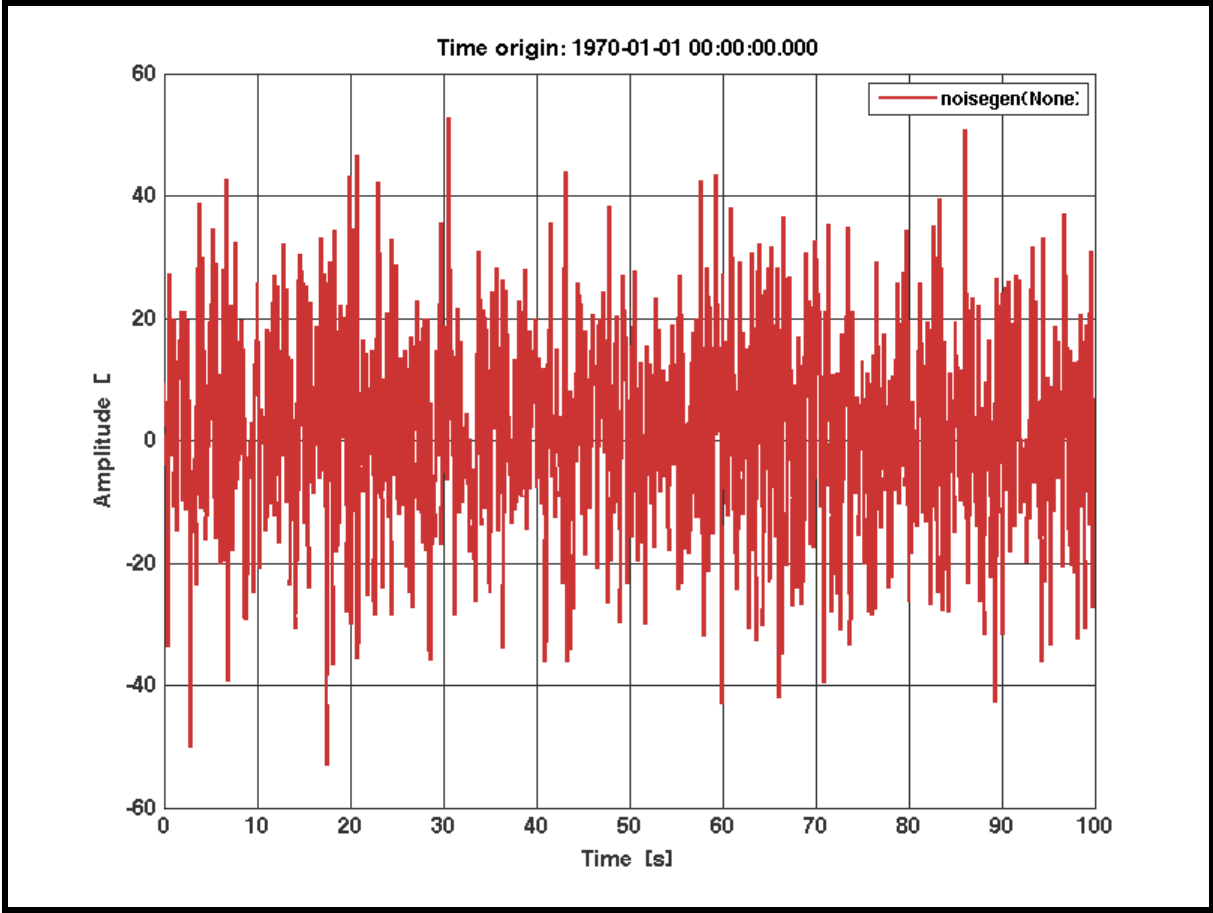
Saving Analysis Objects

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 →

©LTP Team

Saving Analysis Objects (LTPDA Toolbox)

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 01: a1 -----------

      name:   None
      data: (0,0.887479404587351) (0.1,-0.449797172615395) (0.2,0.479022102390939) ...
            -------- tsdata 01 ------------

                fs:   10
                 x:   [200 1], double
                 y:   [200 1], double
                dx:   [0 0], double
                dy:   [0 0], double
            xunits:   [s]
            yunits:   []
             nsecs:   20
                t0:   1970-01-01 00:00:00.000
            ----------------------------

      hist:  ao / ao / SId: fromTSfcn ... $-->$Id: ao ... S
   mdlfile:  empty
description:
      UUID:  67d31c2f-8ee5-42ba-a96b-cfb288ecc9ea
-------------------------------

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

    fs:   10
     x:   [200 1], double
     y:   [200 1], double
    dx:   [0 0], double
    dy:   [0 0], double
xunits:   [s]
yunits:   []
 nsecs:   20
    t0:   1970-01-01 00:00:00.000
-----------------------------
```

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

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

```
>> iplot(a1)
```
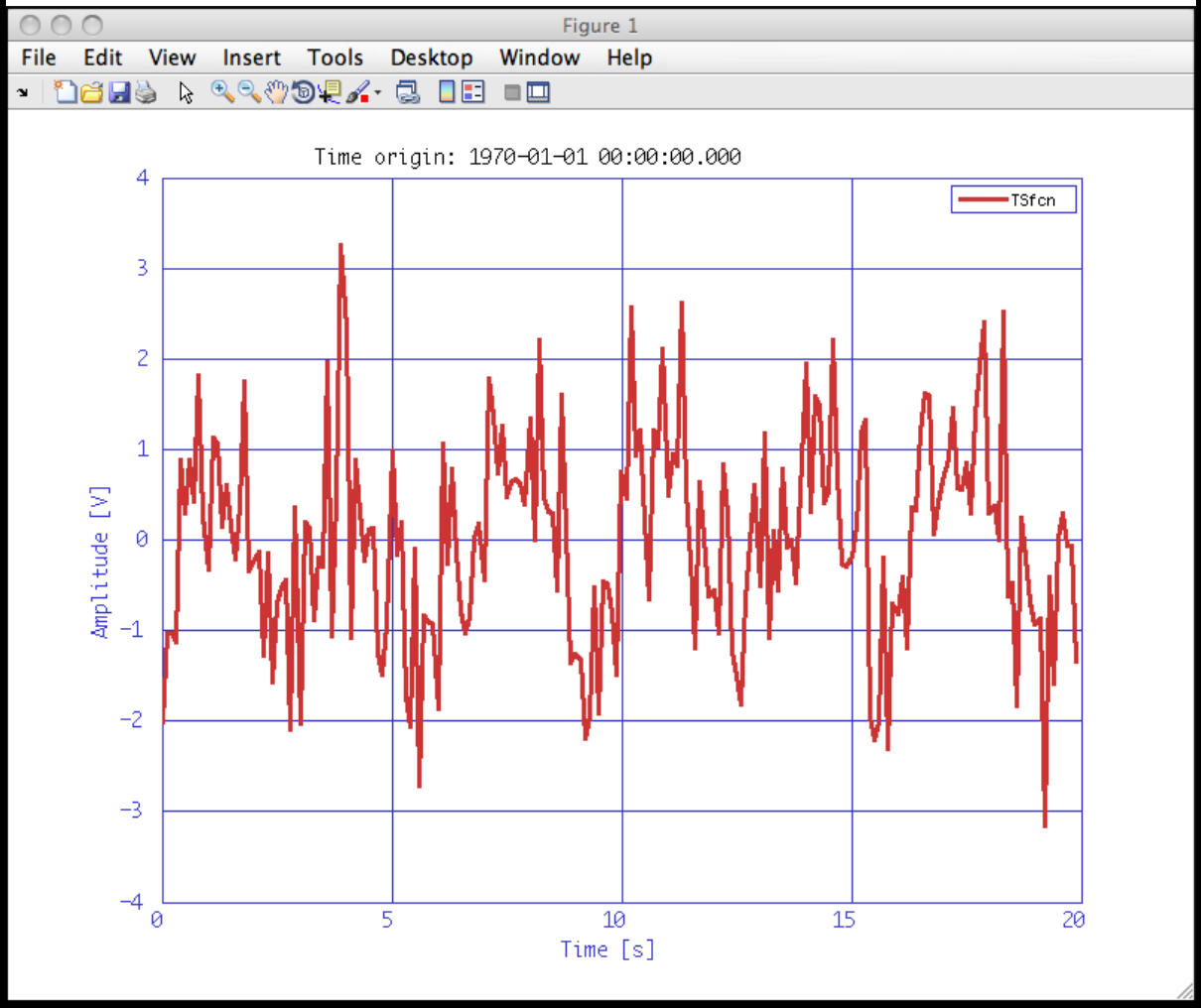
We also see that the time-origin of the data (`t0` field of the `tsdata` class) is displayed as the plot title.

Collection objects

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

You can create a `collection` object like:

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

You can then add objects to the collection by doing:

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

or

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

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

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

```
collection(empty-plist)

ao(VALS=1, NAME='None', DESCRIPTION=
    ", PLOTINFO=(empty-plist),
    N=1, YUNITS=")

        addObjects

pzmodel(GAIN=1, POLES=10,
ZEROS=100, NAME='None', DESCRIPTION=
    ", IUNITS=", OUNITS=")

mfir(empty-plist)

        addObjects

        END
```

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶
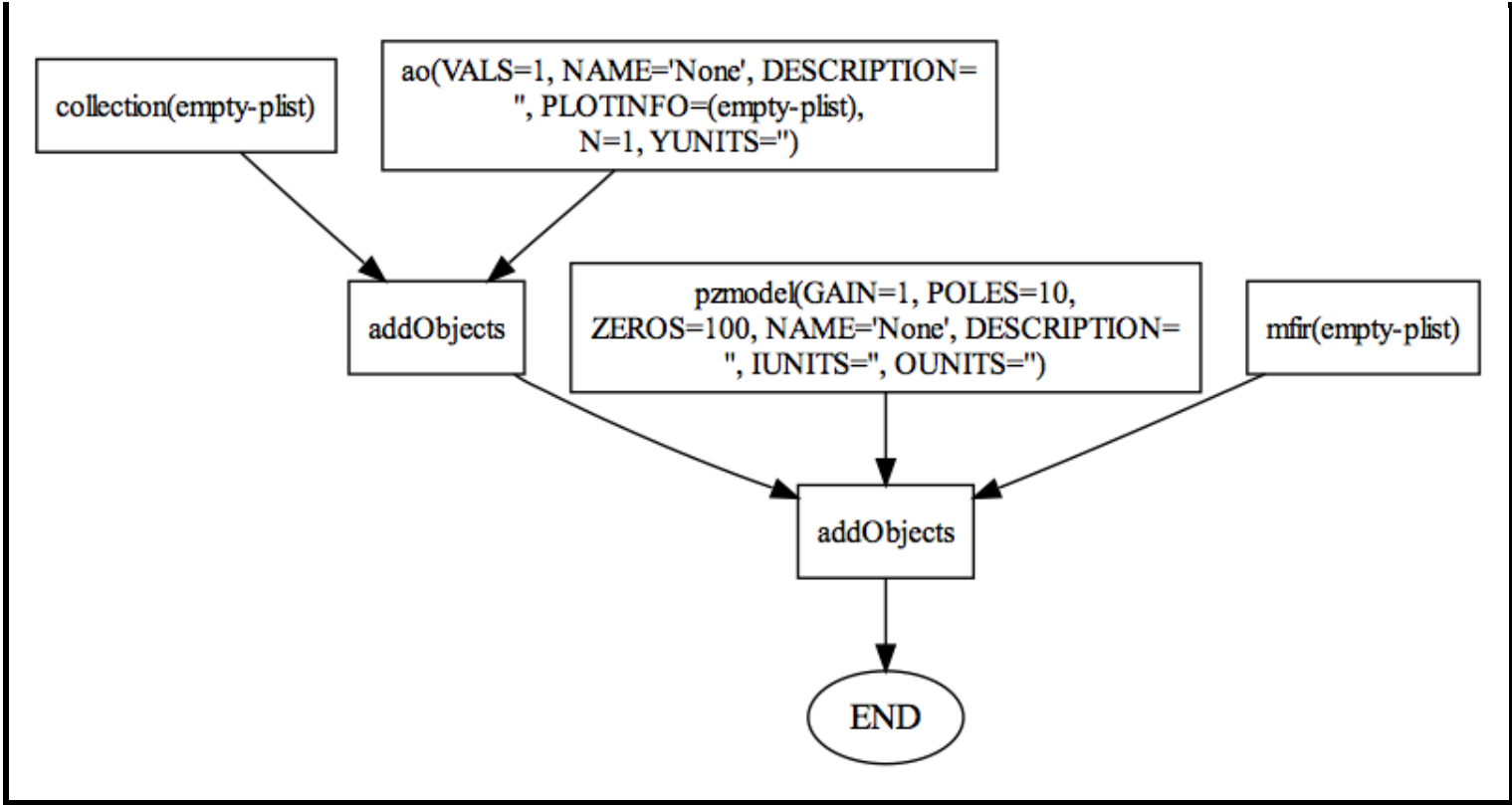
Units in LTPDA

Content needs written...

◀ Collection objects   Parameter Lists ▶

©LTP Team

Parameter Lists

Any algorithm that requires input parameters to configure its behaviour should take a Parameter List ([plist](#)) object as input. A `plist` object contains a vector of Parameter (`param`) objects, though the user never needs to deal with (`param`) objects directly.

The following sections introduce parameter lists:

- [Creating lists of Parameters](#)

©LTP Team

Creating lists of Parameters

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

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

Creating parameter lists from parameters.

Creating parameter lists directly.

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

```
pl = plist('a', 1, 'b', 'hello')
```

will result in the following output

```
    ----------- plist 01 -----------
    n params: 2
    ---- param 1 ----
    key:  A
    val:  1
    ----------------
    ---- param 2 ----
    key:  B
    val:  'hello'
    ----------------
    description:
    UUID:        9cc7d1f1-6ca3-477b-a6cc-2629d52579e6
-------------------------------
```

Appending parameters to a parameter list.

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

```
pl  = append(pl, 'c', 3)  % append a third parameter
```

will result in the following output

```
    ----------- 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.

⬅ Parameter Lists   Simulation/modelling ➡

©LTP Team

Simulation/modelling

Content needs written...

◀ Creating lists of Parameters   Built-in models of LTPDA ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◄ ►

Built-in models of LTPDA

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

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

◄ Simulation/modelling   Generating model noise ►

©LTP Team

Built-in models of LTPDA (LTPDA Toolbox)

LTPDA Toolbox™   [contents]   ◀ ▶

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.

◀ Built-in models of LTPDA   Franklin noise-generator ▶

Franklin noise-generator

The following sections gives an introduction to the [generation of model noise](#) using the noise generator implemented in LTPDA.

- [Franklin's noise generator](#)
- [Description](#)
- [Inputs](#)
- [Outputs](#)
- [Usage](#)

Franklin's noise generator

Franklin's noise generator is a method to generate arbitrarily long time series with a prescribed spectral density. The algorithm is based on the following paper:

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

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

See [Generating model noise](#) for more general information on this.

Franklin's method does not require any 'warm up' period. It starts with a transfer function given as ratio of two polynomials.
The generator operates on a real state vector y of length n which is maintained between invocations. It produces samples of the time series in equidistant steps `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 cross-spectral density ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Noise generation with given cross-spectral density

[Multichannel Spectra](#)	Theoretical background on multichannel spectra.
[Noise generation](#)	Theoretical introduction to multichannel noise generation.
[Multichannel Noise Generation](#)	Generation of multichannel noise with given cross-spectral density matrix.
[Noisegen 1D](#)	Generation of one-dimensional noise with given spectral density.
[Noisegen 2D](#)	Generation of two-dimensional noise with given cross-spectral density.

The following sections gives an introduction to the generation of model noise with a given cross spectral density. Further details can be found in ref. [1].

Theoretical background on multichannel spectra

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

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

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

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

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

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

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:

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

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

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

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 $\ddot{\Sigma}$. A common assumption is $\ddot{\Sigma} = \ddot{I}$ (identity matrix) that is equivalent to assume the white processes having unitary variance and are completely uncorrelated being zero the off diagonal terms of the CSD matrix. Further details can be found in [1 - 3].

Theoretical introduction to multichannel noise generation

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.

Generation of multichannel noise with given cross-spectral density matrix

`LTPDA Toolbox` provides two methods (mchNoisegenFilter and mchNoisegen) of the class `matrix` for the production of multichannel noise coloring filter and multichannel colored noise data series. Noise data are colored Gaussian distributed time series with given cross-spectral density matrix. Noise generation process is properly initialized in order to avoid starting transients on the data series. Details on frequency domain identification of noisegen filters and on the noise generation process can be found in ref. [1]. mchNoisegenFilter needs a model for the one-sided cross-spectral density or power spectral density if we are considering one-dimensional problems. mchNoisegen instead accepts as input the noise generating filter produced by mchNoisegenFilter. Details on accepted parameters can be found on the documentation pages of the two methods:

- mchNoisegenFilter
- mchNoisegen

Generation of one-dimensional noise with given spectral density

`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. The function needs a model for the one-sided power spectral density of the given process. Details on accepted parameters can be found on the [noisegen1D](#) documentation page.

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

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

`noisegen2D` is a nose coloring tool allowing the generation two data series with the given cross-spectral density from two starting white and mutually uncorrelated data series. Coloring filters are constructed by a fitting procedure to a model for the corss-spectral density matrix provided. In order to work with `noisegen2D` you must provide a model (frequency series analysis objects) for the cross-spectral density matrix of the process. Details on accepted parameters can be found on the [noisegen2D](#) documentation page.

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

References

1. L. Ferraioli et. al., Calibrating spectral estimation for the LISA Technology Package with multichannel synthetic noise generation, Phys. Rev. D 82, 042001 (2010).
2. S. M. Kay, Modern Spectral Estimation, Prentice-Hall, 1999
3. G. M. Jenkins and D. G. Watts, Spectral Analysis and Its Applications, Holden-Day 1968.

Parameteric models

Content needs written...

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Introduction to parametric models in LTPDA

Content needs written...

◀ Parameteric models   Statespace models ➡

©LTP Team

Introduction to parametric models in LTPDA (LTPDA Toolbox)

Statespace models

Why use state space modeling?

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

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

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

Generalities on State Space Modeling

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

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

State variables
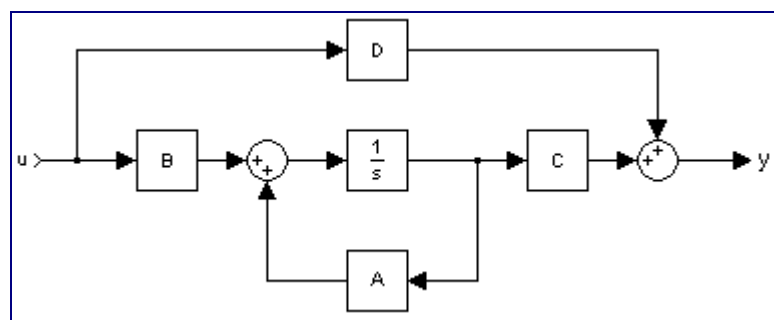
Typical state space model

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

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

Linear systems

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

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

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

where:

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

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

System type	State-space model
Continuous time-invariant	$\dot{\mathbf{x}}(t) = A\mathbf{x}(t) + B\mathbf{u}(t)$ $\mathbf{y}(t) = C\mathbf{x}(t) + D\mathbf{u}(t)$
Continuous time-variant	$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t)$ $\mathbf{y}(t) = \mathbf{C}(t)\mathbf{x}(t) + \mathbf{D}(t)\mathbf{u}(t)$
Discrete time-invariant	$\mathbf{x}(k+1) = A\mathbf{x}(k) + B\mathbf{u}(k)$

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

Discrete time-variant

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

**Laplace domain of
continuous time-invariant**

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

**Z-domain of
discrete time-invariant**

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

Example: Continuous-time LTI case

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

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

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

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

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

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

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

Controllability
Main article: Controllability

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

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

Observability
Main article: Observability

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

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

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

$$\mathrm{rank}\begin{bmatrix} C \\ CA \\ \ldots \\ CA^{n-1} \end{bmatrix} = n$$

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

Transfer function

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

First, taking the Laplace transform of

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

yields

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

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

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

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

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

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

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

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

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

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

Canonical realizations

Any given transfer function which is strictly proper can easily be transferred into state-space

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

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

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

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

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

$$\mathbf{y}(t) = \begin{bmatrix} n_1 & n_2 & n_3 & n_4 \end{bmatrix} \mathbf{x}(t).$$

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

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

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

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

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

Proper transfer functions

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

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

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

Here is an example to clear things up a bit:

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

which yields the following controllable realization

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

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

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

Feedback
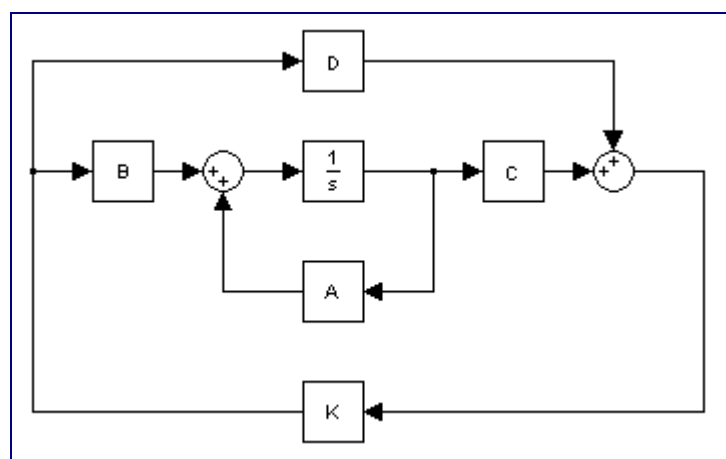
Typical state space model with feedback

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

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

becomes

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

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

$$\dot{\mathbf{x}}(t) = \left(A + BK\left(I - DK\right)^{-1} C\right) \mathbf{x}(t)$$
$$\mathbf{y}(t) = \left(I - DK\right)^{-1} C\mathbf{x}(t)$$

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

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

$$\dot{\mathbf{x}}(t) = (A + BK)\,\mathbf{x}(t)$$
$$\mathbf{y}(t) = \mathbf{x}(t)$$

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

Feedback with setpoint (reference) input
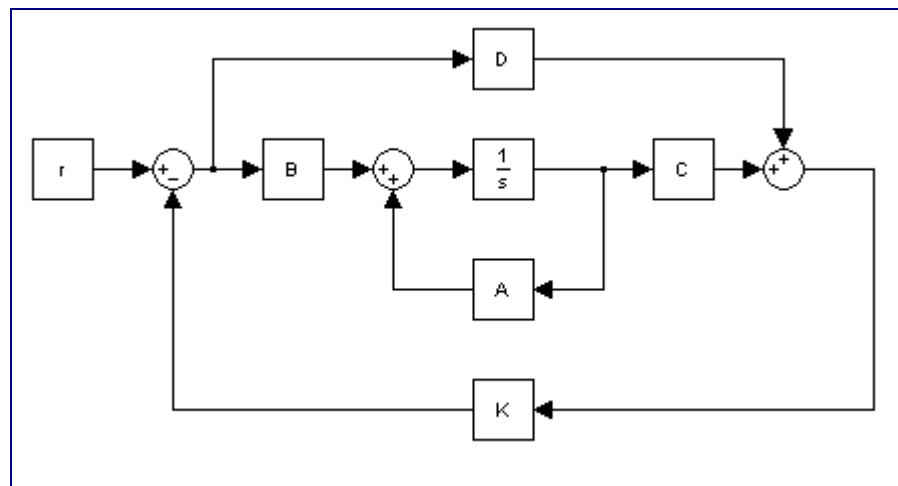
Output feedback with set point

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

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

becomes

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

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

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

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

$$\dot{\mathbf{x}}(t) = (A - BKC)\,\mathbf{x}(t) + B\mathbf{r}(t)$$
$$\mathbf{y}(t) = C\mathbf{x}(t)$$

Moving object example

A classical linear system is that of one-dimensional movement of an object. The Newton's laws of motion for an object moving horizontally on a plane and attached to a wall with a spring

$$m\ddot{y}(t) = u(t) - k_1\dot{y}(t) - k_2 y(t)$$

where

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

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

The state equation would then become

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

where

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

The [controllability](#) test is then

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

which has full rank for all k_1 and m.

The [observability](#) test is then

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

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

Nonlinear systems

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

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

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

Pendulum example

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

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

where

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

The state equations are then

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

where

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

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

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

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

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

for integers n.

◀Introduction to parametric models in LTPDA Introduction to Statespace Models with LTPDA ▶

©LTP Team

Introduction to Statespace Models with LTPDA

What is a ssm object?

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

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

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

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

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

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

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

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

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

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

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

©LTP Team

Building Statespace models

How to build a ssm object.

- The empty constructor creates an empty statespace model

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

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

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

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

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

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

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

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

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

◀ Introduction to Statespace Models with LTPDA   Building from scratch ➡

©LTP Team

Building from scratch

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

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

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

The most retrained set of inputs is

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

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

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

example :

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

```
        3 : input 3 | input 3 > 1 [], input 3 > 2 []
    states:  [1x3 ssmblock]
        1 : state 1 | state 1 > 1 []
        2 : state 2 | state 2 > 1 []
        3 : state 3 | state 3 > 1 [], state 3 > 2 []
   outputs:  [1x3 ssmblock]
        1 : output 1 | output 1 > 1 []
        2 : output 2 | output 2 > 1 []
        3 : output 3 | output 3 > 1 [], output 3 > 2 []
    params: (empty-plist) [1x1 plist]
   version: $Id$
   Ninputs: 3
 inputsizes: [1 1 2]
  Noutputs: 3
outputsizes: [1 1 2]
   Nstates: 3
 statesizes: [1 1 2]
   Nparams: 0
isnumerical: false
      hist: ssm.hist [1x1 history]
   procinfo: (empty-plist) [1x1 plist]
   plotinfo: (empty-plist) [1x1 plist]
      name: sys
description:
    mdlfile:
      UUID: 61f33fcc-f06a-4d71-944f-3ea094c80458
-------------------
```

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

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

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

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

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

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

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

Incomplete description whithout input/state/outputs names

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

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

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

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

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

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

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

```
>> sys.inputs

------ ssmblock/1 -------
      name: myinput1
     ports: input 1 > 1 [] [1x1 ssmport]
description: myinput1 description
   version: $Id$
----------------------
------ ssmblock/2 -------
      name: myinput2
     ports: input 2 > 1 [] [1x1 ssmport]
description:
   version: $Id$
----------------------
------ ssmblock/3 -------
      name: myinput3
     ports: my port 3-1 [kg m^(-2)], my port 3-2 [m] [1x2 ssmport]
description:
   version: $Id$
----------------------

>>sys.inputs(3).ports

------ ssmport/1 -------
      name: my port 3-1
     units: [kg m^(-2)] [1x1 unit]
description: first description
   version: $Id$
----------------------
------ ssmport/2 -------
      name: my port 3-2
     units: [m] [1x1 unit]
description: second description
   version: $Id$
----------------------
```

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

Description whith input/state/outputs names

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

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

Complete description

example :

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

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

This constructor still does not set ports properties (except automatic name) nor the block description.
These may be set by the user later on using the setter functions above.

©LTP Team

Building from built-in models

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

Using existing built-in models

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

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

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

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

This way the numbers and names are displayed in an error message.
Note that the numbers may change if a new model is added, and using the name is highly recommended.

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

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

```
        name: standard_system_params
 description: standard spring-mass-dashpot test system
     mdlfile:
        UUID: 4415784d-79d9-408d-9b82-3f39ace0d518
```

Options for the built-in constructor

There are two options:

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

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

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

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

How to make your own built-in model?

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

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

©LTP Team

Modifying systems

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

Modifying using the setter functions

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

Here is a simple model

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

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

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

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

Modifying the parameter values

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

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

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

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

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

Modifying the inputs/states/outputs

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

```
>> name = 'sys';
>> timestep = 0;
>> amats = {diag([-1 -2 -3 -4])};
>> bmats = {diag([1 2 0 0])};
>> cmats = {[diag([1 0 3 0]) ; [0 0 0 0 ]]};
>> dmats = {[diag([0 0 0 0]) ; [0 0 0 0 ]]};
>> sys = ssm(plist( ...
'amats',amats, 'bmats',bmats, 'cmats',cmats, 'dmats',dmats, ...
'timestep',timestep, 'name',name));
It is possible to obtain the structural realization of the system using sMinReal. This
function deletes states which are either not observable or controllable. However, there may
still exist such linear combinations of states.
The process only modify the states, here the 2nd and 4rt states disappear since they are not
observable, and the 3rd and 4th disappear since they are not controllable.
>> sys.sMinReal
M: running ssm/ssm
```

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

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

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

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

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

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

```
>> sys.simplify(plist('inputs', {[1 2]} , 'states', 'ALL' , 'outputs', { 'output 1 > 1'
'output 1 > 4'}))
M: running ssm/simplify
M: running ssm/ssm
M: running ssm/simplify
M: running display
------ ssm/1 -------
amats: { [1x1] } [1x1]
mmats: { [1x1] } [1x1]
bmats: { [1x2] } [1x1]
cmats: { [2x1] } [1x1]
dmats: { [] } [1x1]
timestep: 0
inputs: [1x1 ssmblock]
1 : input 1 | input 1 > 1 [], input 1 > 2 []
states: [1x1 ssmblock]
1 : state 1 | state 1 > 1 []
outputs: [1x1 ssmblock]
1 : output 1 | output 1 > 1 [], output 1 > 4 []
params: (empty-plist) [1x1 plist]
version: $Id$
Ninputs: 1
inputsizes: 2
```

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

◀ Building from built-in models   Assembling systems ▶

©LTP Team

Assembling systems

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

Managing inputs/outputs when assembling

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

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

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

Example using an existing built-in models

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

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

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

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

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

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

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

We can then use the system to produce a simulation.

◀ Modifying systems   Simulations ▶

©LTP Team

Simulations

The function simulate can use ssm object to produce simulations.

Simulation example.

The following closed loop system is built.

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

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

We want as an output the controller output "U" and the sensor output "y".

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

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

```
>> ao_est = sysCL.kalman( plist(...
   'NOISE VARIABLE NAMES', {'Fn' 'On'}, 'COVARIANCE', [1 0.1 ; 0.1 2] , ...
   'AOS VARIABLE NAMES', {'Fu'} ,'AOS', ao1 ,...
   'CONSTANT VARIABLE NAMES', {'On'}, 'CONSTANTS', 35, ...
   'OUTPUT VARIABLE NAMES', {'y'}, 'OUTPUTS' , ao_out(2), ...
   'RETURN STATES', 1, 'RETURN OUTPUTS', 1 ));
>> iplot(ao_out(2), ao_est(1), ao_out(1))
```
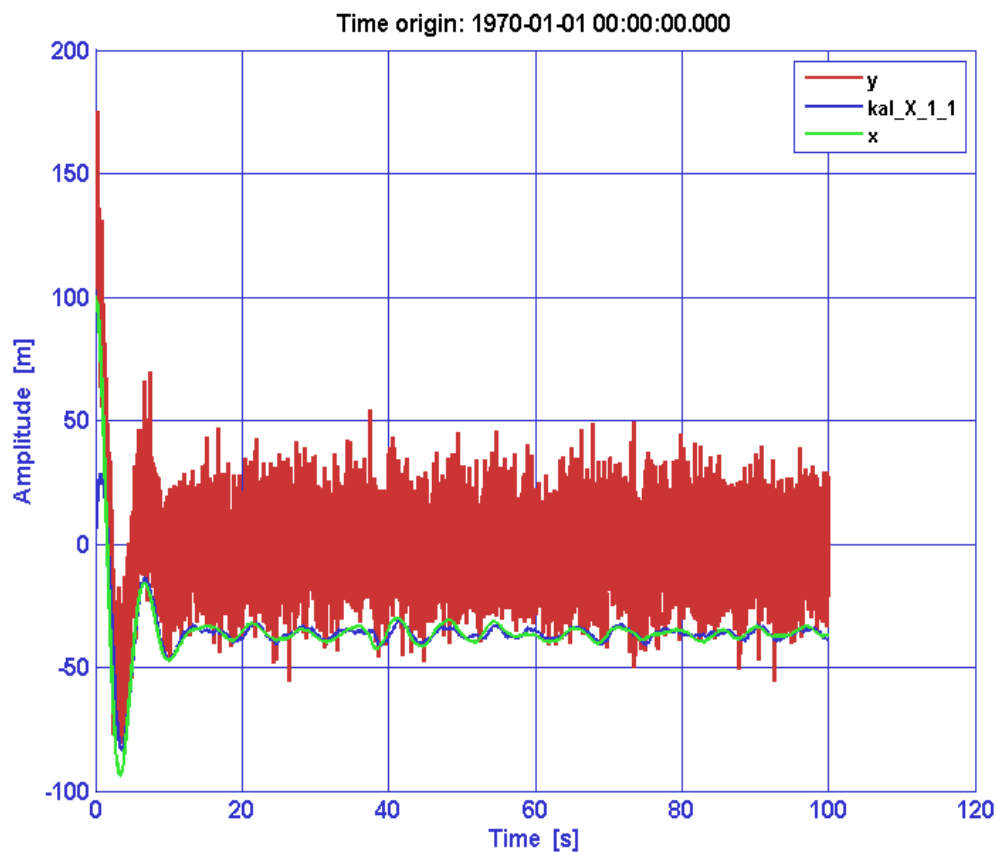
In this example the estimate (blue) of the state (green) is satisfactory. It leads us to think that such a filter should be used to provide with the input of the controller.

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

◀Assembling systems   Transfer Function Modelling▶

©LTP Team

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](#)

©LTP Team

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.

- [Building a model](#)
- [Model helper GUI](#)
- [Converting models to IIR filters](#)

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

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:

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

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

- Direct form
- Creating from a `plist`
- Computing the response of the model

Direct form

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

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

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

Creating from a `plist`

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

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

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

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

Computing the response of the model

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

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

Since no output was specified, this command produces the following plot:
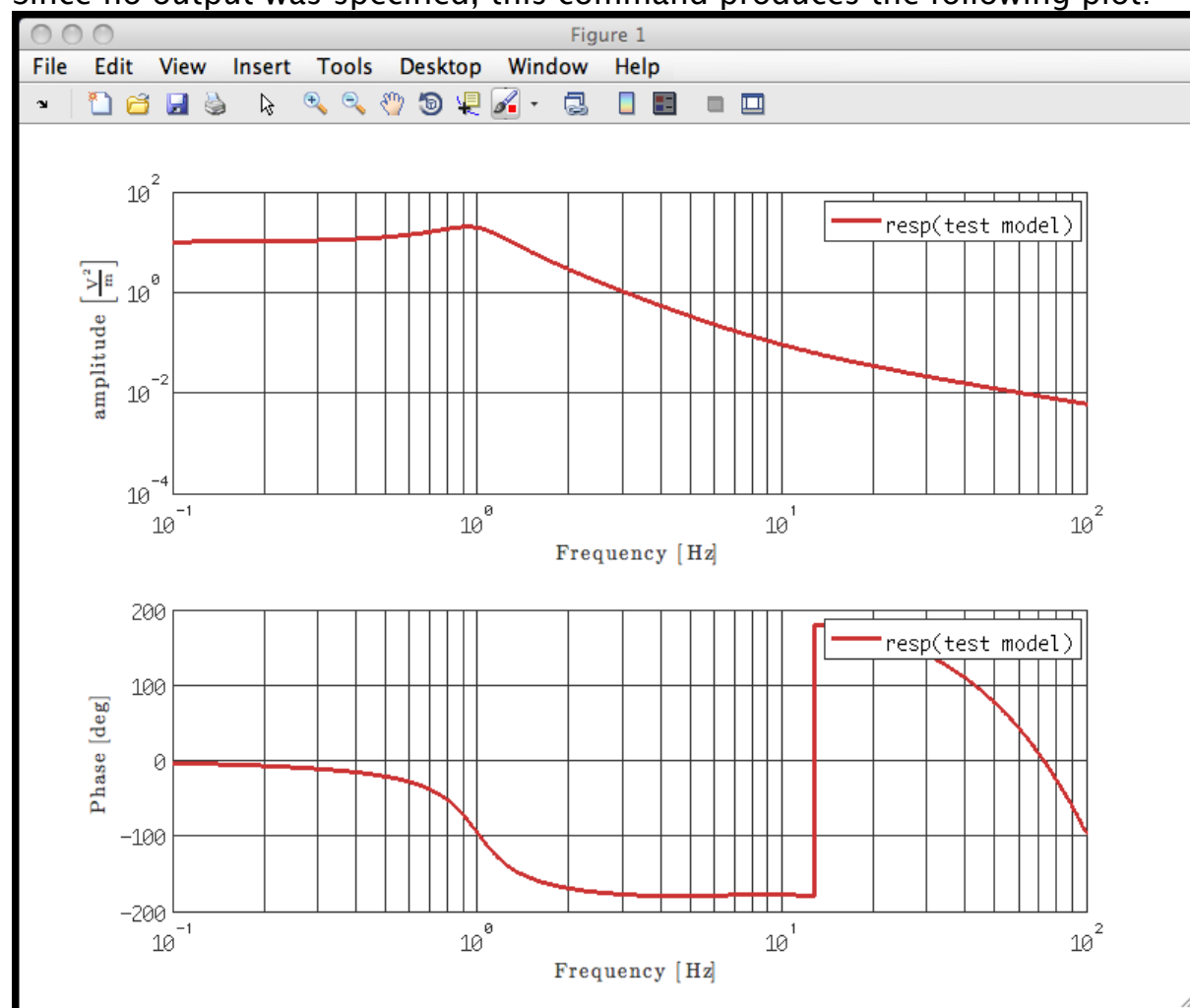
You can also specify the frequency band over which to compute the response by passing a `plist` to the `resp` method, as follows:

```
>> rpl = plist('f1', 0.1, ...
               'f2', 1000, ...
               'nf', 10000);
>> a   = resp(pzm, rpl)
---------- ao 01: resp(test model) -----------

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

          fs:   NaN
```

```
              x:  [1 10000], double
              y:  [1 10000], double
         xunits:  [Hz]
         yunits:  [V^(2)][m^(-1)]
             t0:  1970-01-01 00:00:00.000
           navs:  NaN
         -------------------------------

      hist:  pzmodel / resp / $Id$
  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.

◀ Pole/Zero representation   Model helper GUI ▶

©LTP Team

Model helper GUI

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

```
>> pzmodel_helper
```

More details on the [PZModel Helper GUI](#).

◀ Building a model   Sum of partial fractions representation ▶

©LTP Team

Model helper GUI (LTPDA Toolbox)

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 ▶

©LTP Team

Rational representation

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

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

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

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

	Pole/Zero	Rational	Partial Fraction
Pole/Zero		√	√
Rational	√		√
Partial Fraction	√	√	

From pzmodel to rational

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

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

From rational to pzmodel

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

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

Algorithm

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

◀ Rational representation   Converting models to digital filters ▶

©LTP Team

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)
```

◀ Converting models between different representations   Signal Pre-processing in LTPDA ▶

LTPDA Toolbox™   [contents](#)   ◀ ➡

Signal Pre-processing in LTPDA

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

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

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

◀ Converting models to digital filters   Downsampling data ➡

©LTP Team

Downsampling data

Description

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

Syntax

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

Parameters

The following parameters can be set in this method:

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

Examples

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

```
% create an AO of random data with fs = 10 Hz
pl      = plist('tsfcn', 'randn(size(t))','fs',10,'nsecs',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)
```
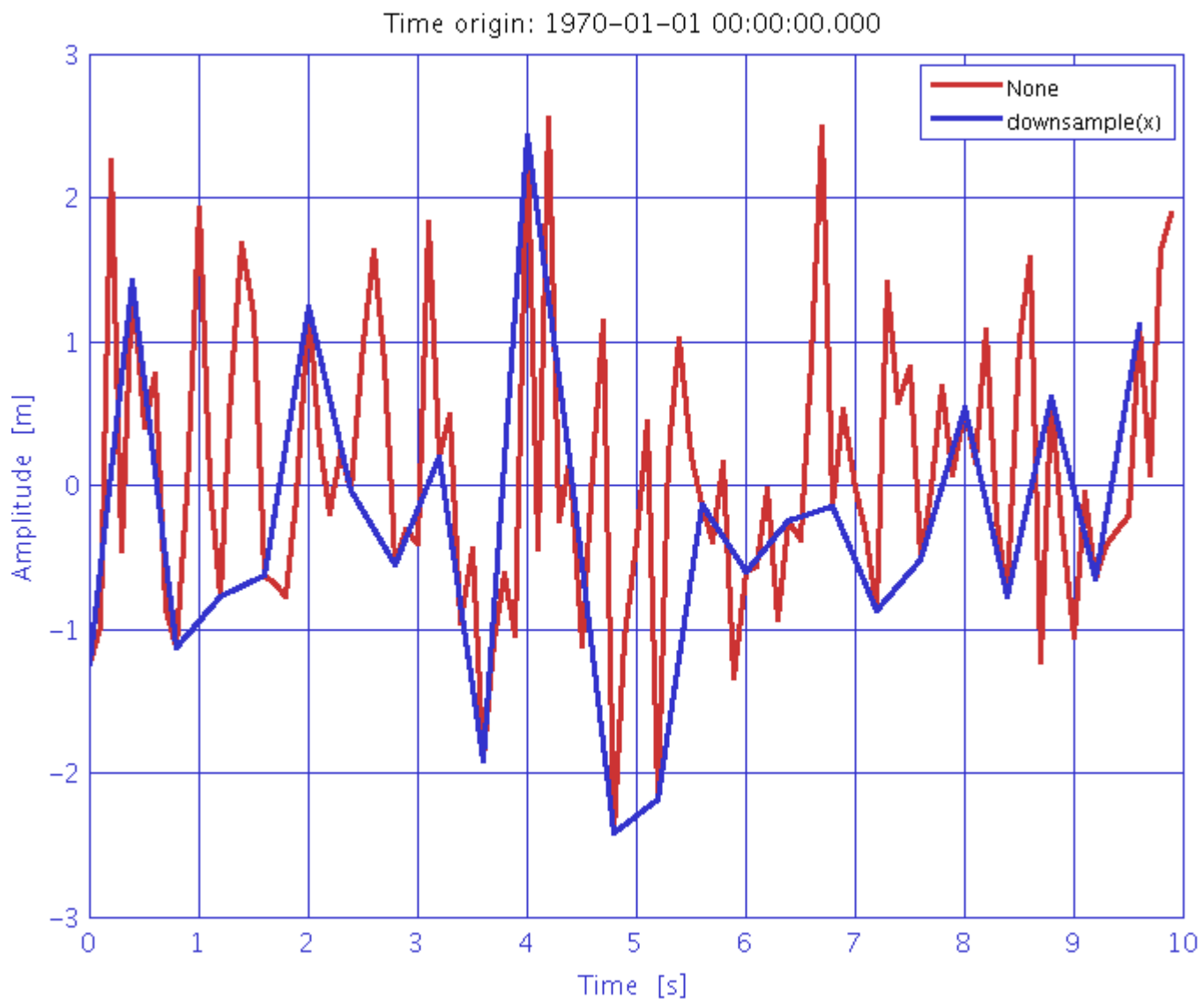
2. Downsampling a sequence of random data at original sampling rate of 10 Hz by a factor of 4 (fsout = 2.5 Hz) and offset = 10.

```
% create an AO of random data with fs = 10 Hz
pl         = plist('tsfcn', 'randn(size(t))','fs',10,'nsecs',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)
```

Time origin: 1970-01-01 00:00:00.000
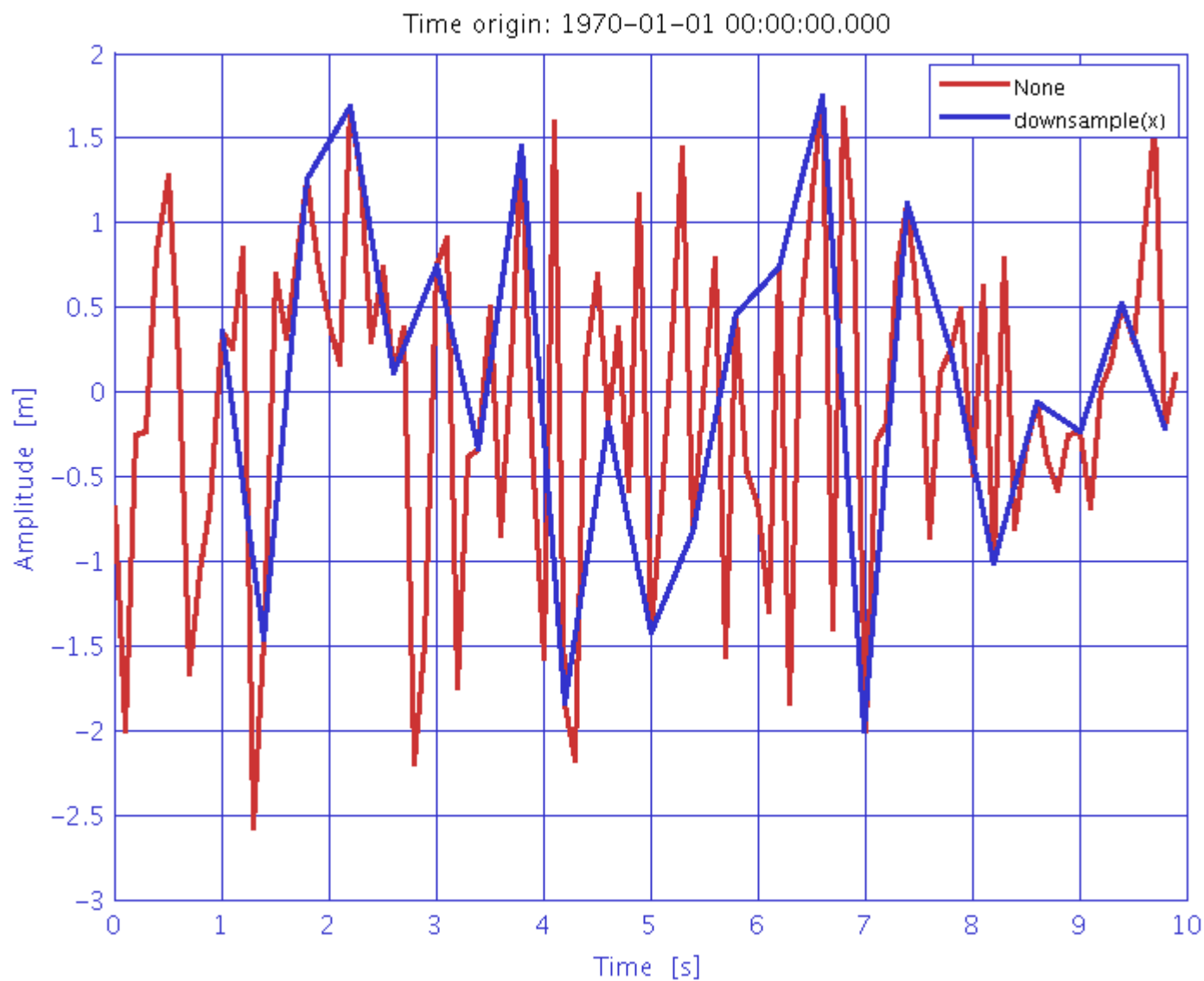
©LTP Team

Upsampling data

Description

Upsampling is the process of increasing the sampling rate of a signal. <u>Upsample</u> increases the sampling rate of the input AOs by an integer factor. LTPDA <u>upsample</u> 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)
```

Parameters

The following parameters can be set in this method:

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

Examples

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

```
pl     = plist('tsfcn', 'randn(size(t))','fs',1,'yunits','m','nsecs',100);
x      = ao(pl);
pl_up = plist('N', 10);      % increase the sampling frequency by a factor of 10
y      = upsample(x, pl_up); % resample the input AO (x) to obtain the upsampled AO (y)
iplot(x, y)
```
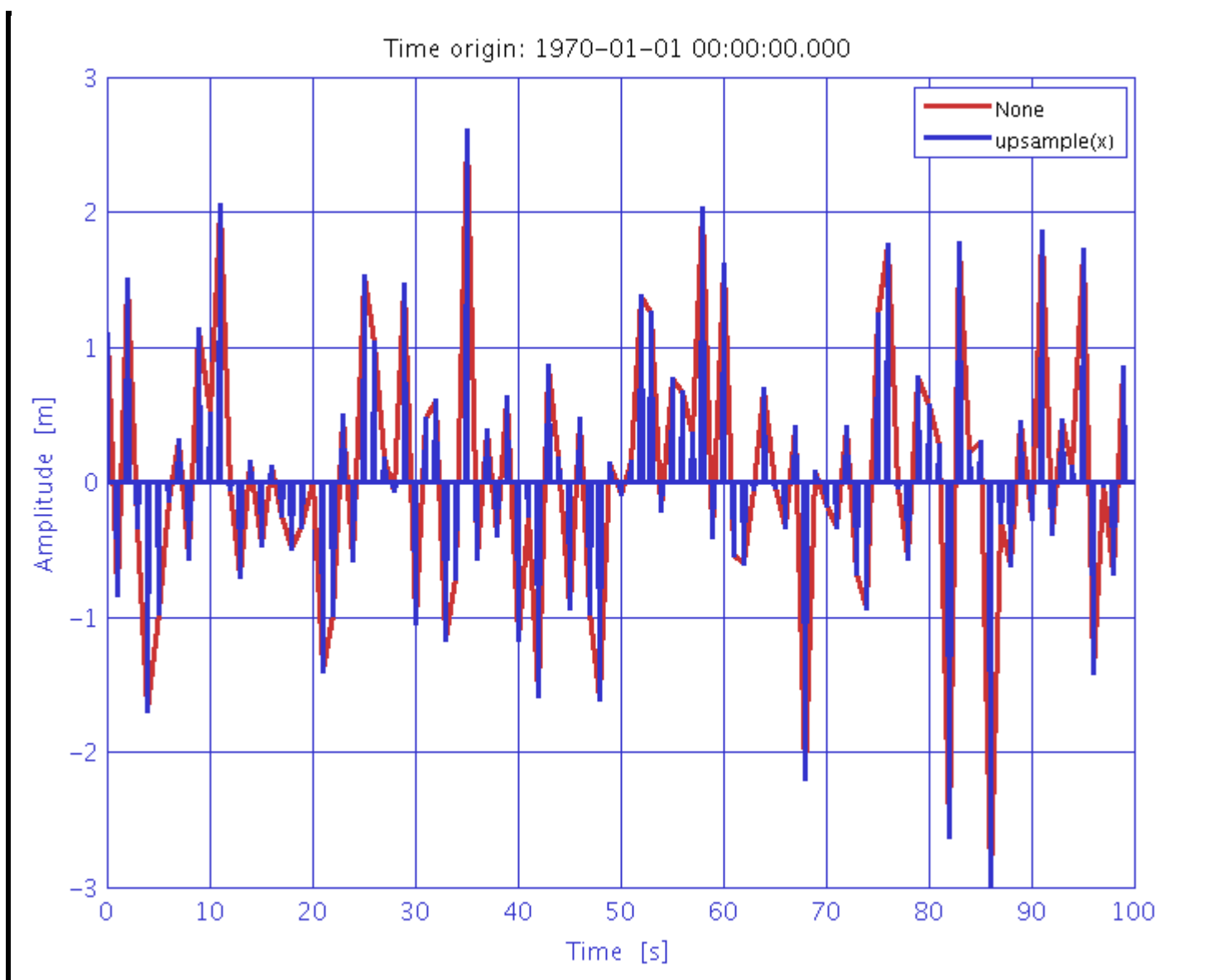
2. Upsampling a sequence of random data at original sampling rate of 1 Hz by a factor of 21 with a phase of 20 samples.

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

Time origin: 1970-01-01 00:00:00.000
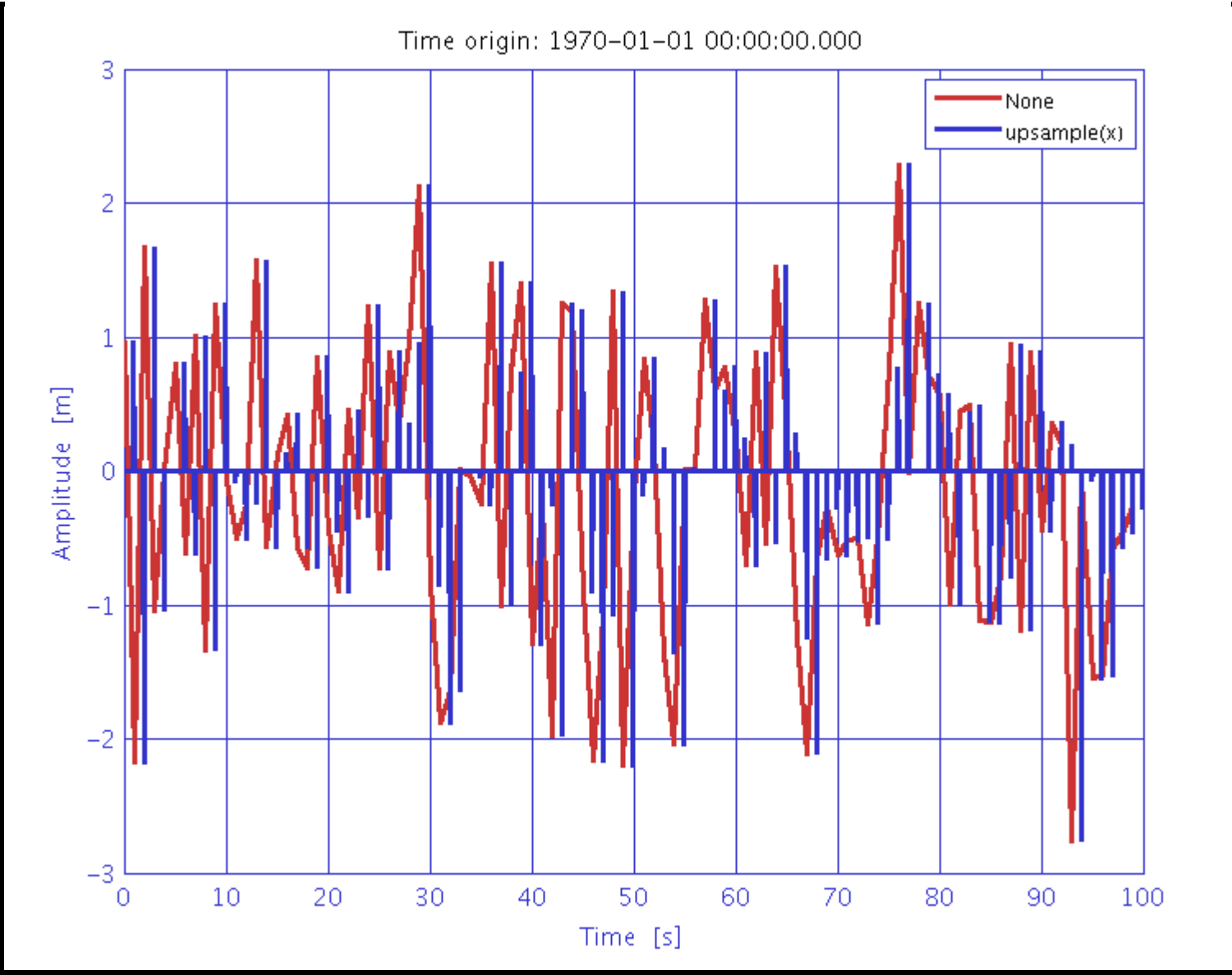
◀ Downsampling data   Resampling data ▶

©LTP Team

Resampling data

Description

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

Syntax

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

Parameters

The following parameters can be set:

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

Examples

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

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

Time origin: 1970-01-01 00:00:00.000
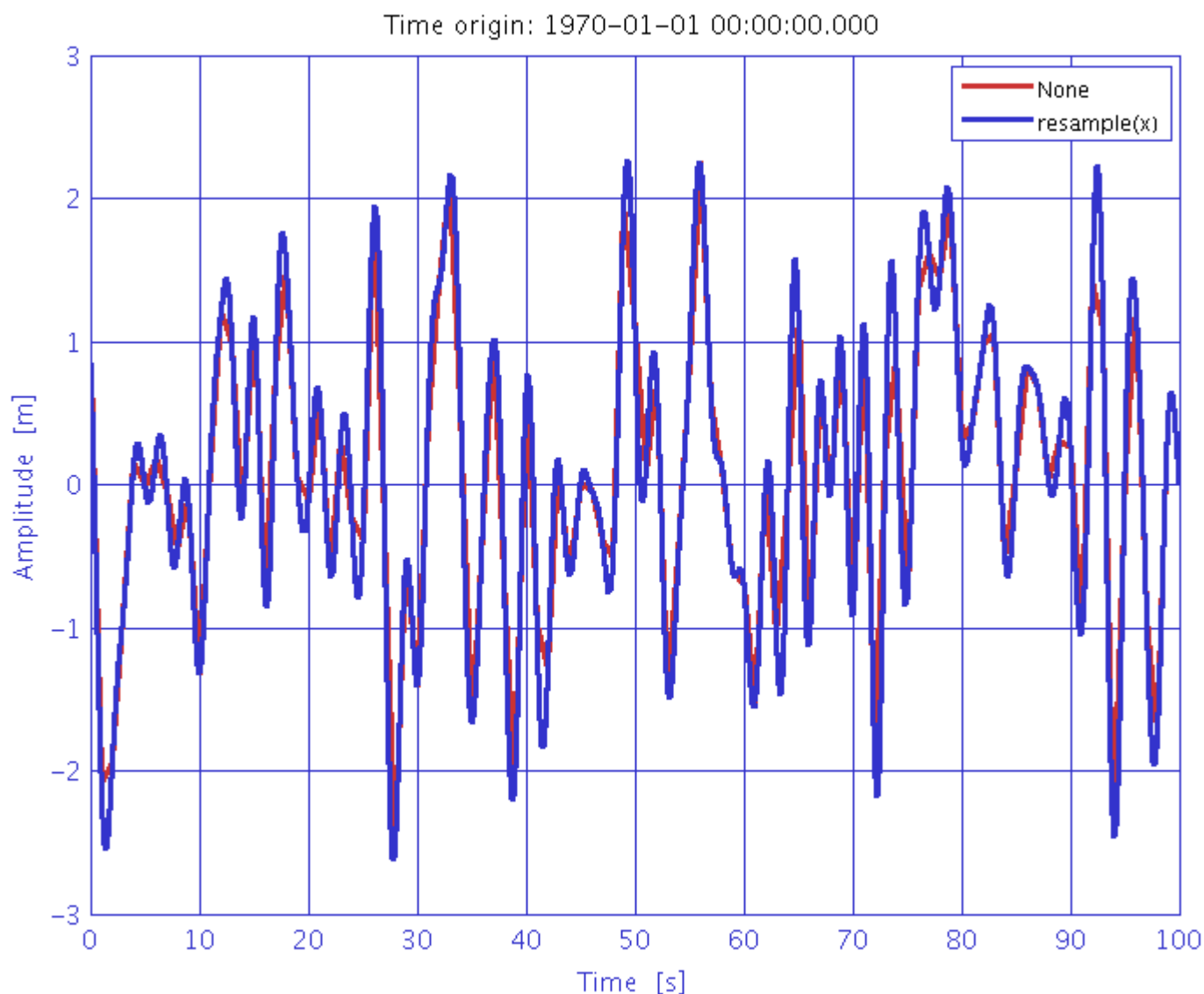
1. Resampling a sequence of random data at original sampling rate of 10 Hz at an output sampling of 1 Hz with a filter defined by the user.

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

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

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

Time origin: 1970-01-01 00:00:00.000
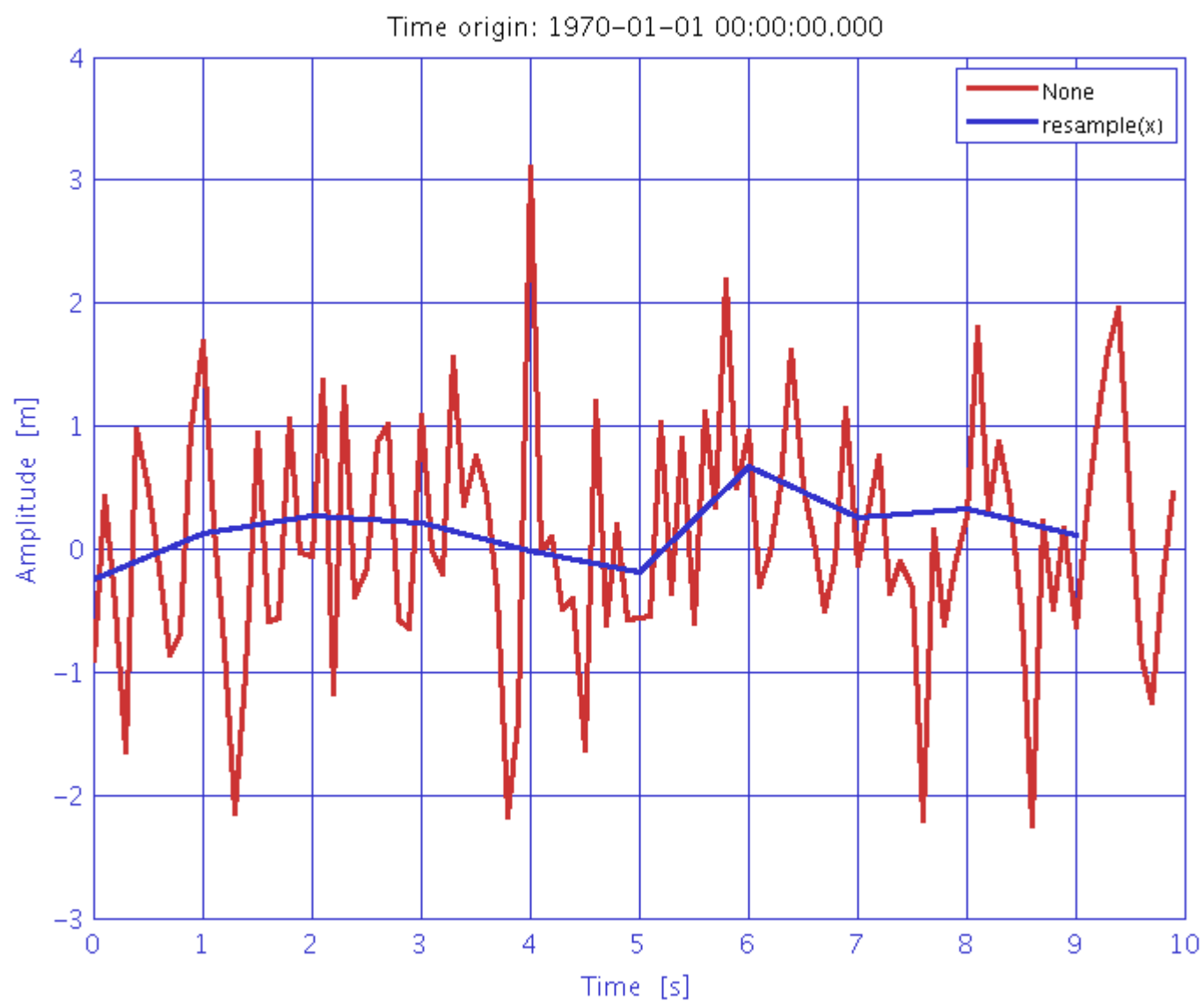
©LTP Team

Interpolating data

Description

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

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

Syntax

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

Parameters

The following parameters can be set in the method:

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

For details see `interp1` help of Matlab.

Examples

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

```
% Signal generation
pl = plist('tsfcn','sin(2*pi*1.733*t)',...
'fs',1,'nsecs',10,...
'yunits','V');
x = ao(pl);
% Interpolate on a new time vector
t = linspace(0, x.data.nsecs - 1/x.data.fs, 2*len(x));
pl_spline  = plist('vertices',t);
pl_nearest = plist('vertices',t,'method','nearest');
x_spline   = interp(x,pl_spline);
x_nearest  = interp(x,pl_nearest);
iplot([x x_spline x_nearest], plist('Markers', {'x', 'o', '+'}, ...
'LineColors', {'k', 'r'}));
```

Time origin: 1970-01-01 00:00:00.000
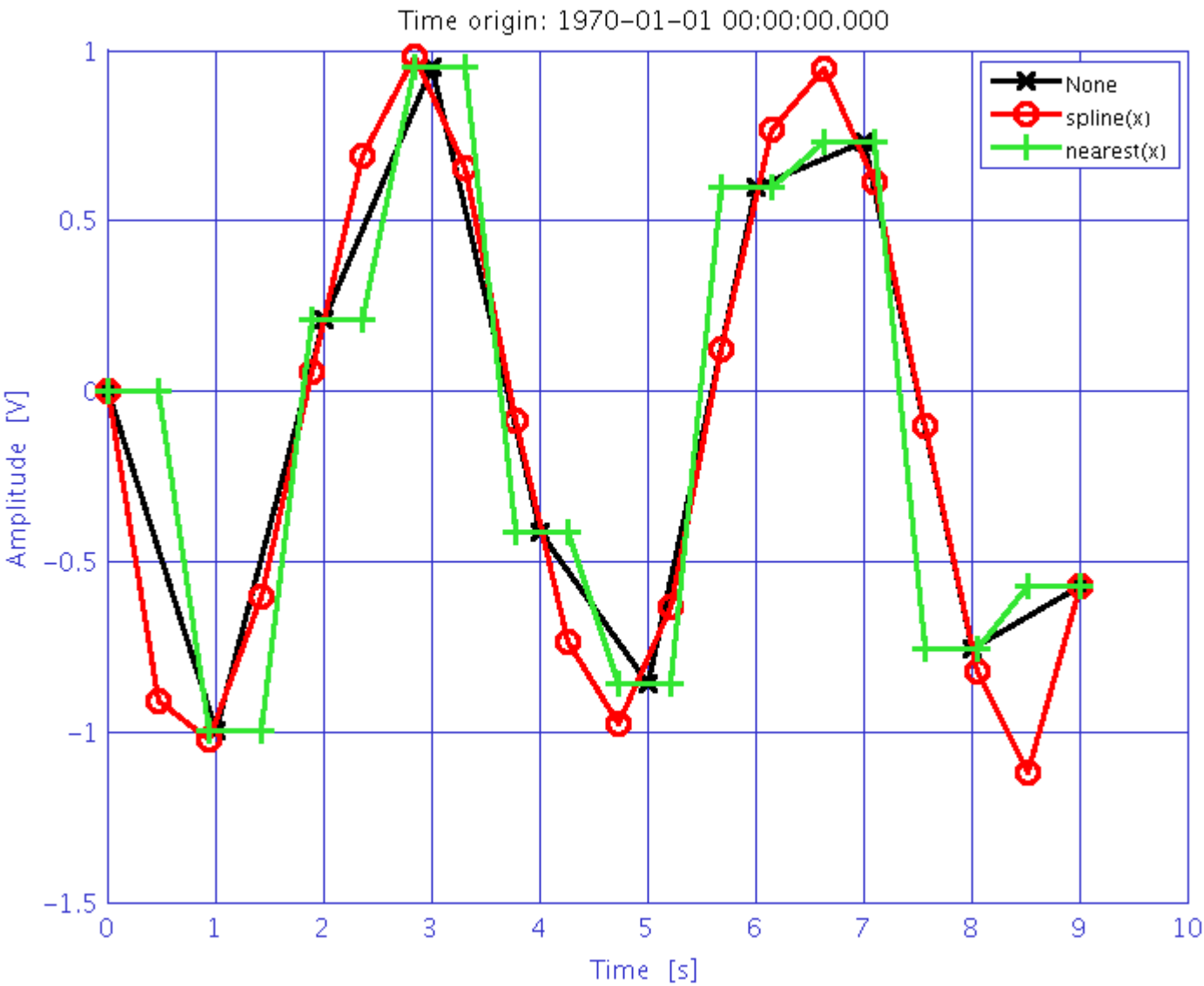
©LTP Team

Spikes reduction in data

Description

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

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

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

Syntax

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

Parameters

The following parameters can be set in this method:

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

Algorithm

Random: this method substitutes the spiky sample by:

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

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

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

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

Previous: the spike is 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(plist( 'waveform', 'noise', 'nsecs',1e4, 'fs',10)); % create an AO of random
data sampled at 1 Hz.
pl = plist( 'kspike', 2);                                  % kspike = 2
y = spikecleaning(x, pl);                                  % spike cleaning function applied to
the input AO, x
iplot(x, y)                                                % plot original and "cleaned" data
```

Time origin: 1970-01-01 00:00:00.000
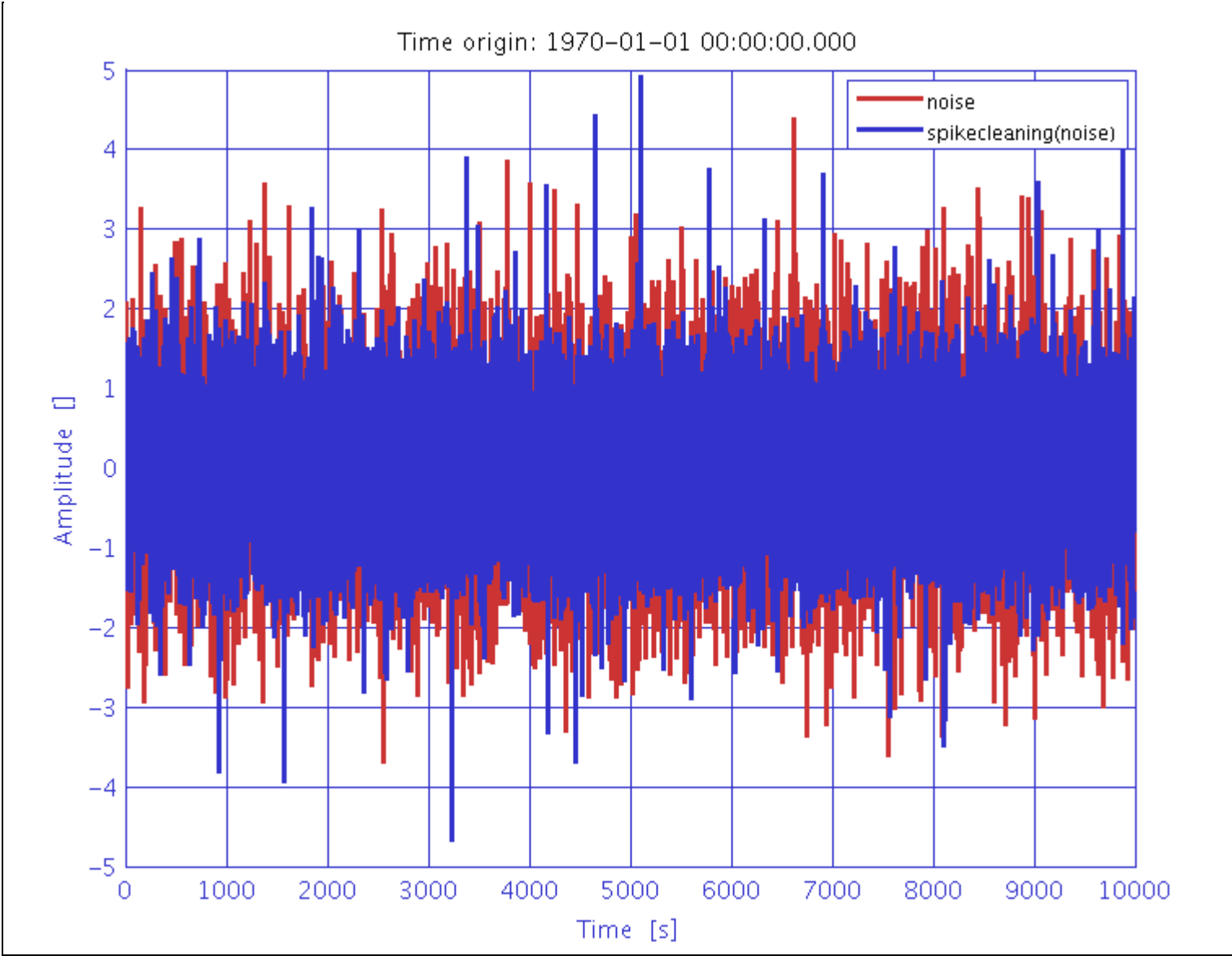
Data gap filling

Description

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

Syntax

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

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

Parameters

The following parameters apply for this method:

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

Algorithm

Linear : The gap is filled using a linear approximation.

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

the notation used in the previous expression is schematically explained in the figure below
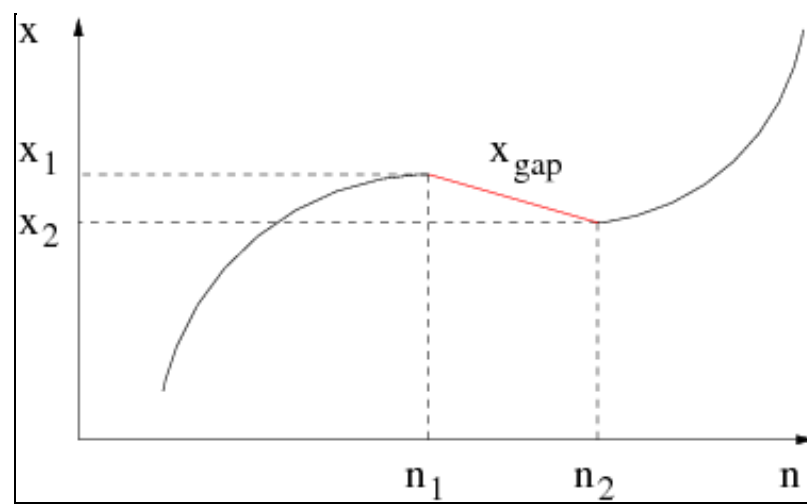
Spline : a third order interpolation is used in this case

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

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

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

The result is schematically shown in the figure below
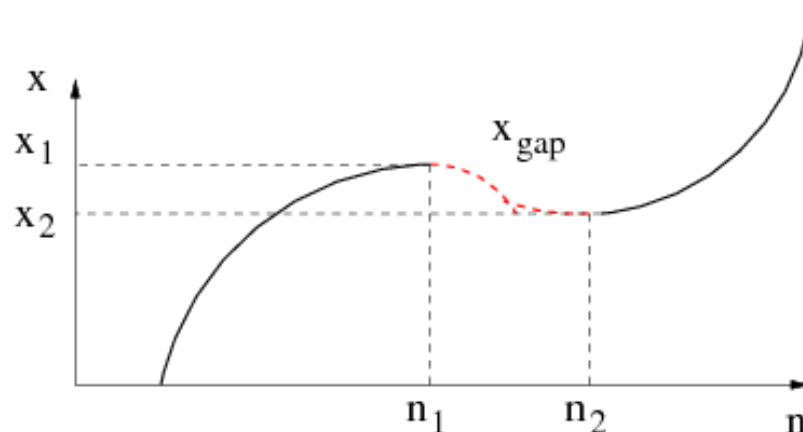
Examples

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

```
   x1 = ao(plist('waveform','noise','nsecs',1e3,'fs',10,'name','noise')); % create an AO of
random data
   xs = split(x1,plist('chunks',3));                          % split in three segments
   pl = plist( 'method',  'linear', 'addnoise', 'no');        % linear gapfilling
   y = gapfilling(xs(1),xs(3), pl);                           % fill between 2nd and 3rd
segment
   iplot(x1,y)
```
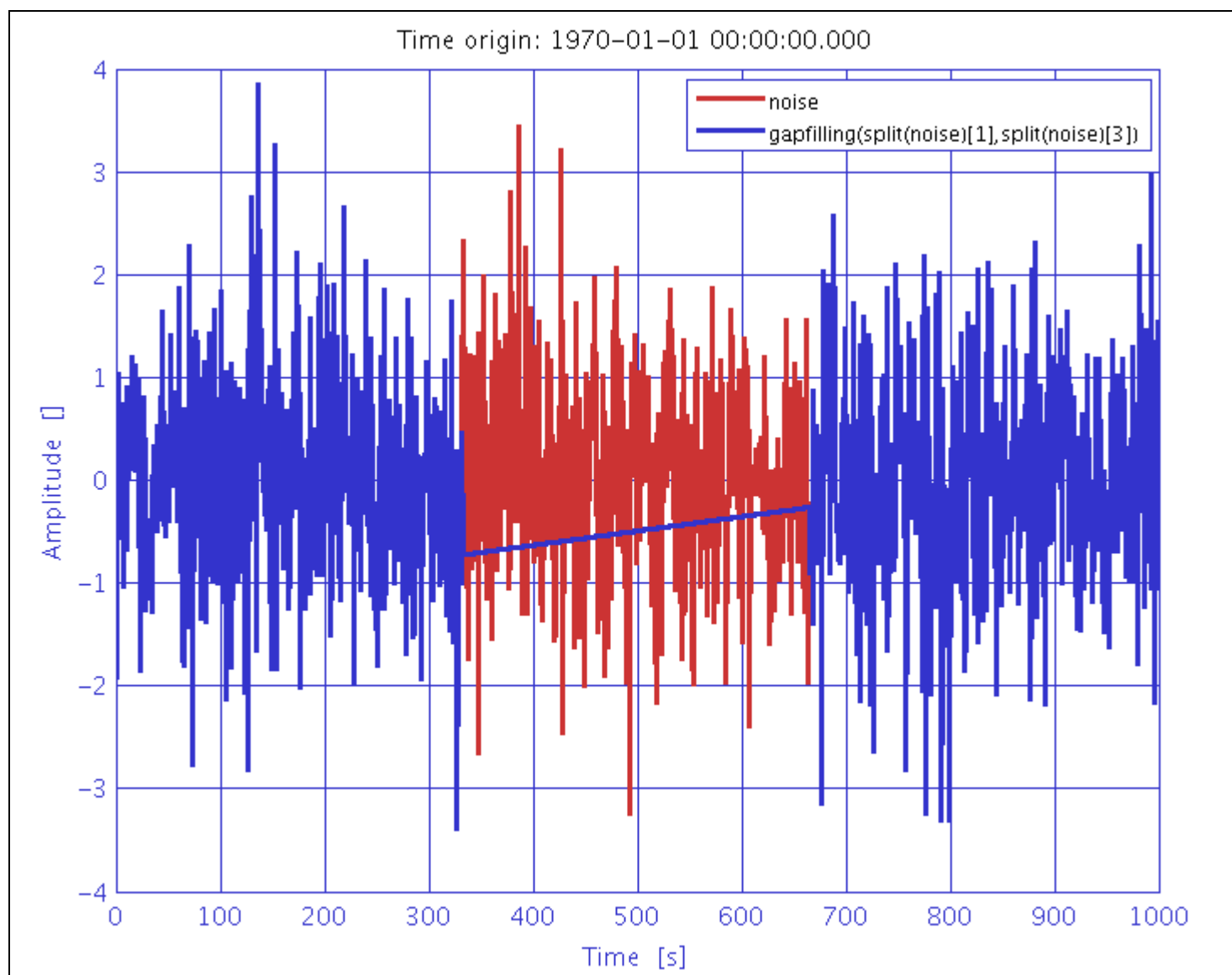

2. Missing data between two data vectors interpolated with the `spline` method.

```
   x1 = ao(plist('tsfcn','t.^1.8 + 1e6*rand(size(t))','nsecs',5e3,'fs',10,'name','noise')); %
create an AO
   xs = split(x1,plist('chunks',3));                          % split in three segments
   pl = plist( 'method',  'spline', 'addnoise', 'no');        % cubic gapfilling
   y = gapfilling(xs(1),xs(3), pl);                           % fill between 2nd and 3rd
segment
   iplot(x1,y)
```
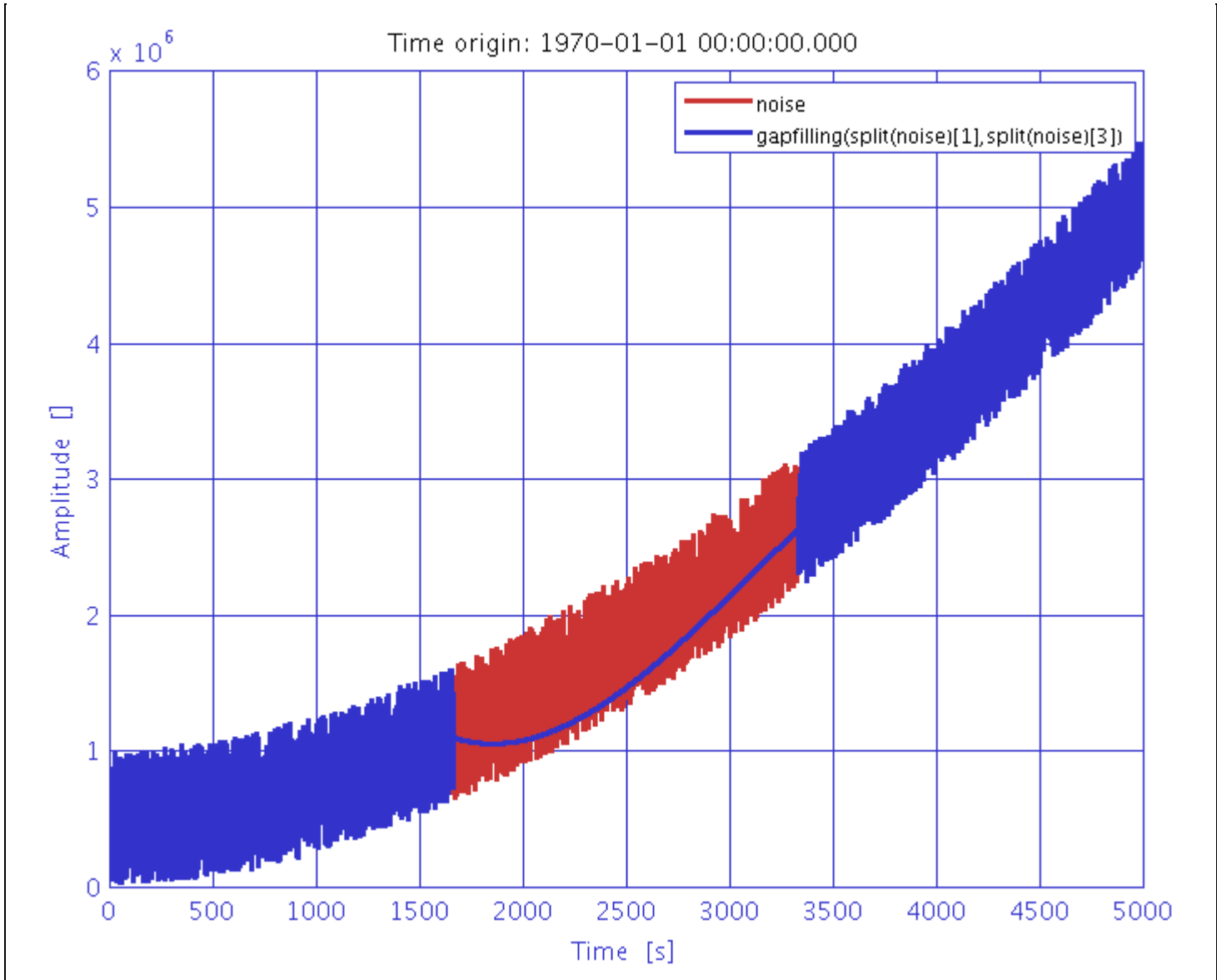
©LTP Team

Noise whitening

[Introduction](#)	Noise whitening in LTPDA.
[Algorithm](#)	Whitening Algorithms.
[1D data](#)	Whitening noise in one-dimensional data.
[2D data](#)	Whitening noise in two-dimensional data.

Noise whitening in LTPDA

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

$$\mu_w = \mathbb{E}\left[w(t)\right] = 0$$

$$C_w\left(t_i, t_j\right) = C_0 \delta\left(t_i - t_j\right)$$

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

$$S(\omega) = C_0$$

In other words, The power per unit of frequency associated to a white noise process is uniformly distributed on the whole available frequency range. An example is reported in figure 1.
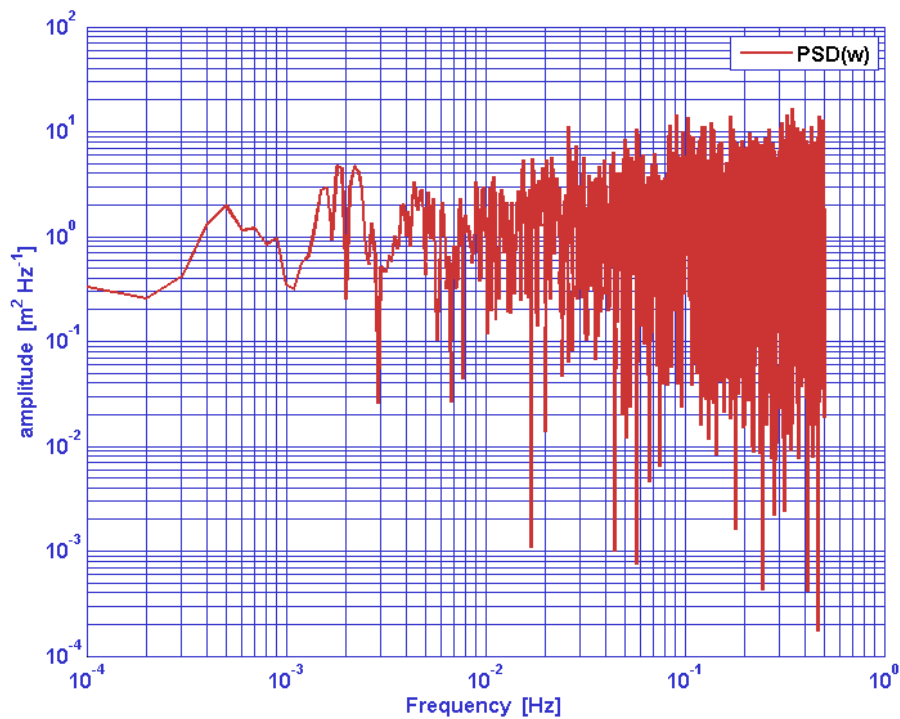
Figure 1: Power spectral density (estimated with the welch method) of a gaussian unitary variance zero mean random process. The process *w(t)* is assumed to have physical units of `m` therefore its power spectral density has physical units of `m^2/Hz`. Note that the power spectral density average value is 2 instead of the expected 1 (unitary variance process) since we calculated one-sided power spectral density.

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

In LTPDA there are different methods for noise whitening:

- buildWhitener1D.m
- whiten1D.m
- firwhiten.m
- whiten2D.m

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

Whitening Algorithms

buildWhitener1D

`buildWhitener1D` performs a frequency domain identification of the system in order to extract the proper whitening filter. The function needs a model for the one-sided power spectral density of the given process. If no model is provided, the power spectral density of the process is calculated with the psd and bin_data algorithm.

1. The inverse of the square root of the model for the power spectral density is fit in z-domain in order to determine a whitening filter.
2. Unstable poles are removed by an all-pass stabilization procedure.
3. Whitening filter is provided at the output.

Whiten1D

`whiten1D` implements the same functionality of `buildWhitener1D` but it adds the filtering step so input data are filtered with the identified filter internally to the method.

Firwhiten

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

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

Whiten2D

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

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

Whitening noise in one-dimensional data

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

```
fs = 1; % sampling frequency

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

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

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

Now we can try to white colored noise.

buildWhitener1D

If you want to try `buildWhitener1D` to get a whitening filter for the present colored noise, you can try the following code. Please refer to the [buildWhitener1D](#) documentation page for the meaning of any parameter. The result of the whitening procedure is reported in figure 2.

```
pl = plist(...
   'MaxIter', 30, ...
   'MinOrder', 9, ...
   'MaxOrder', 15, ...
   'FITTOL', 5e-2);

wfil = buildWhitener1D(af,pl);

aw = filter(af,wfil);
```
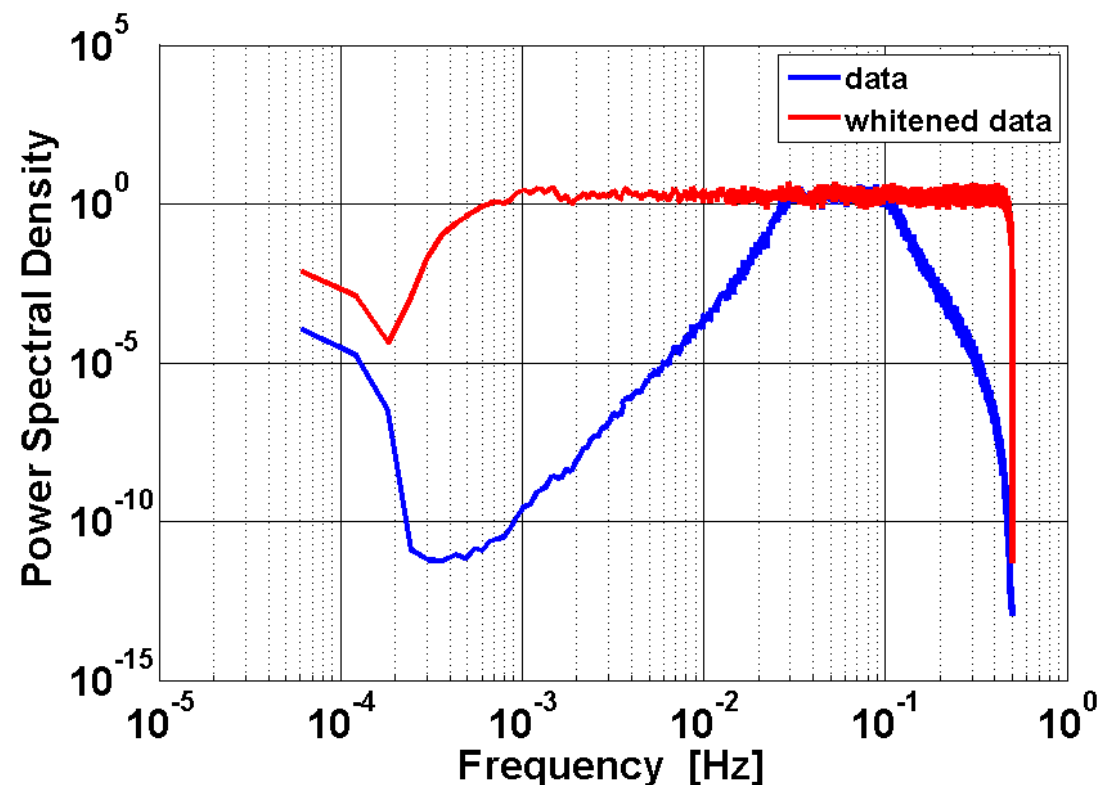

Figure 2: Power spectral density (estimated with the welch method) of colored and whitened processes.

Firwhiten

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

reported in figure 3.

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

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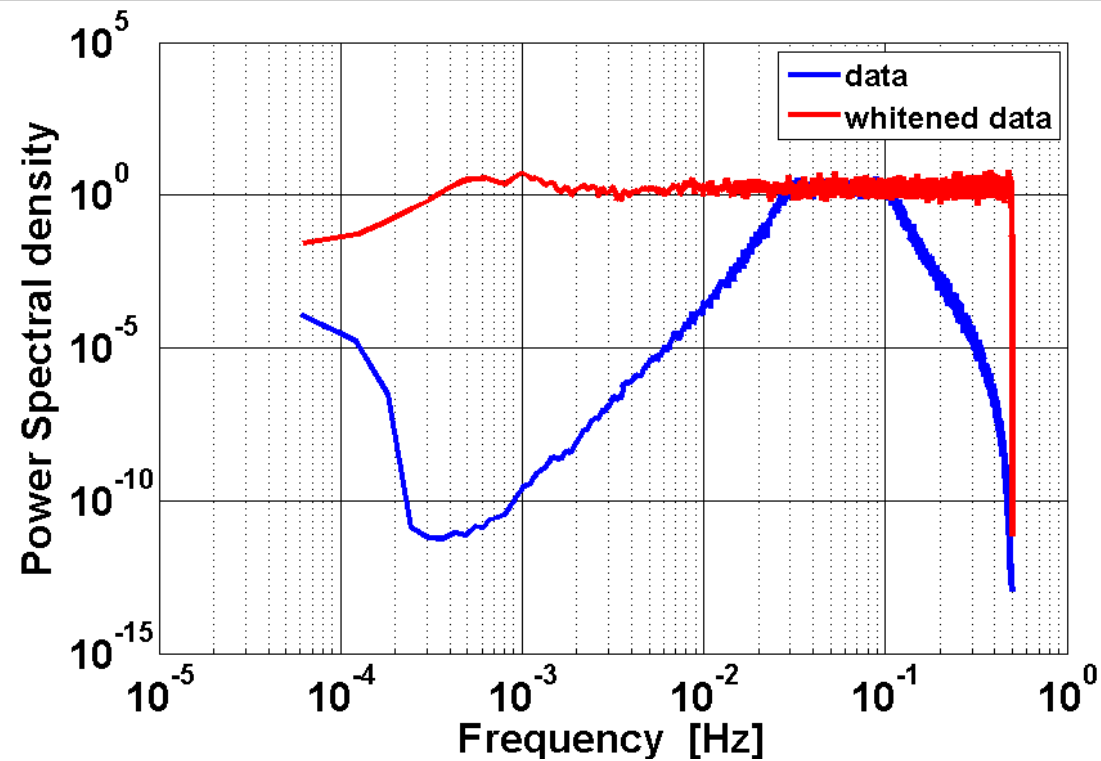

Figure 3: Power spectral density (estimated with the welch method) of colored and whitened processes.

Whitening noise in two-dimensional data

We consider now the problem of whitening cross correlated data series. As a example we consider a typical couple of x–dynamics LTP data series. `a1` and `a2` are interferometer output noise data series. In oreder to whiten data we must input a frequency response model of the cross spectral matrix of the cross–correlated process.

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

Refer to whiten2D documentation page for the meaning of any parameter.

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

```
        'MSEVARTOL', 1e-2,...
        'FITTOL', 1e-3);

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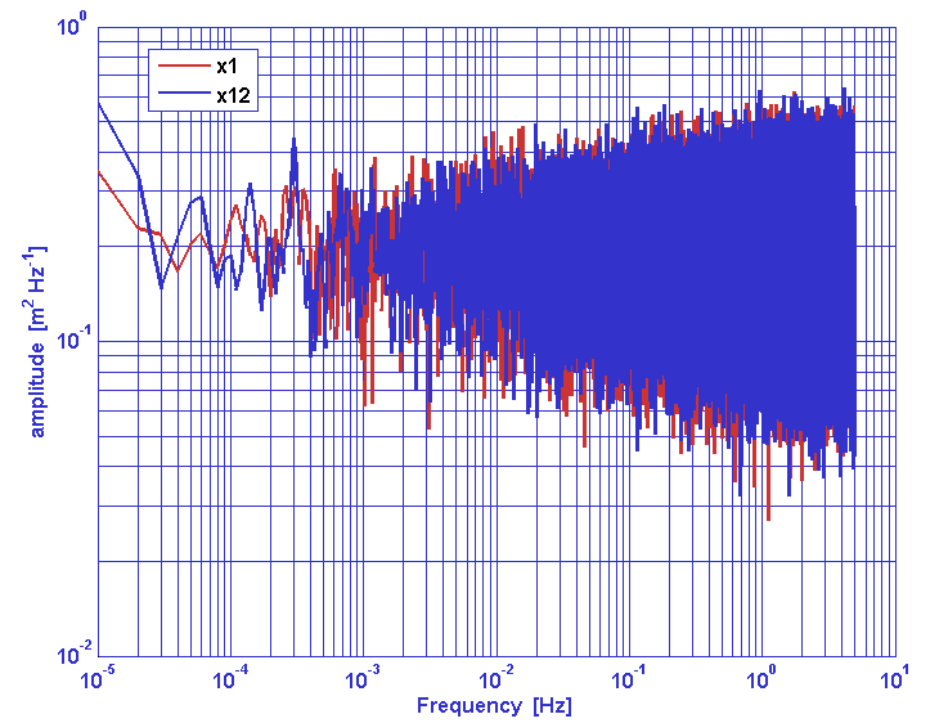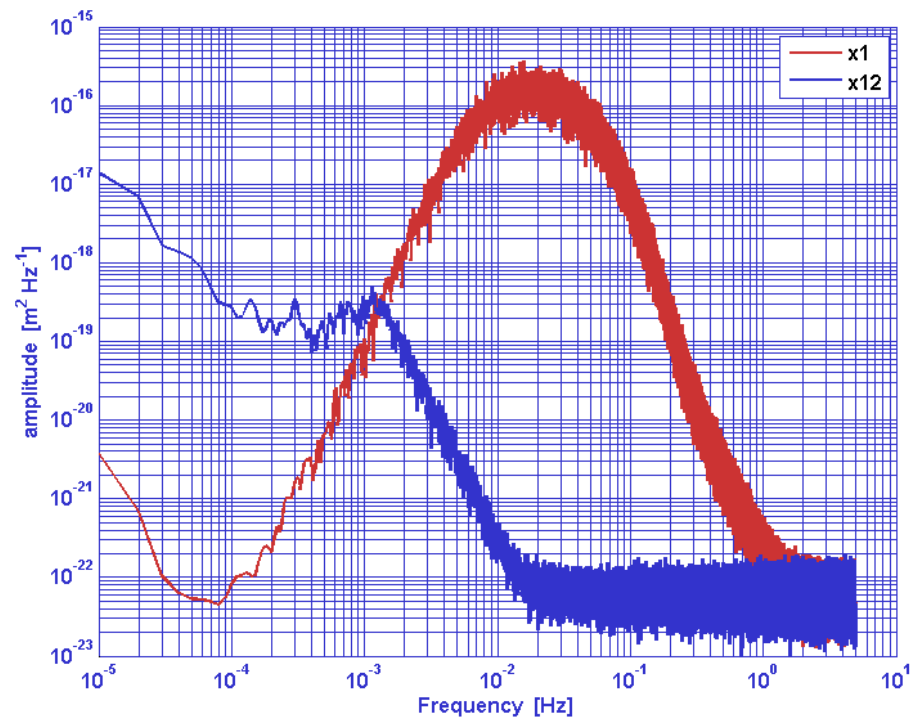

Figure 4: Power spectral density of the noisy data series before (left) and after (right) the whitening.
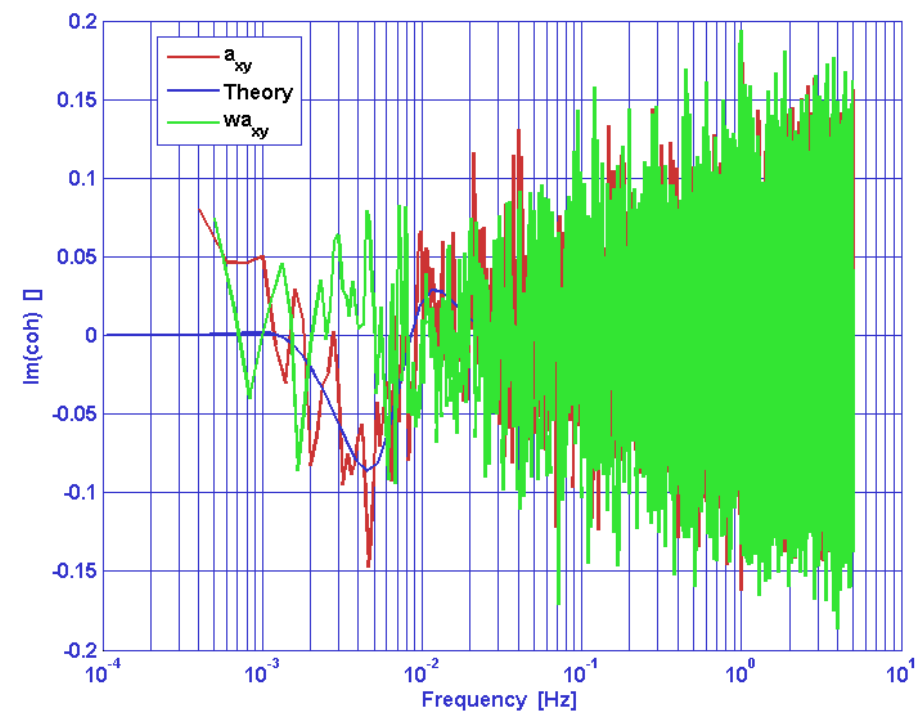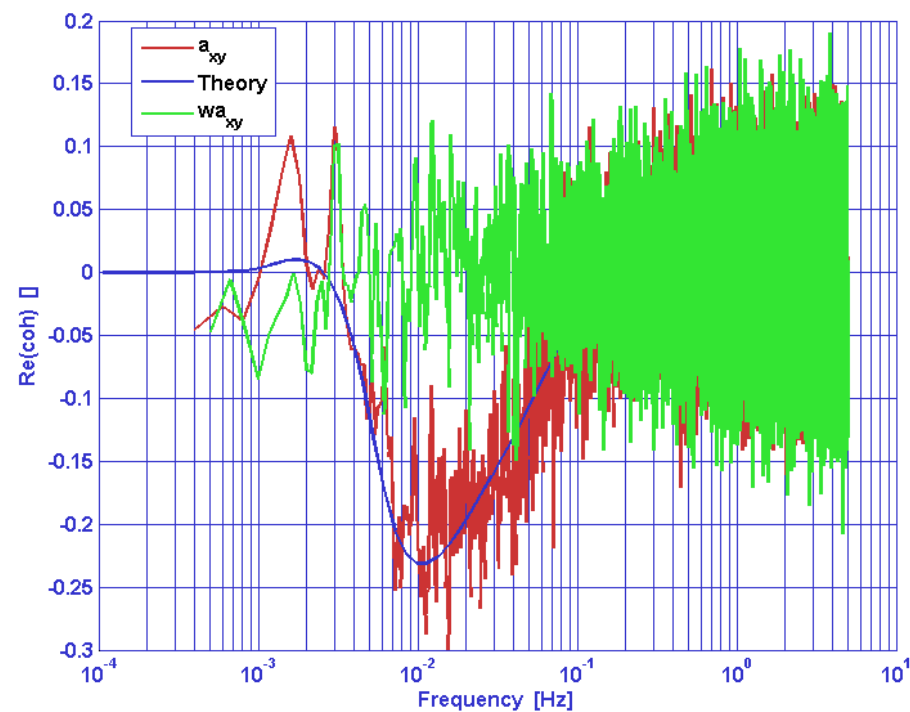
Figure 5: Real (left) and Imaginary (right) part of the coherence function. Blue line refers to theoretical expectation for colored noise data. Red line refers to calculated values for colored noise data. Green line refers to calculated values for whitened noise data.

LTPDA Toolbox™ [contents](#) ◀ ▶

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](#)

◀ Noise whitening   Digital Filtering ▶

©LTP Team

Signal Processing in LTPDA (LTPDA Toolbox)

Digital Filtering

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

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

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

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

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

- Stable: if finite input results in finite output.

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

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

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

Digital filters classification

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

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

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

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

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

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

  MA processes can be also classified as [FIR Filters](#).

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

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

Signal Processing in LTPDA

IIR Filters

©LTP Team

IIR Filters

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

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

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

Creating a IIR filter in the LTPDA

The LTPDA Toolbox allows the implementation of IIR filters by means of the [miir class](#).

Creating from a plist

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

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

Creating from a pzmodel

IIR filters can also be [created from a pzmodel](#) .

Creating from a difference equation

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

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

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

Notice that the 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 ▶

©LTP Team

FIR Filters

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

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

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

Creating a FIR filter in the LTPDA

The LTPDA Toolbox allows the implementation of FIR filters by means of the [mfir class](#).

Creating from a plist

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

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

Creating from a difference equation

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

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

```
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   Filter banks ▶

©LTP Team

Filter banks

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

Description

A filter bank is a container object for collections of MIIR and MFIR filters. The filter bank object can be used to filter a discrete data series in parallel or in series, in accordance to the schematic representations of figures 1 and 2.
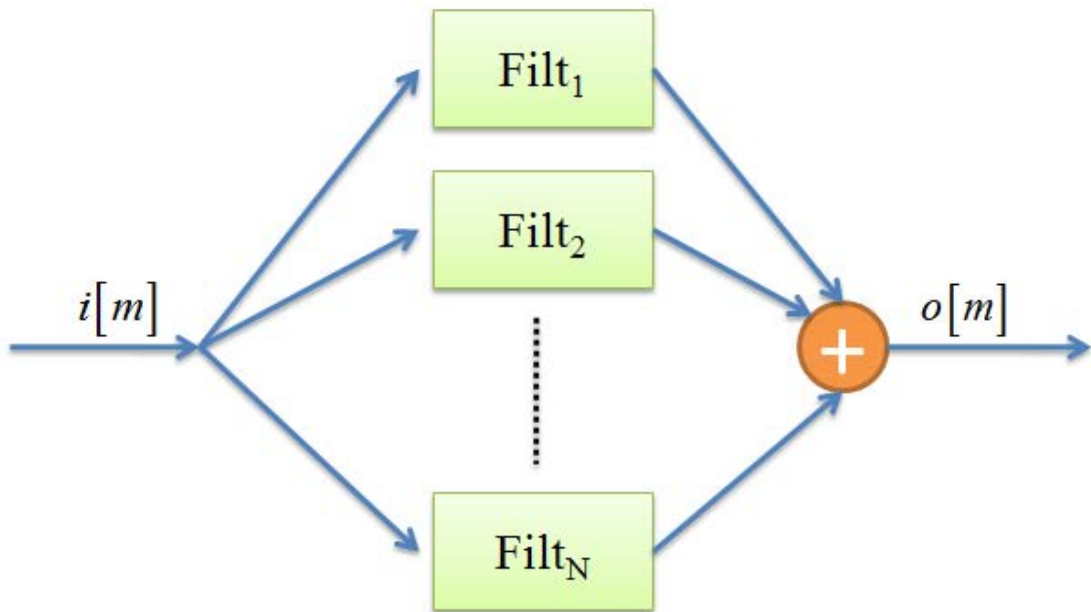
Figure 1: Scheme for parallel filtering. Data is input to the different filters of the bank. Filters output are summed each other in order to obtain the complete output. The different filters of the bank must have the same input and output units.

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

Creating a filter bank in the LTPDA

The LTPDA Toolbox allows the implementation of filter banks by means of the filterbank class.

Creating from a plist

The following example creates a parallel filter bank of MIIR filters.

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

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

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

Creating from filter objects

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

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

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

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

Importing an existing model

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

-
- XML files:

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

- MAT files:

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

- From repository:

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

◀ FIR Filters   Applying digital filters to data ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Applying digital filters to data

[Description](#)	Digital filtering with LTPDA filters objects.
[Examples](#)	How to filter a data series in LTPDA.

Description

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

Examples

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

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

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

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

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

◀ Filter banks   Discrete Derivative ➡

©LTP Team

Discrete Derivative

[Description](#)	Discrete derivative estimation in LTPDA.
[Algorithm](#)	Derivatives Algorithms.
[Examples](#)	Usage examples of discrete derivative estimation tools.
[References](#)	Bibliographic references.

Derivative calculation for dicrete data series

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

Algorithm

Method	Description
'2POINT'	Compute first derivative with two point equation according to:

$$\frac{dy[k]}{dx} \approx \frac{y[k+1]-y[k]}{x[k+1]-x[k]}$$

Method	Description
'3POINT'	Compute first derivative with three point equation according to:

$$\frac{dy[k]}{dx} \approx \frac{y[k+1]-y[k-1]}{x[k+1]-x[k-1]}$$

Method	Description
'5POINT'	Compute first derivative with five point equation according to:

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

Method	Description
'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:

$$y[k] \approx a\,y[k-2] + b\,y[k-1] + c\,y[k] + d\,y[k+1] + g\,y[k+2]$$

$$\frac{dy[k]}{dt} \approx \frac{a'y[k-2] + b'y[k-1] + c'y[k] + d'y[k+1] + g'y[k+2]}{T}$$

$$\frac{d^2 y[k]}{dt^2} \approx \frac{a''y[k-2] + b''y[k-1] + c''y[k] + d''y[k+1] + g''y[k+2]}{T^2}$$

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

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

Examples

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

First derivative

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

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

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

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

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

Second derivative

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

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

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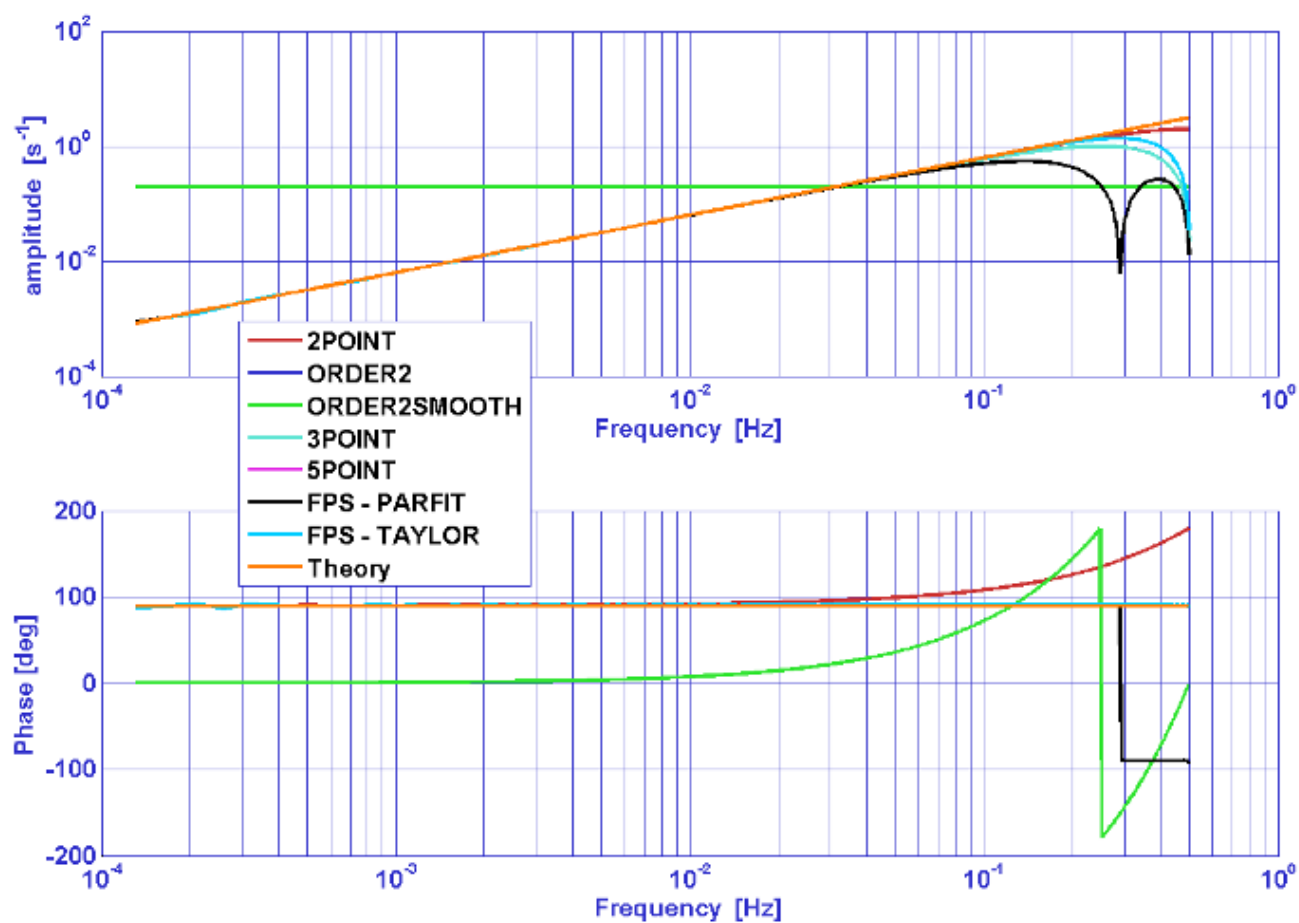

Figure 1: Frequency response of first derivative estimators.
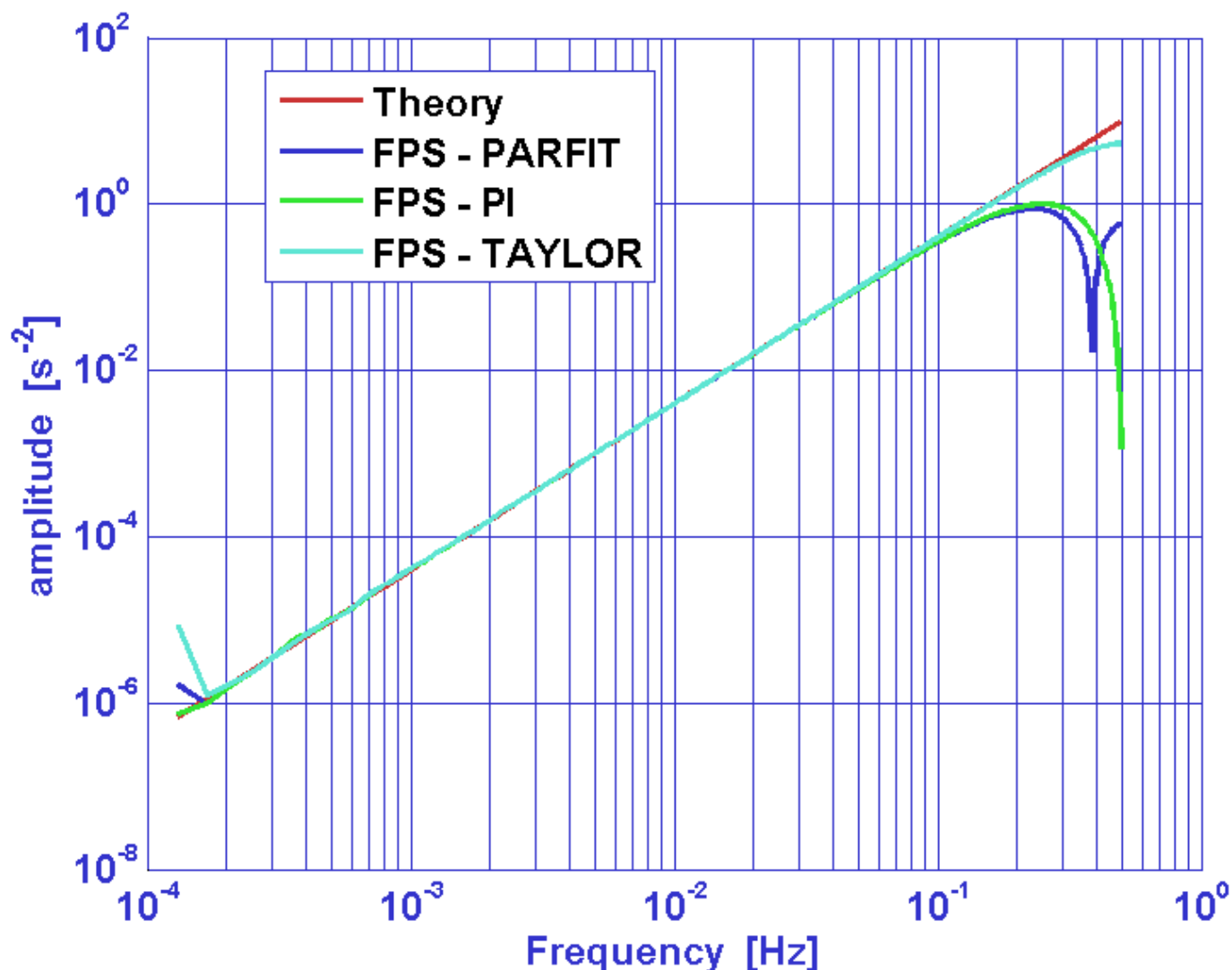
Figure 2: Frequency response of second derivative estimators.

References

1. L. Ferraioli, M. Hueller and S. Vitale, Discrete derivative estimation in LISA Pathfinder data reduction, <u>Class. Quantum Grav. 26 (2009) 094013.</u>.
   L. Ferraioli, M. Hueller and S. Vitale, Discrete derivative estimation in LISA Pathfinder data reduction <u>arXiv:0903.0324v1</u>
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.

◀ *Applying digital filters to data*   *Spectral Estimation* ▶

©LTP Team

Spectral Estimation

Spectral estimation is a branch of the signal processing, performed on data and based on frequency-domain techniques.

Within the LTPDA toolbox many functions of the Matlab [Signal Processing Toolbox](#) (which is required) were rewritten to operate on LTPDA Analysis Objects. Univariate and multivariate technique are available, so to estimate for example the linear power spectral density or the cross spectral density of one or more signals. The focus of the tools is on time-series objects, whose spectral content needs to be estimated.

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

```
>> doc signal
```

Introduction

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

```
>> doc signal
```

in the Matlab terminal.

©LTP Team

Spectral Windows

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

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

- [What are LTPDA spectral windows?](#)
- [Creating spectral windows](#)
- [Visualising spectral windows](#)
- [Using spectral windows](#)

©LTP Team

Spectral Windows (LTPDA Toolbox)

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   Using spectral windows ▶

©LTP Team

LTPDA Toolbox™   [contents](#)

Using spectral windows

Spectral windows are typically used in spectral analysis algorithms. In all LTPDA spectral analysis functions, spectral windows are specified as parameters in an input parameter list. The following example shows the use of `ao/psd` to estimate an Amplitude Spectral Density of the time–series captured in the input AO, `a`. The help for [ao/lpsd](#) reveals that the required parameter for setting the window function is `'Win'`.

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

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

% Add both
x = x1 + x2;

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

% Plot
iplot(z);
```

In this case, the size of the spectral window (number of samples) may not match the length of the segments in the spectral estimation. The `psd` algorithm then recomputes the window using the input design but for the correct length of window function.
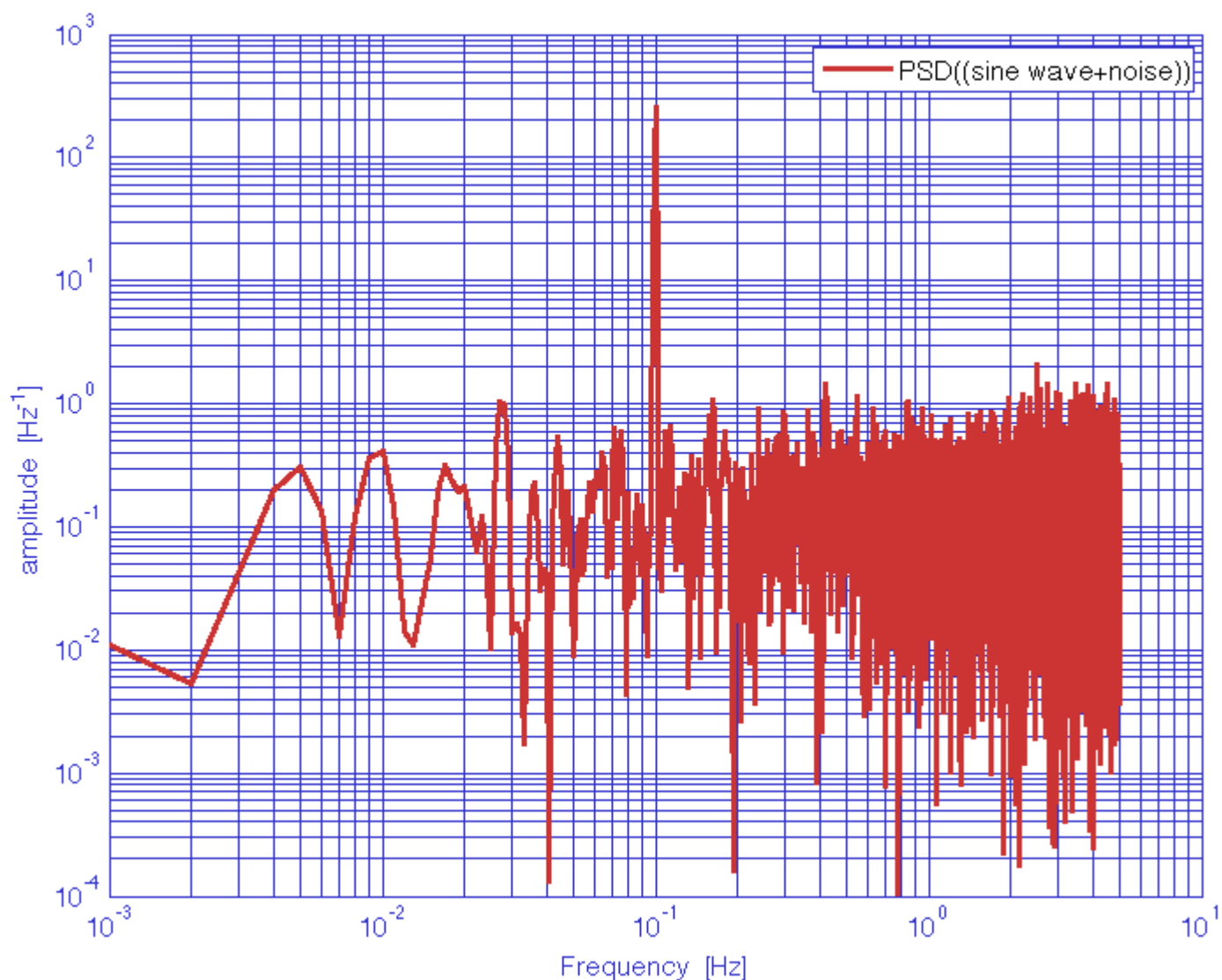
Selecting the Peak Side-Lobe level (psll) with Kaiser's window

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

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

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

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

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

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

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

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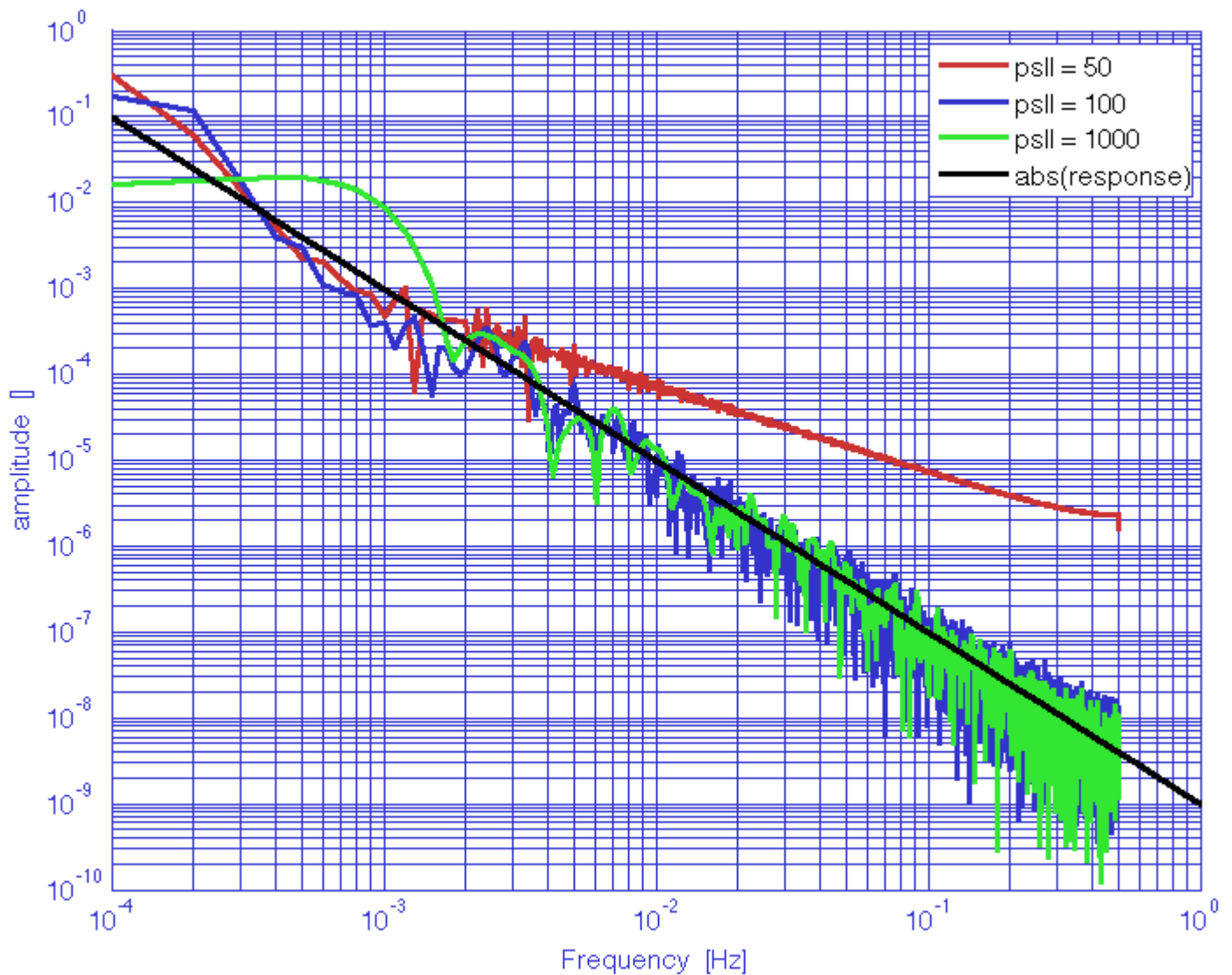
©LTP Team

Spectral Estimation Methods

Linear and Log-scale Methods

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

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

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

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

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

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

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

Computing the sample variance

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

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

The methods listed above store in the `ao.dy` field the **standard deviation of the mean**, defined as

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

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

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

in

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

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

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

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

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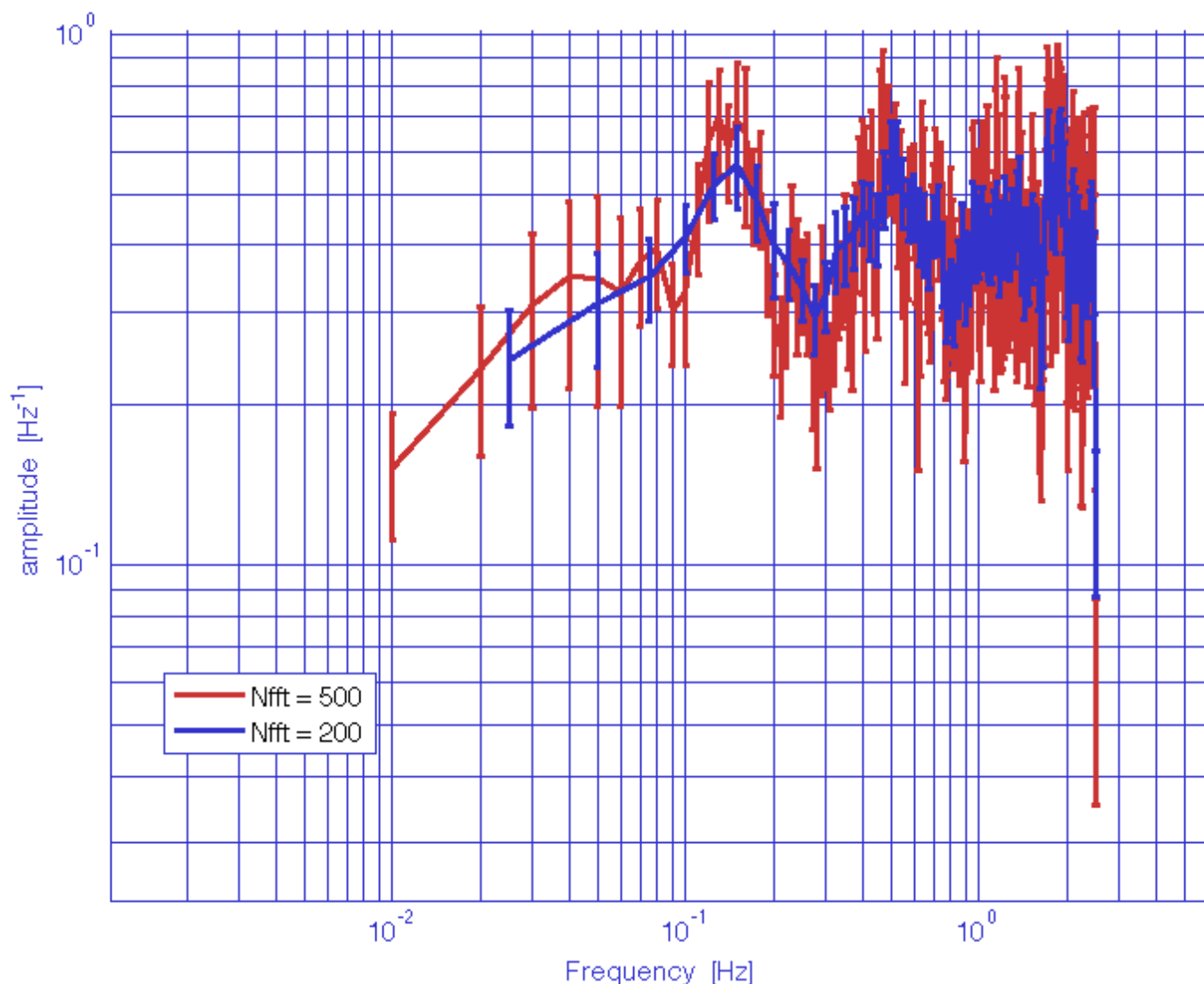

References

1. P.D. Welch, The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Trans. on Audio and Electroacoustics*, Vol. 15, No. 2 (1967), pp. 70 – 73
2. M. Troebs, G. Heinzel, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, *Measurement, Vol. 39 (2006), pp. 120 – 129*. See also the Corrigendum.

Power spectral density estimates

Description

The LTPDA method [ao/psd](#) estimates the power spectral density of time-series signals, included in the input `ao`s following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectrum, by multiplying it with a [spectral window object](#), and can be detrended by a 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) and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] A string value containing the variable 'fs' can also be used, e.g., plist('Nfft', '2*fs')
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: -1, (taken from window function)]
- `'Scale'` – scaling of output. Choose from:
  - 'ASD' – amplitude spectral density
  - 'PSD' – power spectral density [default]
  - 'AS' – amplitude spectrum
  - 'PS' – power spectrum
- `'Order'` – order of segment detrending
  - -1 – no detrending
  - 0 – subtract mean [default]
  - 1 – subtract linear fit
  - N – subtract fit of polynomial, order N
- `'Navs'` – number of averages. If set, and if Nfft was set to 0 or -1, the number of points for each window will be calculated to match the request. [default: -1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.

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

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

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

Algorithm

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

Examples

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

```
% create two AOs
x1   = ao(plist('waveform','sine wave','f',0.1,'A',1,'nsecs',1000,'fs',10));
x2   = ao(plist('waveform','noise','type','normal','nsecs',1000,'fs',10));
% add both AOs
x    = x1 + x2;
% compute the psd changing the 'nfft'
y_lf = psd(x);
y_hf = psd(x,plist('nfft',1000));
% compare
iplot(y_lf, y_hf)
```
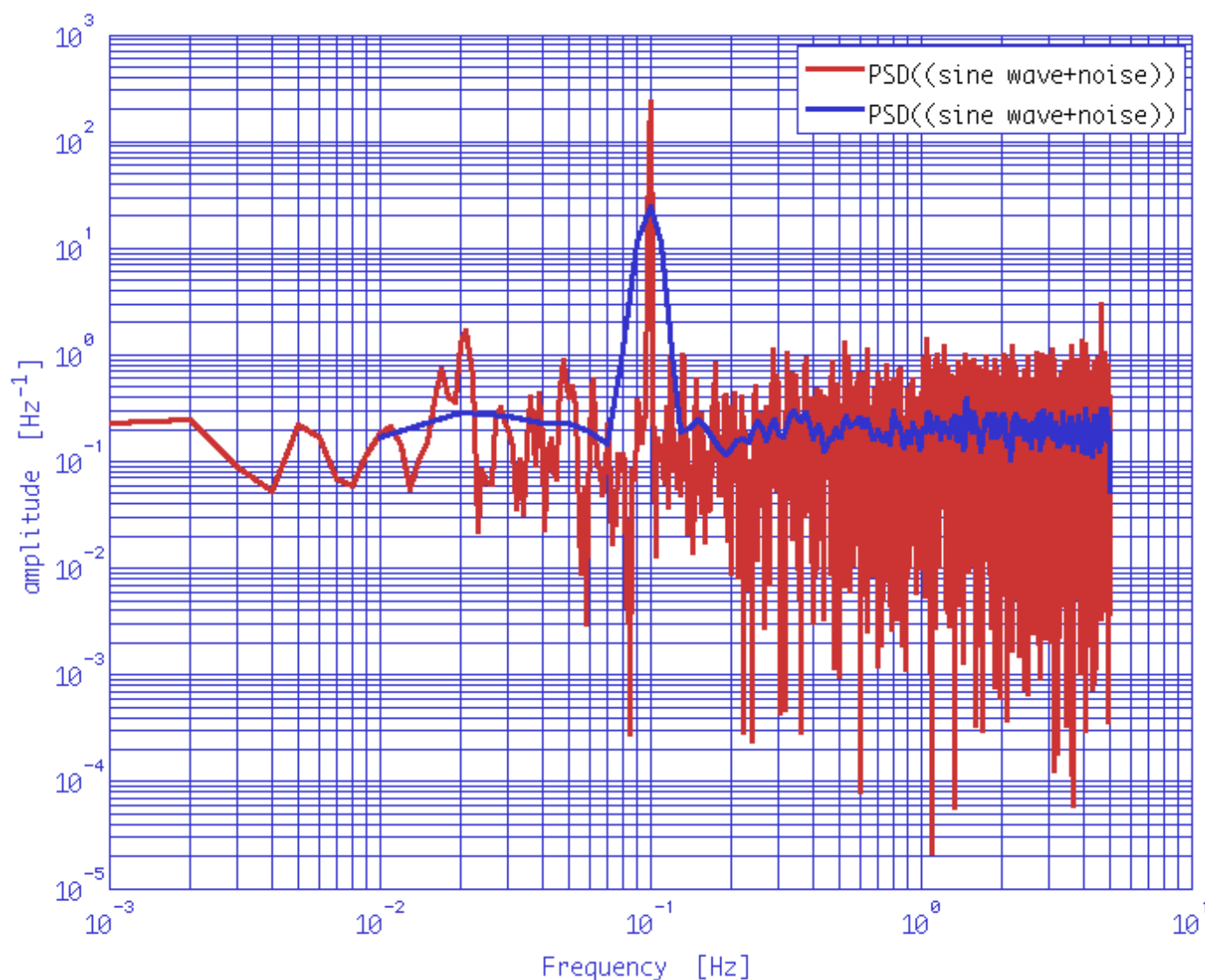
2. Evaluation of the PSD of a time–series represented by a low frequency sinewave signal, superimposed to white noise and to a low frequency linear drift. In the example, the same spectrum is computed with different spectral windows.

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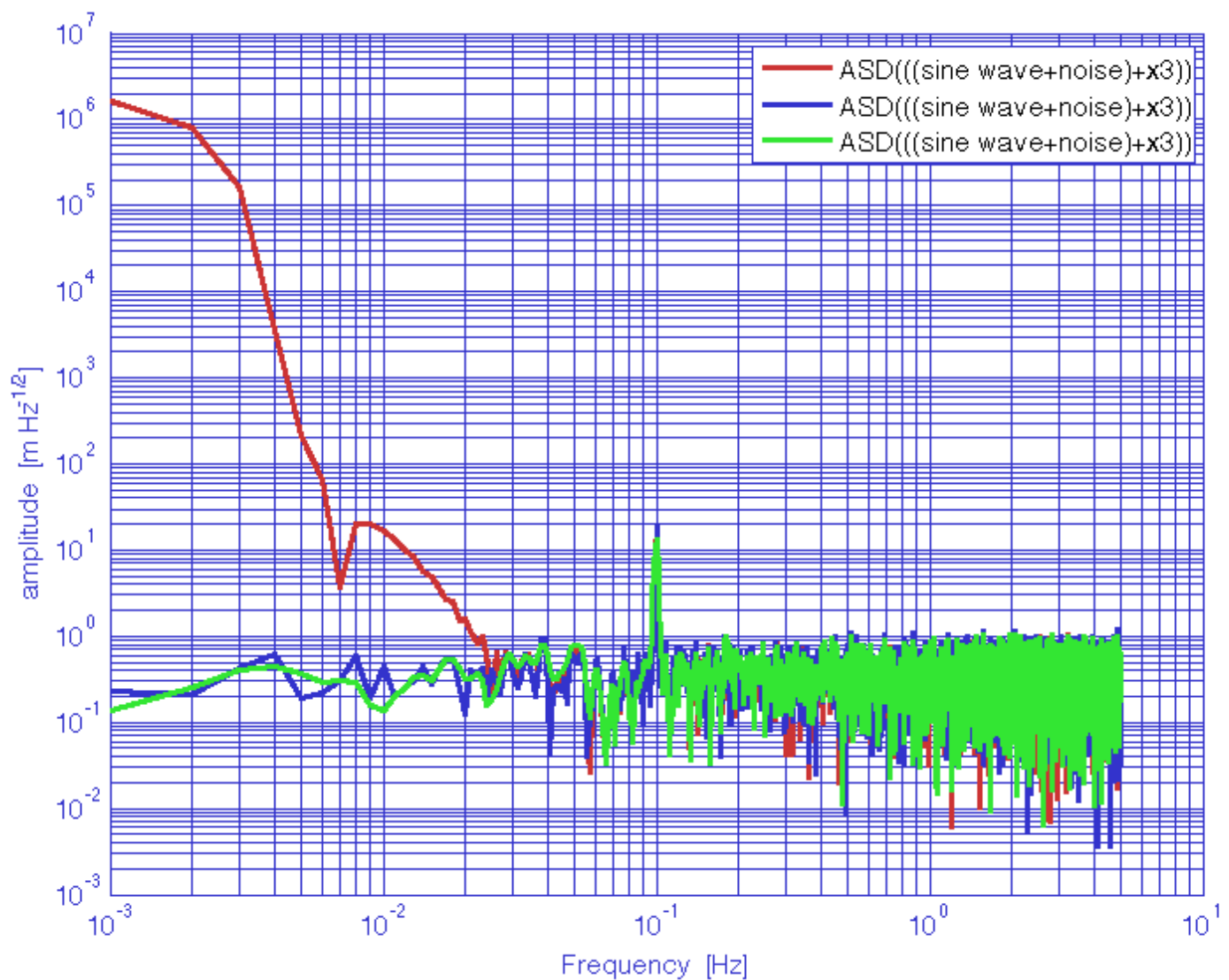
References

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

©LTP Team

Cross-spectral density estimates

Description

Cross-power spectral density is performed by the Welch's averaged, modified periodogram method. The LTPDA method ao/cpsd estimates the cross-spectral density of time-series signals, included in the input `ao`s following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectra, by multiplying it with a spectral window object, and can be detrended by 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.

Syntax

```
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, and `pl` is an optional parameters list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] A string value containing the variable 'fs' can also be used, e.g., plist('Nfft', '2*fs')
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: –1, (taken from window function)]
- `'Order'` – order of segment detrending
  - –1 – no detrending
  - 0 – subtract mean [default]
  - 1 – subtract linear fit
  - N – subtract fit of polynomial, order N
- `'Navs'` – number of averages. If set, and if Nfft was set to 0 or –1, the number of points for each window will be calculated to match the request. [default: –1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.

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

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

subsequent segments.

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

The function makes CPSD estimates between the 2 input `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)`.

Algorithm

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

Example

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

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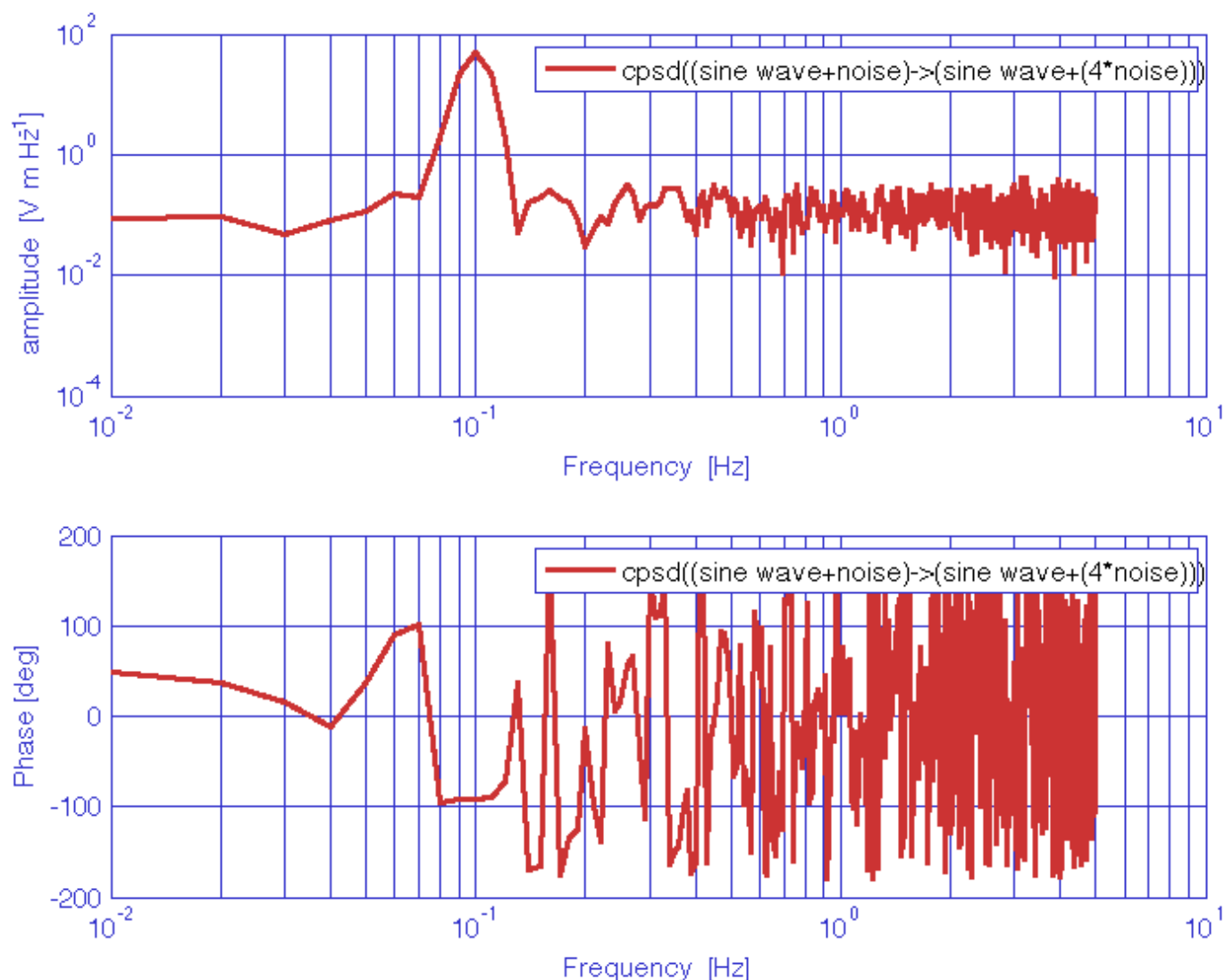
References

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

©LTP Team

Cross coherence estimates

Description

The LTPDA method [ao/cohere](#) estimates the cross-coherence of time-series signals, included in the input `ao`s following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectra, by multiplying it with a [spectral window object](#), and can be detrended by a 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.

Syntax

```
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 and `pl` is an optional parameters list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] Notice: analyzing a single segment produces as a result an object full of 1! A string value containing the variable 'fs' can also be used, e.g., plist('Nfft', '2*fs')
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: –1, (taken from window function)]
- `'Order'` – order of segment detrending
  - –1 – no detrending
  - 0 – subtract mean [default]
  - 1 – subtract linear fit
  - N – subtract fit of polynomial, order N
- `'Navs'` – number of averages. If set, and if Nfft was set to 0 or –1, the number of points for each window will be calculated to match the request. [default: –1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.
- `'Type'` – type of scaling of the coherence function. Choose between:
  - `'C'` – Complex Coherence Sxy / sqrt(Sxx * Syy) [default]
  - `'MS'` – Magnitude-Squared Coherence (abs(Sxy))^2 / (Sxx * Syy)

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

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

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

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

Algorithm

The algorithm is based in standard MATLAB's tools, as the ones used by pwelch. The standard deviation of the mean is computed as [2]

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

where

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

is the coherence function.

Example

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

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

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

% add components
x = x1 + x2 + x3;

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

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

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

%plot
iplot(Cxy);
```
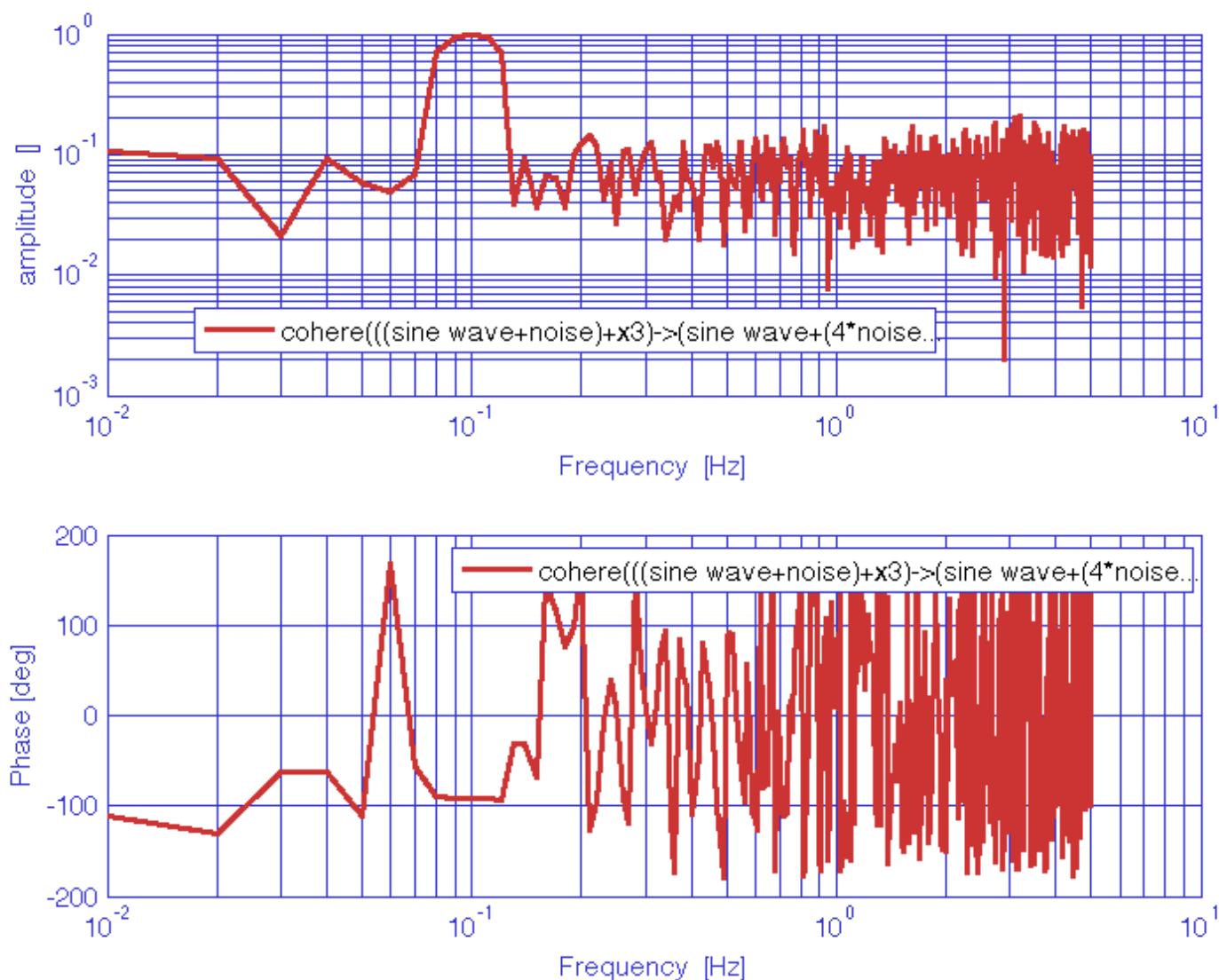
References

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

◀ Cross-spectral density estimates   Transfer function estimates ▶

©LTP Team

Transfer function estimates

Description

The LTPDA method [ao/tfe](#) estimates the transfer function of time-series signals, included in the input `ao`s following the Welch's averaged, modified periodogram method [1]. Data are windowed prior to the estimation of the spectra, by multiplying it with a [spectral window object](#), and can be detrended by 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.

Syntax

```
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 and `pl` is an optional parameters list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Nfft'` – number of samples in each fft [default: length of input data] A string value containing the variable 'fs' can also be used, e.g., plist('Nfft', '2*fs')
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: –1, (taken from window function)]
- `'Order'` – order of segment detrending
  - –1 – no detrending
  - 0 – subtract mean [default]
  - 1 – subtract linear fit
  - N – subtract fit of polynomial, order N
- `'Navs'` – number of averages. If set, and if Nfft was set to 0 or –1, the number of points for each window will be calculated to match the request. [default: –1, not set]
- `'Times'` – interval of time to evaluate the calculation on. If empty [default], it will take the whole section.

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

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

subsequent segments.

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

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

Algorithm

The algorithm is based in standard MATLAB's tools, as the ones used by [pwelch](). The standard deviation of the mean is computed as

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

where

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

is the coherence function.

Example

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

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

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

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

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

% plot
iplot(Txy)
```
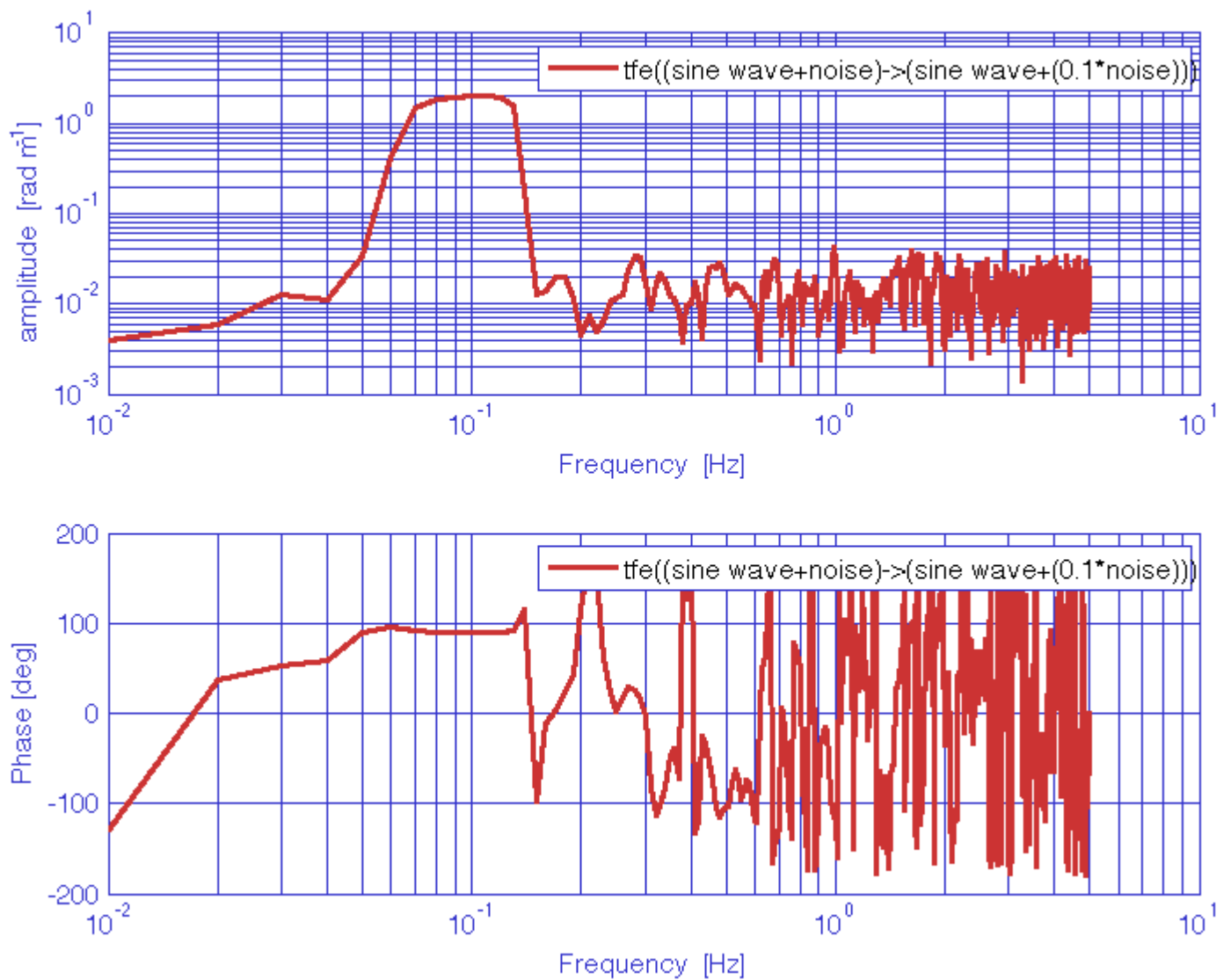
References

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

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

Log-scale power spectral density estimates

Description

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

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

Syntax

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

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

Parameters

The parameter list `pl` includes the following parameters:

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

- −1 – no detrending
- 0 – subtract mean [default]
- 1 – subtract linear fit
- N – subtract fit of polynomial, order N

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

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

Algorithm

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

Examples

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

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

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

 % Compare
iplot(y1, y2)
```
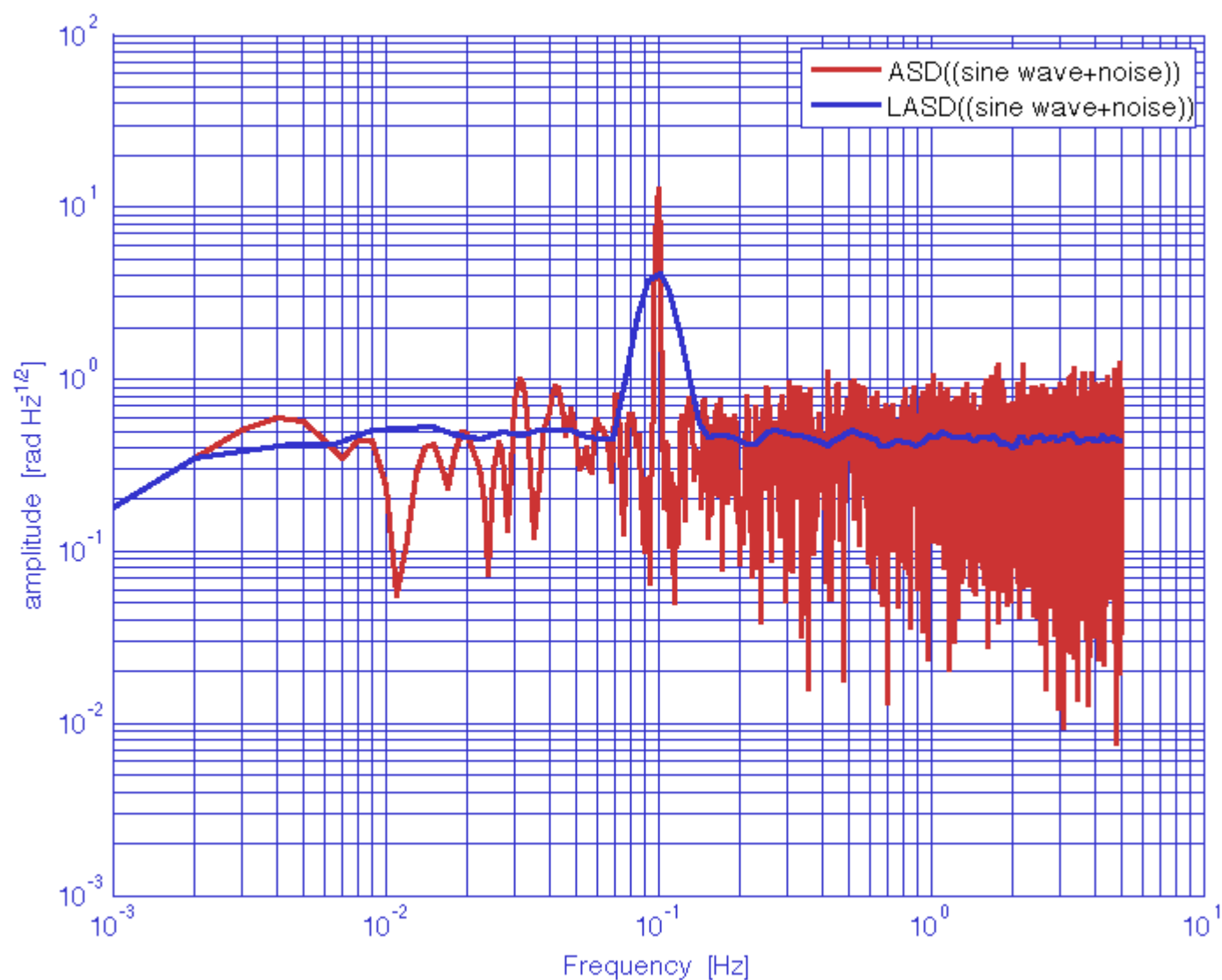
References

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

©*LTP Team*

LTPDA Toolbox™   [contents](#)   ◀ ▶

Log-scale cross-spectral density estimates

Description

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

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

Syntax

```
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 and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Kdes'` – desired number of averages [default: 100]
- `'Jdes'` – number of spectral frequencies to compute [default: 1000]
- `'Lmin'` – minimum segment length [default: 0]
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: -1, (taken from window function)]
- `'Order'` – order of segment detrending
    - -1 – no detrending
    - 0 – subtract mean [default]
    - 1 – subtract linear fit
    - N – subtract fit of polynomial, order N

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

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

The function makes log-scale CPSD estimates between the 2 input `ao`s. 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)`.

Algorithm

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

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

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

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

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

% Plot
iplot(z);
```
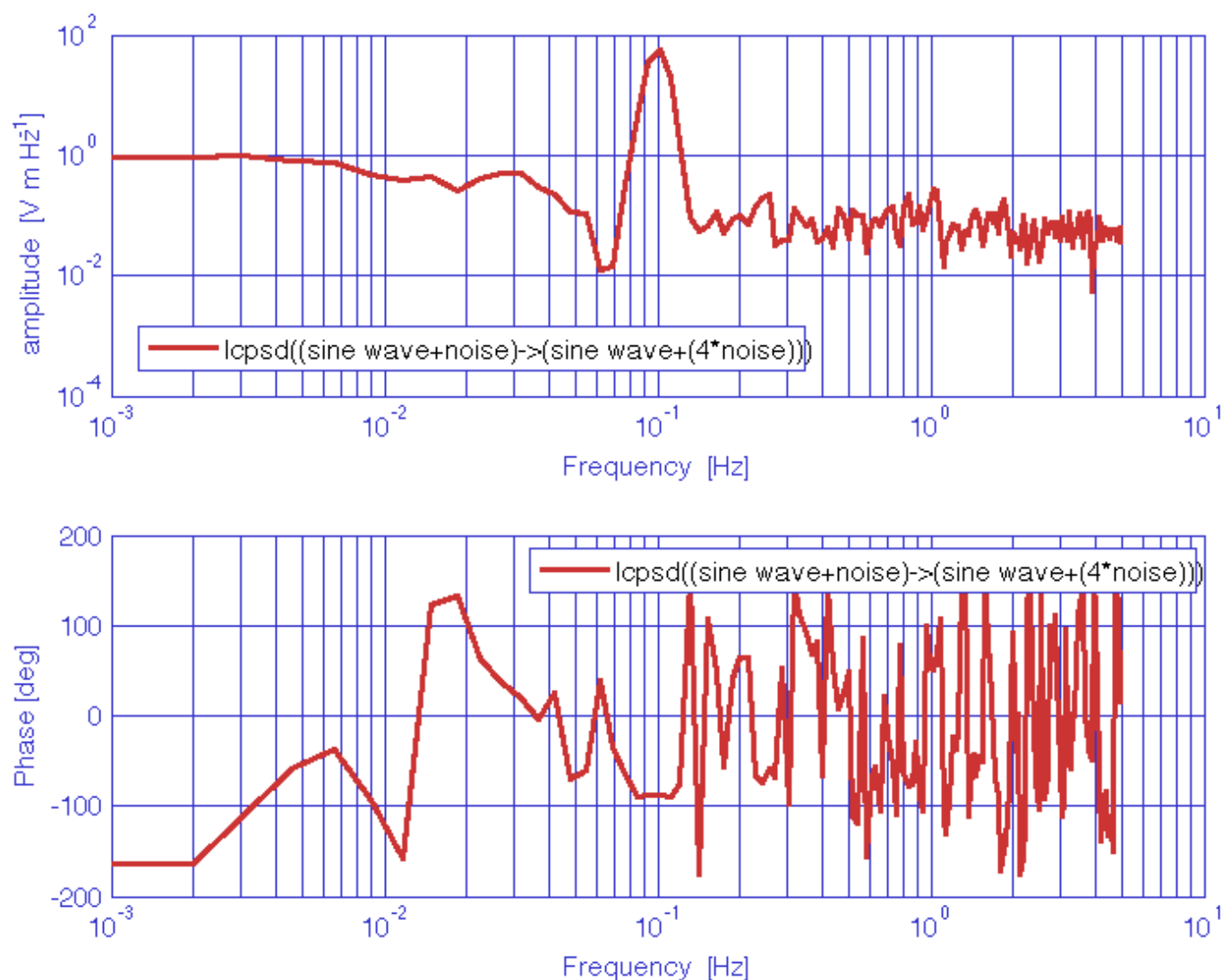
References

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

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

©LTP Team

Log-scale cross coherence density estimates

Description

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

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

Syntax

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

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

Parameters

The parameter list `pl` includes the following parameters:

- `'Kdes'` – desired number of averages [default: 100]
- `'Jdes'` – number of spectral frequencies to compute [default: 1000]
- `'Lmin'` – minimum segment length [default: 0]
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: –1, (taken from window function)]
- `'Order'` – order of segment detrending
  - –1 – no detrending
  - 0 – subtract mean [default]
  - 1 – subtract linear fit
  - N – subtract fit of polynomial, order N
- `'Type'` – type of scaling of the coherence function. Choose between:
  - `'C'` – Complex Coherence Sxy / sqrt(Sxx * Syy) [default]
  - `'MS'` – Magnitude–Squared Coherence (abs(Sxy))^2 / (Sxx * Syy)

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

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

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

Algorithm

The algorithm is implemented according to [1]. The standard deviation of the mean is computed according to [2]:

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

where

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

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

Example

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

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

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

% Plot
iplot(Cxy);
```
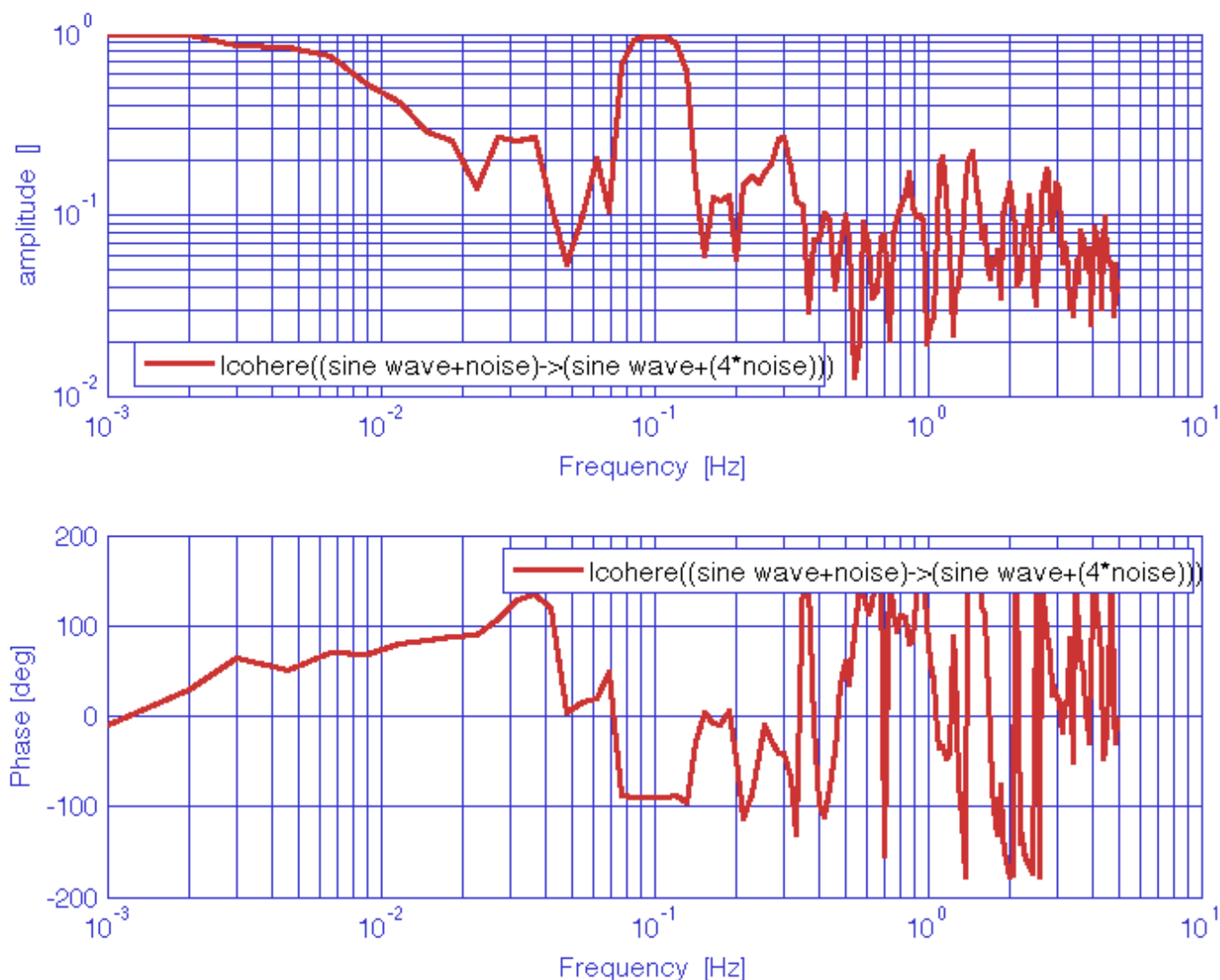
References

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

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

©LTP Team

Log-scale transfer function estimates

Description

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

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

Syntax

```
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 and `pl` is an optional parameter list.

Parameters

The parameter list `pl` includes the following parameters:

- `'Kdes'` – desired number of averages [default: 100]
- `'Jdes'` – number of spectral frequencies to compute [default: 1000]
- `'Lmin'` – minimum segment length [default: 0]
- `'Win'` – the window to be applied to the data to remove the discontinuities at edges of segments. [default: taken from user prefs].
  The window is described by a string with its name and, only in the case of Kaiser window, the additional parameter `'psll'`.
  For instance: plist('Win', 'Kaiser', 'psll', 200).
- `'Olap'` – segment percent overlap [default: –1, (taken from window function)]
- `'Order'` – order of segment detrending
    - –1 – no detrending
    - 0 – subtract mean [default]
    - 1 – subtract linear fit
    - N – subtract fit of polynomial, order N

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

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

The function makes 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.

Algorithm

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

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

where

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

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

Example

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

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

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

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

% Plot
iplot(Txy);
```
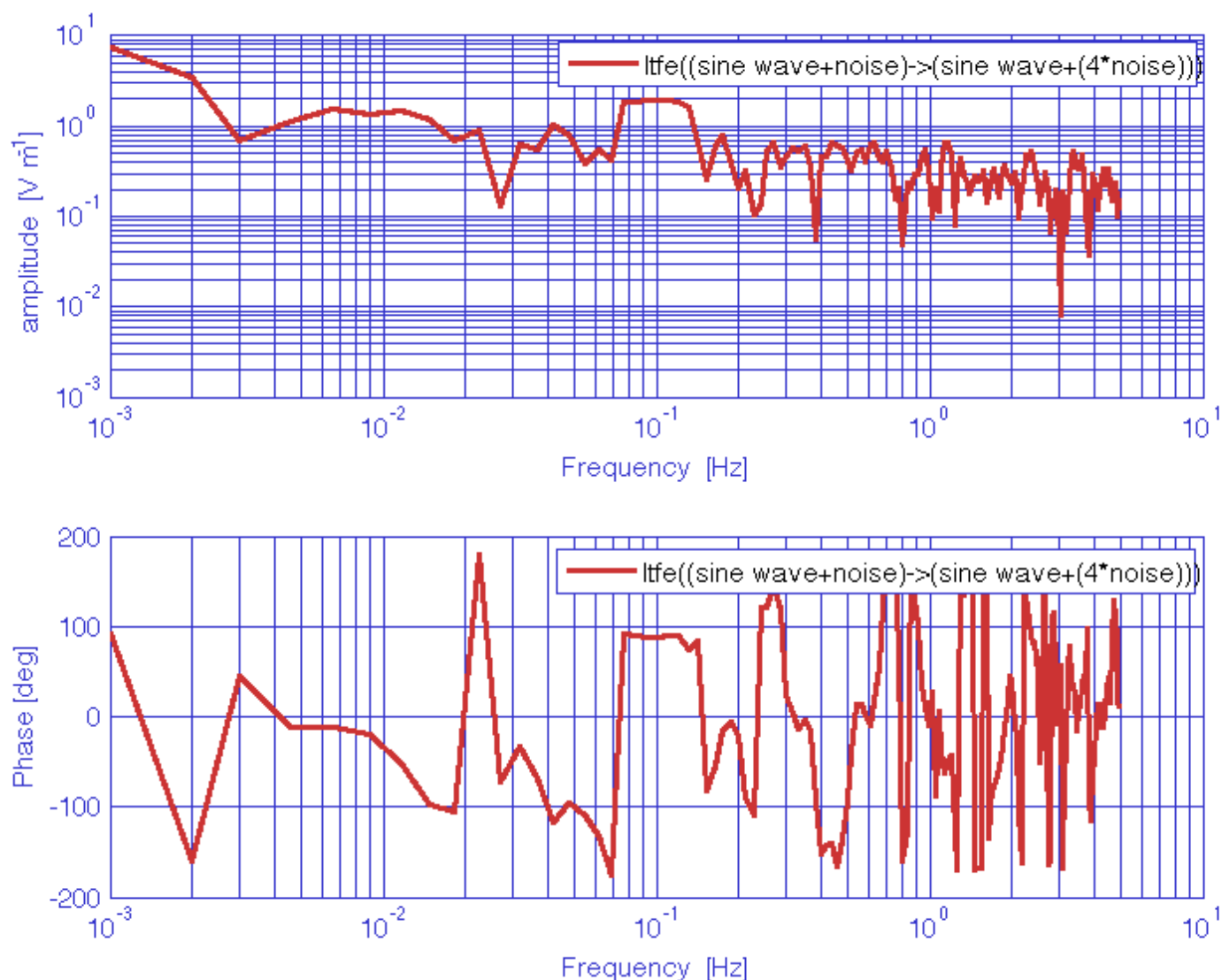
References

1. M. Troebs, G. Heinzel, Improved spectrum estimation from digitized time series on a logarithmic frequency axis, *Measurement*, Vol. 39 (2006), pp. 120 – 129. See also the Corrigendum.

◀ Log–scale cross coherence density estimates   Fitting Algorithms ▶

©LTP Team

Fitting Algorithms

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

- [Polynomial Fitting](#)
- [Time Domain Fit](#)
- [Linear Parameter Estimation with Singular Value Decomposition](#)
- [Z–Domain Fit](#)
- [S–Domain Fit](#)

Fitting Algorithms (LTPDA Toolbox)

Polynomial Fitting

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

- polyfit.m
- polyval.m

Usage

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

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

Markov Chain Monte Carlo

[ao/mcmc](#)	MCMC – single/multiple experiments with AOs as inputs.
[matrix/mcmc](#)	MCMC – single/multiple experiments with matrix objects as inputs.
[References](#)	

The following section is dedicated to Bayesian parameter estimation techniques. Theoretical aspects are explained and some usefull examples are provided.

A little bit of theory.

The MCMC is a statistical method that is based on random sampling from the parameter space and it is considered suitable and efficient when the parameter space is multidimensional. New samples are generated using a Markov Chain mechanism and in each jump (or step) in the parameter space, the likelihood is calculated. After a sufficient number of samples, one can investigate the shape of the likelihood surface.

Syntax

```
% call the method
  p = mcmc(out,plist);
```

Where, `out` is an AO or matrix object of the measured signal (output of the system).

Parameters

The parameter list `plist` includes the following parameters:

- `'Model'` – Input model to the algorithm.
- `'FitParams'` – A cell array of evaluated parameters.
- `'input'` – A matrix array (or vector of AOs) of input signals.
- `'noise'` – A matrix array (or vector of AOs) of noise spectrum (PSD) used to compute the likelihood.
- `'N'` – total number of samples of the chain.
- `'cov'` – covariance of the gaussian jumping distribution.
- `'search'` – Set to true to use bigger jumps in parameter space during annealing and cool down.
- `'frequencies'` – Range of frequencies where the analysis is performed. If an array, only first and last are used.
- `'fsout'` – Desired sampling frequency to resample the input time series.
- `'simplex'` – Set to true to perform a simplex search to find the starting parameters of the MCMC chain.
- `'mhsample'` – Set to `true` to perform a mhsample search. This is set to `true` by default. Only to be set to `false` by the user if we does not want to perform the mcmc search.
  `'heat'`

- – The heat index flattening likelihood surface during annealing.
- `'x0'` – The proposed initial values. A vector of values.
- `'jumps'` – An array of four numbers setting the rescaling of the covariance matrix during the search phase. The first value is the one applied by default, the following thhree apply just when the chain sample is `mod(10)`, `mod(25)` and `mod(100)` respectively.
- `'plot'` – Select indexes of the parameters to be plotted. For example, if the user desires to plot all 6 chains, he should set the key to `[2 3 4 5 6 7]`. In the first chain the log-likelihood values are stored.
- `'debug'` – Set to true to get debug information of the MCMC process.
- `'Navs'` – The number of averages to use when calculating PSD and CPSD.
- `'Tc'` – an array of two values setting the initial and final value for the cooling down.
- `'prior'` – Mean, sigma and normalization factor for priors. Still under test.
- `'range'` – Range where the parameeters are sampled. A cell array containing a vector `[a b]` for each parameter. The algorithm searches for the parameter between `a` and `b`.
- `'inNames'` – Input names. Used for ssm models
- `'outNames'` – Output names. Used for ssm models

Extra attention is needed when filling the `input`, `noise` and `out` fields. The objects must be organized in a arrays or matrices with a special way. The columns of a matrix are the numbers of experiments and the rows are the number of channels. In the case of LTP we have two channels. If we want to import two experiments, then for each key we specify a 2x2 AOs object or a 2x2 matrix object.

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

Examples

There is one simple example for the use of mcmc method estimating parameters of a simple harmonic oscillator ssm model. For a more complex application (full LTP ssm model) refer to the LTPDA training 2, Topic 5.

References

1. M Nofrarias et al, Bayesian parameter estimation in the second LISA Pathfinder mock data challenge, PHYSICAL REVIEW D 82, 122002 (2010)
2. M Nofrarias, L Ferraioli, G Congedo, Comparison of parameter estimates results in STOC Exercise 6, S2-AEI-TN-3070.

Markov Chain Monte Carlo – Simple experiment

In this simple experiment we use a harmonic oscillator ssm model to simulate date from an input and then we use MCMC to recover the parameters of the model.

Contents

- [Create data](#)
- [Simulate the data](#)
- [Declare our model](#)
- [Calculate covariance matrix](#)
- [Do fit with MCMC](#)

Create data

```
fs    = 10;
nsecs = 300;
fin = 0.2

% Create a sinusoidal input and arrange it to matrix object
in = ao(plist('waveform', 'sine wave', 'fs', fs, 'nsecs', nsecs, 'f', fin);
min = matrix(in,plist('shape',[1 1]));
% we do the same for the noise.
ns = ao(plist('waveform', 'noise', 'sigma', 0.1, 'nsecs', nsecs, 'fs', fs));
mnoise = matrix(ns,plist('shape',[1 1]))
```

Simulate the data

```
% declare some variables here
damp = 0.1;
k = 0.1;
values = [k , damp];

% define inNames  and outNames for ssm model. Also declare parameters to be fitted.
inNames = {'COMMAND.force' };
outNames = {'HARMONIC_OSC_1D.position'};
params = {'DAMP','K'};

% The model to simulate the inputs specified above.
mod = ssm(plist('built-in','HARMONIC_OSC_1D',...
'Version','Fitting',...
'Continuous',1,...
'SYMBOLIC PARAMS',params));

% Set the parameters k and damp to the ssm model.
mod.setParameters(plist('names',params,'values',values));

% Modify time step and make the model numerical.
mod.modifyTimeStep('newtimestep', 0.1);

% Generate covariance
mod.generateCovariance;

% Simulate the output after defining the plist.
pl = plist('Nsamples', fs*nsecs, ...
'aos variable names',{'COMMAND.force' 'NOISE.readout'},...
'aos', [in ns],...
'return outputs', outNames,...
'cpsd variable names', cov.find('names'), ...
'cpsd', cov.find('cov'), ...
```

```
'displayTime', true);

out = mod.simulate(pl);
```

Declare our model

```
model = ssm(plist('built-in','HARMONIC_OSC_1D',...
'Version','Fitting',...
'Continuous',1,...
'SYMBOLIC PARAMS',params));
```

```
% if we type model in the terminal we will see
% all the information available for the model we just created

M:   running display
------ ssm/1 ------
amats: {  [2 x2 ]  }  [1x1]
bmats: {  [2 x1 ]    []    }  [1x2]
cmats: {  [1 x2 ]  }  [1x1]
dmats: { [0] [1] }  [1x2]
timestep: 0
inputs:  [1x2 ssmblock]
1 : COMMAND | force [kg m s^(-2)]
2 : NOISE | readout [m]
states:  [1x1 ssmblock]
1 : HARMONIC_OSC_1D | x [m], xdot [m s^(-1)]
outputs:  [1x1 ssmblock]
1 : HARMONIC_OSC_1D | position [m]
numparams: (M=1)
params: (K=1, DAMP=1)
Ninputs: 2
inputsizes: [1 1]
Noutputs: 1
outputsizes: 1
Nstates: 1
statesizes: 2
Nnumparams: 1
Nparams: 2
isnumerical: false
hist: ssm.hist
procinfo: []
plotinfo: []
name: HARMONIC_OSC_1D
description: Harmonic oscillator
mdlfile:
UUID: b94172bc-9851-42cd-b56a-ca05503165ad
-------------------
```

Calculate covariance

```
ranges = [1e-8 1e-8 ;
0.5  0.5];

pl = plist('FitParams',params,...
'paramsValues',values,...
'inNames',inNames,...
'ngrid',20,...
'stepRanges',ranges,...
'outNames',outNames,...
'noise',mnoise,...
'f1',1e-4,...
'f2',1,...
'model',mod,...
'Navs',5,...
'pinv',false);

mcrb = crb(min,pl);
```

Fit with MCMC

```matlab
    % First define some variables.
    c = 1;
    ranges = {[-3 3] [-3 3]};
    h = 3;
    Tc = [20 80];
    numsampl = 200;

    % Then we create the plist for the mcmc.
    pl = plist('N',numsampl,...
    'J',1,...
    'cov',c*mcrb.y,...
    'range',ranges,...
    'Fitparams',params,...
    'input',min,...
    'noise',mnoise,...
    'model',model,...
    'inNames',inNames,...
    'outNames',outNames,...
    'frequencies',[1e-4 0.5],...
    'fsout',0.2,...
    'Navs',5,...
    'search',true,...
    'Tc',Tc,'heat',h,...
    'jumps',[2e0 1e1 5e2 1e3],...
    'x0',values,...
    'simplex',false);

    [b smplr]= mcmc(out,pl);
```

There is also the possibility to use a desired proposal distribution. The user may choose to draw samples from a distribution that is well fitted to the current investigation. An example is shown below:

```matlab
% Suppose that we want to investigate the same problem as above.
% The plist now should include some extra parameters:

% First of all the function that draws the samples from the
% desired distribution. It must be based on the covariance matrix and
% the current position of the chain mu. For example:
propsamp = @(mu,cov) drawSamplesFromMySampler(arg1,arg2,...,mu,cov);

% Secondly, if the proposal PDF is not symmetric, we should add a function
% that calculates the probability density on the new point x.
% If left empty, then symmetricity is assumed
propdpf = @(x,mu,cov) calcDensityFromMyPDF(x,mu,cov,arg1,arg2,...,argN);

    pl = plist('N',numsampl,...
    'J',1,...
    'cov',c*mcrb.y,...
    'range',ranges,...
    'Fitparams',params,...
    'input',min,...
    'noise',mnoise,...
    'model',model,...
    'inNames',inNames,...
    'outNames',outNames,...
    'proposal sampler',propsamp,...
    'proposal pdf',propdpf,...
    'frequencies',[1e-4 0.5],...
    'fsout',0.2,...
    'Navs',5,...
    'search',true,...
    'Tc',Tc,'heat',h,...
    'jumps',[2e0 1e1 5e2 1e3],...
    'x0',values,...
    'simplex',false);
```
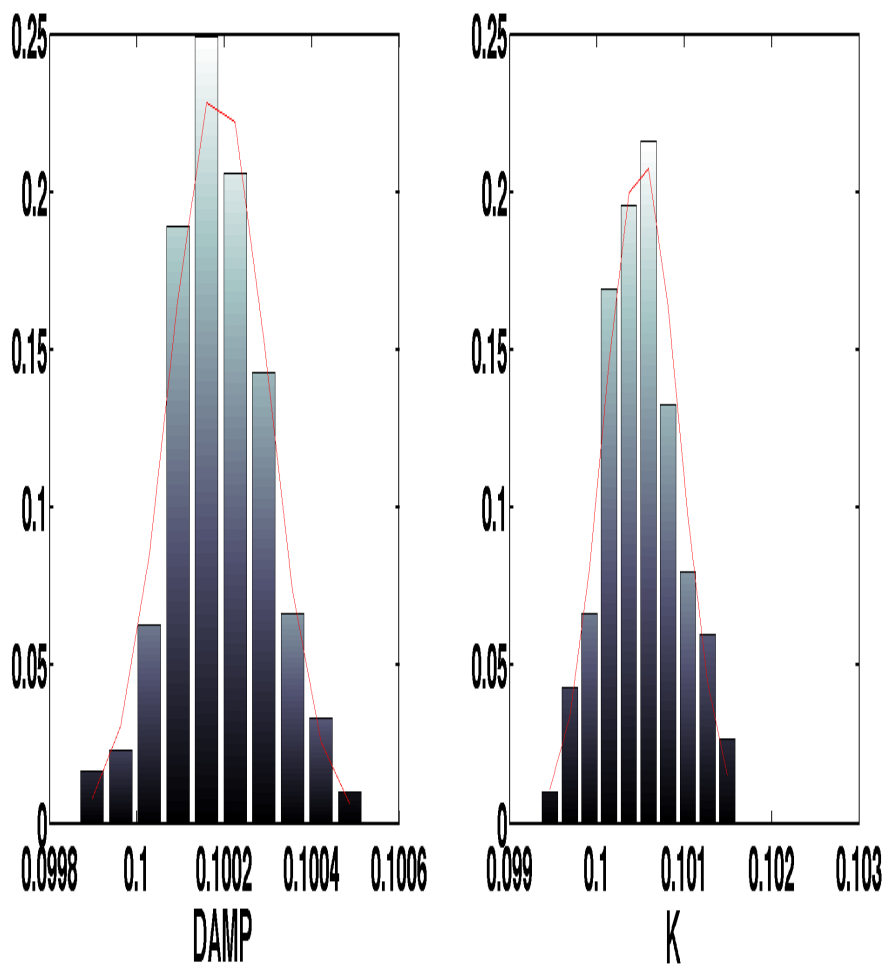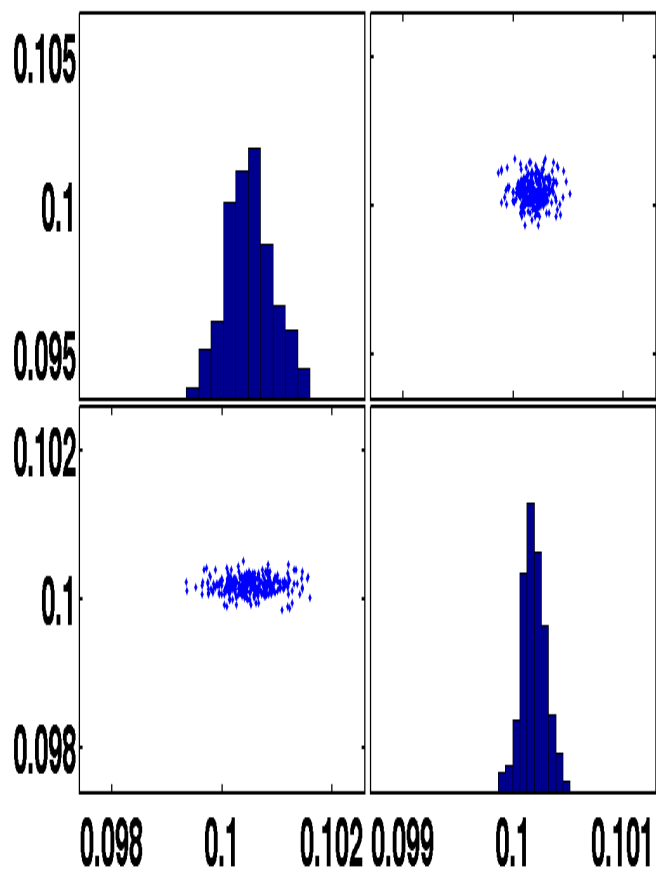
©LTP Team

Linear Parameter Estimation with Singular Value Decomposition

[ao/linlsqsvd](#)	Linear least squares with singular value deconposition – single experiment.
[matrix/linlsqsvd](#)	Linear least squares with singular value deconposition – multiple experiments.
[matrix/linfitsvd](#)	Iterative linear parameter estimation for multichannel systems – symbolic system model in frequency domain.
[matrix/linfitsvd](#)	Iterative linear parameter estimation for multichannel systems – ssm system model in time domain.
[References](#)	

The following sections gives an introduction to the linear parameters estimation methods based on singular value decomposition.

Linear least squares with singular value deconposition – single experiment.

We report an [example](#) of the application of [ao/linlsqsvd](#). The [example](#) shows how to perform a linear parameters estimation for a single data series which is representing the output of an experiment on a given physical system.

Linear least squares with singular value deconposition – multiple experiments.

We report an [example](#) of the application of [matrix/linlsqsvd](#). The [example](#) shows how to perform a linear parameters estimation for multiple data series which are representing the output of multiple experiments on a given physical system.

Iterative linear parameter estimation for multichannel systems – symbolic system model in frequency domain.

We report an [example](#) of the application of [matrix/linfitsvd](#). The [example](#) shows how to perform an iterative linear parameters estimation for a multichannel system. System model is analystic and frequency domain. Fit is performed in time domain. Further details can be found in ref. [1].

Iterative linear parameter estimation for multichannel systems – ssm system model in time domain.

We report an [example](#) of the application of [matrix/linfitsvd](#). The [example](#) shows how to perform an iterative linear parameters estimation for a multichannel system. System model is

ssm and time domain. Fit is performed in time domain.

References

1. M Nofrarias, L Ferraioli, G Congedo, Comparison of parameter estimates results in STOC Exercise 6, S2-AEI-TN-3070.

©LTP Team

LTPDA Toolbox™

Linear least squares with singular value deconposition – single experiment

Determine the coefficients of a linear combination of noises and comapre with lscov

Contents

- [Make data](#)
- [Do fit and check results](#)

Make data

```
fs    = 10;
nsecs = 10;

% Elements of the fit basis
B1 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B1.setName;
B2 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B2.setName;
B3 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B3.setName;

% random additive noise
n  = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'm'));

% coefficients of the linear combination
c1 = ao(1,plist('yunits','m/T'));
c1.setName;

c2 = ao(2,plist('yunits','m/T'));
c2.setName;

c3 = ao(3,plist('yunits','m T^-1'));
c3.setName;

% build output of linear system
y = c1*B1 + c2*B2 + c3*B3 + n;
y.simplifyYunits;
```

Do fit and check results

```
% Get a fit with linlsqsvd
pobj1 = linlsqsvd(B1, B2, B3, y)
```

```
     ---- pest 1 ----
           name: a1*B1+a2*B2+a3*B3
    param names: {'a1', 'a2', 'a3'}
              y: [0.81162366736073077;1.8907151217948008;3.0098623857384701]
             dy: [0.091943725803872112;0.089863977231447567;0.097910574305897308]
         yunits: [m T^(-1)][m T^(-1)][m T^(-1)]
            pdf: []
            cov: [3x3], ([0.00845364871469762 0.000268768332741779 0.000180072770333592;0.000268768332741779 0.00807553440385413
0.00125972375325089;0.000180072770333592 0.00125972375325089 0.00958648056091064])
           corr: [3x3], ([1 0.0325289738130578 0.020003055941376;0.0325289738130578 1 0.143172656986983;0.020003055941376 0.143172656986983 1])
          chain: []
           chi2: 0.87276552675043451
            dof: 97
         models: B1/tsdata Ndata=[100x1], fs=10, nsecs=10, t0=1970-01-01 00:00:00.000 UTC, B2/tsdata Ndata=[100x1], fs=10, nsecs=10, t0=1970-01-01
00:00:00.000 UTC, B3/tsdata Ndata=[100x1], fs=10, nsecs=10, t0=1970-01-01 00:00:00.000 UTC
    description:
           UUID: b8628843-a1e8-4815-b69b-90efdadc16c2
    ---------------
```

```
    % do linear combination: using eval
    yfit = pobj1.eval(B1, B2, B3);

    % Plot - compare data with fit result
    iplot(y, yfit)
```

©LTP Team
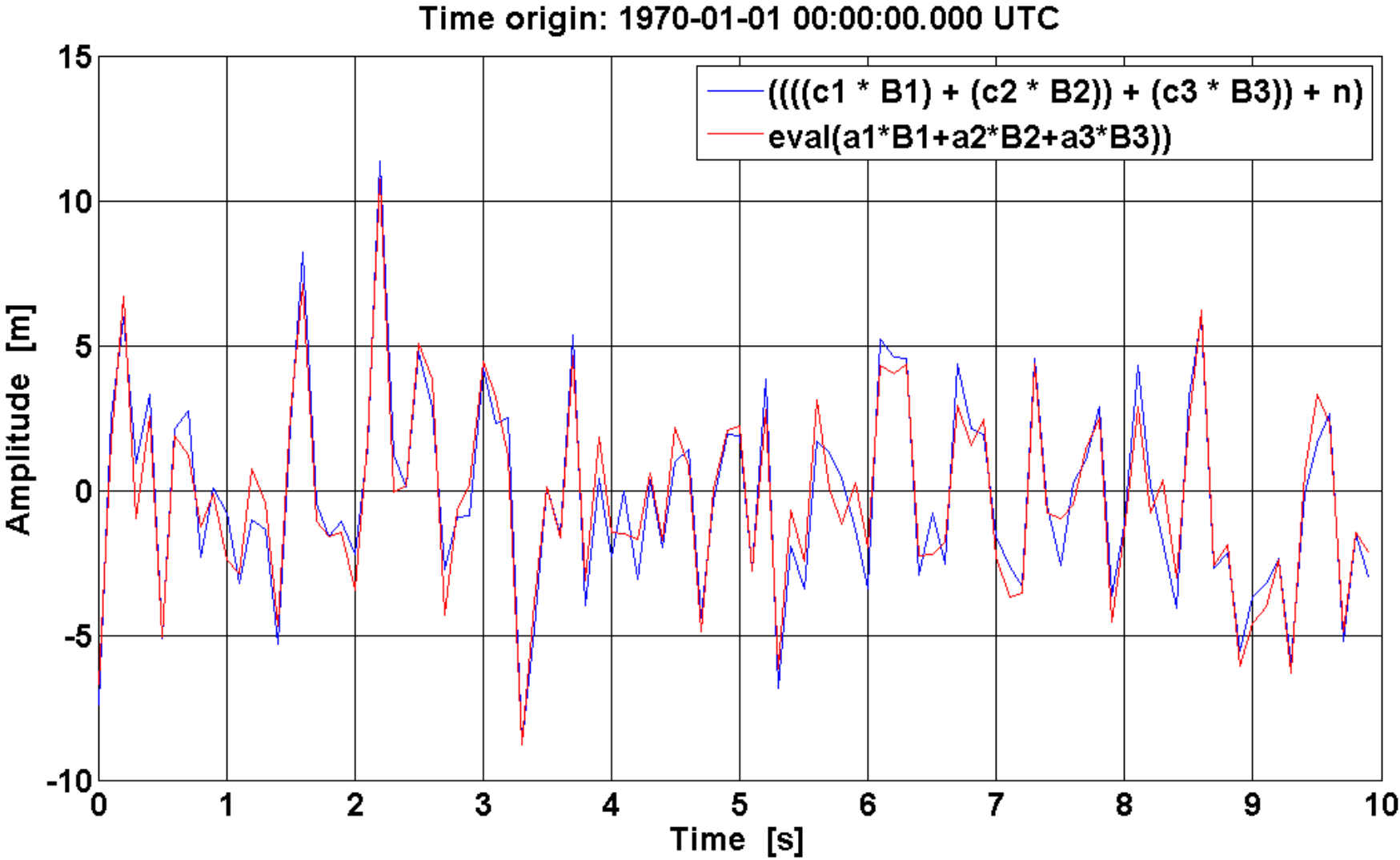
Linear least squares with singular value deconposition – multiple experiments

Determine the coefficients of a linear combination of noises

Contents

- [Make data](#)
- [Do fit](#)

Make data

```
fs    = 10;
nsecs = 10;

% fit basis for 2 experiments case
B1 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B1.setName;
B2 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B2.setName;
B3 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B3.setName;
B4 = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'T'));
B4.setName;

C1 = matrix(B1,B2,plist('shape',[2,1]));
C1.setName;
C2 = matrix(B3,B4,plist('shape',[2,1]));
C2.setName;

% make additive noise
n1  = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'm'));
n1.setName;
n2  = ao(plist('tsfcn', 'randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'm'));
n2.setName;

% coefficients of the linear combination
a1 = ao(1,plist('yunits','m/T'));
a1.setName;
a2 = ao(2,plist('yunits','m/T'));
a2.setName;

% assign output values
% y is a matrix containing the outputs of two experiments:
y1 = a1*B1 + a2*B3 + n1;
y2 = a1*B2 + a2*B4 + n2;
y = matrix(y1,y2,plist('shape',[2,1]));
```

Do fit

```
% Get a fit with linlsqsvd
pobj = linlsqsvd(C1, C2, y)
```

```
---- pest 1 ----
       name: a1*C1+a2*C2
param names: {'a1', 'a2'}
          y: [0.97312642877028477;2.0892132651873916]
         dy: [0.06611444020240001;0.065007088662104057]
      yunits: [T^(-1) m][T^(-1) m]
        pdf: []
        cov: [2x2], ([0.00437111920327673 -0.000390118937121542;-0.000390118937121542 0.00422592157632266])
       corr: []
      chain: []
       chi2: 0.85210029717685576
        dof: 198
     models: matrix(B1/tsdata Ndata=[100x1], fs=10, nsecs=10, t0=1970-01-01 00:00:00.000 UTC, B2/tsdata Ndata=[100x1], fs=10, nsecs=10, t0=1970-
01-01 00:00:00.000 UTC), matrix(B3/tsdata Ndata=[100x1], fs=10, nsecs=10, t0=1970-01-01 00:00:00.000 UTC, B4/tsdata Ndata=[100x1], fs=10, nsecs=10,
t0=1970-01-01 00:00:00.000 UTC)
  description:
       UUID: 545c9699-e749-40d5-bbe1-1322599c9c5d
  ---------------
```

```
% do linear combination: using eval
yfit = pobj.eval;

% extract objects
yfit1 = getObjectAtIndex(yfit,1);
yfit2 = getObjectAtIndex(yfit,2);

% Plot - compare data with fit
iplot(y1, yfit1)
iplot(y2, yfit2)
```
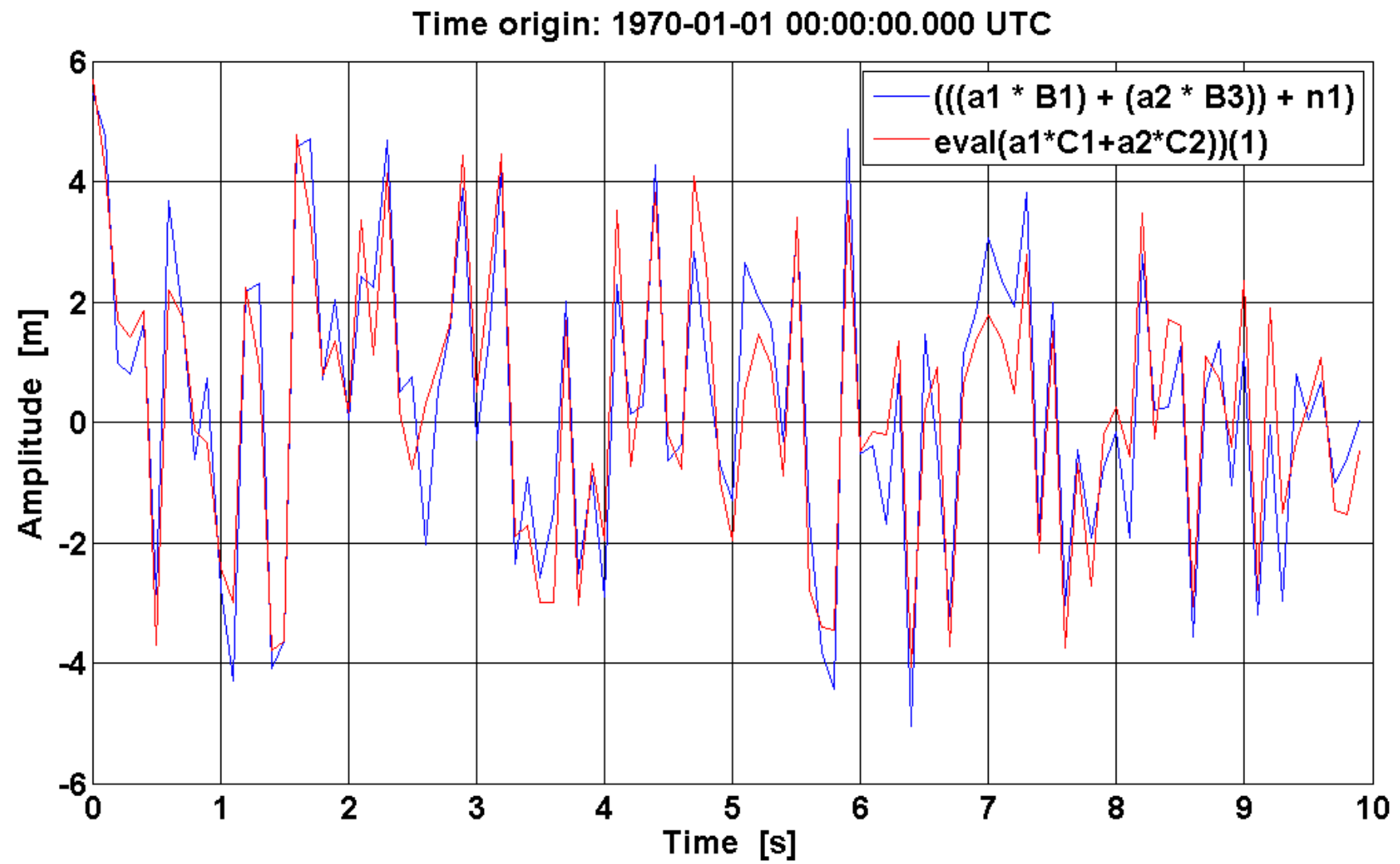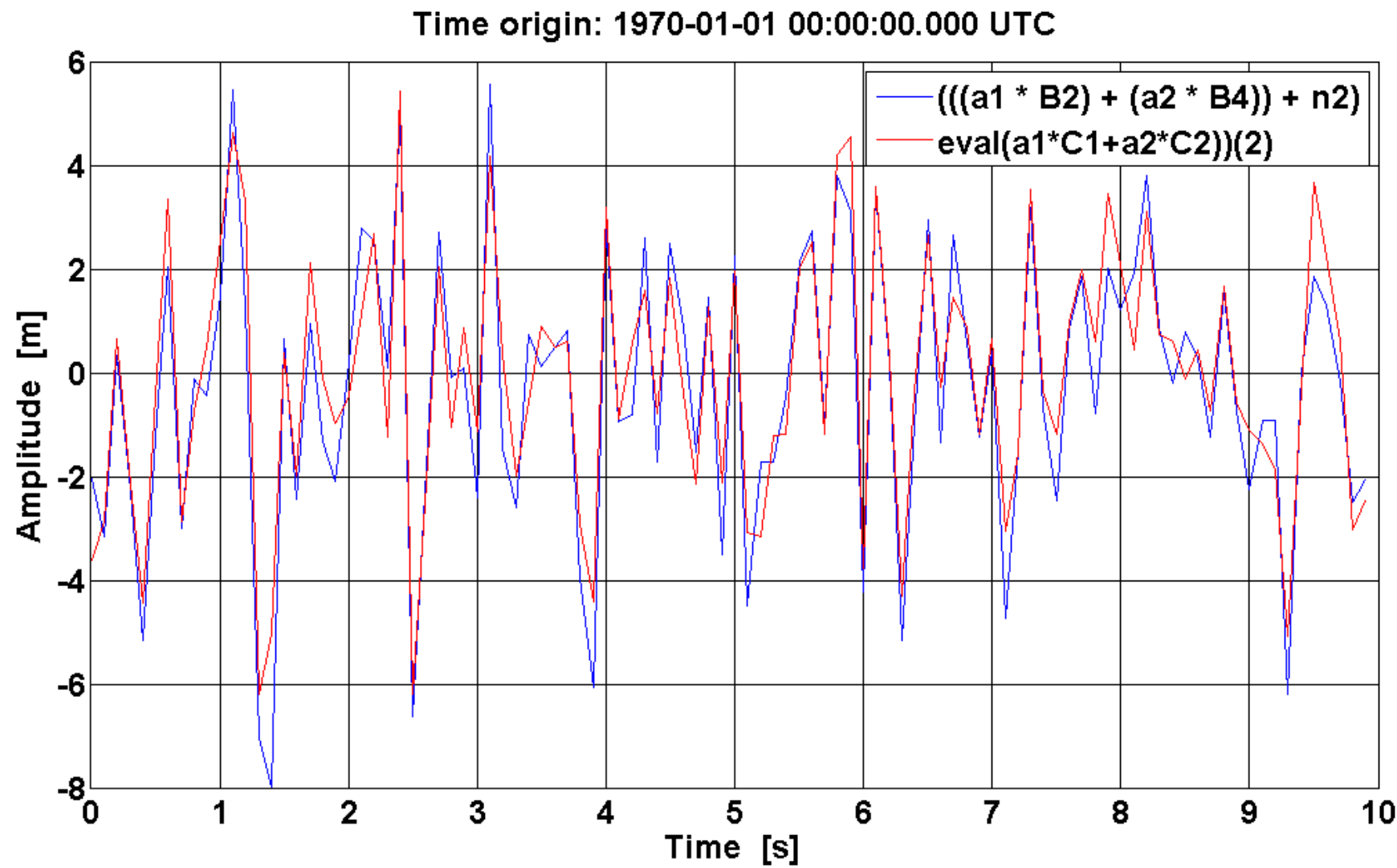
©LTP Team

Iterative linear parameter estimation for multichannel systems – symbolic system model in frequency domain

Iterative linear parameter estimation for multichannel systems – symbolic system model in frequency domain.

Contents

set plist for retriving

```
pl = plist('hostname', 'lpsdas01.esac.esa.int', 'database', 'ex6');
```

retrive data

```
 o1_1 = ao(pl.pset('binary', 'yes', 'id', 169));
o12_1 = ao(pl.pset('binary', 'yes', 'id', 170));

o1_2 = ao(pl.pset('binary', 'yes', 'id', 171));
o12_2 = ao(pl.pset('binary', 'yes', 'id', 172));
```

Load input signal

```
 is1 = matrix(pl.pset('binary', 'yes', 'id', 173));
is2 = matrix(pl.pset('binary', 'yes', 'id', 180));
```

load Whitening filters

% Stoc filter

```
 fil1 = filterbank(pl.pset('binary', 'yes', 'id', 191));
fil2 = filterbank(pl.pset('binary', 'yes', 'id', 192));
fil3 = filterbank(miir());
% build matrix
wf = matrix(fil1,fil3,fil3,fil2,plist('shape',[2 2]));
```

Build input objects

```matlab
  % empty ao
eao = ao();

% exp_3_1
os1 = matrix(o1_1,o12_1,plist('shape',[2 1]));

%  exp_3_2
os2 = matrix(o1_2,o12_2,plist('shape',[2 1]));

% Input signals
iS = collection(is1,is2);

% Fit Params
usedparams = {'A1','A2','S21','w1','w12','del1','del2'};

nsecs = os1.objs(1).data.nsecs;
fs = os1.objs(1).data.fs;
npad = nsecs*fs;

% set bounded params
bdparams = {'del1','del2'};
bdvals   = {[0.1 0.3],[0.1 0.3]};
```

system model 1

```matlab
  H = matrix(plist('built-in','ifo2ifo', 'Version', 'LSS v4.9.2 Phys Params'));
```

Do Fit

```matlab
  plfit = plist(...
  'FitParams',usedparams,...
  'Model',H,...
  'Input',iS,...
  'WhiteningFilter',wf,...
  'tol',1,...
  'Nloops',10,...
  'Npad',npad,...
  'Ncut',1e4);

opars1 = linfitsvd(os1,os2,plfit);
```

system model 2

```matlab
  H2 = matrix(plist('built-in','ifo2ifo', 'Version', 'LSS v4.9.2 Phys Params Alias'));
```

Set Model Alias

```matlab
  plalias = plist('nsecs',nsecs,'npad',npad,'fs',fs);
for ii=1:numel(H2.objs)
  H2.objs(ii).assignalias(H2.objs(ii),plalias);
end
```

Do fit with alias

```matlab
plfit2 = plist(...
  'FitParams',usedparams,...
  'Model',H2,...
  'BoundedParams',bdparams,...
  'BoundVals',bdvals,...
  'Input',iS,...
  'WhiteningFilter',wf,...
  'tol',1,...
  'Nloops',10,... % maximum number of fit iterations
  'Npad',npad,...
```

```
    'Ncut',1e4); % number of data points to skip at the starting of the series to avoid
whitening filter transient

opars2 = linfitsvd(os1,os2,plfit2);
```

©LTP Team

Iterative linear parameter estimation for multichannel systems – ssm system model in time domain

Iterative linear parameter estimation for multichannel systems – ssm system model in time domain.

Contents

set plist for retriving

```
pl = plist('hostname', 'lpsdas01.esac.esa.int', 'database', 'ex6');
```

retrive data

```
o1_1 = ao(pl.pset('binary', 'yes', 'id', 68));
o12_1 = ao(pl.pset('binary', 'yes', 'id', 69));

o1_2 = ao(pl.pset('binary', 'yes', 'id', 74));
o12_2 = ao(pl.pset('binary', 'yes', 'id', 75));
```

Load input signal

```
 is1 = matrix(pl.pset('binary', 'yes', 'id', 78));
is2 = matrix(pl.pset('binary', 'yes', 'id', 79));

is1ao = is1.getObjectAtIndex(1);
is2ao = is2.getObjectAtIndex(2);

% set port names for fit
is1names = {'INPUTGUIDANCE.ifo_x1'};
is2names = {'INPUTGUIDANCE.ifo_x12'};

InputNames = {is1names,is2names};
```

load Whitening filters

```
  % Stoc filter
fil1 = filterbank(pl.pset('binary', 'yes', 'id', 76));
fil2 = filterbank(pl.pset('binary', 'yes', 'id', 77));
fil3 = filterbank(miir());
% build matrix
```

```
wf = matrix(fil1,fil3,fil3,fil2,plist('shape',[2 2]));
```

System Model

```
    H = ssm(plist('built-in','LTP',...
    'Version','Fitting',...
    'Continuous',true,...
    'dim',1,...
    'SYMBOLIC PARAMS',...

{'FEEPS_XX','CAPACT_TM2_XX','IFO_X12X1','EOM_TM1_STIFF_XX','EOM_TM2_STIFF_XX','DELAY_X1','DELAY_
}));
```

Build input objects

```
    % empty ao
```

```
    %%% exp_3_1 %%%
    os1 = matrix(o1_1,o12_1,plist('shape',[2 1]));

    %%% exp_3_2 %%%
    os2 = matrix(o1_2,o12_2,plist('shape',[2 1]));

    %%% output names
    OutputNames = {{'IFO.x1','IFO.x12'},{'IFO.x1','IFO.x12'}};

    %%% Input signals
    iS = collection(is1ao,is2ao);

    %%% Fit Params %%%
    usedparams =
{'FEEPS_XX','CAPACT_TM2_XX','IFO_X12X1','EOM_TM1_STIFF_XX','EOM_TM2_STIFF_XX','DELAY_X1','DELAY_
};

    %%% set bounded params
    bdparams = {'DELAY_X1','DELAY_X12'};
    bdvals   = {[0.1 0.3],[0.1 0.3]};

    %%% set numerical derivative step
    diffStep = [0.01,0.01,1e-7,1e-7,1e-7,0.001,0.001];
```

Do Fit

```
    plfit = plist(...
    'FitParams',usedparams,...
    'BoundedParams',bdparams,...
    'BoundVals',bdvals,...
    'diffStep',diffStep,...
    'Model',H,...
    'Input',iS,...
    'InputNames',InputNames,...
    'OutputNames',OutputNames,...
    'WhiteningFilter',wf,...
    'tol',1,...
    'Nloops',5,...
    'Ncut',1e5);

    opars = linfitsvd(os1,os2,plfit);
```

Z-Domain Fit

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

Z-domain system identification in LTPDA

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

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

Fit algorithm

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

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

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

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

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

Check that the variation of the root mean squared error is lower than `10^(-1*value)`.

3. Residuals spectral flatness and root mean squared error
   - Residuals Spectral Flatness
     In case of a fit on noisy data, the residuals from a good fit are expected to be as much as possible similar to a white noise. This property can be used to test the accuracy of a fit procedure. In particular it can be tested that the spectral flatness coefficient of the residuals is larger than a certain qiantity sf such that `0 < sf < 1`.
   - Root Mean Squared Error
     Check that the variation of the root mean squared error is lower than `10^(-1*value)`.

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

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

Usage example of `z-domain` system identification tool

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

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

    a = ao(pl);
    a.setName;

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

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

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

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

The `filterbank` object contains a parallel bank of 18 filters.

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

Each object of the `filterbank` is a `miir` filter.

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

References

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

©LTP Team

S-Domain Fit

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

S-domain system identification in LTPDA

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

Fit Algorithm

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

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

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

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

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

- o Root Mean Squared Error Check that the variation of the root mean squared error is lower than `10^(-1*value)`.

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

Usage example of `s-domain` system identification tool

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

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

a = ao(pl);
a.setName;

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

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

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

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

```
>> mod.objs
---- parfrac 1 ----
model:       fit(a)
res:         [1.69531090137847e-006;-1.69531095674486e-006;1.39082537801437e-007;-
1.39094453401266e-007;3.9451875151135e-007;-3.94524993613367e-007;4.53671387948961e-007;-
4.53664974359603e-007;1124.81020427899;0.000140057852149302-
i*0.201412268649905;0.000140057852149302+i*0.201412268649905]
poles:       [-1.18514026248382e-006;1.18514354570495e-006;-
0.0045731158205039;0.0045734088943545;-0.0316764149343339;0.0316791653277322;-
0.276256442292693;0.27627799022013;330754.550617933;-0.0199840558095427+i*118.439896186467;-
0.0199840558095427-i*118.439896186467]
dir:         0
pmul:        [1;1;1;1;1;1;1;1;1;1;1]
iunits:      []
ounits:      []
description:
UUID:        2afc4c82-7c2a-4fe3-8910-d8590884d58c
------------------
```

References

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

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

◀ Z–Domain Fit   Graphical User Interfaces in LTPDA ▶

Graphical User Interfaces in LTPDA

LTPDA has a variety of Graphical User Interfaces:

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

©LTP Team

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.

◀ Graphical User Interfaces in LTPDA   The LTPDA Workbench ▶

©LTP Team

The LTPDA Workbench

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

The following sections describe the use of the LTPDA Workbench.

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

Loading the LTPDA Workbench

Overview

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

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

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

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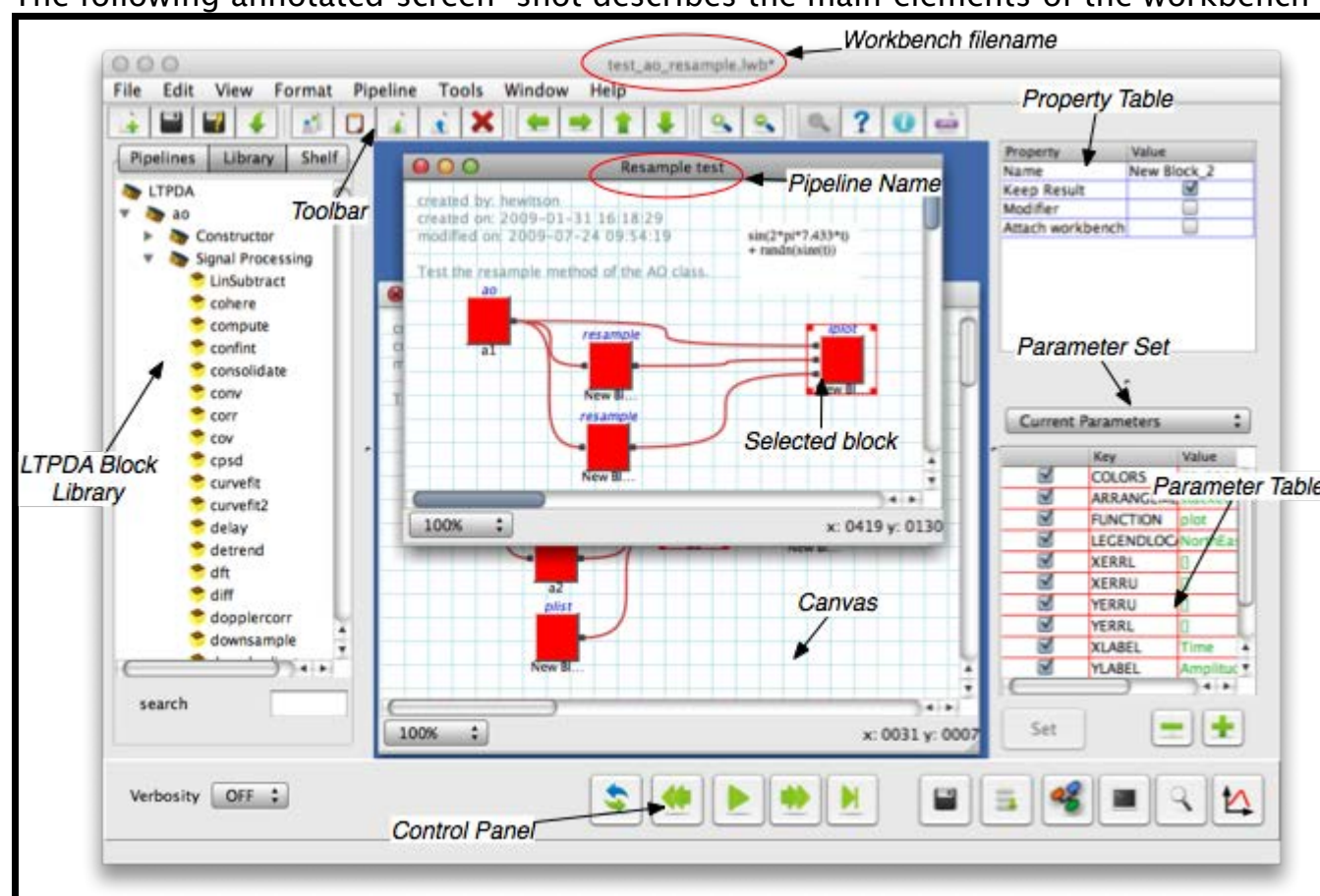
Starting the Workbench

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

```
>> LTPDAworkbench
```

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

```
>> wb = LTPDAworkbench
```

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

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

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

◀ The LTPDA Workbench   Mouse and keyboard actions ▶

©LTP Team

Mouse and keyboard actions

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

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

Keyboard actions on the main workbench

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

Keyboard actions on the Canvas

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

`escape`	same	De-select all blocks.
`delete`	same	Delete selected blocks (or pipes).

Mouse actions on the Canvas

Action (on Windows/Linux)	Action (on Mac OS X)	Description
Mouse-wheel scroll	same	Zoom in and out on the canvas
drag with left-mouse-button	same	Draw rubber-band box to select elements
alt-left-mouse-button	same	Move the canvas around
right-click	right-click (or ctrl-left-click)	Bring up canvas context menu
left-click on canvas	same	De-select all selected blocks or pipes
left-click on a block	same	Select the block and bring up its property and parameter tables
shift-left-click on a block	same	Add the block to the selected blocks
left mouse button down on a block	same	Move this and all other selected blocks
click-drag on selected block handles	same	Resize the block
ctrl-left-click on block	cmd-left-click on block	If a single block is selected before this action then the result of this action is to connect the two blocks.
mouse-drag on a port	mouse-drag on a port	Start drawing a pipe originating from the port.
release left mouse button on a port	same	If a pipe was being dragged, then the source port and destination port are connected by a pipe.
release left mouse button on a block	same	If a pipe was being dragged, then the source port is connected to the first free input of the destination block.

Mouse and keyboard actions (LTPDA Toolbox)

©LTP Team

The canvas

Composing LTPDA pipelines is done on a "Canvas". Each top-level pipeline, and each subsystem, is represented on a canvas. A subsystem block is also a view of pipeline.

Canvas properties

You can set properties of a canvas via the canvas inspector. To open the inspector, hit `ctrl-i` (`cmd-i` on Mac OS X) on the canvas.

Canvas Info

author `Mr Fish`

description

```
My lovely pipeline for processing my
hard earned data.
```

created on 2009-02-02 10:45:18

last modified on 2009-02-02 10:58:20

Cancel   OK
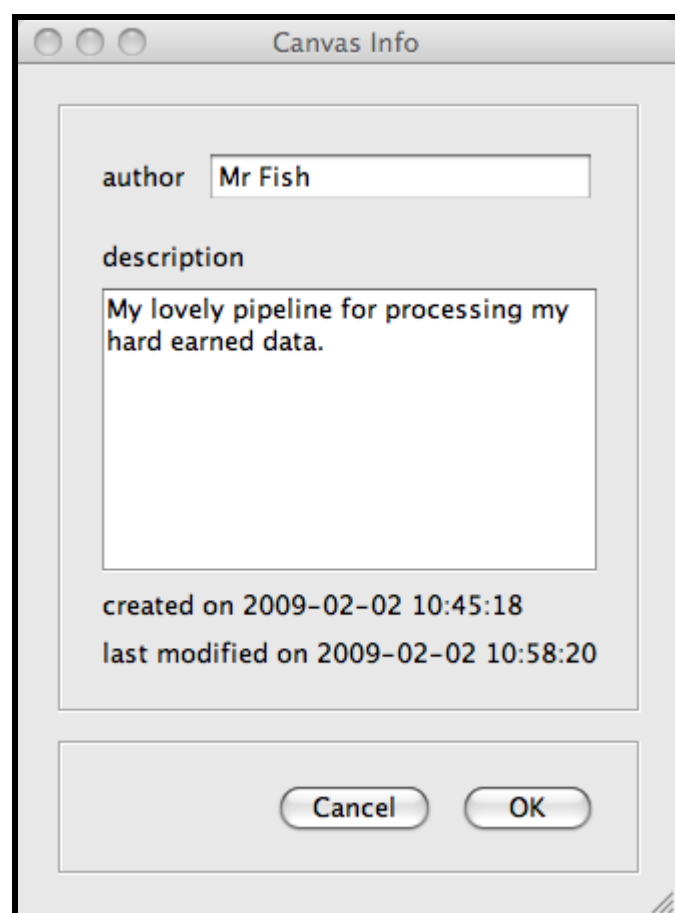
With the canvas inspector you can edit

- 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.

Element	Algorithm	Canvas
Unknown_2	cos	New Document 2
Unknown_3	sin	New Document 2
Unknown_8	times	New Document 2
Unknown_10	times	New Document 2
Unknown_9	plus	New Document 2

Search in

○ Selected Pipeline
◉ All Pipelines

Search for

Unknown

Cancel   OK

◀ Mouse and keyboard actions   Building pipelines by hand ▶

©LTP Team
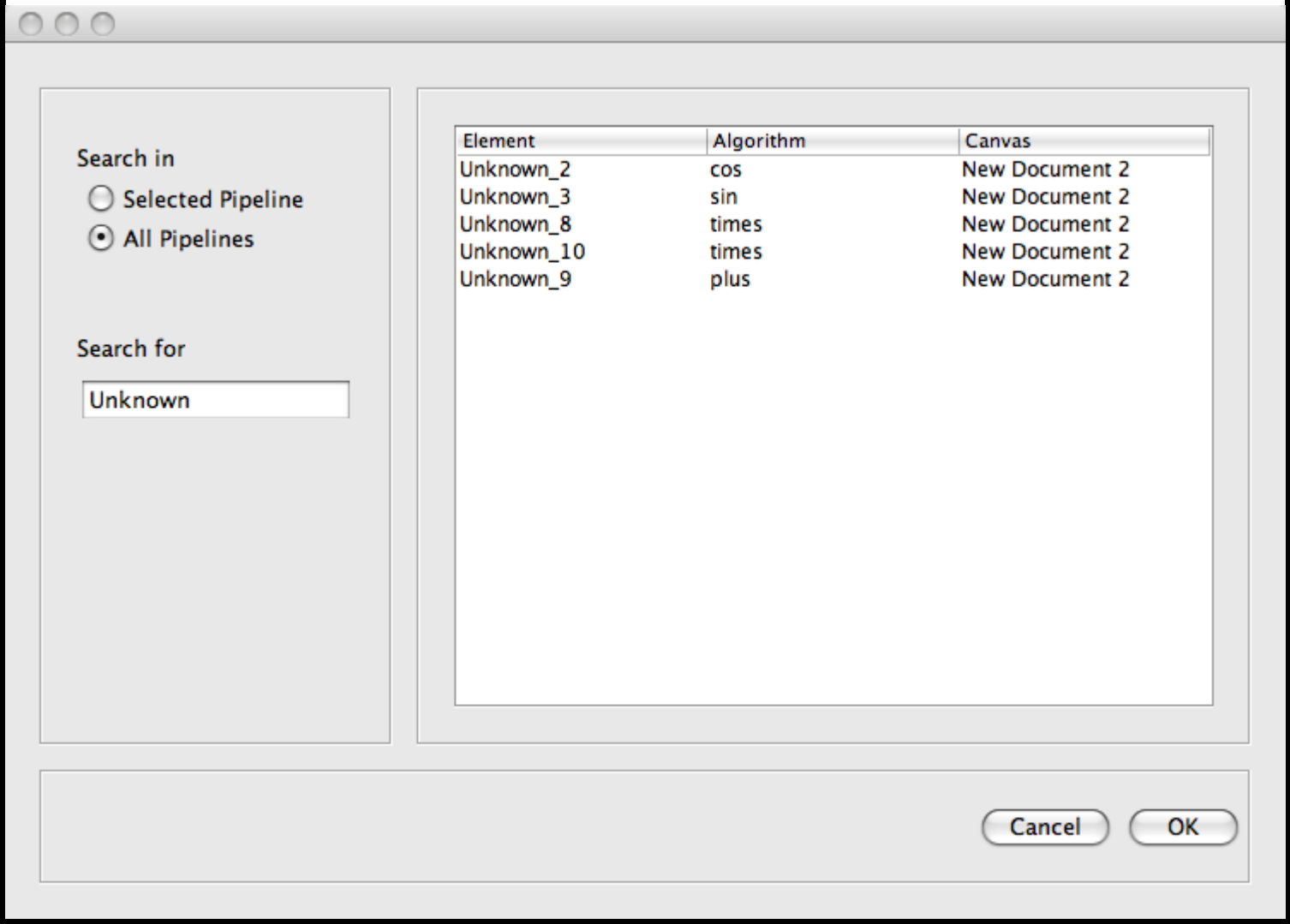
Building pipelines by hand

This section describes how to compose pipelines by hand.

- [Block types](#)
- [Adding blocks to the Canvas](#)
- [Setting block properties and parameters](#)
- [Connecting blocks](#)
- [Creating subsystems](#)

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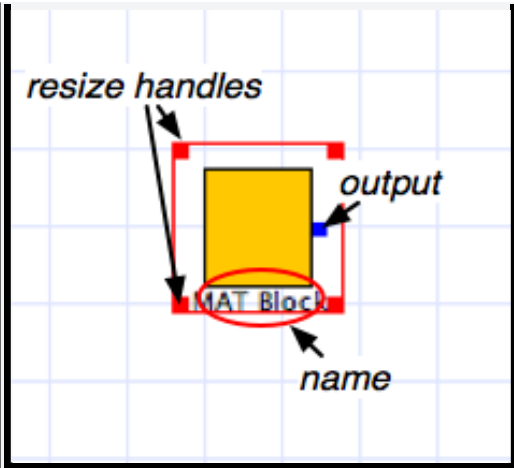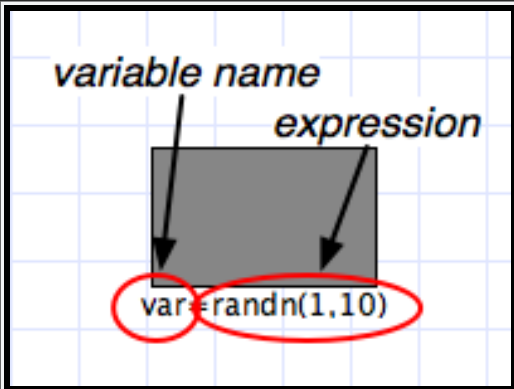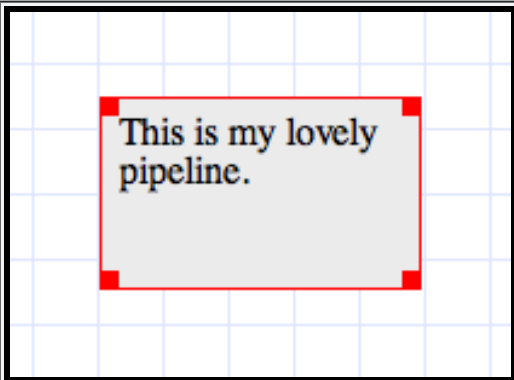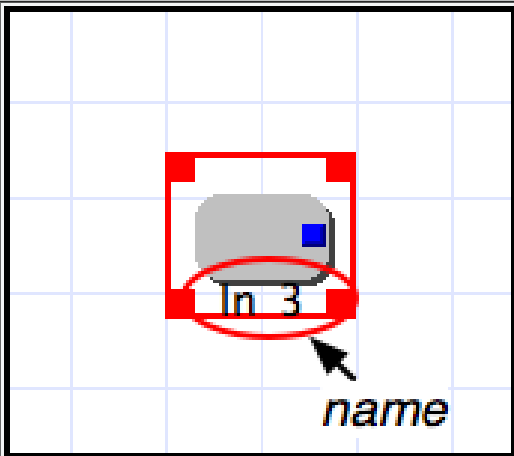
Block types
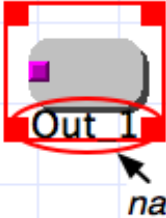
Various types of elements can be present on a pipeline. Blocks can have different states: idle, ready, or executed. These are color coded as

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

resize handles → ▢ *output* MAT Block — *name*			passed to subsequent blocks.
variable name *expression* var=randn(1,10)	MATLAB Constant Block	MConstant	A block evaluates a MATLAB expression and stores the result in the MATLAB workspace with the given variable name.
This is my lovely pipeline.	Annotation Block	MAnnotation	A block containing editable text to allow for annotating pipelines.
In 3 *name*	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.

©LTP Team

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.
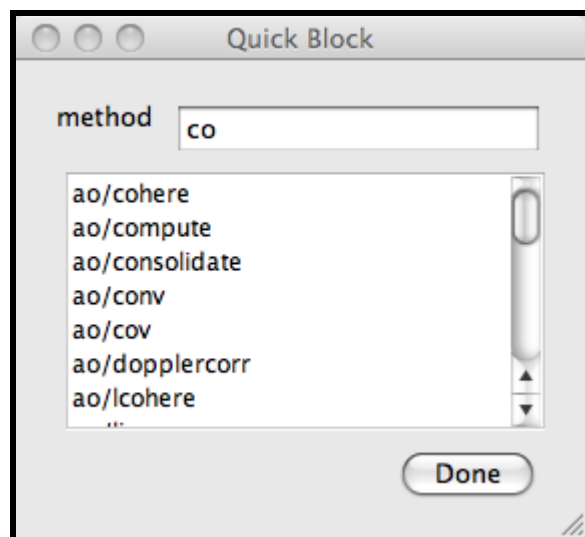
To get the block you want, just start typing in the "method" edit field. Once the block you want is top of the list, just hit `enter` to add it to the canvas. You can also double-click on the block list to add any of the blocks in the list.

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.

◀ Block types   Setting block properties and parameters ▶

Adding blocks to the canvas (LTPDA Toolbox)

©LTP Team

Setting block properties and parameters

The different block types have different properties that the user can set.

LTPDA Algorithm Blocks

LTPDA Algorithm blocks (MBlocks) have both *properties* and *parameters*.
Properties of an MBlock are

Property	Description
Name	The name of the block as it appears on the canvas Block names are unique on a canvas. This is also the string that will be converted to a valid MATLAB variable name when the pipeline is executed.
Modifier	Set this block to be a modifier or not. For more details on modifier blocks in LTPDA see [Calling object methods](#). The accepted values are "true" or "false".

To set the properties of a block, select one or more MBlocks, then double click in the value column entry for the property you want to change. Enter the new value and press return/enter.

Setting the parameter list

LTPDA Algorithm Blocks also have parameters which translate as a parameter list upon execution. To set the parameters of a block, click on a block (or multiple MBlocks which represent the same LTPDA algorithm). You will then see the 'current parameters' that the block holds. To edit the 'key' or 'value' of a parameter, double click the table entry you want to edit, enter the new value, and hit `enter` or click OK.

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

You can also select a set of predefined parameter sets from the drop-down menu above the parameter table. Having selected a parameter set, you need to click the 'set' button to push these parameters to the block. You can then go ahead and add or remove parameters from the 'current parameters' on the block.

Editing of most parameter keys and values is done in a simple editor dialog box. However, there are some key/value pairs which are edited using special dialog boxes:

Built-in models of AO and SSM classes

Both the AO and the SSM classes can be built from pre-defined, built-in models. These are typically created with a plist containing the key `BUILT-IN`. If you try to edit the value for this key for one of these constructors, you will be presented with a dialog box that allows you to choose from the built-in models. For all other classes, editing the value for the key `BUILT-IN` is done via a standard input dialog.
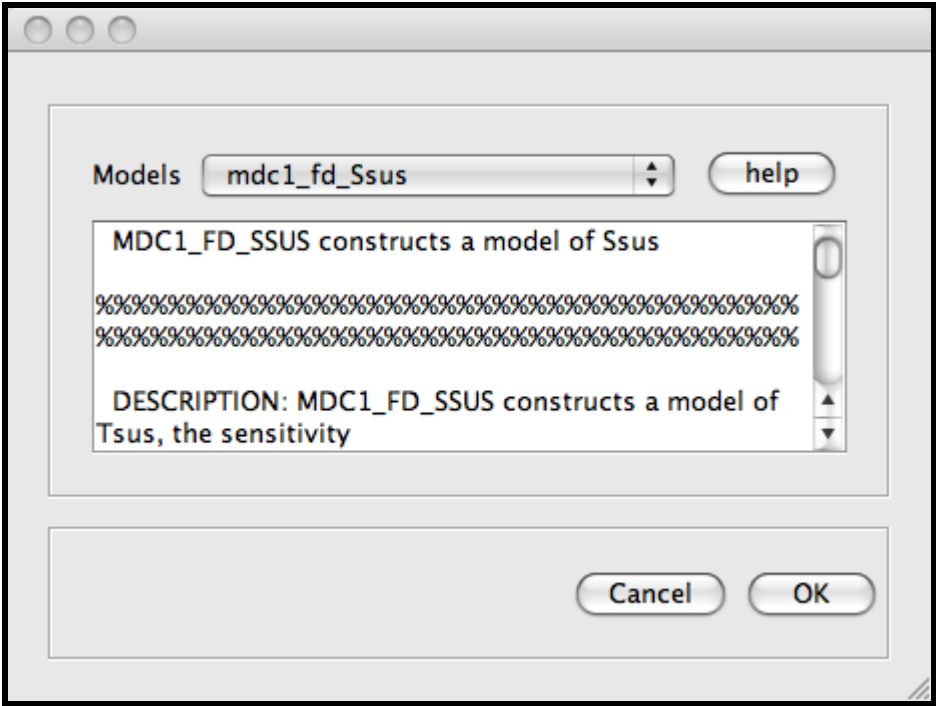
Pole/zero model editor

If any block has a parameter with the key PZMODEL then the corresponding value will be edited via the Pole/zero model editor. Here you can type directly in the constructor edit box, or you can add/remove poles and zeros from the lists. To edit the frequency or Q of a pole or zero, double-click on the table entry. To enter a real pole or zero (no Q), set the Q to 'NaN'.
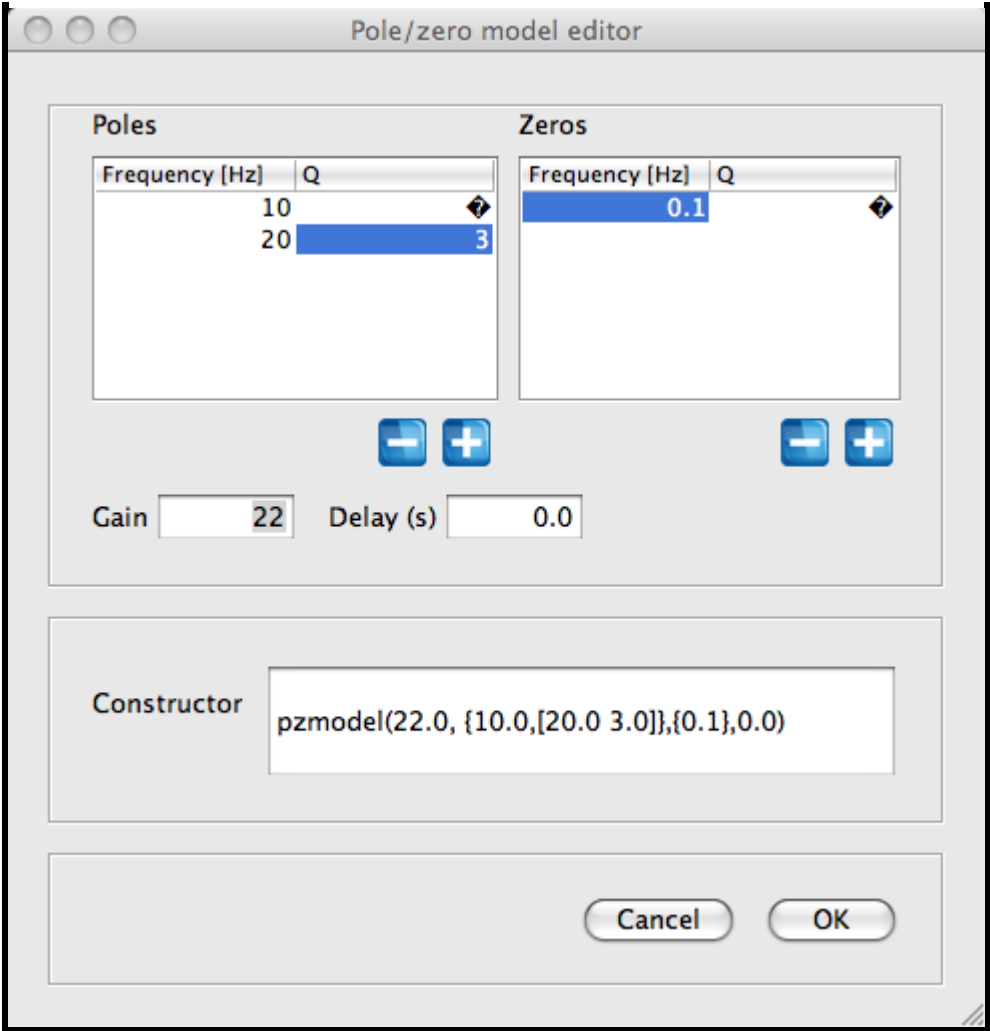
Spectral window selector

Many algorithms in LTPDA accept a parameter with the key WIN for a spectral window parameter. Editing the value for such a parameter presents the user with a dialog where the spectral window can be selected from the list of supported windows. You can also type the constructor directly in the edit box.
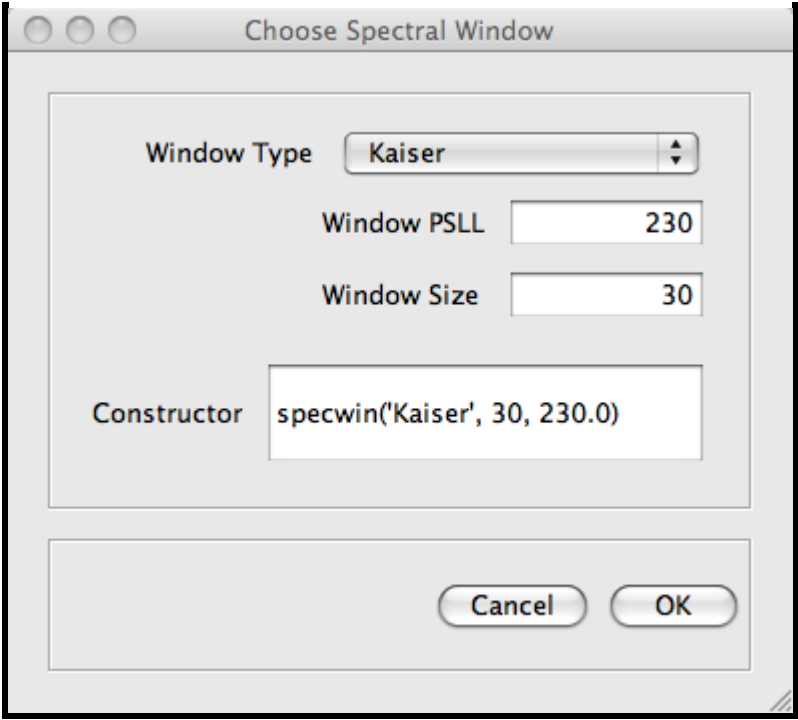
Repository hostname selector

Editing parameters with the key `hostname` will give the user a dialog containing the pop-up menu of possible hostnames. This list of hostnames is taken from the LTPDA Preferences. If the preferences are changed, the workbench needs to be closed and reopened for the changes to 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.

©LTP Team

Connecting blocks

Blocks of type "LTPDA Block" (MBlock), "Subsystem Block" (MSubsystem), "MATLAB Expression Block" (MATBlock), "Terminal Block" (MTerminal) all have input ports or output ports, or both.

These ports are connected together with "pipes" (MPipe is the underlying java class). Output ports can have more than one pipe connected; input ports can have only one pipe at a time. The binding object between a port and pipe is a "node" (MNode is the underlying java class). Nodes are displayed as small black circles.
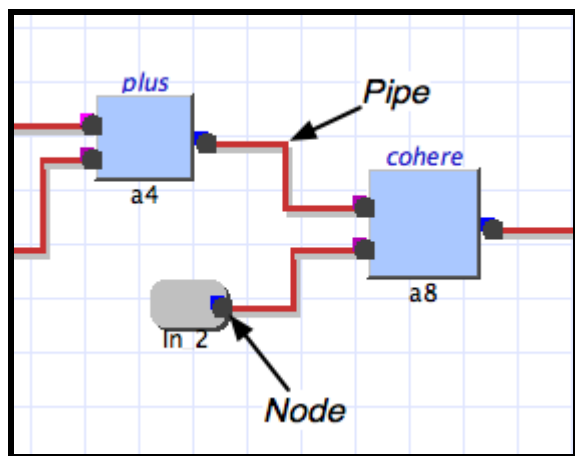
To connect these blocks together, do one of the following:

- Click and drag from one port to another.
- Click and drag from one output node to an input port.
- Click and drag from one output port to a block. Connection is made to the first free input (if there is one).
- Click and drag from one output node to a block. Connection is made to the first free input (if there is one).
- Select a source block, then ctrl-left-click a destination block to join the two. There must be at least one free input on the destination block. On the source block, the next free output is used, or the first output if no free outputs are available.

◀ Setting block properties and parameters   Creating subsystems ▶

©LTP Team

Connecting blocks (LTPDA Toolbox)

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.

©LTP Team

Using the Workbench Shelf

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

It is also possible to export shelf categories to disk and then import them again. This offers the possibility to exchange shelf categories containing pre-built subsystems.

- [Accessing the shelf](#)
- [Creating shelf categories](#)
- [Adding and using shelf subsystems](#)
- [Importing and exporting shelf categories](#)

©LTP Team

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.
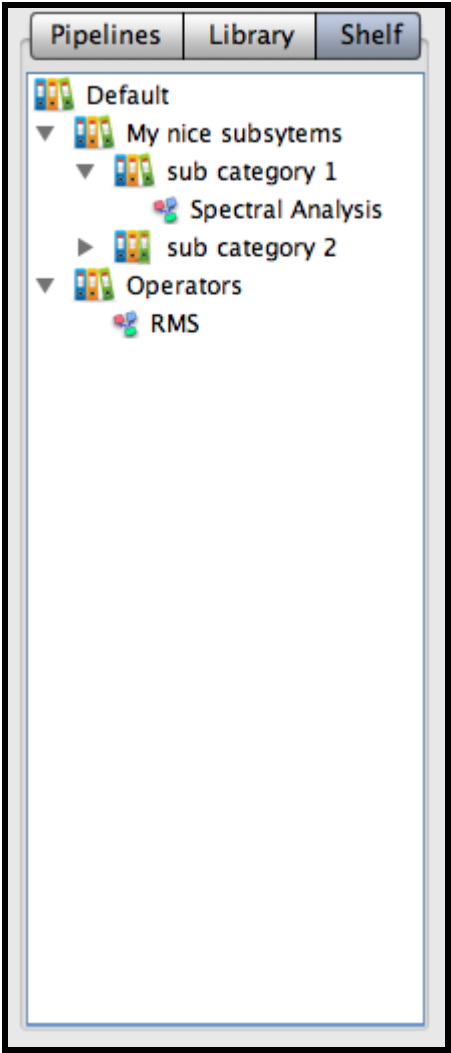
Here you can see the shelf has various sub-categories denoted by the book icons. In addition we see some subsystems in the sub-categories, denoted by the small pipeline icon.

◀ Using the Workbench Shelf   Creating shelf categories ▶

©LTP Team

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...'.
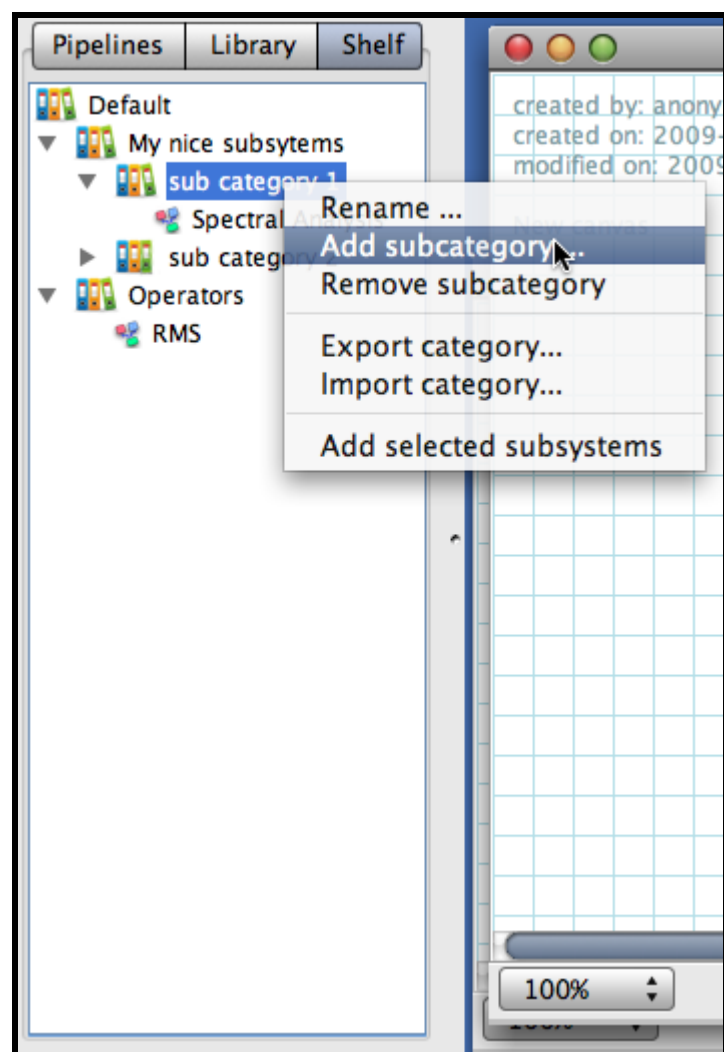
©LTP Team

Adding and using shelf subsystems

Adding subsystems to the shelf can be done in two ways:

1. via the context menu on the shelf
2. via the context menu on a subsystem

Adding subsystems via the shelf

To add one or more subsystems to the shelf from the current pipeline, do the following:

1. Select one or more subsystems on the current pipeline
2. Right-click on a category in the shelf and choose 'Add selected subsystems'

Adding subsystems via the subsystem context menu

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

1. Select a category in the shelf
2. Right-click on a subsystem on the current canvas and select 'Put on shelf'

Using subsystems from the shelf

Subsystems can be dragged from the shelf to the current pipeline. You can also right-click on a subsystem in the shelf and choose 'Add to pipeline'.

Hovering the mouse over a subsystem in the shelf reveals some information about the subsystem:
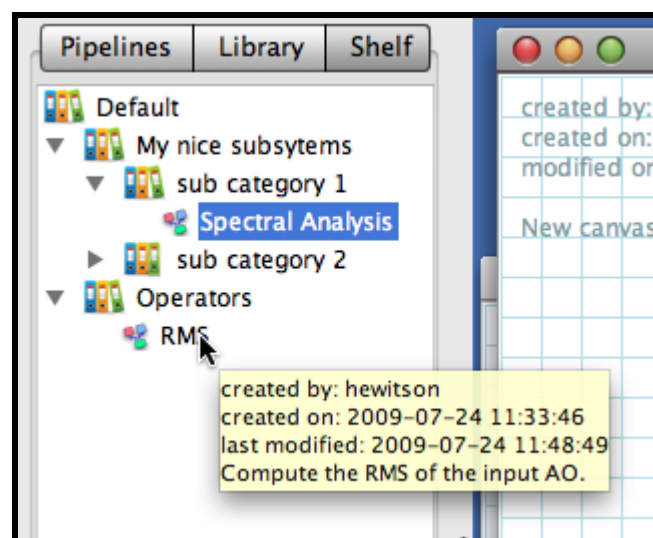
If you want to edit or rename a subsystem from the shelf, add it to a pipeline, then make the required edits, then re-add the edited subsystem to the shelf. When satisfied, you can remove the old subsystem from the shelf by right-clicking on the subsystem and choosing 'Remove from shelf'.

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...

©LTP Team

Execution plans (LTPDA Toolbox)

[contents](#) ◀ ▶

Editing the plan

Content needs written...

◀ Execution plans   Linking pipelines ▶

©LTP Team

Editing the plan (LTPDA Toolbox)

Linking pipelines

Content needs written...

LTPDA Toolbox™   [contents](#)   ← →

Executing pipelines

Content needs written...

← Building pipelines programatically   The LTPDA Workspace Browser →

©LTP Team

The LTPDA Workspace Browser

The LTPDA Workspace Browser allows you to explore the current MATLAB workspace from the point-of-view of LTPDA objects. Only LTPDA user-objects are shown in the list of current variables. In addition, structures containing LTPDA user-objects are also shown (such as those created by the LTPDA Workbench).

The figure below shows the workspace browser with some LTPDA user objects in the workspace. Hovering the mouse over an object reveals a tooltip containing details about the object.
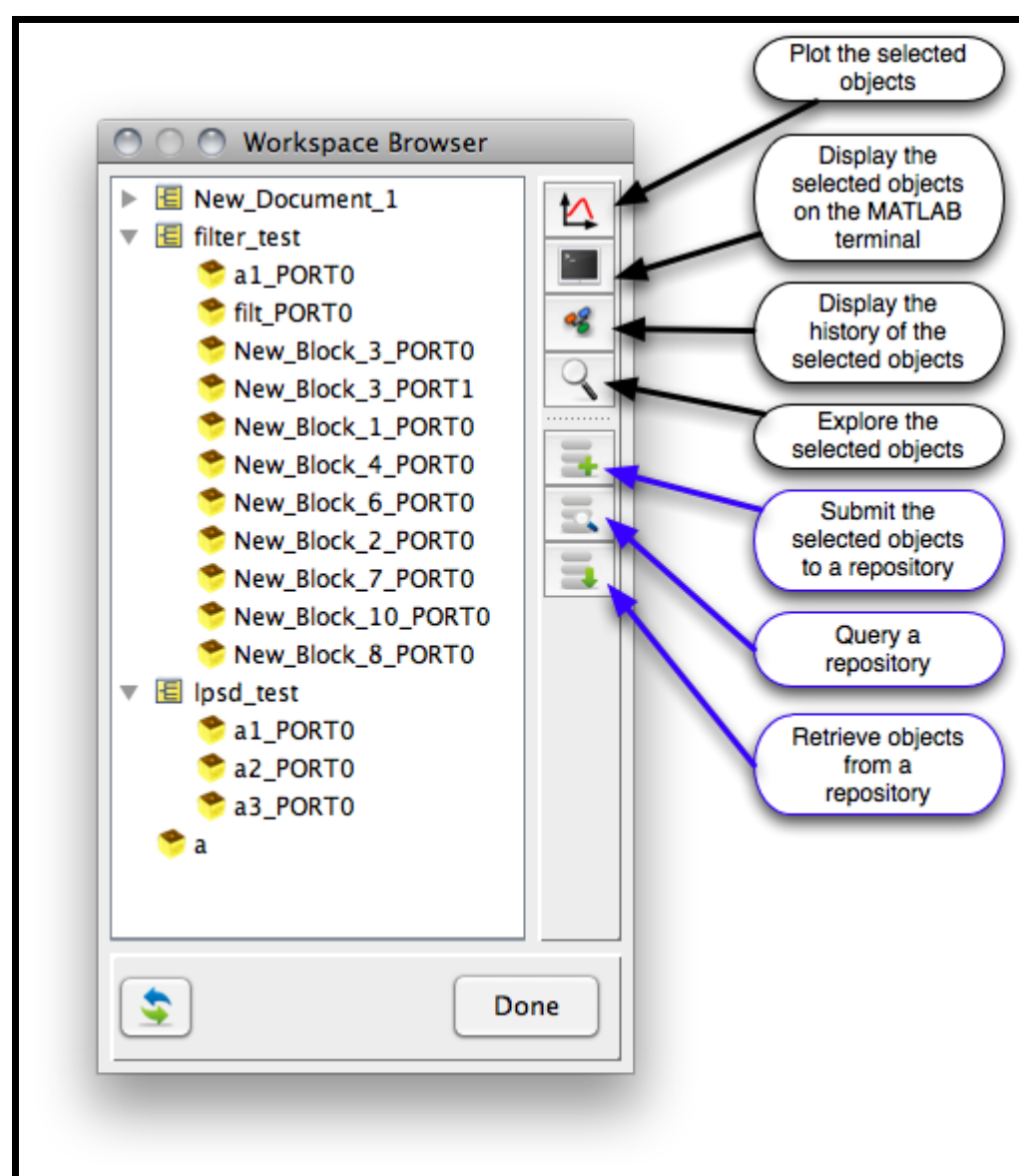
The button panel on the right of the workspace browser allows you to perform various actions on the selected objects, as well as providing a convenient way to interact with an LTPDA repository (see Working with an LTPDA Repository for more details).

◀ Executing pipelines   The pole/zero model helper ▶

©LTP Team

The pole/zero model helper

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

To start the pole/zero model helper:

```
    >> pzmodel_helper
```

or click the appropriate button on the LTPDA Launch Bay.

Once the GUI is loaded, you will see the following figure:
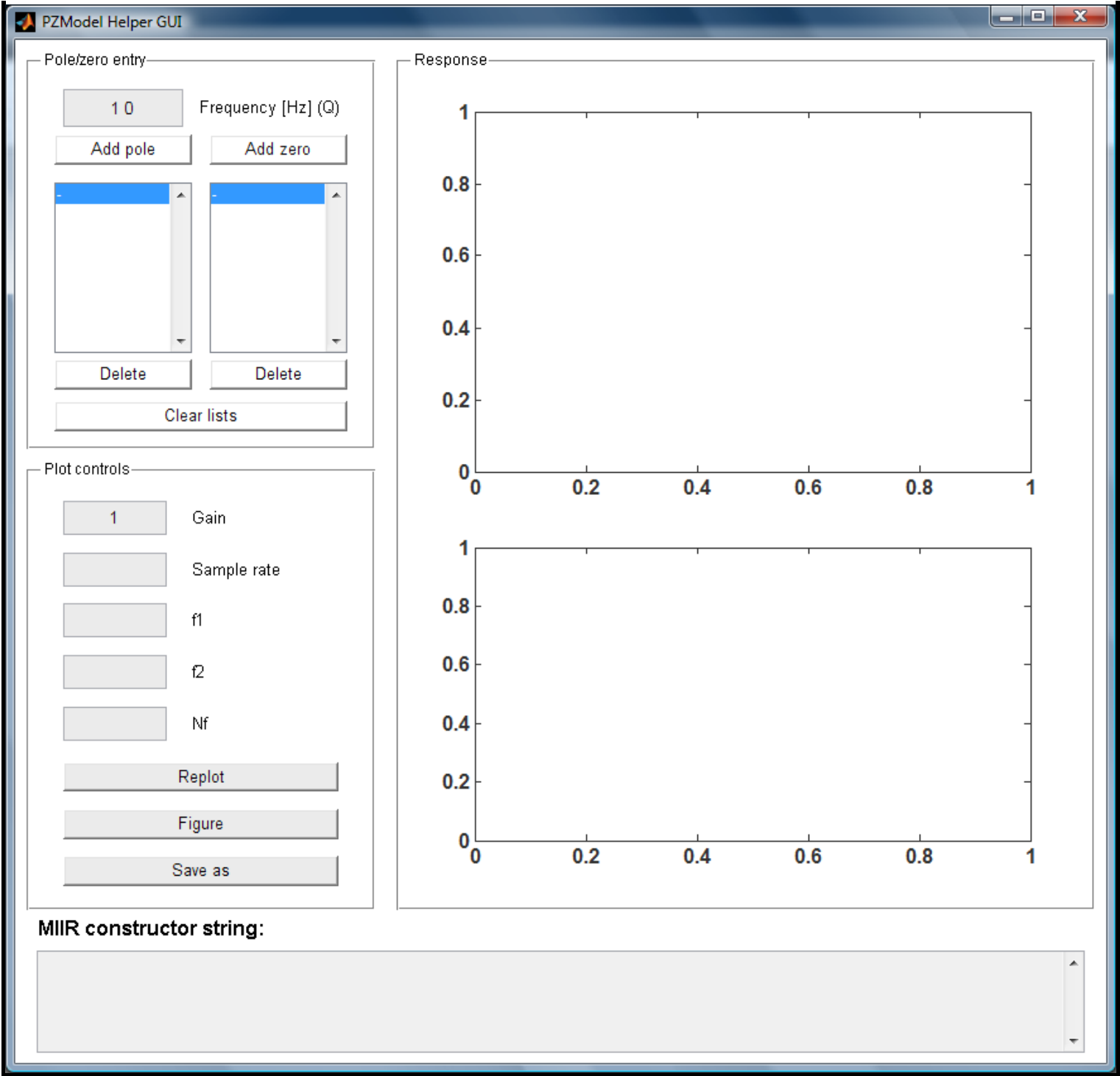
You can add poles and zeros to the model by entering the frequency (and Q) in the edit boxes, then click `Add pole` or `Add zero` as appropriate. The response is then updated in the response axes.

MIIR constructor string:

```
miir([pzmodel([plist('GAIN', [1], 'POLES', pz(1,0) , 'ZEROS', pz(10,NaN) , 'NAME', 'None', 'DESCRIPTION', '', 'IUNITS',
'', 'OUNITS', '')])])
```

◀ The LTPDA Workspace Browser   The Spectral Window GUI ▶

©LTP Team
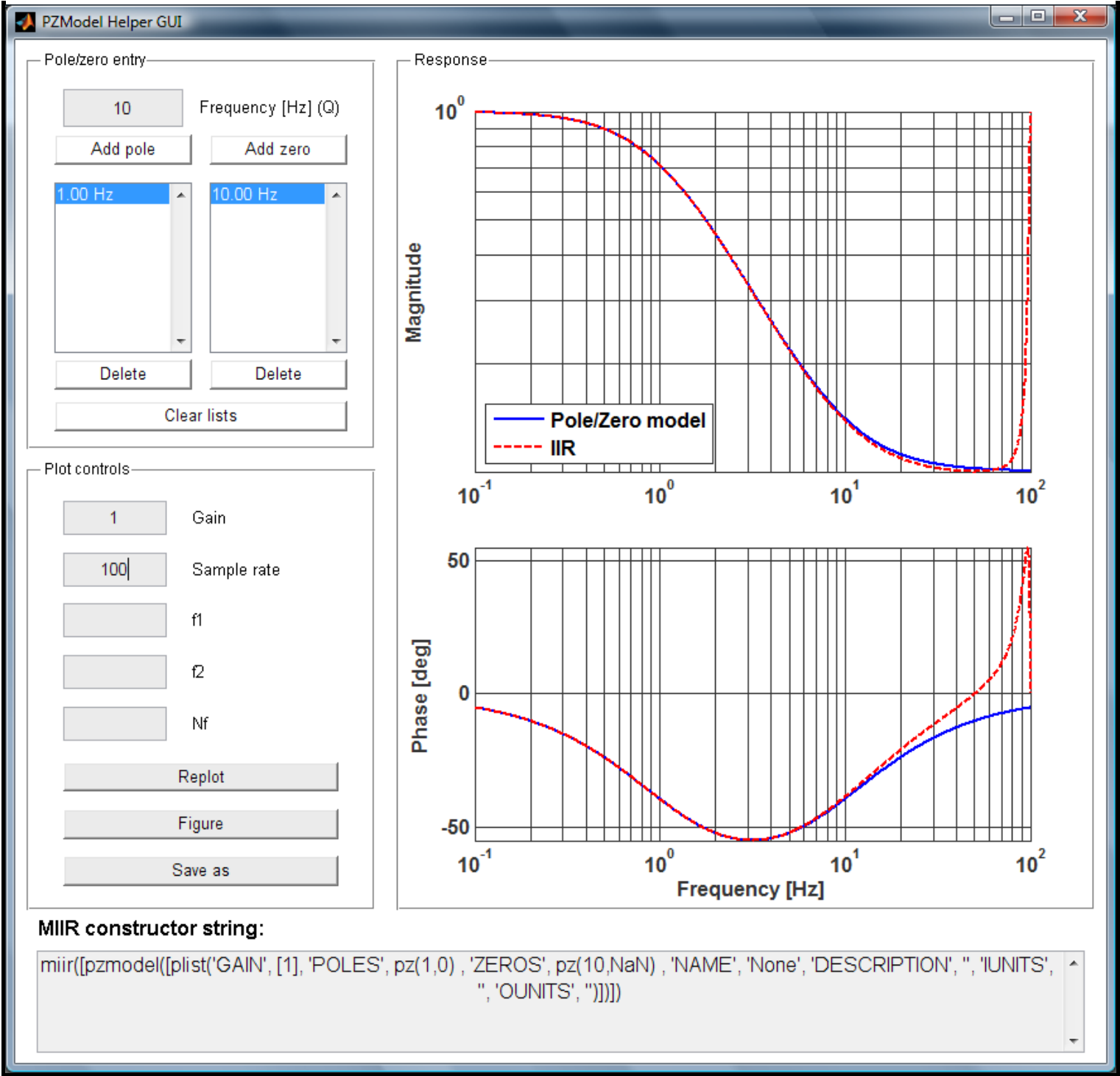
The Spectral Window GUI

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:

```
>> specwinViewer
```

or click the appropriate button on the Launch Bay.
You should then be presented with the following figure:
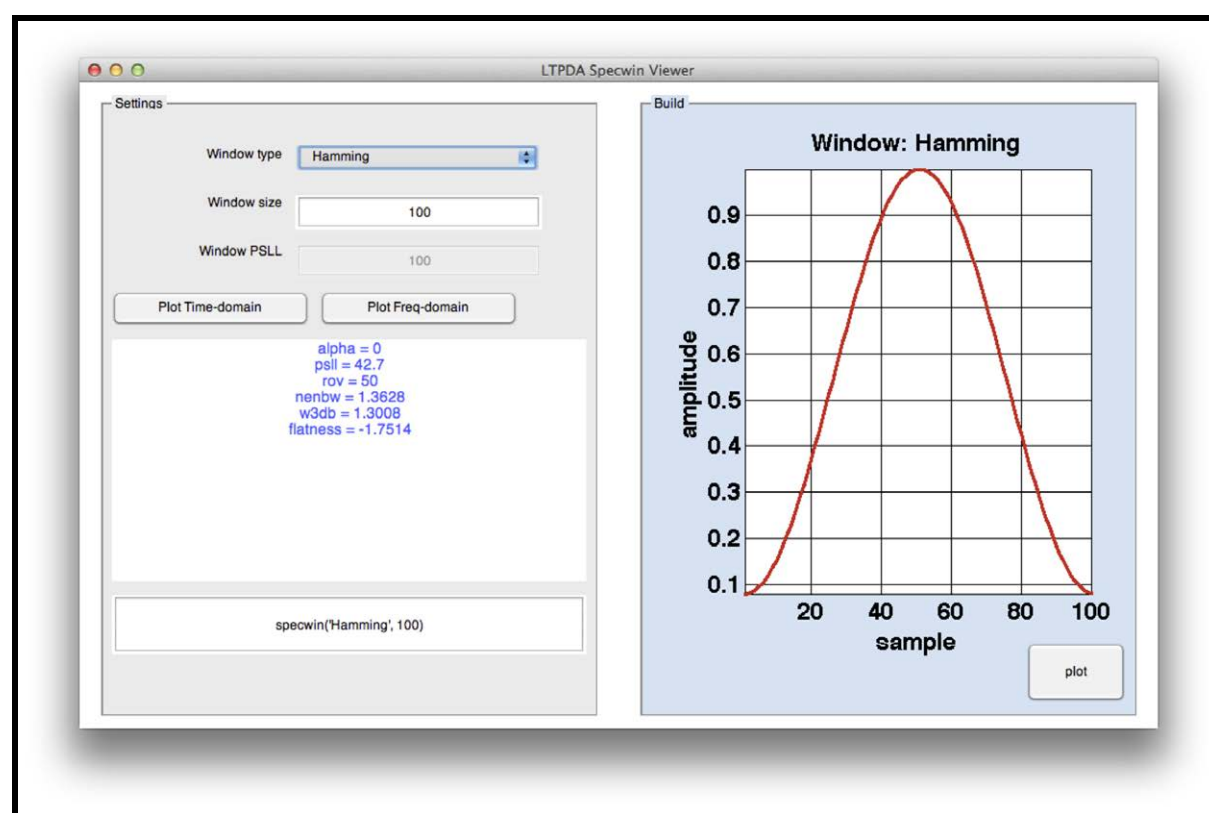
⬅ The pole/zero model helper   The constructor helper ➡

©LTP Team

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:

```
>> constructor
```

or click the appropriate button on the Launch Bay.

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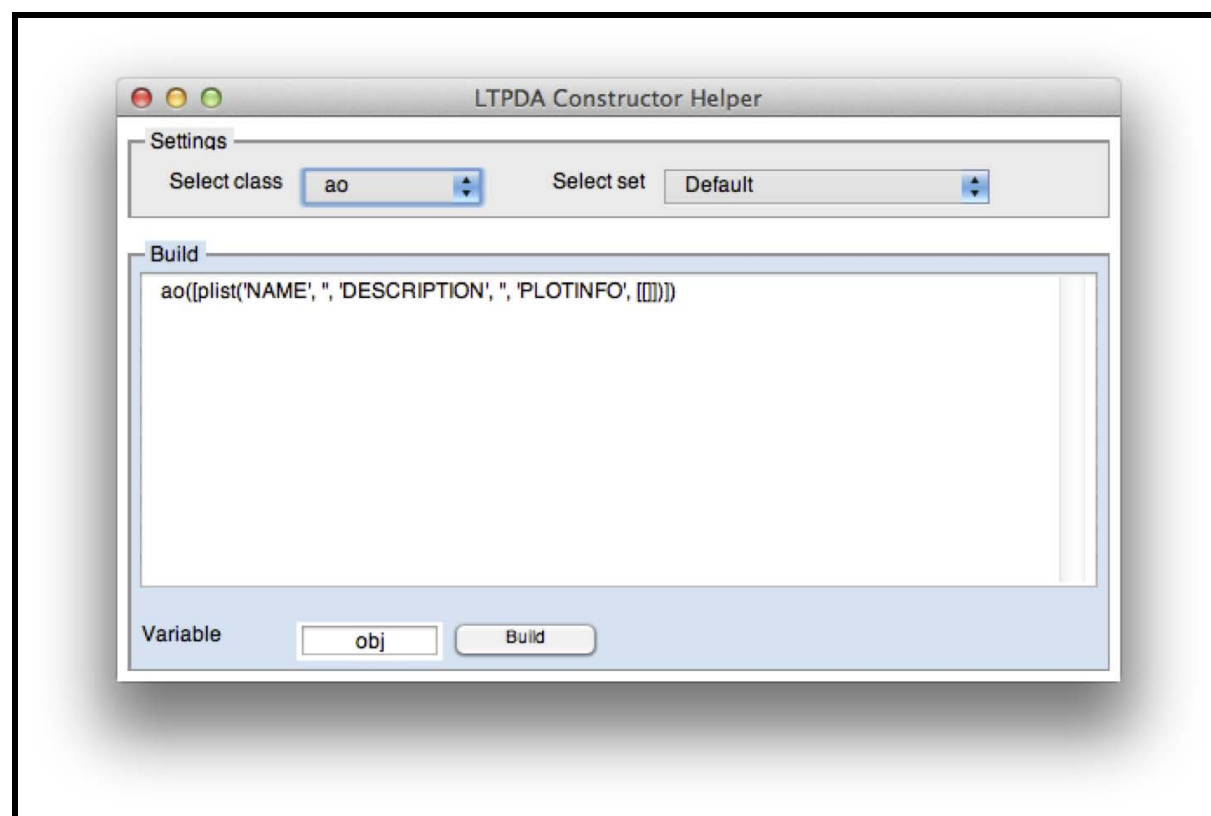
Selecting a class from the drop-down list reveals the possible parameter sets for that class constructor. Selecting a parameter set reveals the default parameter list constructor string for constructing that class object in this way. For example, if we want to construct and Analysis Object using the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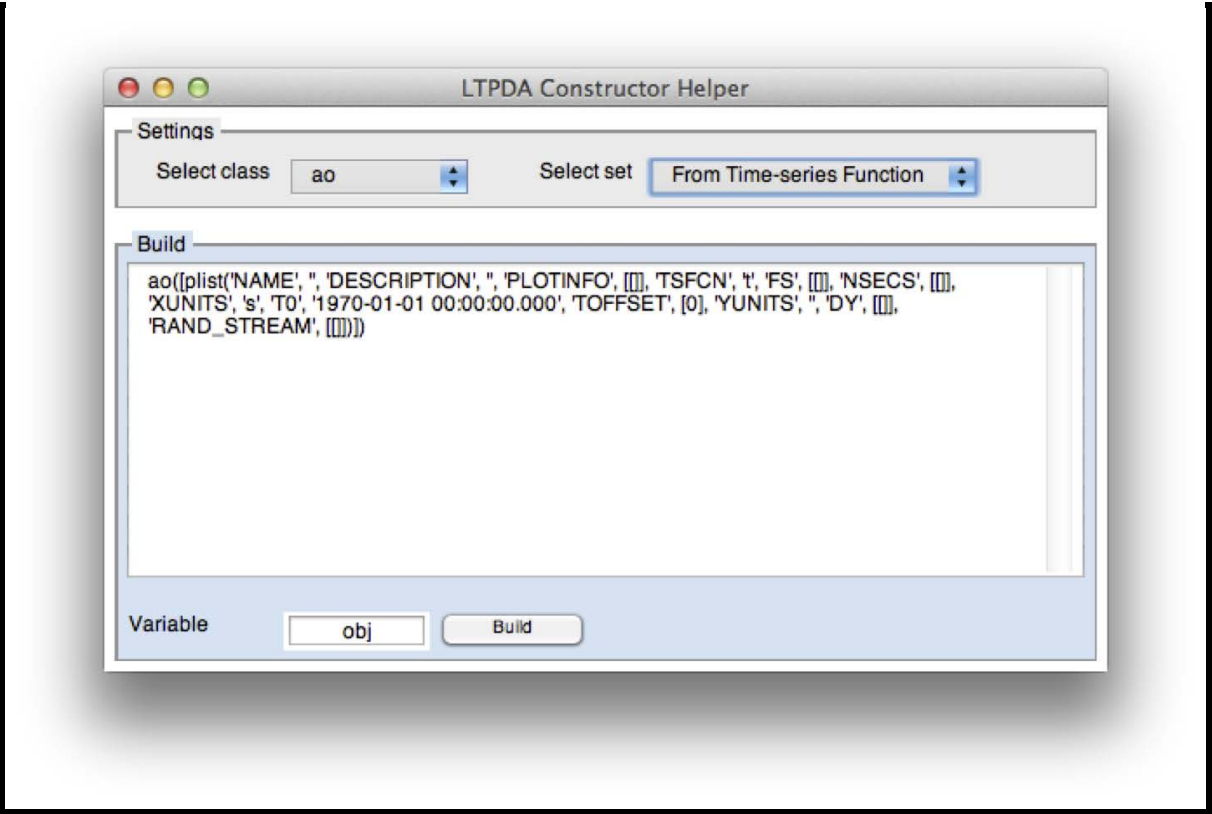
You can then edit the parameter list and build the object by clicking on `Build`.

◀ The Spectral Window GUI   The LTPDA object explorer ▶

©LTP Team

The LTPDA object explorer

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

To start the object explorer:

```
>> ltpda_explorer
```

or click the appropriate button on the Launch Bay.

The user is then presented with the following figure:
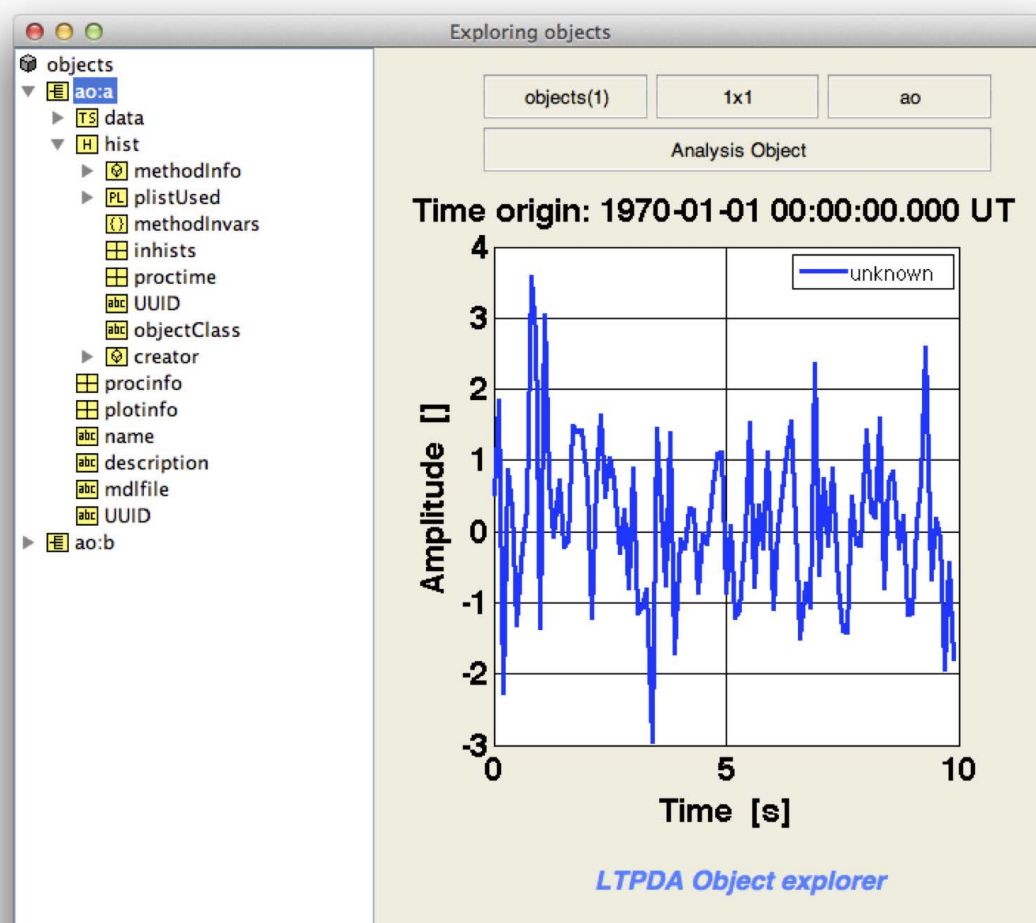
The object list is filled with all LTPDA User Objects currently in the MATLAB workspace.

©LTP Team

LTPDA Toolbox™   contents   ⬅ ➡

Working with an LTPDA Repository

This section introduces the LTPDA Repository.

- [What is an LTPDA Repository](#)
- [Connecting to an LTPDA Repository](#)
- [Submitting LTPDA objects to a repository](#)
- [Exploring an LTPDA Repository](#)
- [Retrieving LTPDA objects from a repository](#)

©LTP Team

What is an LTPDA Repository

Introduction

An LTPDA repository has at its core a database server (in fact, a [MySQL server](#)). A single MySQL server can host multiple databases (LTPDA repositories). A single database/repository comprises a particular set of database tables. These tables hold meta-data about the objects stored in the database.

Since the core engine is a MySQL database, in principle any MySQL client can be used to interface with the repository. In order to submit and retrieve objects in the proper way (entering all expected meta-data), it is strongly suggested that you use the LTPDA Toolbox client commands `submit` and the "From Repository" constructors, or the [LTPDA Workspace Browser](#). In addition, the LTPDA Workbench has full built-in support for interacting with LTPDA Repositories.

Any standard MySQL client can be used to query and search an LTPDA repository. For example, using a web-client or the standard MySQL command-line interface. In addition, the LTPDA Toolbox provides two ways to search the database: using the workspace browser, or the LTPDA Workbench.

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:

users
id : INTEGER
username : VARCHAR

transactions
direction : TEXT
id : INTEGER
obj_id : INTEGER
transdate : DATETIME
user_id : INTEGER

collections
id : INTEGER

collections2objs
id : INTEGER
obj_id : INTEGER

ao
data_type : ENUM
description : TEXT
obj_id : INTEGER

bobjs
mat : LONGBLOB
obj_id : INTEGER

objs
hash : TEXT
id : INTEGER
uuid : TEXT
xml : LONGTEXT

mfir
fs : DOUBLE
in_file : TEXT
obj_id : INTEGER

cdata
obj_id : INTEGER
yunits : TEXT

fsdata
fs : DOUBLE
obj_id : INTEGER
xunits : TEXT
yunits : TEXT

tsdata
fs : DOUBLE
nsecs : DOUBLE
obj_id : INTEGER
t0 : DATETIME
toffset : BIGINT
xunits : TEXT
yunits : TEXT

xydata

obj_id : INTEGER
xunits : TEXT
yunits : TEXT

miir
fs : DOUBLE
in_file : TEXT
obj_id : INTEGER

objmeta
additional_authors : TEXT
additional_comments : TEXT
analysis_desc : TEXT
author : TEXT
created : DATETIME
experiment_desc : TEXT
experiment_title : TEXT
hostname : TEXT
ip : TEXT
keywords : TEXT
name : TEXT
obj_id : INTEGER
obj_type : ENUM
os : TEXT
quantity : TEXT
reference_ids : TEXT
submitted : DATETIME
validated : TINYINT
vdate : DATETIME
version : TEXT

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

Database tables

An LTPDA repository consists of the following database tables:

`objs` table

The `objs` table stores the XML representation of the submitted object and assigns an unique identifier to each object in the database. Together with the database name and the hostname of the server this forms a unique tag for all LTPDA objects. The XML representaion can be dumped to a file and read with LTPDA Toolbox object constructrors.

Field Name	Data Type	Description
id	INTEGER	Object unique identification number used to link all database tables entries.
hash	TEXT	MD5 hash of the XML representation of the object.
uuid	TEXT	Universally Unique Identified assigned to the object at creation time.
xml	LONGTEXT	XML representation of the object.

`objmeta` table

The `objmeta` table stores various pieces of information associated with the object being submitted. The aim of this table is to provide useful fields on which to perform queries.

Field Name	Data	Description

	Type	
obj_id	INTEGER	Object ID.
obj_type	ENUM	Objedct type.
name	TEXT	Object user–assigned name.
created	DATETIME	Object creation time.
version	TEXT	Version of the LTPDA Toolbox used to construct the object.
ip	TEXT	IP address of the machine which submitted the object.
hostname	TEXT	Hostname of the machine which submitted the object.
os	TEXT	Operating system of the machine which submitted the object.
submitted	DATETIME	Object submission time.
experiment_title	TEXT	Title for the experiment to which the object is associated.
experiment_desc	TEXT	Description of the experiment to which the object is associated.
analysis_desc	TEXT	Description of the analysis to which the object is associated.
author	TEXT	Username of the user which submitted the object.
quantity	TEXT	Physical quantity the object describes.
additional_authors	TEXT	Other authors involved in creating the object.
additional_comments	TEXT	Free–form field for additional comments regarding the object.
keywords	TEXT	List of keywords associated to the object.
reference_ids	TEXT	Other object IDs associated with the object.
validated	TINYINT	Object validation status.
vdate	DATETIME	Object validation time.

`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 Name	Data Type	Description
obj_id	INTEGER	Object ID.
user_id	INTEGER	User ID who carried out the transaction.
transdate	DATETIME	Transaction time.
direction	TEXT	Direction of the transaction: "in" or "out".

`users` table

The `users` table stores information about the users allowed to access the database. This is a view collecting informations stored elsewhere in the database server which are replicated here for convenience in obtaining the user ID to populate the `transactions` table.

Field Name	Data Type	Description
id	INTEGER	User ID.
username	TEXT	Username.

`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 a collection composed of the submitted objects is created. This table serves the purpose of assigning an collection ID number. The collection ID allows the user to retrieve multiple objects at time.

Field Name	Data Type	Description
id	INTEGER	Collection ID.

`collections2objs` table

The `collections2objs` table links each collection to the objects that composes it and viceversa.

Field Name	Data Type	Description
id	INTEGER	Collection ID.
obj_id	INTEGER	

		Object ID.

`ao` table

The `ao` table stores additional meta-data specific to analysis objects.

Field Name	Data Type	Description
obj_id	INTEGER	Object ID.
data_type	ENUM	Object data type.
description	TEXT	AO description.

`bobjs` table

The `bobjs` table stores an optional Matlab bionary representation of the submitted object. This object representation can be dumped directly to a file and read with LTPDA Toolbox object constructrors much quicker that the otherwise equivalent XML format.

Field Name	Data Type	Description
obj_id	INTEGER	Object ID.
mat	LONGBLOB	Matlab binary representation of the object.

`mfir` table

The `mfir` table stores additional meta-data specific to MFIR filter objects.

Field Name	Data Type	Description
obj_id	INTEGER	Object ID.
in_file	TEXT	Input filename (if applicable) used to create the filter object.
fs	INTEGER	Sample rate of the data the filter is designed for.

`miir` table

The `miir` table stores additional meta-data specific to MIIR filter objects.

Field Name	Data Type	Description
obj_id	INTEGER	Object ID.
in_file	TEXT	Input filename (if applicable) used to create the filter object.
fs	INTEGER	Sample rate of the data the filter is designed for.

`cdata` table

The `cdata` table stores additional meta-data specific to cdata objects. Cdata objects are used to represent 1D data.

Field Name	Data Type	Description
obj_id	INTEGER	Object ID.
yunits	TEXT	Units for the contained data vector.

`fsdata` table

The `fsdata` table stores additional meta-data specific to fsdata objects. Fsdata objects are used to represent frequency-series data.

Field Names	Data Type	Description
obj_id	INTEGER	Object ID.
xunits	TEXT	Units for the contained frequency vector.
yunits	TEXT	Units for the contained data vector.
fs	DOUBLE	Sample rate of the contained data.

`tsdata` table

The `tsdata` table stores additional meta-data specific to tsdata objects. Tsdata objects are used to store time-series data.

Field	Data Type	Description
obj_id	INTEGER	Object ID.
xunits	TEXT	Units for the contained time vector.
yunits	TEXT	Units for the contained data vector.
fs	INTEGER	Sample rate of the contained data.
nsecs	DOUBLE	Duration of the contained time-series in seconds.
t0	DATETIME	Start time of the contained time-series.
toffset	BIGINT	Difference between t0 and the reference time in seconds.

`xydata` table

The `xydata` table stores additional meta-data specific to xydata objects. Xydata objects are used to store 2D data.

Field Name	Data Type	Description
obj_id	INTEGER	Object ID.
xunits	TEXT	Units for the contained X vector.
yunits	TEXT	Units for the contained Y vector.

◀ Working with an LTPDA Repository   Connecting to an LTPDA Repository ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Connecting to an LTPDA Repository

Connection to a LTPDA Repository is normally carried out as part of other processes. For example, when submitting an object to the repository using the `submit` command, the user is prompted to login to a chosen repository.

It is also possible to connect to a LTPDA Repository through a MATLAB programming API which may be used in scripts and user functions. The obtained connection object can be passed to other methods requiring access to a LTPDA Repository (for example the `submit` method) thus allowing the construction of more automated procedures for interacting with LTPDA Repositories. This is done specifying a `conn` parameter in the method plist instead of the connection parameters.

Connections to a LTPDA Repository can be obtained using the *LTPDA Database Connection Manager* which is responsible for caching connection credentilas accordingly to the user preferences and for limiting the number of simultaneously open connections.

A connection is obtained with the method:

```
% Connect to LTPDA Repository database
LTPDADatabaseConnectionManager().connect(pl)
```

where `pl` is a plist with the parameters `hostname`, `database`, `username`, `password`. All those parameters are optional, if missing the database connection manager will prompt the user for them. It is responsibility of the caller of the `connect` method to close the connection, with the `close` of the returned connection object method, once it has finished to use it.

Alternatively `pl` may contain a `conn` parameter specifying a connection object to use (this feature is useful to chain multiple operations through a single connection). If the specified parameter is not an object of the proper kind an error is thrown. In this case the caller of the `connect` method should not close the connection.

The preferred way to close the connection, accordingly to the rules expressed above, is to use an `onCleanup` handler, in the following way:

```
% Register onCleanup callback to close database connection
if ~isempty(find(pl, 'conn'))
  cleanup = onCleanup(@()conn.close());
end
```

where `pl` is the plist parameter of the `connect` method and `conn` is the returned connection.

The connection object returned by the *LTPDA Database Connection Manager* is a Java object implementing the `java.sql.Connection` interface, and therefore it may be used accordingly to the Java language [documentation](#).

For convenience, the utility function `utils.mysql.execute` is provided. It may be used to issue SQL commands to the database, obtaining the results in a convenient format. For data manipulation SQL statements (such as `INSERT`, `UPDATE`, and `DELETE` statements) it returns the number of affected rows, for data query SQL statements (such as `SELECT` statements) it returns a

2D cell array with the query result.

©LTP Team

Submitting LTPDA objects to a repository

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

There are three different methods which submit/update object(s) to the repository:

- **submit**

  Submits the given collection of objects to an LTPDA Repository. If multiple objects are submitted together, a corresponding collection entry will be made. The objects are stored as a XML representation **and** if possible a binary representation (see bsubmit).

- **bsubmit**

  Submits the given collection of objects to an LTPDA Repository. If multiple objects are submitted together, a corresponding collection entry will be made. The objects are stored **only** as a binary representation.
  In order to retrieve this object by calling a constructor it is necessary to set the key `'BINARY'` to 'yes' because the XML representation doesn't exist. For example the AO constructor ['From Repository'](#)

- **update**

  Updates the specified object with a new object.

The submission process

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

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

Submitting objects

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

`object`	The object(s) to

	submit
`pl`	A plist containing various pieces of information (see below)

The `plist` details can be found at: [Parameter Sets](#)

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

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

% Build a plist containing connection parameters and submission information
pl = plist(...
    'hostname', 'localhost', ...
    'database', 'test', ...
    'experiment title', 'Interferometer noise', ...
    'experiment description', 'Spectral estimation of interferometer output signal', ...
    'analysis description', 'Spectrum of the recorded signal', ...
    'quantity', 'photodiode output', ...
    'keywords', 'interferometer, noise, spectrum', ...
    'reference ids', '', ...
    'additional comments', 'none', ...
    'additional authors', 'no one');

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

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

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

If the `plist` contains sufficient information, it is possible to overide the appearance of the 'submission' dialog by giving the key/value pair: 'no dialog', `true`.

Submitting collections

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

◀ Connecting to an LTPDA Repository   Exploring an LTPDA Repository ▶

©LTP Team

Exploring an LTPDA Repository

Exploring an LTPDA repository is most easily achieved using the purpose-built graphical user interface. This interace is accesible either from the LTPDA workbench, or via a toolbar button the workspace browser.

The figure belows shows the query dialog that appears through either interface:
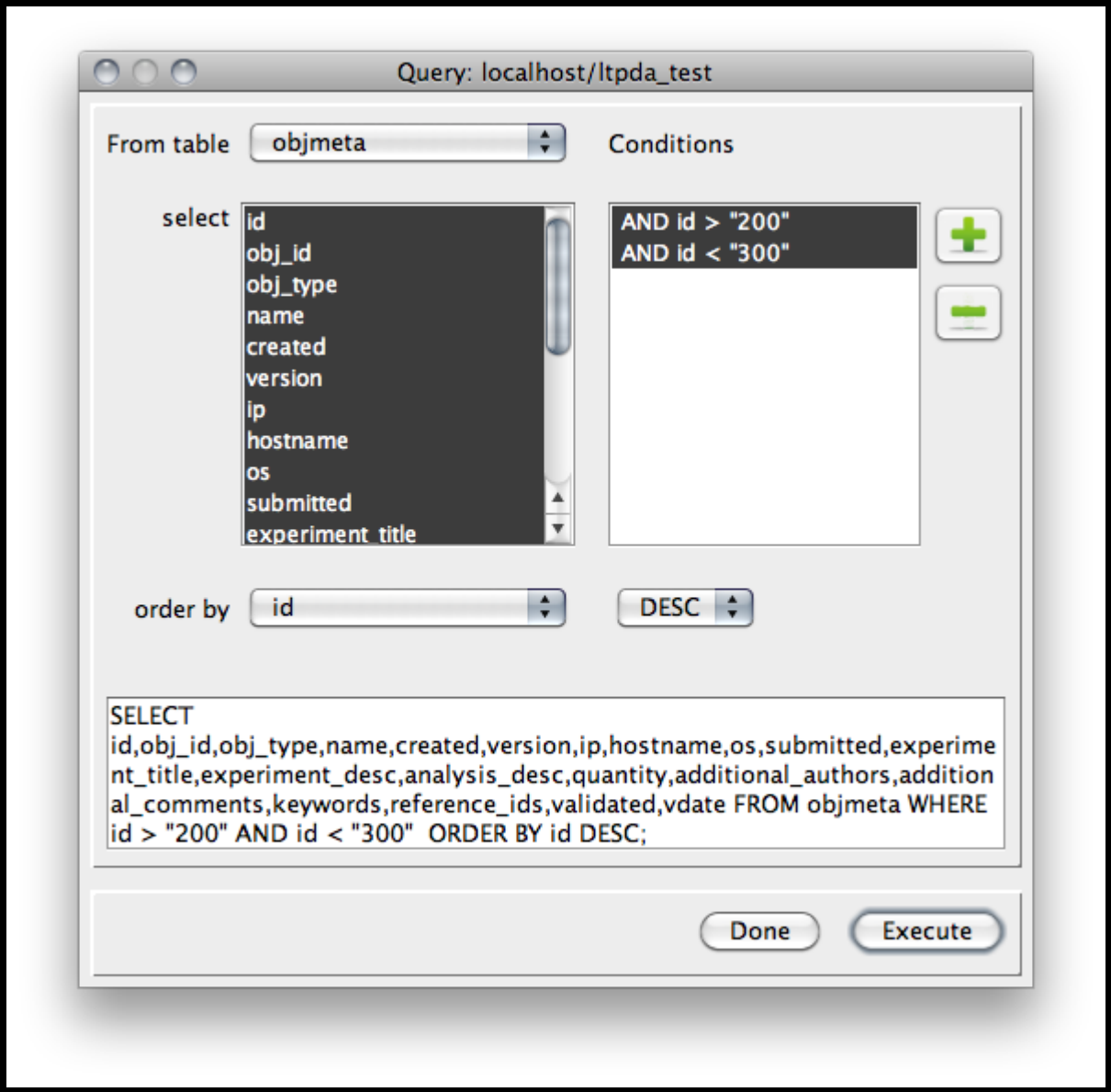
You can construct a query by use of the drop-down menus and buttons, then execute the query to retreive a table of results, like the one shown below:

If the query dialog is launched from within the LTPDA Workbench, the results table has an additional button which can be used to create constructor blocks on the current pipeline corresponding to the selected records in the results table.

◀ Submitting LTPDA objects to a repository   Retrieving LTPDA objects from a repository ▶

©LTP Team

Retrieving LTPDA objects from a repository

Objects can be retrieved from the repository either by specifying an object ID or a collection ID. The LTPDA Toolbox provides class construtors to retrieve objects. In additon, to retrieve a collection of objects, the `collection` class can be used.

The retrieval process

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

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

Retrieving objects

To retrieve an object, you must know its object ID, or the ID of the collection that contains that object. If you don't know the class of the object, you can use the `collection` class as a container, as follows:

```
a = collection(plist('hostname', 'localhost', 'database', 'ltpda_test', 'id', 1))
---- collection 1 ----
name: none
num objs: 1
01: ao | New Block/tsdataNdata=[10x1], fs=1, nsecs=10, t0=1970-01-01 00:00:00.000
description:
UUID: 9ec7a613-9442-4451-9b40-af18f4afea00
--------------------
```

If you already know the class of the object (for example, `ao`), you can directly call the class constructor method:

```
% Define the hostname and database
hostname = 'localhost';
database = 'ltpda_test';

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

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

```
a = ao(plist('hostname', 'localhost', 'database', 'ltpda_test', 'id', [1 2 3]));
```

Retrieving object collections

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

```
a = collection(plist('hostname', 'localhost', 'database', 'ltpda_test', 'cid', 1));
```

The output is a `collection` object containing the objects retrieved.

Retrieving binary objects

The retrieval process may be speeded up by taking advantage of the fact that the objects are stored in the databases also in binary form. This can be achieved by using the parameter 'binary', that will build the object from the corresponding binary representation, if stored in the database.

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

If the binary representation is not in the database, the object will be built from the XML one.

◀ Exploring an LTPDA Repository   Using the LTPDA Repository GUI ▶

LTPDA Extension Modules

As of Version 2.4, LTPDA now supports extension modules. This should allow users to extend LTPDA to provide more specific functionalities for their own context.

- [What is an Extension Module?](#)
- [Building your own Extension Module](#)
- [Installing Extension Modules](#)
- [Adding New Methods](#)
- [New User Classes](#)
- [Unit Tests](#)

What is an Extension Module?

Extension modules are a collection of class methods, new user classes, built-in models, utility functions, examples, source files, and unit tests. All extension modules have the following structure on disk:

```
My_Module/
|-- README.txt
|-- classes
|    |-- README_classes.txt
|-- examples
|-- functions
|    |-- README_functions.txt
|-- jar
|    |-- README_jar.txt
|-- models
|    |-- README_models.txt
|-- moduleinfo.xml
|-- pipelines
|    |-- README_pipelines.txt
|-- tests
     |-- README_tests.txt
     |-- classes
     |    |-- README_class_tests.txt
     |-- models
          |-- README_model_tests.txt
```

The file `moduleinfo.xml` contains the name and version of the module. It is not necessary that the module name and the containing directory are the same, though they may be. To build an extension module, see Section [Building your own Extension Module](#). The various directories and their uses are described in the following sections.

classes

The `classes` directory can contain either new LTPDA user classes (see Section [New User Classes](#)) or by adding methods to existing LTPDA user classes (see Section [Adding New Methods](#)).

examples

The `examples` directory is meant to contain useful examples for users.

jar

Since MATLAB is built on top of java, one useful way to extend the functionalities is to create java classes and methods, or even graphical user interfaces. The LTPDA startup script

(`ltpda_startup`) will take care of properly installing any jar files (java archive files) contained within this directory.

models

Here you should place any built-in models, either for existing LTPDA user classes, or for new user classes defined in this module. The LTPDA built-in model framework looks in this directory (and any sub-directories) for built-in models.

pipelines

This directory is meant to hold any LTPDA pipelines or analysis workflows which you want to distribute to users.

tests

The `tests` directory should contain unit-tests for all new class methods, user classes and built-in models in this module. For help in writing unit tests see Section Unit Tests.

Building your own Extension Module

Building your own extension modules starts by preparing the directory structure on disk. For this we have a convenient utility method:

```
>> utils.modules.buildModule('/some/path/', 'My_Module')
```

Installing Extension Modules

Installing an LTPDA Extension Module is straightforward. Start the LTPDA Preferences:

```
>> LTPDAprefs
```

Select the 'Extensions' tab. Click the 'Browse' button to locate the module directory on disk, or directly type the path to the module in the input text field. Click the 'plus' button to add this extension to the list. You should see some activity on the MATLAB terminal as LTPDA will start installing any extension methods for existing user classes. Removing an extension module is just a case of selecting the module in the list and clicking the 'minus' button.

> Note: after installing an extension module, in order to make new methods available to the workbench, you need to rebuild the library from the workbench menu: "Tools -> Rebuild LTPDA Library". This will take a couple of minutes, but afterwards the new methods from the extension module should be available on the workbench.

Adding New Methods

New methods can be added to existing LTPDA user classes. For example, you can add a new method to the Analysis Object class (`ao`) by creating a sub-directory of the `classes` directory called `ao` then put your new method in there. For example, suppose we want to create a new AO method called `myCalibration`. We create a directory `ao` in `My_Module/classes` then add the new

file `myCalibration.m` to that directory. In order for the new method to work, the LTPDA startup function `ltpda_startup` copies all methods for core LTPDA user classes in to their correct class directories. In order to write a correct LTPDA user-class method, it is recommended to look at some examples, such as `ao/abs`, `ao/average`, or `ao/psd`. You can also extend other existing user classes in the same way. Just make a directory of the correct class name (remember to leave off the `@` from the directory name; this is not supposed to be a 'normal' MATLAB class directory) and put your new methods in there.

New User Classes

This is an advanced topic, and it is assumed that you are familiary with creating MATLAB classes already. To get familiar, read the MATLAB documentation on Object-Oriented Programming in the user manual.

You can create new user classes in the `classes` directory. These follow MATLAB rules for classes, i.e., the directory name begins with a `@` and contains a constructor file with the same name. For example, suppose we want to create a new user class which stores trigger events from some experiment. Each trigger event has the following properties: a trigger time, an amplitude, and a frequency. We would create a new directory under `classes` called `@Trigger` and an associated constructor file like this:

```
>> cd 'My_Module/classes'
>> mkdir @Trigger
>> edit @Trigger/Trigger.m
```

The constructor file should declare that this new Trigger class is a subclass of the LTPDA user-object base class `ltpda_uoh`.

```matlab
% TRIGGER constructor for Trigger class.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% DESCRIPTION: TRIGGER constructor for Trigger class.
%
% CONSTRUCTOR:
%
%       t = Trigger()        - creates an empty trigger object
%
% Parameter Sets
%
% VERSION:       $Id$
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
classdef Trigger < ltpda_uoh

  %---------- Public (read/write) Properties  ----------
  properties
    time      = time();
    amplitude = [];
    frequency = [];
  end

  methods
    function obj = Trigger(varargin)

      import utils.const.*
      utils.helper.msg(msg.PROC3, 'running %s/%s', mfilename('class'), mfilename);

      % do some initialisation of the object

    end % End constructor

  end

end
```

Various methods need to be defined by any new user class, in particular, the following abstract methods need to be created:

Method name	Description
attachToDom	Defines how the object is serialized to an XML DOM.
char	Creates a character representation of the object, typically for use in display.
copy	Makes a deep or shallow copy of the object, depending on the passed argument.
display	Defines how the object is displayed on the MATLAB terminal.
fromDom	Constructs an object from an XML DOM.
loadobj	This function is called is MATLAB is unable to load an object from a MAT file, for example if the class structure changes between versions. This gives an opportunity to update the loaded structure before trying to create an object from it. For more details, see MATLAB's help topic `>>help loadobj`
update_struct	Define rules how to update structure representation of an object between versions.
generateConstructorPlist	Given an instance of the user object, this method generates a plist which can be used to construct an object with the same properties.

In most cases, copying these methods from an existing LTPDA user class, for example, `ao`, is a good start.

In addition to defining these abstract methods, you typically need to overload some static methods (which we usually place inside the class constructor file). The following code fragment shows the necessary methods needed to complete our Trigger example.

```matlab
methods (Static)

  % This provides the hook for the command <class>.getBuiltInModels.
  function mdls = getBuiltInModels(varargin)
    mdls = ltpda_uo.getBuiltInModels('Trigger');
  end

  % Here we typically return the CVS version or some other version string
  function out = VEROUT()
    out = '$Id$';
  end

  % This provides the hook for the command <class>.getInfo.
  function ii = getInfo(varargin)
    ii = utils.helper.generic_getInfo(varargin{:}, 'Trigger');
  end

  % Here we return a list of parameter sets that this constructor can handle.
  function out = SETS()
    out = [SETS@ltpda_uoh, ...
      {'Default'}          ...
      ];
  end
```

```matlab
    % This returns a parameter list for a given parameter set.
    % The use of the MATLAB 'persistent' keyword means we don't repeatedly build the same
plist.
    function plout = getDefaultPlist(set)
      persistent pl;
      persistent lastset;
      if exist('pl', 'var')==0 || isempty(pl) || ~strcmp(lastset, set)
        pl = Trigger.buildplist(set);
        lastset = set;
      end
      plout = pl;
    end

    % This builds a parameter list for the given set name.
    function out = buildplist(set)

      if ~utils.helper.ismember(lower(Trigger.SETS), lower(set))
        error('### Unknown set [%s]', set);
      end

      out = plist();
      out = Trigger.addGlobalKeys(out);
      out = buildplist@ltpda_uoh(out, set);

      % Build the requested parameter list.
      switch lower(set)
        case 'Default'
          % time
          p = param({'time','The time of the trigger. Give either a string representation or
a time object.'}, paramValue.EMPTY_STRING);
          out.append(p);

          % amplitude
          p = param({'amplitude','The trigger amplitude.'}, paramValue.EMPTY_DOUBLE);
          out.append(p);

          % frequency
          p = param({'frequency','The trigger frequency.'}, paramValue.EMPTY_DOUBLE);
          out.append(p);

      end
    end % function out = getDefaultPlist(varargin)

    % This creates arrays of the given size containing empty Trigger objects.
    function obj = initObjectWithSize(n,m)
      obj = Trigger.newarray([n m]);
      for ii = 1:numel(obj)
        obj(ii).UUID = char(java.util.UUID.randomUUID);
      end
    end

  end % End static methods

  methods (Static, Access=protected)

    % Global keys are added to all parameter lists so that properties common to all
    % user objects can be set.
    function pl = removeGlobalKeys(pl)
      pl.remove('name');
      pl.remove('description');
    end

    function pl = addGlobalKeys(pl)
      % Name
      p = param({'Name','The name of the constructed trigger object.'},
paramValue.STRING_VALUE('None'));
      pl.append(p);

      % Description
      p = param({'Description','The description of the constructed trigger object.'},
paramValue.EMPTY_STRING);
      pl.append(p);
    end

  end % End static, private methods

  methods (Static = true, Hidden = true)
    varargout = loadobj(varargin)
    varargout = update_struct(varargin);
  end

  methods
    varargout = char(varargin)
    varargout = display(varargin)
```

```
    varargout = copy(varargin)
  end

  methods (Hidden = true)
    varargout = attachToDom(varargin)
  end

  methods (Access = protected)
    obj = fromStruct(obj, a_struct)
    varargout = fromDom(varargin)
  end
```

Unit Tests

LTPDA provides a unit test framework that aims to make it easy to test your new methods, classes and built-in models. Unit tests are methods of a unit test class. These unit test classes should inherit from one of the base classes `ltpda_utp` or `ltpda_builtin_model_utp`. The directory `ltpda_toolbox/ltpda/classes/tests/` contains these test classes and examples for testing classes and built-in models. To run the unit tests you can use the unit test runner class `ltpda_test_runner`. For example,

```
>> ltpda_test_runner.RUN_ALL_TESTS
>> ltpda_test_runner.RUN_TESTS('@my_class_tests')
>> ltpda_test_runner.RUN_TESTS('@my_class_tests', 'test_a_particular_method_feature')
```

◀ Retrieving objects and collections from a repository   Class descriptions ▶

©LTP Team

Class descriptions

[MCMC class](#)	### Please define a description for the [MCMC] class in docHelper.getDescriptionForClass.m
[AO class](#)	A class of object that implements Analysis Objects. Such objects can store various types of numeric data: time-series, frequency-series , x-y data series, x-y-z data series, and arrays of numbers.
[COLLECTION class](#)	A class of object that allows storing other User Objects in a cell-array. This is purely a convenience class, aimed at simplifying keeping a collection of different classes of user objects together. This is useful, for example, for submitting or retrieving a collection of user objects from/to an LTPDA Repository.
[FILTERBANK class](#)	A class of object that represents a bank of digital filters. The filter bank can be of type 'parallel' or 'serial'.
[MATRIX class](#)	A class of object that allows storing other User Objects in a matrix-like array. There are various methods which act on the object array as if it were a matrix of the underlying data. For example, you can form a matrix of Analysis Objects, then compute the determinant of this matrix, which will yield another matrix object containing a single AO.
[MFH class](#)	### Please define a description for the [mfh] class in docHelper.getDescriptionForClass.m
[MFIR class](#)	A class of object that implements Finite Impulse Response (FIR) digital filters.
[MIIR class](#)	class of object that implements Infinite Impulse Response (IIR) digital filters.
[PARFRAC class](#)	A class of object that represents transfer functions given as a series of partial fractions.
[PEST class](#)	A class of objects that represents parameter estimation which aims to capture details of the various fitting procedures in LTPDA.
[PLIST class](#)	A class of object that allows storing a set of key/value pairs. This is the equivalent to a 'dictionary' in other languages, like python or Objective-C. Since a history step must contain a plist object, this class cannot itself track history, since it would be recursive.
[PZMODEL class](#)	A class of object that represents transfer functions given in a pole/zero format.
[RATIONAL class](#)	A class of object that represents transfer functions given in rational form.
[SMODEL class](#)	A class of object that represents a parametric model of a chosen x variable. Such objects can be combined, manipulated symbolically, and then evaluated numerically.

| SSM class | A class of object that represents a statespace model. |
| TIMESPAN class | A class of object that represents a span of time. A timespan object has a start time and an end time. |

◀ LTPDA Extension Modules   ao Class ▶

©LTP Team

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	Defining Class
[data](#)	Data object associated with this AO	ao
[hist](#)	History of the object (history object)	ltpda_uoh
[procinfo](#)	plist with additional information for an object.	ltpda_uoh
[plotinfo](#)	plotinfo object	ltpda_uoh
[name](#)	name of the object	ltpda_uo
[description](#)	description of the object	ltpda_uo
[UUID](#)	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

[Arithmetic Operator](#)	Arithmetic Operator
[Constructor](#)	Constructor of this class
[Converter](#)	Convertor methods
[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	Defining Class
[and](#)	(&) overloads the and (&) method for analysis objects.	ao
[eq](#)	Overloads == operator for analysis objects. Compare the y-axis values.	ao
[ge](#)	Overloads >= operator for analysis objects. Compare the y-axis values.	ao
[gt](#)	Overloads > operator for analysis objects. Compare the y-axis values.	ao
[le](#)	Overloads <= operator for analysis objects. Compare the y-axis values.	ao
[lt](#)	Overloads < operator for analysis objects. Compare the y-axis values.	ao
[minus](#)	Implements subtraction operator for analysis objects.	ao
[mpower](#)	Implements mpower operator for analysis objects.	ao
[mrdivide](#)	Implements mrdivide operator for analysis objects.	ao

mtimes	Implements mtimes operator for analysis objects.	ao
ne	Overloads ~= operator for analysis objects. Compare the y-axis values.	ao
or	(|) overloads the or (|) method for Analysis objects.	ao
plus	Implements addition operator for analysis objects.	ao
power	Implements power operator for analysis objects.	ao
rdivide	Implements division operator for analysis objects.	ao
times	Implements multiplication operator for analysis objects.	ao

🔺 Back to Top of Section

Constructor

Methods	Description	Defining Class
ao	Analysis object class constructor.	ao
rebuild	Rebuilds the input objects using the history.	ltpda_uoh

🔺 Back to Top of Section

Converter

Methods	Description	Defining Class
cast	– converts the numeric values in an AO to the data type specified by type.	ao
double	Overloads double() function for analysis objects.	ao
logical	Overloads logical() function for analysis objects.	ao

🔺 Back to Top of Section

Helper

Methods	Description	Defining Class
convert	Perform various conversions on the ao.	ao
demux	Splits the input vector of AOs into a number of output AOs.	ao
dx	Get the data property 'dx'.	ao
dy	Get the data property 'dy'.	ao
dz	DX Get the data property 'dz'.	ao
find	Particular samples that satisfy the input query and return a new AO.	ao
fromProcinfo	Returns for a given key-name the value of the procinfo-plist	ao
fs	Get the data property 'fs'.	ao
join	Multiple AOs into a single AO.	ao
len	Overloads the length operator for Analysis objects. Length of the data samples.	ao
nsecs	Get the data property 'nsecs'.	ao
scatter3D	Creates from the y-values of the input AOs a new AO with a xyz-data object	ao
scatterData	Creates from the y-values of two input AOs an new AO(xydata)	ao
setDx	Sets the 'dx' property of the ao.	ao
setDy	Sets the 'dy' property of the ao.	ao
setDz	Sets the 'dz' property of the ao.	ao
setEnbw	Sets the 'enbw' property of the ao/fsdata.	ao
setFs	Sets the 'fs' property of the ao.	ao
setNavs	Sets the 'navs' property of the ao/fsdata.	ao
setReferenceTime	Sets the t0 to the new value but doesn't move the data in time	ao
setT0	Sets the 't0' property of the ao.	ao
setToffset	Sets the 'toffset' property of the ao with tsdata	ao
setX	Sets the 'x' property of the ao.	ao
setXY	Sets the 'x' and 'y' properties of the ao.	ao
setXaxisName	Sets the x-axis name of the ao.	ao
setXunits	Sets the 'xunits' property of the ao.	ao
setY	Sets the 'y' property of the ao.	ao
setYaxisName	Sets the y-axis name of the ao.	ao
setYunits	Sets the 'yunits' property of the ao.	ao
setZ	Sets the 'z' property of the ao.	ao
setZaxisName	Sets the z-axis name of the ao.	ao
setZunits	Sets the 'zunits' property of the ao.	ao

simplifyXunits	Simplify the 'xunits' of the ao.	ao
simplifyYunits	Simplify the 'yunits' property of the ao.	ao
simplifyZunits	Simplify the 'zunits' of the ao.	ao
t0	Get the data property 't0'.	ao
timeshift	For AO/tsdata objects, shifts data in time by the specified value in seconds.	ao
toffset	Get the data property 'toffset'.	ao
validate	Checks that the input Analysis Object is reproducible and valid.	ao
x	Get the data property 'x'.	ao
xaxisname	Get the x axis name of the underlying data object.	ao
xunits	Get the data property 'xunits'.	ao
y	Get the data property 'y'.	ao
yaxisname	Get the y axis name of the underlying data object.	ao
yunits	Get the data property 'yunits'.	ao
z	Get the data property 'z'.	ao
zaxisname	Get the z axis name of the underlying data object.	ao
zunits	Get the data property 'zunits'.	ao
get	Get a property of a object.	ltpda_obj
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh

🔺 Back to Top of Section

Internal

Methods	Description	Defining Class
char	Overloads char() function for analysis objects.	ao
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	ao
md5	Computes an MD5 checksum from an analysis objects.	ao
plot	The analysis objects on the given axes.	ao
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh

🔺 Back to Top of Section

Operator

Methods	Description	Defining

	Class
abs	Overloads the Absolute value method for analysis objects.	ao
all	Overloads the all operator for analysis objects. True if all elements	ao
angle	Overloads the angle operator for analysis objects.	ao
any	Overloads the any operator for analysis objects. True if any element	ao
ao2numMatrices	AO2NUMMATRICES.m	ao
bilinfit	Is a linear fitting tool	ao
cat	Concatenate AOs into a row vector.	ao
complex	Overloads the complex operator for Analysis objects.	ao
conj	Overloads the conjugate operator for analysis objects.	ao
ctranspose	Overloads the ' operator for Analysis Objects.	ao
cumsum	Overloads the cumsum operator for analysis objects.	ao
det	Overloads the determinant function for analysis objects.	ao
diag	Overloads the diagonal operator for analysis objects.	ao
eig	Overloads the eigenvalues/eigenvectors function for analysis objects.	ao
exp	Overloads the exp operator for analysis objects. Exponential.	ao
hypot	Overloads robust computation of the square root of the sum of squares for AOs.	ao
imag	Overloads the imaginary operator for analysis objects.	ao
inv	Overloads the inverse function for analysis objects.	ao
ln	Overloads the log operator for analysis objects. Natural logarithm.	ao
log	Overloads the log operator for analysis objects. Natural logarithm.	ao
log10	Overloads the log10 operator for analysis objects. Common (base 10) logarithm.	ao
max	Computes the maximum value of the data in the AO	ao
mean	Computes the mean value of the data in the AO.	ao
median	Computes the median value of the data in the AO.	ao
min	Computes the minimum value of the data in the AO	ao
mode	Computes the modal value of the data in the AO.	ao
norm	Overloads the norm operator for analysis objects.	ao
offset	Adds an offset to the data in the AO.	ao
phase	Is the phase operator for analysis objects.	ao
quasiSweptSine	QUASISWEPTSING computes a transfer function from swept-sine measurements	ao
real	Overloads the real operator for analysis objects.	ao
rotate	Applies rotation factor to AOs	ao
round	Overloads the Round method for analysis objects.	ao
scale	Scales the data in the AO by the specified factor.	ao
sign	Overloads the sign operator for analysis objects.	ao
sort	The values in the AO.	ao
spsdSubtraction	Makes a sPSD-weighted least-square iterative fit	ao
sqrt	Computes the square root of the data in the AO.	ao
std	Computes the standard deviation of the data in the AO.	ao
sum	Computes the sum of the data in the AO.	ao
sumjoin	Sums time-series signals togther	ao
svd	Overloads the svd (singular value decomposition) function for analysis objects.	ao
transpose	Overloads the .' operator for Analysis Objects.	ao
uminus	Overloads the uminus operator for analysis objects.	ao
unwrap	Overloads the unwrap operator for analysis objects.	ao
var	Computes the variance of the data in the AO.	ao

🔺 Back to Top of Section

Output

Methods	Description	Defining Class
display	Implement terminal display for analysis object.	ao
export	Export the data of an analysis object to a text file.	ao
gnuplot	A gnuplot interface for AOs.	ao
iplot	Provides an intelligent plotting tool for LTPDA.	ao
iplotyy	Provides an intelligent plotting tool for LTPDA.	ao
table	Display the data from the AO in a table.	ao
save	Overloads save operator for ltpda objects.	ltpda_uo

| report | Generates an HTML report about the input objects. | ltpda_uoh |
| viewHistory | Displays the history of an object as a dot-view or a MATLAB figure. | ltpda_uoh |

🔺 Back to Top of Section

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

🔺 Back to Top of Section

Signal Processing

Methods	Description	Defining Class
average	Averages aos point-by-point	ao
bandpass	AOs containing time-series data.	ao
bandreject	AOs containing time-series data.	ao
bicohere	Computes the bicoherence of two input time-series	ao
bin_data	Rebins aos data, on logarithmic scale, linear scale, or arbitrarly chosen.	ao
buildWhitener1D	Builds a whitening filter based on the input frequency-series.	ao
cdfplot	Makes cumulative distribution plot	ao
cgfill	Fills specified gaps in the data given an inital guess for the spectrum.	ao
cohere	Estimates the coherence between time-series objects	ao
compute	Performs the given operations on the input AOs.	ao
confint	Calculates confidence levels and variance for psd, lpsd, cohere, lcohere and curvefit parameters	ao
consolidate	Resamples all input AOs onto the same time grid.	ao
conv	Vector convolution.	ao
corr	Estimate linear correlation coefficients.	ao
cov	Estimate covariance of data streams.	ao
cpsd	Estimates the cross-spectral density between time-series objects	ao
crb	Computes the inverse of the Fisher Matrix	ao
crbound	Computes the inverse of the Fisher Matrix	ao
delay	Delays a time-series using various methods.	ao
delayEstimate	Estimates the delay between two AOs	ao
detectOutliers	Locates outliers in data.	ao
detrend	Detrends the input analysis object using a polynomial of degree N.	ao
dft	Computes the DFT of the input time-series at the requested frequencies.	ao
diff	Differentiates the data in AO.	ao
dispersionLoop	Computes the dispersion function	ao
downsample	AOs containing time-series data.	ao
dropduplicates	Drops all duplicate samples in time-series AOs.	ao
dsmean	Performs a simple downsampling by taking the mean of every N samples.	ao
ecdf	Calculate empirical cumulative distribution function	ao
edgedetect	Detects edges in a binary pulse-train.	ao
eqmotion	Solves numerically a given linear equation of motion	ao
evaluateModel	Evaluate a curvefit model.	ao
fft	Overloads the fft method for Analysis objects.	ao
fftfilt	Overrides the fft filter function for analysis objects.	ao
filtSubtract	Subtracts a frequency dependent noise contribution from an input ao.	ao
filter	Overrides the filter function for analysis objects.	ao
filtfilt	Overrides the filtfilt function for analysis objects.	ao
firwhiten	Whitens the input time-series by building an FIR whitening filter.	ao
fixfs	Resamples the input time-series to have a fixed sample rate.	ao
fngen	Creates an arbitrarily long time-series based on the input PSD.	ao
gapfilling	Fills possible gaps in data.	ao
gapfillingoptim	Fills possible gaps in data.	ao
getAbsTimeRange	Returns a timespan object which span the absolute time range of an AO	ao
getKSCValPSD	Provides critical value for KStest on the PSD	ao
getdof	Calculates degrees of freedom for psd, lpsd, cohere and lcohere	ao
heterodyne	Heterodynes time-series.	ao
highpass	AOs containing time-series data.	ao
hist	Overloads the histogram function (hist) of MATLAB for Analysis Objects.	ao
iacf	Computes the inverse auto-correlation function from a spectrum.	ao

ifft	Overloads the ifft operator for Analysis objects.	ao
integrate	Integrates the data in AO.	ao
interp	Interpolate the values in the input AO(s) at new values.	ao
interpmissing	Interpolate missing samples in a time-series.	ao
intersect	Overloads the intersect operator for Analysis objects.	ao
kstest	Perform KS test on input AOs	ao
lcohere	Implement magnitude-squadred coherence estimation on a log frequency axis.	ao
lcpsd	Implement cross-power-spectral density estimation on a log frequency axis.	ao
linSubtract	Subtracts a linear contribution from an input ao.	ao
lincom	Make a linear combination of analysis objects	ao
linedetect	Find spectral lines in the ao/fsdata objects.	ao
linfit	Is a linear fitting tool	ao
linlsqsvd	Linear least squares with singular value decomposition	ao
lisovfit	Uses LISO to fit a pole/zero model to the input frequency-series.	ao
lowpass	AOs containing time-series data.	ao
lpsd	Implements the LPSD algorithm for analysis objects.	ao
lscov	Is a wrapper for MATLAB's lscov function.	ao
ltfe	Implements transfer function estimation computed on a log frequency axis.	ao
map3D	Maps the input 1 or 2D AOs on to a 3D AO	ao
mcmc	Estimates paramters using a Monte Carlo Markov Chain.	ao
mcmc_td	Estimates paramters using a Monte Carlo Markov Chain.	ao
mhsample	The Metropolis - Hastings algorithm	ao
modelSelect	%%%	ao
mve	MVE: Minimum Volume Ellipsoid estimator	ao
noisegen1D	Generates colored noise from white noise.	ao
noisegen2D	Generates cross correleted colored noise from white noise.	ao
normdist	Computes the equivalent normal distribution for the data.	ao
overlap	This method cuts out the the overlapping data of the input AOs.	ao
pad	Pads the input data series to a given value.	ao
play	Plays a time-series using MATLAB's audioplay function.	ao
polyfit	Overloads polyfit() function of MATLAB for Analysis Objects.	ao
polyfitSpectrum	Does a polynomial fit to the log of the input spectrum.	ao
polynomfit	Is a polynomial fitting tool	ao
powerFit	Fits a piecewise powerlaw to the given data.	ao
ppsd	Ppsd.m	ao
preprocessDataForMCMC	PreprocessDataForMCMC.	ao
psd	Makes power spectral density estimates of the time-series objects	ao
psdconf	Calculates confidence levels and variance for psd	ao
qqplot	QQFPLOT makes quantile-quantile plot	ao
removeVal	Removes values from the input AO(s).	ao
resample	Overloads resample function for AOs.	ao
rjsample	%%%	ao
rms	Calculate RMS deviation from spectrum	ao
sDomainFit	Performs a fitting loop to identify model order and	ao
select	Select particular samples from the input AOs and return new AOs with only those samples.	ao
simplex	Multidimensional unconstrained nonlinear minimization (Nelder-Mead)	ao
sineParams	Estimates parameters of sinusoids	ao
smoother	Smooths a given series of data points using the specified method.	ao
spcorr	Calculate Spearman Rank-Order Correlation Coefficient	ao
spectrogram	Computes a spectrogram of the given ao/tsdata.	ao
spikecleaning	Detects and corrects possible spikes in analysis objects	ao
split	Split an analysis object into the specified segments.	ao
spsd	Implements the smoothed (binned) PSD algorithm for analysis objects.	ao
subsData	Performs actions on ao objects.	ao
svd_fit	Estimates parameters for a linear model using SVD	ao
tdfit	Fit a set of smodels to a set of input and output signals..	ao
tfe	Estimates transfer function between time-series objects.	ao
timeaverage	Averages time series intervals	ao
toSI	Converts the units of the x, y and z axes into SI units.	ao
upsample	Overloads upsample function for AOs.	ao
whiten1D	Whitens the input time-series.	ao
whiten2D	Whiten the noise for two cross correlated time series.	ao
xcorr	Makes cross-correlation estimates of the time-series	ao

xfit	Fit a function of x to data.	ao
zDomainFit	Performs a fitting loop to identify model order and	ao
zeropad	Zero pads the input data series.	ao

🔺 Back to Top of Section

Trigonometry

Methods	Description	Defining Class
acos	Overloads the acos method for analysis objects.	ao
asin	Overloads the asin method for analysis objects.	ao
atan	Overloads the atan method for analysis objects.	ao
atan2	Overloads the atan2 operator for analysis objects. Four quadrant inverse tangent.	ao
cos	Overloads the cos method for analysis objects.	ao
sin	Overloads the sin method for analysis objects.	ao
tan	Overloads the tan method for analysis objects.	ao

🔺 Back to Top of Section

◀ Class descriptions	collection Class ▶

©LTP Team

collection Class

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

🔺 Back to Class descriptions

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:
obj2 = obj1.setName('my name')  % This command creates a copy of obj1 (obj1 ~= obj2)
obj.setName('my name');         % This command applies to obj
```

Properties	Description	Defining Class
objs	objects in collection	collection
names	-- No description found --	collection
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

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

Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

[▲](#) Back to Top

Constructor

Methods	Description	Defining Class
collection	Constructor for collection class.	collection
rebuild	Rebuilds the input objects using the history.	ltpda_uoh

[▲](#) Back to Top of Section

Helper

Methods	Description	Defining Class
addObjects	Adds the given objects to the collection.	collection
getObjectAtIndex	Index into the inner objects of one collection object.	collection
getObjectsOfClass	Returns all objects of the specified class in a collection-object.	collection
nobjs	Returns the number of objects in the inner object array.	collection
removeObjectAtIndex	Removes the object at the specified position from the collection.	collection
setObjectAtIndex	Sets an input object to the collection.	collection
setObjs	Sets the 'objs' property of a collection object.	collection
search	Selects objects inside the collection/matrix object that match the given name.	ltpda_container
get	Get a property of a object.	ltpda_obj
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
requirements	Returns a list of LTPDA extension	ltpda_uoh

	requirements for a given object.	
setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh

 Back to Top of Section

Internal

Methods	Description	Defining Class
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	collection
plot	The collection objects.	collection
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB	ltpda_uoh

functions.

Operator

Methods	Description	Defining Class
conj	Implements conj operator for ltpda_container objects.	ltpda_container

Output

Methods	Description	Defining Class
char	Convert a collection object into a string.	collection
display	Overloads display functionality for collection objects.	collection
unpack	Unpacks the objects in a collection and sets them to the given output	collection
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

Signal Processing

Methods	Description	Defining Class
cohere	Estimates the coherence between time-series objects in a collection object.	collection
cpsd	Estimates the cross-spectral density between time-series objects in a collection object.	collection
filter	Overrides the filter function for analysis objects in a collection object.	collection
filtfilt	Overrides the filtfilt function for analysis objects in a collection object.	collection
tfe	Estimates the transfer function between time-series objects in a collection object.	collection
abs	Overloads the Absolute value method for analysis objects in a ltpda_container object.	ltpda_container
consolidate	Resamples all input AOs in a ltpda_container	ltpda_container

	object onto the same time grid.	
detrend	Detrends the analysis objects in a ltpda_container object using a polynomial of degree N.	ltpda_container
diff	Differentiates the data in a ltpda_container object.	ltpda_container
downsample	Downsamples each time-series AO in the ltpda_container.	ltpda_container
dsmean	Resamples each time-series AO in the ltpda_container.	ltpda_container
fft	Implements the fft operator for ltpda_container objects.	ltpda_container
fixfs	Adjusts the sample frequency of each time-series AO in the ltpda_container.	ltpda_container
heterodyne	Heterodynes time-series in a ltpda_container object.	ltpda_container
interp	Interpolate the values of each AO in the ltpda_container at new values.	ltpda_container
interpmissing	Interpolate missing samples of each time-series AO in the ltpda_container.	ltpda_container
lpsd	Computes the log-scale PSD of the time-series AOs in a ltpda_container object.	ltpda_container
polyfit	Overloads polyfit() function of MATLAB for ltpda_container objects.	ltpda_container
psd	Computes the PSD of the time-series in a ltpda_container object.	ltpda_container
removeVal	Removes values from each AO in the ltpda_container.	ltpda_container
resample	Resamples each time-series AO in the ltpda_container.	ltpda_container
split	Splits a ltpda_container object into the specified segments.	ltpda_container
sqrt	Computes the sqrt of each object in the ltpda_container.	ltpda_container
timeaverage	Averages time series intervals in a ltpda_container object.	ltpda_container
uminus	Overloads the uminus operator for all AOs in the ltpda_container.	ltpda_container

 Back to Top of Section

©LTP Team

filterbank Class

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

🔺 Back to Class descriptions

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:
obj2 = obj1.setName('my name')  % This command creates a copy of obj1 (obj1 ~= obj2)
obj.setName('my name');         % This command applies to obj
```

Properties	Description	Defining Class
filters	Filters of the bank	filterbank
type	Type of the bank	filterbank
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

Arithmetic Operator	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
Signal Processing	Signal processing methods

Arithmetic Operator

Methods	Description	Defining Class
addFilters	This method adds a filter to the filterbank	filterbank

Constructor

Methods	Description	Defining Class
filterbank	Constructor for filterbank class.	filterbank
rebuild	Rebuilds the input objects using the history.	ltpda_uoh

Helper

Methods	Description	Defining Class
setIunits	Sets the 'iunits' property of each filter–object inside the filterbank–object.	filterbank
setOunits	Sets the 'ounits' property of each filter–object inside the filterbank–object.	filterbank
get	Get a property of a object.	ltpda_obj
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
plot	Plots the user object on a figure.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's	ltpda_uoh

	plotinfo.	
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh

 Back to Top of Section

Internal

Methods	Description	Defining Class
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	filterbank
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh

 Back to Top of Section

Output

Methods	Description	Defining Class
char	Convert a filterbank object into a string.	filterbank
display	Overloads display functionality for filterbank objects.	filterbank
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh

▲ Back to Top of Section

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

▲ Back to Top of Section

Signal Processing

Methods	Description	Defining Class
resp	Make a frequency response of a filterbank.	filterbank

▲ Back to Top of Section

◀ collection Class	matrix Class ▶

©LTP Team

matrix Class

[Properties](#)	Properties of the class
[Methods](#)	All Methods of the class ordered by category.
[Examples](#)	Some constructor examples

🔺 [Back to Class descriptions](#)

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defining Class
[objs](#)	objects in matrix	matrix
[hist](#)	history of the object (history object)	ltpda_uoh
[procinfo](#)	plist with additional information for an object.	ltpda_uoh
[plotinfo](#)	plotinfo object	ltpda_uoh
[name](#)	name of the object	ltpda_uo
[description](#)	description of the object	ltpda_uo
[UUID](#)	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 [Back to Top](#)

Methods

[Arithmetic Operator](#)	Arithmetic Operator
[Constructor](#)	Constructor of this class
[Converter](#)	Convertor methods
[Helper](#)	Helper methods only for internal usage
[Internal](#)	Internal methods only for internal usage
[Operator](#)	Operator methods
[Output](#)	Output methods
[Relational Operator](#)	Relational operator methods
[Signal Processing](#)	Signal processing methods

🔺 [Back to Top](#)

Arithmetic Operator

Methods	Description	Defining Class
[ctranspose](#)	Implements conjugate transpose operator for matrix objects.	matrix
[filter](#)	Implements N-dim filter operator for matrix objects.	matrix
[filtfilt](#)	Overrides the filtfilt function for matrices of analysis objects.	matrix
[minus](#)	Implements subtraction operator for ltpda model objects.	matrix
[mtimes](#)	Implements mtimes operator for matrix objects.	matrix
[plus](#)	Implements addition operator for matrix objects.	matrix
[power](#)	TIMES implements multiplication operator for matrix objects.	matrix
[rdivide](#)	Implements division operator for matrix objects.	matrix
[times](#)	Implements multiplication operator for matrix objects.	matrix
[transpose](#)	Implements transpose operator for matrix objects.	matrix

Constructor

Methods	Description	Defining Class
rebuild	Rebuilds the input objects using the history.	ltpda_uoh
matrix	Constructor for matrix class.	matrix

Converter

Methods	Description	Defining Class
double	– converts a matrix of objects into matrix of numbers	matrix

Helper

Methods	Description	Defining Class
search	Selects objects inside the collection/matrix object that match the given name.	ltpda_container
get	Get a property of a object.	ltpda_obj
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
getObjectAtIndex	Index into the inner objects of one matrix object.	matrix
osize	Returns the size of the inner object array.	matrix
setObjs	Sets the 'objs' property of a matrix object.	matrix

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	matrix
plot	The matrix objects on the given axes.	matrix

Operator

Methods	Description	Defining Class
conj	Implements conj operator for ltpda_container objects.	ltpda_container
det	Evaluates the determinant for matrix object.	matrix
fftfilt	Fft filter for matrix objects	matrix
inv	Evaluates the inverse for matrix object.	matrix
linearize	Output the derivatives of the model relative to the parameters.	matrix
lscov	Is a wrapper for MATLAB's lscov function.	matrix
rotate	Applies rotation factor to matrix objects	matrix
simplify	Each model in the matrix.	matrix
spsdSubtraction	Makes a sPSD−weighted least−square iterative fit	matrix

⬆ Back to Top of Section

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot−view or a MATLAB figure.	ltpda_uoh
char	Convert a matrix object into a string.	matrix
display	Overloads display functionality for matrix objects.	matrix
iplot	Calls ao/iplot on all inner ao objects.	matrix
unpack	Unpacks the objects in a matrix and sets them to the given output	matrix

⬆ Back to Top of Section

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

⬆ Back to Top of Section

Signal Processing

Methods	Description	Defining Class
abs	Overloads the Absolute value method for analysis objects in a ltpda_container object.	ltpda_container
consolidate	Resamples all input AOs in a ltpda_container object onto the same time grid.	ltpda_container
detrend	Detrends the analysis objects in a ltpda_container object using a polynomial of degree N.	ltpda_container
diff	Differentiates the data in a ltpda_container object.	ltpda_container
downsample	Downsamples each time−series AO in the ltpda_container.	ltpda_container
dsmean	Resamples each time−series AO in the ltpda_container.	ltpda_container
fft	Implements the fft operator for ltpda_container objects.	ltpda_container
fixfs	Adjusts the sample frequency of each time−series AO in the ltpda_container.	ltpda_container
heterodyne	Heterodynes time−series in a ltpda_container object.	ltpda_container
interp	Interpolate the values of each AO in the ltpda_container at new values.	ltpda_container
interpmissing	Interpolate missing samples of each time−series AO in the ltpda_container.	ltpda_container
lpsd	Computes the log−scale PSD of the time−series AOs in a ltpda_container object.	ltpda_container
polyfit	Overloads polyfit() function of MATLAB for ltpda_container objects.	ltpda_container
psd	Computes the PSD of the time−series in a ltpda_container object.	ltpda_container
removeVal	Removes values from each AO in the ltpda_container.	ltpda_container
resample	Resamples each time−series AO in the ltpda_container.	ltpda_container
split	Splits a ltpda_container object into the specified segments.	ltpda_container
sqrt	Computes the sqrt of each object in the ltpda_container.	ltpda_container
timeaverage	Averages time series intervals in a ltpda_container object.	ltpda_container
uminus	Overloads the uminus operator for all AOs in the ltpda_container.	ltpda_container
cohere	Estimates the coherence between elements of the vector.	matrix
cpsd	Estimates the cross−spectral density between elements of the vector.	matrix
crb	Computes the inverse of the Fisher Matrix	matrix
dispersion	Computes the dispersion function	matrix
dispersionLoop	Computes the dispersion function	matrix
lcohere	Estimates the coherence between elements of the vector using	matrix

[lcpsd](#)	Estimates the coherence between elements of the vector using a	matrix
[lincom](#)	Make a linear combination of analysis objects	matrix
[linfitsvd](#)	Linear fit with singular value decomposition	matrix
[linlsqsvd](#)	Linear least squares with singular value decomposition	matrix
[loglikelihood](#)	LOGLIKELIHOOD: Compute log-likelihood for MATRIX objects	matrix
[ltfe](#)	Estimates the transfer function between elements of the vector using	matrix
[mchNoisegen](#)	Generates multichannel noise data series given a model	matrix
[mchNoisegenFilter](#)	Construct a matrix filter from cross-spectral density matrix	matrix
[mcmc](#)	Estimates paramters using a Monte Carlo Markov Chain.	matrix
[modelSelect](#)	%%	matrix
[tdfit](#)	Fit a MATRIX of transfer function SMODELs to a matrix of input and output signals.	matrix
[tfe](#)	Estimates the transfer functions between elements of the vector.	matrix

mfir Class

[Properties](#)	Properties of the class
[Methods](#)	All Methods of the class ordered by category.
[Examples](#)	Some constructor examples

🔺 Back to Class descriptions

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defining Class
[gd](#)		mfir
[fs](#)	sample rate that the filter is designed for	ltpda_filter
[infile](#)	filename if the filter was loaded from file	ltpda_filter
[a](#)	set of numerator coefficients	ltpda_filter
[histout](#)	output history values of the filter	ltpda_filter
[iunits](#)	input units of the object	ltpda_tf
[ounits](#)	output units of the object	ltpda_tf
[hist](#)	history of the object (history object)	ltpda_uoh
[procinfo](#)	plist with additional information for an object.	ltpda_uoh
[plotinfo](#)	plotinfo object	ltpda_uoh
[name](#)	name of the object	ltpda_uo
[description](#)	description of the object	ltpda_uo
[UUID](#)	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

[Constructor](#)	Constructor of this class
[Helper](#)	Helper methods only for internal usage

[Internal](#)	Internal methods only for internal usage
[Output](#)	Output methods
[Relational Operator](#)	Relational operator methods
[Signal Processing](#)	Signal processing methods

[▲ Back to Top](#)

Constructor

Methods	Description	Defining Class
rebuild	Rebuilds the input objects using the history.	ltpda_uoh
mfir	FIR filter object class constructor.	mfir

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defining Class
setA	Set the property 'a'	ltpda_filter
setHistout	Sets the 'histout' property of the filter object.	ltpda_filter
get	Get a property of a object.	ltpda_obj
setIunits	Sets the 'iunits' property a transfer function object.	ltpda_tf
setOunits	Sets the 'ounits' property a transfer function object.	ltpda_tf
simplifyUnits	Simplify the input units and/or output units of the object.	ltpda_tf
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's	ltpda_uoh

	plotinfo.	
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
redesign	Redesign the input filter to work for the given sample rate.	mfir
setGd	Set the property 'gd'	mfir

 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
plot	The transfer function objects on the given axes.	ltpda_tf
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh

[generateConstructorPlist](#)	Generates a PLIST from the properties which can rebuild the object.	mfir

Output

Methods	Description	Defining Class
conj	Overloads conjugate functionality for ltpda_filter objects.	ltpda_filter
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot–view or a MATLAB figure.	ltpda_uoh
char	Convert a mfir object into a string.	mfir
display	Overloads display functionality for mfir objects.	mfir

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

Signal Processing

Methods	Description	Defining Class
impresp	Make an impulse response of the filter.	ltpda_filter
resp	Returns the complex response of a transfer function as an Analysis Object.	ltpda_tf

◀ matrix 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	Defining Class
b	set of denominator coefficients	miir
histin	input history values to filter	miir
fs	sample rate that the filter is designed for	ltpda_filter
infile	filename if the filter was loaded from file	ltpda_filter
a	set of numerator coefficients	ltpda_filter
histout	output history values of the filter	ltpda_filter
iunits	input units of the object	ltpda_tf
ounits	output units of the object	ltpda_tf
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

▲ Back to Top

Methods

Constructor	Constructor of this class

[Helper](#)	Helper methods only for internal usage
[Internal](#)	Internal methods only for internal usage
[Output](#)	Output methods
[Relational Operator](#)	Relational operator methods
[Signal Processing](#)	Signal processing methods

🔺 Back to Top

Constructor

Methods	Description	Defining Class
[rebuild](#)	Rebuilds the input objects using the history.	ltpda_uoh
[miir](#)	IIR filter object class constructor.	miir

🔺 Back to Top of Section

Helper

Methods	Description	Defining Class
[setA](#)	Set the property 'a'	ltpda_filter
[setHistout](#)	Sets the 'histout' property of the filter object.	ltpda_filter
[get](#)	Get a property of a object.	ltpda_obj
[setIunits](#)	Sets the 'iunits' property a transfer function object.	ltpda_tf
[setOunits](#)	Sets the 'ounits' property a transfer function object.	ltpda_tf
[simplifyUnits](#)	Simplify the input units and/or output units of the object.	ltpda_tf
[search](#)	Select objects that match the given name.	ltpda_uo
[setDescription](#)	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
[setName](#)	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
[created](#)	Returns a time object of the last modification.	ltpda_uoh
[creator](#)	Extract the creator(s) from the history.	ltpda_uoh
[csvexport](#)	Exports the data of an object to a csv file.	ltpda_uoh
[index](#)	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
[requirements](#)	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh

setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
redesign	Redesign the input filter to work for the given sample rate.	miir
setB	Set the property 'b'	miir
setHistin	Set the property 'histin'	miir

 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
plot	The transfer function objects on the given axes.	ltpda_tf
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh

type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	miir

🔺 [Back to Top of Section](#)

Output

Methods	Description	Defining Class
conj	Overloads conjugate functionality for ltpda_filter objects.	ltpda_filter
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	Convert a miir object into a string.	miir
display	Overloads display functionality for miir objects.	miir

🔺 [Back to Top of Section](#)

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

🔺 [Back to Top of Section](#)

Signal Processing

Methods	Description	Defining Class
impresp	Make an impulse response of the filter.	ltpda_filter
resp	Returns the complex response of a transfer function as an Analysis Object.	ltpda_tf

🔺 [Back to Top of Section](#)

◀ mfir 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	Defining Class
res	residuals [R]	parfrac
poles	poles (real or complex numbers) [P]	parfrac
pmul	Represents the pole multiplicity	parfrac
dir	direct terms [K]	parfrac
iunits	input units of the object	ltpda_tf
ounits	output units of the object	ltpda_tf
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

Constructor	Constructor of this class
Helper	Helper methods only for internal usage

[Internal](#)	Internal methods only for internal usage	
[Output](#)	Output methods	
[Relational Operator](#)	Relational operator methods	
[Signal Processing](#)	Signal processing methods	

[▲ Back to Top](#)

Constructor

Methods	Description	Defining Class
rebuild	Rebuilds the input objects using the history.	ltpda_uoh
parfrac	Partial fraction representation of a transfer function.	parfrac

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defining Class
get	Get a property of a object.	ltpda_obj
setIunits	Sets the 'iunits' property a transfer function object.	ltpda_tf
setOunits	Sets the 'ounits' property a transfer function object.	ltpda_tf
simplifyUnits	Simplify the input units and/or output units of the object.	ltpda_tf
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the	ltpda_uoh

	object's plotinfo.	
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh

▲ Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
plot	The transfer function objects on the given axes.	ltpda_tf
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	parfrac
getlowerFreq	Gets the frequency of the lowest pole in the model.	parfrac
getupperFreq	Gets the frequency of the highest pole in the model.	parfrac

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	Convert a parfrac object into a string.	parfrac
display	Overloads display functionality for parfrac objects.	parfrac

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

Signal Processing

Methods	Description	Defining Class
resp	Returns the complex response of a transfer function as an Analysis Object.	ltpda_tf

©LTP Team

pest 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	Defining Class
[dy](#)	standard errors of the parameters.	pest
[y](#)	input pest objects as – input pest objects array OUTPUTS: val – matrix with 'y', one column for each input object [Parameters Description](#) %%%	pest
[names](#)	names of the parameters, if any	pest
[yunits](#)	the units of each parameter	pest
[pdf](#)	posterior probability distribution of the parameters	pest
[cov](#)	covariance matrix of the parameters	pest
[corr](#)	correlation matrix of the parameters	pest
[chi2](#)	reduced chi^2 of the final fit	pest
[dof](#)	degrees of freedom	pest
[chain](#)	monte carlo markov chain	pest
[models](#)	models that were fit	pest
[hist](#)	history of the object (history object)	ltpda_uoh
[procinfo](#)	plist with additional information for an object.	ltpda_uoh
[plotinfo](#)	plotinfo object	ltpda_uoh
[name](#)	name of the object	ltpda_uo
[description](#)	description of the object	ltpda_uo
[UUID](#)	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 [Back to Top](#)

Methods

[Constructor](#)	Constructor of this class
[Converter](#)	Convertor methods
[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	Defining Class
rebuild	Rebuilds the input objects using the history.	ltpda_uoh
pest	Constructor for parameter estimates (pest) class.	pest

🔺 Back to Top of Section

Converter

Methods	Description	Defining Class
double	Overloads double() function for pest objects.	pest
y	Get the data property 'y'.	pest

🔺 Back to Top of Section

Helper

Methods	Description	Defining Class
get	Get a property of a object.	ltpda_obj
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
combineExps	Combine the results of different parameter estimation	pest
computePdf	Computes Probability Density Function from a pest object	pest
find	Creates analysis objects from the selected parameter(s).	pest
mcmcPlot	MCMCPLOT.M – Tool to plot results of a MCMC sampling.	pest
setChain	SETCHI2 Set the property 'chain'	pest
setChi2	Set the property 'chi2'	pest
setCorr	Set the property 'corr'	pest
setCov	Set the property 'cov'	pest
setDof	Set the property 'dof'	pest
setDy	Set the property 'dy'	pest
setDyForParameter	Sets the according dy–error for the specified parameter.	pest
setModels	Set the property 'models'	pest
setNames	Set the property 'names'	pest
setPdf	Set the property 'pdf'	pest
setXvals	Sets the 'xvals' property of the underlying smodel object.	pest
setY	Set the property 'y'	pest

setYforParameter	Sets the according y−value for the specified parameter.	pest
setYunits	Set the property 'yunits'	pest
setYunitsForParameter	Sets the according y−unit for the specified parameter.	pest
tdChi2	Computes the chi−square for a parameter estimate.	pest
viewResults	Displays the content of the pest object as an html report.	pest

🔺 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	pest
plot	The pest objects on the given axes.	pest

🔺 Back to Top of Section

Operator

Methods	Description	Defining Class
mve	MVE: Minimum Volume Ellipsoid estimator	pest

🔺 Back to Top of Section

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot−view or a MATLAB figure.	ltpda_uoh
char	Convert a pest object into a string.	pest
display	Overloads display functionality for pest objects.	pest
jtable	Display the parameters from PEST objects in a java table.	pest
table	Display the parameters from PEST objects in a java table.	pest

🔺 Back to Top of Section

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

🔺 Back to Top of Section

Signal Processing

Methods	Description	Defining Class
eval	Evaluate a pest object	pest

🔺 Back to Top of Section

◀ parfrac Class   plist Class ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ← →

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	Defining Class
[params](#)	list of param-objects	plist
[readonly](#)	flag to indicate that the plist is locked (or not)	plist
[keys](#)	-- No description found --	plist
[defaultKeys](#)	-- No description found --	plist
[name](#)	name of the object	ltpda_uo
[description](#)	description of the object	ltpda_uo
[UUID](#)	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 [Back to Top](#)

Methods

[Constructor](#)	Constructor of this class
[Helper](#)	Helper methods only for internal usage
[Internal](#)	Internal methods only for internal usage
[Output](#)	Output methods
[Relational Operator](#)	Relational operator methods

Constructor

Methods	Description	Defining Class
plist	Plist class object constructor.	plist

Helper

Methods	Description	Defining Class
get	Get a property of a object.	ltpda_obj
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
addAlternativeKeys	Adds some alternative key names to an existing key.	plist
append	Append a param-object, plist-object or a key/value pair to the parameter list.	plist
combine	Multiple parameter lists (plist objects) into a single plist.	plist
find	Overloads find routine for a parameter list.	plist
getAllKeys	Return all keys (even the alternative key names) of the parameter list.	plist
getIndexForKey	Returns the index of a parameter with the given key.	plist
getKeys	Return all the default keys of the parameter list.	plist
getOptionsForParam	Returns the options for the specified parameter key.	plist
getParamValueForParam	Returns the paramValue for the specified parameter key.	plist
getSelectionForParam	Returns the selection mode for the specified parameter key.	plist
isparam	Look for a given key in the parameter lists.	plist
merge	The values for the same key of multiple parameter lists together.	plist
mfind	Multiple-arguments find routine for a parameter list.	plist
nparams	Returns the number of param objects in the list.	plist
propertiesForParam	Returns the properties structure for a given parameter.	plist
propertyForParam	Returns the value of the specified	plist

	property for a given parameter.	
pset	Set or add a key/value pair or a param-object into the parameter list.	plist
regexp	Performs a regular expression search on the input plists.	plist
remove	Remove a parameter from the parameter list.	plist
removeKeys	Removes keys from a PLIST.	plist
search	Returns a subset of a parameter list.	plist
setDefaultForParam	Sets the default value of the param object in dependencies of the 'key'	plist
setDescriptionForParam	Sets the property 'desc' of the param object in dependencies of the 'key'	plist
setOptionsForParam	Sets the options of the param object in dependencies of the 'key'	plist
setSelectionForParam	Sets the selection mode of the param object in dependencies of the 'key'	plist
string	Converts a plist object to a command string which will recreate the plist object.	plist
subset	Returns a subset of a parameter list.	plist
type	Converts the input plist to MATLAB functions.	plist

▲ Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
getDescriptionForParam	Returns the description for the specified parameter key.	plist
getSetRandState	Gets or sets the random state of the MATLAB functions 'rand' and 'randn'	plist
matchKey	Returns a logical array with the same size of the parametes with a 1 if the input key matches to the key name(s) and a 0 if not.	plist
matchKeyWithRegexp	Returns a logical array with the same size of the parametes with a 1 if the input string matches to the key name(s) and a 0 if not.	plist

[matchKeys](#)	Returns a logical array with the same size of the parametes with a 1 if one of the input key(s) matches to the key name(s) and a 0 if not.	plist
[parse](#)	A plist for strings which can be converted into numbers	plist
[plist2cmds](#)	Convert a plist to a set of commands.	plist
[shouldIgnore](#)	True for plists which have the key 'ignore' with the value true.	plist
[simplify](#)	Simplifies a plist.	plist

▲ Back to Top of Section

Output

Methods	Description	Defining Class
[save](#)	Overloads save operator for ltpda objects.	ltpda_uo
[char](#)	Convert a parameter list into a string.	plist
[display](#)	Display plist object.	plist

▲ Back to Top of Section

Relational Operator

Methods	Description	Defining Class
[isequal](#)	Overloads the isequal operator for ltpda objects.	ltpda_obj

▲ Back to Top of Section

◀ pest Class   pzmodel Class ▶

©LTP Team

pzmodel Class

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

▲ Back to Class descriptions

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:
obj2 = obj1.setName('my name')  % This command creates a copy of obj1 (obj1 ~= obj2)
obj.setName('my name');         % This command applies to obj
```

Properties	Description	Defining Class
poles	pole-array of the model	pzmodel
zeros	zero-array of the model	pzmodel
gain	gain of the model	pzmodel
delay	delay of the pole/zero model	pzmodel
iunits	input units of the object	ltpda_tf
ounits	output units of the object	ltpda_tf
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

▲ Back to Top

Methods

Arithmetic Operator	Arithmetic Operator
Constructor	Constructor of this class

Helper	Helper methods only for internal usage
Internal	Internal methods only for internal usage
Operator	Operator methods
Output	Output methods
Relational Operator	Relational operator methods
Signal Processing	Signal processing methods

Back to Top

Arithmetic Operator

Methods	Description	Defining Class
mrdivide	Overloads the division operator for pzmodels.	pzmodel
mtimes	Overloads the multiplication operator for pzmodels.	pzmodel
rdivide	Overloads the division operator for pzmodels.	pzmodel
times	Overloads the multiplication operator for pzmodels.	pzmodel

Back to Top of Section

Constructor

Methods	Description	Defining Class
rebuild	Rebuilds the input objects using the history.	ltpda_uoh
pzmodel	Constructor for pzmodel class.	pzmodel

Back to Top of Section

Helper

Methods	Description	Defining Class
get	Get a property of a object.	ltpda_obj
setIunits	Sets the 'iunits' property a transfer function object.	ltpda_tf
setOunits	Sets the 'ounits' property a transfer function object.	ltpda_tf
simplifyUnits	Simplify the input units and/or output units of the object.	ltpda_tf
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last	ltpda_uoh

	modification.	
[creator](#)	Extract the creator(s) from the history.	ltpda_uoh
[csvexport](#)	Exports the data of an object to a csv file.	ltpda_uoh
[index](#)	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
[requirements](#)	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
[setPlotAxes](#)	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
[setPlotColor](#)	Sets the color of a the object's plotinfo.	ltpda_uoh
[setPlotFigure](#)	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
[setPlotFillmarker](#)	Defines if the plot function fill the marker or not.	ltpda_uoh
[setPlotLineStyle](#)	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
[setPlotLinewidth](#)	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
[setPlotMarker](#)	Sets the marker of a the object's plotinfo.	ltpda_uoh
[setPlotMarkerEdgeColor](#)	Sets the color of a the object's marker edge.	ltpda_uoh
[setPlotMarkerFaceColor](#)	Sets the color of a the object's marker face.	ltpda_uoh
[setPlotMarkersize](#)	Sets the markersize of a the object's plotinfo.	ltpda_uoh
[setPlotinfo](#)	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
[setPlottingStyle](#)	Sets the style property of a the object's plotinfo.	ltpda_uoh
[setProcinfo](#)	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
[setShowsErrors](#)	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
[setDelay](#)	Sets the 'delay' property of the pzmodel object.	pzmodel
[setGain](#)	Sets the 'gain' property of the pzmodel object.	pzmodel
[setPoles](#)	Set the property 'poles' of a pole/zero model.	pzmodel
[setZeros](#)	Set the property 'zeros' of a pole/zero model.	pzmodel

 [Back to Top of Section](#)

Internal

Methods	Description	Defining

		Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
plot	The transfer function objects on the given axes.	ltpda_tf
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	pzmodel
getlowerFreq	Gets the frequency of the lowest pole or zero in the model.	pzmodel
getupperFreq	Gets the frequency of the highest pole or zero in the model.	pzmodel

 Back to Top of Section

Operator

Methods	Description	Defining Class
tomfir	Approximates a pole/zero model with an FIR filter.	pzmodel
tomiir	Converts a pzmodel to an IIR filter using a bilinear transform.	pzmodel

 Back to Top of Section

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	Convert a pzmodel object into a string.	pzmodel
conj	Overloads conjugate functionality for pzmodel objects.	pzmodel
display	Overloads display functionality for pzmodel objects.	pzmodel

 Back to Top of Section

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

🔺 Back to Top of Section

Signal Processing

Methods	Description	Defining Class
resp	Returns the complex response of a transfer function as an Analysis Object.	ltpda_tf
fngen	Creates an arbitrarily long time-series based on the input pzmodel.	pzmodel
simplify	Simplifies pzmodels by cancelling like poles with like zeros.	pzmodel

🔺 Back to Top of Section

◀ plist Class   rational Class ▶

©LTP Team

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	Defining Class
num	numerator coefficients [a]	rational
den	denominator coefficients [b]	rational
iunits	input units of the object	ltpda_tf
ounits	output units of the object	ltpda_tf
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

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

[Relational Operator](#)	Relational operator methods	
[Signal Processing](#)	Signal processing methods	

🔺 [Back to Top](#)

Constructor

Methods	Description	Defining Class
[rebuild](#)	Rebuilds the input objects using the history.	ltpda_uoh
[rational](#)	Rational representation of a transfer function.	rational

🔺 [Back to Top of Section](#)

Helper

Methods	Description	Defining Class
[get](#)	Get a property of a object.	ltpda_obj
[setIunits](#)	Sets the 'iunits' property a transfer function object.	ltpda_tf
[setOunits](#)	Sets the 'ounits' property a transfer function object.	ltpda_tf
[simplifyUnits](#)	Simplify the input units and/or output units of the object.	ltpda_tf
[search](#)	Select objects that match the given name.	ltpda_uo
[setDescription](#)	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
[setName](#)	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
[created](#)	Returns a time object of the last modification.	ltpda_uoh
[creator](#)	Extract the creator(s) from the history.	ltpda_uoh
[csvexport](#)	Exports the data of an object to a csv file.	ltpda_uoh
[index](#)	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
[requirements](#)	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
[setPlotAxes](#)	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
[setPlotColor](#)	Sets the color of a the object's plotinfo.	ltpda_uoh
[setPlotFigure](#)	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
[setPlotFillmarker](#)	Defines if the plot function fill the marker or not.	ltpda_uoh

setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh

🔺 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
plot	The transfer function objects on the given axes.	ltpda_tf
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	rational
getlowerFreq	Gets the frequency of the lowest pole in the model.	rational
getupperFreq	Gets the frequency of the highest pole in the model.	rational

🔺 Back to Top of Section

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot–view or a MATLAB figure.	ltpda_uoh
char	Convert a rational object into a string.	rational
display	Overloads display functionality for rational objects.	rational

▲ Back to Top of Section

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

▲ Back to Top of Section

Signal Processing

Methods	Description	Defining Class
resp	Returns the complex response of a transfer function as an Analysis Object.	ltpda_tf

▲ Back to Top of Section

◀ pzmodel Class | smodel Class ▶

©LTP Team

smodel Class

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

🔺 Back to Class descriptions

Properties

The LTPDA toolbox restrict access of the properties.
The get access is 'public' and thus it is possible to get the values with the dot-command (similar to structures).

```
For example:
val = obj.prop(2).prop;
```

The set access is 'protected' and thus it is only possible to assign a value to a property with a set-method.

```
For example:
obj2 = obj1.setName('my name') % This command creates a copy of obj1 (obj1 ~= obj2)
obj.setName('my name');        % This command applies to obj
```

Properties	Description	Defining Class
expr	Expression of the model	smodel
params	Parameters which are used in the model	smodel
values	Default values for the parameters	smodel
trans	Transformation strings mapping xvals in terms of xvar to X in the model	smodel
aliasNames	{'v', 'H'};	smodel
aliasValues	{'a*b', [1:20]};	smodel
xvar	Cell-array with x-variable(s)	smodel
xvals	Cell-array of double-values for the different x-variable(s)	smodel
xunits	vector of units of the different x-axis	smodel
yunits	units of the y-axis	smodel
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

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

▲ Back to Top

Arithmetic Operator

Methods	Description	Defining Class
conj	Gives the complex conjugate of the input smodels	smodel
minus	Implements subtraction operator for smodel objects.	smodel
mrdivide	Implements mrdivide operator for smodel objects.	smodel
mtimes	Implements mtimes operator for smodel objects.	smodel
plus	Implements addition operator for smodel objects.	smodel
rdivide	Implements division operator for smodel objects.	smodel
times	Implements multiplication operator for smodel objects.	smodel

▲ Back to Top of Section

Constructor

Methods	Description	Defining Class
rebuild	Rebuilds the input objects using the history.	ltpda_uoh
smodel	Constructor for smodel class.	smodel

▲ Back to Top of Section

Helper

Methods	Description	Defining Class
get	Get a property of a object.	ltpda_obj
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo

[setName](#)	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
[created](#)	Returns a time object of the last modification.	ltpda_uoh
[creator](#)	Extract the creator(s) from the history.	ltpda_uoh
[csvexport](#)	Exports the data of an object to a csv file.	ltpda_uoh
[index](#)	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
[plot](#)	Plots the user object on a figure.	ltpda_uoh
[requirements](#)	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
[setPlotAxes](#)	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
[setPlotColor](#)	Sets the color of a the object's plotinfo.	ltpda_uoh
[setPlotFigure](#)	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
[setPlotFillmarker](#)	Defines if the plot function fill the marker or not.	ltpda_uoh
[setPlotLineStyle](#)	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
[setPlotLinewidth](#)	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
[setPlotMarker](#)	Sets the marker of a the object's plotinfo.	ltpda_uoh
[setPlotMarkerEdgeColor](#)	Sets the color of a the object's marker edge.	ltpda_uoh
[setPlotMarkerFaceColor](#)	Sets the color of a the object's marker face.	ltpda_uoh
[setPlotMarkersize](#)	Sets the markersize of a the object's plotinfo.	ltpda_uoh
[setPlotinfo](#)	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
[setPlottingStyle](#)	Sets the style property of a the object's plotinfo.	ltpda_uoh
[setProcinfo](#)	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
[setShowsErrors](#)	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
[addAliases](#)	Add the key-value pairs to the alias-names and alias-values	smodel
[addParameters](#)	Add some parameters to the symbolic model (smodel) object	smodel
[assignalias](#)	Assign values to smodel alias	smodel
[clearAliases](#)	Clear the aliases.	smodel
[double](#)	Returns the numeric result of the model.	smodel
[eval](#)	Evaluates the symbolic model and returns an AO containing the numeric	smodel

	data.	
fitfunc	Returns a function handle which sets the 'values' and 'xvals' to a ltpda model.	smodel
hessian	Compute the hessian matrix for a symbolic model.	smodel
op	Add a operation around the model expression	smodel
setAliasNames	Set the property 'aliasNames'	smodel
setAliasValues	Set the property 'aliasValues'	smodel
setAliases	Set the key-value pairs to the alias-names and alias-values	smodel
setParameters	Set some parameters to the symbolic model (smodel) object	smodel
setParams	Set the property 'params' AND 'values'	smodel
setTrans	Sets the 'trans' property of the smodel object.	smodel
setValues	Set the property 'values'	smodel
setXunits	Sets the 'xunits' property of the smodel object.	smodel
setXvals	Sets the 'xvals' property of the smodel object.	smodel
setXvar	Sets the 'xvar' property of the smodel object.	smodel
setYunits	Sets the 'yunits' property of the smodel object.	smodel
simplifyUnits	Simplify the x and/or y units of the model.	smodel
subs	Substitutes symbolic parameters with the given values.	smodel

🔺 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties	smodel

	which can rebuild the object.

[▲ Back to Top of Section](#)

Operator

Methods	Description	Defining Class
convol_integral	Implements the convolution integral for smodel objects.	smodel
det	Evaluates the determinant of smodel objects.	smodel
diff	Implements differentiation operator for smodel objects.	smodel
fourier	Implements continuous f–domain Fourier transform for smodel objects.	smodel
ifourier	Implements continuous f–domain inverse Fourier transform for smodel objects.	smodel
ilaplace	Implements continuous s–domain inverse Laplace transform for smodel objects.	smodel
inv	Evaluates the inverse of smodel objects.	smodel
iztrans	Implements continuous z–domain inverse Z–transform for smodel objects.	smodel
laplace	Implements continuous s–domain Laplace transform for smodel objects.	smodel
linearize	Output the derivatives of the model relative to the parameters.	smodel
simplify	Implements simplify operator for smodel objects.	smodel
sum	Adds all the elements of smodel objects arrays.	smodel
ztrans	Implements continuous z–domain Z–transform for smodel objects.	smodel

[▲ Back to Top of Section](#)

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot–view or a MATLAB figure.	ltpda_uoh
char	Convert a smodel object into a string.	smodel
display	Overloads display functionality for smodel objects.	smodel

[▲ Back to Top of Section](#)

Relational Operator

Methods	Description	Defining Class

[isequal](#)	Overloads the isequal operator for ltpda objects.	ltpda_obj

🔺 [Back to Top of Section](#)

◀ rational Class   ssm Class ▶

©LTP Team

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	Defining Class
[amats](#)	A matrix representing a difference/differential term in the state equation, block stored in a cell array	ssm
[bmats](#)	B matrix representing an input coefficient matrix in the state equation, block stored in a cell array	ssm
[cmats](#)	C matrix representing the state projection in the observation equation, block stored in a cell array	ssm
[dmats](#)	D matrix representing the direct feed through term in the observation equation, block stored in a cell array	ssm
[timestep](#)	Timestep of the difference equation. Zero means the representation is time continuous and A defines a differential equation.	ssm
[inputs](#)	ssmblock for input blocks	ssm
[states](#)	ssmblock describing state blocks	ssm
[outputs](#)	ssmblock describing the output blocks	ssm
[numparams](#)	nested plist describing the numeric (substituted) parameters	ssm
[params](#)	nested plist describing the symbolic parameters	ssm
[hist](#)	history of the object (history object)	ltpda_uoh
[procinfo](#)	plist with additional information for an object.	ltpda_uoh
[plotinfo](#)	plotinfo object	ltpda_uoh

[name](#)	name of the object	ltpda_uo
[description](#)	description of the object	ltpda_uo
[UUID](#)	Universally Unique Identifier of 128-bit	ltpda_uo

[▲ Back to Top](#)

Methods

[Constructor](#)	Constructor of this class	
[Converter](#)	Convertor methods	
[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	Defining Class
[rebuild](#)	Rebuilds the input objects using the history.	ltpda_uoh
[ssm](#)	Statespace model class constructor.	ssm

[▲ Back to Top of Section](#)

Converter

Methods	Description	Defining Class
[ssm2miir](#)	Converts a statespace model object to a miir object	ssm
[ssm2pzmodel](#)	Converts a time-continuous statespace model object to a pzmodel	ssm
[ssm2rational](#)	Converts a statespace model object to a rational frac. object	ssm
[ssm2ss](#)	Converts a statespace model object to a MATLAB statespace object.	ssm

[▲ Back to Top of Section](#)

Helper

Methods	Description	Defining

		Class
get	Get a property of a object.	ltpda_obj
search	Select objects that match the given name.	ltpda_uo
setDescription	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
setName	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
created	Returns a time object of the last modification.	ltpda_uoh
creator	Extract the creator(s) from the history.	ltpda_uoh
csvexport	Exports the data of an object to a csv file.	ltpda_uoh
index	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
plot	Plots the user object on a figure.	ltpda_uoh
requirements	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
setPlotAxes	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
setPlotColor	Sets the color of a the object's plotinfo.	ltpda_uoh
setPlotFigure	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
setPlotFillmarker	Defines if the plot function fill the marker or not.	ltpda_uoh
setPlotLineStyle	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
setPlotLinewidth	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
setPlotMarker	Sets the marker of a the object's plotinfo.	ltpda_uoh
setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
addParameters	Adds the parameters to the model.	ssm
duplicateInput	Copies the specified input blocks.	ssm
parameterDiff	Makes a ssm that produces the output	ssm

	and state derivatives.	
setBlockProperties	Sets the specified properties of the specified SSM blocks.	ssm
setParameters	Sets the values of the given parameters.	ssm
setParams	Sets the parameters of the model to the given plist.	ssm
setPortProperties	Sets names of the specified SSM ports.	ssm

🔺 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
generateConstructorPlist	Generates a PLIST from the properties which can rebuild the object.	ssm
ssm2dot	Converts a statespace model object a DOT file.	ssm

🔺 Back to Top of Section

Operator

Methods	Description	Defining Class
append	Appends embedded subsytems, with exogenous inputs	ssm
assemble	Assembles embedded subsytems, with exogenous inputs	ssm
cpsd	Computes the output theoretical CPSD shape with given inputs.	ssm
cpsdForCorrelatedInputs	Computes the output theoretical CPSD shape with given inputs.	ssm
cpsdForIndependentInputs	Computes the output theoretical CPSD shape with given inputs.	ssm
kalman	Applies Kalman filtering to a discrete ssm with given i/o	ssm
keepParameters	Enables to substitute symbollic	ssm

	patameters	
modifyTimeStep	Modifies the timestep of a ssm object	ssm
optimiseForFitting	Reduces the system matrices to doubles and strings.	ssm
psd	Computes the output theoretical PSD shape with given inputs.	ssm
removeEmptyBlocks	Enables to do model simplification	ssm
reorganize	REOGANIZE rearranges a ssm object for fast input to BODE, SIMULATE, PSD.	ssm
sMinReal	Gives a minimal realization of a ssm object by deleting unreached states	ssm
simplify	Enables to do model simplification	ssm
simulate	Simulates a discrete ssm with given inputs	ssm
steadyState	Returns a possible value for the steady state of an ssm.	ssm
subsParameters	Enables to substitute symbolic patameters	ssm

🔺 Back to Top of Section

Output

Methods	Description	Defining Class
save	Overloads save operator for ltpda objects.	ltpda_uo
report	Generates an HTML report about the input objects.	ltpda_uoh
viewHistory	Displays the history of an object as a dot-view or a MATLAB figure.	ltpda_uoh
char	Convert a ssm object into a string.	ssm
display	Display ssm object.	ssm
displayProperties	DISPAYPROPERTIES displays the ssm model porperties.	ssm
dotview	View an ssm object via the DOT interpreter.	ssm
double	Convert a statespace model object to double arrays for given i/o	ssm
findParameters	Returns parameter names matching the given pattern.	ssm
getParameters	Returns parameter values for the given names.	ssm
getParams	Returns the parameter list for this SSM model.	ssm
getPortNamesForBlocks	Returns a list of port names for the given block.	ssm
isStable	Tells if ssm is numerically stable	ssm

| settlingTime | Retunrns 1% the settling time of the system. | ssm |

Relational Operator

Methods	Description	Defining Class
isequal	Overloads the isequal operator for ltpda objects.	ltpda_obj

Signal Processing

Methods	Description	Defining Class
bode	Makes a bode plot from the given inputs to outputs.	ssm
bodecst	Makes a bodecst plot from the given inputs to outputs.	ssm
loglikelihood	LOGLIKELIHOOD: Compute log−likelihood for SSM objects	ssm
projectNoise	Performs actions on ao objects.	ssm
resp	Gives the timewise impulse response of a statespace model.	ssm
respcst	Gives the timewise impulse response of a statespace model.	ssm
viewDetails	Performs actions on ssm objects.	ssm

| ◀ smodel 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	Defining Class
startT	Start time of the time span. (time-object)	timespan
endT	End time of the time span. (time-object)	timespan
interval	Interval between start/end time	timespan
timeformat	timeformat of the time span (see preferences)	timespan
timezone	timezone of the time span (see preferences)	timespan
hist	history of the object (history object)	ltpda_uoh
procinfo	plist with additional information for an object.	ltpda_uoh
plotinfo	plotinfo object	ltpda_uoh
name	name of the object	ltpda_uo
description	description of the object	ltpda_uo
UUID	Universally Unique Identifier of 128-bit	ltpda_uo

🔺 Back to Top

Methods

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

[Output](#)	Output methods
[Relational Operator](#)	Relational operator methods

[Back to Top](#)

Constructor

Methods	Description	Defining Class
[rebuild](#)	Rebuilds the input objects using the history.	ltpda_uoh
[timespan](#)	Timespan object class constructor.	timespan

[Back to Top of Section](#)

Helper

Methods	Description	Defining Class
[get](#)	Get a property of a object.	ltpda_obj
[search](#)	Select objects that match the given name.	ltpda_uo
[setDescription](#)	Sets the 'description' property of a ltpda_uoh object.	ltpda_uo
[setName](#)	Sets the property 'name' of an ltpda_uoh object.	ltpda_uo
[created](#)	Returns a time object of the last modification.	ltpda_uoh
[creator](#)	Extract the creator(s) from the history.	ltpda_uoh
[csvexport](#)	Exports the data of an object to a csv file.	ltpda_uoh
[index](#)	Index into a 'ltpda_uoh' object array or matrix. This properly captures the history.	ltpda_uoh
[requirements](#)	Returns a list of LTPDA extension requirements for a given object.	ltpda_uoh
[setPlotAxes](#)	Sets the 'axes' property of a the object's plotinfo.	ltpda_uoh
[setPlotColor](#)	Sets the color of a the object's plotinfo.	ltpda_uoh
[setPlotFigure](#)	Sets the 'figure' property of a the object's plotinfo.	ltpda_uoh
[setPlotFillmarker](#)	Defines if the plot function fill the marker or not.	ltpda_uoh
[setPlotLineStyle](#)	Sets the linestyle of a the object's plotinfo.	ltpda_uoh
[setPlotLinewidth](#)	Sets the linewidth of a the object's plotinfo.	ltpda_uoh
[setPlotMarker](#)	Sets the marker of a the object's plotinfo.	ltpda_uoh

setPlotMarkerEdgeColor	Sets the color of a the object's marker edge.	ltpda_uoh
setPlotMarkerFaceColor	Sets the color of a the object's marker face.	ltpda_uoh
setPlotMarkersize	Sets the markersize of a the object's plotinfo.	ltpda_uoh
setPlotinfo	Sets the 'plotinfo' property of a ltpda_uoh object.	ltpda_uoh
setPlottingStyle	Sets the style property of a the object's plotinfo.	ltpda_uoh
setProcinfo	Sets the 'procinfo' property of a ltpda_uoh object.	ltpda_uoh
setShowsErrors	Sets the 'showErrors' property of a the object's plotinfo.	ltpda_uoh
getEndT	Get the timespan property 'endT'.	timespan
getNsecs	Get the timespan property 'nsecs'.	timespan
getStartT	Get the timespan property 'startT'.	timespan
inTimespan	Checks if an input time is inbetween a timespan.	timespan
merge	The input timespan objects into one output timespan object.	timespan
minus	Implements subtraction operator for timespan objects.	timespan
plus	Implements addition operator for timespan objects.	timespan
setEndT	Sets the 'endT' property of the timespan objects.	timespan
setStartT	Sets the 'startT' property of the timespan objects.	timespan

 Back to Top of Section

Internal

Methods	Description	Defining Class
isprop	Tests if the given field is one of the object properties.	ltpda_obj
bsubmit	Submits the given collection of objects in binary form to an LTPDA repository	ltpda_uo
submit	Submits the given collection of objects to an LTPDA repository	ltpda_uo
update	Updates the given object in an LTPDA repository	ltpda_uo
string	Writes a command string that can be used to recreate the input object(s).	ltpda_uoh
type	Converts the input objects to MATLAB functions.	ltpda_uoh
double	Overloads double() function for timespan objects.	timespan

[generateConstructorPlist](#)	Generates a PLIST from the properties which can rebuild the object.	timespan
[plot](#)	The timespan objects on the given axes.	timespan

🔺 [Back to Top of Section](#)

Output

Methods	Description	Defining Class
[save](#)	Overloads save operator for ltpda objects.	ltpda_uo
[report](#)	Generates an HTML report about the input objects.	ltpda_uoh
[viewHistory](#)	Displays the history of an object as a dot−view or a MATLAB figure.	ltpda_uoh
[char](#)	Convert a timespan object into a string.	timespan
[display](#)	Overloads display functionality for timespan objects.	timespan

🔺 [Back to Top of Section](#)

Relational Operator

Methods	Description	Defining Class
[isequal](#)	Overloads the isequal operator for ltpda objects.	ltpda_obj

🔺 [Back to Top of Section](#)

Constructor Examples

Constructor examples

[General Constructor examples](#)
[Constructor examples of the AO class](#)
[Constructor examples of the MFIR class](#)
[Constructor examples of the MIIR class](#)
[Constructor examples of the PZMODEL class](#)
[Constructor examples of the PARFRAC class](#)
[Constructor examples of the RATIONAL class](#)
[Constructor examples of the TIMESPAN class](#)
[Constructor examples of the PLIST class](#)
[Constructor examples of the SPECWIN class](#)

General constructor examples

[Copy a LTPDA object](#)

[Construct an empty LTPDA object](#)

[Construct an empty LTPDA object with the size zero](#)

[Construct a LTPDA object by loading the object from a file](#)

[Construct an LTPDA object from a parameter list object (PLIST)](#)

All here described constructors works for all LTPDAobjects.

Copy a LTPDA object

The following example creates a copy of a LTPDA object (blue command) by an example of the PARFRAC object. The copy constructor should work for all classes.

```
>> pf1 = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3)
---- parfrac 1 ----
model:        None
res:          [1;2+i*1;2-i*1]
poles:        [6;1+i*3;1-i*3]
dir:          3
pmul:         [1;1;1]
iunits:       []
ounits:       []
description:
UUID:         7c57ab84-3f49-486b-be49-6ed9926fbd66
------------------
>> pf2 = parfrac(pf1)
---- parfrac 1 ----
model:        None
res:          [1;2+i*1;2-i*1]
poles:        [6;1+i*3;1-i*3]
dir:          3
pmul:         [1;1;1]
iunits:       []
ounits:       []
description:
UUID:         c892f70e-d113-407c-accc-67f988b31e7e
------------------
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> pf1 = parfrac()
---- parfrac 1 ----
model:        none
res:          []
poles:        []
dir:          0
pmul:         []
iunits:       []
ounits:       []
description:
UUID:         bd7c9c71-e633-402b-964b-0a270f80764a
------------------
>> pf2 = pf1;
>> pf2.setName('my new name')
---- parfrac 1 ----
model:        my new name
res:          []
poles:        []
```

```
dir:            0
pmul:           []
iunits:         []
ounits:         []
description:
UUID:           47a5e458-84fd-4d6b-beb2-11ff574ae661
------------------
```

If we display pf1 again then we see that the property 'name' was changed although we only have changed pf2.

```
>> pf1
---- parfrac 1 ----
model:          my new name
res:            []
poles:          []
dir:            0
pmul:           []
iunits:         []
ounits:         []
description:
UUID:           47a5e458-84fd-4d6b-beb2-11ff574ae661
------------------
```

Construct an empty LTPDA object

The following example creates an empty LTPDA object by an example of a rational object.

```
>> rat = rational()
---- rational 1 ----
model:          none
num:            []
den:            []
iunits:         []
ounits:         []
description:
UUID:           eb240e62-ff2f-4c31-a683-0f395b9a5242
-------------------
```

Construct an empty LTPDA object with the size zero.

Sometimes it is necessary to create LTPDA objects which have at least one dimension of zero. This mean that the MATLAB method "isempty" is true for these objects. This constructor is shown by an example of a matrix object.

```
>> m = matrix.initObjectWithSize(1,0)
------ matrix -------
empty-object [1,0]
--------------------
```

It is also possible with this constructor to create a matrix of objects with a different input as zero.

```
>> m = matrix.initObjectWithSize(2,1);
---- matrix 1 ----
name: none
size: 0x0
description:
UUID: ed13da80-92a7-4d1f-8d78-a88f7d77cec3
-----------------
---- matrix 2 ----
name: none
size: 0x0
description:
UUID: ed13da80-92a7-4d1f-8d78-a88f7d77cec3
```

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

Construct a LTPDA object by loading the object from a file

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

```
        pf = ao'parfrac_object.mat')
        pf = ao'parfrac_object.xml')
```

Construct an LTPDA object from a parameter list object (PLIST)

Each constructor have different sets of PLISTS which create the object in a different way.

AO class	[Parameter Sets](#)
SSM class	[Parameter Sets](#)
SMODEL class	[Parameter Sets](#)
TIMESPAN class	[Parameter Sets](#)
MATRIX class	[Parameter Sets](#)
COLLECTION class	[Parameter Sets](#)
PZMODEL class	[Parameter Sets](#)
PARFRAC class	[Parameter Sets](#)
RATIONAL class	[Parameter Sets](#)
FILTERBANK class	[Parameter Sets](#)
MIIR class	[Parameter Sets](#)
MFIR class	[Parameter Sets](#)
PLIST class	[Parameter Sets](#)

Constructor examples of the AO class

[Copy an AO](#)
[Construct an AO by loading the AO from a file](#)
[Construct an AO from a data file](#)
[Construct an AO from spectral window](#)
[Construct an AO from a parameter set](#)

Copy an AO

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

```
>> a1 = ao([1:12]);
>> a2 = ao(1)
----------- ao: a -----------

name:  none
creator:  created by hewitson@bobmac.aei.uni-hannover.de[130.75.117.65] on MACI/7.6
description:
data:  None
hist:  ao / ao / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
mfilename:
mdlfilename:
-----------------------------
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> a1 = ao()
----------- ao 01: a1 -----------
      name:  none
description:
      data:  None
      hist:  ao / ao / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
   mfilename:
mdlfilename:
---------------------------------
>> a2 = a1;
>> a2.setName('my new name')
----------- ao 01: my new name -----------
      name:  my new name
description:
      data:  None
      hist:  ltpda_uoh / setName / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
   mfilename:
mdlfilename:
-----------------------------------------
```

If we display a1 again then we see that the property 'name' was changed although we only have changed a2.

```
>> a1
----------- ao 01: my new name -----------
      name:  my new name
description:
      data:  None
      hist:  ltpda_uoh / setName / $Id: ao.m,v 1.220 2009/02/25 18:51:24 ingo Exp
```

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

Construct an AO by loading the AO from a file

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

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

or in a PLIST

```
pl = plist('filename', 'a1.xml')
a  = ao(pl)
```

Construct an AO from a data file

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

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

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

Key	Description
'type'	'tsdata','fsdata','xydata' [default: 'tsdata']
'fs'	If this value is set, the x-axes is computed by the fs value. [default: empty]
'columns'	[1 2 1 4] Each pair represents the x- and y-axes (each column pair creates an analysis object). If the value 'fs' is set, then each column is converted to the y vector of a time-series AO. [default: [1 2]]
'comment_char'	The comment character in the file [default: '%']
'description'	To set the description in the analysis object
'...'	Every property of the data object e.g. 'name'

```
% Each pair in col represents the x- and y-axes.
% 'fs' is not used !!!

pl = plist('filename', 'data.dat',  ...
'description', 'my ao description',    ...
'type', 'xydata',    ...
'xunits', 's', ...
'yunits', {'Volt', 'Hz'}, ...
'columns', [1 2 1 3],     ...
```

```
       'comment_char', '//');

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

Another example where the time vector is specified by the sample rate (`fs`) and each column of data is converted in to a single AO.

```
% 'fs is used. As such, each column in col creates its own AO with the specified sample
rate.

pl = plist('filename', 'data.dat',...
'type', 'tsdata',       ...
'fs', 100,          ...
't0', {'14:00:00', '14:00:20', '14:00:30'},   ...
'columns', [1 2 3]);
out = ao('data.dat', pl);
out = ao(pl);
```

Construct an AO from a spectral window

The following example creates a cdata type AO containing the window values.

```
win = specwin('Kaiser', 100, 10);
>> a = ao(win)
---------- ao 01: Kaiser -----------
        name:  Kaiser
description:
        data: 0.7145 0.7249 0.7351 0.7452 0.7551 0.7649 0.7746 0.7840 0.7934 0.8025 ...
              -------- cdata 01 -----------
                  y:  [1x100], double
              yunits:  []
              ------------------------------
        hist:  ao / ao / $Id: fromSpecWin.m,v 1.11 2008/12/05 10:47:14 hewitson Exp -->
    mfilename:
mdlfilename:
----------------------------------
```

Construct an AO from a parameter sets

Constructs an analysis object from the description given in the parameter list (in order of priority).

From ASCII File

From complex ASCII File

From Values

From Function

From XY Function

From Time-series Function

From Frequency-series Function

From Window

From Waveform

From Repository

From Polynomial

From Pzmodel

From ASCII File

The following example creates an analysis object from a datafile. Click here to get more information about the parameters.

```
% Construct two analysis-objects with time-series data from
% time data in the first column and the data in column 2 and 3 of the file.
a = ao(plist('filename', 'data.txt', 'columns', [1 2 1 3], 'type', 'tsdata'));

% Construct two analysis-objects with the given frequency and the data in column 1 and
2.
a = ao(plist('filename', 'data.txt', 'fs', 12.1, 'columns', [1 2], 'type', 'tsdata'));

% Define a comment character for skip comments.
a = ao(plist('filename', 'data.txt', 'comment_char', '#', 'columns', [1 2 1 3], 'type',
'tsdata'));
```

From complex ASCII File

The following example creates an analysis object from a complex datafile. Click here to get more information about the parameters.

```
a = ao(plist('filename', 'data.txt', 'complex_type', 'real/imag', 'type', 'tsdata'));
a = ao(plist('filename', 'data.txt', 'complex_type', 'real/imag', 'type', 'fsdata',
'columns', [1,2,4]));
```

From Values

The following example creates an AO from a set of values. The data type depends on the parameters. If you use the parameter key **'vals'** then you will get an analysis object with cdata. Click here for information about the **value** parameter set or here to get more information about the **xy-value** parameter set.

```
% cdata
a = ao(plist('vals', [1 2 3], 'N', 10));

% xydata
a = ao(plist('xvals', [1 2 3], 'yvals', [10 20 30]));

% tsdata
a = ao(plist('xvals', [1 2 3], 'yvals', [10 20 30], 'type', 'tsdata'));
a = ao(plist('fs', 1, 'yvals', [10 20 30]));

% fsdata
a = ao(plist('xvals', [1 2 3], 'yvals', [10 20 30], 'type', 'fsdata'));
a = ao(plist('fs', 1, 'yvals', [10 20 30], 'type', 'fsdata'));
a = ao(plist('fs', 1, 'yvals', [10 20 30], 'type', 'fsdata', 'xunits', 'mHz', 'yunits',
'V'));
```

From Function

The following example creates an AO from the description of any valid MATLAB function. The data object is of type `cdata`. Click here to get more information about the parameters.

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

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

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

From XY Function

Construct an AO from a function f(x) string. The data object is from type `xydata`. Click [here](#) to get more information about the parameters.

```
a = ao(plist('xyfcn', 'cos(2*pi*x) + randn(size(x))', 'x', [1:1e5]));
a = ao(plist('xyfcn', 'log(x)', 'x', [1:50,52:2:100,110:10:1000]));
```

From Time-series Function

Construct an AO from a function of time, `t` f(t). The data object is from type `tsdata` (time-series data). Click [here](#) to get more information about the parameters.

```
a = ao(plist('fs',     10, ...
             'nsecs', 10, ...
             'tsfcn', 'sin(2*pi*1.4*t) + 0.1*randn(size(t))', ...
             't0',     '1980-12-01 12:43:12'));
a = ao(plist('tsfcn', 'cos(pi*t) + randn(size(t))', ...
             'fs',     1, ...
             'nsecs', 100));
a = ao(plist('fs',     10, ...
             'nsecs', 10, ...
             'tsfcn', 'sin(2*pi*1.4*t)+0.1*randn(size(t))', ...
             't0',     time('1980-12-01 12:43:12')));
```

From Frequency-series Function

Construct an AO from a function of frequency, f f(f). The data object is from type `fsdata` (frequency-series). Click [here](#) to get more information about the parameters.

```
a = ao(plist('fsfcn', 'f', 'f1', 1e-5, 'f2', 1, 'yunits', 'V'));
a = ao(plist('fsfcn', 'f', 'f', [0.01:0.01:1]));
a = ao(plist('fsfcn', '1./f.^2', 'scale', 'lin', 'nf', 100));
```

From Window

Construct an AO from a spectral window object.
Click [here](#) for a list of supported window functions and [here](#) to get more information about the parameters.

```
a = ao(plist('win', specwin('Hanning', 100)))
a = ao(plist('win', specwin('Kaiser', 10, 150)))
```

% >> % >> ao(plist('waveform','noise','type','normal','sigma',2,'nsecs',1000,'fs',1)); % >> % >> % >> % >>

From Waveform

Construct an AO from a waveform with the following waveform types. Click [here](#) to get more information about the parameters.

```
% Construct random noise
a = ao(plist('waveform', 'noise', 'type', 'Normal', 'sigma', 2, 'nsecs', 1000, 'fs', 1));
```

```
% Construct uniform random noise
a = ao(plist('waveform', 'noise', 'type', 'Uniform', 'nsecs', 1000, 'fs', 1));

% Construct a sine wave
a = ao(plist('waveform', 'sine wave', 'A', 3, 'f', 1, 'phi', pi/2, 'toff', 0.1, 'nsecs',
10, 'fs', 100));

% Construct a chirp waveform
a = ao(plist('waveform', 'chirp', 'f0', 0.1, 'f1', 1, 't1', 1, 'nsecs', 5, 'fs', 1000));

% Construct a Gaussian pulse waveform
a = ao(plist('waveform', 'gaussian pulse', 'f0', 1, 'bw', 0.2, 'nsecs', 20, 'fs', 10));

% Construct a Square wave
a = ao(plist('waveform', 'square wave', 'f', 2, 'duty', 40, 'nsecs', 10, 'fs', 100));

% Construct a Sawtooth wave
a = ao(plist('waveform', 'sawtooth', 'f', 1.23, 'width', 1, 'nsecs', 10/1.23, 'fs',
50));
```

From Repository

Construct an AO by retrieving it from a LTPDA repository. Click here to get more information about the parameters.

```
% Retrieves the objects with the object ID 1..10
a = ao(plist('hostname', '123.123.123.123', 'database', 'ltpda_test', 'ID', [1:10],
'binary', true));
```

From Polynomial

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: []]

Click here to get more information about the parameters.

```
a = ao(plist('polyval', [1 2 3], 'Nsecs', 10, 'fs', 10));
```

From Pzmodel

Generates an AO with a timeseries with a prescribed spectrum. Click here to get more

information about the parameters.

```
p   = [pz(1,2) pz(10)]
z   = [pz(4)]
pzm = pzmodel(1, p, z)

fs    = 10
nsecs = 100
a = ao(plist('pzmodel', pzm, 'Nsecs', nsecs, 'Fs', fs));
```

◀ General constructor examples   Constructor examples of the SMODEL class ▶

©LTP Team

Constructor examples of the SMODEL class

A simple example

The following example shows you how to create a symbolic model.

```
>> s = smodel('a.*x.^2+b.*x+c');
```

You define with the command above the symbolic model. Please take care that each symbolic can represent a vector/matrix of numbers. This mean that you should use the array operators like '.*' or '.+' or ... and not the matrix operators like '*' or '^' or ....

Now, it is necessary to define the parameters with their default values

```
>> s.setParams({'a', 'b', 'c'}, {1 2 3})
---- symbolic model 1 ----
       name: None
       expr: a.*x.^2+b.*x+c
     params: {'a', 'b', 'c'}
     values: {1, 2, 3}
       xvar:
      xvals: []
     xunits: []
     yunits: []
description:
       UUID: 7a084750-89c2-47f0-8c6b-65e9079bf6b3
     -------------------------
```

Finally it is necessary to set the 'x' variable and the values.

```
>> s.setXvar('x')
---- symbolic model 1 ----
       name: None
       expr: a.*x.^2+b.*x+c
     params: {'a', 'b', 'c'}
     values: {1, 2, 3}
       xvar: x
      xvals: []
     xunits: []
     yunits: []
description:
       UUID: ca9276b0-cc97-4001-9135-00c6799ec8b1
     -------------------------

>> s.setXvals(-100:100)
---- symbolic model 1 ----
       name: None
       expr: a.*x.^2+b.*x+c
     params: {'a', 'b', 'c'}
     values: {1, 2, 3}
       xvar: x
      xvals: [-100 -99 -98 -97 -96 -95 -94 -93 -92 -91 -90 -89 -88 -87 -86 ...]
     xunits: []
     yunits: []
description:
       UUID: f43e2f9b-3fb6-4e3c-92ba-7cba31c6a6c3
     -------------------------
```

After all steps above it is possible to plot the model.

```
>> s.setXunits('s');
>> s.setYunits('V');
>> s.setName('Just a test');
>> iplot(s.eval)
```

⬅ Constructor examples of the AO class   Constructor examples of the MFIR class ➡

©LTP Team

Constructor examples of the MFIR class

[Copy an MFIR object](#)
[Construct a MFIR object by loading the object from a file](#)
[Construct a MFIR object from an Analysis Object (AO)](#)
[Construct a MFIR object from a pole/zero model (PZMODEL)](#)
[Construct a MFIR object from a standard type](#)
[Construct a MFIR object from an existing filter model](#)
[Construct a MFIR object from a difference equation](#)

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 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 true to retrieve from stored binary representation (not always available). [default: true]

```
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 SMODEL class   Constructor examples of the MIIR class ▶

©LTP Team

LTPDA Toolbox™ ◄ ►

Constructor examples of the MIIR class

[Copy a MIIR object](#)
[Construct a MIIR object by loading the object from a file](#)
[Construct a MIIR object from a parfrac object](#) `(PARFRAC)`
[Construct a MIIR object from a pole/zero model](#) `(PZMODEL)`
[Construct a MIIR object from a standard type](#)
[Construct a MIIR object from an existing model](#)
[Construct a MIIR object from a difference equation](#)

Copy an IIR filter object

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

```
>> iir1 = miir(plist('type', 'lowpass'));
>> iir2 = miir(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 true to retrieve from stored binary representation (not always available). [default: true]

```
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 ➡

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Constructor examples of the PZMODEL class

[General information about PZMODEL objects](#)
[Construct empty PZMODEL object](#)
[Construct a PZMODEL object by loading the object from a file](#)
[Construct a PZMODEL object from gain, poles and zeros](#)
[Construct a PZMODEL object from an existing model](#)
[Construct a PZMODEL object from a parameter list (PLIST) object](#)

General information about pole/zero models

For help in designing a PZMODEL object, see also the PZMODEL Helper GUI documentation

---

Construct empty PZMODEL object

The following example creates an empty pzmodel object

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

Construct a PZMODEL object by loading the object from a file

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

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

Construct a PZMODEL object from gain, poles and zeros

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

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

pzm = pzmodel(gain, poles, zeros)
```

```
---- pzmodel 1 ----
model:    None
gain :    10
pole 001: pole(1,2)
pole 002: pole(40)
zero 001: zero(10,3)
zero 002: zero(100)
------------------
```

It is possible to give the model direct a name.

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

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

Construct a PZMODEL object from an existing model

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

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

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

Construct a PZMODEL from its definion.

'gain'   Model gain [default: 1]

'poles'   Vector of pole objects [default: empty pole]

'zeros'   Vector of zero objects [default: empty zero]

'name'   Name of model [default: 'None']

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

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

Constructor examples of the 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 true to retrieve from stored binary representation (not always available). [default: true]

```
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]
------------------
```

©LTP Team

Constructor examples of the RATIONAL class

[Copy an rational object](#)
[Construct an rational object by loading the object from a file](#)
[Construct an rational object from an parfrac object](#)
[Construct an rational object from a pole/zero model](#)
[Construct an rational object from numerator and denominator coefficients](#)
[Construct an rational object from a parameter list object (PLIST)](#)

Copy an rational object

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

```
>> ra1 = rational([1 2 3], [4 5 6 7], 'my rational');
>> ra2 = rational(ra1)
---- rational 1 ----
model:    my rational
num:      [1 2 3]
den:      [4 5 6 7]
iunits:   []
ounits:   []
--------------------
```

REMARK: The following command copies only the handle of an object and doesn't create a copy of the object (as above). This means that everything that happens to the copy or original happens to the other object.

```
>> ra1 = rational()
---- rational 1 ----
model:    none
num:      []
den:      []
iunits:   []
ounits:   []
--------------------
>> ra2 = ra1;
>> ra2.setName('my new name')
---- rational 1 ----
model:    my new name
num:      [1 2 3]
den:      [4 5 6 7]
iunits:   []
ounits:   []
--------------------
```

If we display ra1 again then we see that the property 'name' was changed although we only have changed ra2.

```
>> ra1
---- rational 1 ----
model:    my new name
num:      [1 2 3]
den:      [4 5 6 7]
iunits:   []
```

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

Construct an rational object by loading the object from a file

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

```
ra = rational('rational_object.mat')
ra = rational('rational_object.xml')
```

or in a PLIST

```
pl = plist('filename', 'rational_object.xml');
ra = rational(pl)
```

Construct an rational object from an parfrac object

The following example creates a new rational object from an parfrac object.

```
>> pf = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3, 'my parfrac')
---- parfrac 1 ----
model:    my parfrac
res:      [1;2+i*1;2-i*1]
poles:    [6;1+i*3;1-i*3]
dir:      3
pmul:     [1;1;1]
iunits:   []
ounits:   []
-------------------
>> ra = rational(pf)
---- rational 1 ----
model:    rational(my parfrac)
num:      [3 -19 30 -110]
den:      [1 -8 22 -60]
iunits:   []
ounits:   []
-------------------
```

or in a plist

```
>> pf = parfrac([1 2+1i 2-1i], [6 1+3i 1-3i], 3, 'my parfrac');
>> pl = plist('rational', rat)
>> ra = rational(pl)
```

Construct an rational object from a pole/zero model

The following example creates a new rational object from a pole/zero model.

```
>> pzm = pzmodel(1, {1 2 3}, {4 5}, 'my pzmodel')
---- pzmodel 1 ----
    name: my pzmodel
    gain: 1
   delay: 0
  iunits: []
  ounits: []
pole 001: (f=1 Hz,Q=NaN)
pole 002: (f=2 Hz,Q=NaN)
pole 003: (f=3 Hz,Q=NaN)
zero 001: (f=4 Hz,Q=NaN)
zero 002: (f=5 Hz,Q=NaN)
-------------------
```

```
>> ra = rational(pzm)
---- rational 1 ----
model:    rational(None)
num:      [0.0012  0.0716  1]
den:      [0.0001  0.0036  0.0401  0.2122  1]
iunits:   []
ounits:   []
------------------
```

or in a plist

```
>> pzm = pzmodel(1, {1 2 3}, {4 5}, 'my pzmodel')
>> pl  = plist('pzmodel', pzm)
>> ra  = rational(pl)
```

Construct an rational object from numerator and denominator coefficients

The following example creates a new rational direct from the numerator and denominator coefficients.

```
>> num    = [1 2 3];
>> den    = [4 5 6];
>> name   = 'my rational';
>> iunits = 'Hz';
>> ounits = 'V';

>> ra = rational(num, den)
>> ra = rational(num, den, name)
>> ra = rational(num, den, name, iunits, ounits)
---- rational 1 ----
model:    my rational
num:      [1 2 3]
den:      [4 5 6]
iunits:   [Hz]
ounits:   [V]
------------------
```

Construct an rational object from a parameter list (`plist`)

Constructs an rational object from the description given in the parameter list.

[Use the key word 'hostname'](#)
[Use the key word 'num' or 'den'](#)

Use the key word 'hostname'

Construct an rational object by retrieving it from a LTPDA repository.

The relevant parameters are:

Key	Description
'hostname'	the repository hostname. [default: 'localhost']
'database'	The database name [default: 'ltpda']
'id'	A vector of object IDs. [default: []]
'cid'	Retrieve all rational objects from a particular collection

'binary'	Set to true to retrieve from stored binary representation (not always available). [default: true]

```
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]
-------------------
```

©LTP Team

Constructor examples of the TIMESPAN class

[Construct empty TIMESPAN object](#)
[Construct a TIMESPAN object by loading the object from a file](#)
[Construct a TIMESPAN object with a start and end time](#)
[Construct a TIMESPAN object from a parameter list (PLIST) object](#)

Construct empty TIMESPAN object

The following example creates an empty timespan object

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

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

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

Construct a TIMESPAN object by loading the object from a file

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

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

Construct a TIMESPAN object with a start and end time

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

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

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

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

Construct an TIMESPAN by its properties definition

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

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

Additional parameters:

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

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

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

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

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

©LTP Team

Constructor examples of the PLIST class

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

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

Creating parameter lists from parameters.

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

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

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

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


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

Creating parameter lists directly.

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

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

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

Appending parameters to a parameter list.

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

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

Finding parameters in a parameter list.

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

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

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

Removing parameters from a parameter list.

You can also remove parameters from a parameter list:

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

Setting parameters in a parameter list.

You can also set parameters contained in a parameter list:

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

Combining multiple parameter lists.

Parameter lists can be combined:

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

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

◀ Constructor examples of the TIMESPAN class   Constructor examples of the SPECWIN class ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Constructor examples of the SPECWIN class

[Construct empty SPECWIN object](#)
[Construct a SPECWIN object by loading the object from a file](#)
[Construct a SPECWIN of a particular window type an length](#)
[Construct a SPECWIN Kaiser window with the prescribed psll](#)
[Construct a SPECWIN object from a parameter list (PLIST) object](#)

For help in designing a SPECWIN object, see also the Spectral Window GUI documentation

Construct empty SPECWIN object

The following example creates an empty specwin object

```
w = specwin()
-------- None ------------

    alpha: -1
     psll: -1
      rov: -1
    nenbw: -1
     w3db: -1
 flatness: -1
       ws: -1
      ws2: -1
      win: 1

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

Construct a SPECWIN object by loading the object from a file

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

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

Construct a SPECWIN of a particular window type an length

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

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

```
        -------- Hanning ------------

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


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

[List of available window functions](#)

Construct a SPECWIN Kaiser window with the prescribed psll

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

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

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


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

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

Construct a SPECWIN from its definion.

'Name'	Spectral window name [default: 'Kaiser']
'N'	Spectral window length [default: 10]
'PSLL'	Peak sidelobe length (only for Kaiser type) [default: 150]

```
pl = plist('Name', 'Kaiser', 'N', 1000, 'PSLL', 75);

w = specwin(pl)
-------- Kaiser ------------

   alpha: 3.21394
    psll: 75
```

```
       rov: 63.4095
     nenbw: 1.85055
      w3db: 1.75392
  flatness: -0.963765
        ws: 389.305
       ws2: 280.892
       win: 1000

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

```
pl = plist('Name', 'Hanning', 'N', 1000);

w =specwin(pl)
-------- Hanning -----------

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

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

◀ Constructor examples of the PLIST class   LTPDA Training Session 1 ▶

©LTP Team

LTPDA Toolbox™   [contents]   ◀ ▶

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 web-site].

1. [Topic 1 – The basics of LTPDA]
2. [Topic 2 – Pre-processing of data]
3. [Topic 3 – Spectral Analysis]
4. [Topic 4 – Transfer function models and digital filtering]
5. [Topic 5 – Model fitting]

In addition, throughout the course of this training session, we will perform a full analysis of some lab data. The inputs to the analysis are two time-series data streams, the first is the recorded output of an interferometer, the second is a recording of the room temperature in the vicinity of the interferometer. Both are recorded with different sample rates and on different sampling grids. The temperature data is unevenly sampled, and may 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

LTPDA Toolbox™   [contents](#)   ◀ ➡

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

©LTP Team

Introducing Analysis Objects

Analysis objects are one type of LTPDA user object. The job of an analysis object is to bring together some numerical data with a set of descriptive meta-data. In addition, AOs are clever and can keep track of all the things you do to them. So at any time, you can look at the history of the AO to see what processing steps have happened in the past.

Example history trees are shown below:
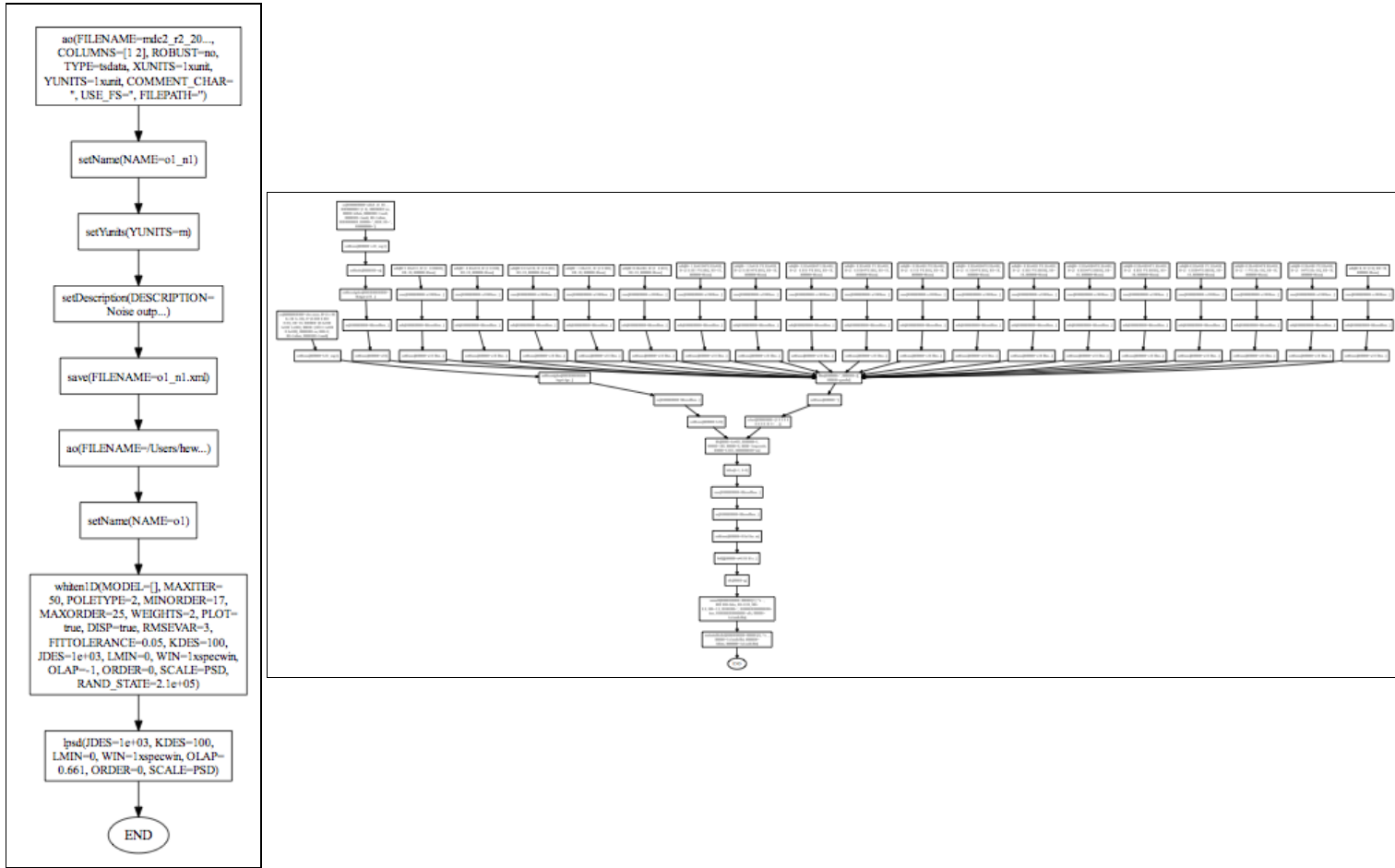
Making AOs

Exercise 1 – Your first Analysis Object

AOs can be constructed in many different ways. Each of the different ways is called a constructor. For example, there is a constructor to make an AO from a set of numeric values, there is also a constructor to make an AO from a data file. Each time you construct an AO, you make an instance of the class, `ao`. The variable you have in MATLAB is then just a reference to the object you constructed.

This may all sound confusing to start with, but will become clearer as we go through the examples below.

Let's make an AO. On the MATLAB terminal, type the following: `a = ao` and hit return. You should see an output like the following:

```
>> a = ao()
M:         running ao/ao
M:         running ao/display
----------- ao 01: a -----------

      name: ''
      data: None
      hist: ao / ao / b078ca046a8f6c078b4bc5895a6ed5492ec3b7bd
description:
      UUID: eeed5558-00bc-4c44-beb4-f17c2835f780
-----------------------------
```

Note that the number of lines beginning with the "M:" syntax may vary depending on the level of "verbosity" that can be set via the [LTPDA Preferences](#) You have just made your first AO. It's not a very exciting AO since it contains no data, but it is an AO 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)
----------- ao 01: a -----------

      name: ''
      data: 1
            -------- cdata [vals] ------------
                 y:  [1x1], double
                dy:  [0x0], double
            yunits:
            ---------------------------------

      hist: ao / ao / b078ca046a8f6c078b4bc5895a6ed5492ec3b7bd
description:
      UUID: f6de0f9b-1067-48c2-bf2b-4dd04e1d358f
-----------------------------
```

Now you can see that your AO has some data. The data is of type `cdata` (more on that later), it has no Y units, and it contains a single value, 1.

Note also that the information shown in the "hist" field may vary depending on the version of the LTPDA Toolbox you installed. The string reported there is a hash of the git repository hosting the source code of LTPDA.

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')
----------- ao 01: Bob -----------

      name: Bob
      data: 1
            -------- cdata [vals] -----------
              y:  [1x1], double
             dy:  [0x0], double
          yunits:
            ---------------------------------

      hist: ltpda_uo / setName / b078ca046a8f6c078b4bc5895a6ed5492ec3b7bd
description:
      UUID: aed61cbd-13ae-45c3-a63b-8d38b98813e1
------------------------------
```

The ao has a new name. The function (or more strictly, the 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.

Exercise 3 – Viewing the history

Keeping trace of the analysis step applied to the data contained in one or more objects is one of the key features supported in LTPDA. Analysis Objects, and more complicated data objects that we will cover in following sections, can be modified by LTPDA functions (simple mathematical operators, property settings, complicated algorithms). The basic components acting on LTPDA Objects are called methods. Each method adds one step in the history of the object, so that the flow of the actions happened on the data can be tracked, inspected, and re-run. This might become very useful in particular when dealing with many steps of analysis, but also in simple cases. LTPDA supports viewing the history in two ways. The more elegant one, especially in the case of complicated analysis flows, for instance involving many objects, is 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'))
% or
a.viewHistory();
```

and you should get a figure something like that below in your system pdf viewer. You can see the only things we have done are to construct the object and set its name.
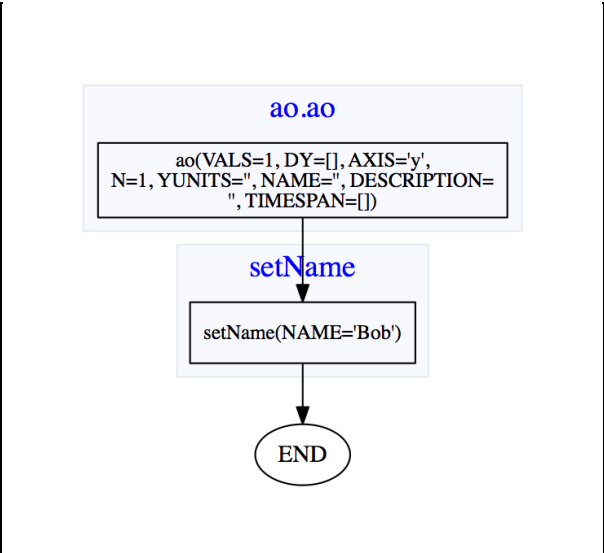
Don't worry about all this `plist` business, we'll get to that soon enough. For now it's enough to know that the conversion to pdf is done by the graphviz engine, and this needs to write the pdf to a file. That's the 'filename' specified in that last command.

Installation of graphviz is covered in the LTPDA user manual under the section System Requirements.

For short history trees, and in case the call to `viewHistory` fails because graphviz is not installed, the history can also be viewed using the method `type()`. In the MATLAB terminal, type `a.type` and hit return. You should get a MATLAB list of lines looking something like:

```
>> a.type
--------------------------------------------------
----------------------Bob------------------------
--------------------------------------------------

a8692974 = ao([plist('VALS', [1], 'DY', [], 'AXIS', 'y', 'N', [1], 'YUNITS', '', 'NAME', '',
'DESCRIPTION', '', 'TIMESPAN', [])]); % ao
a8693125 = setName(a8692974, [plist('NAME', 'Bob')]); % ltpda_uo
a_out = a8693125;

--------------------------------------------------
----------------------Bob------------------------
--------------------------------------------------
```

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 5 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.
xyzdata	For storing an arbitrary set of x-y-z data sets, 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')
----------- ao 01: Bob -----------

        name: Bob
        data: 1
              -------- cdata [vals] ------------
                  y:  [1x1], double
                 dy:  [0x0], double
            yunits:  [km]
              --------------------------------

        hist: ao / setYunits / b078ca046a8f6c078b4bc5895a6ed5492ec3b7bd
description:
        UUID: a90feb2e-a9dd-4690-9455-65ae48ca7176
--------------------------------
```

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`.)

Additional, user-defined units can be added to LTPDA via the [LTPDA Preferences](#).

⬅ 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:      constructing from Y values and fs
---------- ao 01: a -----------

        name: ''
        data: (0,1) (0.5,2) (1,3) (1.5,4) (2,5)
              -------- tsdata [x, y] --------

                 x:  [1 5], double
                 y:  [1 5], double
                dx:  [0 0], double
                dy:  [0 0], double
            xunits:  [s]
            yunits:
                fs:  2
             nsecs:  2.5
                t0:  1970-01-01 00:00:00.000 UTC
              ------------------------------

        hist: ao / ao / a7151bd50f5dbc983e14bff537ac39ebf07824c5
description:
        UUID: 1025a1c4-8dac-4ffd-99b3-552c41a98f14
      ------------------------------
```

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('2014-06-01 12:23:44');
a.setDescription('My lovely time-series')
```

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], 'type',...
       'tsdata'))
```

> The ellipsis (...) 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 'type', 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], ...
             'type', 'tsdata', ...
             'name', 'Bob', ...
             't0', '2014-06-01'))
```

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:      constructing from plist
---------- ao 01: a ----------

      name: ''
      data: (0,0) (0.1,0.11) (0.2,0.24) (0.3,0.39) (0.4,0.56) ...
            -------- tsdata [x, y] --------

                x:  [10000 1], double
                y:  [10000 1], double
               dx:  [0 0], double
               dy:  [0 0], double
           xunits:  [s]
           yunits:
               fs:  10
            nsecs:  1000
               t0:  1970-01-01 00:00:00.000 UTC
            ------------------------------

      hist: ao / ao / a7151bd50f5dbc983e14bff537ac39ebf07824c5
description:
      UUID: ebedd13b-730b-4175-84e9-67740d4bb31e
      ------------------------------
```

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)
```

LTPDA provides also some convenient 'factory' methods to quickly simulate random data. For instance:

```
% We can also use the random factory:
% a = ao.randn(nsecs, fs)
a3 = ao.randn(1000, 10)
```

Here we have made three 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.
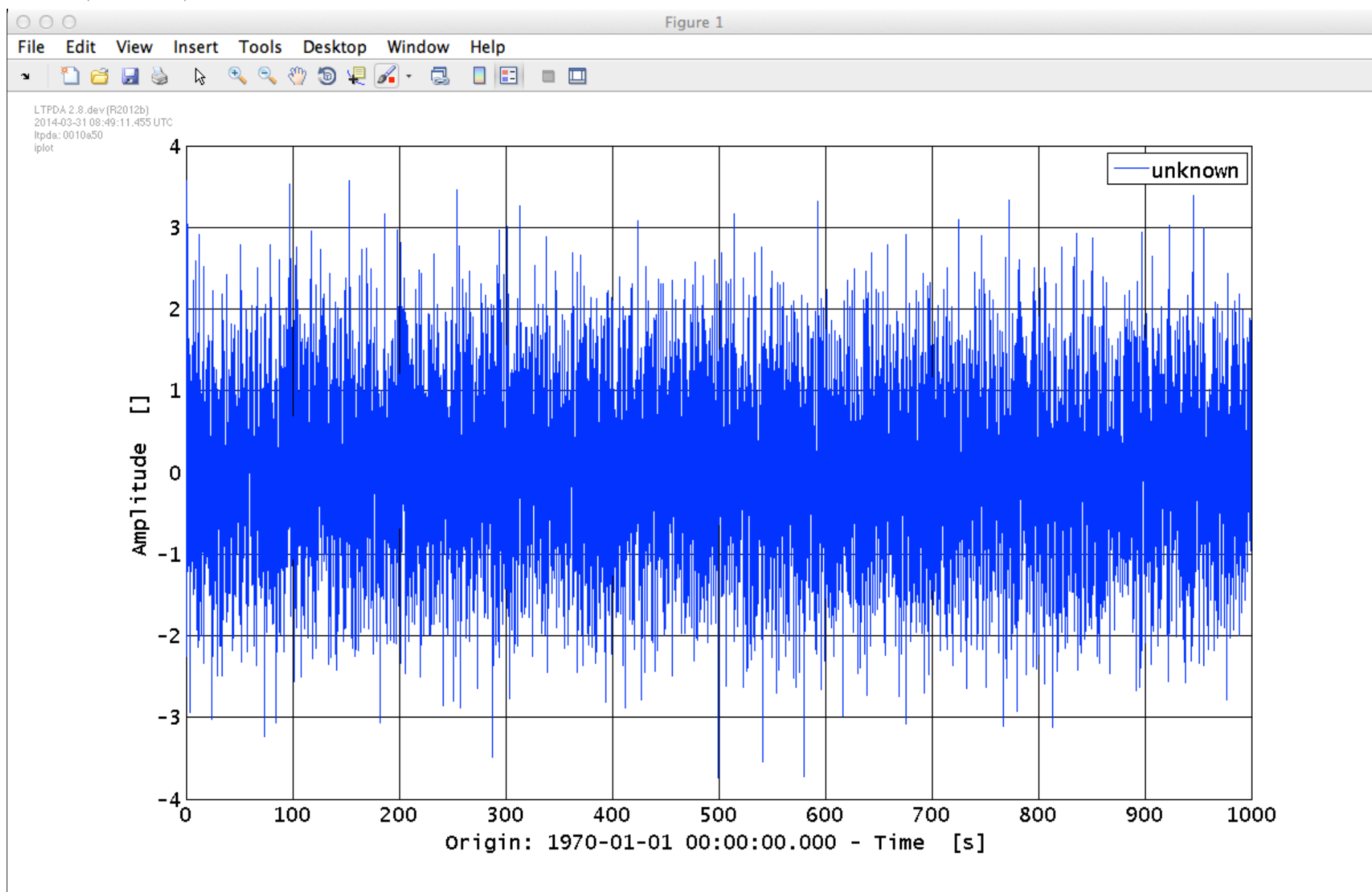
We can make a more interesting plot if we first specify some of the properties of the AOs. For example, type the following commands to set the names and Y units of the two AOs we made earlier:

```
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:
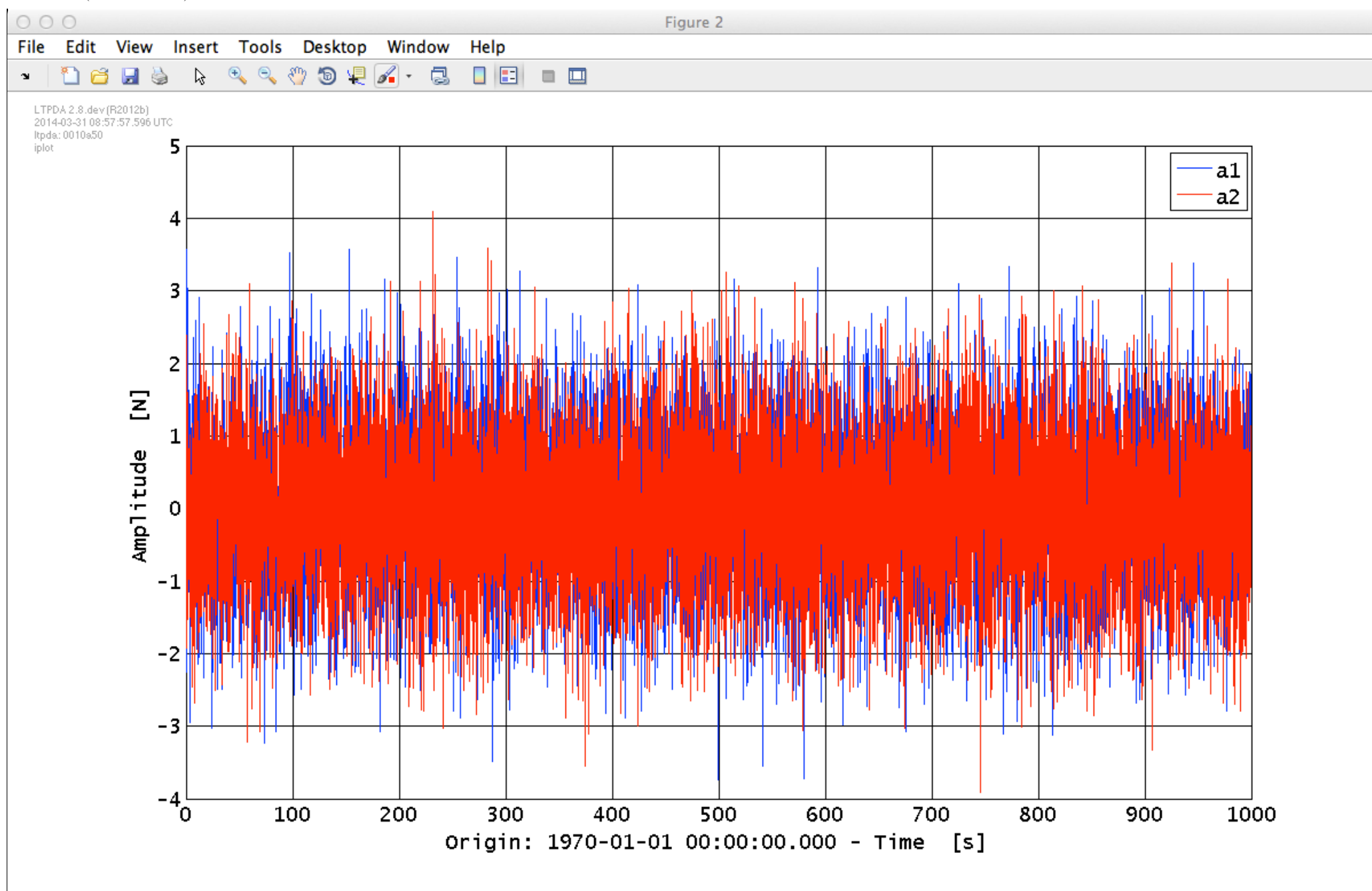
Calling the `setName` method with no input argument causes the AO to be named with the variable name.

`iplot` has many configurable parameters which are (mostly) documented in the help. Additionally, you can specify in detailed way the plotting

preferences by accessing the appropriate [section](#) in the LTPDA Preferences.

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:
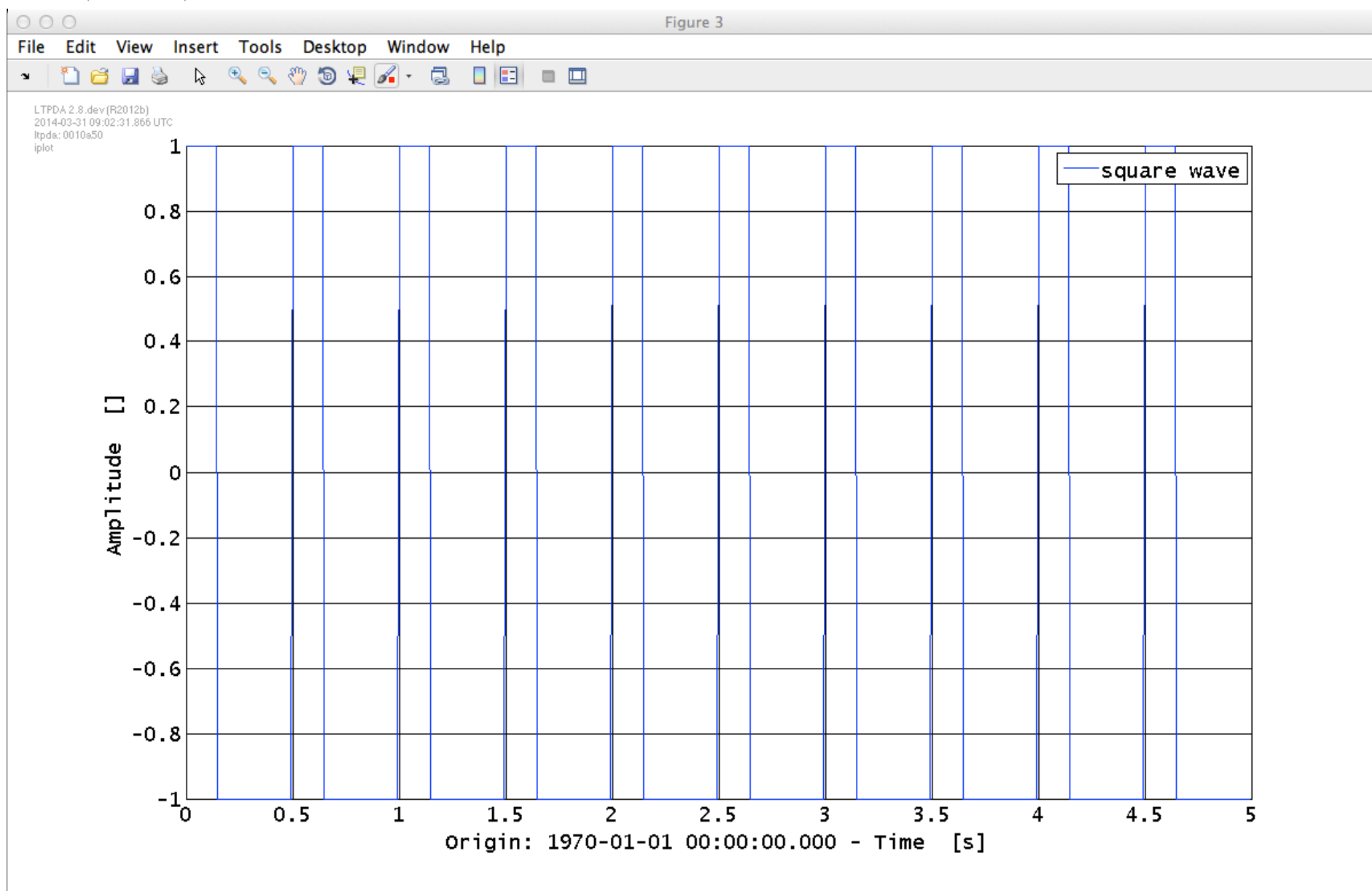
You can construct various different waveforms, but each has different parameters to set. The help of the AO method details the possibilities (`help ao ->` `click on "Parameters Description" -> select "From Waveform"`); for the constructor we are describing the the relevant extract is [here](#).

©LTP Team

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
c.viewHistory
```

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.

Addition operator on AOs by default acts on the y field, and it can be used to 'smartly' sum objects. For instance, time-series AOs can be added together provided they have the same time-base:

```
a = ao(plist('waveform', 'noise', 'fs', 10, 'nsecs', 100, ...
'sigma', 0.1));
b = ao(plist('waveform', 'sinewave', 'fs', 10, 'nsecs', 100, ...
'A', 1, 'f', 0.1));
c = a + b
iplot(a, b, c)
```

Yielding a plot like the following:
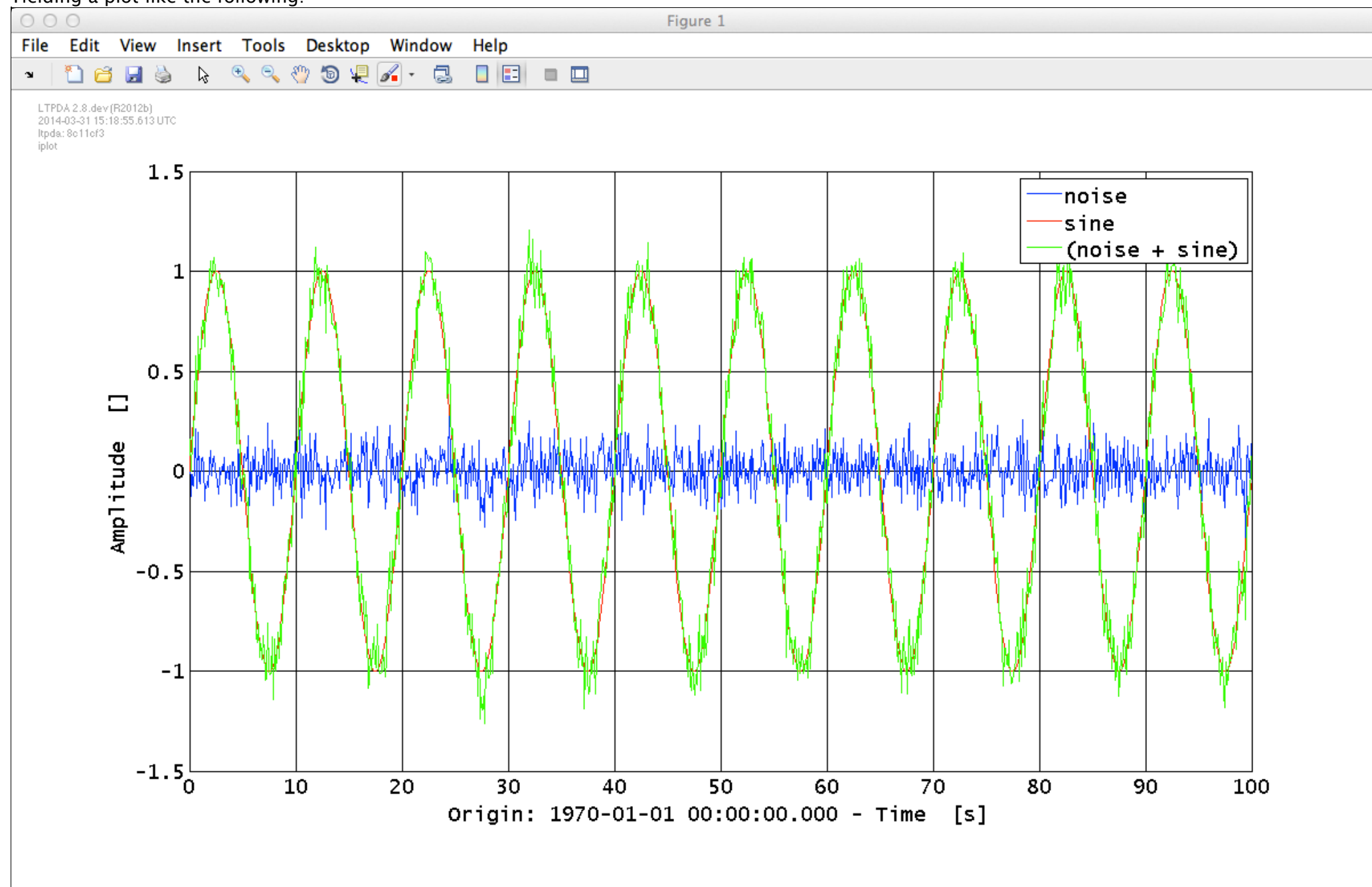
©LTP Team

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.

Two 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');
```

For both the operations of saving and loading, the `plist` syntax is also available:

```
pl = plist('filename', 'foo.mat');
a1.save(pl);
a2 = ao(pl);
```

◀ Basic math with AOs   Constructing AOs from data files ▶

©LTP Team

Saving and loading AOs (LTPDA Toolbox)

Constructing AOs from data files

You can build AOs from existing ASCII data files. Various formats are supported, for example: multiple columns, or 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:

```
>> pl = plist('filename', 'topic1/simpleASCII.txt', ...
              'columns', [1 2], ...
              'type', 'tsdata');
>> a = ao(pl)
M:      constructing from plist
M:      load file: simpleASCII.txt
M:      constructing from filename and/or plist
M:      constructing from data object tsdata
----------- ao 01: a -----------

       name: ''
       data: (0,1.13549060238654) (0.1,2.37202031808076) (0.2,-0.531181753855465)
(0.3,1.25684477541224) (0.4,1.30895517480354) ...
              -------- tsdata [x, y] --------

                 x:  [1000 1], double
                 y:  [1000 1], double
                dx:  [0 0], double
                dy:  [0 0], double
             xunits:  [s]
             yunits:
                 fs:  10
              nsecs:  100
                 t0:  1970-01-01 00:00:00.000 UTC
              -----------------------------

       hist: ao / ao / 8c11cf35c322925062d6fb1759d7c589b696a19b
description:
       UUID: a1757c14-e6e7-4857-b213-0707d7a282ae
-----------------------------
```

From the output on the screen you can see that

1. the sample rate of the data is 10Hz, as expected
2. the length of the data is 100s
3. no name of the AO has automatically been set

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', ...
                'type', 'tsdata', ...
                '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 ▶

©LTP Team

Writing LTPDA scripts

Up to now, all the activity of the tutorial has been carried out on the MATLAB command terminal. It is, of course, much more convenient to collect the commands together in to a MATLAB script. In this sense, LTPDA scripting is just the same as normal MATLAB scripting; just using LTPDA commands.

Here's an example script which makes two white-noise time-series and estimates the power-spectral-density of each.

```matlab
%% 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 PSD Estimates

psds = psd(a1, a2);
a1xx = psds(1);
a2xx = psds(2);

% The following also works.
b1xx = psd(a1);
b2xx = psd(a2);
```

You can explore the commands used to rebuild the object by using the `type` method. Or you can inspect (with `type` or `viewHistory` the history as we did earlier. For example, try this command for each of the cases above:

```matlab
   type(a1xx)
   type(b1xx)
```

← Constructing AOs from data files   IFO/Temperature Example – Introduction →

©LTP Team

Writing LTPDA scripts (LTPDA Toolbox)

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
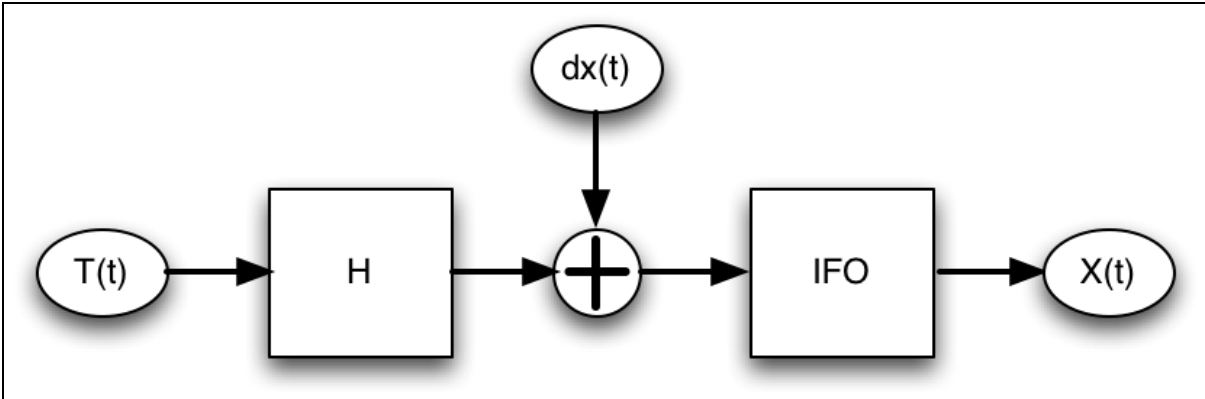
We have measured the two signals, T(t) and x(t). The displacement input to the interferometer, dx(t), is inaccessible to us, and is in fact the data we want to recover from the data analysis.

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:

$$Y(t)[\text{m}] = X(t)\frac{\lambda}{2\pi}[\text{rad}]$$

where lambda=1064nm.

To do the calibration, you can use the method `ao/scale`, specifying the `factor` and the `yunits` of the factor itself. Then save the resulting time-series to disk in `ifo_temp_example/ifo_disp.xml`.

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 Celsius.
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.

> 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 ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◄ ►

Topic 2 - Pre-processing of data

In the signal pre-processing session we will learn about a variety of functions with which we can process our data prior to further analysis. Below are the functions we will learn about in this topic.

- [Topic 2.1 - Downsampling a time-series AO](#)
- [Topic 2.2 - Upsampling a time-series AO](#)
- [Topic 2.3 - Resampling a time-series AO](#)
- [Topic 2.4 - Interpolation of a time-series AO](#)
- [Topic 2.5 - Remove trends from a time-series AO](#)
- [Topic 2.6 - Whitening data](#)
- [Topic 2.7 - Select and find data from an AO](#)
- [Topic 2.8 - Split and join AOs](#)
- [Topic 2.9 - Pre-processing the IFO/Temperature Example](#)

◄ IFO/Temperature Example - Introduction   Downsampling a time-series AO ►

©LTP Team

Downsampling a time-series AO

Downsampling reduces the sampling rate of the input AOs by an integer factor, which can be very useful for example to reduce data load.

The `downsample` method takes the following [parameters](#):

Key	Description
FACTOR	The decimation factor [by default is 1: no downsampling] (must be an integer)
OFFSET	The sample offset for where the downsampling starts counting. By default, this value is zero so it starts counting from the first sample.

Example 1

First we'll create a time-series of random white noise at 10Hz. To do that, define a `plist` with the key/value pairs shown below, and pass these to the `ao` constructor. If you're using the workbench, add an `ao` constructor block to the canvas and select the parameter set "From Time-series Function" and set the parameters as they are in the `plist` below.

```
% create an AO of random data with fs = 10 Hz;
pl      = plist('name', 'None', ...
                'tsfcn',  'randn(size(t))', ...
                'fs',      10, ...
                'yunits', 'm', ...
                'nsecs',   10);
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
```
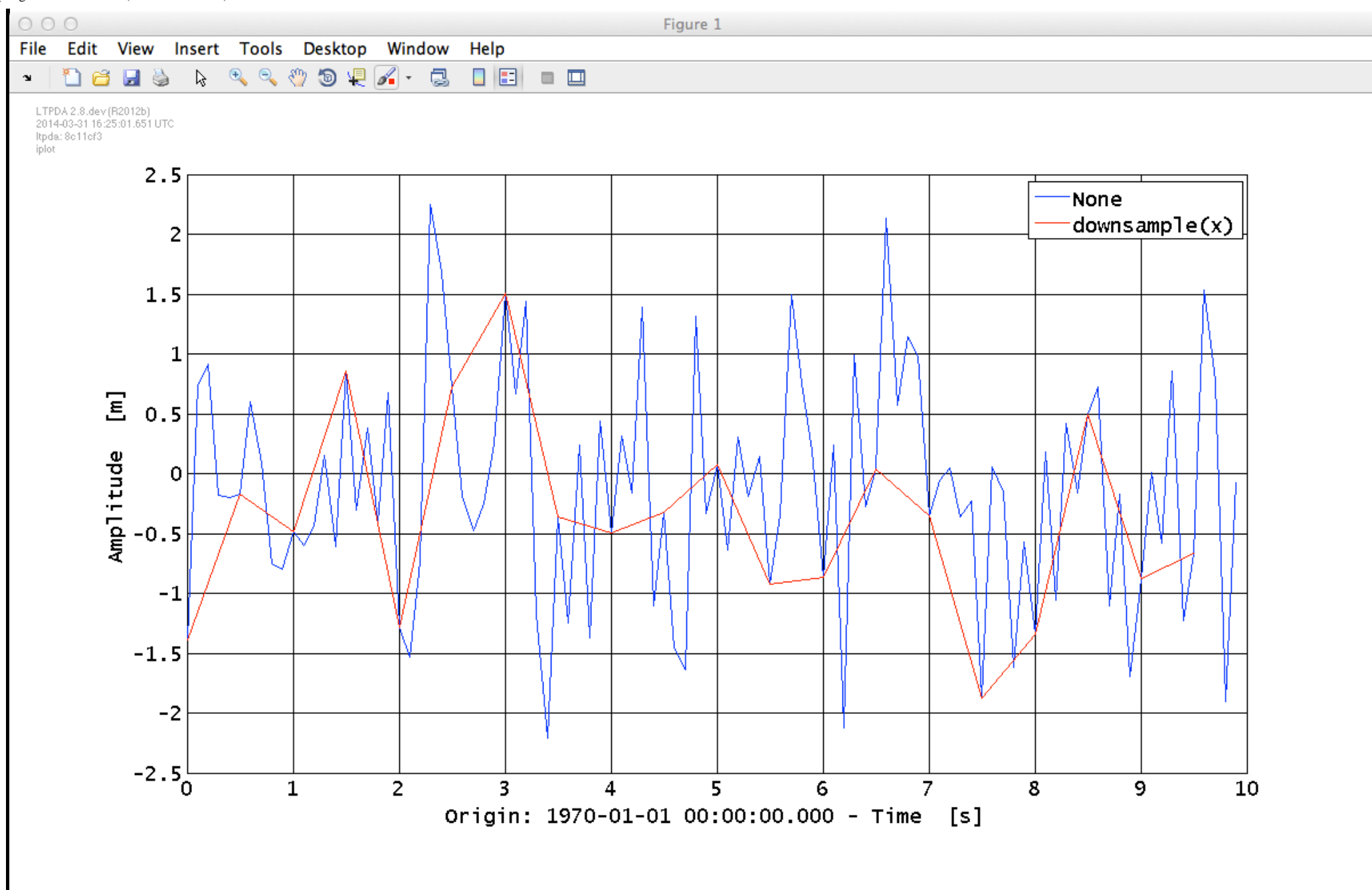
Example 2

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
```
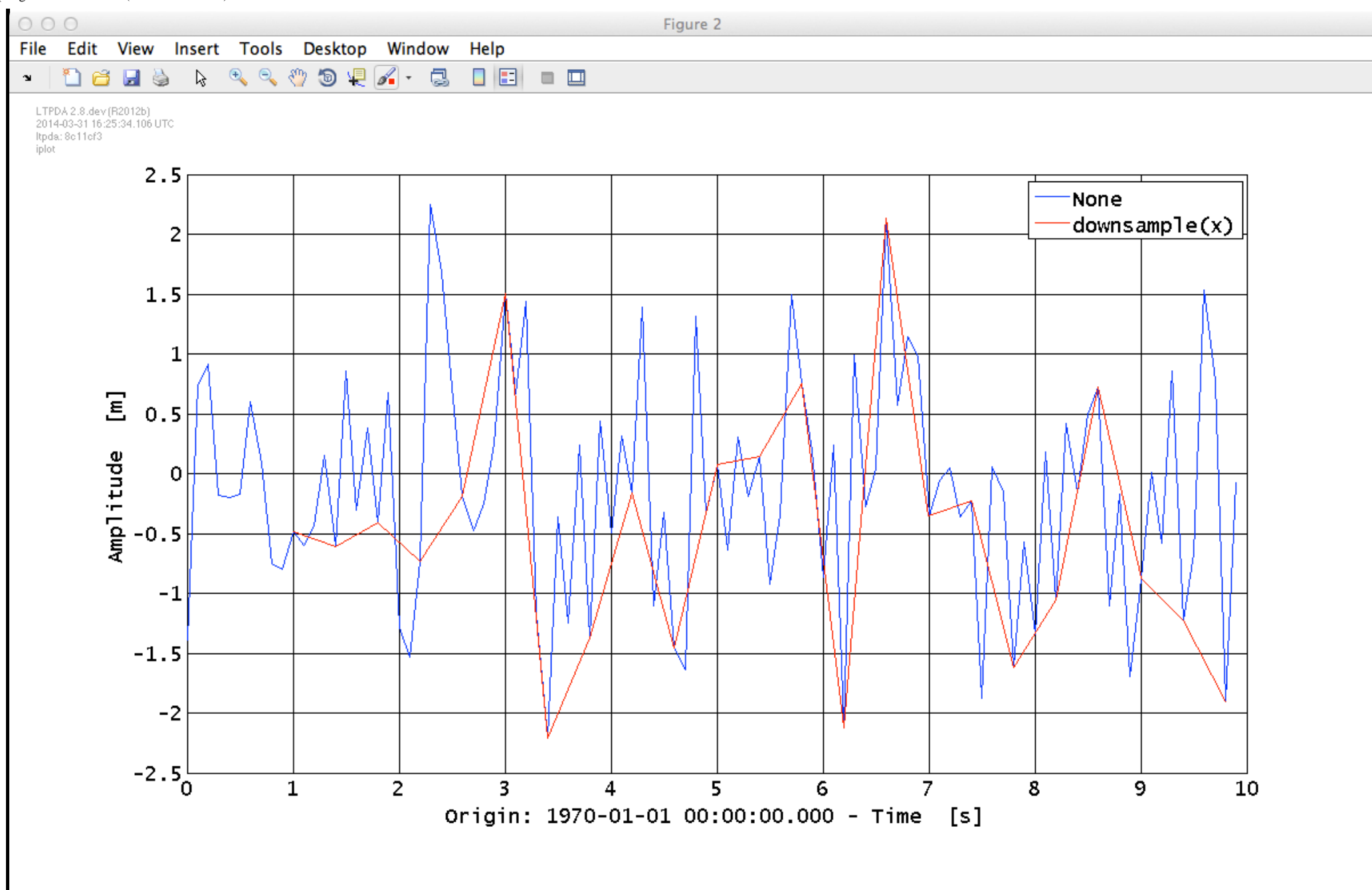
Upsampling a time-series AO

Upsampling increases the sampling rate of the input AOs by an integer factor

The `ao/upsample` method can take the following [parameters](#):

Key	Description
N	The upsample factor. The algorithm places 'N-1' zeros between each of the original samples.
PHASE	This parameter specifies an additional sample offset. The value must be between 0 and N-1.

Example 1

We will upsample a sine-wave by a factor of 3 with no initial phase offset.

Start by creating a sine-wave at 1Hz with a 30Hz sample rate and 10 seconds long. We can use the `ao` "From Waveform" parameter set to do this. (Equally, we can do this with the "From Time-series Function" parameter set.)

```
    pl = plist('Waveform', 'sine wave', 'f', 1, 'fs', 30, 'nsecs', 10);
    x  = ao(pl);
```

Now we can proceed to upsample this data by a factor 3. This will place 2 zero samples between each of the original samples.

```
    pl_up = plist('N', 3); % increase the sampling frequency by a factor of 10
    x_up  = upsample(x, pl_up); % resample the input AO (x) to obtain the upsampled AO (y)
    iplot(x, x_up, plist('XRanges', [0 1], 'Markers', {'o', 's'})) % plot original and upsampled data
```
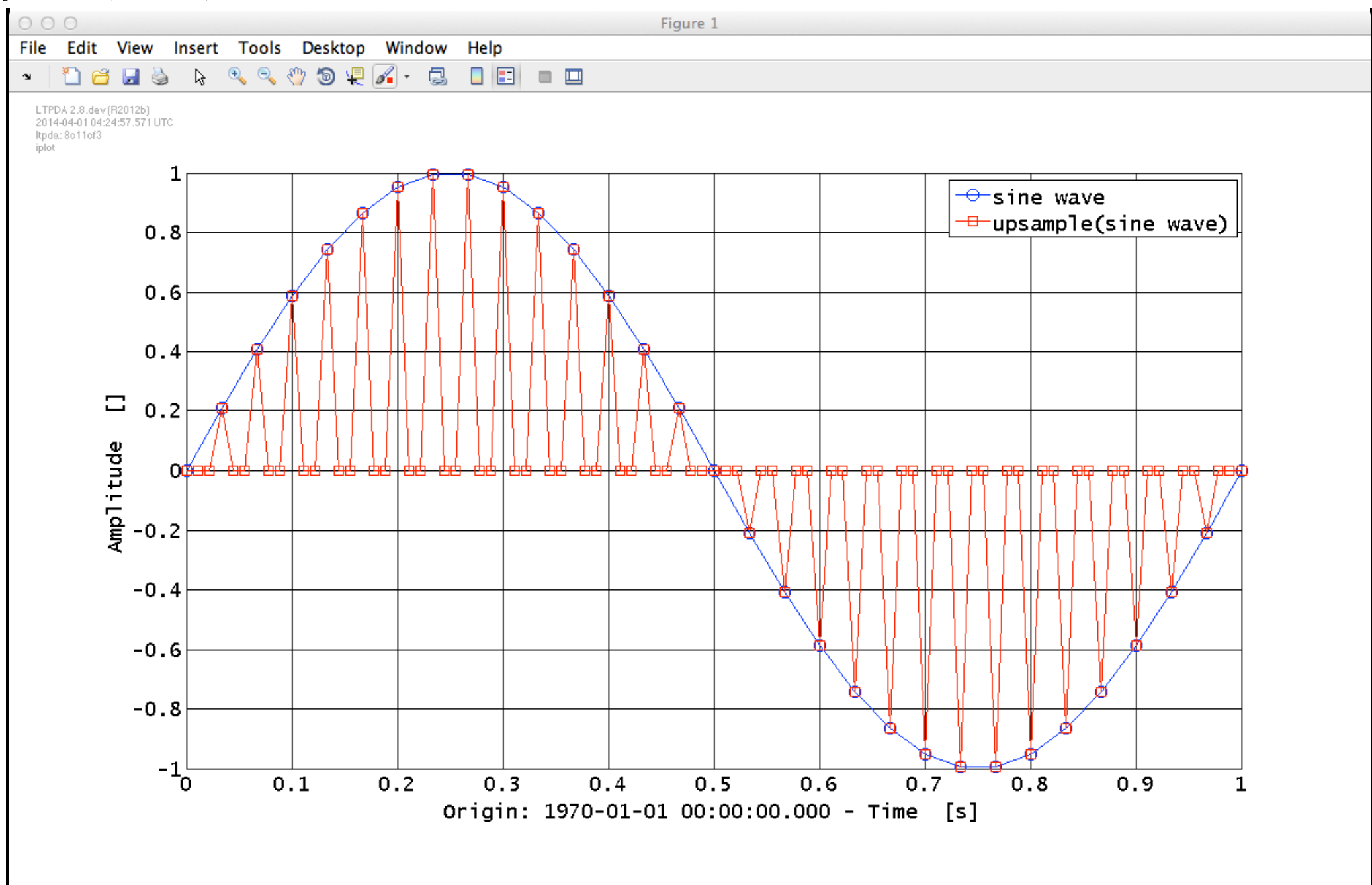
Example 2

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, 'yunits', 'm');
x           = ao(pl);
pl_upphase = plist('N', 4, 'phase', 2); % increase the sampling frequency and add phase of 2 samples to the upsampled data
x_upphase  = upsample(x, pl_upphase); % resample the input AO (x) to obtain the upsampled and delayed AO
iplot(x, x_upphase, plist('XRanges', [0 1], 'Markers', {'o', 's'})) % plot original and upsampled data
```

Upsampling a time-series AO (LTPDA Toolbox)

©LTP Team
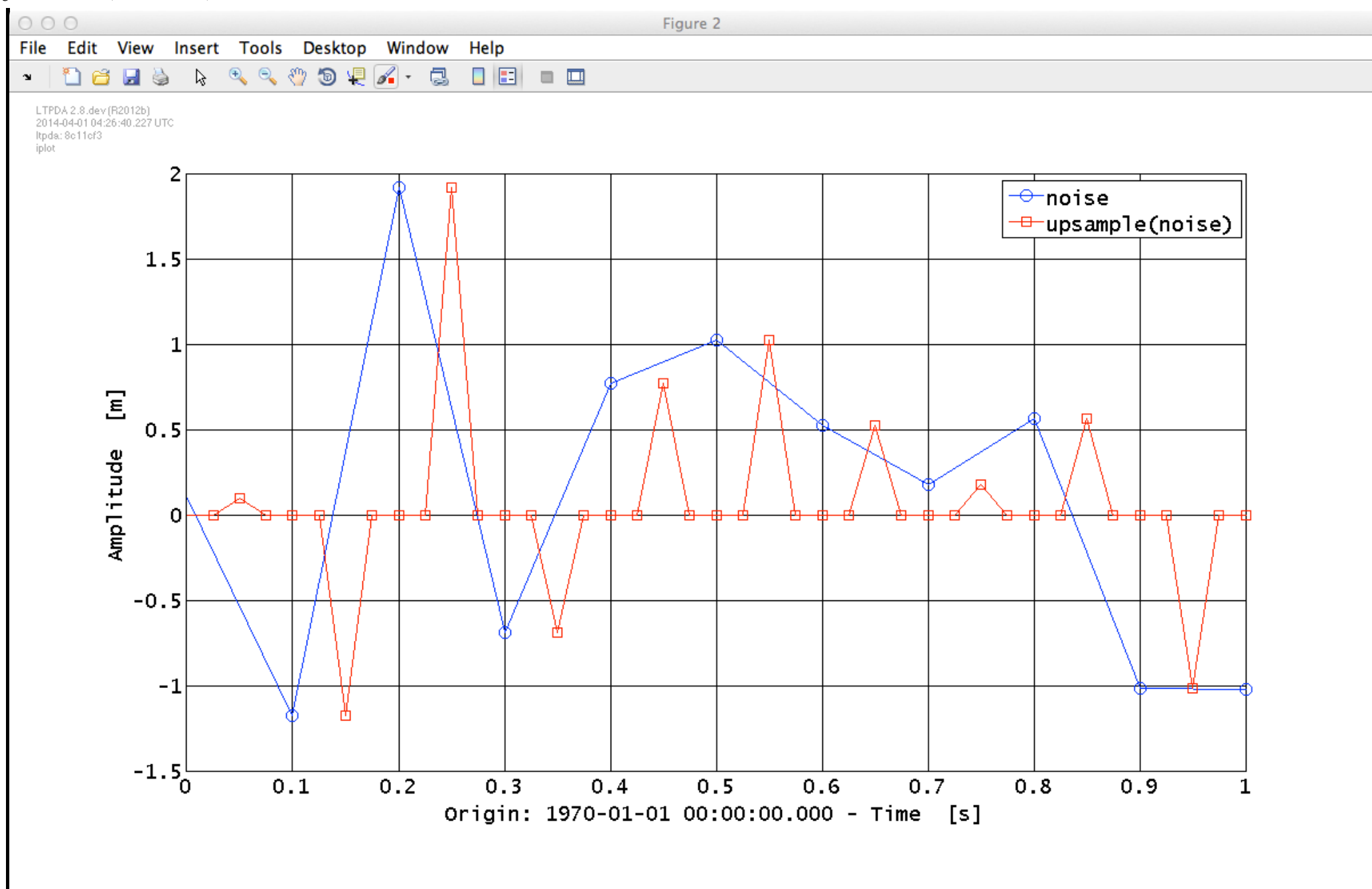
Resampling a time-series AO

Resampling is the process of changing the sampling rate of data. The method `ao/resample` changes the sampling rate of the input AOs to the desired output sampling frequency by performing band-limited interpolation, or interpolation.

If the ratio of the input and the desired output sample rate can be expressed as an integer ratio P/Q where both P and Q then band-limited interpolation can be performed. In this case, the data-series is upsampled (by inserting zeros) and then a low-pass filter is applied at the original Nyquist frequency.

The `ao/resample` method can be called with the syntax:

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

and can accept the following [parameters](#):

Key	Description
FSOUT	The desired output frequency (must be positive and integer)
FILTER	The filter to apply in the resampling process

Example 1

Here we will resample a sequence of random data from the original sampling rate of 1 Hz to an output sampling rate of 50 Hz.

```
pl     = plist('name', 'None', '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
```
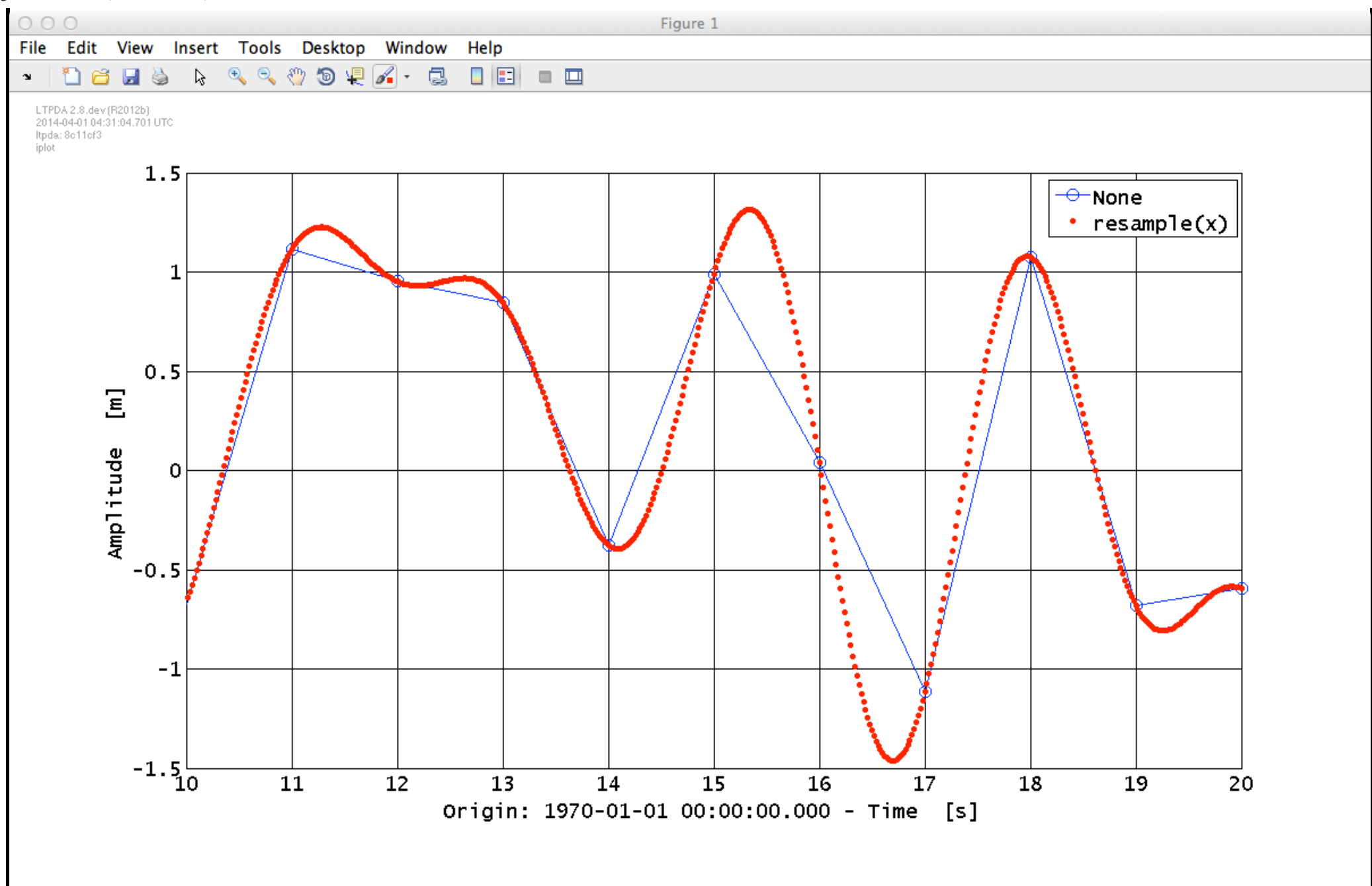
Interpolation of a time-series AO

The `ao` class has a method for interpolating data using different forms of interpolation. This method is called `ao/interp`.

To configure `ao/interp`, use the following [parameters](#):

Key	Description
VERTICES	A new set of vertices (relative to the data reference time t0) on which to resample.
METHOD	The method by which to interpolate. Choose from • 'nearest' – nearest neighbour • 'linear' – linear interpolation • 'spline' – for piecewise cubic spline interpolation • 'pchip' – for shape-preserving piecewise cubic interpolation • 'v5cubic' – same as 'pchip' • 'cubic' – for the cubic interpolation from MATLAB 5

Example

Here we will interpolate a sinusoidal signal on to a new time-grid. The result will be to increase the sample rate by a factor 2.

First we create a time-series ao:

```
pl = plist('Name', 'None', '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.

```
tt   = linspace(0, x.nsecs - 1/x.fs, 2*(x.len));
```

Please notice the 'dot' notation we used to call a couple methods to access properties of this tsdata (time-series) AO:

```
x.nsecs
```

will provide a double with the duration of the timeseries, in seconds.

```
x.fs
```

will provide a double with the sampling rate of the timeseries, in Hz.

Finally, we can apply our new time-grid to the data using `interp`. We test two of the available interpolation methods:

```
pl_spline  = plist('vertices', tt);
pl_nearest = plist('vertices', tt, '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]));
```
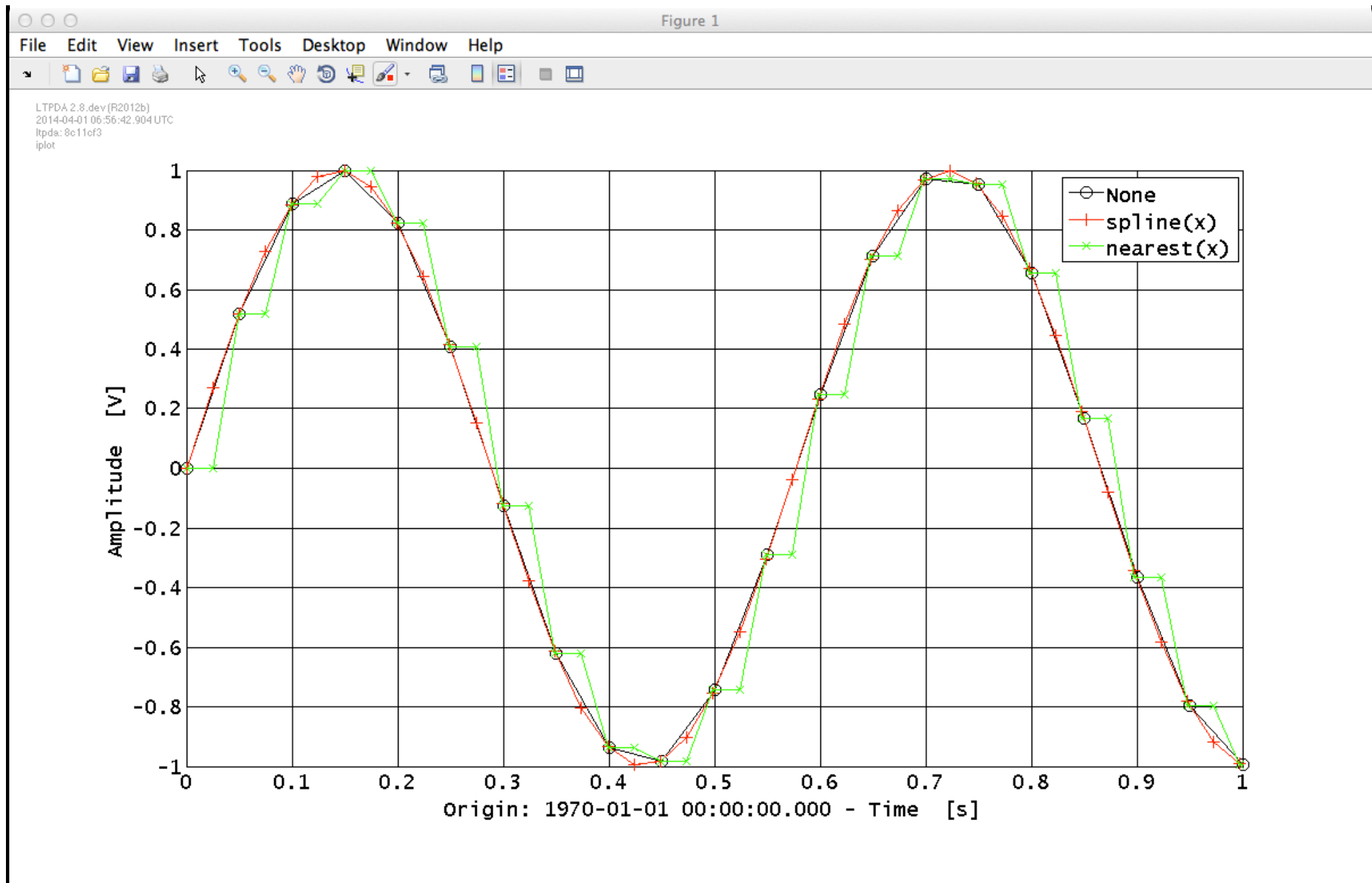
Notice the settings in the plist we used for the plotting, that allow changing the marker of the plotted lines, as well as the settings for adjusting the colors of the line themselves.

Remove trends from a time-series AO

The `ao/detrend` method offers the possibility to remove polynomial trends from a data series.

The method can be configured with the following [parameters](#):

Key	Description
N (or 'ORDER')	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 consisting of noise plus a known quadratic trend. We will then remove that trend using `ao/detrend` and compare the detrended time-series with the original noise.

First let's create the time-series series consisting of the noise plus trend.

```matlab
% Construct noise data stream
fs = 10;
nsecs = 10;
pl = plist('name', 'None', 'tsfcn', '5+randn(size(t))', 'fs', fs, 'nsecs', nsecs, 'yunits', 'V');
x  = ao(pl);
% Construct a quadratic data series
pl_trend = plist('tsfcn', 't.^2', 'fs', fs, 'nsecs', nsecs);
trend = ao(pl_trend);
% Add them together
fcn_trend = x + trend;
```

> The offset of 5 is added to the noise to ensure the data series doesn't come close to zero; we want to divide by it later in the example.

Next we will detrend the data and compare the result to the noise data `x` we made above.

```matlab
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', '-', '--'}.
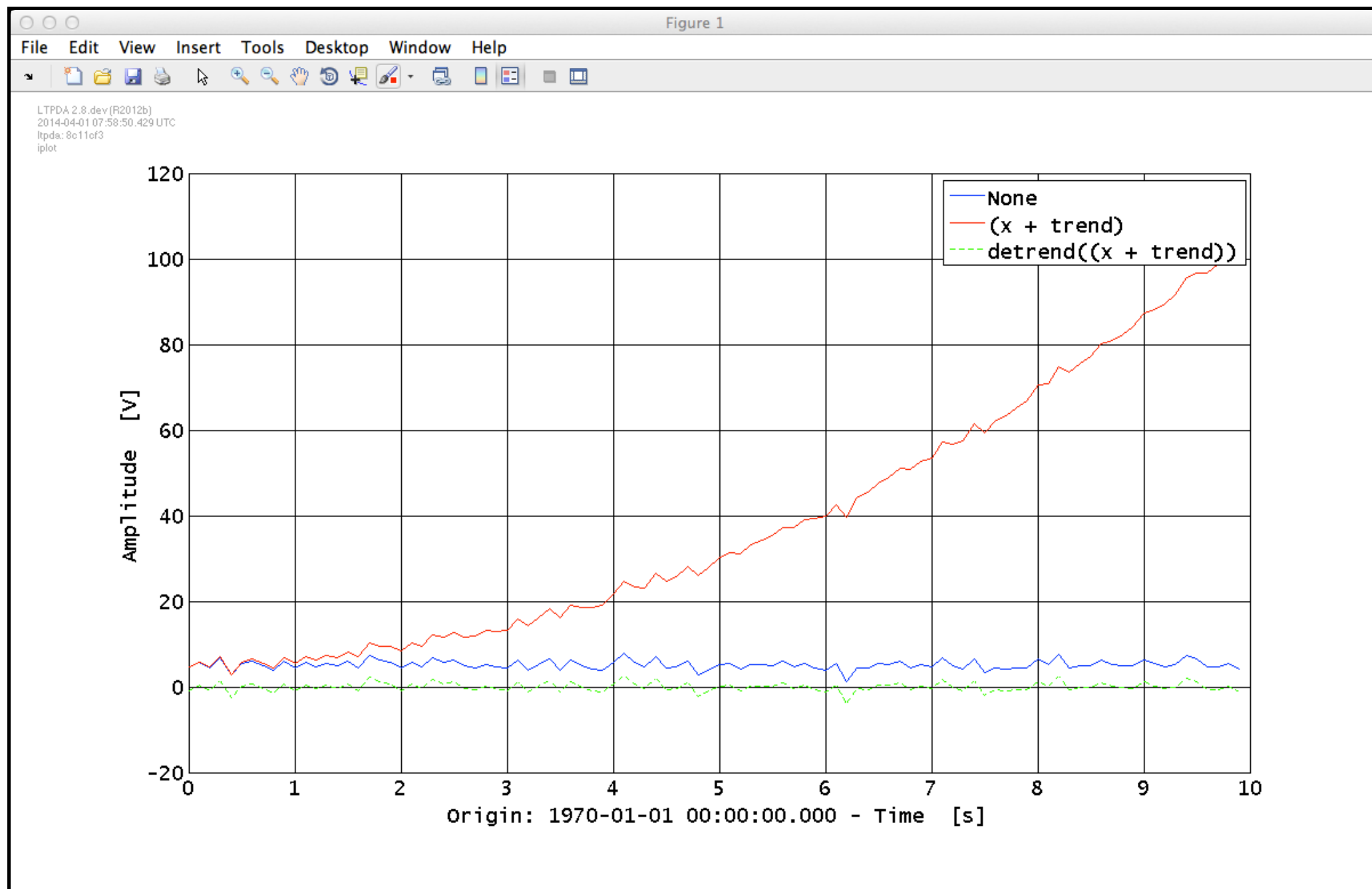
From this plot, it is not very easy to see how well our detrending worked. Let's form the fractional difference of the original x data and the detrended

data and plot that instead.

```
detr5 = detr + 5;
differ  = 100 .* (x - detr5) ./ x;
iplot(differ);
```

The result is shown below. We added the value 5 to the detrended time-series just to ensure that we don't divide by any values close to zero.
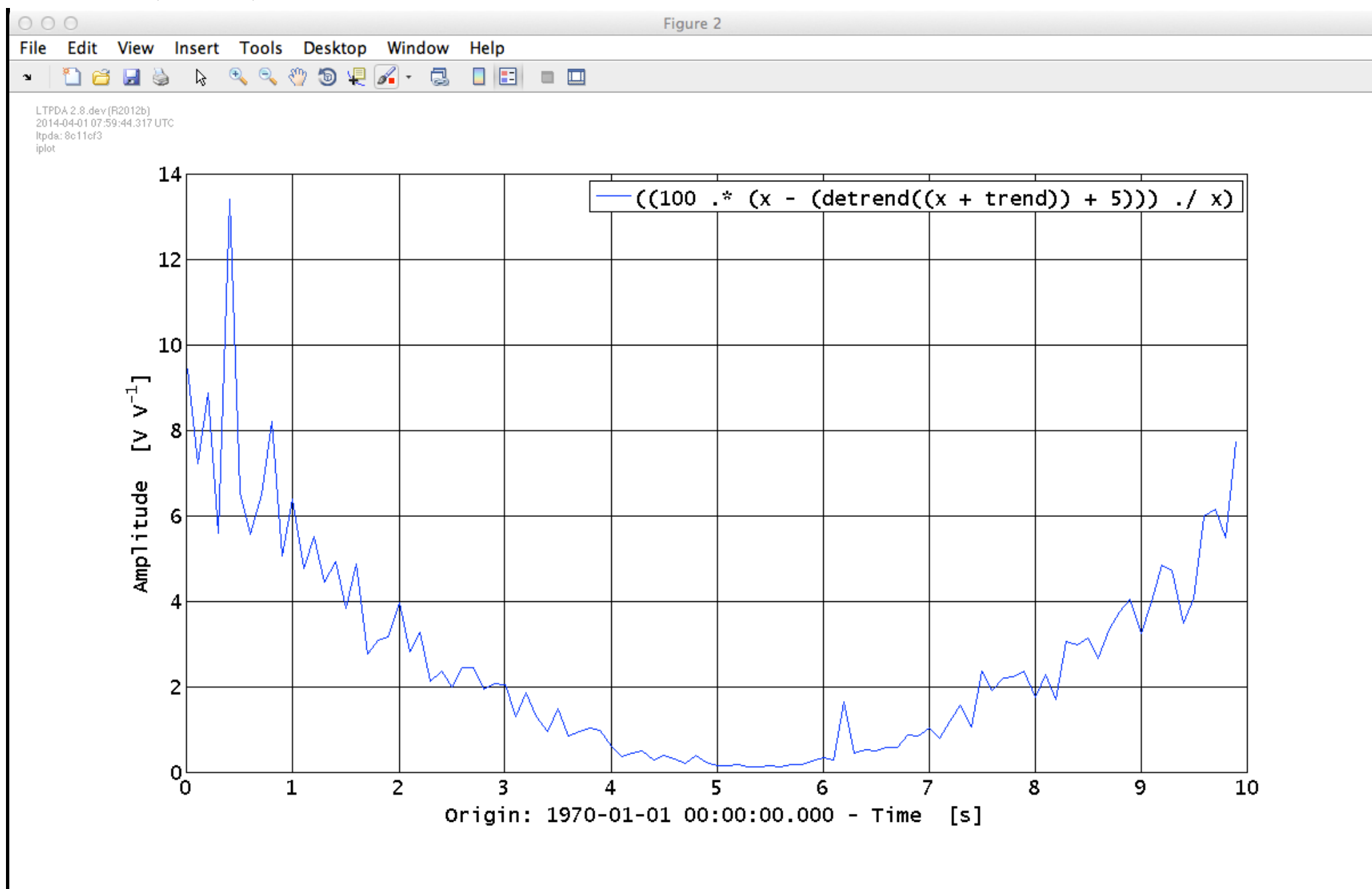
Try increasing the length of the data series to say, 1000 or 10000 seconds, to see how the detrending improves.

The value of the coefficient describing the subracted trend are included in the field `procinfo` of the `ao` objects. The `procinfo` is actually a `plist` object, so

we can search for parameters, in this case the key is 'coeffs':

```
c = find(detr.procinfo, 'coeffs');
% Remember also, that you will loose the 'coeffs' if you make any operation on the "detr"
```

◀ Interpolation of a time-series AO   Whitening noise ▶

©LTP Team

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 using the method `ao/filter`. 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, re-bins the spectrum so to reduce the individual points fluctuations, and fits a model of the spectrum as a series of partial fractions z-domain filters.

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 $10^{\wedge}(-b)$, 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);
```

The result should be similar to:
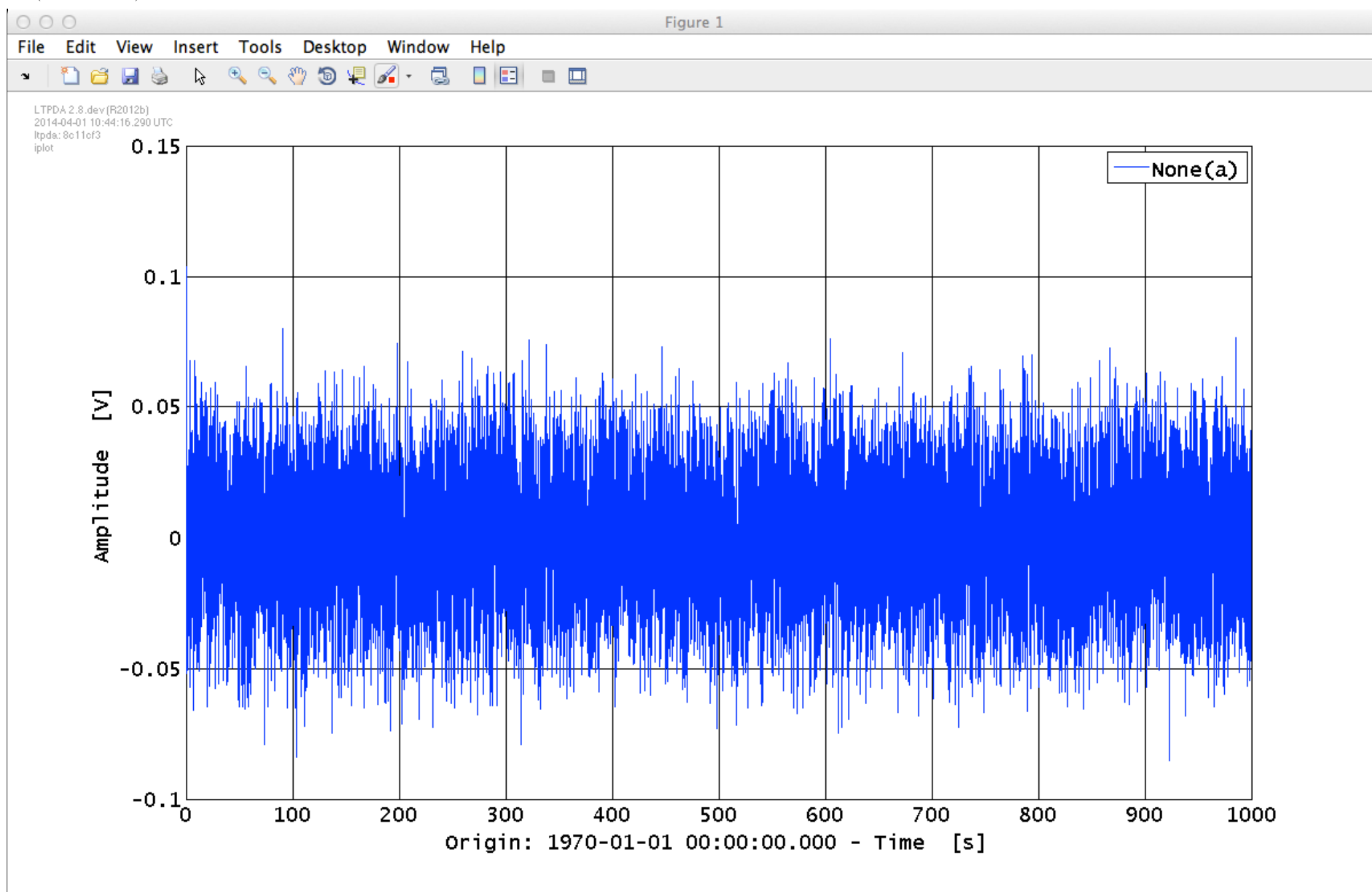
Before we can whiten the data, we have to define the parameter list for the whitening tool:

```
pl = plist(...
           'Plot', true, ...
           'MaxOrder', 9, ...
```

```
                    'Weights', 2);
```

Now we can call the whitening function `whiten1D` with our input AO, `a` and the parameter list `pl`:

```
>> aw = whiten1D(a, pl);
```

To compare the whitened data with the coloured noise we compute the power spectrum (for details see [Power spectral density estimation](#)):

```
awxx = aw.lpsd;
axx  = a.lpsd;
```

and finally plot our result in the frequency domain; in particular we plot the whitened data (`awxx`) compared to the coloured noise that was our input (`axx`). Please notice the choice of showing the whitened and colored data in two independent subplots, so to exploit their different range of values. Additionally, the units of the whitened data are reset to empty by the method, since the 'application' of the units is delegated to the whitening filter which is calculated as an intermediate step of the process.

```
iplot(axx, awxx, plist('arrangement', 'subplots'));
```

Figure 2

File   Edit   View   Insert   Tools   Desktop   Window   Help

LTPDA 2.8.dev (R2012b)
2014-04-01 11:04:32.711 UTC
ltpda: 8c11cf3
iplot

⬅Remove trends from a time-series AO   Select and find data from an AO➡
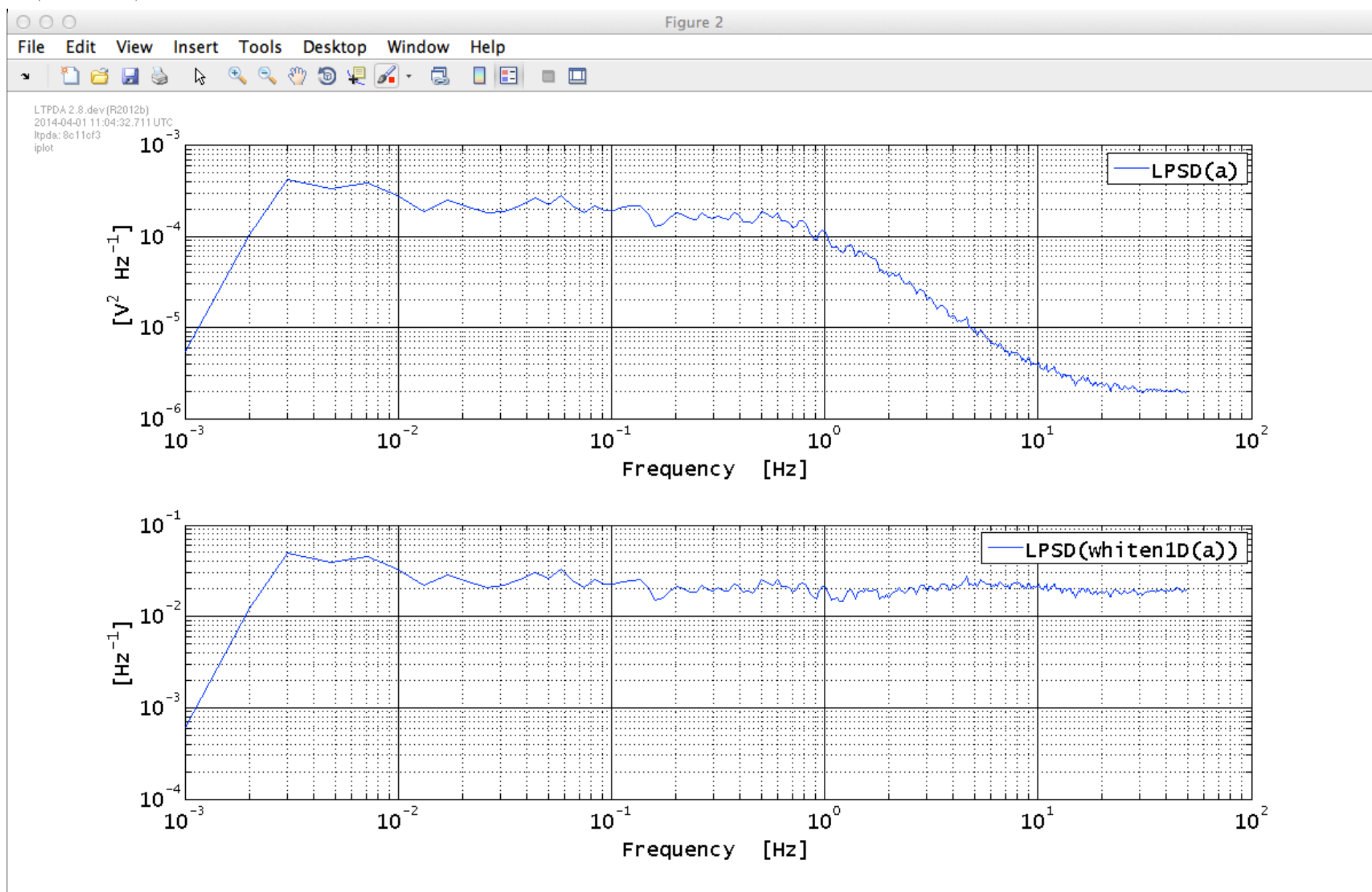
Select and find data from an AO

LTPDA contains a set of methods that can be used for the selection of data from an AO. In this section we will in particular look at `ao/find` and `ao/select`.

We will start by generating a sine wave:

```
pl = plist('waveform', 'sine wave', 'f', 1, 'fs', 100, 'nsecs', 10, 'yunits', 'V');
a  = ao(pl)
```

Example 1 – using `find`

Now let us use the `find` method to extract parts of the data we are interested in.

The `find` method can take a [parameters](#) 'query' for defining which data points you want to find. The query string can be any valid MATLAB logical expression, and in particular can be expressed in terms of the x and y data of the input AO.

In this example, we want to find all x values between 3 and 6. The following code does just that:

```
a_find = find(a, plist('query', 'x>3 & x<6'));
iplot(a, a_find)
```
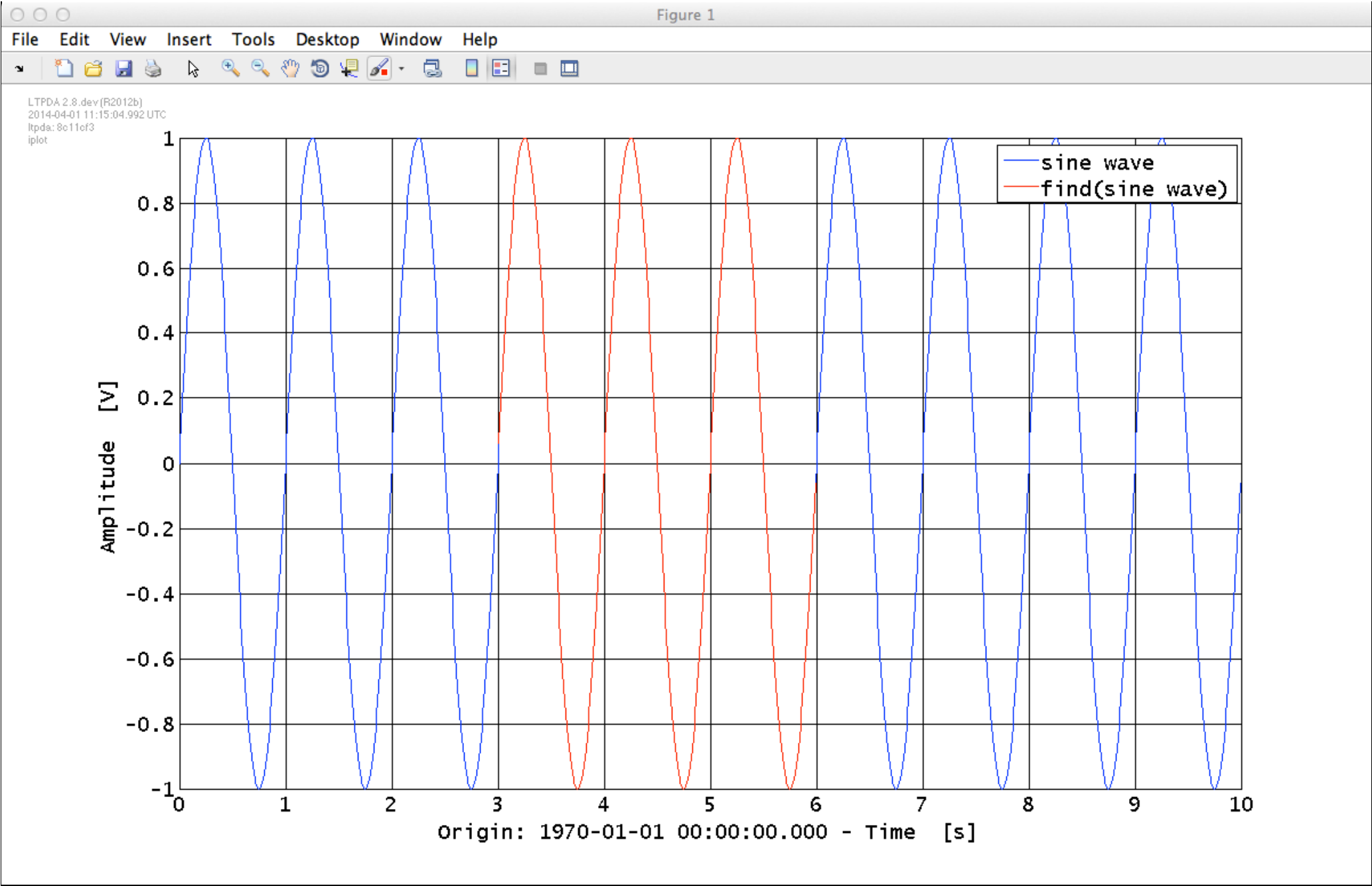
Example 2 – using `select`

The `select` method lets us select a set of data samples from our AO. For this we need to specify a [parameter](#) via 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)
```

Select and find data from an AO (LTPDA Toolbox)
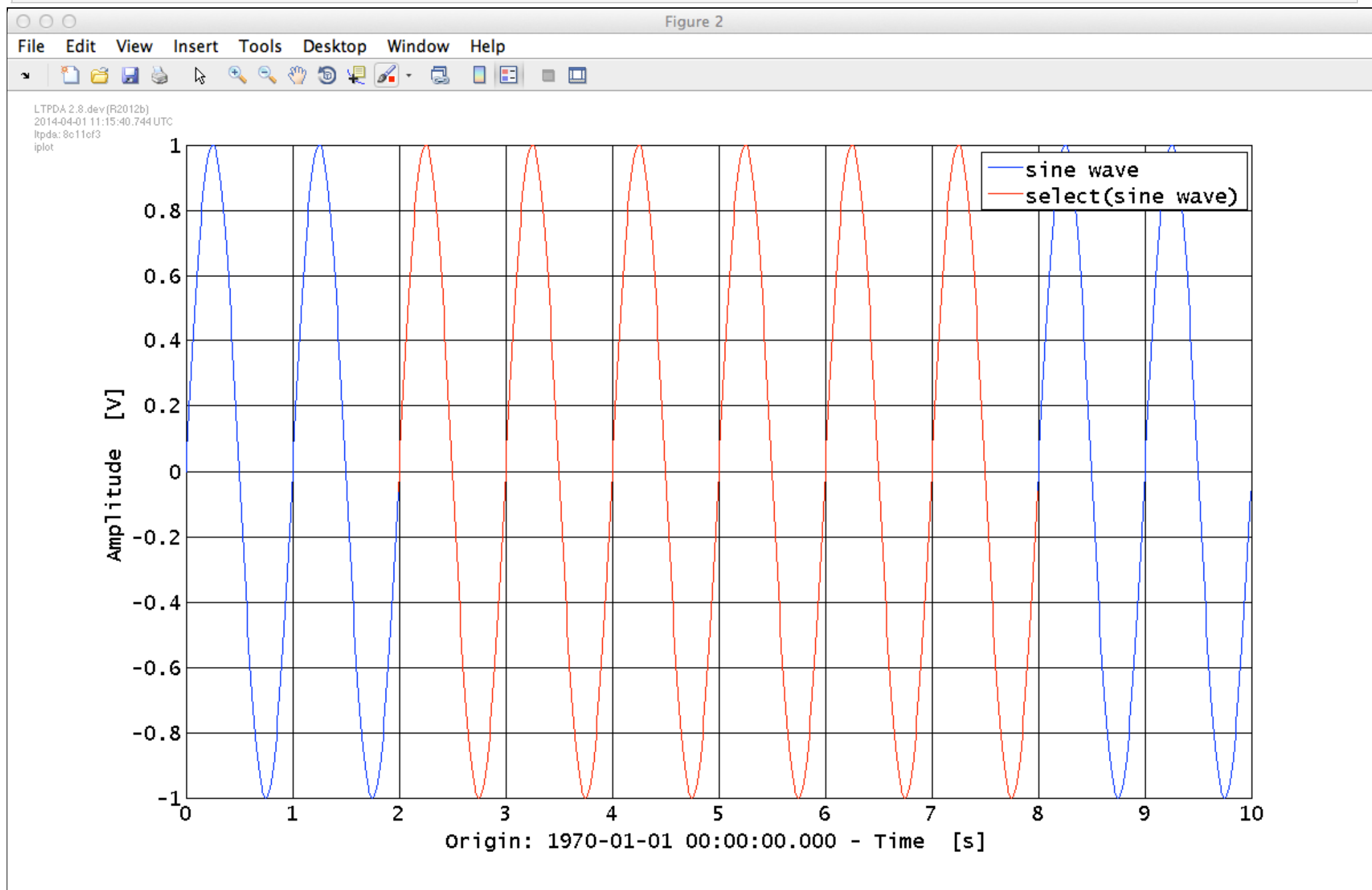
Split and join AOs

You can split the data inside an AO to produce one or more output AOs. The `ao/split` method splits an AO by samples, times (if the AO contains time series data), frequencies (if the AO contains frequency data), intervals, or a number of pieces. We can control this as usual by defining our parameters. This is a very flexible method, so take your time and check all its [possibilities](#).

Split by times

Let us create a new time series AO for these examples.

```
pl = plist('name', 'None', 'nsecs', 10, 'fs', 1000, 'tsfcn', 'sin(2*pi*7.433*t) + randn(size(t))', 'yunits', 'V');
a = ao(pl);
```

For splitting in time we need to define a time vector for the parameter list and pass it to `ao/split`:

```
pl_time = plist('times', [2 3]);
a_time  = split(a, pl_time);
iplot(a, a_time)
```
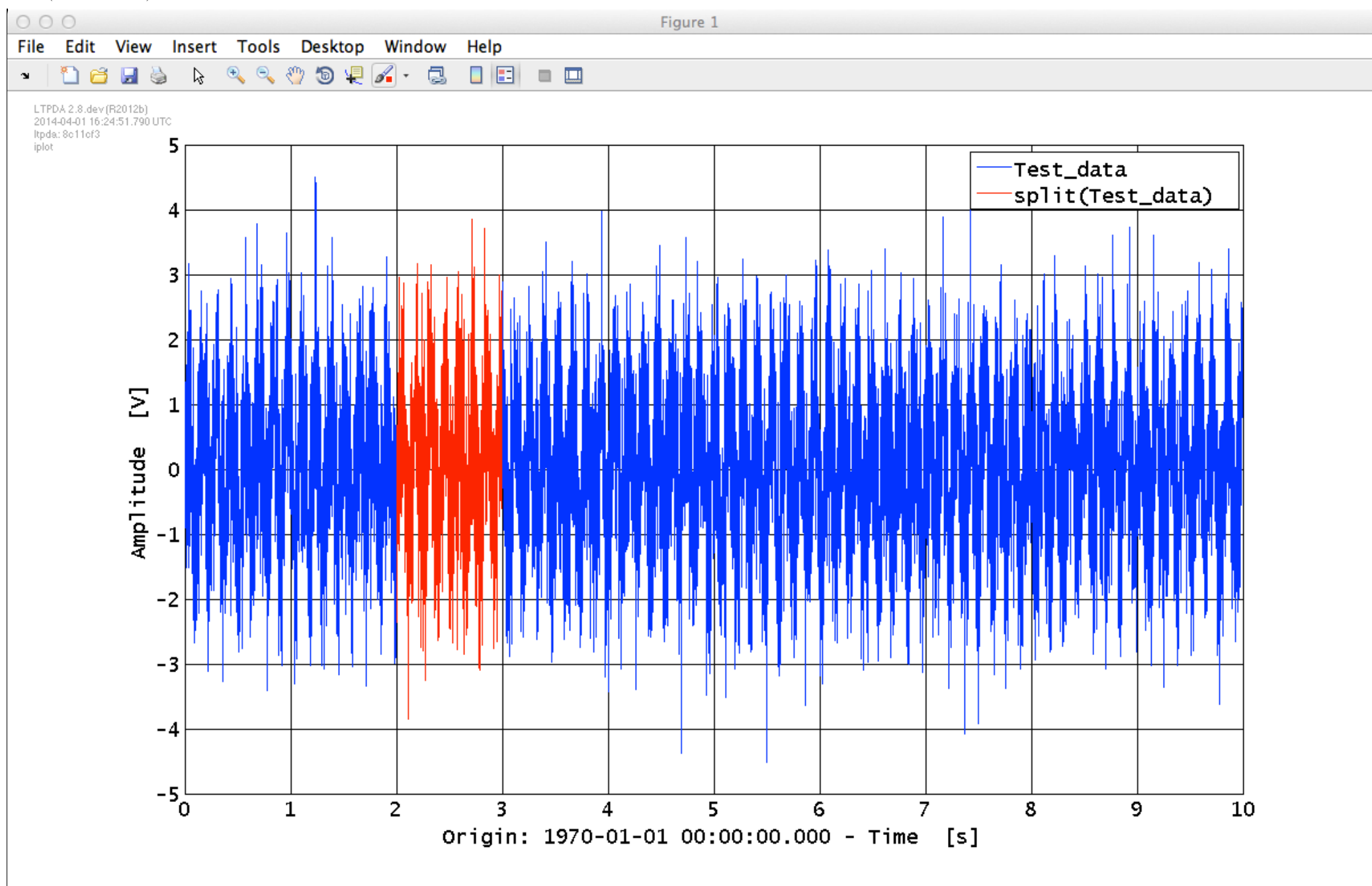
Split by frequencies

For this we need a frequency data AO. One easy way to get this is by computing the power spectrum using `ao/psd`.

```
axx = a.psd;
```

Again we need a vector for the parameter list and pass it to `ao/split`:

```
pl_freq  = plist('frequencies', [10 100]);
axx_freq = split(axx, pl_freq);
iplot(axx, axx_freq)
```
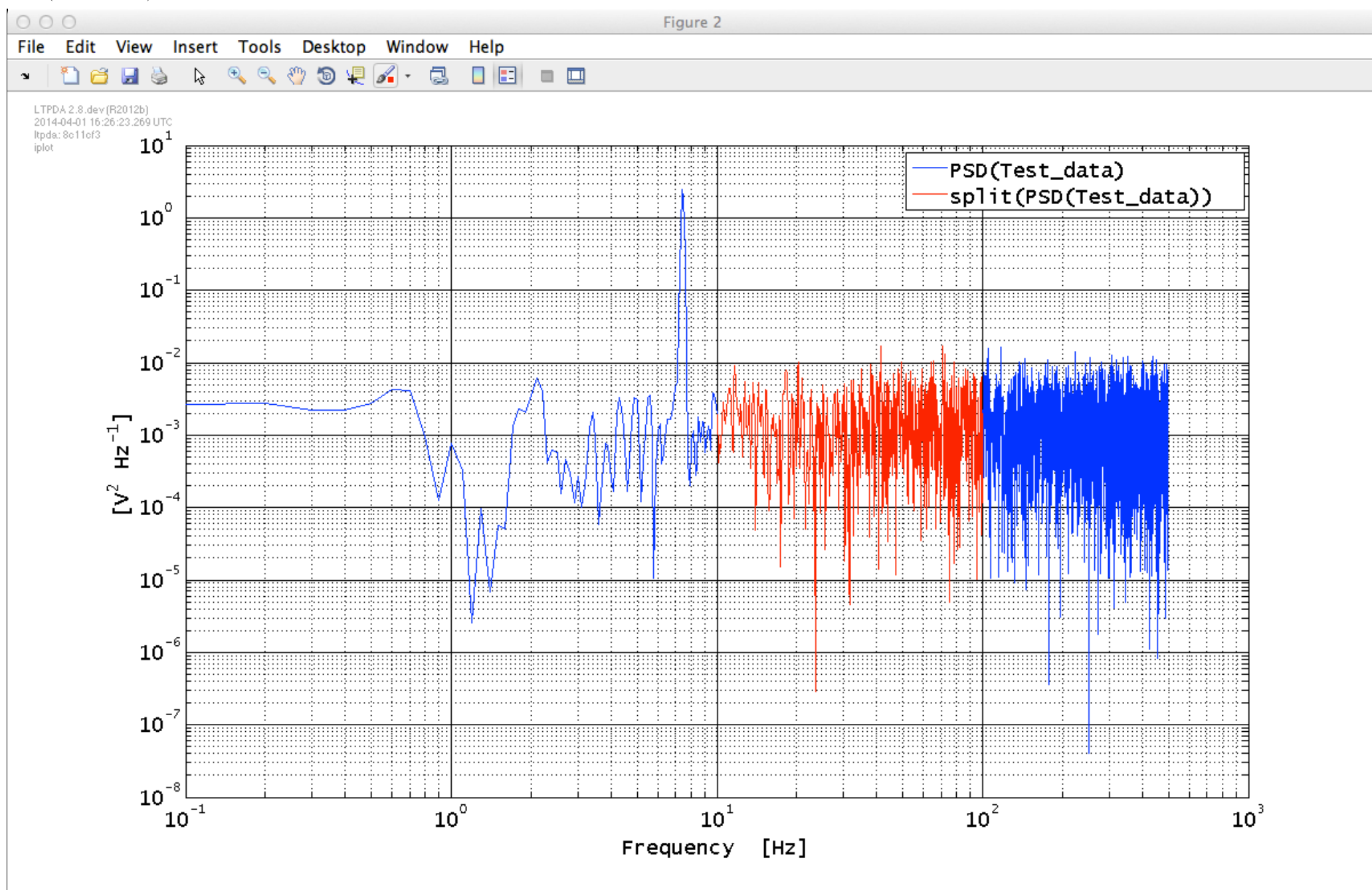
Split by intervals

We can also split the AO by passing a time interval to the `ao/split` method:

```
pl_interv = plist('start_time', 4, 'end_time', 6);
a_interv = split(a, pl_interv);
iplot(a, a_interv)
```
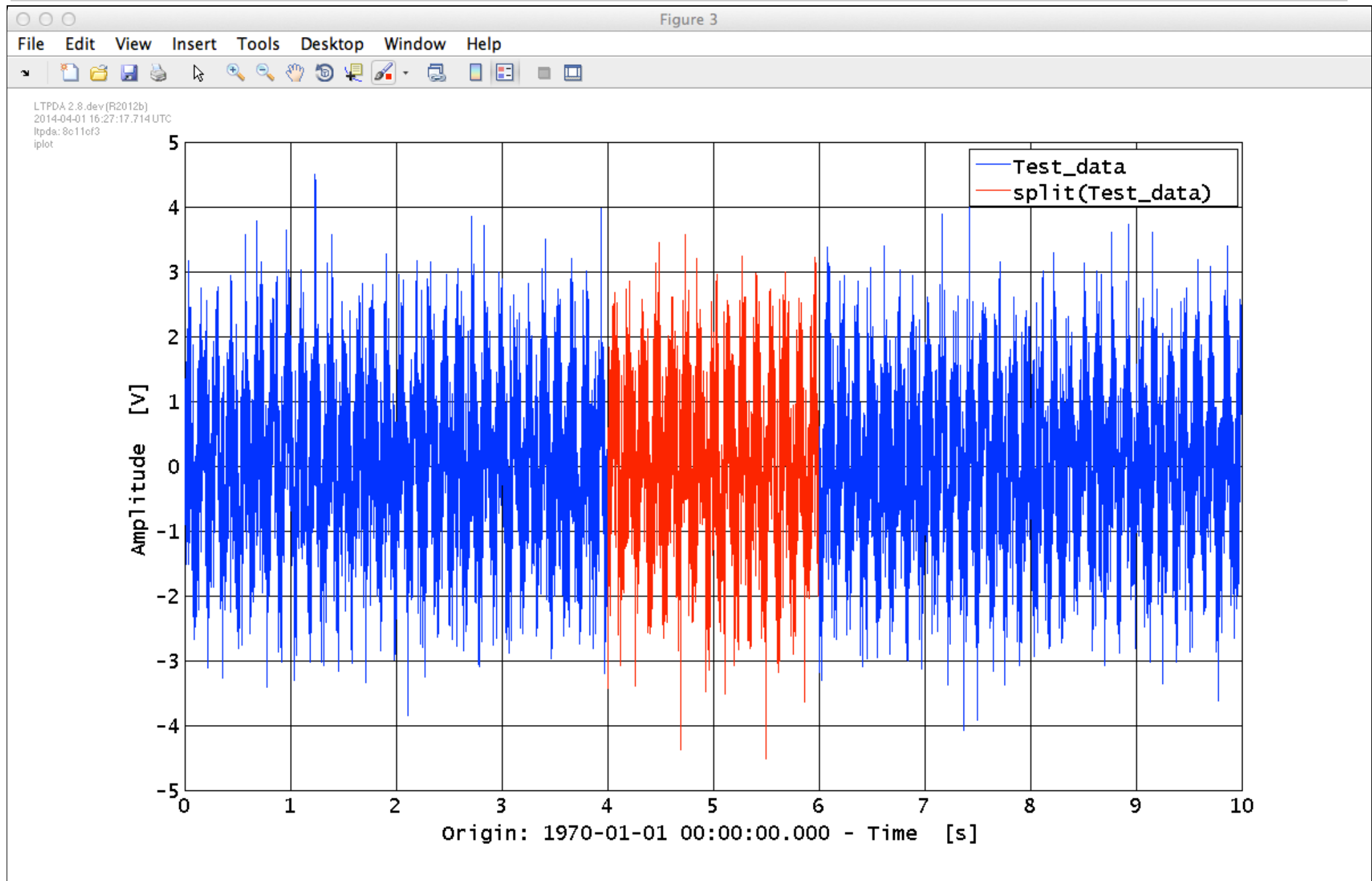

Split by samples

This type of splitting method we can use on any type of data. Let us use the frequency type, `axx`.

Again we need a vector for the parameter list and pass it to `ao/split`, only that this time we will split our AO in to two parts.

```
pl_samp = plist('samples', [50 100 101 300]);
[axx_samp1 axx_samp2] = split(axx, pl_samp)
iplot(axx, axx_samp1, axx_samp2)
```
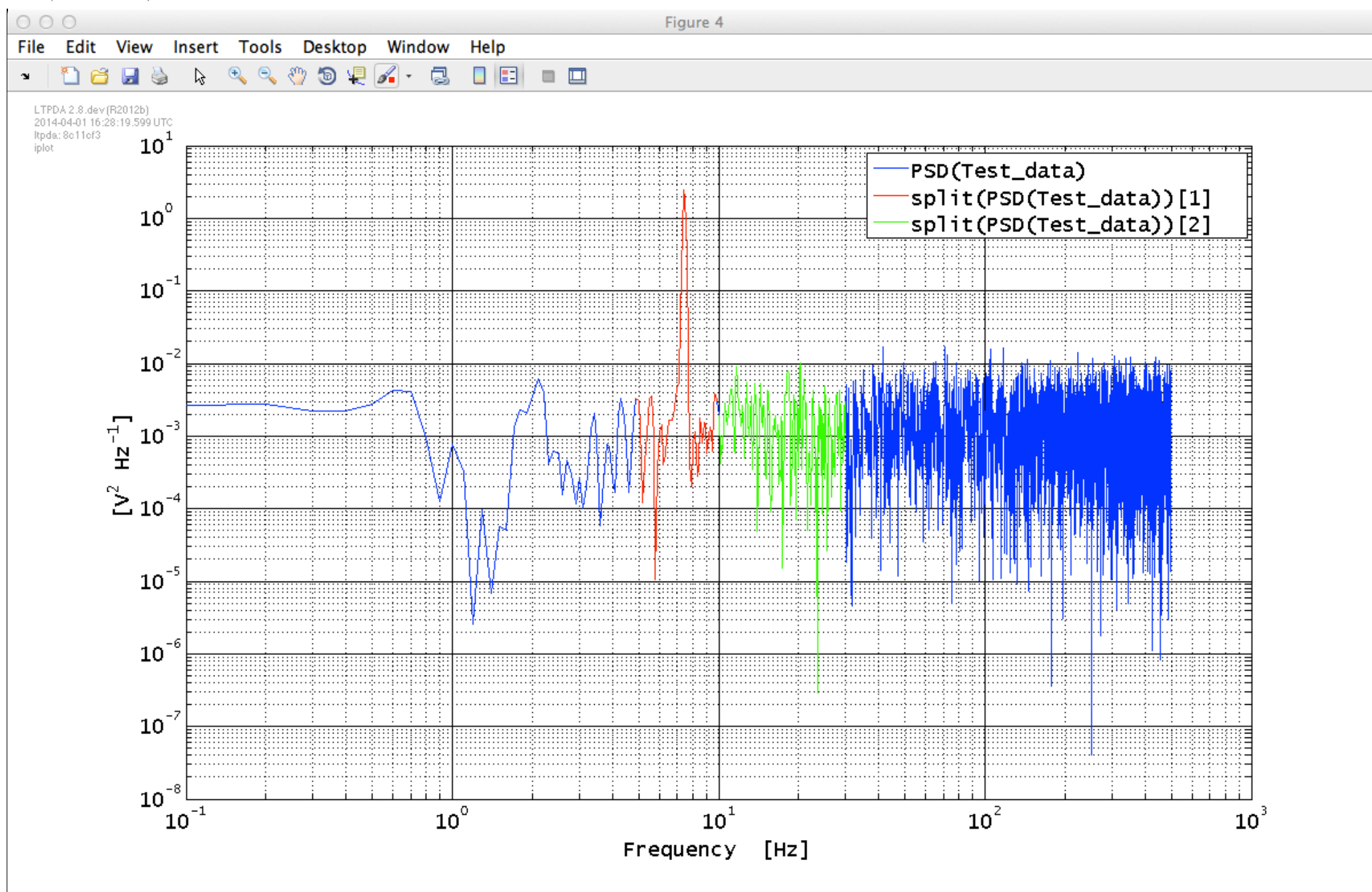
Although in this example the two resulting AOs are contiguous, they need not to be.

Join AOs

We can join our two AOs back together using `ao/join`:

```
axx_join = join(axx_samp1, axx_samp2);
iplot(axx, axx_join)
```
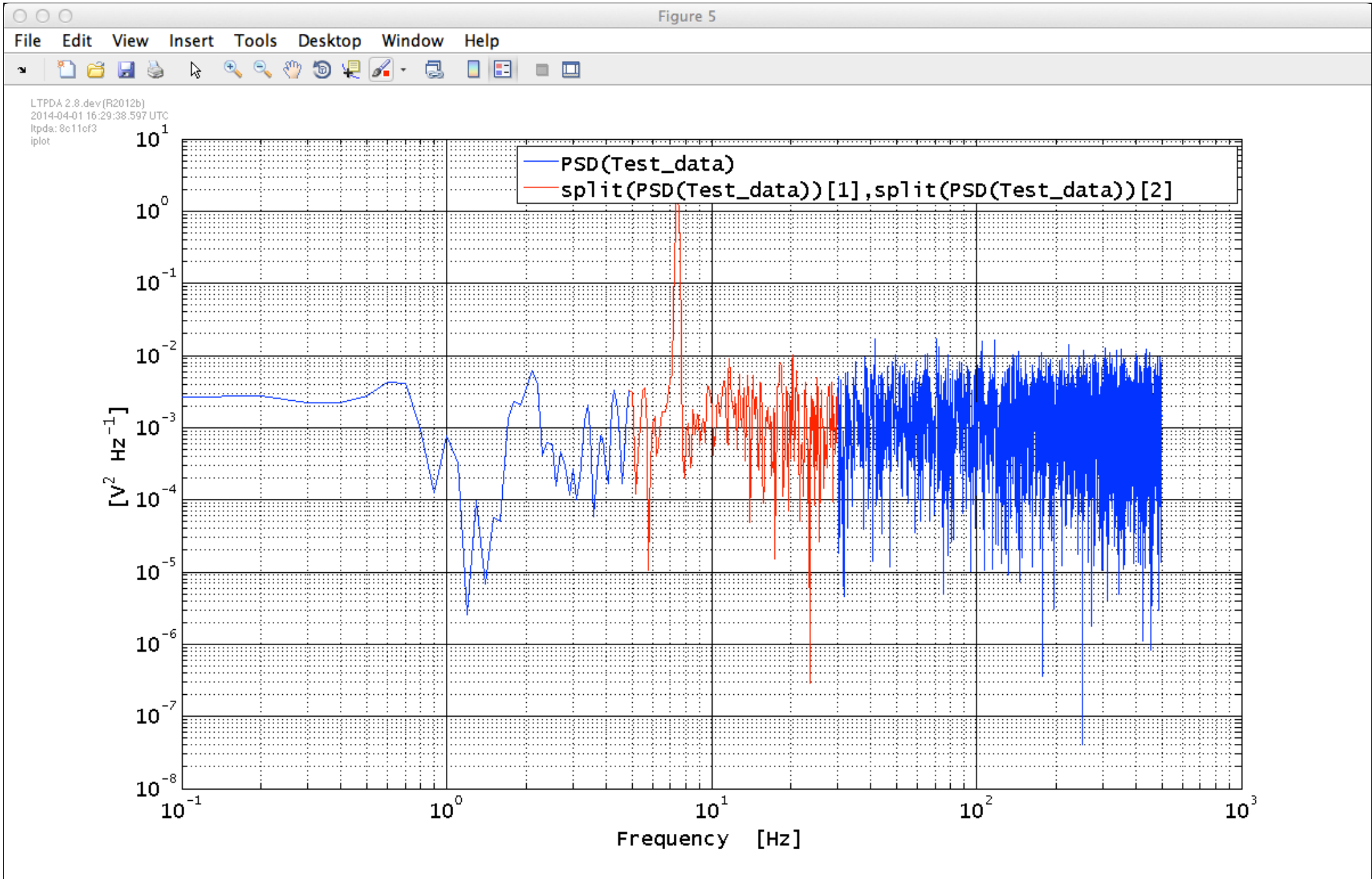

If we look at the history for `axx_join` (by entering `axx_join.viewHistory`), we will see the following:

History Graph for [split(PSD(Test_data))[1],split(PSD(Test_data))[2]/141ead33-b0b6-47ae-bc8d-fdff77614ab0]
Generated from terminal
Created 2014-04-01 16:30:44.812 UTC

ao.ao

ao(TSFCN='sin(2*pi*7.433*t)
+ randn(size(t))', FS=1000,
NSECS=10, XUNITS=[s], T0='1970-01-01
00:00:00.000', TOFFSET=0,
YUNITS='V', DY=[], RAND_STREAM=
1x1 [struct], NAME='Test_data',
DESCRIPTION='', TIMESPAN=[])

xspec

psd(NFFT=10000, WIN=(BH92,
length=10000, alpha=0, psll=
92, rov=66.1, nenbw=2.0044,
w3db=1.8962, flatness=-0.8256),
PSLL=200, OLAP=66.099999999999994,
ORDER=0, NAVS=-1, TIMES=[],
SCALE='PSD')

split

split(SAMPLES=[101 300])   split(SAMPLES=[50 100])

join

join(ZEROFILL=0, SAMET0=0,
FSTOL=9.9999999999999995e-07,
SORT=1)

END

Since the two AOs that are output from the 'split by samples' stage are independent, the history tree reflects this, showing two independent branches leading to the `join` step.

IFO/Temperature Example – Pre-processing

Now we return to the IFO/Temperature example that was started in Topic 1.

Loading and checking the calibrated data sets from topic2

In the last topic you should have saved your calibrated data files as
`ifo_temp_example/ifo_disp.xml` and
`ifo_temp_example/temp_kelvin.xml`

Now load each file into an AO, and simplify the name of the objects by assigning them the name of the Matlab workspace variable containing them:

```
    ifo  = ao('ifo_temp_example/ifo_disp.xml');
    temp = ao('ifo_temp_example/temp_kelvin.xml');
    ifo.setName;
    temp.setName;
```

Let's see what kind of pre-processing we have to apply to our data prior to further analysis.

You can have a look at the data by for example displaying the AOs on the terminal and by plotting them, of course. Since the two data series have different Y units, we should plot them on subplots. To do that with `iplot`, pass the key 'arrangement' in a plist. For example:

```
    pl = plist('arrangement', 'subplots');
```

If you plot the two time-series you should see something like the following:
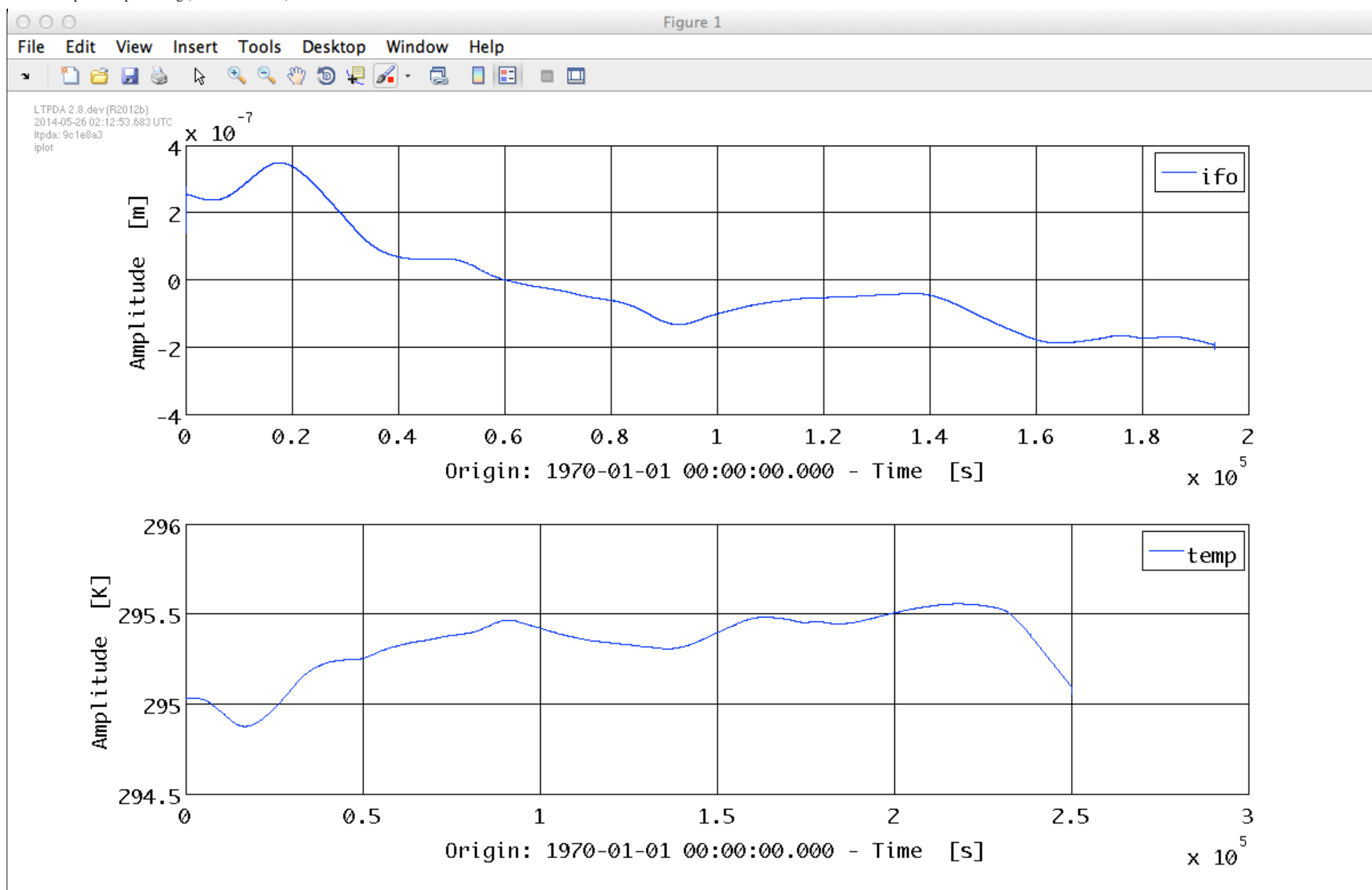
Some points to note:

1. The two data streams:

- - do not have the same sampling frequency.
  - are not of the same length (nsecs).
2. The interferometer data has a small transient at the start

To have a closer look at the samples we plot markers at each sample and only plot a zoomed-in section. You can do this by creating a parameter list with the following key/value pairs (remember the 'key' properties for `plist` entries is not case sensitive):

Key	Value
ARRANGEMENT	'subplots'
LINESTYLES	{'none', 'none'}
MARKERS	{'+', '+'}
XRANGES	{'all', [200 210]}
YRANGES	{[2e-7 3e-7], [200 350]}

Notice the use of the keyword 'all' in the value for the 'XRANGES' parameter. Many of the `iplot` options support this keyword, which tells `iplot` to use the same value for all plots and subplots. For the 'YRANGES' we specify different values for each subplot.

Please store the parameter lists for plot settings in 2 different variables. We can reuse them for plotting our results later.

Passing such a parameter list to `iplot` together with the two AO time-series should yield a plot something like:
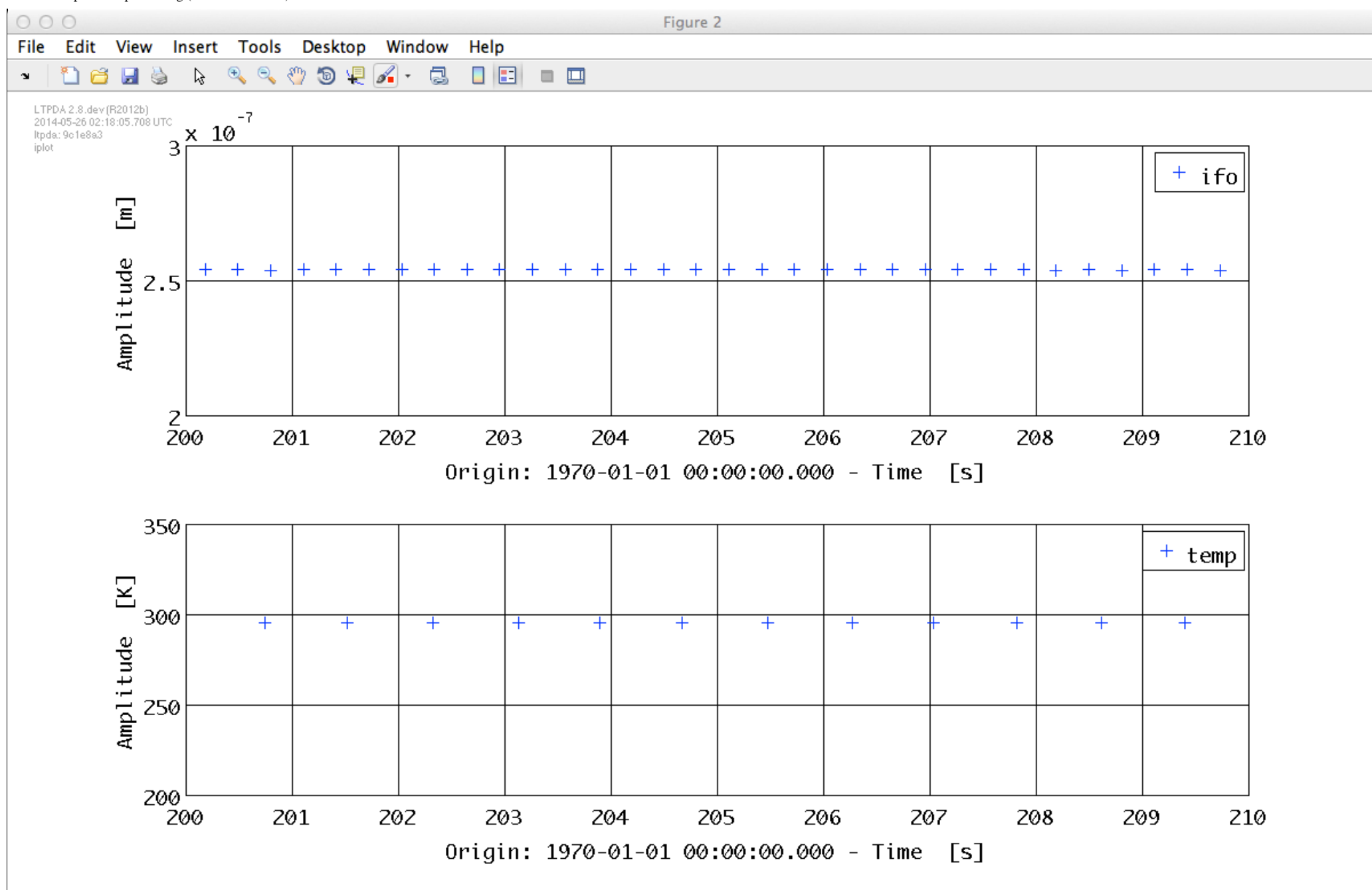
From this plot you may be able to see that the temperature data is unevenly sampled.

To confirm this, enter the following (standard) MATLAB commands on the terminal:

```
dt = diff(temp.x);
min(dt)
max(dt)
```

Don't forget the semicolon at the end of the `diff` calculation; this is a long data series and will be printed to the terminal if you do forget.

You see that the minimum and maximum difference in the time-stamps of the data is different, showing that the data are not evenly sampled.

Before we proceed with the later analysis of this data, we need to

- Fix the uneven sampling of the temperature data
- Resample both data streams to the same rate
- Resample both data streams on to the same timing grid
- Select the segment of interferometer data that matches the temperature data

Each of these steps can, in principle, be done by hand. However, LTPDA provides a 'data fixer' method called `ao/consolidate` which attempts to automate this process. The call to `consolidate` is shown below:

```
[temp_fixed ifo_fixed] = consolidate(temp, ifo, plist('fs', 1));
```

We tell `consolidate` that we want to have our data resampled to 1 Hz by specifying the parameter key 'fs'.

Now we can inspect the time-series of these data. The result should look something like the figure below:
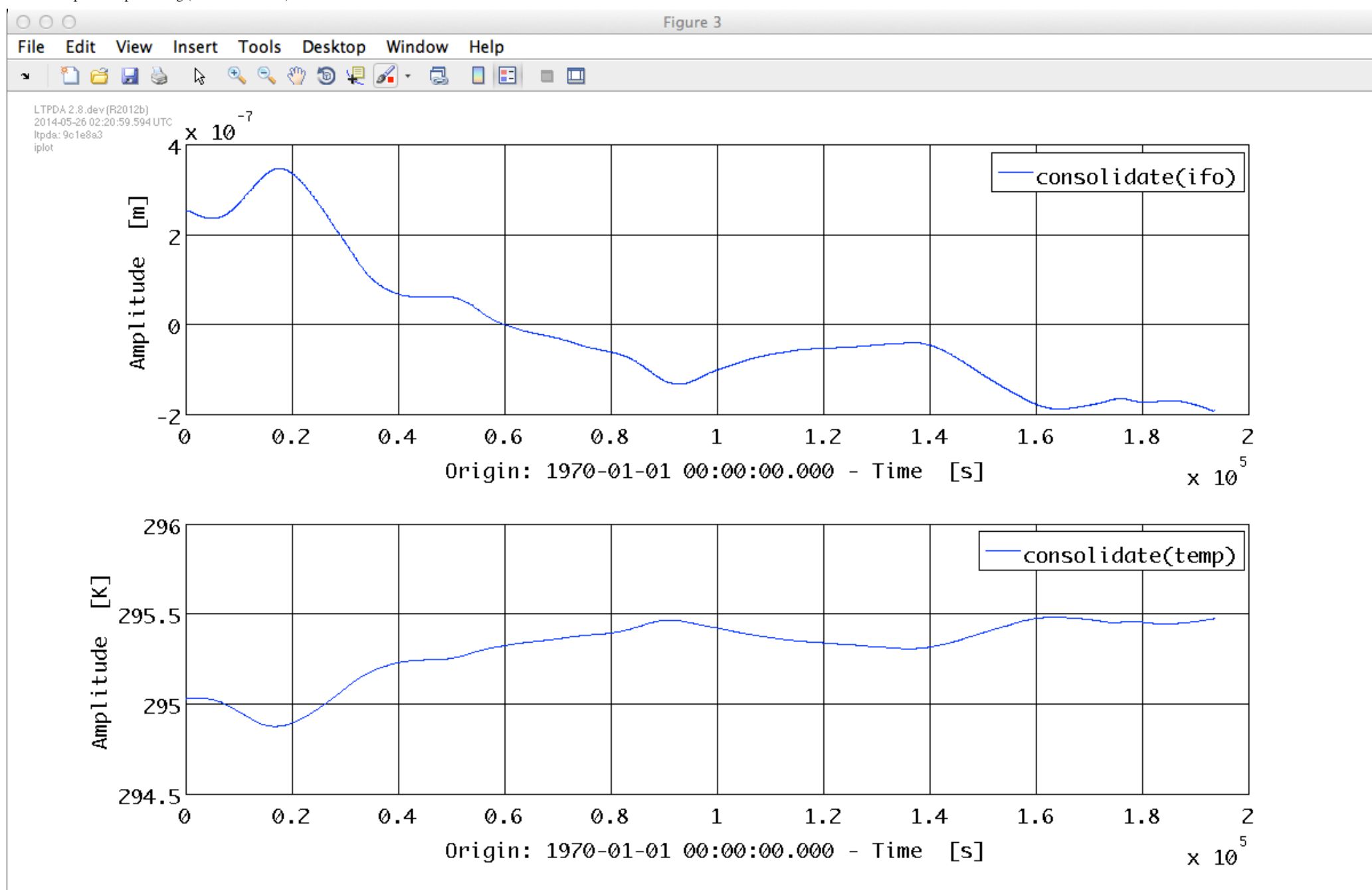
If we also plot the zoomed-in view again, we should see something like:
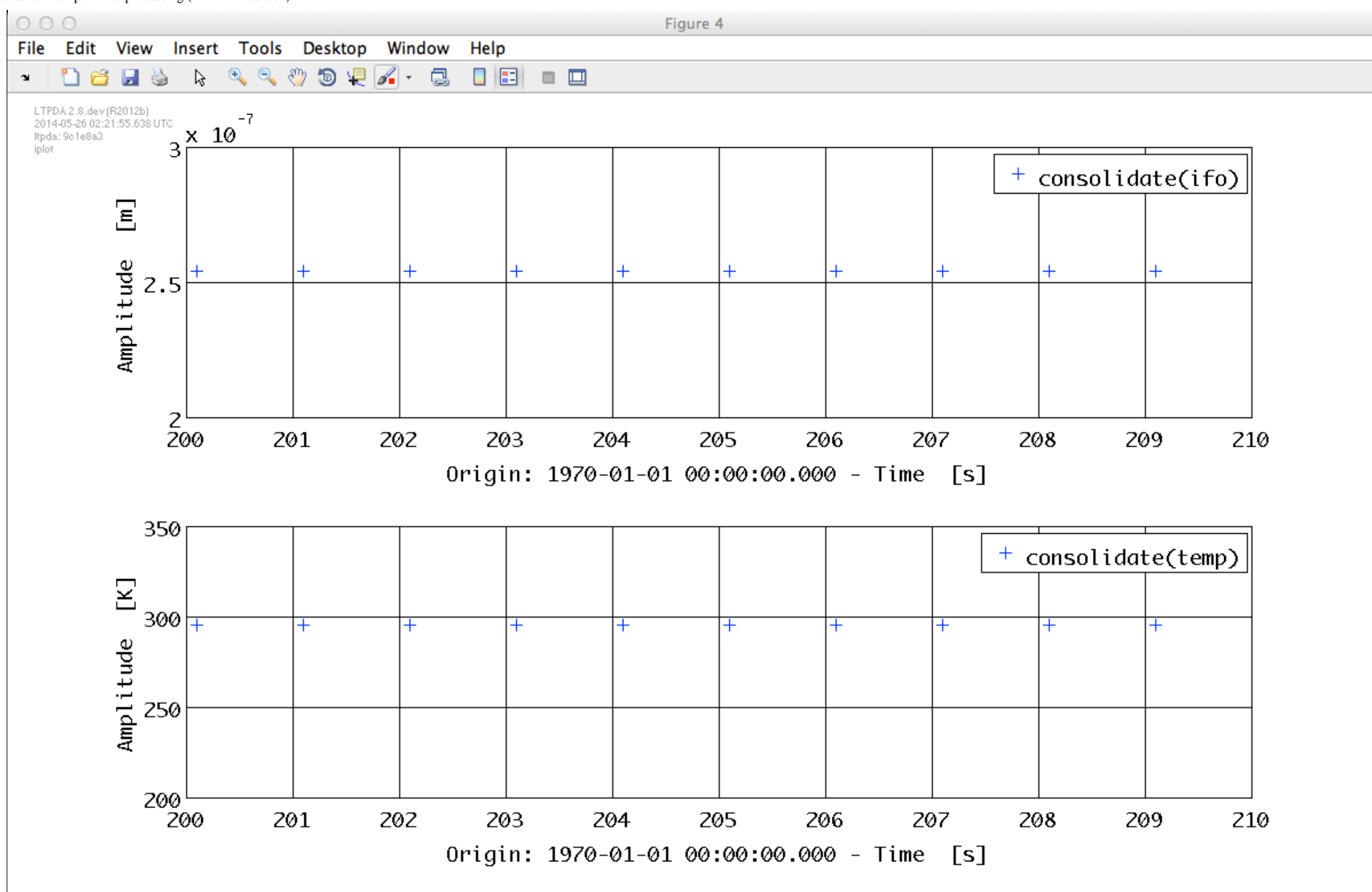
As you can see `consolidate` solved all our issues with these two data streams. They now start at the same time and are evenly sampled at the same sampling frequency.

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 ➡

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Topic 3 – Spectral Analysis

Training session 3 is a tutorial of how to estimate spectral properties of the signals, employing the instruments provided by the LTPDA Toolbox. As described in the devoted User Manual [section](#), spectral estimation is a branch of the signal processing, performed on data and based on frequency-domain techniques.

The focus of the tools is on time-series objects, whose spectral content needs to be estimated.

We will learn here how to use the tools to evaluate univariate and multivariate analyses.

The topic is divided as follows:

- [Exercises on Power Spectral Density estimation](#)
- [Exercises on Transfer Functions estimation](#)
- [IFO/Temperature Example – Spectral Analysis](#)

©LTP Team

Power Spectral Density estimation

In this subsection we will focus on the evaluation of the PSD (Power Spectral Density) for a given time-series signal. The functionality is provided by a method of the `ao` class called

`ao/psd`

which implements the Welch method of averaging modified periodograms (also referred to as WOSA). More details can found in the dedicated [section] of the user manual.

In this tutorial, we propose the following exercises based on the estimation of the PSD of suitable time-series data:

1. [Simply PSD]: a very basic starting exercise
2. [Windowing data]: introducing the usage of segment averaging/windwing/detrending
3. [Log-scale PSD on MDC1 data]: a "realistic" example from the first LTPDA Mock Data Challenge, employing the log-scale PSD estimation method

◀ Topic 3 – Spectral Analysis   Example 1: Simply PSD ➡

©LTP Team

Power Spectral Density estimation (LTPDA Toolbox)

| **LTPDA Toolbox™** | contents | ← → |

Example 1: Simply PSD

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:
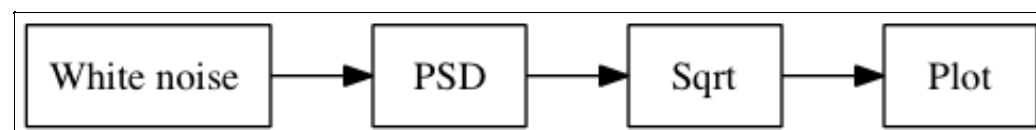
Simulate a time-series of white noise data

There are many different ways to simulate a white noise time series data. Here we choose a pretty powerful one: the "From Waveform" AO constructor.

Let's start building a plist to tune the parameters of the `ao` constructor: among the available constructor methods, we choose the noise generator via the key `WAVEFORM` assigning the value "noise". Then let's make the time series last longer by setting the number of seconds (`nsecs`) to 1000.

```
pl_noise  = plist('waveform', 'noise', 'fs', 1, 'nsecs', 1000);
```

Eventually, we set the units of the noise to be `meters` by selecting means of the plist method `pset`: the "Yunits" parameter and entering `'m'`.

```
pl_noise.pset('yunits', 'm');
```

To actually build the time-series AO, we then call the `ao` constructor with the plist we just prepared:

```
n = ao(pl_noise);
```

Evaluate the Power Spectrum of the data

The next step is to choose the parameters for the `psd` method of the `ao` class. In order to know the key name, the description, the default and possible

values for the parameters, we can type

```
help ao/psd
```

and then click the "Parameters Description" link. Let's discuss the parameters and their meaning:

- `Nfft` allows to choose the number of samples in each periodogram evaluation
- `Win` allows to choose the type of spectral window to be employed to reduce the edge effects at beginning/end of the data sections.
- `Psll` allows to choose the peak-sidelobe level in the case of Kaiser windows.
- `Olap` allows to choose the percentual overlap between subsequent segments
- `Order` allows to choose the degree of detrending applied to each segment prior to windowing
- `Navs` allows to choose the number of averages to perform for each ferquency bin
- `Times` allows to choose a reduced set of the inputs by asking the are within a given time range
- `Scale` allows to choose the quantity to be sent in output (ASD, PSD, AS, PS)

If the user does not specify any value for the parameters, the routine applies the default values.

In this case, we will use the default parameters:

- `Nfft` -1 so to estimate the PSD only on one window, including the whole data set
- `Win` = The value set by the user in the preferences
- `Psll` = 200 in the case of Kaiser window
- `Olap` = -1 so the overlap will be chosen based on the windows properties
- `Order` = 0 so to remove the mean value from the data before applying the window
- `Navs` = -1 so to set the lenght of the window (all data, in this case) rather than asking for a given number of means at each frequency
- `Times` = [] so to analyze all the input data
- `Scale` = 'PSD' so we evaluate the Power Spectral Density [output units] will be [input units]^2 / Hz

In this case, we would not need any specification plist, but let's prepare one. The we can go ahead and do the calculation:

```
pl_psd = plist('Scale', 'PSD');
s = psd(n, pl_psd);
```

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 for the `ao/psd` method 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` step to the flow by applying it to the output of the `ao/psd` method.

```
a = sqrt(s);
```

Plot the results

What remains is just to plot the calculated square root of the PSD of the input white noise. Just call `ao/iplot`. The output one should be similar to the following:

Example 1: Simply PSD (LTPDA Toolbox)
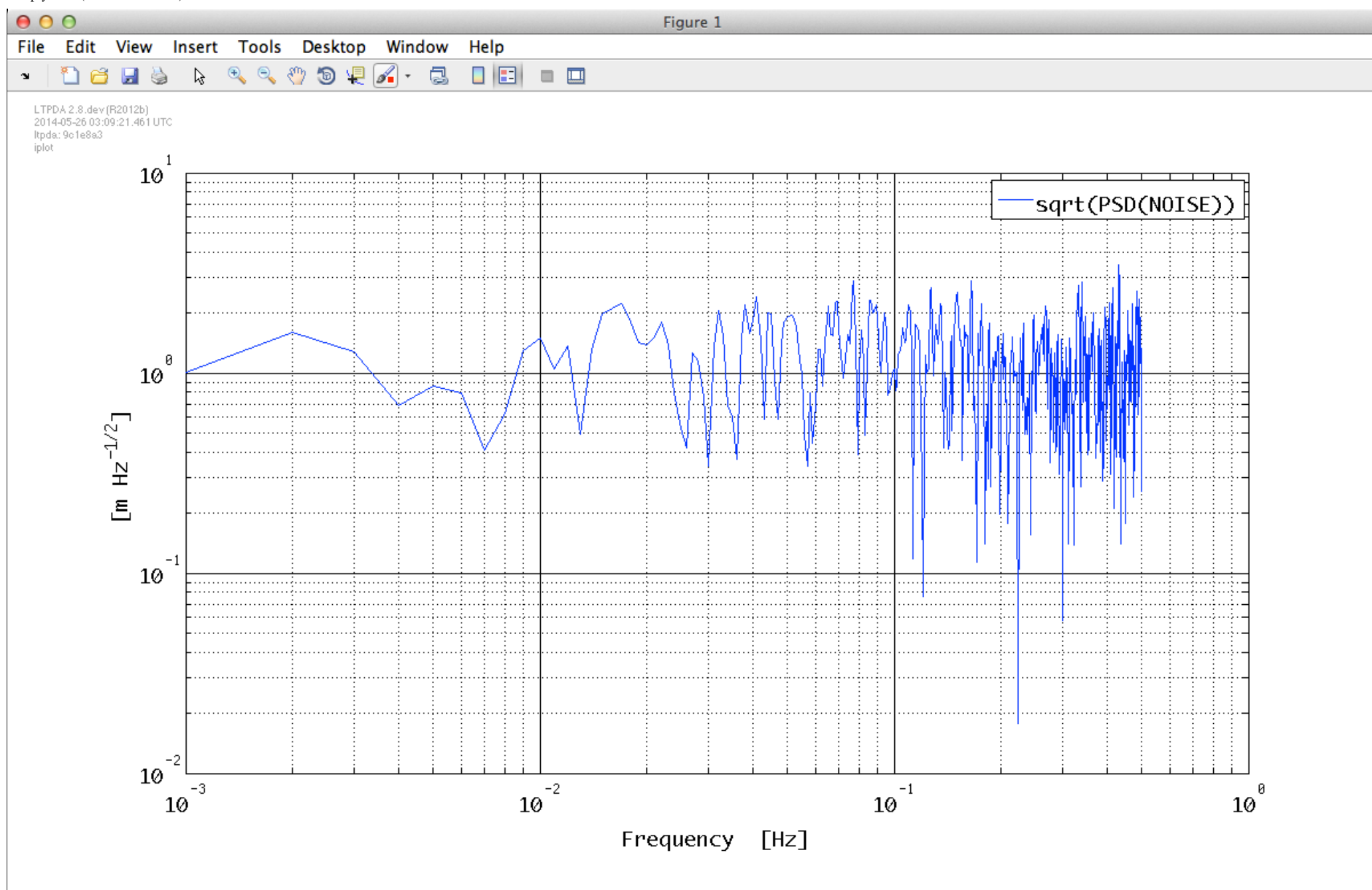
Please notice:

- The log-log shape of the plot
- The x-axis units

- The y-axis units
- The history steps reported on the legend

©LTP Team

Example 2: Windowing data

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:
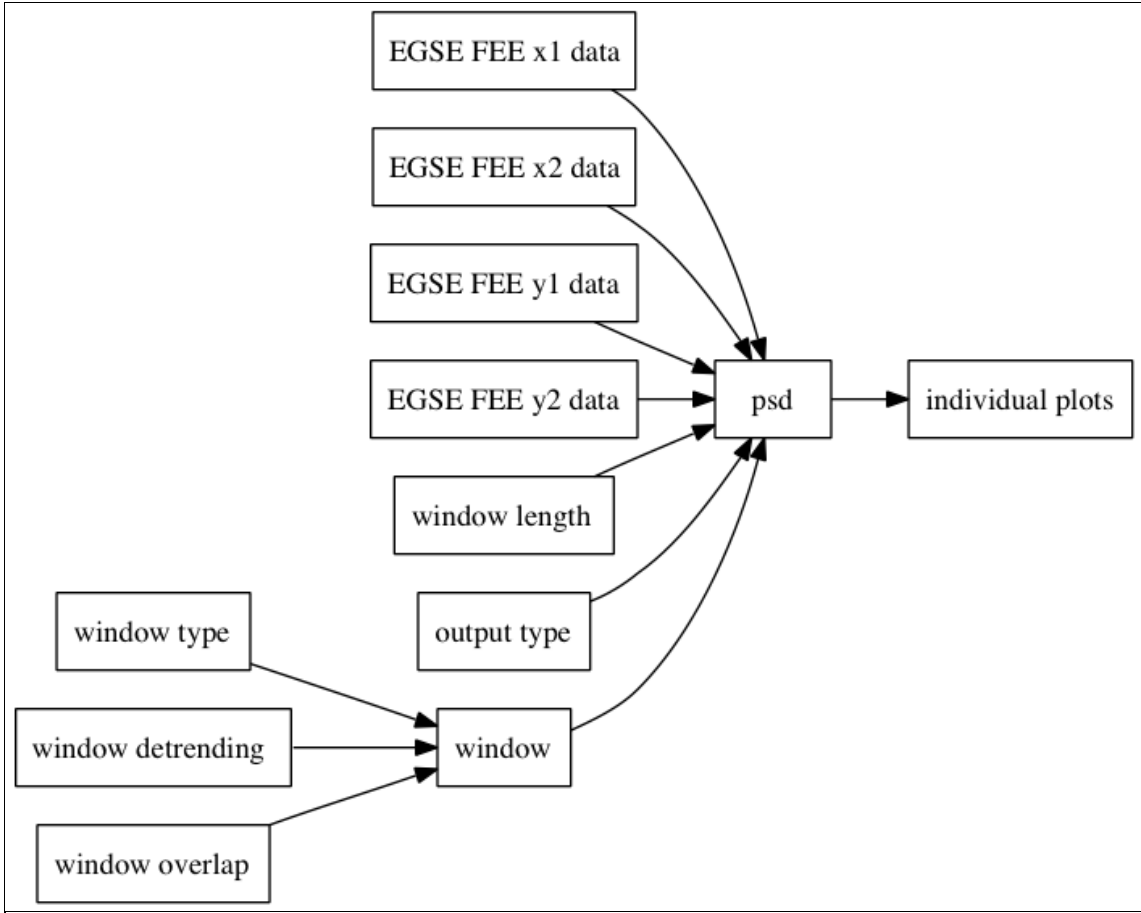
Loading experimental data time-series from data files

The step of building `ao`s by loading data from files was touched upon in [previous steps](#) and in the [user manual](#). Here we go ahead by calling the `ao` constructor method, that we want to specify by means of a plist, with parameters:

Key	Value	Description
FILENAME	'topic3/EGSE_FEE_x1.dat'	The name of the file to read the data from.
TYPE	'tsdata'	Interpret the data in the file as time-series data.

COLUMNS	[1 2]	Load the data x-y pairs from columns 1 (as x) and 2 (as y).
XUNITS	's'	Set the units of the x-data to seconds (s).
YUNITS	'F'	Set the units of the y-data to farad (F).
COMMENT_CHAR	'%'	Indicateds which header lines to skip in the ASCII data file.
FS	[]	Indicates to load time series from the first data column.
ROBUST	'yes'	Use robust data reading for this file format.
DESCRIPTION	'EGSE FEE x1 data'	Set some text to the 'description' field of the AO.

This procedure can be repeated for all 4 channels we want to analyze, just by calling the `ao` method and the `plist/pset` method:

```
FEE_x1 = ao(pl_data);
FEE_x2 = ao(pl_data.pset('FILENAME', 'topic3/EGSE_FEE_x2.dat'));
FEE_y1 = ao(pl_data.pset('FILENAME', 'topic3/EGSE_FEE_y1.dat'));
FEE_y2 = ao(pl_data.pset('FILENAME', 'topic3/EGSE_FEE_y2.dat'));
```

Evaluate the Power Spectrum of the different data sets

Now let's proceed by adding the `psd` step to the diagram, as we did previously. Here we want to call the method only once, without the need of calling the function 4 times; 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 pass all the inputs to the `psd` method:

```
[S_x1, S_x2, S_y1, S_y2] = psd(FEE_x1, FEE_x2, FEE_y1, FEE_y2);
```

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.
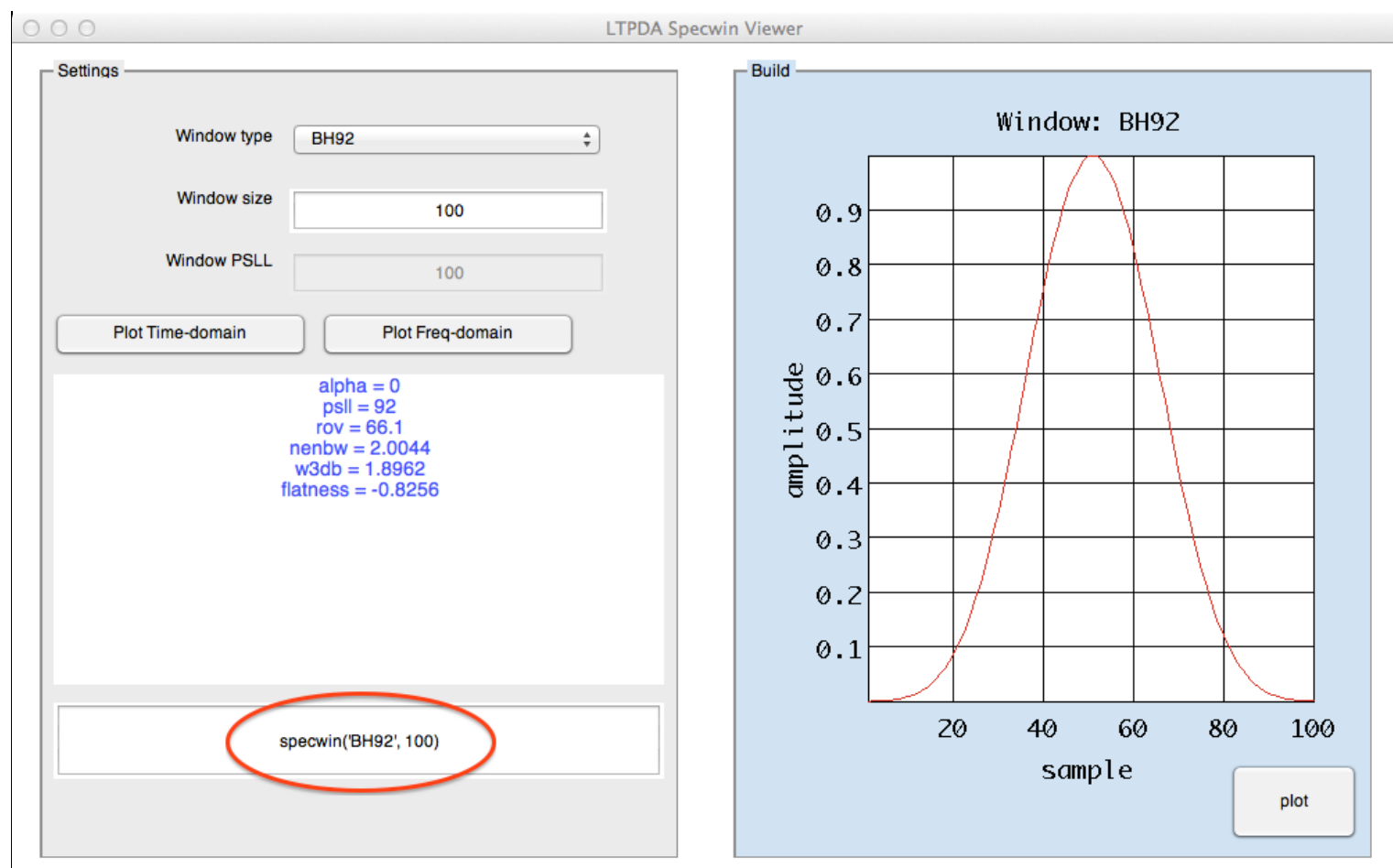
Once we are happy with the choice, we can go apply it to the data, by specifying the value for the Key `win` via a parameter list for the `psd`. One parameter is alreay selected, based on the user preferences. Here I choose to use the Blackman–Harris window, called 'BH92'.

```
pl_psd = plist('win', '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):

```
pl_psd.pset('nfft', 2e4);
```

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 (meaning we leave the default values of −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 set the value corresponding to the "Scale" entry to be the 'ASD'.

```
pl_psd.pset('scale', '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 let's just enter 1 for the value of the key "Order".

```
pl_psd.pset('order', 1);
```

We can now go ahead and call the method. In this case, since we pass multiple inputs, we can decide what to do with the outputs (we will get one output AO per input AO). Let's keep all of them into a single array:

```
S = psd(FEE_x1, FEE_x2, FEE_y1, FEE_y2, pl_psd);
```

Plotting the spectra

As per our choice, `psd` would give as an output an array of `ao`s corresponding to each input. So in our case, we'd have an array with 4 `ao`s. We can also make them individuals, by indexing the array:

```
S_x1 = S(1);
```

We can now go ahead and plot the individual `ao`s in many plots, just by calling `iplot`. 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]}

At the end we can look at the output plots and check the results, the units, the frequency range.

Saving the spectra

Once we obtained the objects we are interested on, we can go ahead and save at least the *x1* data as described here and here.

◀ Example 1: Simply PSD   Example 3: Log-scale PSD on MDC1 data ▶

©LTP Team

Example 3: Log-scale PSD on MDC1 data

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:
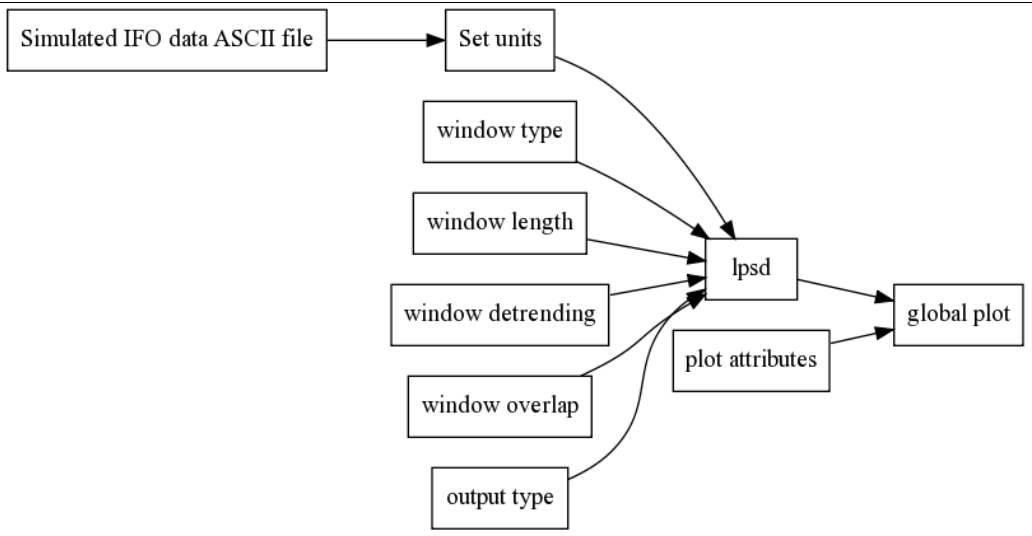
Load a time-series of IFO noise data simulated for the first LTPDA Mock Data Challenge (MDC1)

The step of building `ao`s by loading data from files was touched upon in [previous steps](#) and in the [user manual](#). Here we go ahead by calling the `ao` constructor method, that we want to specify by means of a plist, with parameters:

Key	Value	Description
FILENAME	'topic3/mockdata_16_48_17_11_2007_1.dat'	The name of the file to read the data from.
TYPE	'tsdata'	Interpret the data in the file as time-series data.
COLUMNS	[1 2 1 3]	Load the data x-y pairs from columns 1 (as x) and 2 (as

		y), in the first `ao`, and from columns 1 (as x) and 3 (as y), in the second `ao`.
XUNITS	's'	Set the units of the x-data to seconds (s).
YUNITS	''	We leave this empty for now.
COMMENT_CHAR	'%'	Indicateds which header lines to skip in the ASCII data file.
FS	[]	Indicates to load time series from the first data column.
ROBUST	'no'	We don't need robust data reading for these simulated data.
NAME	{'IFO x1', 'IFO x12'}	Specify the names of the objects.
DESCRIPTION	'MDC1 set #1, 17/11/2007'	Set some text to the 'description' field of the AO.

```
x = ao(pl_load);
```

The output will be a vector of `ao`s, containing:

- the *x1* IFO measurement (SC to TM1 *x* position)
- the *x12* IFO measurement (TM2 to TM1 relative *x* position)

We can then run all the analyses, and be sure to be applying the same parameters, by passing the vector of `ao`s to the various methods.

Set data units

We forgot to set the units ... luckily both displacements are expressed in meters, so let's call the method

```
ao/setYunits
```

The relevant parameters to set 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 use the `lpsd` method on the `ao` class. It needs to be specified by setting its parameters. Many 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)
- `Times` Allows to choose a reduced set of the inputs by asking the are within a given time range

In this case, we will use the following parameters:

Key	Value	Description
WIN	'BH92'	Or a different one, if you want.
OLAP	−1	Overlap will be chosen based on the window properties
ORDER	2	Segment-wise detrending up to order 2
SCALE	'ASD'	Evaluate the Amplitude Spectral Density so that [output units] will be [input units] / sqrt(Hz)

We also need to set other 2 parameters that are typical of this method, discussed in the devoted user manual [section](#).

- `'Jdes'` – the number of spectral frequencies to compute
- `'Kdes'` – the desired number of averages
- `'Lmin'` – the minimum segment length [default: 0]

We will act on the first one, so to decide how many ASD bins to estimate, leaving to the algorithm the choice of their location and the length of the windows on each bin.

Key	Value	Description
JDES	2000	Compromising execution time and resolution
KDES	10	Slightly less than the default value
LMIN	0	The default value

Plot the results

As we did in previous topics, let's add one `ao/iplot` step after the call to the `ao/lpsd` method. We make a little additional exercise: let's specify the color

via a plist with parameters:

Key	Value	Description
COLORS	{[0 0 1],[1 0 0]}	Setting *x1* to blue and *x12* to red

The execution may take some while ... and here are the results:

Example 3: Log-scale PSD on MDC1 data (LTPDA Toolbox)

Example 3: Log-scale PSD on MDC1 data (LTPDA Toolbox)

©LTP Team
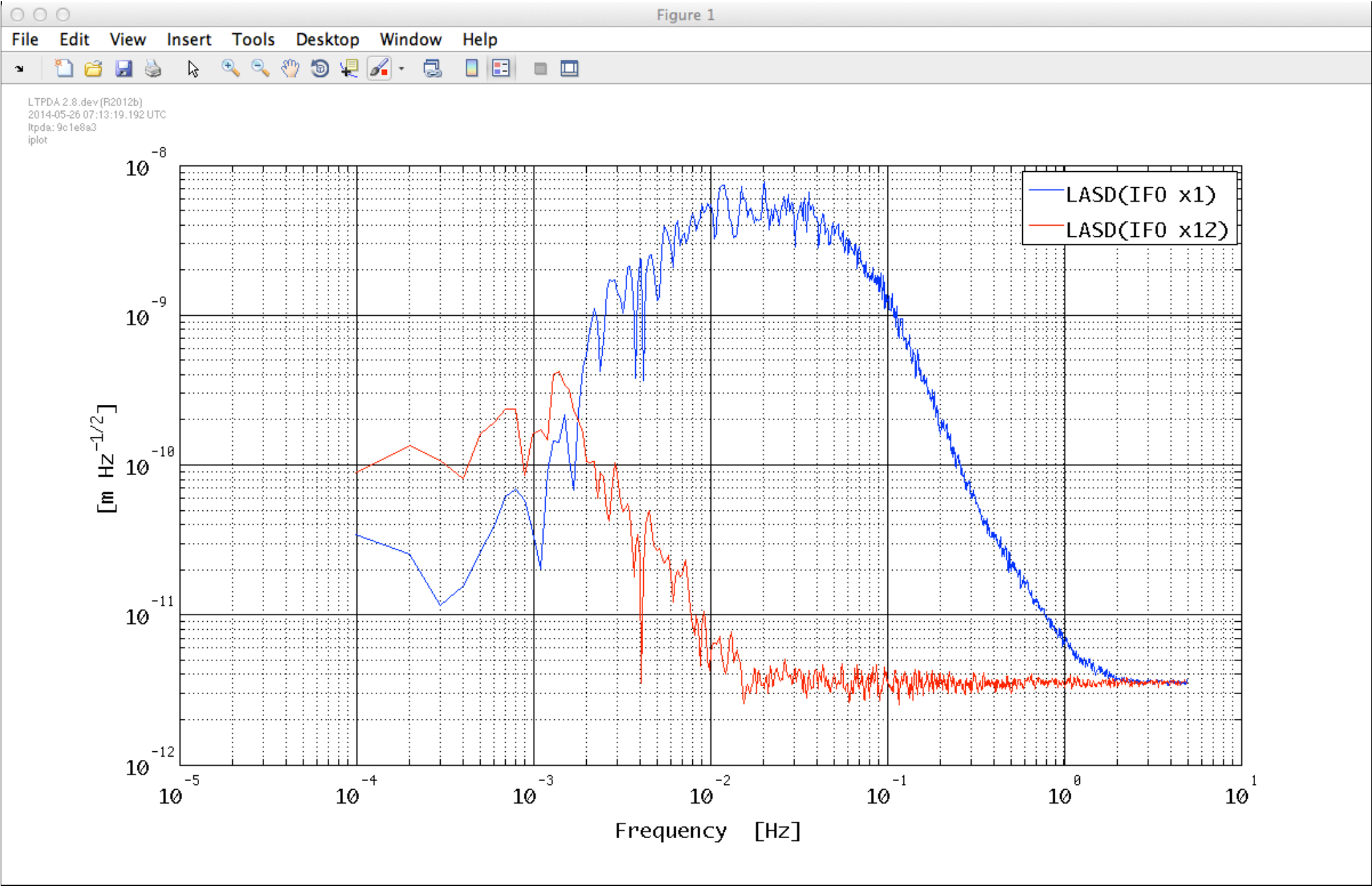
Empirical Transfer Function estimation

Let's run this exercise on empirical estimation of Transfer Functions.

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:
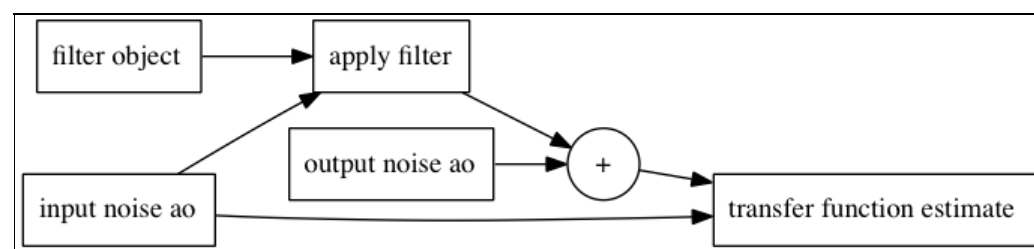
The command-line sequence is the following:

```matlab
%% 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]}));
```

Empirical Transfer Function estimation (LTPDA Toolbox)
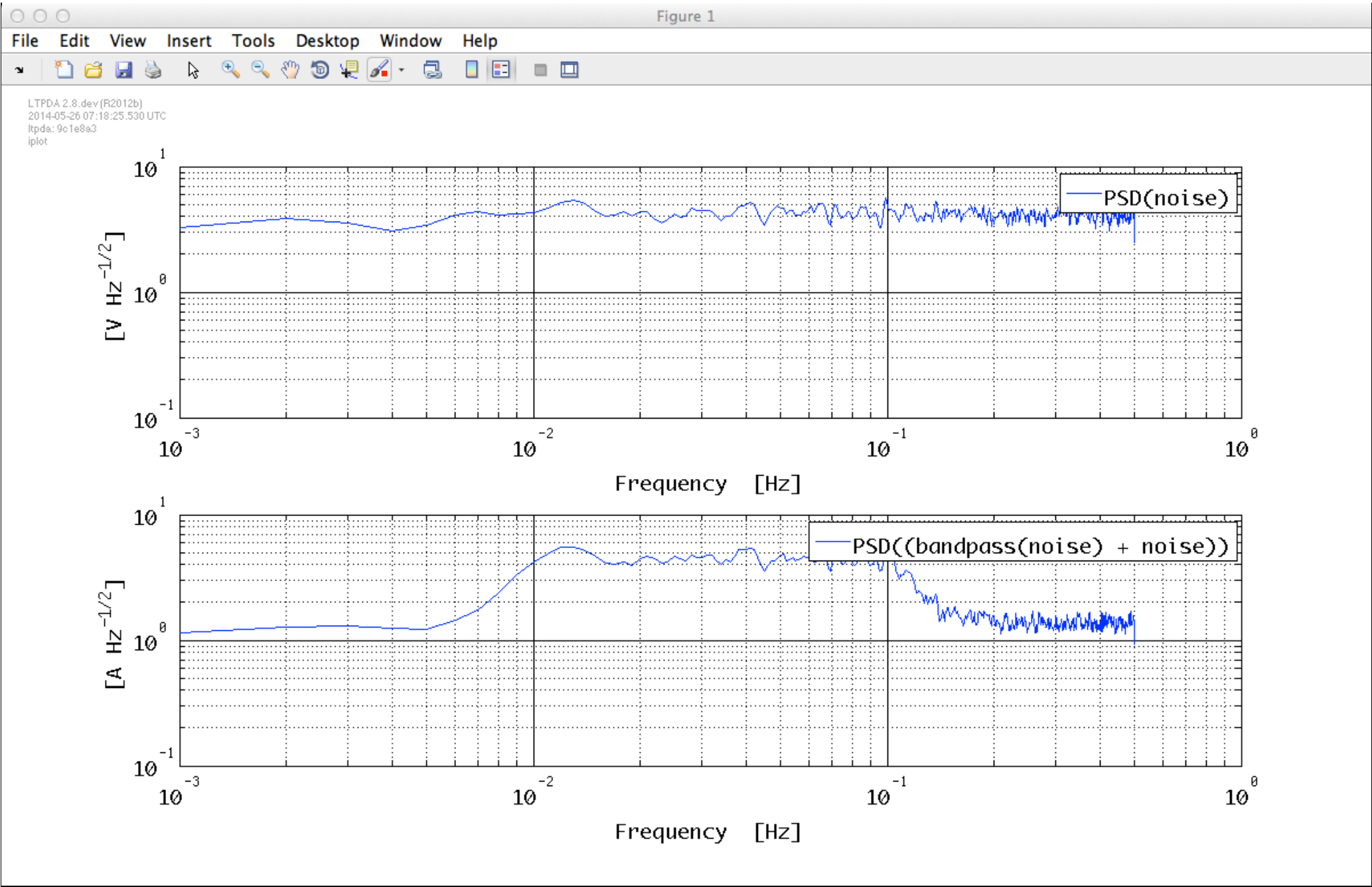
Now we can proceed with the call to the `ao/tfe` method. The parameter list is very similar to the one employed for the other spectral estimators:

Key	Value	Description
NFFT		

	1000	The number of samples defining the length of the window to apply
WIN	**'BH92'**	Or a different one, if you want.
OLAP	**−1**	Overlap will be chosen based on the window properties
ORDER	**0**	Segment−wise detrending up to order 0

The command line is the following:

```
%% Estimate the x->y transfer function
tfxy = tfe(x, y, plist('nfft', 1000, 'win', 'BH92', 'olap', -1, 'order', 0));
```

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-2 1e2], [-200 200], [1e-2 1e2], [-200 200]}))
```

Empirical Transfer Function estimation (LTPDA Toolbox)

©LTP Team

LTPDA Toolbox™   [contents](#)   ← →
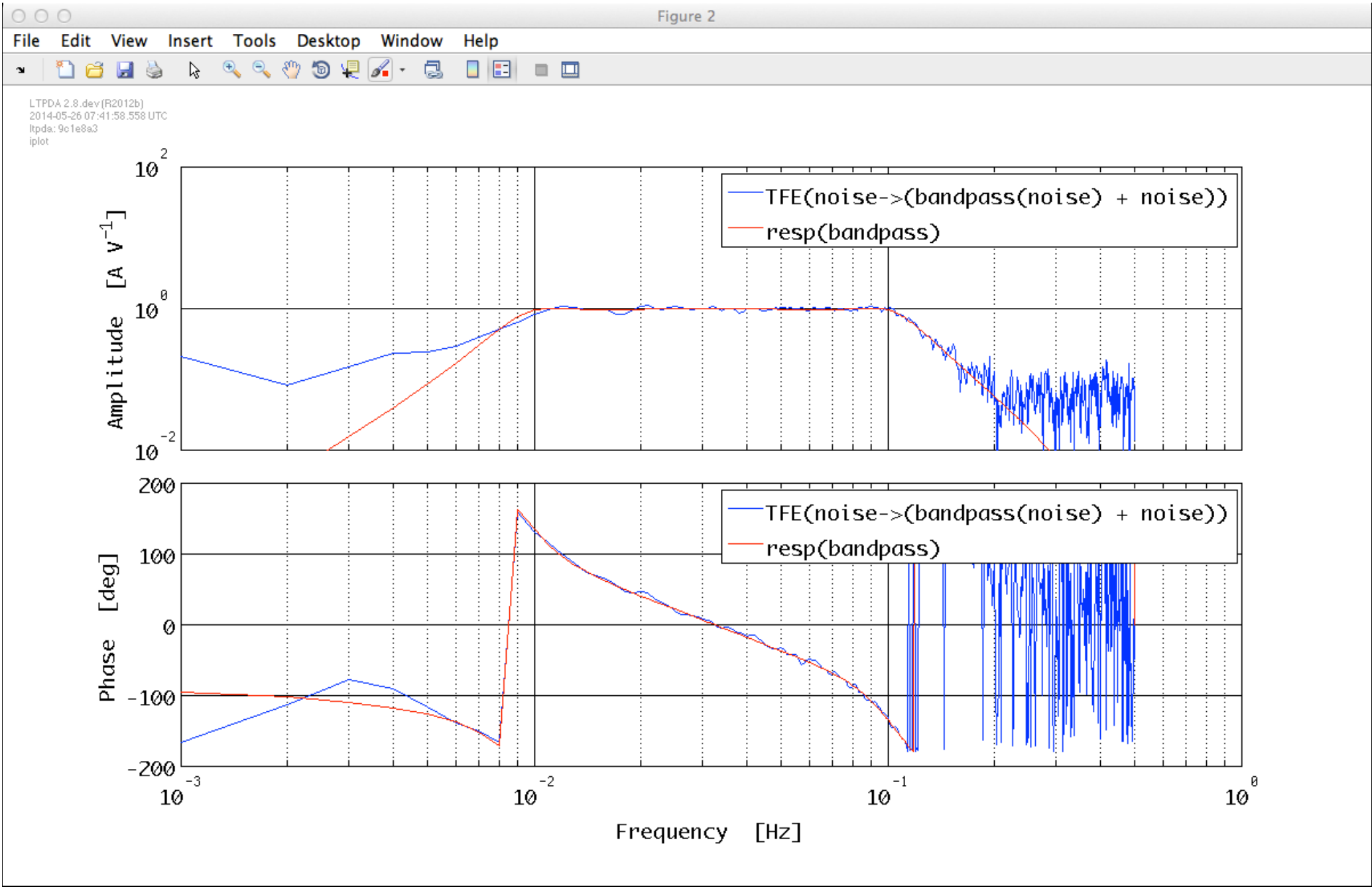
IFO/Temperature Example – Spectral Analysis

Loading the consolidated data sets from topic 2

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:

```
%% 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 proceed, evaluating 2 more `aos` to compare the effect of skipping the "glitch". After doing that, we can plot them in comparison.

Hint:

```
%% PSD
x_red_psd = lpsd(x_red);
x_red_psd.setName('Interferometer');

T_red_psd = lpsd(T_red)
T_red_psd.setName('Temperature');

% Plot estimated PSD
pl_plot = plist('Arrangement', 'stacked', 'LineStyles', {'-','-'},'Linecolors', {'b', 'r'});
iplot(sqrt(x_psd), sqrt(x_red_psd), pl_plot);
iplot(sqrt(T_psd), sqrt(T_red_psd), pl_plot);
```
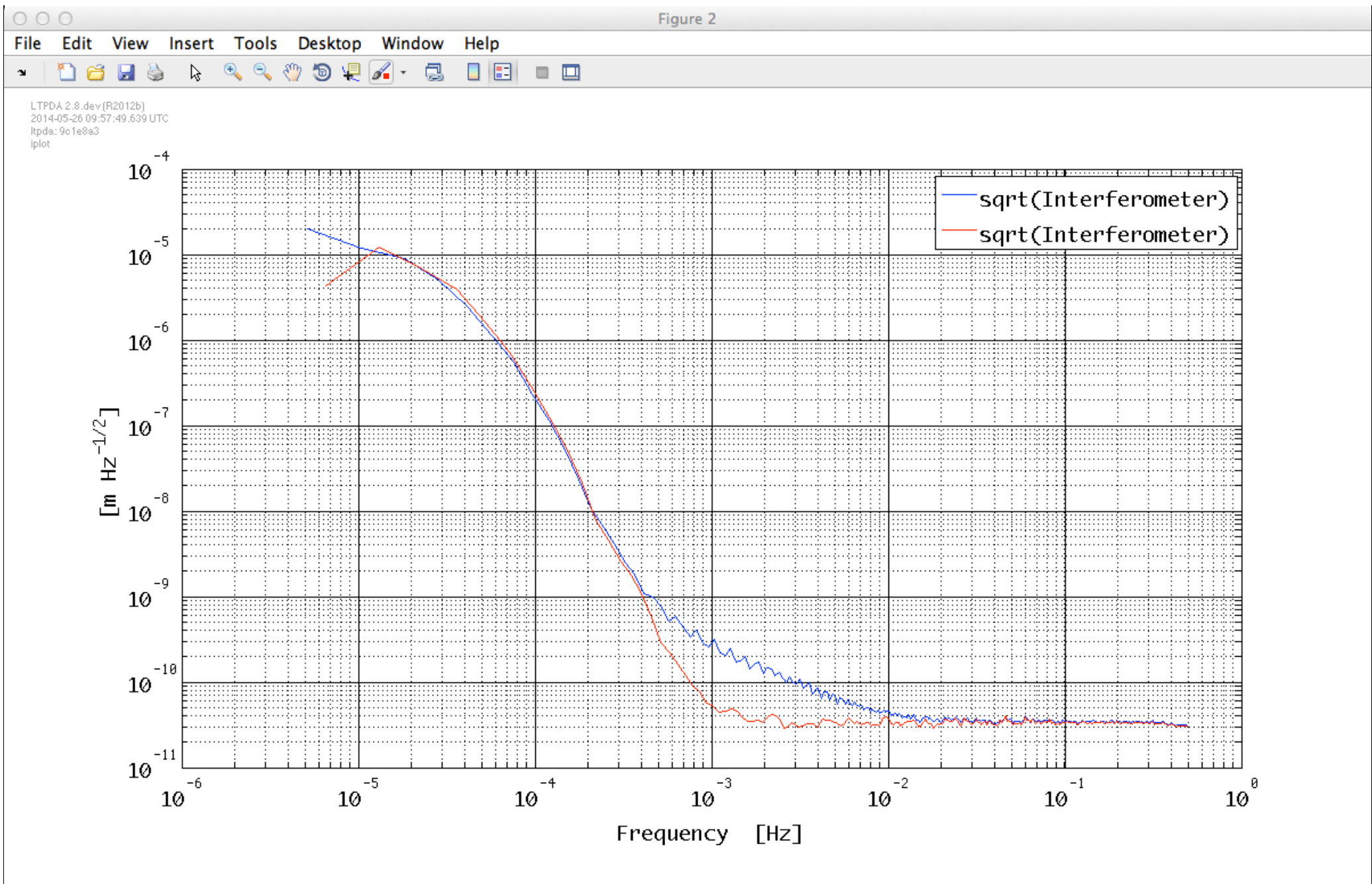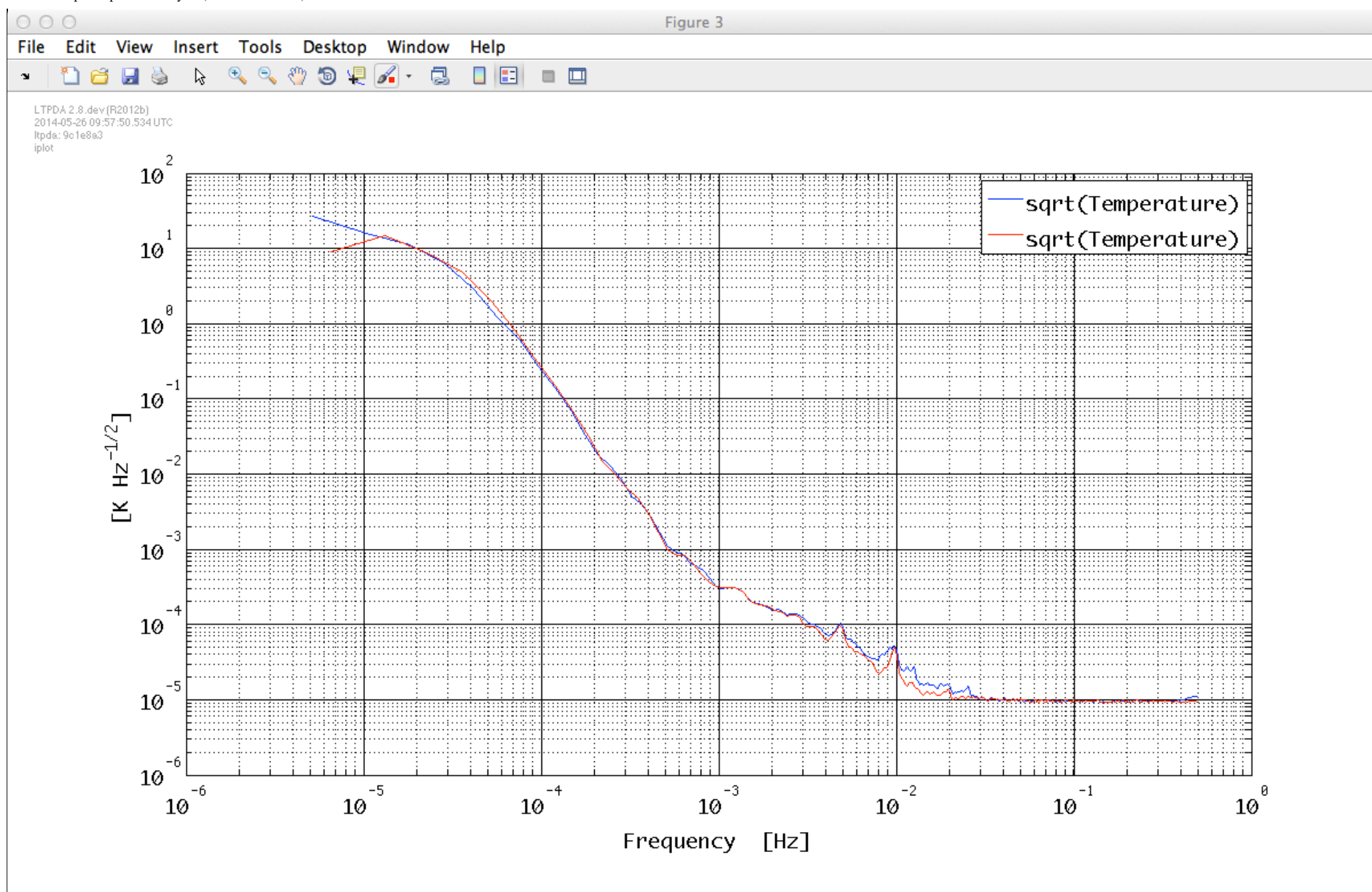
Estimating the cross-spectra

We can now proceed and use the

ao/lcpsd

method to evaluate the cross-spectra of the signals, employing a shorter window in order to reduce the scatter of the estimated values.

Hint:

```
%% CPSD estimate
CTx = lcpsd(T_red, x_red);
CxT = lcpsd(x_red, T_red);

% Plot estimated CPSD
iplot(CTx);
iplot(CxT);
```
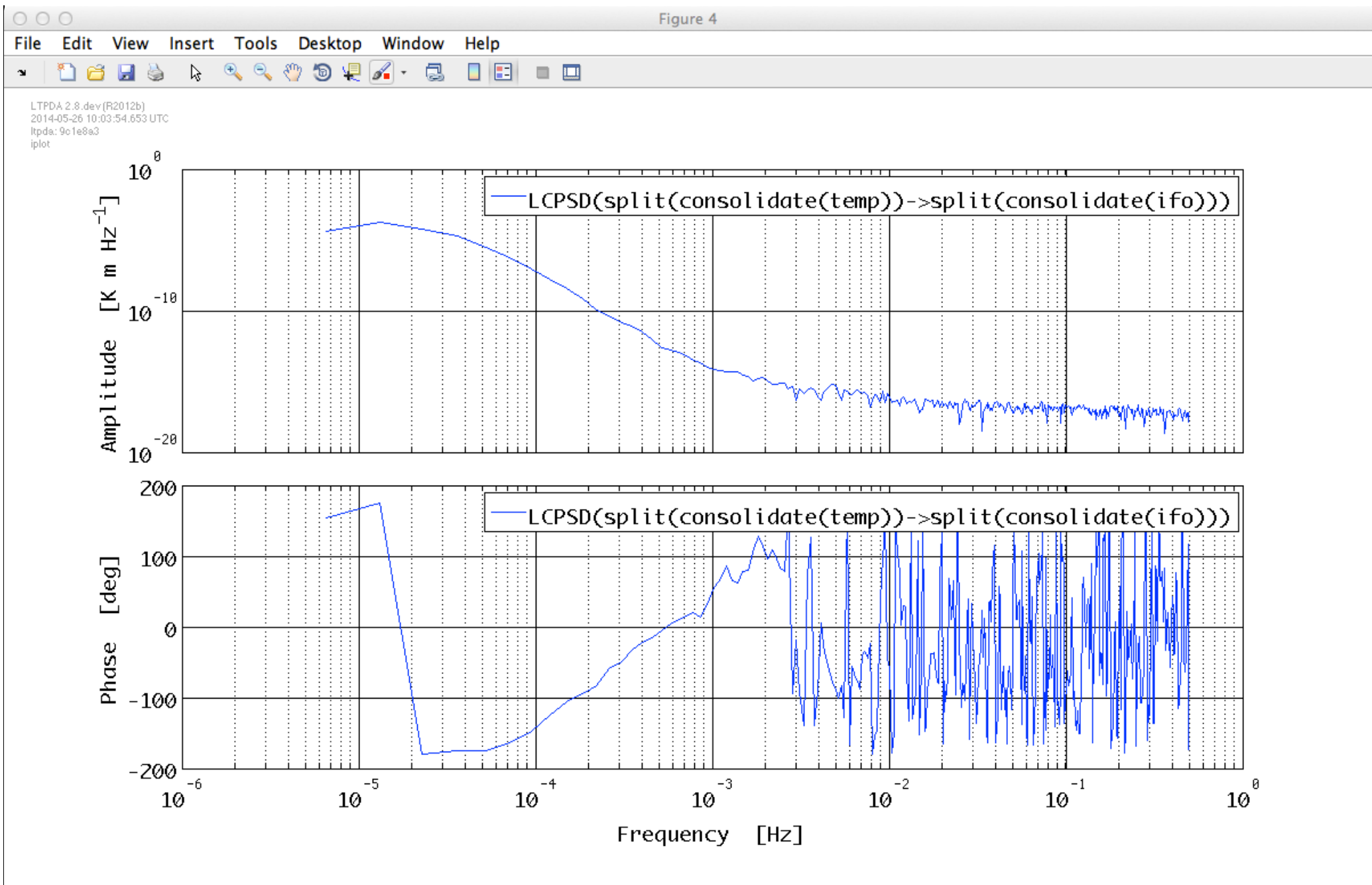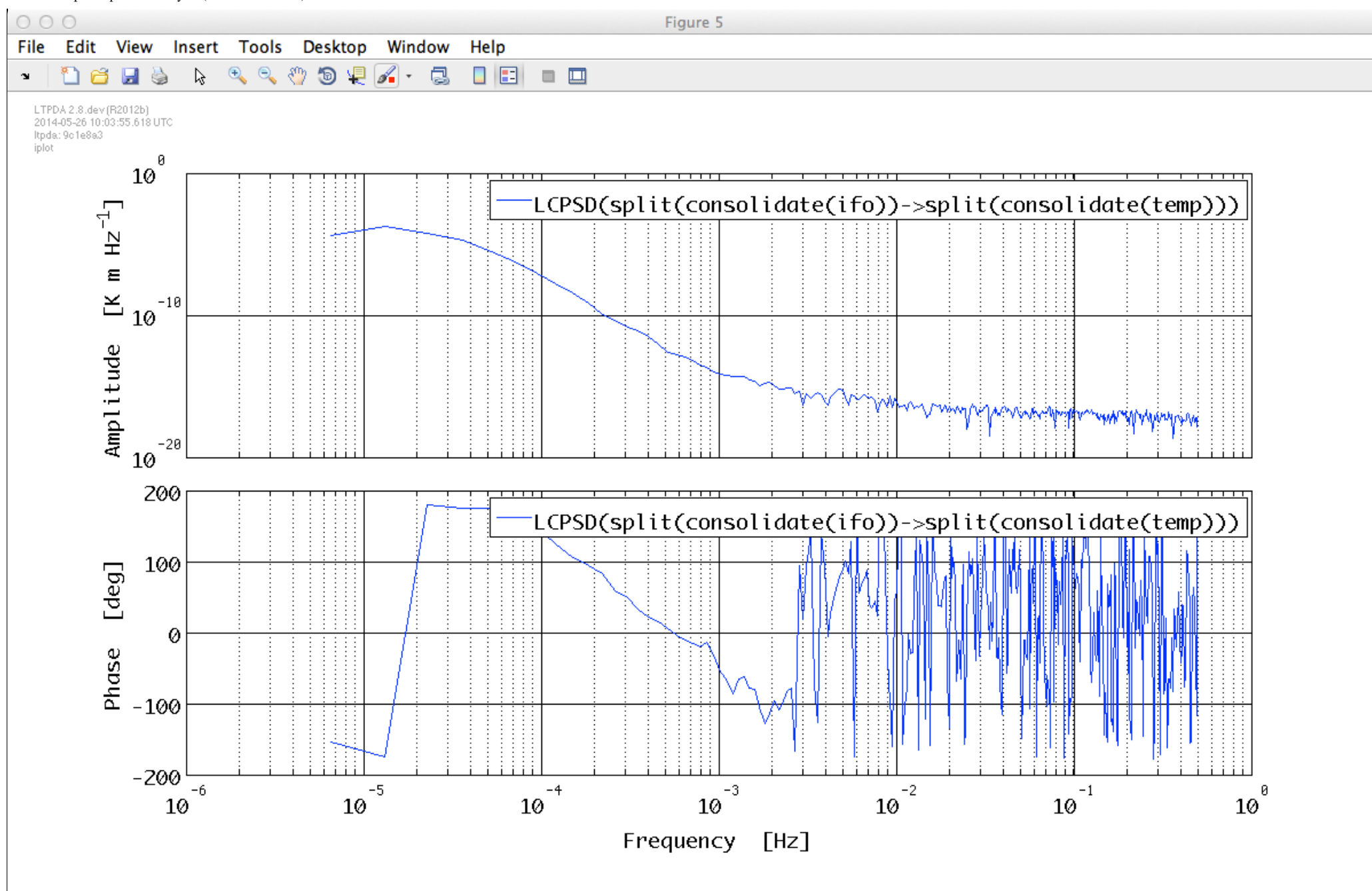
As expected, there is a strong low-frequency correlation between the IFO data x and the temperature data T.

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 `aos`, and we want to look at one of the off-diagonal terms:

```
%% Coherence estimate
coh = lcohere(T_red, x_red);

% Plot estimated cross-coherence
iplot(coh, plist('YScales', 'lin'))
```
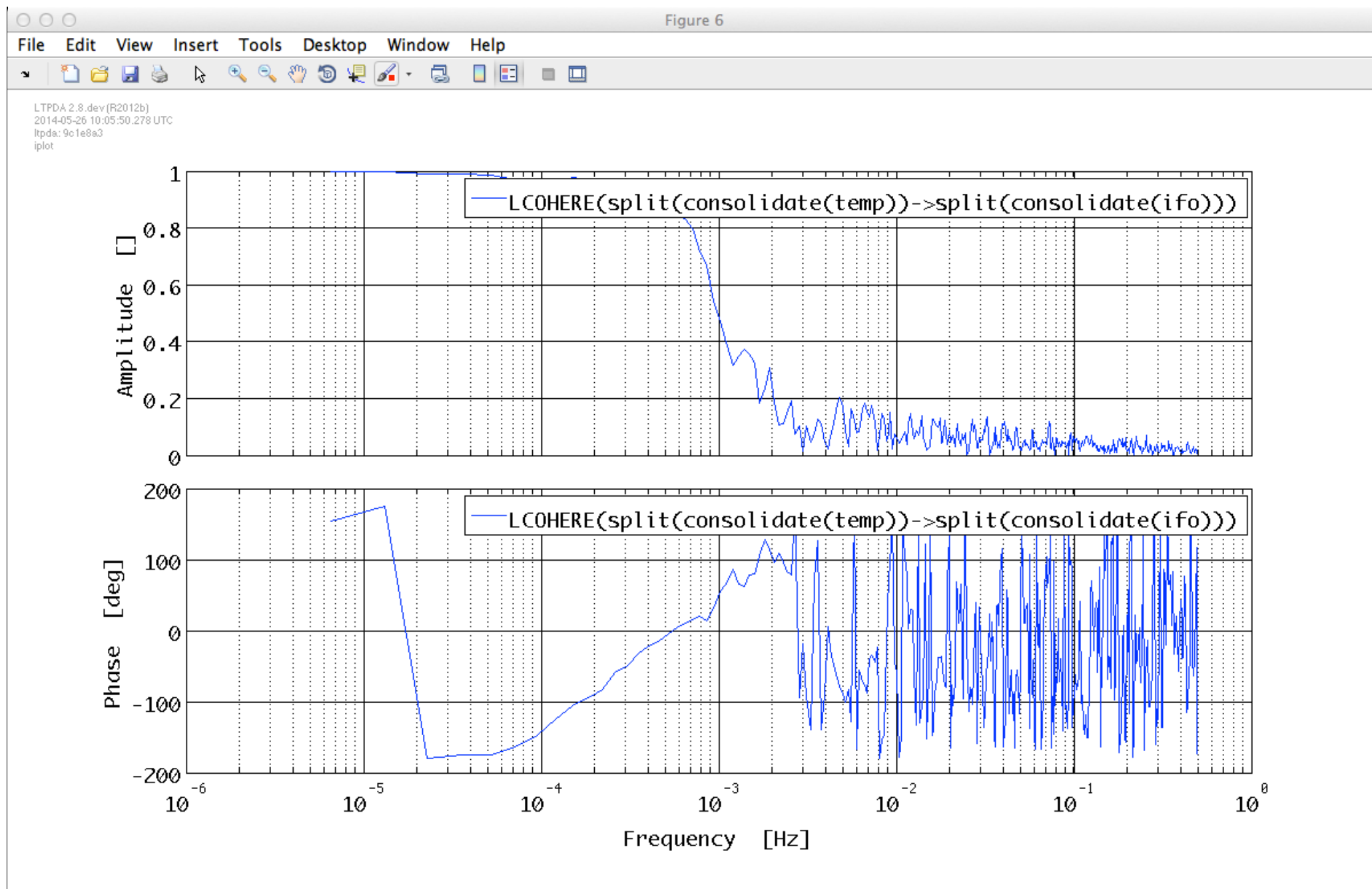
The coherence approaches 1 at low frequency.

Estimating the transfer function of temperature

We want now to perform noise projection, trying to estimate the transfer function of the temperature signal into the interferometer output. In order to do that, we can use the

`ao/ltfe`

method.

We also want to plot this transfer function to check that the units are correct.
Hint:

```
%% transfer function estimate
tf = ltfe(T_red, x_red)

% Plot estimated TF
iplot(tf);
```
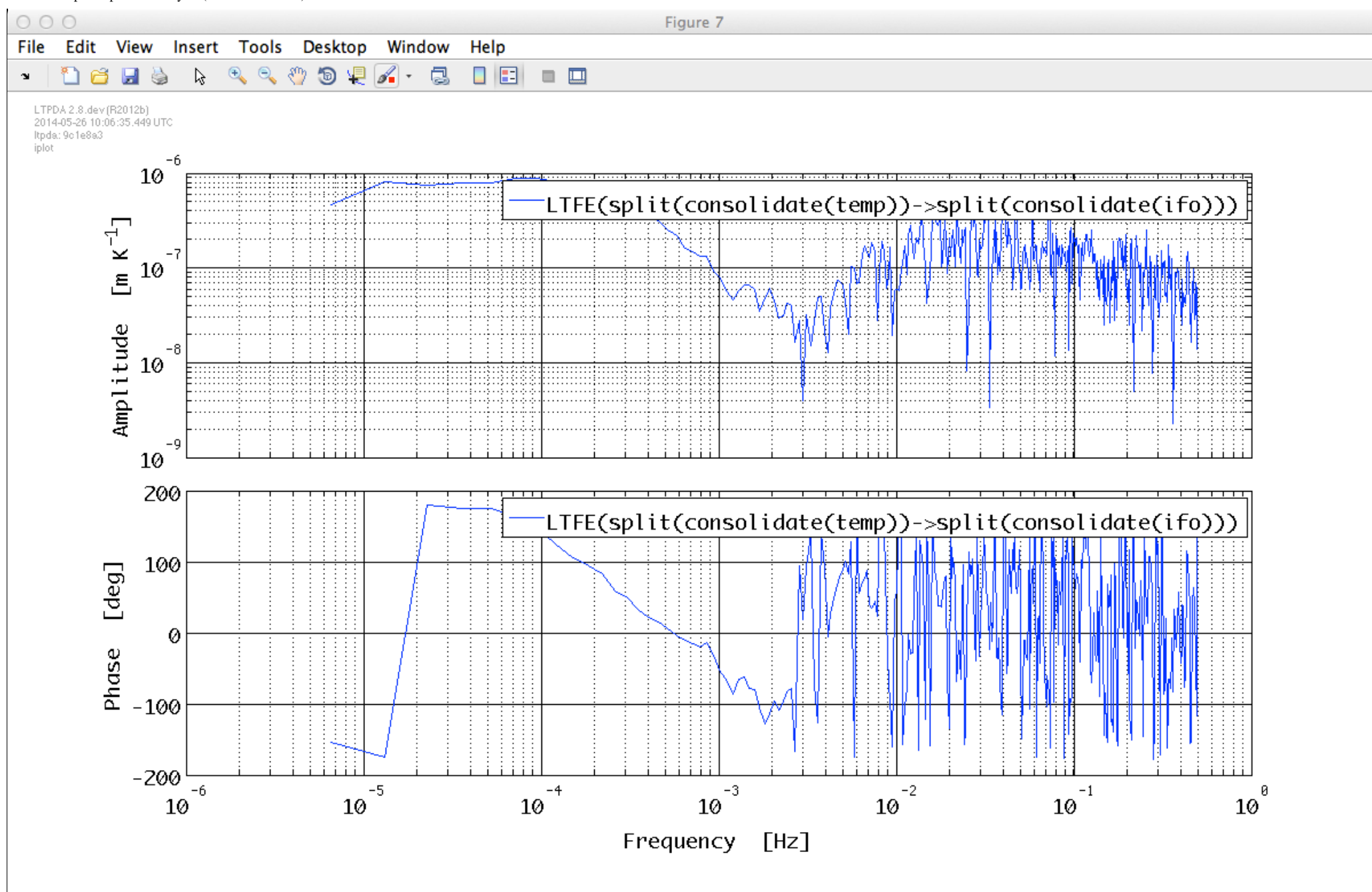
As expected, the transfer function T –> IFO is well measured at low frequencies.

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:

```matlab
%% Noise projection in frequency domain

proj = T_red_psd .* (abs(tf)).^2;
proj.simplifyYunits;
proj.setName('temp. contrib. projection')

%% Plotting the noise projection in frequency domain
iplot(x_red_psd, proj);
```

The contribution of the temperature fluctuations is clearly estimated.
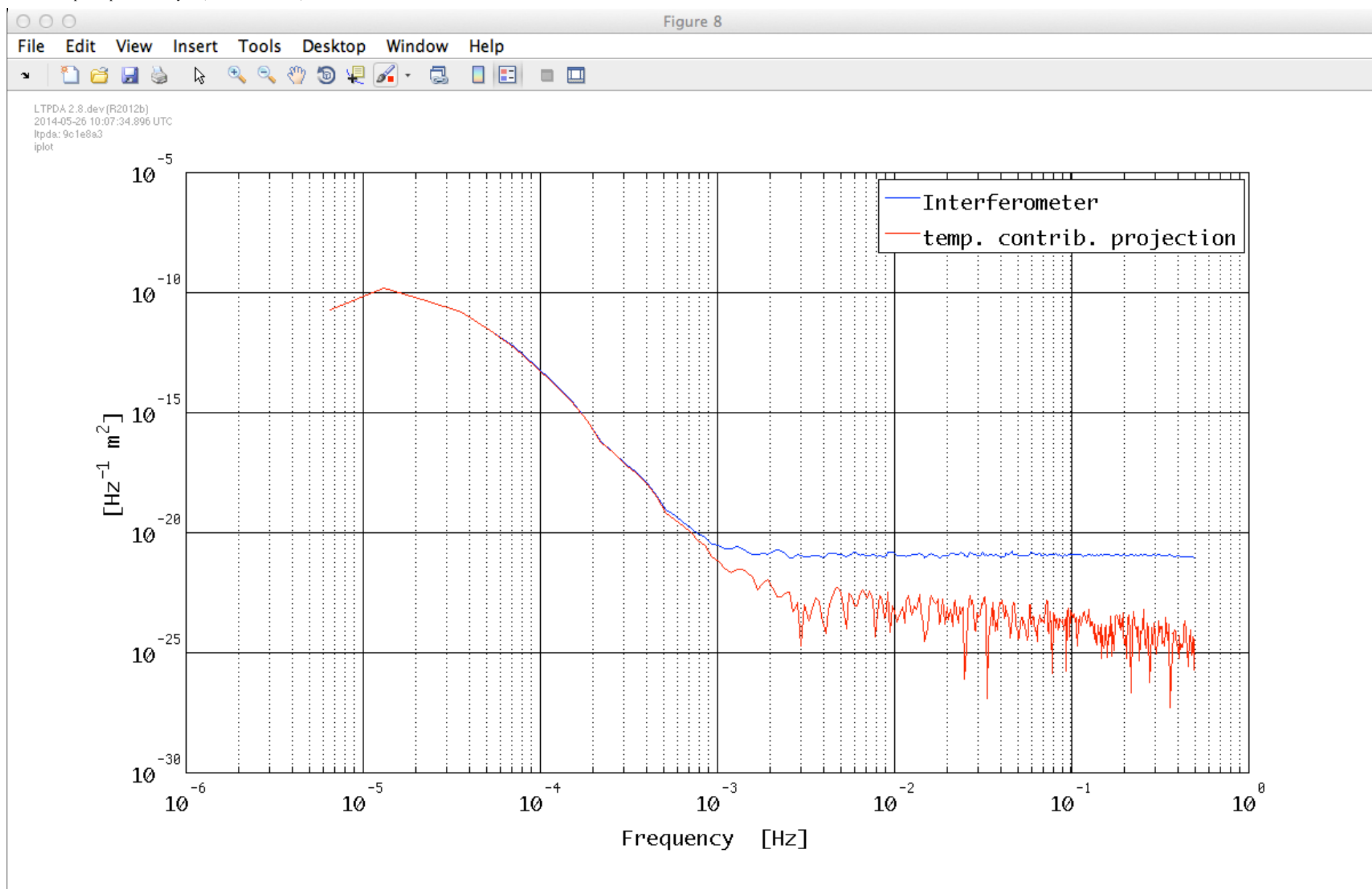
Saving the results

Let's fininsh this section of the exercise by saving the results on disk, in xml or mat 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:

```matlab
%% Save the PSD data
% 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

```matlab
% 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

```matlab
% Save
x_red_psd.save(pl_save_x_PSD);
T_red_psd.save(pl_save_T_PSD);
CxT.save(pl_save_xT_CPSD);
coh.save(pl_save_xT_cohere);
tf.save(pl_save_xT_TFE);
```

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Topic 4 – Transfer function models and digital filtering

Training session 4 is a tutorial of how to build transfer function models. These transfer functions can be either defined by the user or derived from an input/output time series. The tutorial also shows how to extract digital filters from the transfer functions and how to use these them to filter data. The topic is divided as follows:

- build transfer functions models in s domain
- model system and basic operation with models
- extract digital filters from transfer functions
- build digital filters
- filter data

◀ IFO/Temperature Example – Spectral Analysis   Create transfer function models in s domain ➡

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Create transfer function models in s domain

Building transfer function models

In this first part we will focus on the creation of transfer function models to then see how these can be translated into digital filters and applied to data. In the LTPDA toolbox, there are three possible ways to express a transfer function which can be more or less suitable depending on your particular application:

- Pole zero models
- Partial fraction models
- Rational models

◀ Topic 4 – Transfer function models and digital filtering   Pole zero model representation ▶

©LTP Team

Pole zero model representation

Creating a pole zero model

Let's build a pole zero model with the following characteristics:

Key	Value
GAIN	5
POLES	(f = 10 Hz, Q = 2)
ZEROS	(f = 1 Hz), (f = 0.1 Hz)

Our pole zero model has a pole at 10 Hz with a quality factor of Q = 2, and two zeros, one at 1 Hz, one at 0.1 Hz.

```
m = pzmodel(5, [10 2], {1, 0.1})
```

We can easily obtain the response of this transfer function by using the `pzmodel/resp` method.

```
resp(m)
```

The result is the figure shown below.

Pole zero model representation (LTPDA Toolbox)
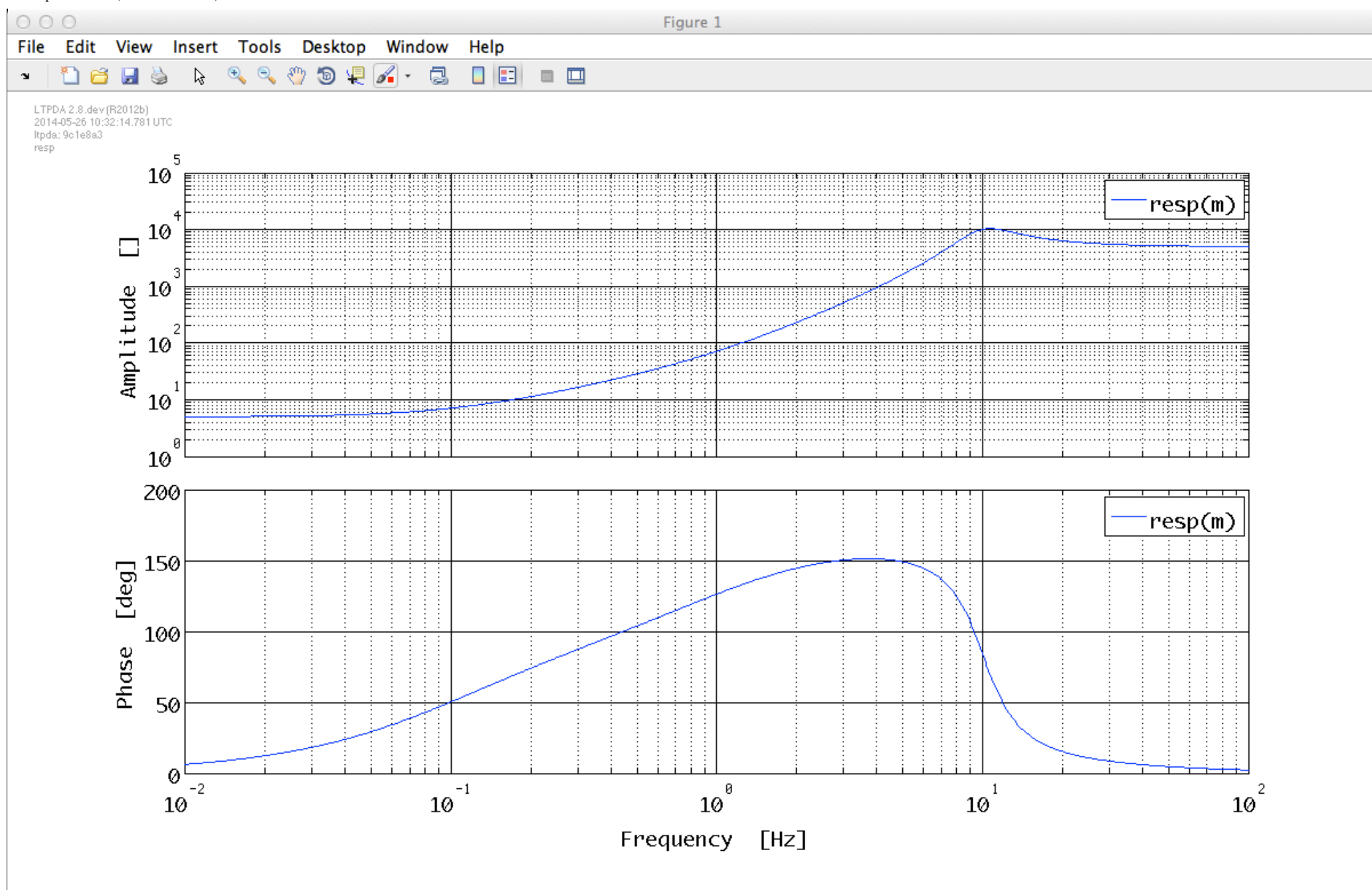
About the quality factor Q

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 ▶

©LTP Team

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], 0)
```

We can now take a look at the response of the transfer function that we have just created. We can specify the frequency region in what we are interested with the plist of the `resp` function. For example, we would like to evaluate our transfer function model from f1 = 0.1 Hz to f2 = 10 Hz.

```
resp(m, plist('f1', 0.1, 'f2', 10))
```

Partial fraction representation (LTPDA Toolbox)
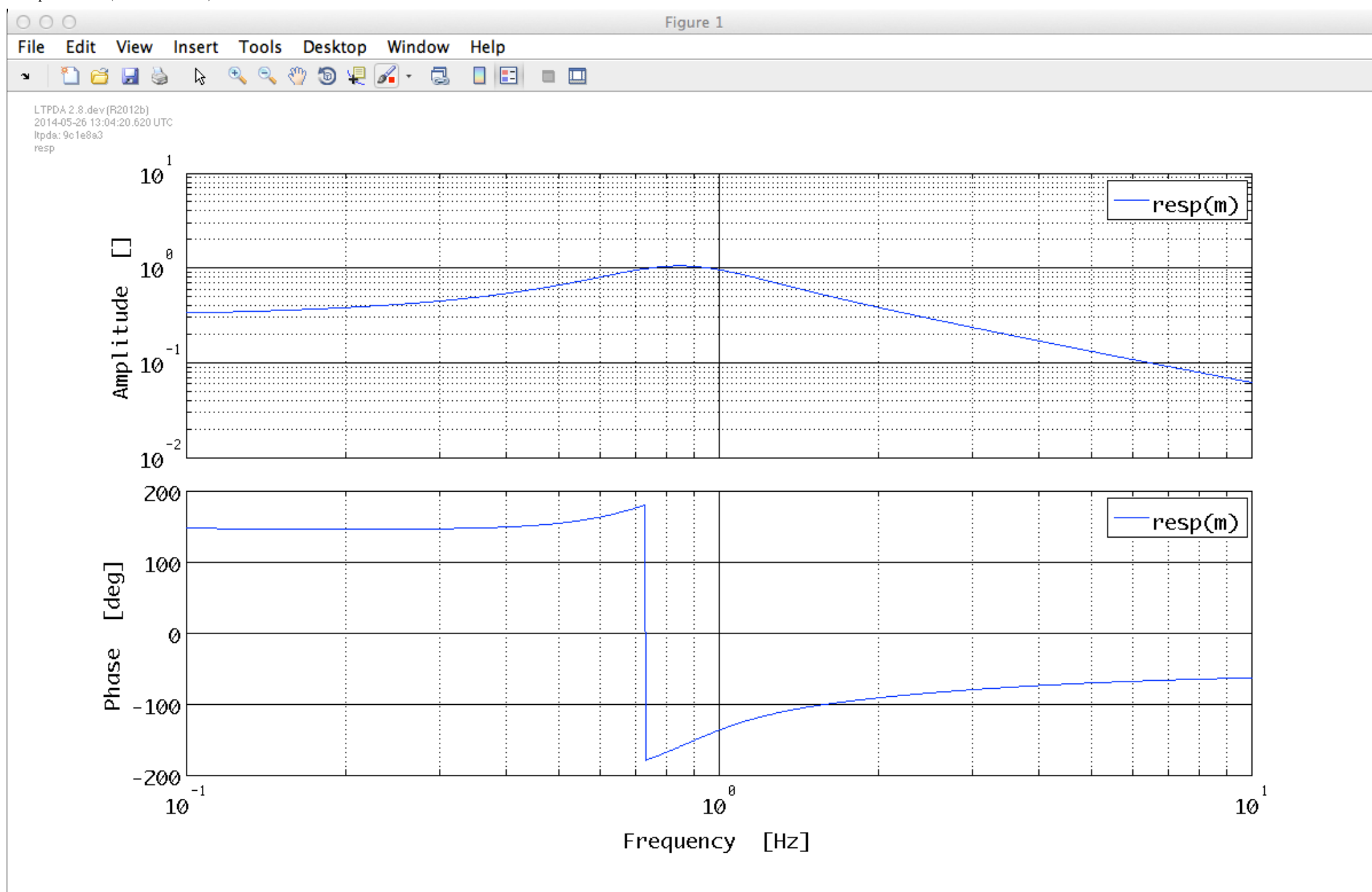
◀ Pole zero model representation   Rational representation ▶

©LTP Team

Rational representation

Creating a rational model

Another possibility to build a transfer function model is to use the `rational` constructor, which implements a transfer function as a quotient of polynomials:

```
% DESCRIPTION: RATIONAL rational representation of a transfer function.
%
%              a(1)s^m + a(2)s^{m-1} + ... + a(m+1)
%     H(s) = ------------------------------------
%              b(1)s^n + b(2)s^{n-1} + ... + b(n+1)
%
```

For example if the numerator and denominator coefficients are given by:

- num = [1 3 5]
- den = [1 8 10]

then we need to write down the following

```
m = rational([1 3 5], [1 8 10])
```

And now, as with the previous models, we can evaluate the response in a given frequency region

```
resp(m, plist('f1', 0.1, 'f2', 10))
```
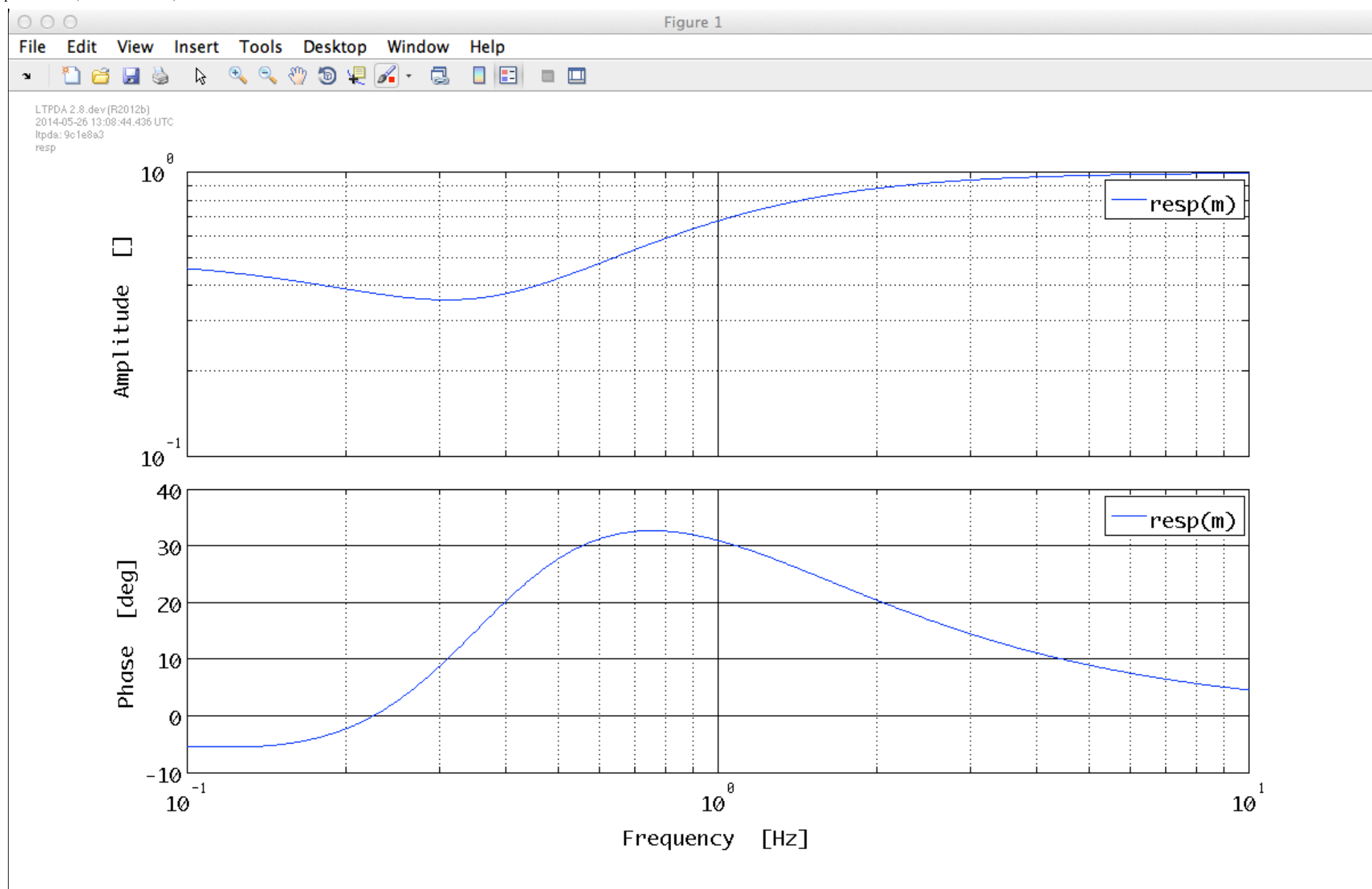
Transforming models between representations

Transforming between different representations

The LTPDA toolbox allows you to go from one representation to the other. However, in the current version (v2.3), 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: []
     description:
            UUID: 93420391-23c8-4829-b5a5-3988f1109be6
     pole 001: (f=10 Hz, Q=2, ri=-15.71+60.84i)
     zero 001: (f=1 Hz, Q=NaN, ri=-6.283)
     zero 002: (f=0.1 Hz, Q=NaN, ri=-0.6283)
     -------------------
```

To transform it into its rational equivalent we just need to give it as a input to the `rational` constructor

```
      rat = rational(pzm)
      ---- rational 1 ----
model:        rational(mymodel)
num:          [1.26651479552922 8.75352187005424 5]
den:          [0.000253302959105844 0.00795774715459477 1]
iunits:       []
ounits:       []
description:
UUID:         deb9a67f-6237-4112-9025-17fd4c42ce53
-------------------
```

The transformation returns again the first `pzmodel` however the name property allows us to follow the operations that we've been performing to this object.

```
pzm2 = pzmodel(rat)
---- pzmodel 1 ----
       name: pzmodel(rational(mymodel))
       gain: 5
      delay: 0
     iunits: []
     ounits: []
description:
       UUID: b2177fde-1465-4c5c-a94f-29d73d9aed67
pole 001: (f=10 Hz,Q=2)
zero 001: (f=1 Hz,Q=NaN)
zero 002: (f=0.1 Hz,Q=NaN)
------------------
```

◀ Rational representation   Modelling a system ▶

Modelling a system

To show some of the possibilities of the toolbox to model digital system we introduce the usual notation for a closed loop model:
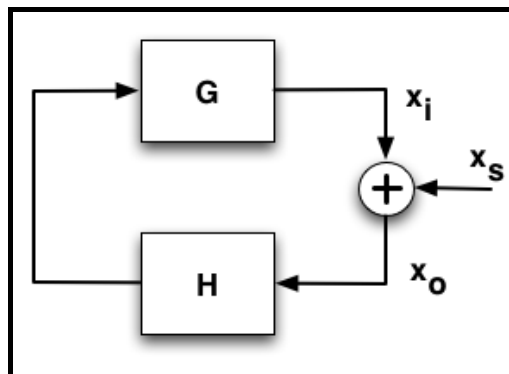
In our example we will assume that we know the `pzmodel` of the filter, `H`, and the open loop gain (OLG). These are related with the closed loop gain (CLG) by the following equation:

$$\frac{x_o}{x_s} = \text{CLG} = \frac{1}{1 - \text{OLG}} = \frac{1}{1 - H \cdot G}$$

We want to determine G and CLG. We would also like to find a digital filter for H, but we will deal with this in the following section.

Loading the starting models

Imagine that we have somehow managed to find the following model for OLG:

Key	Value
GAIN	4e6
POLES	1e-6

then we can create a `pzmodel` with these parameters as follows

```
    OLG = pzmodel(4e6, 1e-6, [], 'OLG')
    ---- pzmodel 1 ----
          name: OLG
          gain: 4000000
         delay: 0
        iunits: []
        ounits: []
```

```
    description:
            UUID: 3d8bce32-a9a9-4e72-ab4d-183da69a9b5d
    pole 001: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    ------------------
```

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')
    ---- pzmodel 1 ----
          name: none
          gain: 1000000000
         delay: 0
        iunits: []
        ounits: []
    description:
            UUID: e683b32f-4653-474e-b457-939d33aeb63c
        pole 001: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    pole 002: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    zero 001: (f=0.001 Hz, Q=NaN, ri=-0.006283)
------------------
```

and we see how the constructor recognizes and translates the poles and zeros in the file. The model gets the name from the file but we can easily change it to have the name of our model:

```
    H.setName();
```

According to our previous definition we can get the `plant` by dividing the OLG by H. We can do so directly when dealing with `pzmodel` objects since multiplication and division are allowed for these objects, then

```
    G = OLG/H
    ---- pzmodel 1 ----
          name: (OLG./H)
          gain: 0.004
         delay: 0
        iunits: []
        ounits: []
    description:
            UUID: fcd166f7-d726-4d39-ad2e-0f3b7141415b
        pole 001: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
        pole 002: (f=0.001 Hz, Q=NaN, ri=-0.006283)
        zero 001: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
        zero 002: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    ------------------
```

which we need to simplify to get rid of cancelling poles and zeros. We also set the model name here.

```
    G.setName();
    G.simplify
    ---- pzmodel 1 ----
          name: simplify(G)
          gain: 0.004
         delay: 0
        iunits: []
        ounits: []
    description:
            UUID: c1883713-b860-4942-a127-e42ea565460f
        pole 001: (f=0.001 Hz, Q=NaN, ri=-0.006283)
        zero 001: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    ------------------
```

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();
CLG.iplot();
```

which gives us an AO that we can plot:
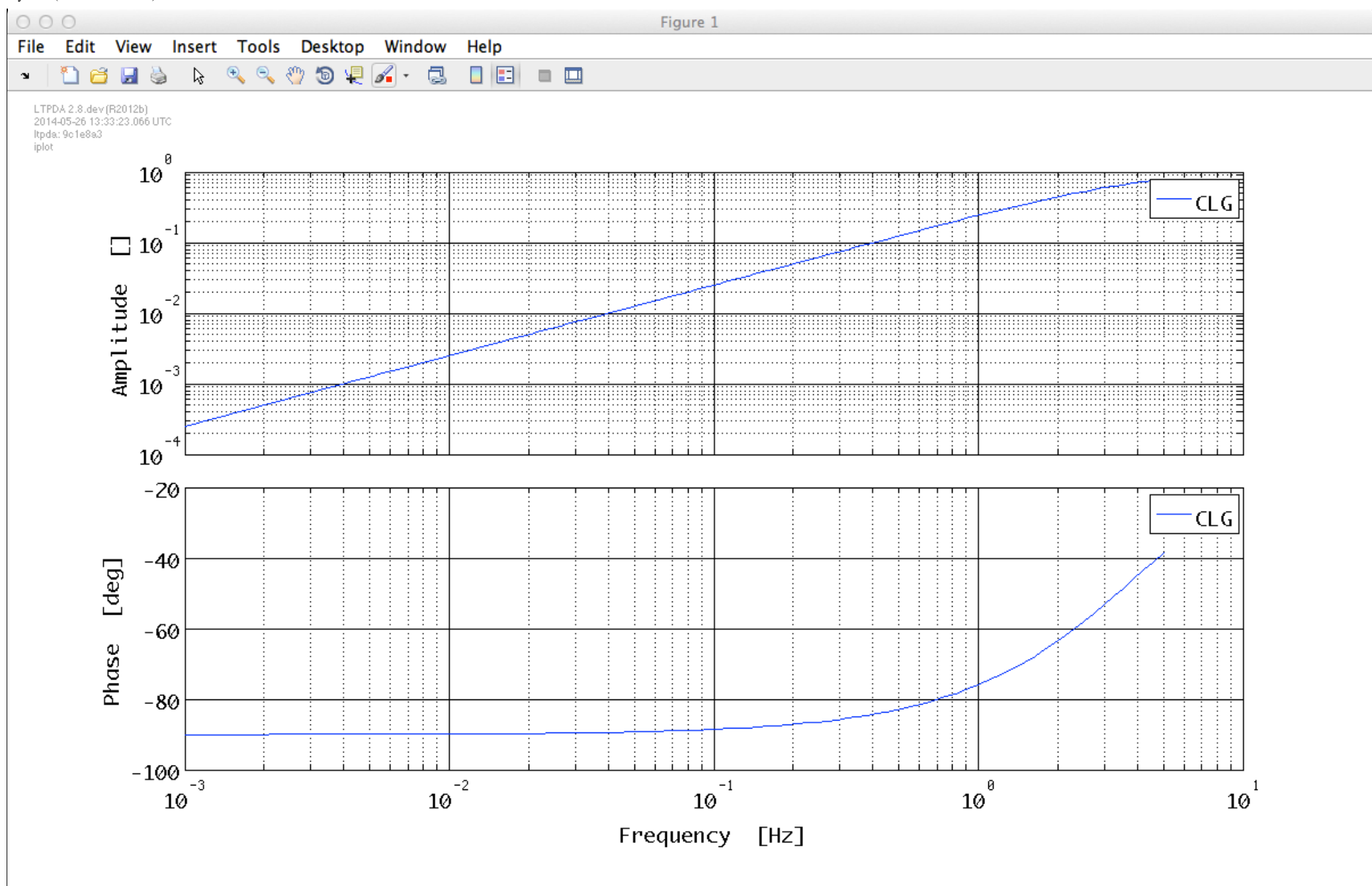
Fine, but my (real) system has a delay...

You can now repeat the same procedure but loading a H model with a delay from the LISO file 'LISOFileDelay.fil':

```
H_Delay = pzmodel('topic4/LISOfileDelay.fil')
---- pzmodel 1 ----
        name: none
        gain: 1000000000
       delay: 0.125
      iunits: []
      ounits: []
description:
        UUID: 6dfbddc5-b186-4405-8f6e-02a2822a22c5
    pole 001: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    pole 002: (f=1e-06 Hz, Q=NaN, ri=-6.283e-06)
    zero 001: (f=0.001 Hz, Q=NaN, ri=-0.006283)
------------------
```

and calculate the response:

```
H_Delay.setName();
pl = plist('f1', 1e-3, 'f2', 5, 'nf', 100);
resp([H, H_Delay], pl)
```

You will see how the delay is correctly handled, meaning that it is added when we multiply two models and subtracted if the models are divided.
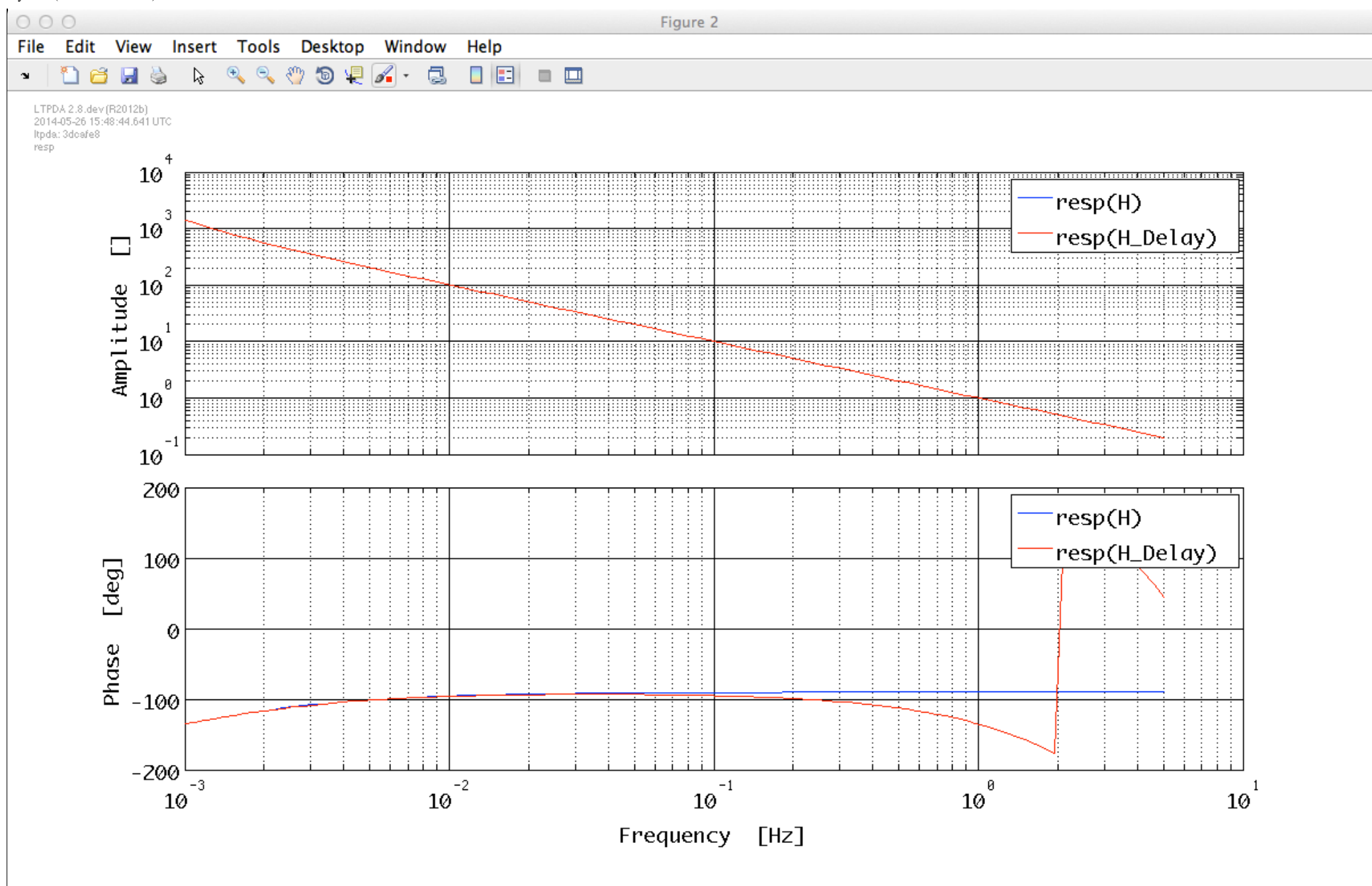
How to filter data

How to filter data

In the previous sections we've been dealing with continuous (in s domain) systems. Now we want to apply these systems 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.

©LTP Team

By discretizing transfer function models

By discretizing transfer functions

In the following we want to show how to go from continuous (s) domain to digital (z) domain when working with transfer function models. We will keep working with the models from our previous closed loop example.

Discretizing a transfer function model

Once we have our continuous models it is fairly simple to obtain their digital representation. We can insert them into the `miir` constructor specifying a sampling frequency, for instance fs = 10 Hz, as follows:

```
G_digital = miir(G, plist('fs', 10));
H_digital = miir(H, plist('fs', 10));
OLG_digital = miir(OLG, plist('fs', 10));
```

We want to check if the discretization went right, but in order to do that we need to compute the response for the digital and the continuous versions separately, since the `resp` method can not process inputs from different types. We can do the following:

```
%% Compare response

pl_resp = plist('f1', 1e-3, 'f2', 5, 'nf', 100);

% Digital
rG_digital = resp(G_digital, pl_resp);
rG_digital.setName('G_digital');
rH_digital = resp(H_digital, pl_resp);
rH_digital.setName('H_digital');
rOLG_digital = resp(OLG_digital, pl_resp);
rOLG_digital.setName('OLG_digital');

% Continuous
rG = resp(G, pl_resp);
rG.setName('G');
rH = resp(H, pl_resp);
rH.setName('H');
rOLG = resp(OLG, pl_resp);
rOLG.setName('OLG');

% Plot
iplot(rG_digital, rG)
iplot(rH_digital, rH)
iplot(rOLG_digital, rOLG)
```

By discretizing transfer function models (LTPDA Toolbox)

By discretizing transfer function models (LTPDA Toolbox)
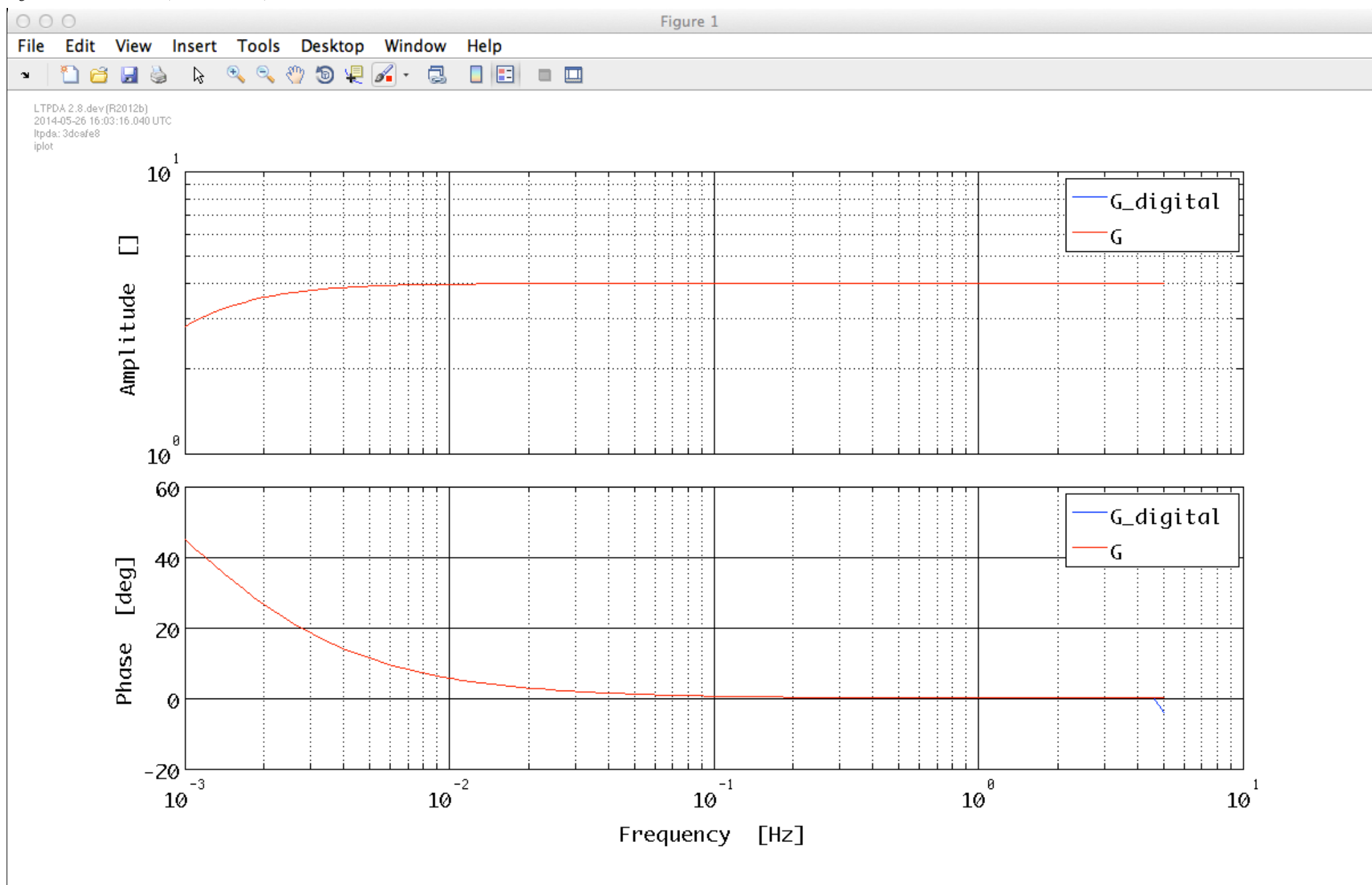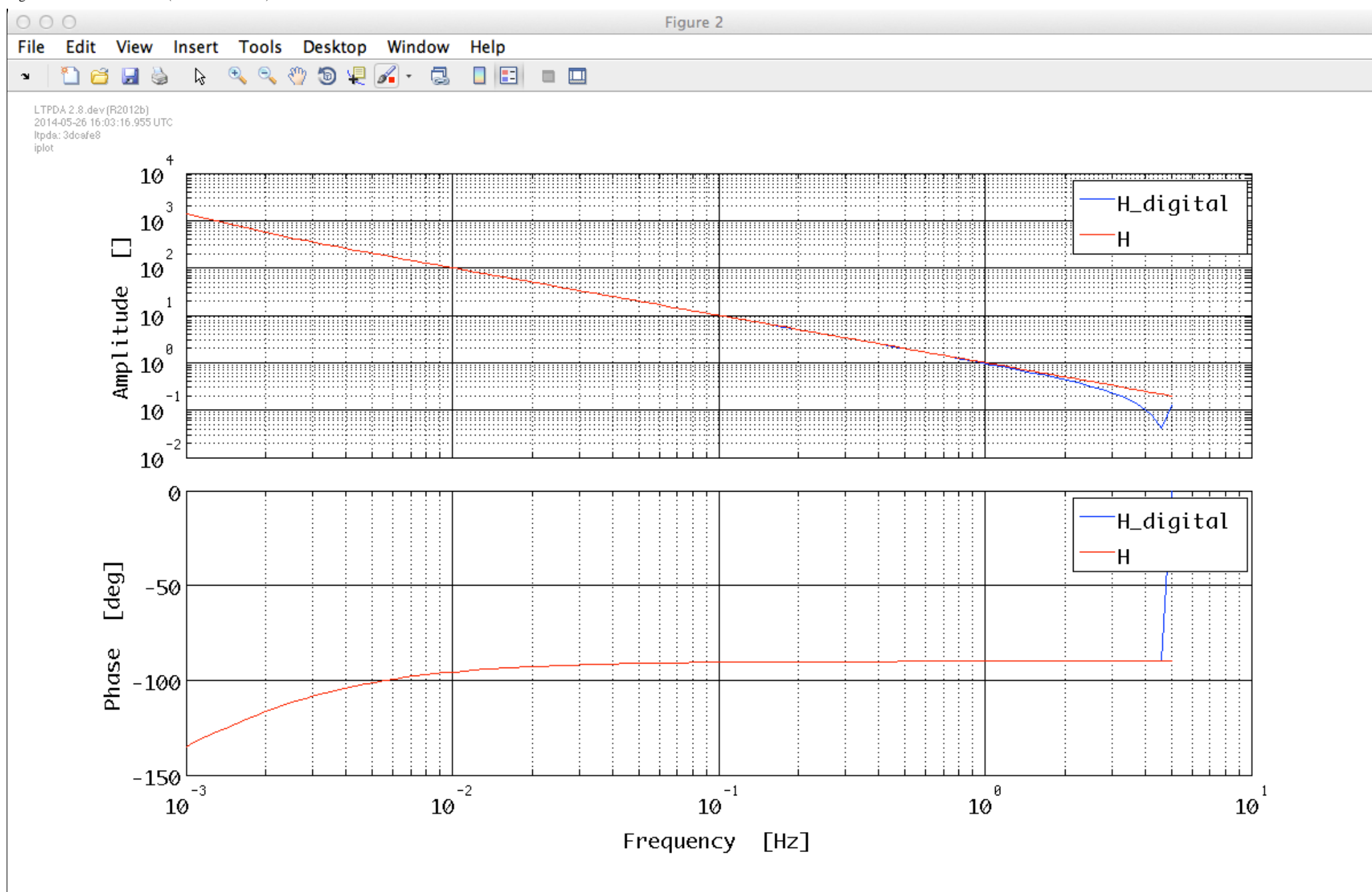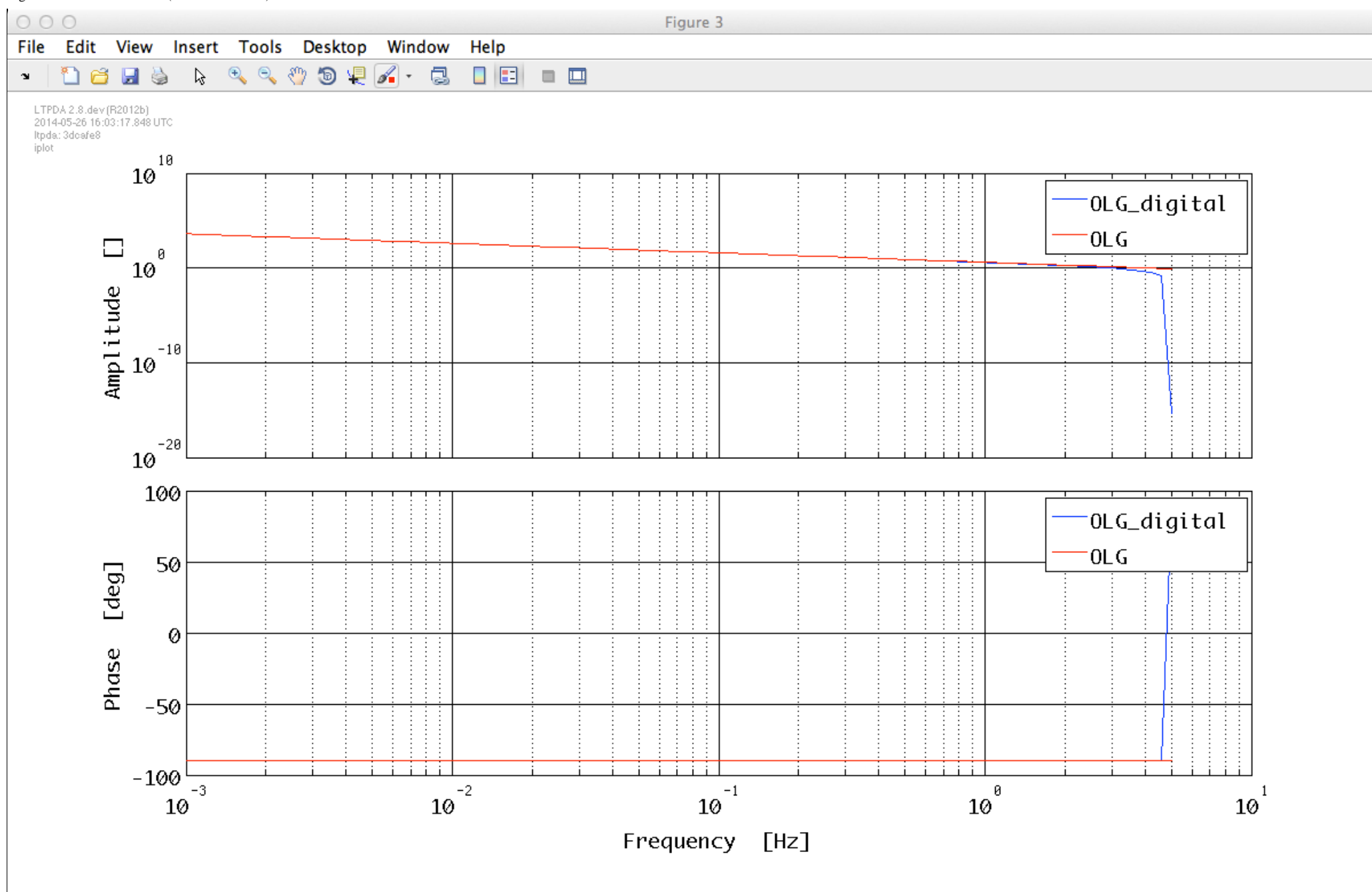
Once the comparison is done, one is usually interested in the coefficients of the digital implementation. These are directly accessible as, for example, `Gd.a` or `Gd.b`

```
>> Gd.a

ans =

3.99874501384116        2.51248480253707e-06         -3.99874250135635

>> Gd.b

ans =

1       0.000628121200622944          -0.999371878799377
```

The Delay strikes back

The current version of the toolbox (v2.8) 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:

By discretizing transfer function models (LTPDA Toolbox)

By discretizing transfer function models (LTPDA Toolbox)

By discretizing transfer function models (LTPDA Toolbox)

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶
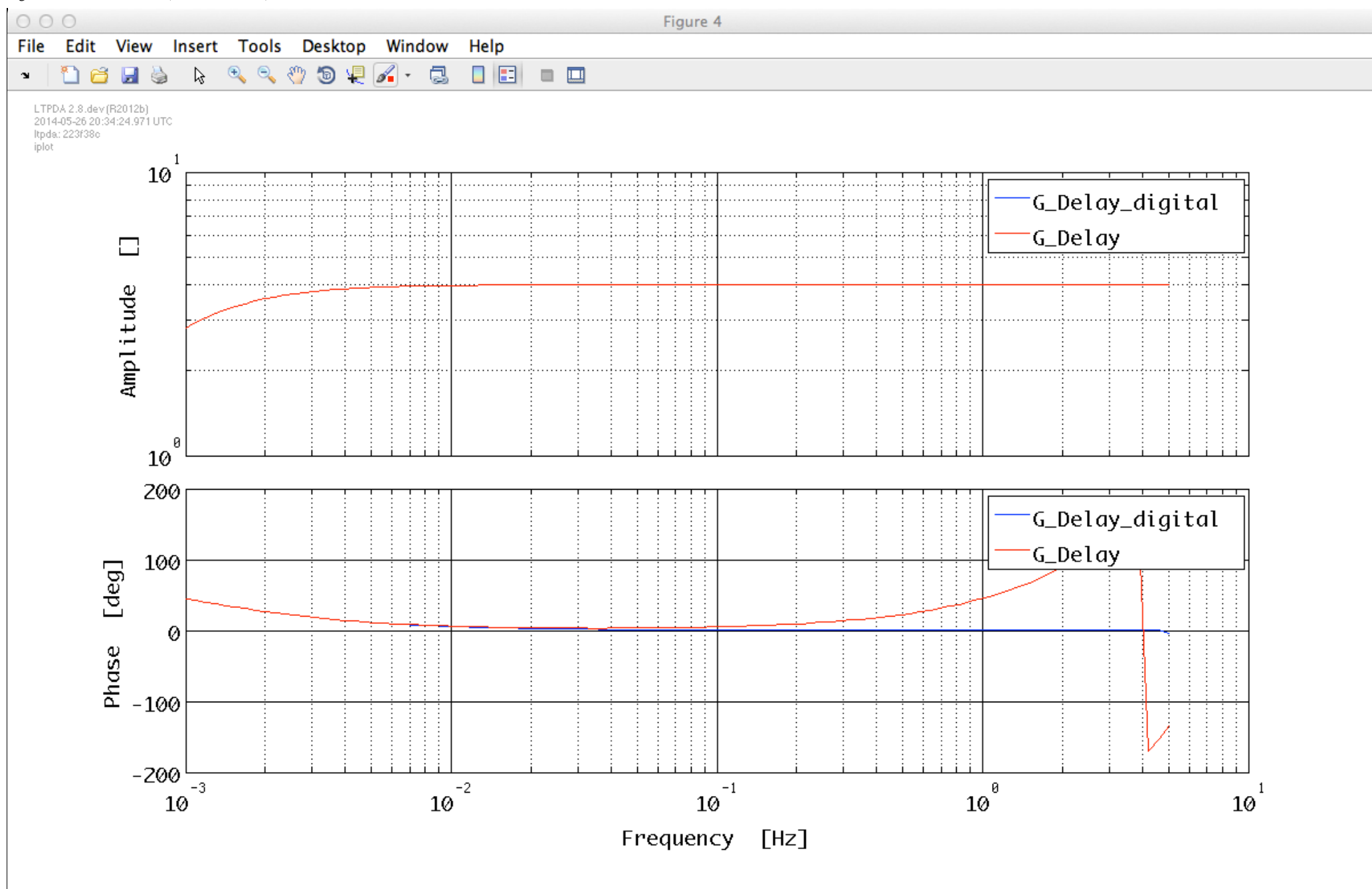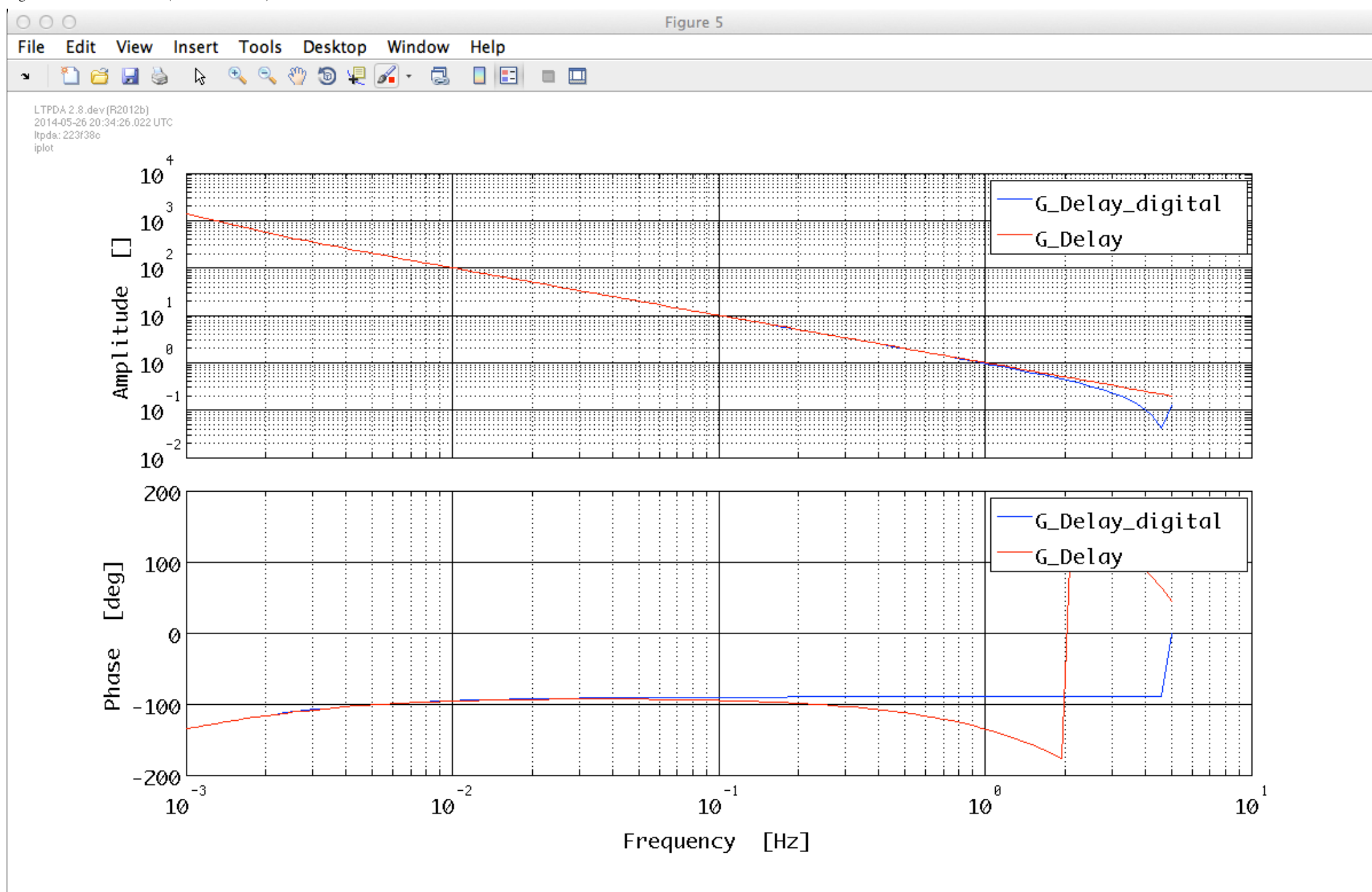
By defining filter properties

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
```
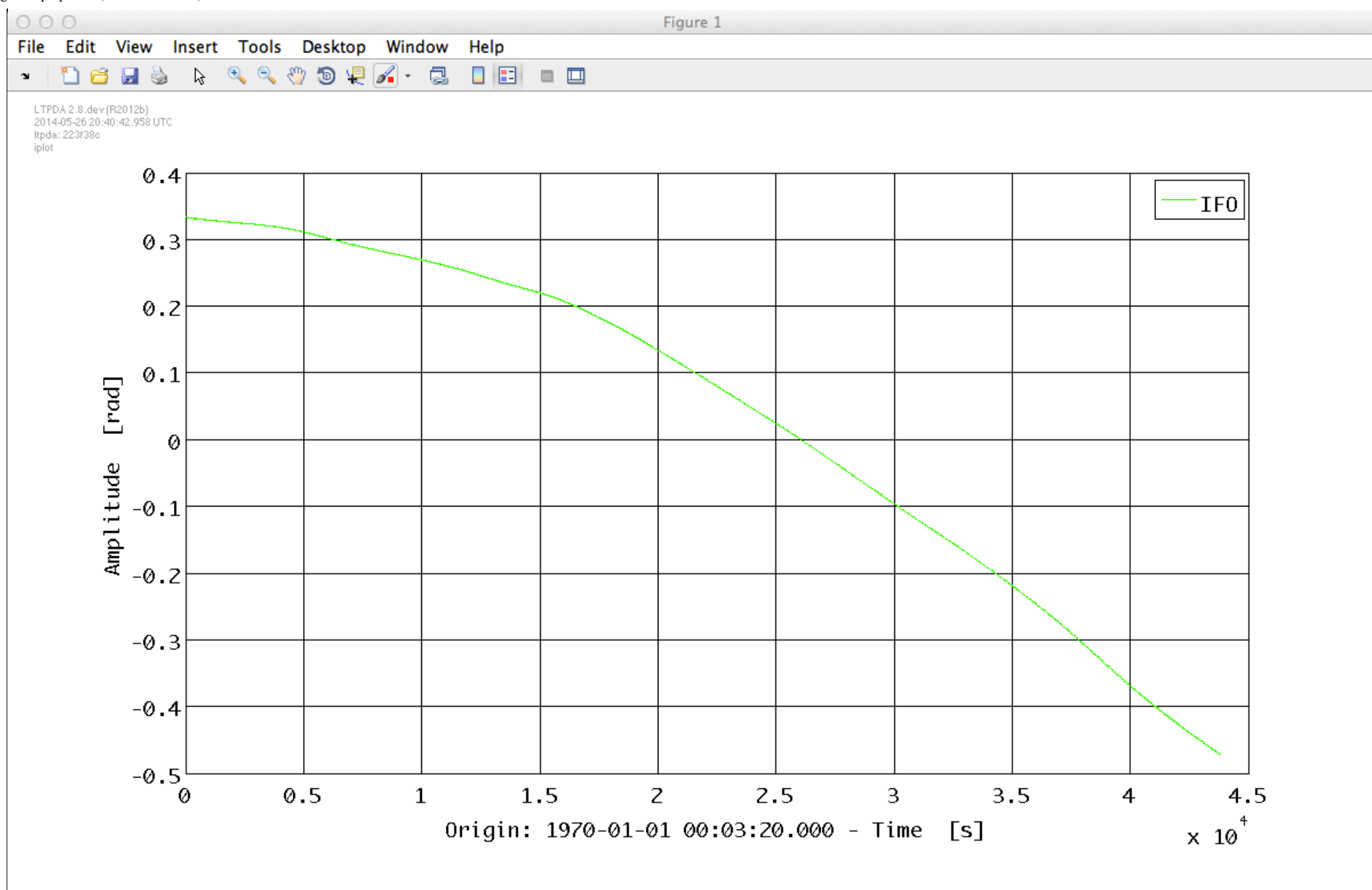
Before estimating the power spectrum we would like to remove all the contributions outside our band and, in particular, the trend which will add up as a low frequency feature into our spectrum. We could apply the `detrend` method but we can also design a bandpass filter to do the job. The properties 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 below and above the interesting bandwidth trying to avoid any kind of unwanted effect in the band. All the previous properties of the filter can be set as parameters in a `plist` that we can then insert in the constructor. We will use in our example the `miir` constructor to create a IIR filter.

```
pl = plist('type', 'bandpass', 'order', 2, 'fs', 3.247, 'fc', [5e-4 5e-2]);
bp = miir(pl);
```

`bp` is a `filter` object. Although we could set units, we don't need units here since the input and output are `rad` and therefore the filter will remain unitless. The response of the filter can be easily evaluated using the `resp` method, showing us the expected shape. We specify the frequecy range and the number of points we want with a plist.

```
bp.resp(plist('f1', 1e-4, 'f2', 1, 'nf', 200));
```
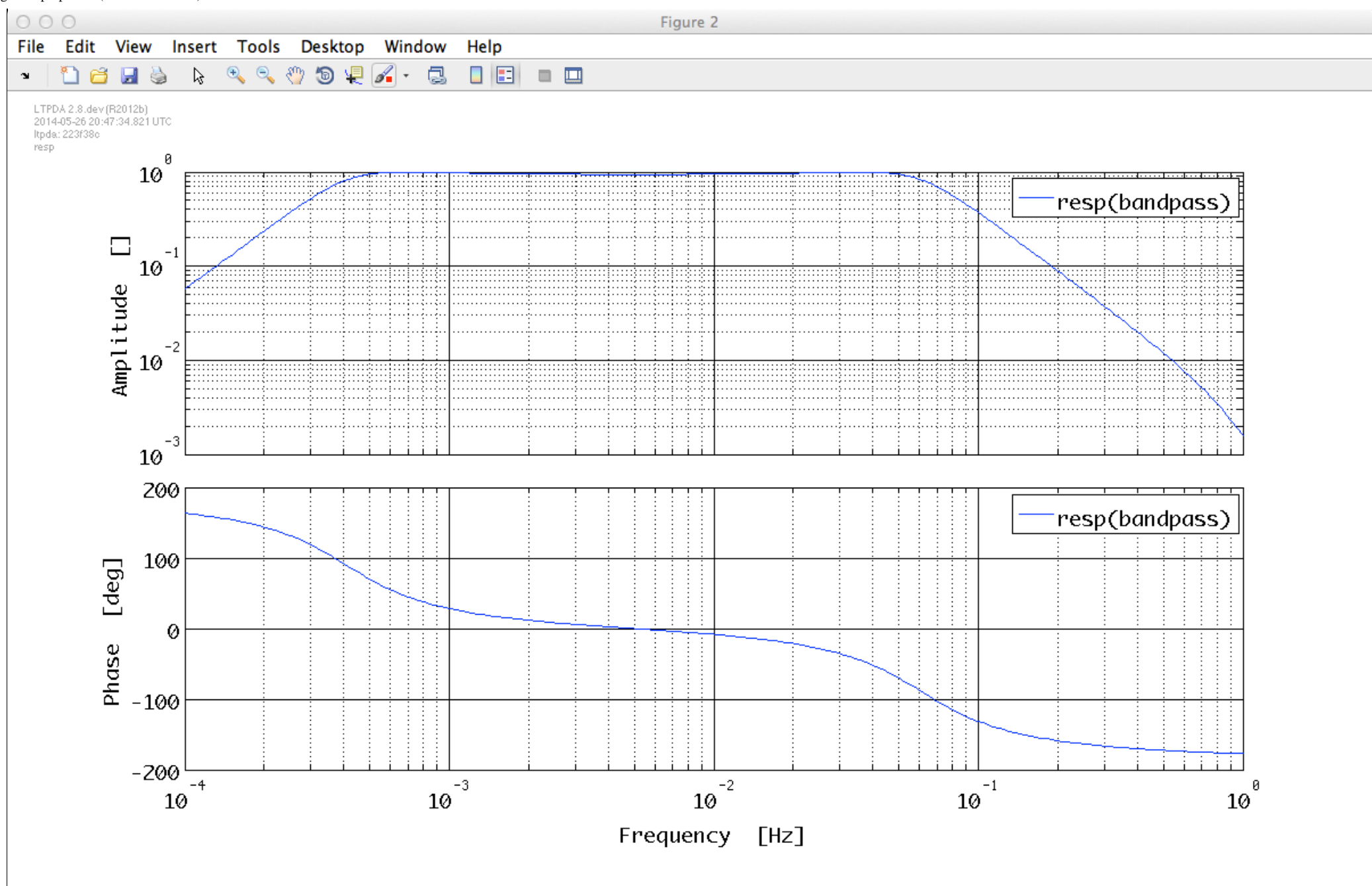
Once we have the `filter` object defined, to apply it to the data is straightforward. The standard method to do so is `filter(AO, filter_obj)`, however we will use here the `filtfilt` method which processes the data in both the forward and reverse direction producing a zero-phase filter but also removing transients.

```
    df = filtfilt(d, bp);
```

We can see how the filter has removed the trend by plotting both time series:

```
    df.setPlotinfo();
    iplot(d, df)
```

Please notice here that we had to reset the `plotinfo` properties that were propagated by the `filtfilt` method.
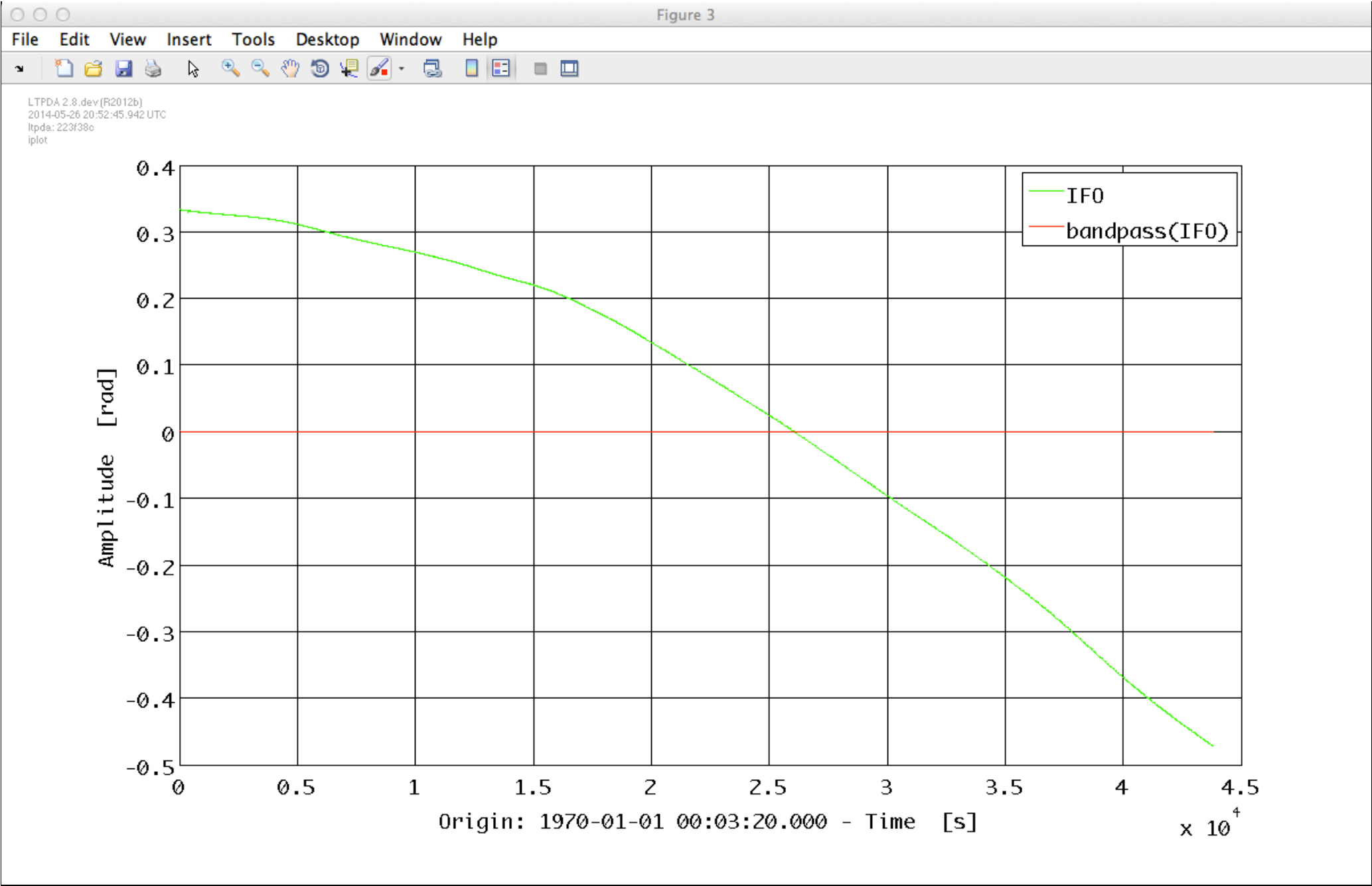
We are now ready to estimate the power spectrum using `psd` and the following parameters:

Key	Value
NFFT	

SCALE	1e4
	'PSD'

This can be done by typing the following in the command window

```
pl = plist('Nfft', 1e4, 'scale', 'PSD');
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.sqrt.iplot
```

By defining filter properties (LTPDA Toolbox)

By defining filter properties (LTPDA Toolbox)

©LTP Team
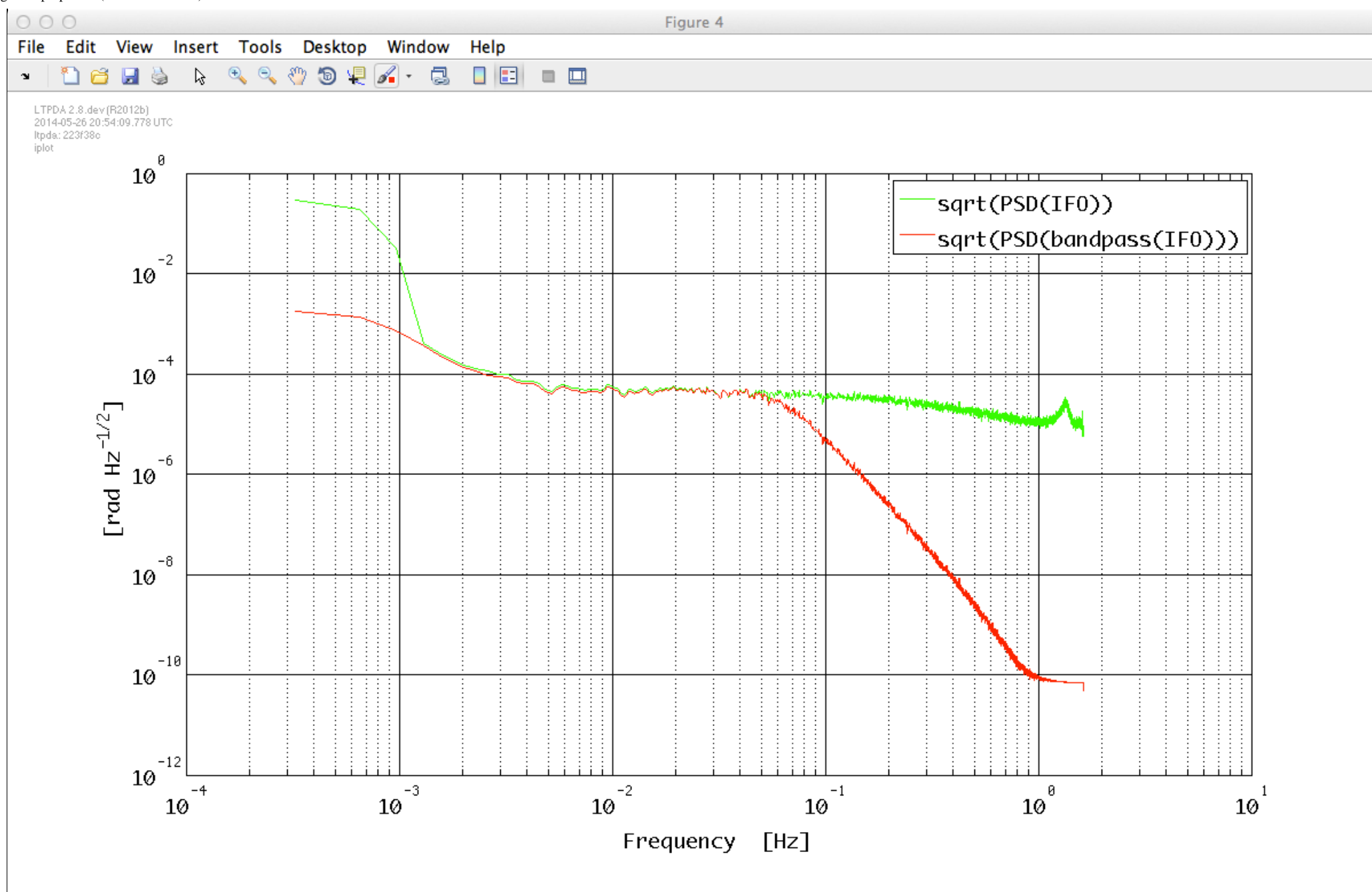
IFO/Temperature Example – Simulation

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.
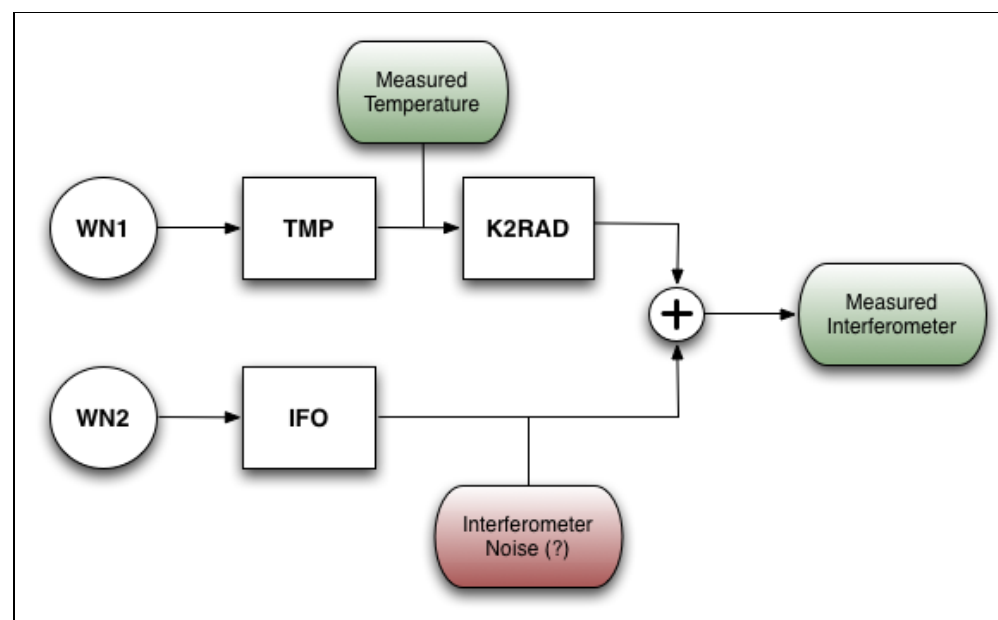
Since we've been through the same steps that you need to apply here in the previous section we will give here the step by step description of the task and let you play with the models.

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_mod = pzmodel(10, 1e-5, []);
TMP_mod.setOunits('K')
TMP_mod.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_mod * TMP_mod, IFO_mod, pl)
```
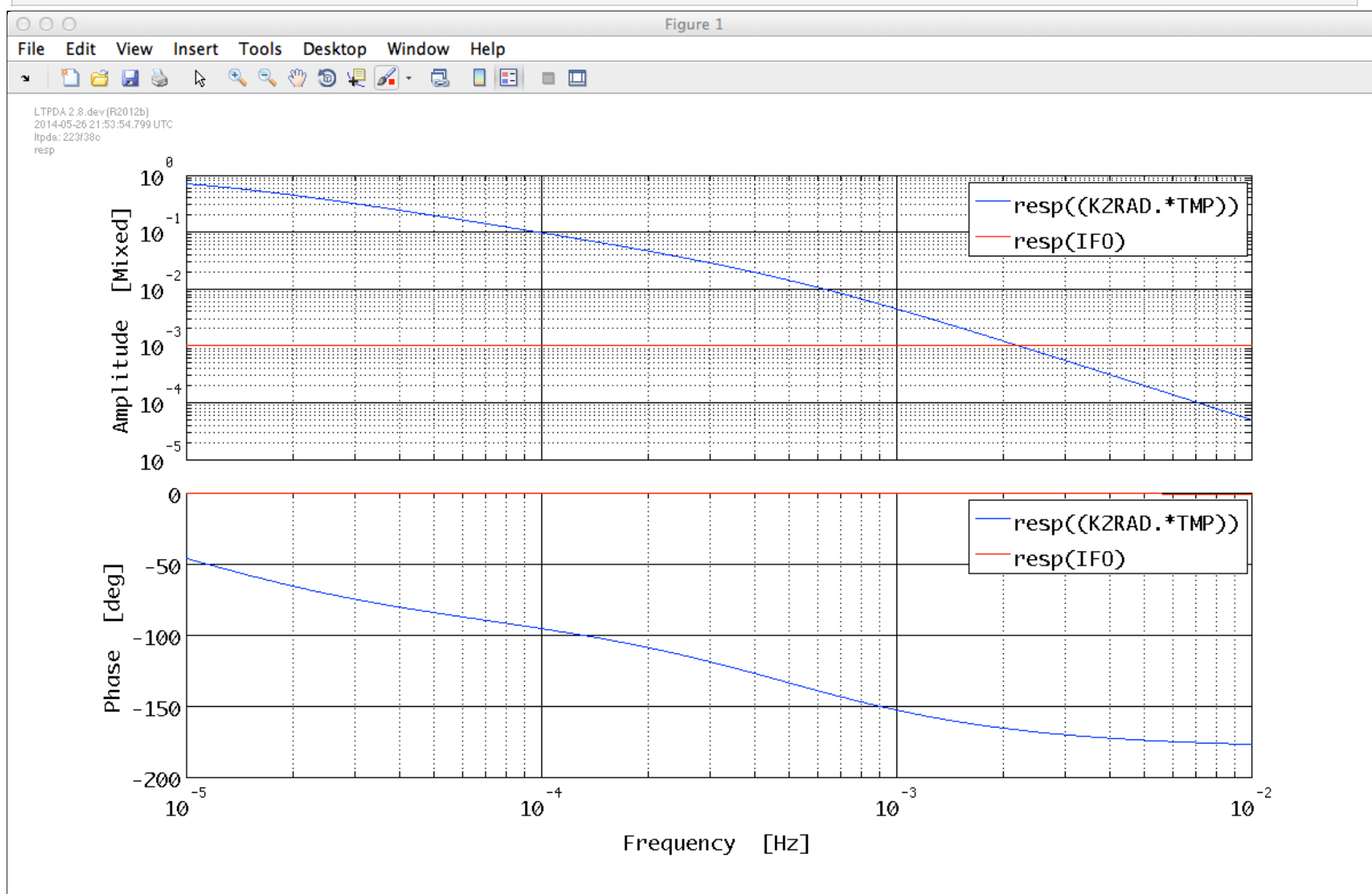
Discretize the models

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 functions (TMP_mod, IFO_mod, K2RAD_mod) with the `miir` constructor

For example, the model related to temperature noise would be discretized like this:

```
TMPd = miir(TMP_mod, 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 for instance use the following settings

Key	Value
`'name'`	**'WN1'**
`'tsfcn'`	**'randn(size(t))'**
`'fs'`	**1**
`'nsecs'`	**250000**

Key	Value
`'name'`	**'WN2'**
`'tsfcn'`	**'randn(size(t))'**
`'fs'`	**1**
`'nsecs'`	**250000**

Generate the noise data streams

For each noise source you will need to apply the filter that you have designed to the white noise data:

STEP 6: Filter white noise WN1 with the TMPd filter

For example, following our notation:

```
T = filter(WN1, TMPd);
```

STEP 7: Filter white noise WN2 with the IFOd filter

Temperature and interferometric noise are uncorrelated, so we need to use here the second noise time series **WN2**.

For example, following our notation:

```
IFO = filter(WN2, IFOd);
```

STEP 8: Filter white noise WN1 with the TMPd and the K2RADd filter

In this case you need to apply both filters in serial, you can do this in one command by using the 'bank' property of the `filter` method.

Hint: you can input a vector of filters into the `filter` method and ask it to filter the data in 'parallel' or in 'serial' (the one we are interested here) by doing the following

```
K2RAD = filter(WN1, [TMPd K2RADd], plist('bank', 'serial'));
```

STEP 9: Add the IFO noise to the K2RAD noise

At this point the IFO represents the purely interferometric noise and the K2RAD the contribution to interferometric noise coming from temperature. You need to add both to get the final interferometric data. This only requires to add both AOs.

Perform the noise projection

Here we will reproduce the main steps performed in topic 3 analysis: power spectral and transfer function estimation.

STEP 10: Split the data streams

We will do the analysis with data in the region going from 1e5 to 2e5 seconds to avoid initial transients. You must then split your two data streams introducing the following parameters in the `split` method.

Key	Value
'times'	[1e5 2e5]

After the splitting you must have two data streams that plot together should look like the ones below. The code should be similar to one in the

following lines:

```
Ts = split(T, plist('times', [1e5 2e5]));
Ts.setName('Temperature');
IFOs = split(IFO_all, plist('times', [1e5 2e5]));
IFOs.setName('Interferometer');
% Plot in different panels with 'subplots' options
iplot(Ts, IFOs, plist('arrangement', 'subplots'))
```
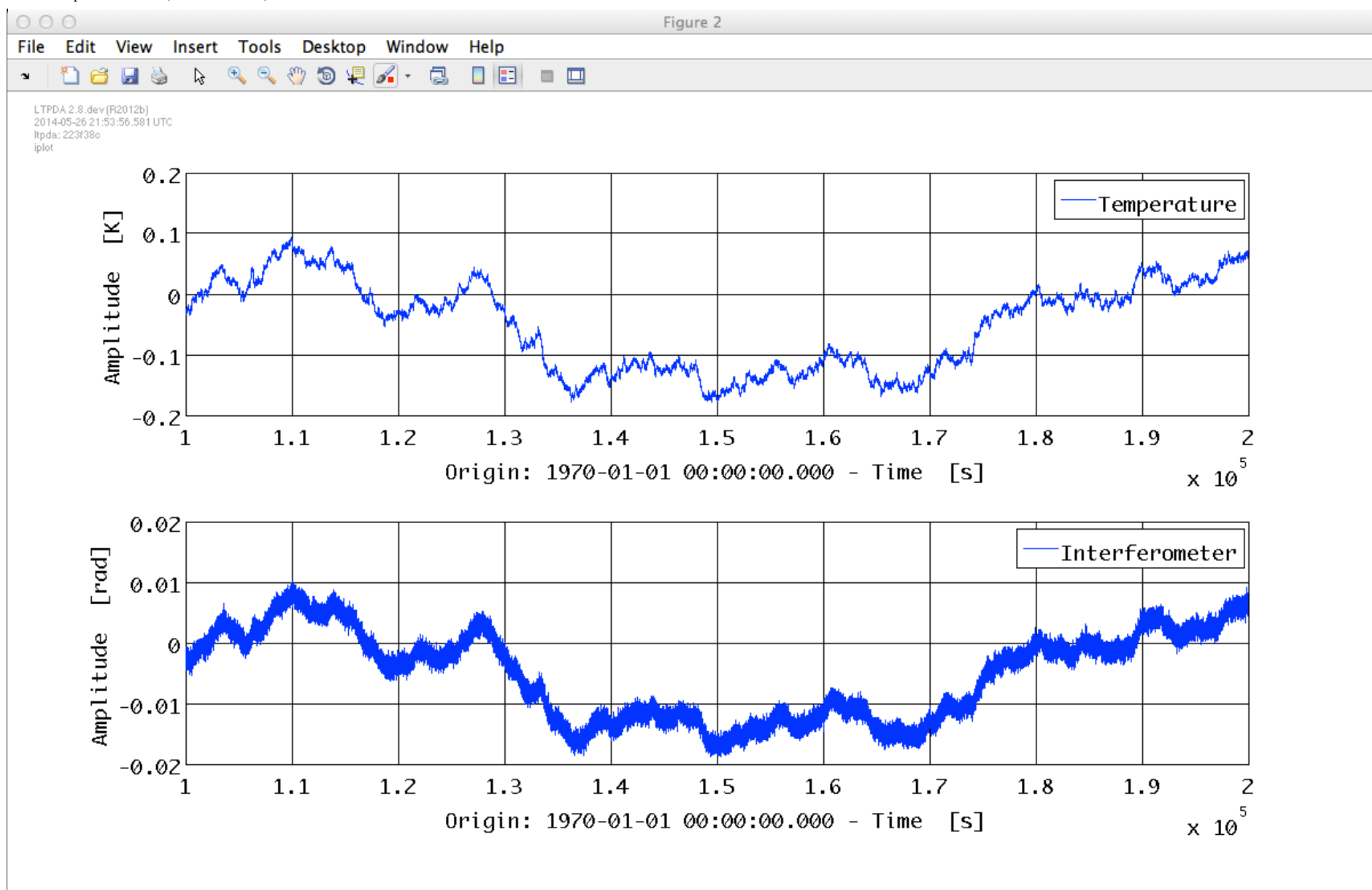
STEP 11: Compute power spectral estimates for the temperature and interferometric data

Here you need to apply `lpsd` or `psd` methods. For example:

```
pl_psd = plist('order', 1, 'scale', 'ASD');
[psd_T, psd_IFO] = lpsd(Ts, IFOs, pl_psd);
iplot(psd_T, psd_IFO, plist('arrangement', 'subplots'))
```

The resulting spectrum should look like this
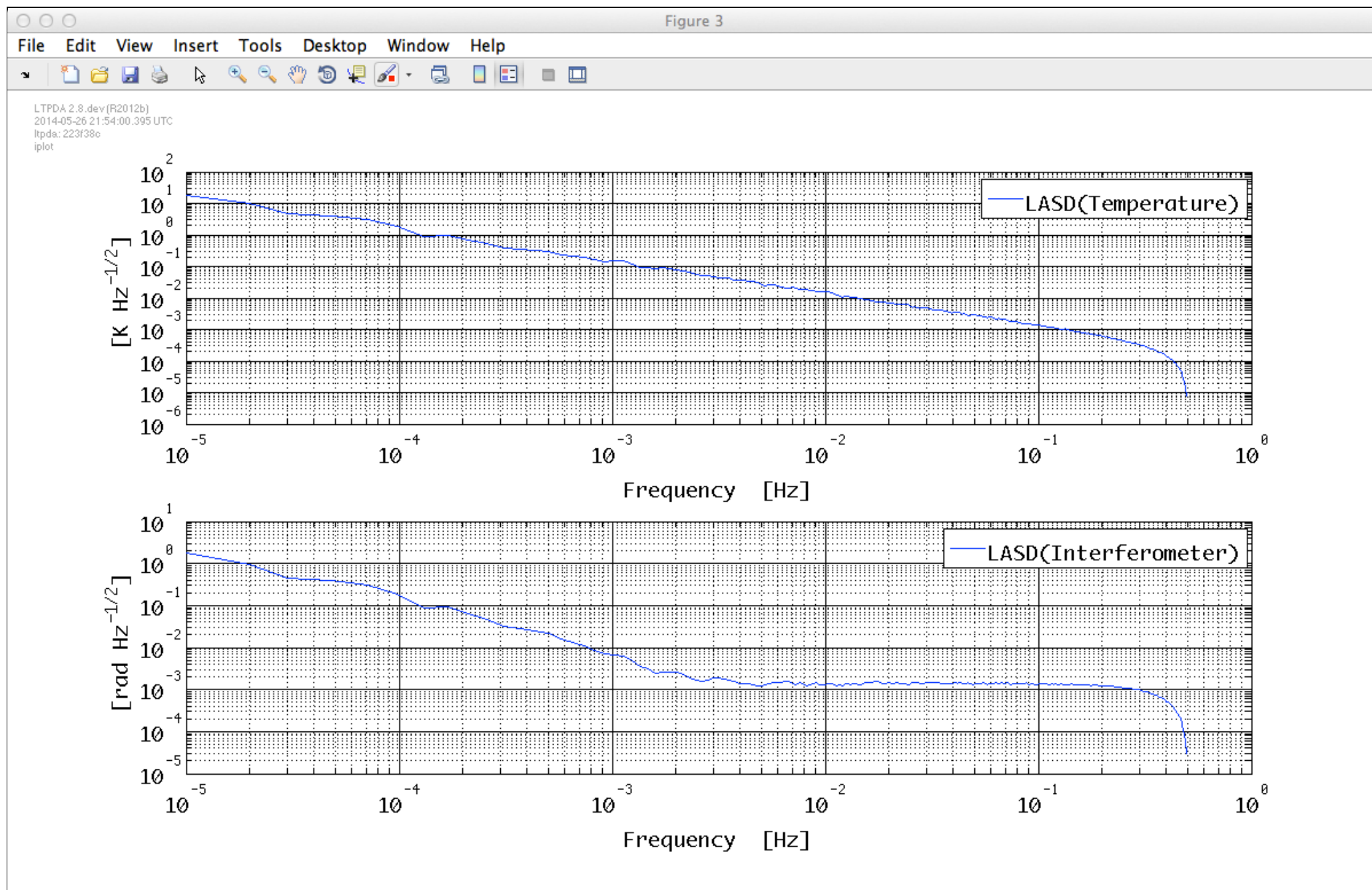
STEP 12: Compute transfer function estimate for the temperature and interferometric data

Here you need to apply `ltfe` or `tfe` methods. For example:

```
    T2IFO = ltfe(Ts, IFOs);
    T2IFO.setName('Transfer function')
```

You can now compare the transfer function model with the estimation obtained from the data:

```
    pl_resp = plist('f1', 1e-5, 'f2', 1);
    iplot(T2IFO, resp(K2RADd, pl_resp))
```
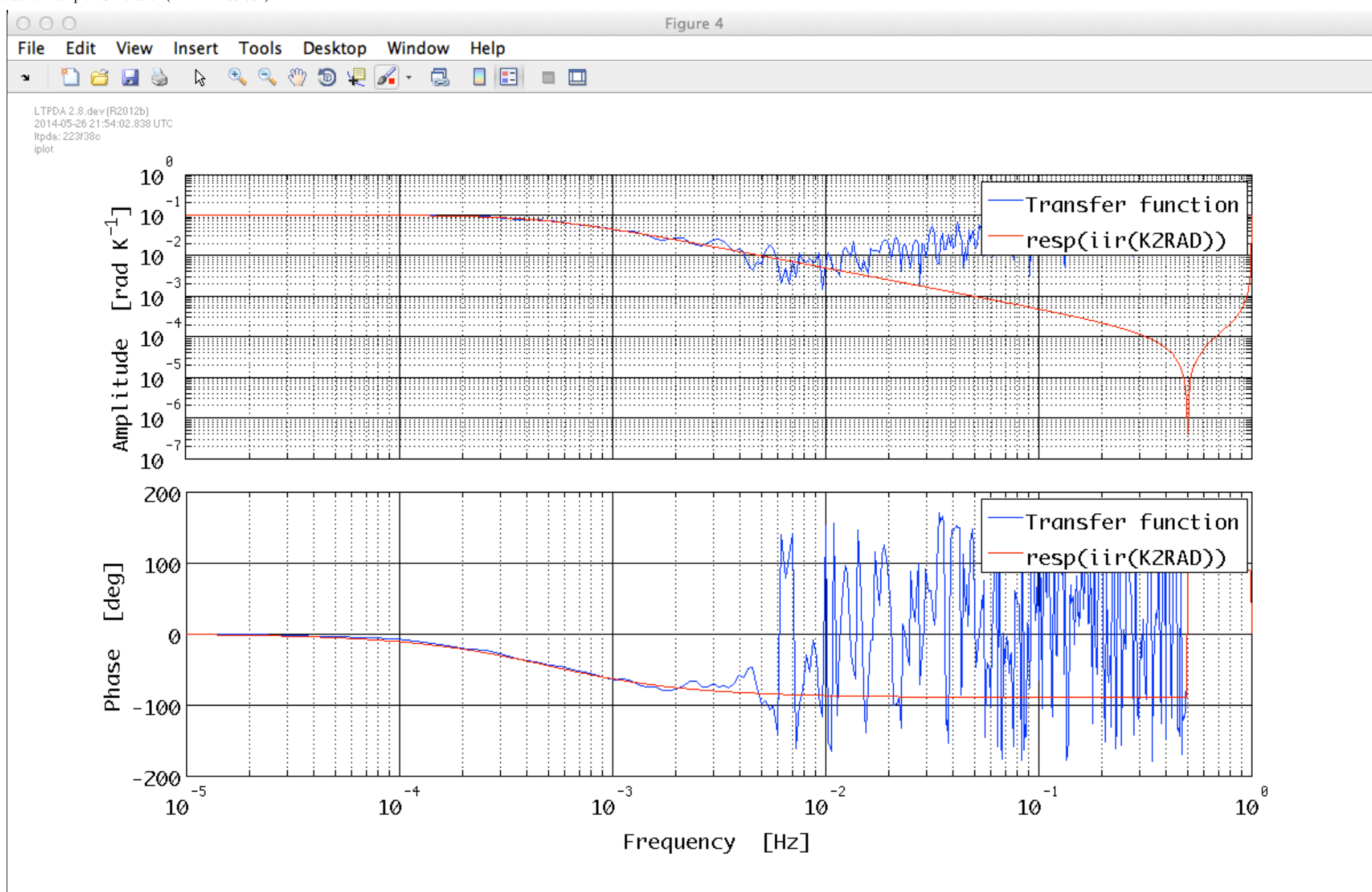
STEP 13: Project the temperature noise

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:

```matlab
% Compute projection
Projection = abs(T2IFO) .* psd_T;
Projection.simplifyYunits;
Projection.setName();
% Plot against interferometer noise
iplot(psd_IFO, Projection)
```

IFO/Temperature Example - Simulation (LTPDA Toolbox)

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡
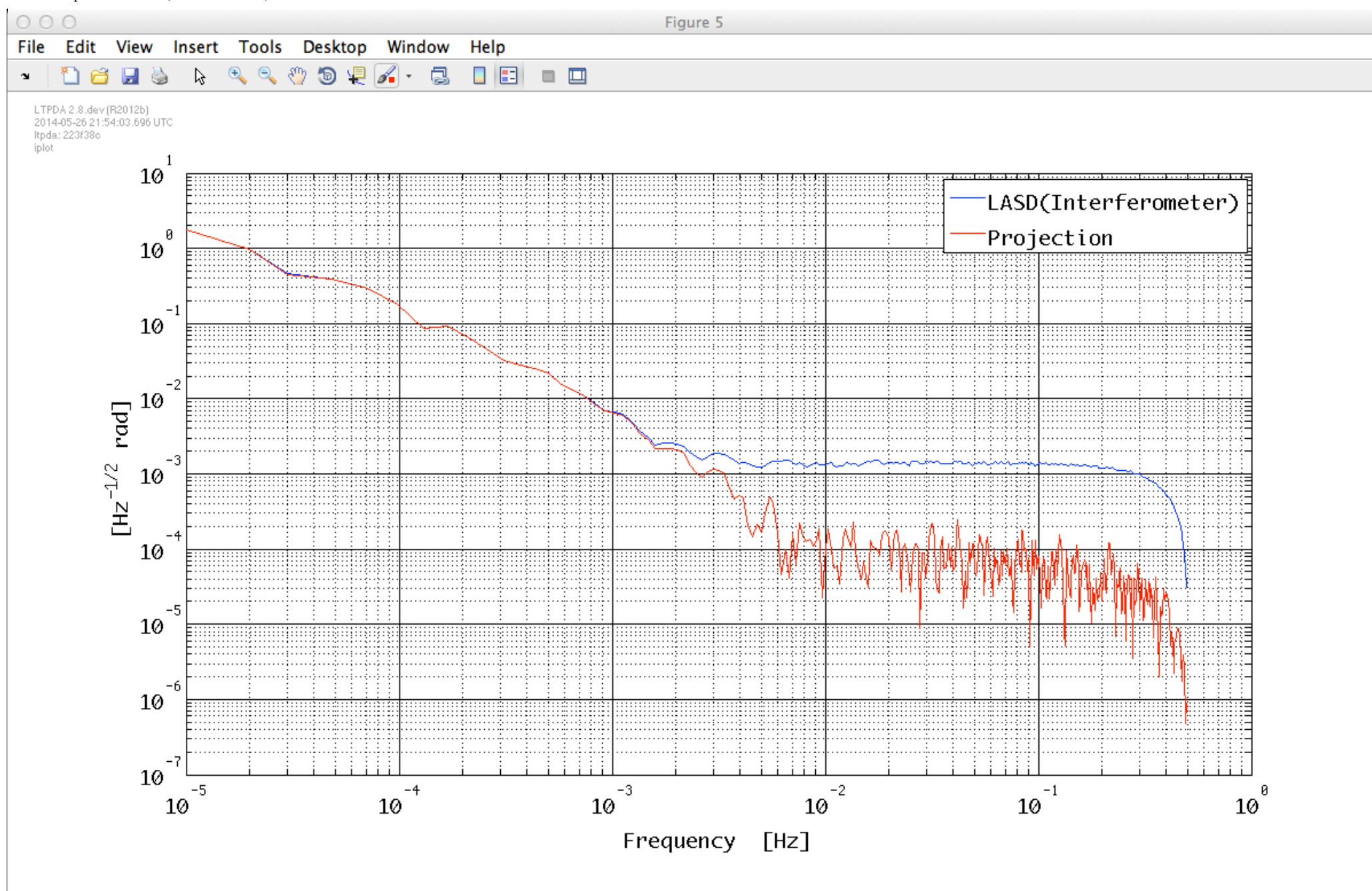
Topic 5 – Model fitting

Topic 5 of the training session aims to briefly introduce the advanced fitting capabilities offered by LTPDA. After working through the examples you should be familiar with:

- [System identification in z-domain](#)
- [Generation of noise with given psd](#)
- [Fitting time series with polynomials](#)
- [Non-linear least square fitting of time series](#)
- [Time-domain subtraction of temperature contribution to interferometer signal](#)

◀ IFO/Temperature Example – Simulation   System identification in z-domain ➡

©LTP Team

System identification in z-domain

System identification in Z-domain is performed with the function `ao/zDomainFit`. It is based on a modified version of the vector fitting algorithm that was adapted to fit in the Z-domain. Details of the agorithm can be found in the [Z-domain fit documentation page](#).

System identification in Z-domain

During this exercise we will:

1. Generate white noise
2. Filter white noise with a `miir` filter generated by a `pzmodel`
3. Extract the transfer function from data
4. Fit the transfer function with `ao/zDomainFit`
5. Check results

Let's start by generating some white noise.

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

This command generates a time series of gaussian distributed random noise with a sampling frequency (**'fs'**) of 1 Hz, 10000 seconds long (**'nsecs'**) and with **'yunits'** set to meters (**'m'**).

Now we are ready to move on the second step where we will:

- Build a pole-zero model (`pzmodel`)
- Construct a `miir` filter from the `pzmodel`
- Filter white noise data with the `miir` filter in order to obtain a colored noise time series.

```
    % Build a pole-zero model
pzm = pzmodel(1, [0.005 2], [0.05 4]);

    % Construct a miir filter from the pzmodel
filt   = miir(pzm, plist('fs', 1, 'name', 'None'));
filt.setIunits('m');
filt.setOunits('V');

% Filter white noise data with the filter in order to obtain a colored noise time series
ac = filter(a, filt);
ac.simplifyYunits;
```

We can calculate the PSD of the data streams in order to check the difference between the coloured noise and the white noise. Let's choose the log-scale estimation method.

```
axx  = lpsd(a);
acxx = lpsd(ac);
```

```
iplot(axx,acxx)
```

You should obtain a plot similar to this:
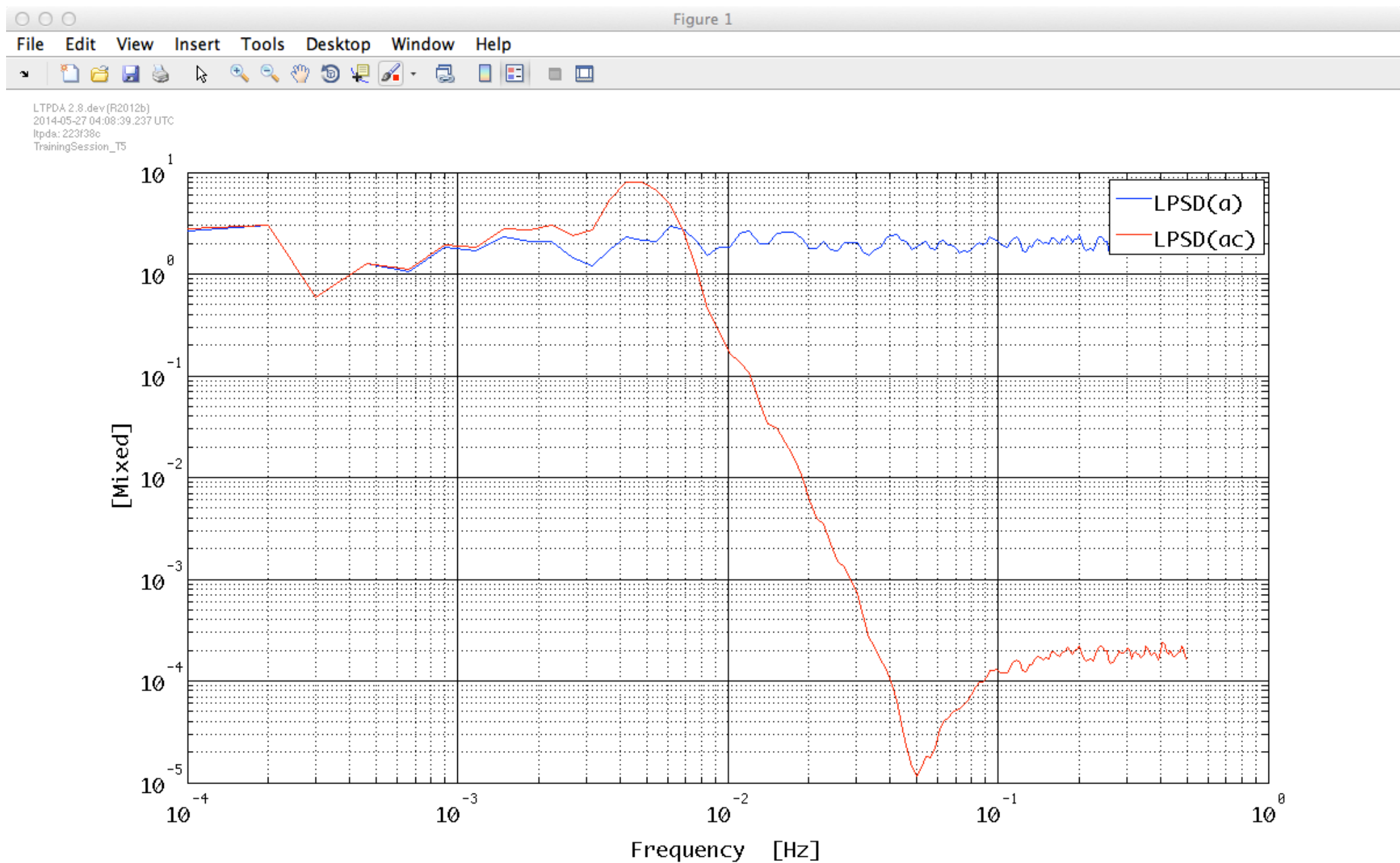
Let us move to the third step. We will generate the transfer function from the data and split it in order to remove the first 3 bins. The last operation is useful for the fitting process.

```
tf = ltfe(a, ac);
tfsp = split(tf, plist('frequencies', [5e-4 5e-1]));

iplot(tf, tfsp)
```

The plot should look like the following:
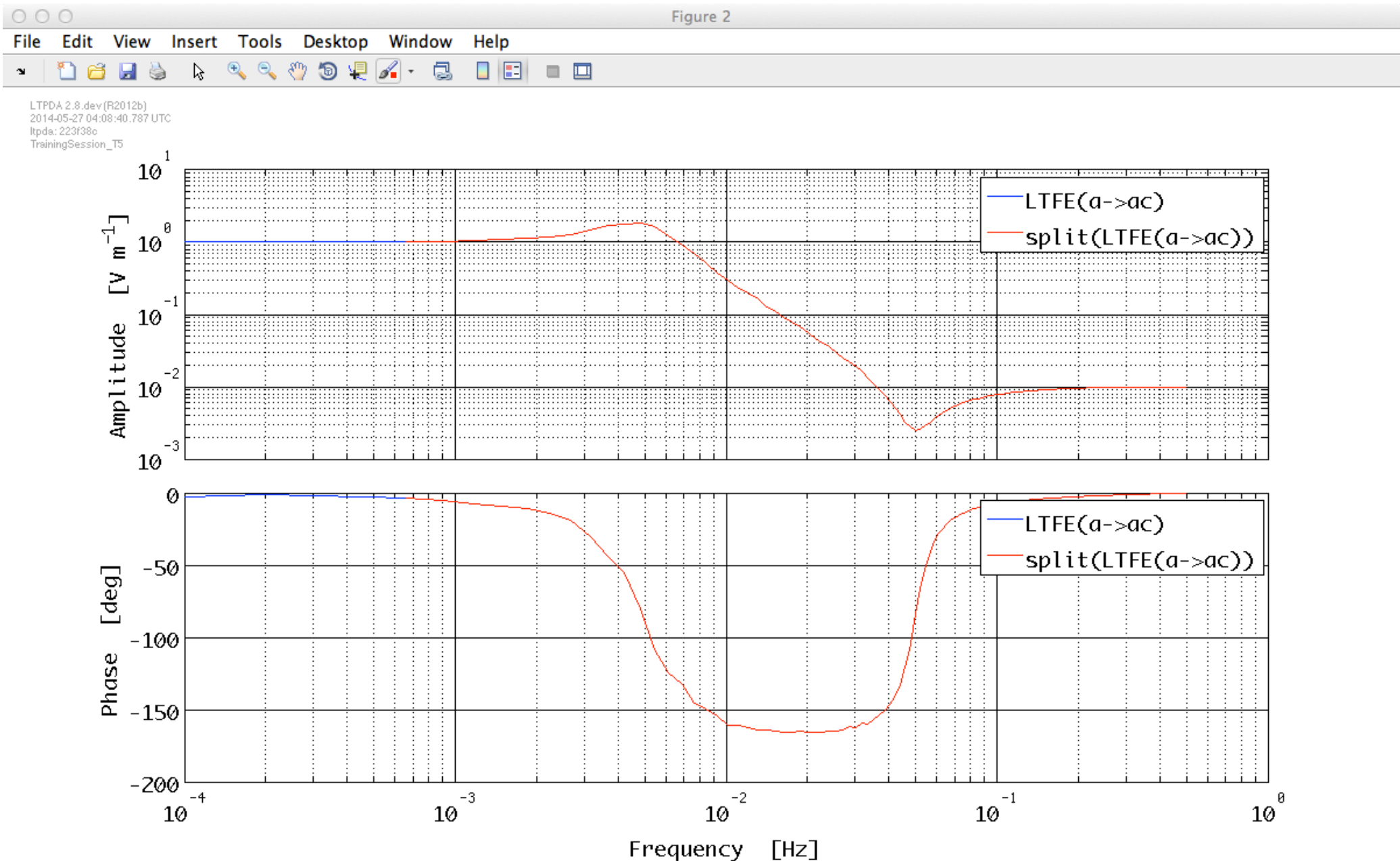
It is now the moment to start fitting with `zDomainFit`. As reported in the function help page we can run an automatic search loop to identify the proper model order. In such a case we have to define a set of conditions to check fit accuracy and to exit the fitting loop.

We can start checking Mean Squared Error and variation (CONDTYPE = 'MSE'). It checks if the normalized mean squared error is lower than the value specified in the parameter FITTOL and if the relative variation of the mean squared error is lower than the value specified in the parameter MSEVARTOL.

	Default		
Key	**Default Value**	**Options**	**Description**
CONDTYPE	'MSE'	**'MSE'** **'RLD'** **'RSF'**	Fit conditioning type. Admitted values are: 'MSE' Mean Squared Error and variation 'RLD' Log residuals difference and mean squared error variation 'RSF' Residuals spectral flatness and mean squared error variation
FITTOL	0.001	*none*	Fit tolerance.
MSEVARTOL	0.01	*none*	Mean Squared Error Variation – Check if the relative variation of the mean squared error is smaller than the value specified. This option is useful for finding the minimum of the Chi-squared.

You will find a list of all Parameters of zDomainFit here: [Parameter of zDomainFit](#)

Now let's run the fit:

```matlab
% Set up the parameters
plfit = plist('fs', 1, ...        % Sampling frequency for the model filters
    'AutoSearch', 'on', ...       % Automatically search for a good model
    'StartPolesOpt', 'clog', ...  % Define the properties of the starting poles - complex distributed in the unitary circle
    'maxiter', 50 ,...            % Maximum number of iteration per model order
    'minorder', 2, ...            % Minimum model order
    'maxorder', 9, ...            % Maximum model order
    'weightparam', 'abs', ...     % Assign weights as 1./abs(data)
    'condtype', 'MSE', ...        % Mean Squared Error and variation
    'fittol', 1e-2, ...           % Fit tolerance
    'msevartol', 1e-1, ...        % Mean Squared Error Variation tolerance
    'Plot', 'on', ...             % Set the plot on or off
    'ForceStability', 'on', ...   % Force to output a stable poles model
    'CheckProgress', 'off');      % Display fitting progress on the command window

% Do the fit
fobj = zDomainFit(tfsp, plfit);

% Set the input and output units for fitted model
fobj.setIunits('m');
fobj.setOunits('V');
```

When **'Plot'** parameter is set to **'on'** the function plots the fit progress.
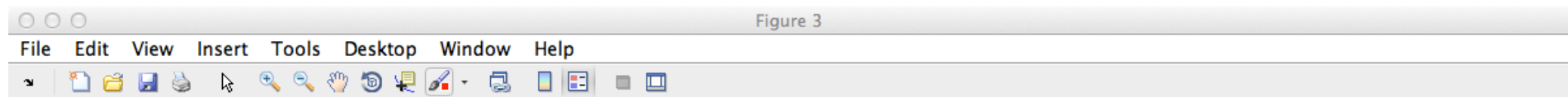
We can now check the result of our fitting procedures. We calculate the frequency response of the fitted models (filters) and compare them with the starting IIR filter response, then we will plot the percentage error on the filters magnitudes.

Note that the result of the fitting procedure is a `matrix` object containing a `filterbank` object, which itself contains a parallel bank of 3 IIR filters.

Note that at the moment we need to access the individual filter objects inside the `matrix` object that was the result of the fitting procedure.

```
% set plist for filter response
plrsp = plist('bank', 'parallel', 'f1', 1e-5, 'f2', 0.5, 'nf', 100, 'scale', 'log');

% compute the response of the original noise-shape filter
rfilt = resp(filt, plrsp);
rfilt.setName;

% compute the response of our fitted filter bank
rfobj = resp(fobj.filters, plrsp);
rfobj.setName;

% compare the responses
iplot(rfilt, rfobj)

% and the percentage error on the magnitude
pdiff = 100 .* abs((rfobj - rfilt) ./ rfilt);
pdiff.simplifyYunits;
iplot(pdiff, plist('YRanges', [1e-2 100]))
```

The first plot shows the response of the original filter and the fitted filter bank, whereas the second plot reports the percentage difference between fitted model and target filter magnitude.
As can be seen, the difference between filters magnitude is at most 10%.
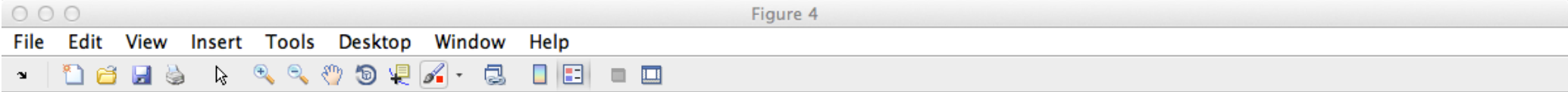
System identification in z-domain (LTPDA Toolbox)

System identification in z-domain (LTPDA Toolbox)

Figure 5

File   Edit   View   Insert   Tools   Desktop   Window   Help
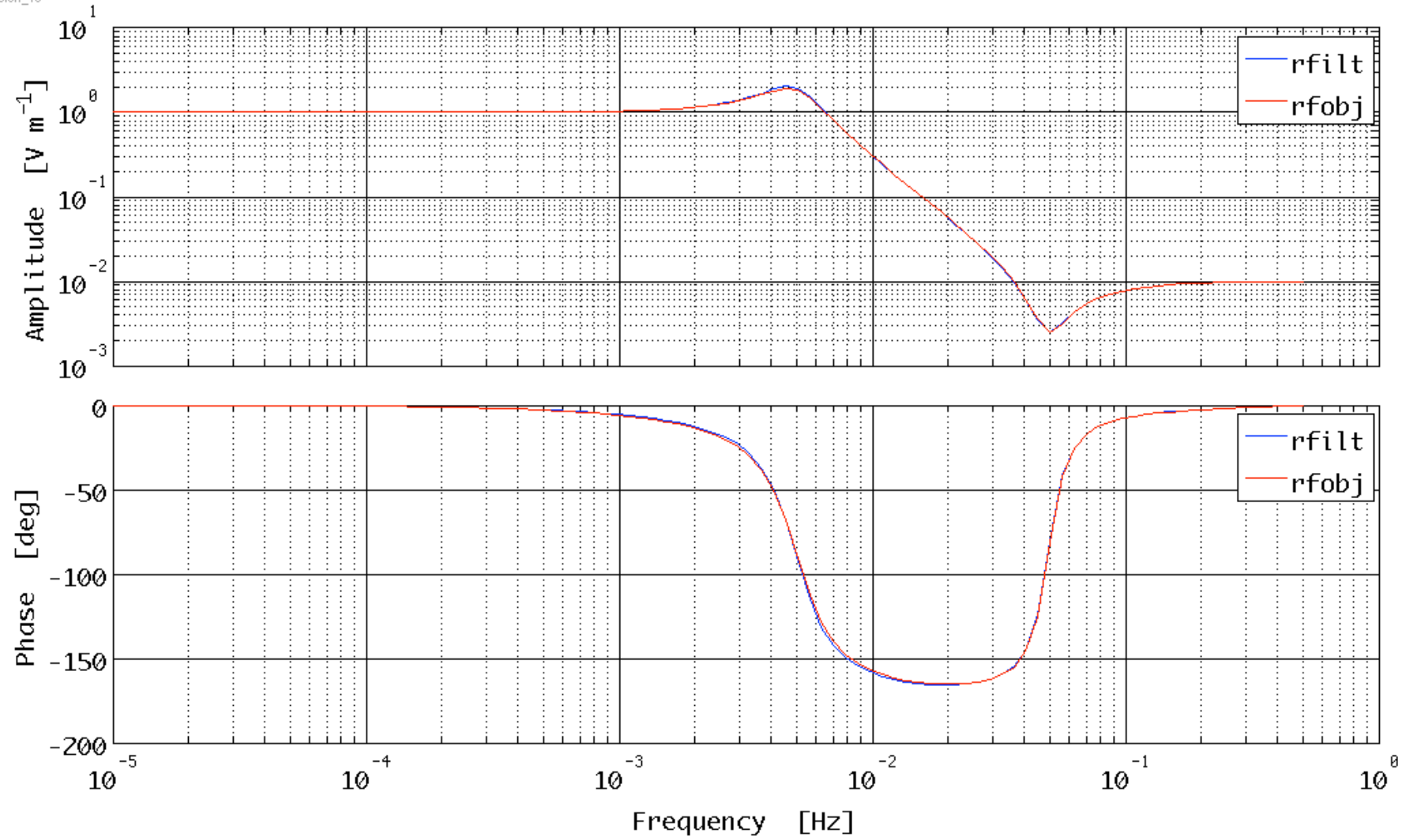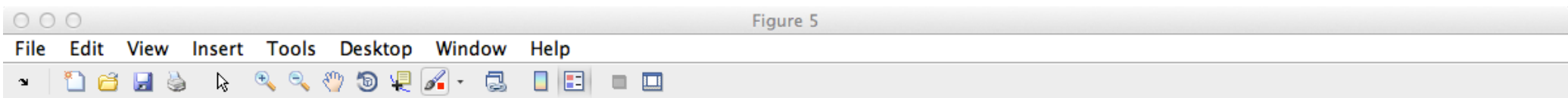
LTPDA 2.8.dev (R2012b)
2014-05-27 04:08:47.441 UTC
ltpda: 223f38c
TrainingSession_T5

(100 .* abs(((rfobj - rfilt) ./ rfilt)))

Frequency   [Hz]

◀ Topic 5 – Model fitting   Generation of noise with given PSD ▶

©LTP Team
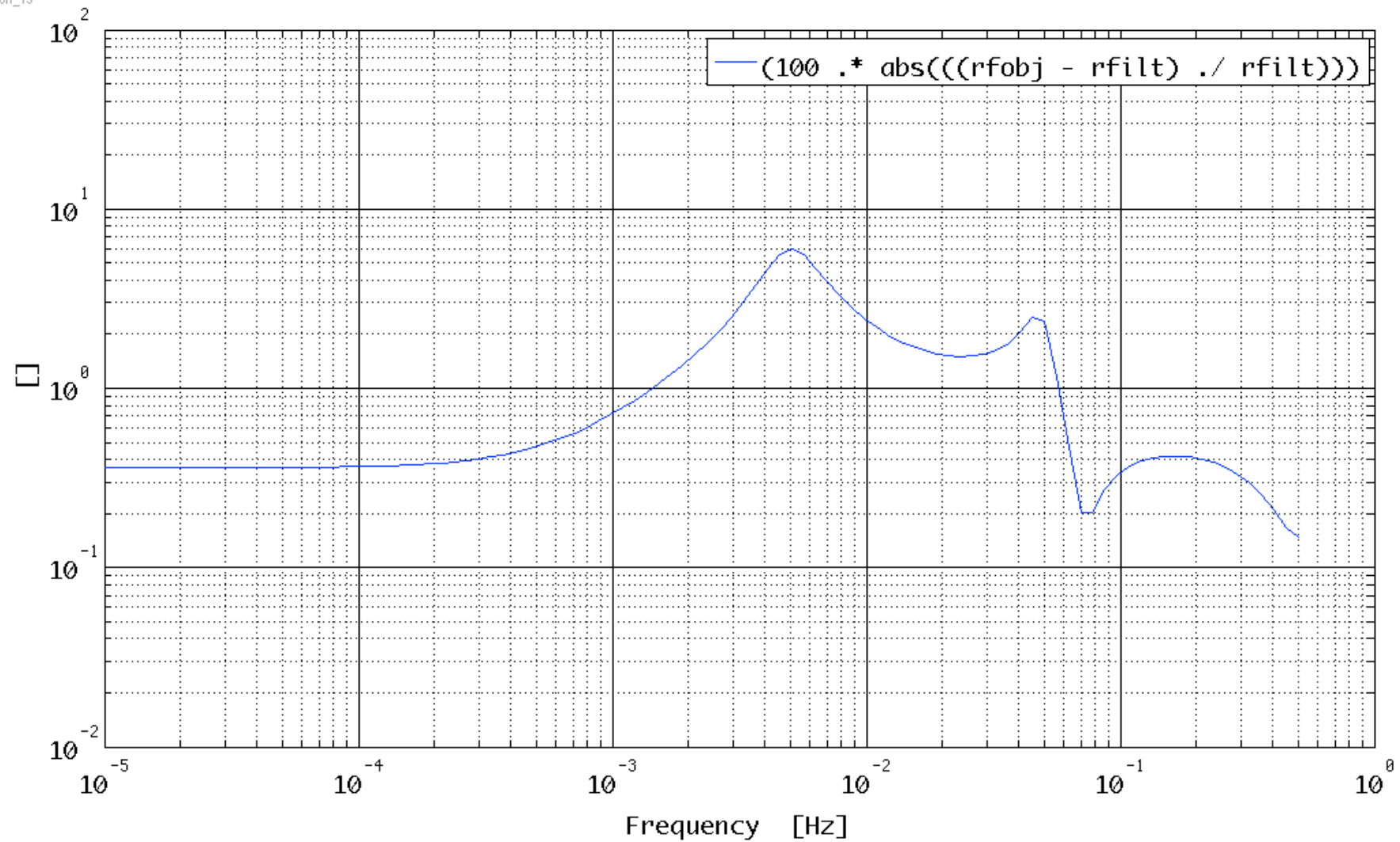
Generation of noise with given PSD

Generation of model noise is performed with the function `ao/noisegen1D`. Details on the algorithm can be found in [noisegen1D help page](#).

Generation of noise with given PSD

During this exercise we will:

1. Load from file an fsdata object with the model (obtained with a fit to the the PSD of test data with zDomainFit)
2. Genarate noise from this model
3. Compare PSD of the generated noise with original PSD

Let's open a new editor window and load the 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, sampled at 1 Hz. The command `setName` sets the name of the AO to be the same as the variable name, in this case `tn`.

Now let's calculate the PSD of our data. We apply some averaging, in order to decrease the fluctuations in the data.

```
tnxx = tn.psd(plist('Nfft', 2000));
```

Additionally, we load a smooth model that represents well our data. It was obtained, as described [here](#), by fitting the target PSD with z-domain fitting. We load the data from disk, and plot them against the target PSD. Please note that the colouring filters (whose response represents our model) have no units, so we force them to be the same as the PSD we compare with:

```
fmod = ao(plist('filename', 'topic5/T5_Ex03_ModelNoise.xml'));
iplot(tnxx, fmod.setYunits(tnxx.yunits))
```

The comparison beyween the target PSD and the model should look like:
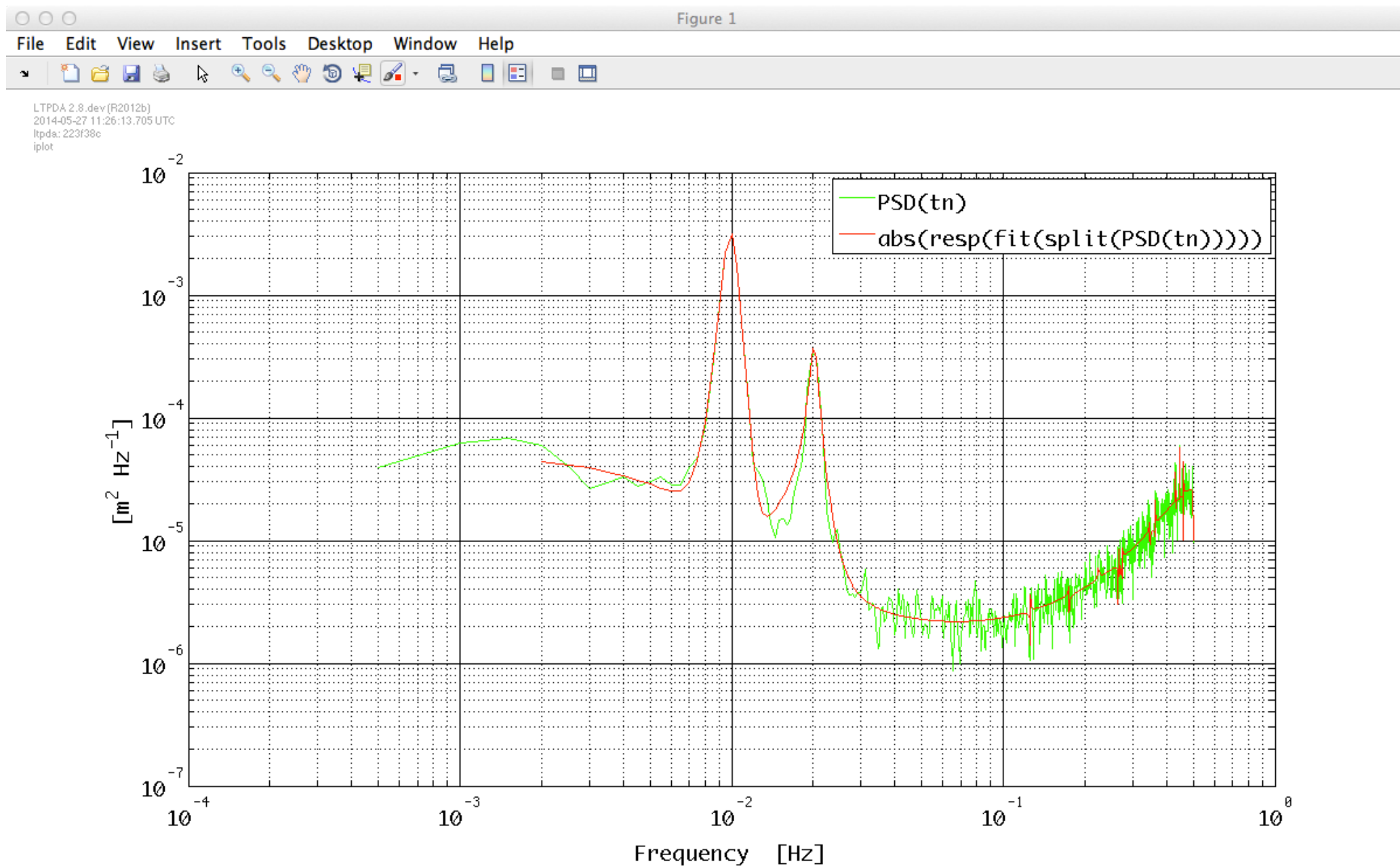
We can now start the noise generation process. The first step is to generate a white time series Analysis Object, with the desired duration and units:

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

Then we run the noise coloring process calling `noisegen1D`

```matlab
plng = plist(...
    'model', 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

ac = noisegen1D(a, plng);
```

Let's check the result. We calculate the PSD of the generated noise and compare it with the PSD of the target data.

```matlab
acxx = ac.psd(plist('Nfft', 2000));
iplot(tnxx, acxx)
```

As can be seen, the result is in quite satisfactory agreement with the original data
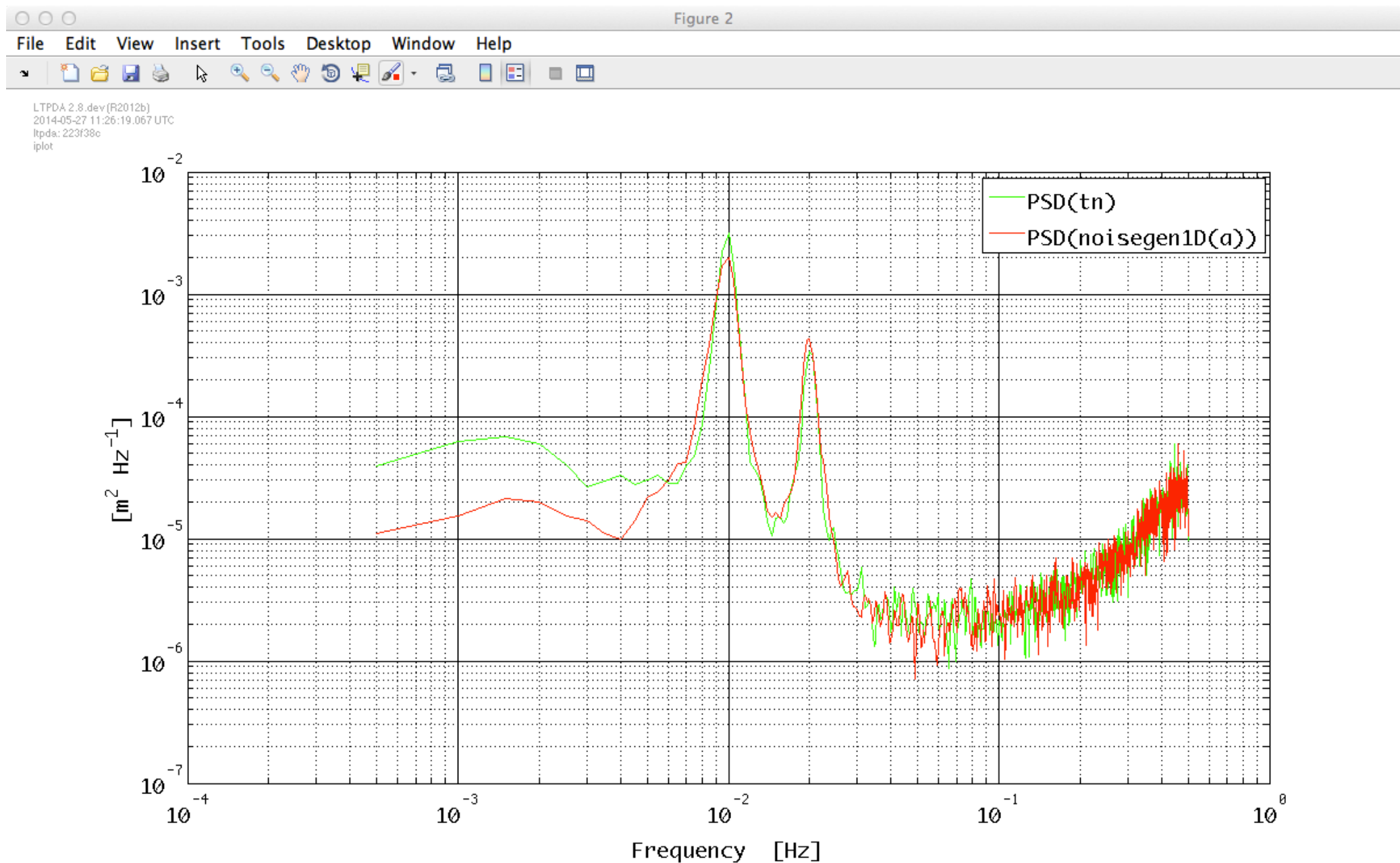
Appendix: evaluation of the model for the noise PSD

The smooth model for the data, that we used to reproduce the synthesized noise, was actually obtained by applying the procedure of z-domain fitting

that we discussed in [the previous section](). If you want to practise more with this fitting technique, we repost here the steps.

In order to extract a reliable model from PSD data we need to discard the first frequency bins; we do that by means of the `split` method.

```
tnxxr = split(tnxx, plist('frequencies', [2e-3 +inf]));
iplot(tnxx, tnxxr)
```

The result should look like:
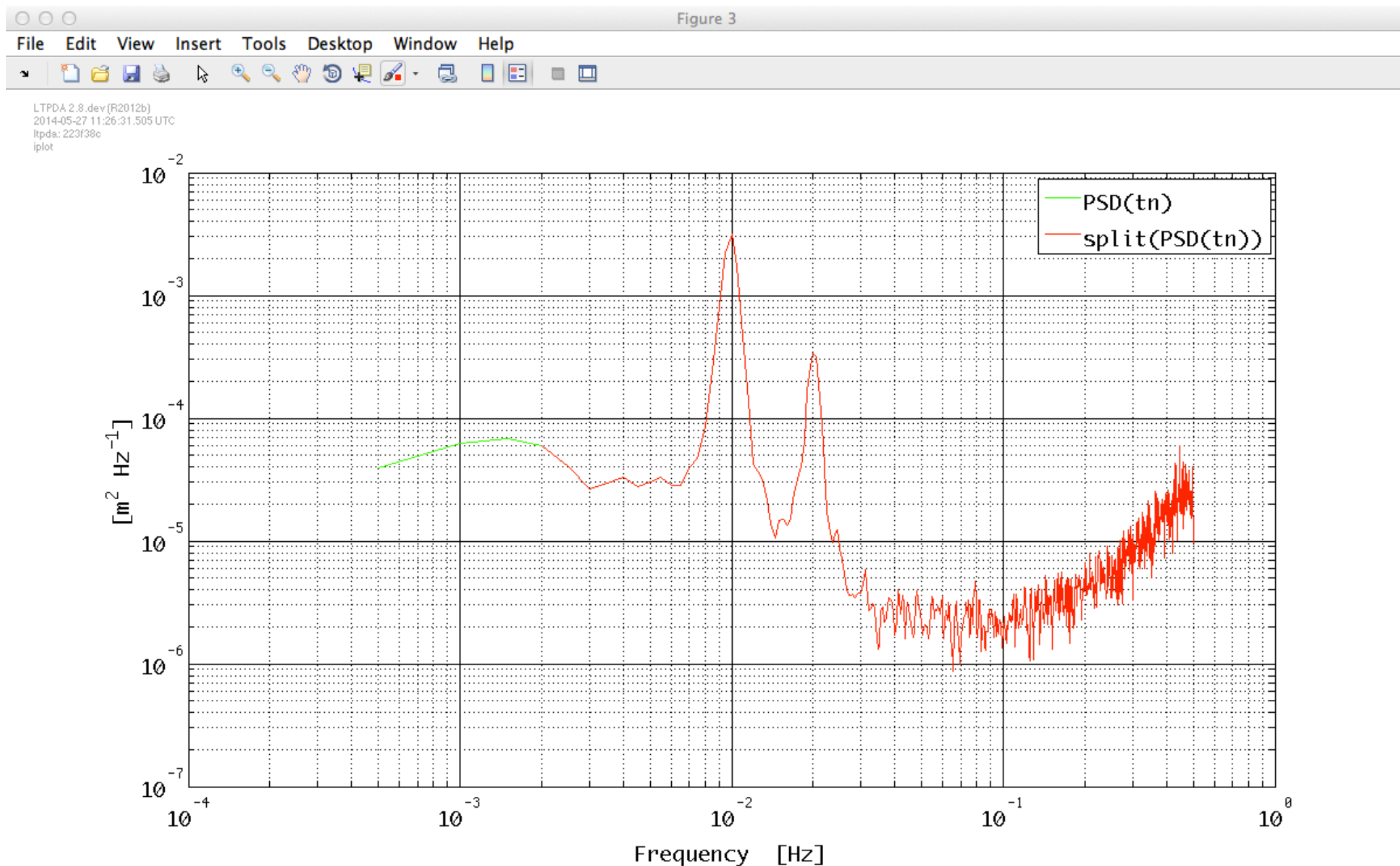
Now it's the moment to fit our PSD to extract a smooth model to pass to the noise generator. First of all we should define a set of proper weights for our fit process. We smooth our PSD data and then define the weights as the inverse of the absolute value of the smoothed PSD. This should help the fit function to do a good job with noisy data. It is worth noting here that weights are not univocally defined and there could be 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:
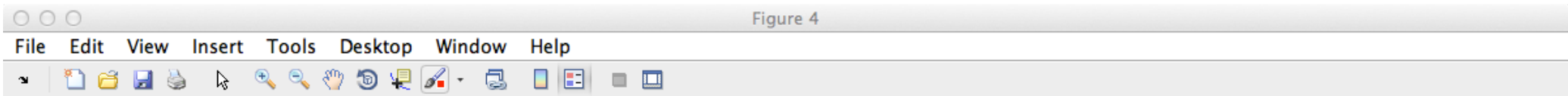
Now let's run an automatic search for the proper model and pass the set of externally defined weights. The first output of `zDomainFit` is a `miir` filter model; the second output is the model response. Note that we are setting 'ResFlat' parameter to define the exit condition. 'ResFlat' check the spectral flatness of the absolute value of the fit residuals.

```
        plfit = plist('AutoSearch', 'on', ...
                      'StartPolesOpt','clog', ...
                      'maxiter', 50, ...
                      'minorder', 30, ...
                      'maxorder', 45, ...
                      'weights', wgh  ,... % assign externally calculated weights
                      'condtype', 'MSE', ...
                      'msevartol', 0.1, ...
                      'fittol', 0.01, ...
                      'Plot', 'on', ...
                      'ForceStability', 'off', ...
                      'CheckProgress', 'off');

        % Do the fit
        fit_results = zDomainFit(tnxxr, plfit);
```

Fit result should look like:
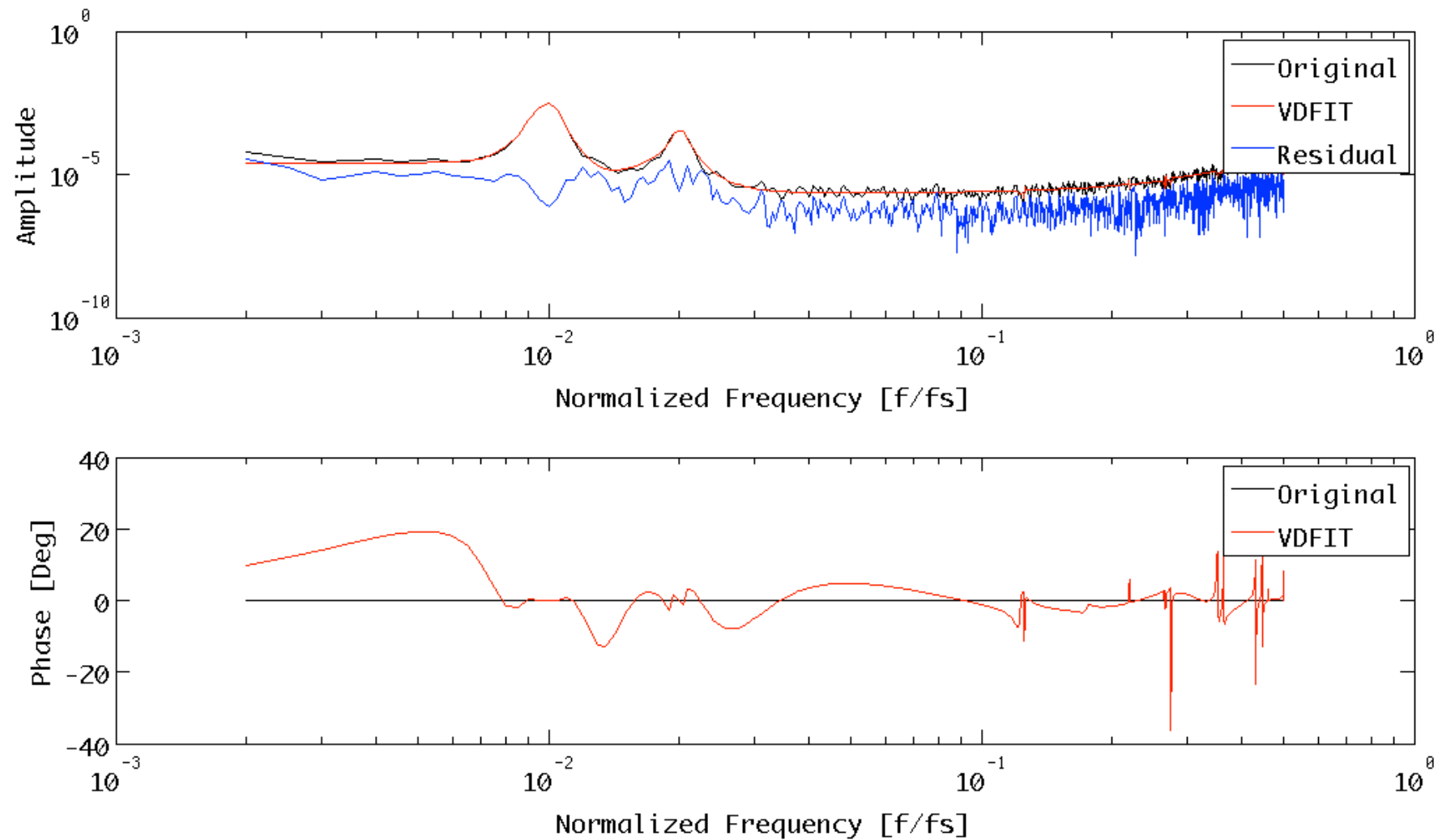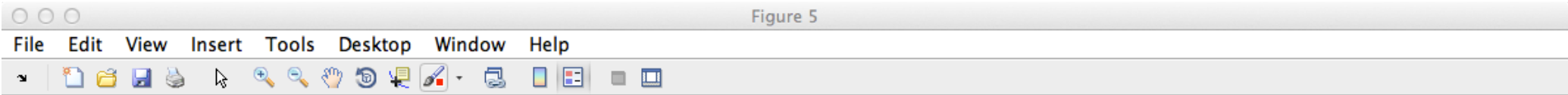
The fit results consist in a `filterbank` object; we can evaluate the absolute values of the response of these filters at the frequencies defined by the `x` field of the PSD we want to match.

```matlab
% Evaluate the absolute value of the response of the colouring filter
b = resp(fit_results, plist('f', tnxxr.x));
b.abs;

% Save the model on disk
b.save(plist('filename', 'topic5/T5_Ex03_ModelNoise.xml'));
```

◀ System identification in z-domain   Fitting time series with polynomials ▶

©LTP Team

Fitting time series with polynomials

Fitting time series with polynomials exploits the function `ao/polyfit`. Details on the agorithm can be found in the [appropriate help page](#).

Fitting time series with polynomials

During this exercise we will:

1. Load time series noise
2. Fit data with ao/polyfit
3. Check results

Let's open a new editor window and load test data.

```
a = ao(plist('filename', 'topic5/T5_Ex04_TestNoise.xml'));
a.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 output of the polifit method is a parameter estimation object (`pest`–object). This object contains the coefficients of the fitted polynomial.

```
---- pest 1 ----
      name: polyfit(a)
 parameters:
          P1:    9.1738842e-15 +-    5.0781997e-14 [m s^(-6)]
          P2:   -1.0105535e-11 +-    1.5272591e-10 [m s^(-5)]
          P3:    1.1467469e-08 +-    1.7567488e-07 [m s^(-4)]
          P4:   -2.8397623e-06 +-    9.6440306e-05 [m s^(-3)]
          P5:    -0.0044447573 +-      0.025624775 [m s^(-2)]
          P6:      0.13779743 +-        2.9321071 [m s^(-1)]
          P7:       47.519031 +-        105.42745 [m]

         pdf: []
         cov: [7x7], ([2.57881118597054e-27 -7.72869712435383e-24 8.77207944145014e-21 -4.67120229268015e-18 1.16517894641315e-15 -1.16052934006908e-
13 2.73279821621369e-12;-7.72869712435383e-24 2.33252022407021e-20 -2.66952587411623e-17 1.43594424166059e-14 -3.6267488596848e-12
3.66996931772136e-10 -8.8246258706484e-09;8.77207944145014e-21 -2.66952587411623e-17 3.08616635389935e-14 -1.68087475028813e-11 4.3130451226497e-09
-4.4559741785348e-07 1.10252789031329e-05;-4.67120229268015e-18 1.43594424166059e-14 -1.68087475028813e-11 9.30073263760312e-09 -2.43649023481382e-
06 0.00025896803296049 -0.00667866538500405;1.16517894641315e-15 -3.6267488596848e-12 4.3130451226497e-09 -2.43649023481382e-06 0.000656629115560015
-0.0727240078893827 2.0026099434953;-1.16052934006908e-13 3.66996931772136e-10 -4.4559741785348e-07 0.00025896803296049 -0.0727240078893827
8.59725202675112 -266.884491131194;2.73279821621369e-12 -8.8246258706484e-09 1.10252789031329e-05 -0.00667866538500405 2.0026099434953 -
266.884491131194 11114.9468971226])
        corr: []
       chain: []
        chi2: 1
         dof: 993
       models: smodel(P1*X.^6 + P2*X.^5 + P3*X.^4 + P4*X.^3 + P5*X.^2 + P6*X.^1 + P7*X.^0)
 description:
        UUID: 08970689-fe18-4581-8040-cff939363a32
```

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

Once we have the `pest` object with the coefficients, we can evaluate it. In order to construct an object with the same time base we can pass the input AO, and specify to use its 'x' field to build the 'x' field of the output.

```
b = p.eval(a, plist('type', 'tsdata', 'xfield', 'x'));
b.setName();
```

Now, let's check the fit results with some plotting. We can compare data with fitted model and look at the fit residuals.

```
iplot(a, b)
iplot(a - b)
```

Fitting time series with polynomials (LTPDA Toolbox)

Figure 1

File Edit View Insert Tools Desktop Window Help
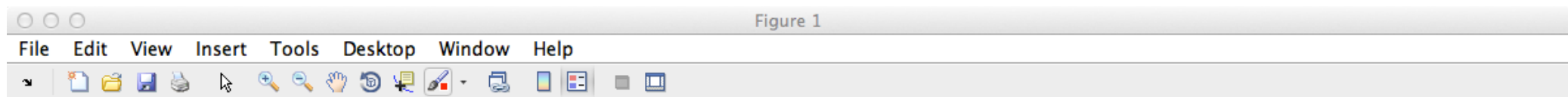
LTPDA 2.8.dev (R2012b)
2014-05-27 12:25:06.140 UTC
ltpda: 223f38c
TrainingSession_T5_3

Origin: 1970-01-01 00:00:00.000 - Time [s]
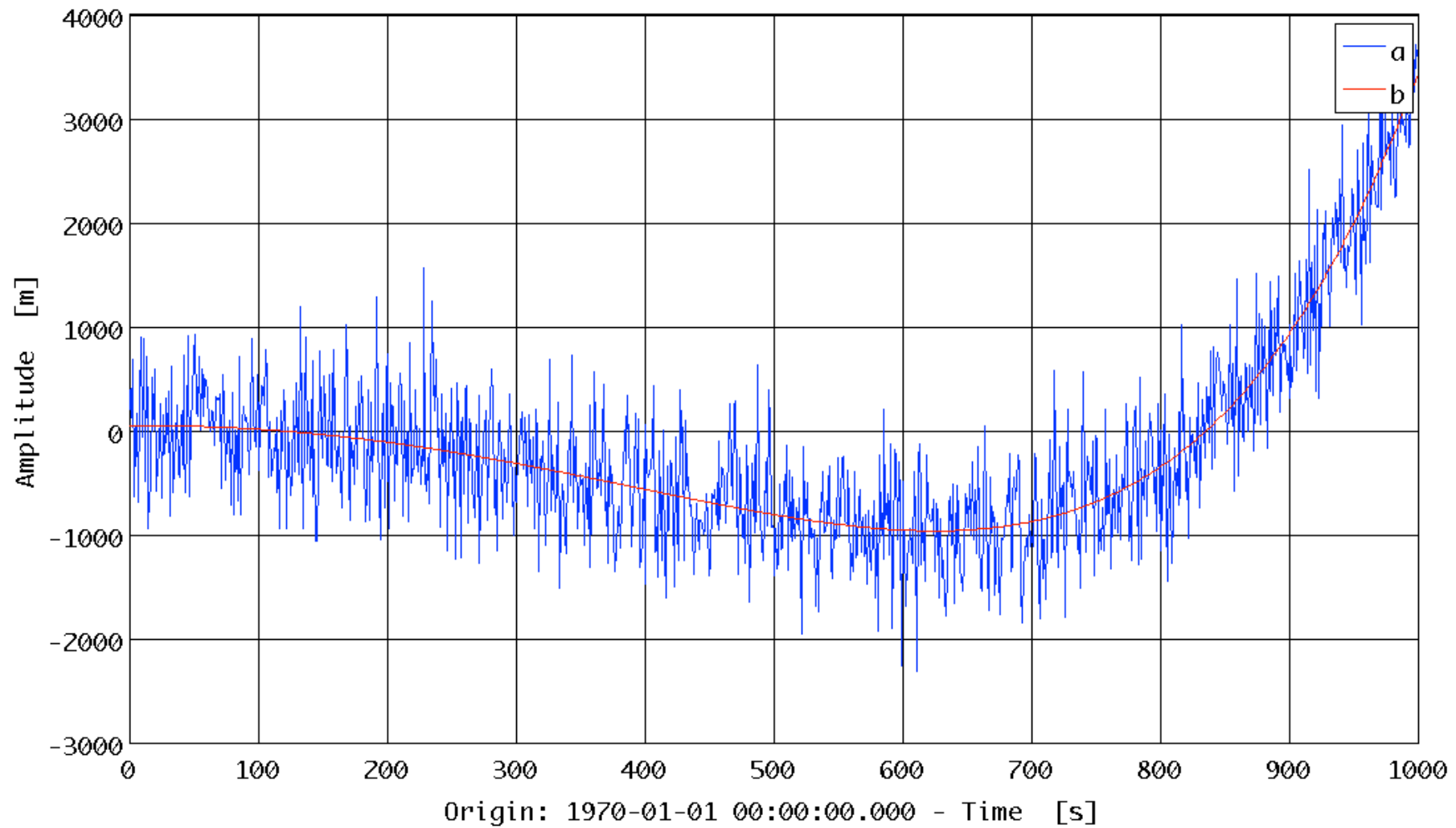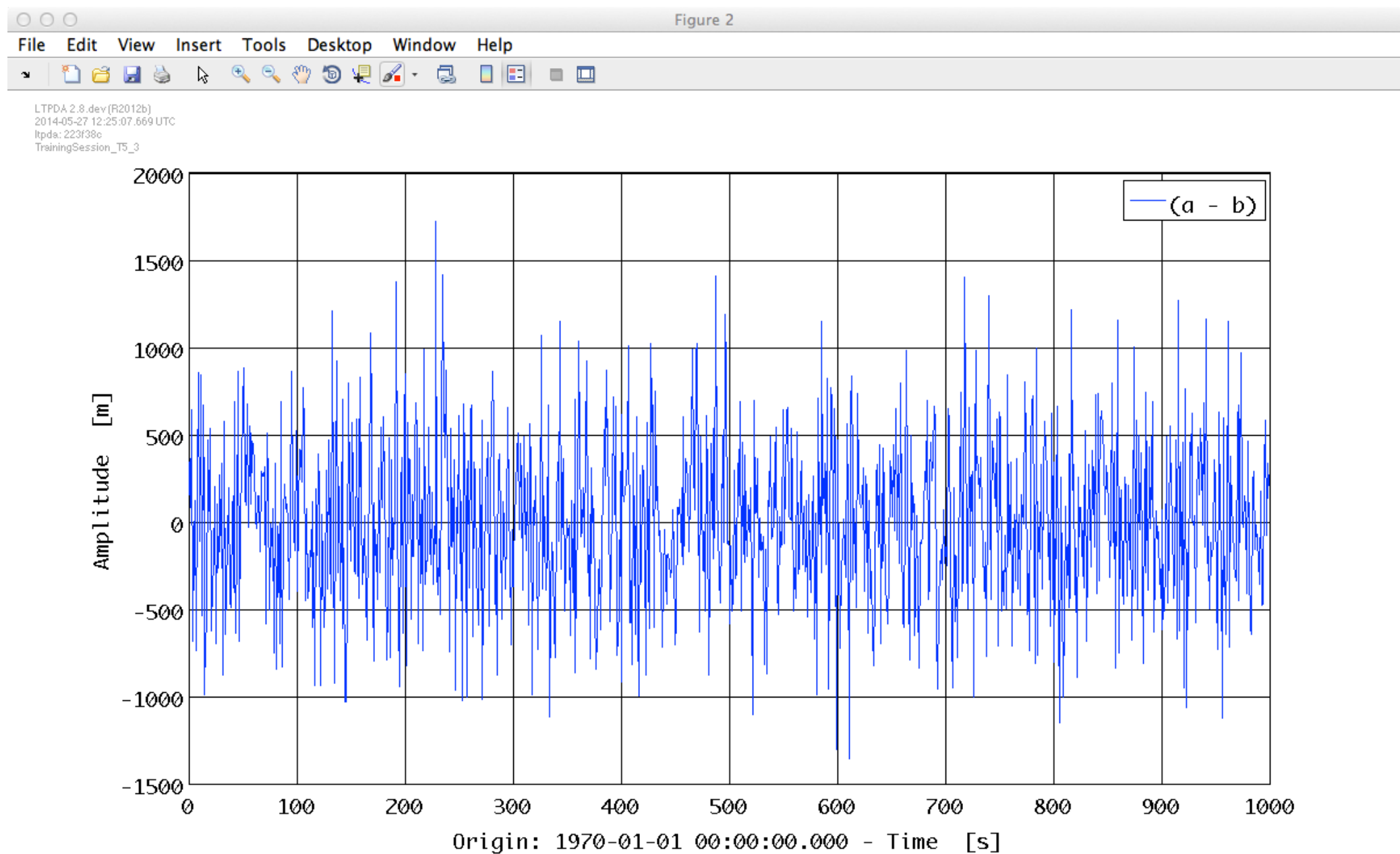
You could also try using `ao/detrend` on the input time-series to yield a very similar result as that shown in the last plot.

©LTP Team

Non-linear least squares fitting of time series

Non-linear least square fitting of time-series exploits the function `ao/xfit`.

Non-linear least square fitting of time series

During this exercise we will:

1. Load time series data
2. Fit data with `ao/xfit`
3. Check results
4. Refine the fit with a Monte Carlo search

Let us open a new editor window and load test data.

```
a = ao(plist('filename', 'topic5/T5_Ex05_TestNoise.xml'));
a.setName('data');
iplot(a)
```

As can be seen this is a chirped sine wave with some noise.
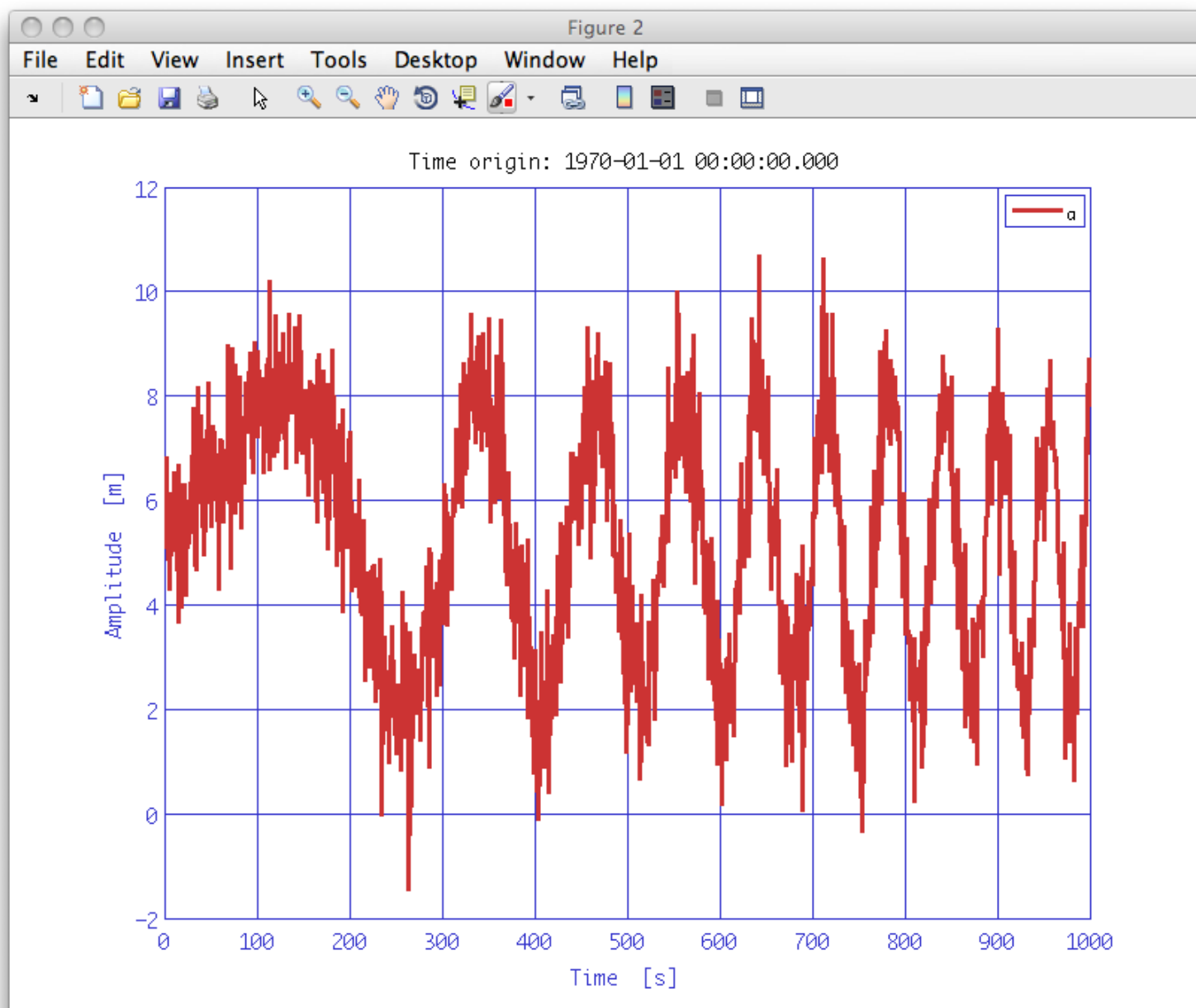
We could now try the fit. The first parameter to pass to `xfit` is a fit model. In this case we assume that we are dealing with a linearly chirped sine wave according to the equation:

$$F(t) = A \sin\left[2\pi\left(f_0 + kt\right)t + \varphi\right]$$

The previous function can be stored within a `smodel` analysis object to pass to the fitting machinery:

```
mdl = smodel(plist('Name', 'chirp', ...
             'expression', 'A.*sin(2*pi*(f + f0.*t).*t + p) + c', ...
             'params', {'A', 'f', 'f0', 'p', 'c'}, ...
             'xvar', 't', ...
             'xunits', 's', 'yunits', 'm'));
```

We need to specify a starting guess for the model parameters. The output of `ao/xfit` is a `pest` analysis objects containing fit parameters.

```
plfit1 = plist('Function', mdl, ...
          'P0', [5, 9e-5, 9e-6, 0, 5]);

params1 = xfit(a, plfit1);
```

Once the fit is done, we can evaluate our model to check fit results.

```
b = eval(params1, plist('xdata', a, 'xfield', 'x'));
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. There are two possibile solutions to such kind of problems: the first one is to refine step by step the fit by looking at the data; the second one is to perform a Monte Carlo search in the parameter space. This way, the fitting machinery extracts the number of points you define in the `'Npoints'` key, evaluates the chi square at those points, reoders by ascending chi square, selects the first guesses and fit starting from them.

```
plfit2 = plist(...
               'Function', mdl, ...
               'MonteCarlo', 'yes', ...
               'Npoints', 1000, ...
               'LB', [1, 5e-5, 5e-6, 0, 2], ...
               'UB', [10, 5e-4, 5e-5, 2*pi, 7]);

    params2 = xfit(a, plfit2);

    c = eval(params2, plist('xdata', a, 'xfield', 'x'));
    c.setName();
    iplot(a, c)
```

The fit now looks like better...
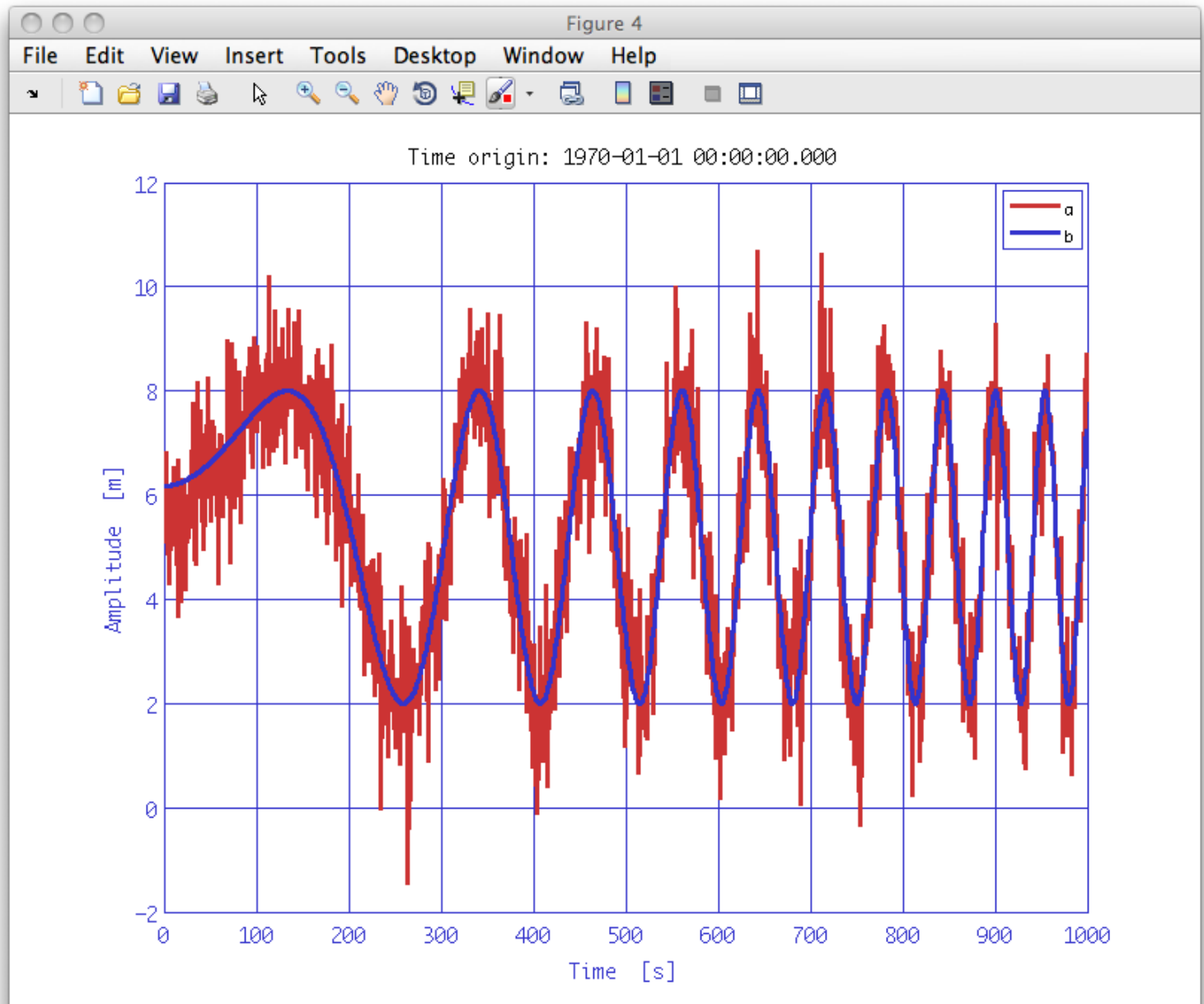
Let us compare fit results with nominal parameters.
Data were generated with the following set of parameters:

```
A   = 3
f   = 1e-4
f0  = 1e-5
p   = 0.3
c   = 5
```

Fitted parameters are instead:

```
A   = 3.02 +/- 0.05
f   = (7 +/- 3)e-5
f0  = (1.003 +/- 0.003)e-5
p   = 0.33 +/- 0.04
c   = 4.97 +/- 0.03
```

The correlation matrix of the parameters, the chi square, the degree of freedom, the covariance matrix are store in the output `pest`. Other useful information are stored in the `procinfo` (processing information) field. This field is a `plist` and is used to additional information that can be returned from algorithms. For example, to extract the chi square, we write:

```
params2.chi2
1.0253740840052
```

And to know the correlation matrix:

```
params2.corr
Columns 1 through 3

                  1          0.120986348157139         -0.0970894969803509
  0.120986348157139                        1           -0.966114904879414
 -0.0970894969803509       -0.966114904879414                         1
 -0.156801230958825        -0.848296014553159          0.717376893765734
 -0.0994358284166703        0.187645552903433         -0.169496082635319

Columns 4 through 5

 -0.156801230958825        -0.0994358284166703
 -0.848296014553159         0.187645552903433
  0.717376893765734        -0.169496082635319
                  1         -0.199286767157984
 -0.199286767157984                         1
```

Not so bad!

⬅ Fitting time series with polynomials	IFO/Temperature Example – signal subtraction ➡

©LTP Team

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:

```matlab
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...

```matlab
iplot(ifo_red, T_red, plist('arrangement', 'subplots'))
```
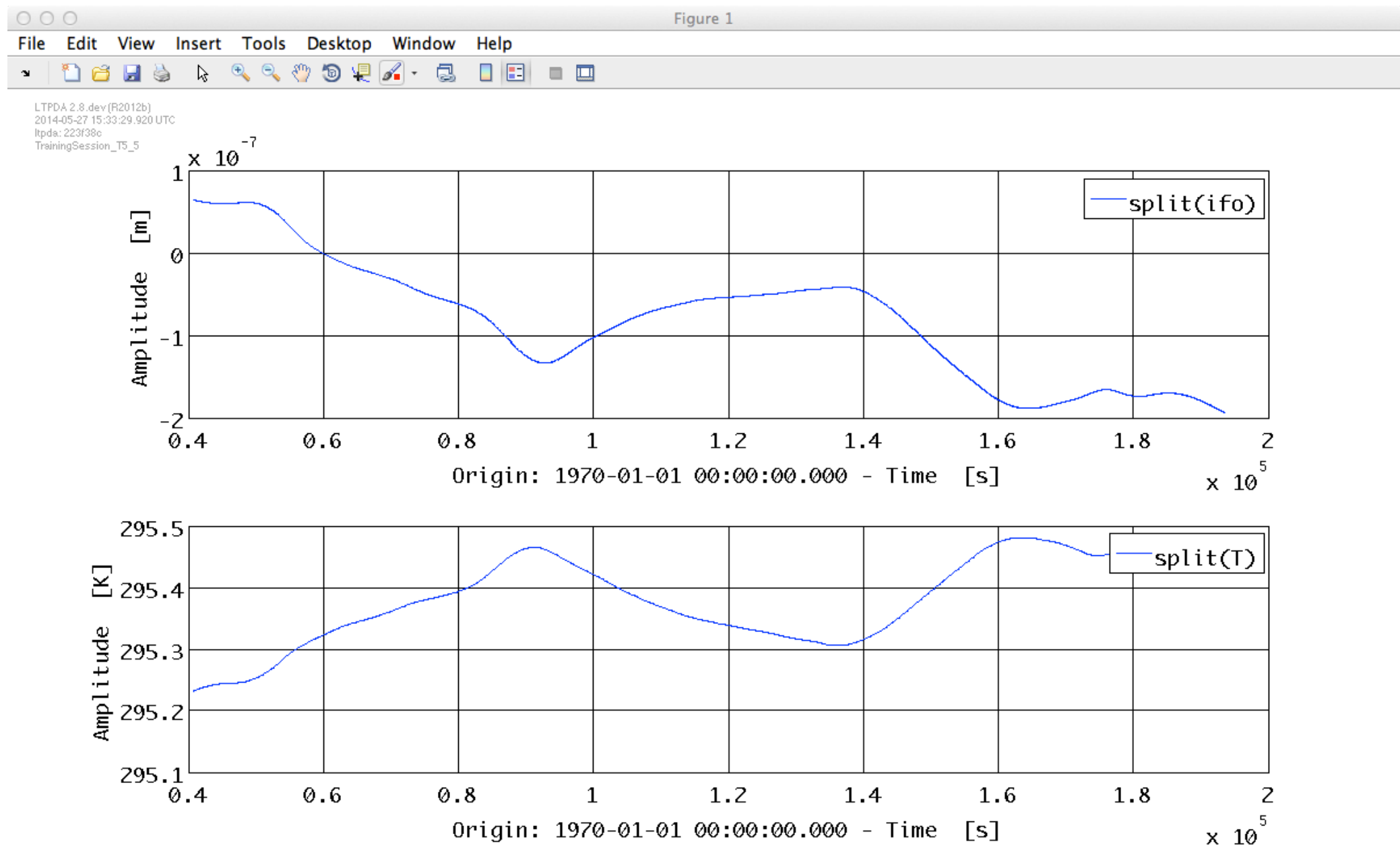
Let us load the transfer function estimate we calculated in Topic 3.

```
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:
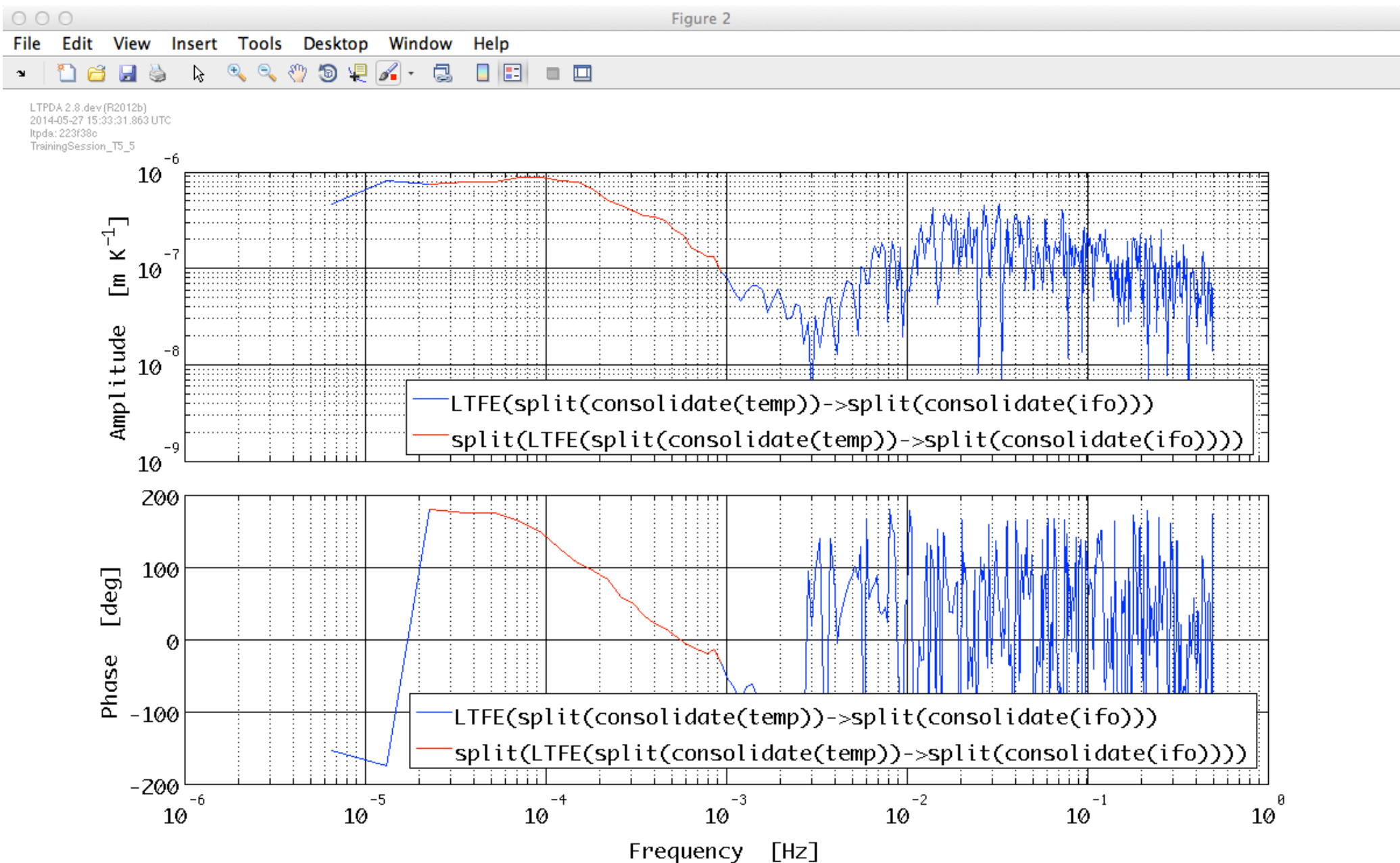
Once we have the proper transfer function, we could start the fitting process. A rapid look to the TF data should convince us that we need a very simple object to fit our data so we could try a fitting session "by hand". In other words, it is more convenient to skip the automathic functionality of `zDomainFit`. Moreover, we force `zDomainFit` to fit a stable model to data because we want to output a stable filter.

```
plfit = plist(...
    'AutoSearch', 'off', ...
    'StartPolesOpt', 'clog', ...
    'maxiter', 5, ...
    'minorder', 3, ...
    'maxorder', 3, ...
    'weightparam', 'abs', ...
    'Plot', 'on', ...
    'ForceStability', 'on', ...
    'CheckProgress', 'off');

fobj = zDomainFit(tfsp, plfit);
fobj.filters.setIunits('K');
fobj.filters.setOunits('m');
```
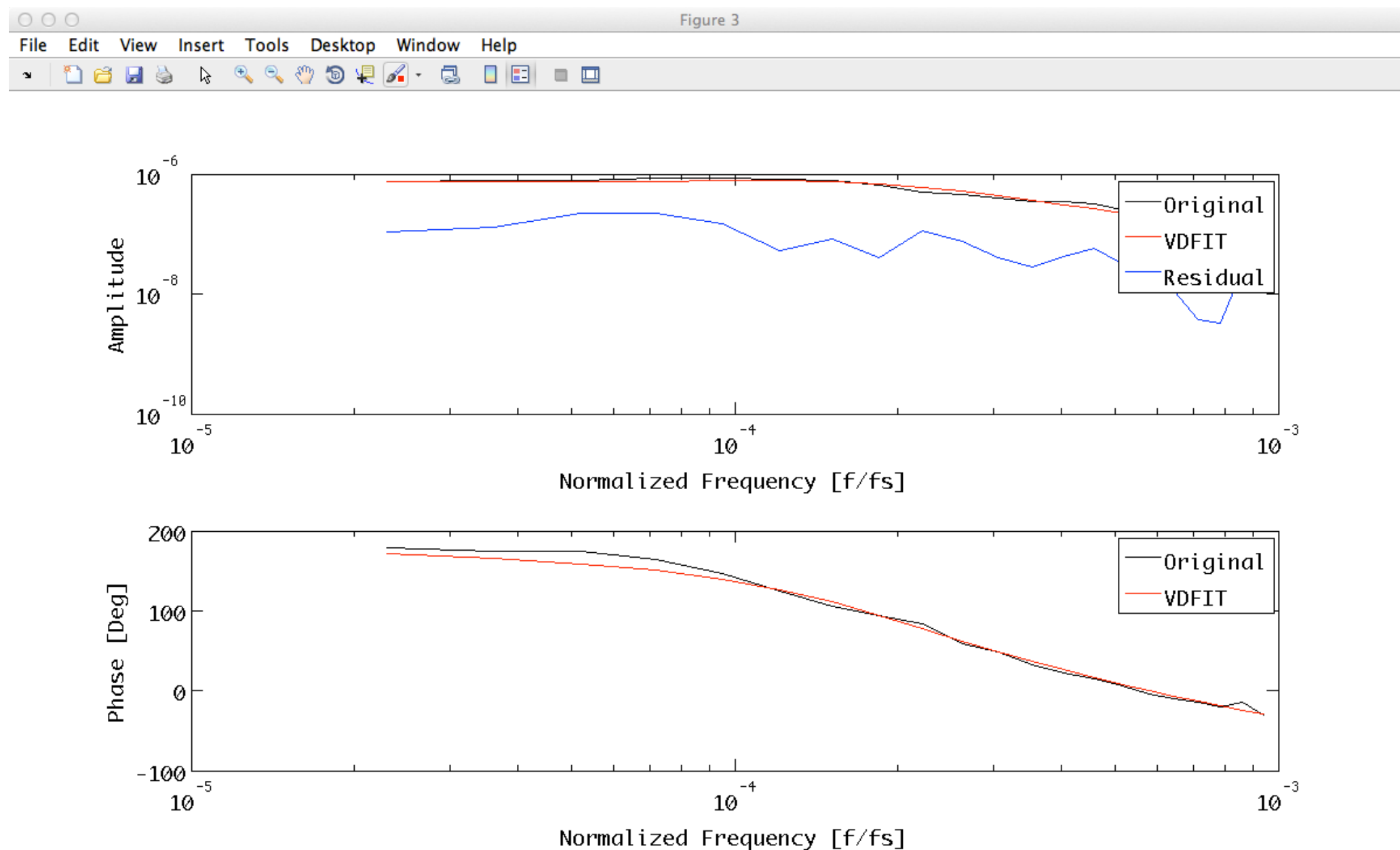
It is time to filter temperature data with the fit output in order to extract temperature contribution to interferometer output. Detrend after the filtering is performed to subtract mean to data (bias subtraction).

```
ifoT = filter(T, fobj, plist('bank','parallel'));
ifoT = split(ifoT, pl_split);
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)
```
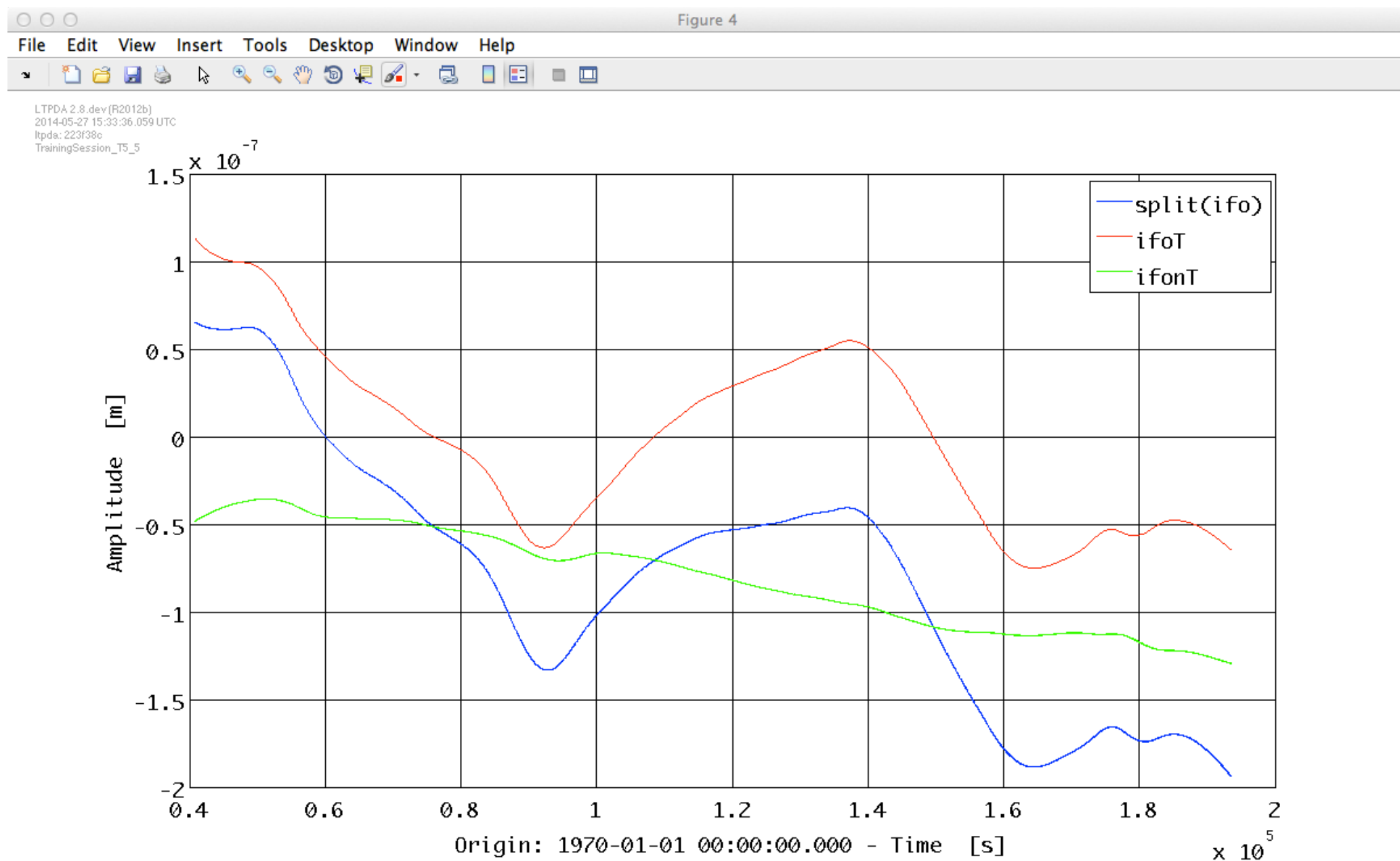
If you now compare spectra of the original IFO signal and the one with the temperature contribution removed, you should see something like the figure below:

```
ifoxx = ifo_red.lpsd;
ifonTxx = ifonT.lpsd;
iplot(ifoxx, ifonTxx)
```

©LTP Team
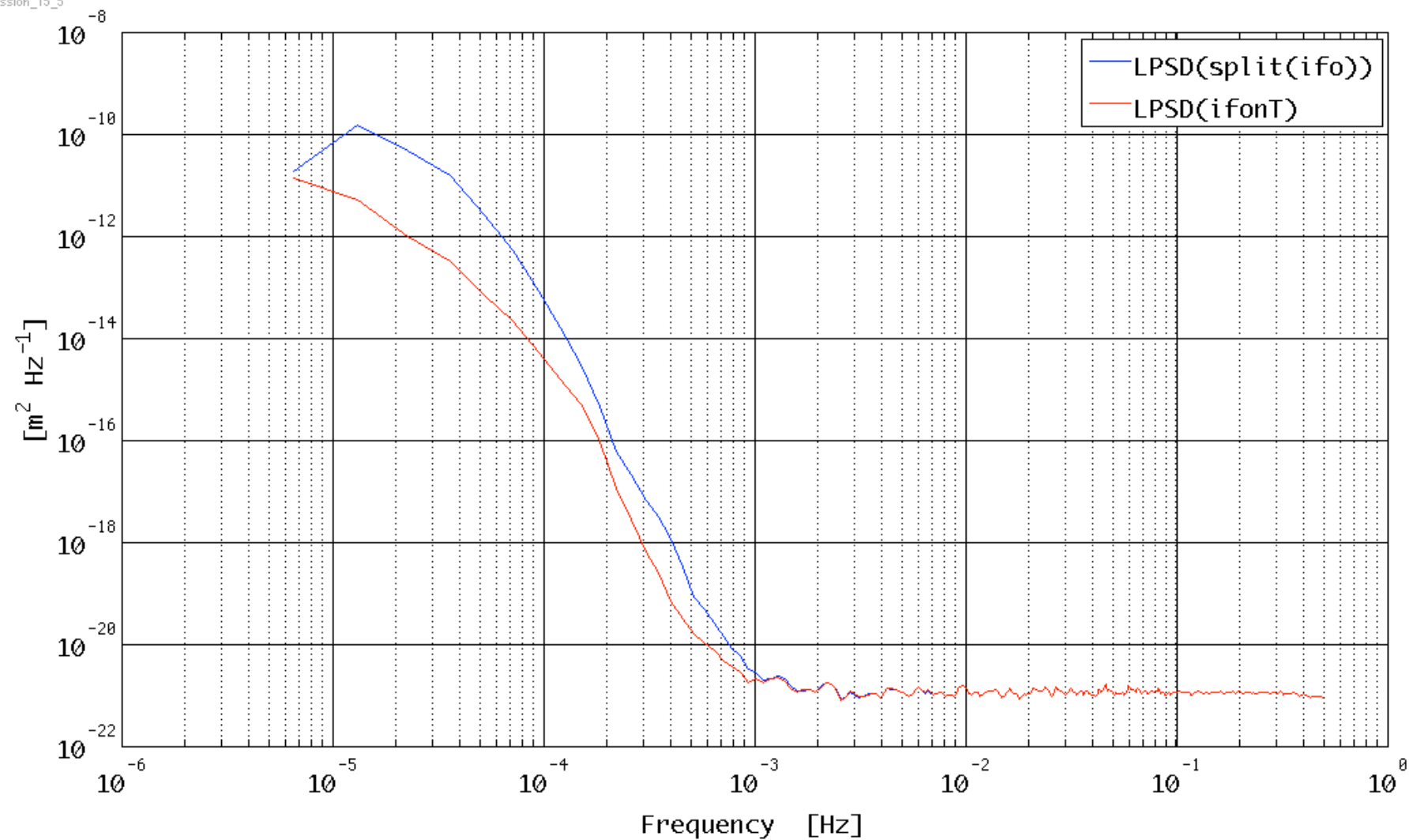
LTPDA Training Session 2

This series of help pages consitute the second training session of LTPDA. The various data-packs used throughout the tutorials are available for download on the [LTPDA web-site](#). This tutorial focusses primarily on data analysis activities associated with the LISA Pathfinder mission. It requires access to the LPF extension module for LTPDA (LPF_DA_Module). As such this tutorial is not intended for the general public.

1. [Topic 1 – LTPDA Review](#)
2. [Topic 2 – Simulating LPF noise](#)
3. [Topic 3 – Estimating residual acceleration](#)
4. [Topic 4 – Simulating LPF with injected signals](#)
5. [Topic 5 – Introduction to LPF System Identification](#)

Throughout the course of this training session, we will perform a full analysis of some LPF data. The data will be generated using our LPF simulator which is built in to the LTPDA toolbox and associated extension module.

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 estimate some parameters of the LPF system and use those to estimate the residual differential acceleration of the two test-masses. In particular the steps will be:

1. [Topic 1](#) LTPDA Review.
   1. Introducing objects in LTPDA
   2. Saving and loading objects
   3. Review of spectral estimators
   4. Preparing data segments (splitting)
   5. Review of filtering and whitening in LTPDA
   6. LTPDA scripting best practices
   7. Introduction to LTPDA extension modules
2. [Topic 2](#) Simulating LPF noise in LTPDA.
   1. Introduction to state-space models in LTPDA
   2. Introduction to the LPF state-space models in LTPDA
   3. Building an LTP model
   4. Introduction to the various LPF noise models
   5. Building an LPF model
   6. Simulating LPF noise
   7. Changing system parameters
   8. Simulate LPF with matched stiffness
3. [Topic 3](#) Estimation of residual differential acceleration.
   1. Principles and theory
   2. Tools for estimating the residual differential acceleration in LTPDA
   3. Estimate equivalent accelerations from simulation data
4. [Topic 4](#) Simulating LPF with injected signals.
   1. LPF model inputs
   2. Building signals
   3. How to inject signals
   4. Simulate LTP with injected signals (no noise)
   5. Inject noise signals to LTP

6. Estimate tranfser functions from simulated signals, compare with Bode estimates
7. Simulate LPF with injected signals

5. Topic 5 Introduction to system identification of LPF.
   1. Introduction to LTPDA's fitting tools (theory, implementation, usage)
   2. A simplified LPF system identification experiment
   3. Create simulated experiment data sets
   4. Build state-space models for system identification
   5. Calculate expected covariance of the parameters (FIM)
   6. Perform system identification to estimate desired parameters
   7. Results and Comparison
   8. Use parameter estimates to estimate residual differential acceleration

◀ IFO/Temperature Example – signal subtraction   Topic 1 – LTPDA Review. ▶

©LTP Team

Topic 1 – LTPDA Review.

Topic 1 of the training session aims to review the basic use of LTPDA. After working through the examples you should be familiar with:

1. Introducing objects in LTPDA
2. Saving and loading objects
3. Review of spectral estimators
4. Preparing data segments (splitting)
5. Review of filtering and whitening in LTPDA
6. LTPDA scripting best practices
7. Introduction to LTPDA extension modules

Introducing objects in LTPDA

LTPDA uses an object-oriented approach for data analysis. That means that the user deals with objects. So what are objects? Objects are instances of a particular class. And what's a class? Well, a class just describes an object. For example, you may have a class 'Car' which has certain properties that describe the features of a car. Once you build a Car according to the class description, then you have an instance of a Car - you have an object. We mentioned properties here. Part of the description of a real-world object (or idea) are the features of that object. These features are captured as properties of the class. For example, our 'Car' class may have a property 'color'. Once we build (instantiate) a 'Car' object, then the property 'color' will have a particular value.

Once you have an object, what can you do with it? Well, classes don't only have properties, they also have methods. These are actions you can perform on or with the object. For example, if we return to our Car analogy, the class 'Car' may have a method called 'start' which, when applied to an instance of the 'Car' class (an object), will start the car.

To bring this more in to the field of data analysis, let's look at another example. Suppose we have a class which defines a digital filter. Let's call this class 'miir' - a MATLAB Infinite Impulse Response filter. Such a class would need to have properties which capture features of a digital filter, for example, numerator and denominator coefficients (or 'a' and 'b' coefficients). We might have methods in the class such as 'resp' to calculate the response of the filter.

One special set of methods that a class typically has are the constructor methods. These don't act on objects, but rather act on the class itself. And, as the name suggests, they are the methods which actually build the objects. They always have the class name and typically, they take a set of input parameters which describe how to build the object we want. Returning to our digital filter example, in LTPDA we have such a class 'miir'. To build an miir object (or make an instance of the class 'miir') you can do:

```
    filt = miir()
```

This will create an 'empty' miir object. On the terminal you should see:

```
    ------ miir/1 -------
          b: []
     histin: []
      ntaps: 0
         fs: []
     infile:
          a: []
    histout: []
     iunits: []
     ounits: []
       hist: miir.hist
    procinfo: []
    plotinfo: []
       name:
description:
       UUID: 70b4b3cd-2e65-4332-94cc-cffea1b0028c
    --------------------
```

You can see that the class 'miir' has a number of properties, including 'name', 'description', 'a', 'b', 'iunits', and 'ounits'. To build a more interesting filter you can do something like:

```
    filt = miir(plist('type', 'lowpass', ...
        'order', 1, ...
        'fs', 10, ...
        'fc', 1))
```

If you run that command, you will see the following on the terminal:

```
       ------ miir/1 -------
              b: [1 -0.50952544949442879]
         histin: 0
          ntaps: 2
             fs: 10
         infile:
              a: [0.2452372752527856 0.2452372752527856]
        histout: 0
         iunits: []
         ounits: []
           hist: miir.hist
        procinfo: []
        plotinfo: []
           name: lowpass
    description:
           UUID: 6729d0ab-b29c-4c4c-9f90-0aca13b496fc
       --------------------
```

Now you're asking yourself "How do I know which parameters I pass to the constructors?" To find out, read the following section.

Getting Help

There are various places where you can get help with using LTPDA. The most obvious is the User Manual. To access the LTPDA user manual, click on the menu "Help->Product Help" to launch MATLAB's documentation browser (or type 'doc' on the terminal). From there you can browse to the LTPDA Toolbox section.
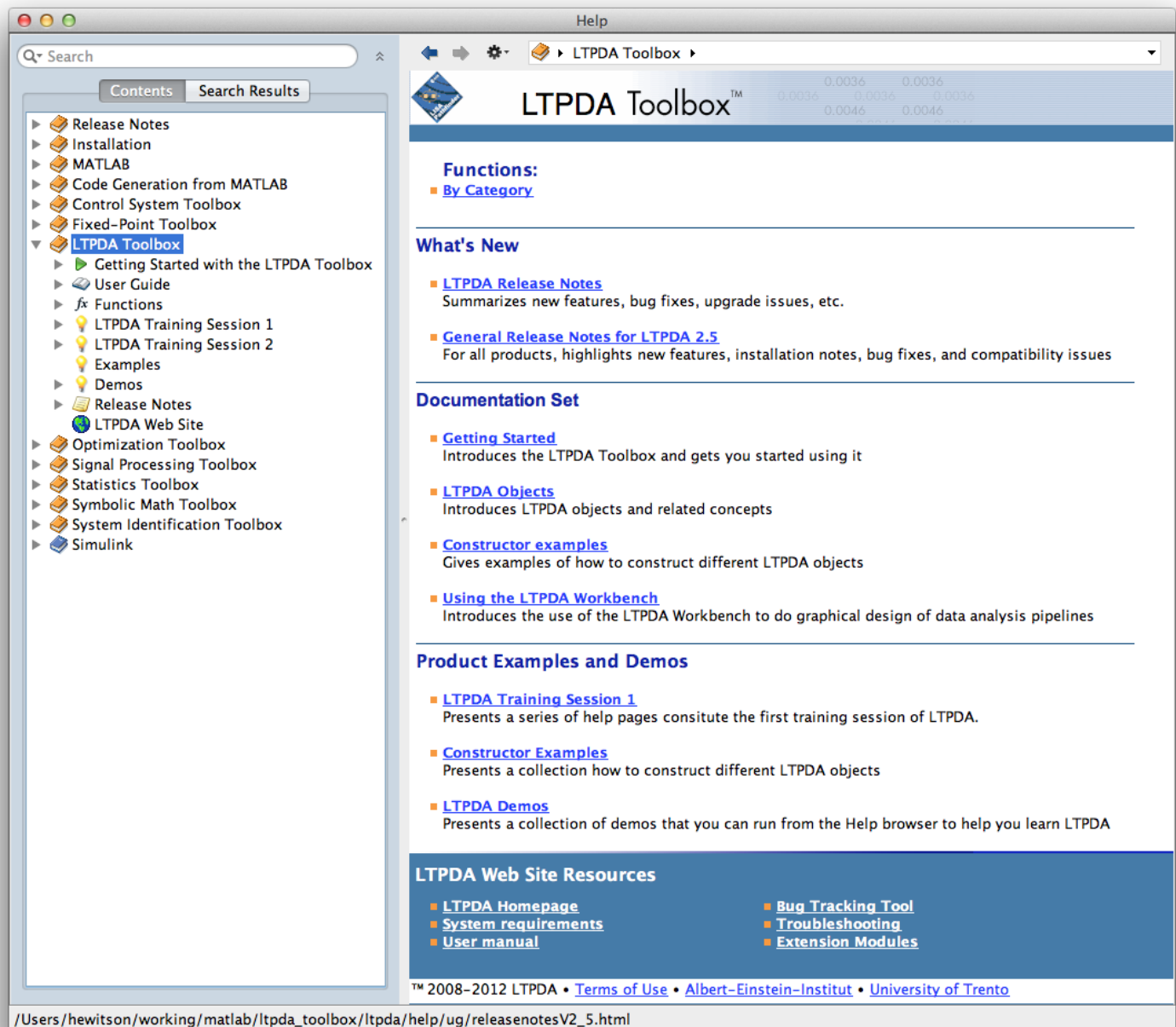
Help for methods

To get help on individual methods, you use the built-in documentation. For our `miir` class, you do:

```
help miir
```

which will display some description of the class constructor. The important part is the "Parameters Description" link.

```
Parameters Description
```

Clicking on the link will open a window which describes the possible sets of parameters that you can pass to the constructor. In fact, these "Parameters Description" links are available for all methods in LTPDA, not just class constructors. The figure below shows an example for our miir case.
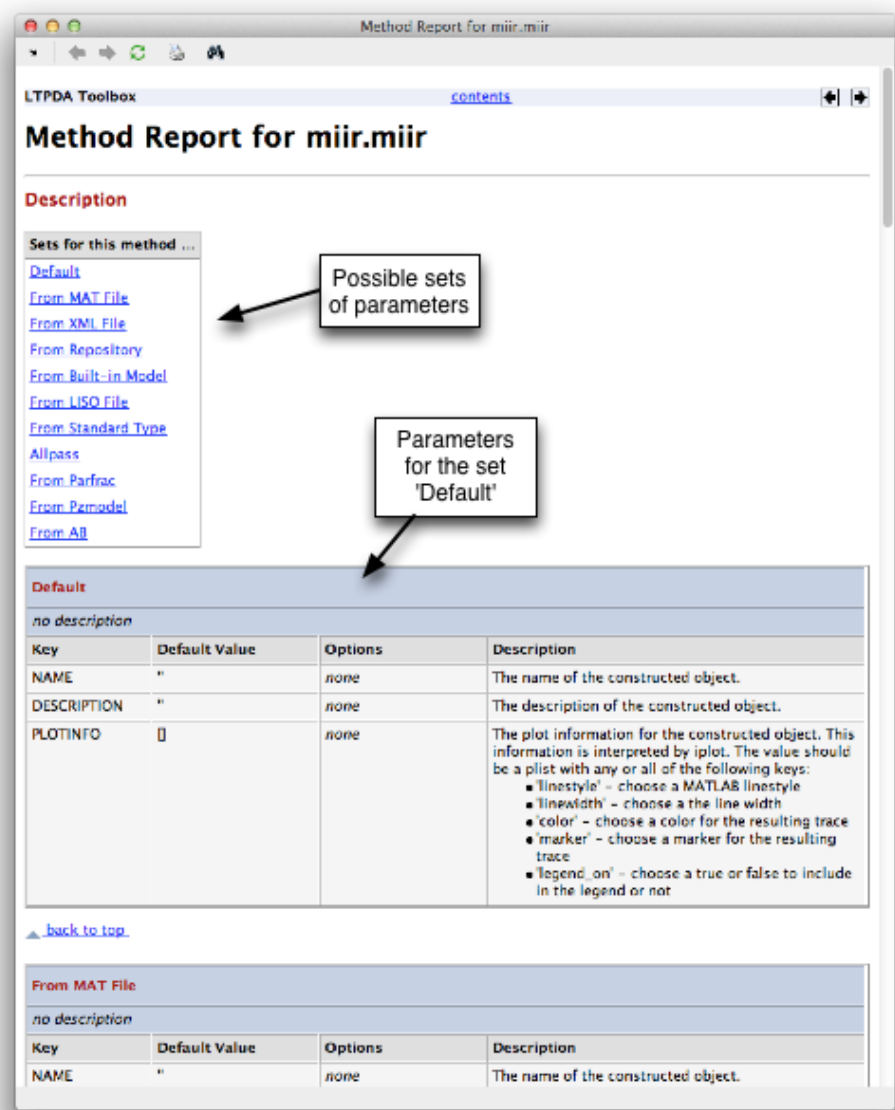
The following table summarises the terms used in this section.

Term	Description
Class	A description of a real-word object or idea
Property	Classes have properties which encapsulate features of the real-world object.
Object	An example of a realisation (or instantiation) of a class.
Method	A function that acts on instances of a class (objects).

Saving and loading objects

Saving and loading objects in LTPDA is straightforward. You have two formats available to you: the standard MATLAB .mat binary format, or the text–based XML format.

Saving single objects

Suppose you have an AO in your MATLAB workspace, you can save it to disk like this:

```
% Generate a time-series of random numbers
a = ao.randn(10, 10);
% Save to a file called 'a.mat'
save(a);
```

This will save the file to disk in MAT format with the filename **'a.mat'**. If you want to specify a filename, you can do that by doing:

```
% Generate a time-series of random numbers
a = ao.randn(10, 10);
% Save the AO to a MAT file
save(a, 'myNiceAO.mat');
```

You can also use a plist to configure save:

```
f = miir(plist('type', 'lowpass', 'fc', 0.1, 'fs', 10, 'order', 1))
save(f, plist('filename', 'myNiceFilter.mat'));
```

Saving multiple objects

You can save multiple objects in to one file with LTPDA:

```
a1 = ao.randn(10, 10);
a2 = ao.randn(10, 100);
save(a1, a2, 'myObjects.mat');
% If you have a vector of objects...
a = [a1 a2];
save(a, 'myVectorOfObjects.mat');
```

To save a vector of objects to individual files, you can do:

```
a1 = ao(plist('waveform', 'sine wave', 'a', 1, 'f', 1, 'phi', pi/2, 'fs', 50, 'nsecs',
10, 'name', 'sig1'))
a2 = ao(plist('waveform', 'sine wave', 'a', 1, 'f', 0.2, 'phi', pi/4, 'fs', 50,
'nsecs', 10, 'name', 'sig2'))
a  = [a1 a2];
save(a, plist('individual files', true));
```

If you want to specify a filename, then as before, you do:

```
a1 = ao(plist('waveform', 'sine wave', 'a', 1, 'f', 1, 'phi', pi/2, 'fs', 50, 'nsecs',
10, 'name', 'sig1'))
a2 = ao(plist('waveform', 'sine wave', 'a', 1, 'f', 0.2, 'phi', pi/4, 'fs', 50,
'nsecs', 10, 'name', 'sig2'))
a  = [a1 a2];
save(a, plist('filename', 'myAOs.xml', 'individual files', true));
```

Loading objects from disk

Loading objects from disk is typically done using the relevant constructor. However, this does

imply that you know the class of the object in the file on disk. If you don't, you need to try to load with a particular constructor (say, AO) then if that is not correct, the error message will tell you the class of the object in the file.

Let's load one of the AOs we saved above:

```
a = ao('myNiceAO.mat');
```

Above we saved a file called 'myNiceFilter.mat'. If we don't know that this is an `miir` object, and instead try to load it as an AO, then you will see the following:

```
>> a = ao('myNiceFilter.mat');
    Error using ltpda_uo/fromFile (line 174)
You tried to load objects of class [ao] from myNiceFilter.mat, but that file contains
objects of class [miir]
Error in ao (line 348)
                    obj = fromFile(obj, args{1});
```

◀ Introducing objects in LTPDA   Review of spectral estimators ▶

©LTP Team

Review of spectral estimators

We have two classes of spectral estimators in LTPDA: one class based on the standard WOSA (Welch's Overlapped Segmented Average) and a modified version of WOSA which estimates the spectral quantities at frequencies spaced logarithmically.

Estimating Power Spectral Densities

To estimate the Power Spectral Density of a time–series, you can use the method `ao/psd` or the logarithmically spaced frequencies version, `ao/lpsd`. For example:

```
% Generate a time-series of random numbers
a   = ao.randn(100, 10);

% Compute a PSD with the default configuration.
axx = psd(a);

% Plot the result
iplot(axx)
```

The `psd` method has various configuration parameters to adjust its behaviour. These are described in the method's documentation. If you do:

```
help ao/psd
```

and then click on the 'Parameters Description' link, you should see a documentation window with a full table describing the parameters for `psd`. To get a brief list of the `plist` keys available, you can simply do:

```
>> keys('ao', 'psd')
-------------------------------------------------
Default
    -------------------------------------------------
    NFFT, WIN, PSLL, OLAP, ORDER, NAVS, TIMES, SCALE
```

So, to compute the Amplitude Spectral Density of a time–series with a certain number of averages, you do:

```
% Generate a time-series of random numbers
a   = ao.randn(100, 10);

% Create a plist to configure ao/psd
psdPlist = plist('navs', 10, 'scale', 'asd');

% Compute the PSD of the time-series with the given configuration plist
axx = psd(a, psdPlist);

% Plot the result
iplot(axx)
```

The logarithmic–spaced version of PSD (`ao/lpsd`) shares some of the configuration keys with `ao/psd`. As such, you can, more often than not, simply interchange them:

```
% Generate a time-series of random numbers
a   = ao.randn(100, 10);

% Create a plist to configure ao/psd
psdPlist = plist('navs', 10, 'scale', 'asd');
```

```
        % Compute the logarithmically spaced PSD of the time-series with the given
configuration plist
        axx = lpsd(a, psdPlist);

        % Plot the result
        iplot(axx)
```

(You will notice that `lpsd` doesn't respond to the key 'NAVS' since it computes the number of averages per frequency bin itself. You should see that LTPDA warns you that the key 'NAVS' is ignored.)

Estimating transfer functions, cross-spectral densities and coherence

Transfer functions can be estimated from input and output data using the `ao/tfe` method (or the corresponding log-spaced method, `ao/ltfe`). Here's an example:

```
        % Sample frequency of our data
        fs = 10;

        % Generate a time-series of random numbers
        input   = ao.randn(100, fs);

        % Create a digital filter
        myFilter = miir(plist('type', 'bandpass', 'fc', [0.5 1], 'fs', fs, 'order', 3));

        % Filter the noise data
        output = filter(input, myFilter);

        % Estimate the transfer function from input to output
        T = tfe(input, output);

        % Plot the result
        iplot(T)
```

Cross-spectral densities can be estimated using `ao/cpsd` (or `ao/lcpsd`) and coherence can be estimated using `ao/cohere` (or `ao/lcohere`).

◀ Saving and loading objects   Preparing data segments (splitting) ▶

©LTP Team

Preparing data segments (splitting)

One of the most common tasks in data analysis is the preparing of the data segments to analyse. LTPDA offers various tools for preprocessing data. Here we will review the splitting of data segments using the `ao/split` method.

Suppose you have a segment of data and you want to split that up. Here are various examples showing how you might do that.

Split in to parts

To split the data in to `N` parts you can do:

```matlab
% Generate a time-series
waveformPlist = plist(...
             'waveform', 'square wave', ...    % Generate a square wave
             'f', 0.1, ...                     % with 0.1Hz frequency
             'fs', 10, ...                     % sampled at 10Hz
             'nsecs', 100, ...                 % lasting for 100s
             't0', '2012-03-10 12:00:00', ... % with the specified reference time
             'toffset', 10 ...                 % and the first sample starts 10s after
the reference time
             );
a    = ao(waveformPlist);

% Split into 5 parts
splitPlist1 = plist('N', 5);

% Split the time-series
parts = split(a, splitPlist1);

% Plot the result
iplot(parts)
```

Split by time relative to t0

To split the data by elapsed time since the reference time t0 you can do

```matlab
% Split a segment starting a t=20 and finishing at t=75
splitPlist2 = plist('times', [20 75]);

% Split the time-series
timeSegment = split(a, splitPlist2);

% Plot the result
iplot(timeSegment)
```

Split by offsets

To split the data by an offset in seconds relative to the first and last samples, you do

```matlab
% Split a segment starting 10s from the start to 40s from the start
splitPlist3 = plist('offsets', [10 40]);

% Split the time-series
offsetSegment = split(a, splitPlist3);

% Plot the result
iplot(offsetSegment)
```

You can also split relative to the end of the data by doing the following:

```
    % Make a segment which drops 10s from the beginning and end of the original
    splitPlist4 = plist('offsets', [10 -10]);

    % Split the time-series
    dropSegment = split(a, splitPlist4);

    % Plot the result
    iplot(dropSegment)
```

Split by absolute time

To split the data by absolute times, you do

```
    % Split a segment starting and ending at particular times
    splitPlist4 = plist('start_time', '2012-03-10 12:00:15', 'end_time', '2012-03-10
12:00:36');

    % Split the time-series
    absTimeSegment = split(a, splitPlist4);

    % Plot the result
    iplot(absTimeSegment)
```

◀ Review of spectral estimators   Review of filtering and whitening in LTPDA ▶

Review of filtering and whitening in LTPDA

Digital Filtering in LTPDA

LTPDA supports two types of digital filters: Infinite Impulse Response filters (IIR) and Finite Impulse Response filters (FIR). These two filter types are represented by the classes `miir` and `mfir`.

To create a simple lowpass filter, you can do:

```
% Configure a 2nd order lowpass at 1Hz for 100Hz data
filterPlist = plist(...
        'type', 'lowpass', ...
        'fc', 1, ...
        'order', 2, ...
        'fs', 100 ...
        );

% Configure a 2nd order lowpass at 1Hz for 100Hz data
myFilter = miir(filterPlist);
```

Once you have your filter object, you can check its response by doing the following:

```
% Plist for log-spaced frequencies from 0.1Hz to 10Hz
respPlist = plist('f', logspace(-1, 1, 1000));

% Compute filter response
filterResponse = resp(myFilter, respPlist);

% Plot the response
iplot(filterResponse)
```

You can now use the filter to filter some time-series data by doing the following:

```
% Generate a time-series of random numbers
input = ao.randn(100, 100);

% Filter the data
output = filter(input, myFilter);

% Plot the input and output
iplot(input, output)
```

Whitening Data in LTPDA

Often it is useful to be able to whiten a data segment. There are a number of ways you can go about doing this, but in LTPDA a simple way is to use the methods `ao/buildWhitener1D` and `ao/whiten1D`. The first method actually builds the whitening filter by fitting to a spectral estimate of the data. The second method uses `buildWhitener1D` to generate the whitening filter and then applies it to the data.

Here's an example of using `whiten1D` to whiten the data we filtered above.

```
% Whiten the data
outputWhite = whiten1D(output);

% Plot the filtered and whitened data
iplot(outputWhite, output)

% Create a plist to configure ao/psd
psdPlist = plist('navs', 10, 'scale', 'asd');

% Compute the PSD of the filtered and whitened data
[output_xx, outputWhite_xx] = psd(output, outputWhite, psdPlist);

% Plot the spectra filtered and whitened data
iplot(outputWhite_xx, output_xx)
```

There are many parameters for configuring `whiten1D` which can be tuned to improve the results. Consult the method documentation to see the parameters.

◀ Preparing data segments (splitting)   LTPDA scripting best practices ▶

©LTP Team

LTPDA scripting best practices

Although LTPDA is built on top of MATLAB, there are a number of differences in the way it should be used. What follows here is a set of best-practices that should help produce readable and reusable LTPDA scripts which properly capture history.

1. [Laying out scripts](#)
2. [Workflow and script pipelines](#)
3. [Variable naming](#)
4. [Documenting scripts](#)
5. [Copying and modifying objects](#)
6. [Method usage and parameter lists](#)

Laying out scripts

The MATLAB editor has a number of powerful features which can be leveraged to ensure the maximum readability of scripts. The following features are recommended when scripting for LTPDA:

1. The default right-hand text limit of 75 characters should be used and it is recommended to keep commands within this length where possible. Use the elipses (...) syntax to break long text lines at logical places. For example, avoid this:

   ```
   pl = plist('param1', 1, 'param2', 'my name', 'param3', a, 'param4', 'some value', 'param5', 2, 'param6', 'x');
   ```

   Instead do this:

   ```
   pl = plist(...
           'param1', 1, ...            % My first parameter
           'param2', 'my name', ...    % My second parameter
           'param3', a, ...            % My third parameter
           'param4', 'some value', ... % My fourth parameter
           'param5', 2, ...            % My fifth parameter
           'param6', 'x' ...           % My sixth parameter
           );
   ```

   This not only is more readable, but it allows you to comment each parameter.

2. Make use of MATLAB's cell structure in the editor. This not only splits your script into logical blocks of code, but it also makes for power automatic documentation using MATLAB's built-in publishing feature. You can even have the editor configured so that the background of the currently active cell is highlighted. Here's an example:

   ```
   %% Create some objects

   % build an AO with value 1
   test_ao_1 = ao(1);

   % build an AO with value 2
   test_ao_2 = ao(2);

   %% Add the objects together

   aoSum = test_ao_1 + test_ao_2;
   ```

3. If the length of a script exceeds a couple of hundred lines you should consider refactoring

some of the code into sub-functions or methods. Creating your LTPDA methods and properly namespaced functions is possible but is an advanced topic. For details look at the documentation for creating extension modules (section "LTPDA Extension Modules") in the main part of the user guide. Also look at the following help:

```
help utils.modules.buildModule
help utils.modules.makeMethod
```

Workflow and script pipelines

An investigation can typically be broken down into logical steps. For example, it's likely you can break-down any investigation into the following steps:

1. Set up configuration parameters and general plists
2. Load (or download) starting data
3. Perform some analysis
4. Save (or upload) the results

By keeping the analysis short, you can break-down a full investigation into a series of scripts, each performing a clear part of the overall analysis. By starting and ending a script from a well-defined state (either on disk, or in a repository) you can chain the scripts together. It may also be desirable to have an overall driver script which simply runs all the sub-scripts in the desired order.

What follows here is a set of best-practices to help develop a modular and clear scripting work-flow:

1. Try to break-down an investigation into a sequence of short scripts.
2. Provide a driver script to which clearly explains the flow of the investigation and calls the sub-scripts (or functions) in the desired order.
3. Always include a 'clear all' statement at the beginning of a script. Note: if you use functions, this is not necessary since functions automatically get their own local workspace when executed.
4. A script should represent a logical set of stand-alone actions. It is recommended that a script starts from and ends at a defined state. Typically scenarious would be:
   - Start from objects on disk; end by saving objects to disk.
   - Start from objects in a repository; end by submitting objects to a repository.
   Sometimes a mixture of the two is desired. The use of a repository ensures that your script can be run by anyone else who has access to the same repository.
5. If your work-flow involves saving the state of the investigation to files on disk, try to use relative file paths, rather than absolute file paths. This increases the chance that someone else can run the script(s) without resetting all the file paths.

If we take the focus of this training session as an example, the system identification investigation can be broken down into the following steps:

1. Create simulated data set
2. Build statespace model for fitting
3. Calculate expect covariance of parameter set
4. Perform parameter estimation
5. Post-process results

It would then make sense to have one script (or a function) per step. You would then create a driver script to run the full analysis, something like the one below. The individual scripts (or functions) and the driver script would reside in a single (sensibly named) directory on disk. This

directory then represents a defined pipeline which can be reused by others.

```
% A script which simulates a full system-identification investigation of
% LISA Pathfinder using statespace models to generate the data. The parameter
% estimation is performed using different techniques and the results compared.
%
%
% The script requires the use of the 'LPF_DA_Module' extension module.
%
% M Hewitson 2024-02-30
%
% VERSION: 1.0
%

% Create the simulated data set
createSimulatedDataSet;

% Build the ssm model to be used for fitting
buildFittingModel;

% Calculate the expected covariance of the parameters given the model
calculateExpectedParameterCovariance;

% Perform parameter estimation using MCMC method
performMCMCParameterEstimation;

% Perform parameter estimation using linear fit
performLinearParameterEstimation;

% Compare results
compareResults;
```

Variable naming

As well as the formal MATLAB rules about variable names (documented here: [Variables](#)), here are some further recommendations about variable names:

1. Variables should be given meaningful names. Do not truncate variable names to meaningless symbols simply at the expense of typing characters. Avoid single letter variable names, unless the meaning is clear. Avoid variable names like 'a1', 'a2', ..., 'foo', 'dummy', 'tmp'. Also, avoid variable names which typically are used for functions/methods, for example, 'sum', 'psd', 'mean'.

2. Use underscores to seperate parts of a name to improve readability. Sometimes camel-case names are preferred (myNiceVariable), and even a mix of the two conventions can be used. Both systems are very readable.

3. Generally, variables should begin with a lower-case letter, except where using an upper-case letter results in increased clarity. For example, a lower-case 'f' is typically used to represent frequency, whereas an upper-case 'F' might be used to represent a force.

4. When computing spectral estimates, use prefixes together with the original variable name to retain the connection to the time-series data. For example, when estimating the PSD of a time-series pointed to by the variable 'x1', use the variable name 'S_x1' to point to the PSD. When measuring the transfer function between two time-series data pointed to by variables 'x1' and 'y1', use a variable name something like 'T_x1_y1' for the transfer function.

Here are some examples of bad variable names:

```
% A plist for using when calculating a PSD
p = plist('navs', 10);
```

```
% An ao created from a file on disk
a = ao('command_force_x2.mat');

% PSD of some data
p = psd(a);
```

Here are examples of better names:

```
% A plist for using when calculating a PSD
psdPlist = plist('navs', 10);

% An ao created from a file on disk
F_cmd_x2 = ao('command_force_x2.mat');

% PSD of some data
S_x2 = psd(x2);
```

Characters are free; understanding is expensive!

Documenting scripts

A script can not have too much documentation. MATLAB's documentation system is extensive and allows for automatic document publishing when used sensibly. Here are some best-practices that should be followed for documenting scripts.

1. All scripts should begin with a header that contains the following:
   - The overall purpose of the script
   - Any prerequisits needed in the form of extension modules or other scripts
   - The author of the script
   - The creation date of the script
   - When under version control, a suitable version number tag

   Here's an example of a good script header:

   ```
   % A script which takes measurements of the thingemijig and estimates
   % the amplitude of the thrust manipulator by extracting the coherence
   % between the first and second whizzmeters.
   %
   % The script requires the use of the 'BigMachine' extension module and
   % assumes the preprocessing of the raw data has been done with the script
   % entitled 'preprocess_BigMachine_data.m'.
   %
   % M Hewitson 2024-02-30
   %
   ```

2. Use cells to structure the code into logical sections with sensible 'chapter' headings.

3. Each line of code in a script should come with a line of comment explaining in plain language what you are doing. It should be possible to remove all the code from a script and still read what happened. This is a lofty goal, and does not always improve readability. Sometimes grouping a small number of lines of code together under one comment is better. In short, good judgement is required. Ask yourself, "will somebody else understand this script?" Or even, "will I understand this script if I look at it again in 1 year from now?"

Copying and modifying objects

Many methods in LTPDA can be used to modify an object. Generally, if you give an output variable, the original object(s) will be copied, rather than modified. Here's an example:

```
% Create a time-series ao
timeSeries = ao.randn(100, 10);

% take the absolute value of the data and store in another object
absData = timeSeries.abs(); % the original timeSeries object is left untouched

% take the absolute value of the data
timeSeries.abs(); % the original timeSeries object is modified; the original data is
discarded.
```

If you modify an object you should ask yourself if the variable name still makes sense after the modification. For example, the following would result in confusion:

```
% Create an ao
minusOne = ao(-1);

% take the absolute value of the data
minusOne.abs(); % The value of the ao is no longer -1 --> confusing!

% Create a time-series ao
timeSeries = ao.randn(100, 10);

% Estimate the PSD of the data in timeSeries
timeSeries.psd(); % The data in timeSeries is no longer a time-series --> confusing!
```

This is discussed further in the "Working with LTPDA objects" section of the LTPDA user manual.

Method usage and parameter lists

The following rules should be adhered to, whenever possible, in the use of LTPDA methods.

1. Don't chain methods together in a single command line. Don't do this:

```
% Create a time-series ao, take its PSD and plot it
iplot(psd(ao.randn(100,10)));

% An alternative to the above, but still hard to read
ao.randn(100,10).psd.iplot;
```

Rather do this:

```
% Create a time-series ao
noiseData = ao.randn(100, 10);

% Estimate PSD
S_noiseData = psd(noiseData);

% Plot the PSD
iplot(S_noiseData);
```

2. If a configuration plist contains multiple parameters, or if the same plist will be useful in multiple method call, consider creating the plist and holding a pointer to it with its own variable name. Don't do this:

```
% Create a some random noise
randomNoise1 = ao(plist('waveform', 'noise', 'fs', 10, 'nsecs', 100));

% Create some more random noise
randomNoise2 = ao(plist('waveform', 'noise', 'fs', 10, 'nsecs', 100));
```

Instead do:

```
% A plist for creating random noise
noisePlist = plist(...
                   'waveform', 'noise', ...
                   'fs', 10, ...
                   'nsecs', 100 ...
                   );

% Create a some random noise
randomNoise1 = ao(noisePlist);

% Create some more random noise
randomNoise2 = ao(noisePlist);
```

◀ Review of filtering and whitening in LTPDA   Introduction to LTPDA extension modules ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Introduction to LTPDA extension modules

LTPDA can be extended through the use of the imaginatively named 'ltpda extension modules'. You can create your own extension modules (see section "LTPDA Extension Modules" of the ltpda user manual) and you can use extension modules provided by others. For this training session, you will need to install the `LPF_DA_Module` extension module. It can be downloaded from 'Extension Modules' section of the LTPDA website: http://www.lisa.aei-hannover.de/ltpda/extension_modules/extension_modules.html.

To install an extension module, follow these steps:

1. Open the ltpda preferences by typing `LTPDAprefs` on the MATLAB terminal.
2. Select the 'Extensions' tab.
3. Click the 'Browse' button and navigate to the directory that contains the extension module on disk. For example, if you downloaded the extension module `LPF_DA_Module` to `/path/to/my/extensions/LPF_DA_Module`, then you need to select that `LPF_DA_Module` in the file browser. Alternatively you can simply type the path in the text field.
4. Once the correct path to the module is in the text field, click the '+' button to add that extension module. If all is well, you should see some activity on the MATLAB terminal as LTPDA installs the various methods of the module.
5. You should now type `clear classes` on the MATLAB terminal, or run `ltpda_startup` to complete the installation.

◀ LTPDA scripting best practices   Topic 2 – Simulating LPF noise in LTPDA. ➡

Introduction to LTPDA extension modules (LTPDA Toolbox)

LTPDA Toolbox™   ⬅ ➡

Topic 2 – Simulating LPF noise in LTPDA.

Topic 2 focusses on simulating noise outputs of LPF using the state-space models of LPF that are integrated in LTPDA.

1. Introduction to state-space models in LTPDA
2. Introduction to the LPF state-space models in LTPDA
3. Building an LTP model
4. Introduction to the various LPF noise models
5. Building an LPF model
6. Simulating noise
    1. Simulating harmonic oscillator noise
    2. Simulating capacitive actuation noise
    3. Simulating LPF noise
7. Changing system parameters
8. Simulate LPF with matched stiffness

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Introduction to state-space models in LTPDA

Introducing state-space models in LTPDA

State-space models in LTPDA are encapsulated by the `ssm` class. Building state-space models by hand, while possible, is complicated and beyond the scope of this training session. The most useful class constructor we can use for building `ssm` objects is the one that builds from built-in models.

For the purposes of this training session, we can treat the resulting state-space models as black boxes. They have inputs and outputs and describe a system in terms of a state-space realisation of the system's dynamical response. More details about the implementation is provided in the main LTPDA user manual.

State-space model objects in LTPDA are structured in the following way. Each model has a number of

- input blocks
- state blocks
- output blocks.

Each block has a number of ports. The exact numbers of blocks and ports depends on the underlying state-space structure of the system, but they need to properly match in order to allow the model to be consistent. The structure of a model is shown in the following figure:

IOS			states			inputs	
	blocs		ssblock1	ssblock2	ssblock3	inblock1	inblock2
		vars					
states	ssblock1	var1	A11				B12
		var2					
	ssblock2	var1					
		var2	A21	A22	A32		
		var3					
	ssblock3	var1	A31		A33	B31	
outputs	outBlock1	var1		C12			D12
	outBlock2	var1	C21		C23		
		var2					
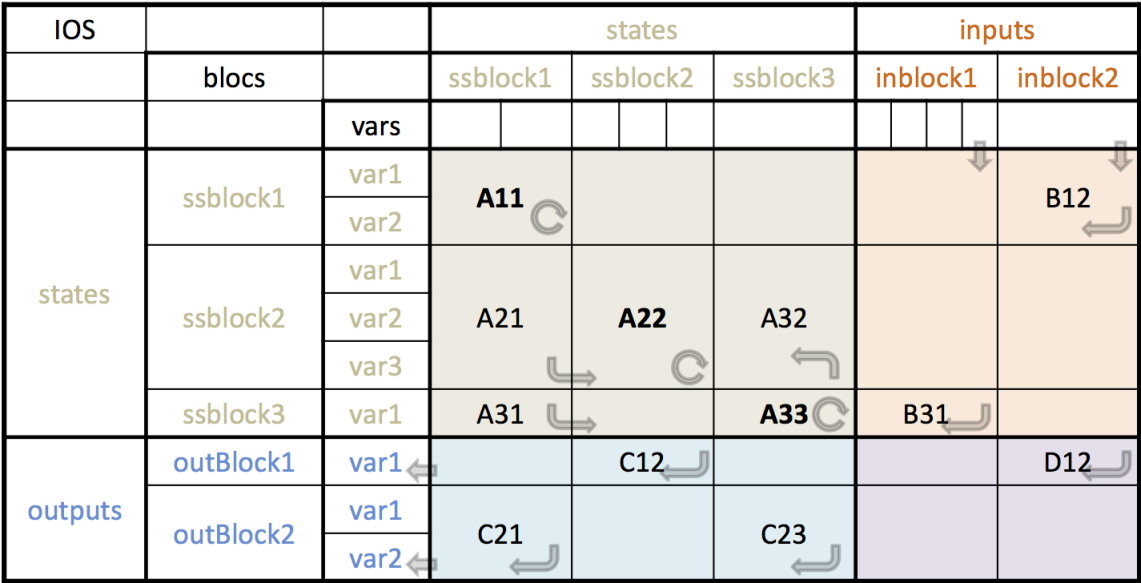
Building a state-space model in LTPDA

LTPDA comes with some example built-in models for the `ssm` class. One such example is a simple 1D harmonic oscillator. To build this model we use the 'built-in' model constructor of the `ssm` class. (Note: the 'built-in' model constructor can be used for all the different user classes in LTPDA.)

To see a list of the `ssm` built-in models available to you, you can do

```
% Get a list of available built-in ssm models
ssm.getBuiltInModels
```

Or you can use the graphical interface for browsing models:

```
% Use the model browser
LTPDAModelBrowser
```

To build one of these models, for example, the 1D Harmonic Oscillator, you can do:

```
% Build a Harmonic Oscillator model
sys = ssm(plist('built-in', 'HARMONIC_OSC_1D'))
```

You should then see details of the model on the terminal. To get a nicer view of the inputs and outputs for this model, you can do:

```
% View details of the model
sys.viewDetails
```

You should then see a window like the following:
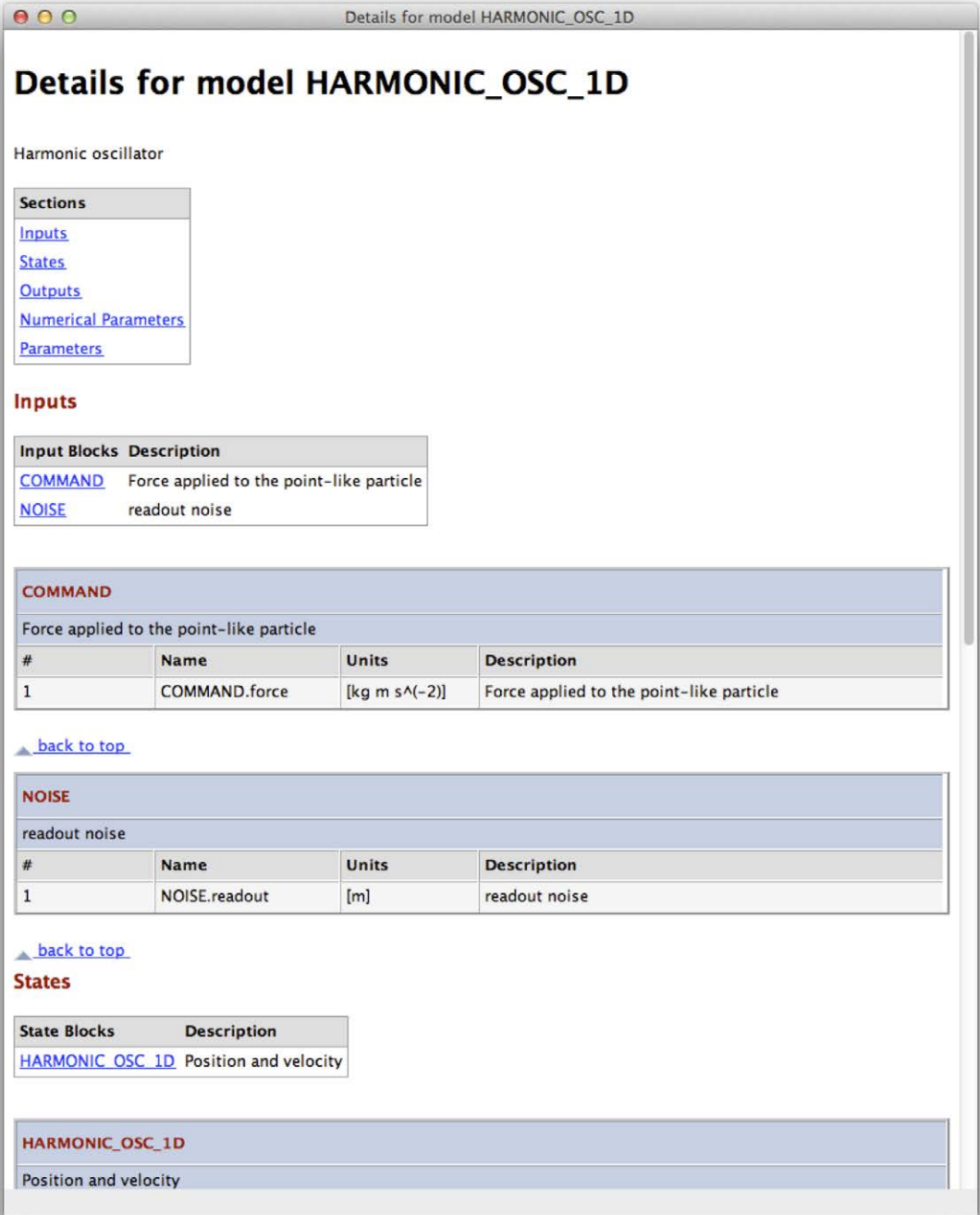
This should show details about the different blocks (inputs, states, outputs) and then the ports within those blocks.

In this case, the system has two input blocks, called "command" and "noise", that can be used to respectively the external force acting on the particle, and the additive noise of the readout observing the position of the particle itself. Each input block has a name, a description, and (in this case) only one port. Each port has an index, a name, a description, and some physical units for the quantity in input. The system has one state block, with two states (position and velocity of the particle). Each state has an index, a name, a description, and some physical units for the quantity it represents. In this case, the system has one output block, called "position", that represents the particle position as observed via the readout (meaning, with the readout noise added). The output block has a name, a description, and (in this case) only one port. Each port has an index, a name, a description, and some physical units for the quantity in output. Additionally, the report includes a description of the parameters, if any, that can be adjusted to characterize the system, and are represented in symbolic or numeric form.

Assembling state-space models in LTPDA

If you build multiple `ssm` objects, you can assemble them together using the `ssm/assemble` method. Although we don't need to do that in this training session, it may be useful to know the rules in case you want to build your own models, and because the LPF models we will be using are built in this way, assembling simpler subsystems. Essentially, if you ask LTPDA to assemble 2 or more models, then it will look for matching input and output blocks and connect them together. Here, matching means that the input block *name* matches the output block name, and that they both have the same *number of ports*. The ports are then connected in numerical order. The inputs of any block which gets connected to an output block will not appear in the final composite model.

Using state-space models in LTPDA

Once you have built your `ssm` model in LTPDA, you can go on and use it in various ways. The most common ways are to produce transfer functions (using `ssm/bode`) responses of the system or to simulate the time-domain system response to different inputs (using `ssm/simulate`). Both of these topics will be covered in later sections.

©LTP Team

Introduction to the LPF state-space models in LTPDA

LTP System Models

Let's start with a word on terminology. The current implementation of the LPF models in LTPDA uses:

- the term 'LTP' to refer to the entire system but without noise models
- the term 'LPF' to refer to 'LTP' + noise models

All of the LPF `ssm` models are provided by the `LPF_DA_Module` LTPDA extension module.

The 3D closed-loop dynamical system of LTP is modelled using a number of sub-models which are then assembled together to produce the final LTP model. The following diagram shows the full set of `ssm` models which are assembled together to give the main composite model 'LPF'.

The built-in models.
Marked are model names, followed by input/output blocks. Underlined are user inputs, italic are the systems normal outputs.
Constructor call uses the model name, with a small change if multiple models have the same name (eg: DFACS)
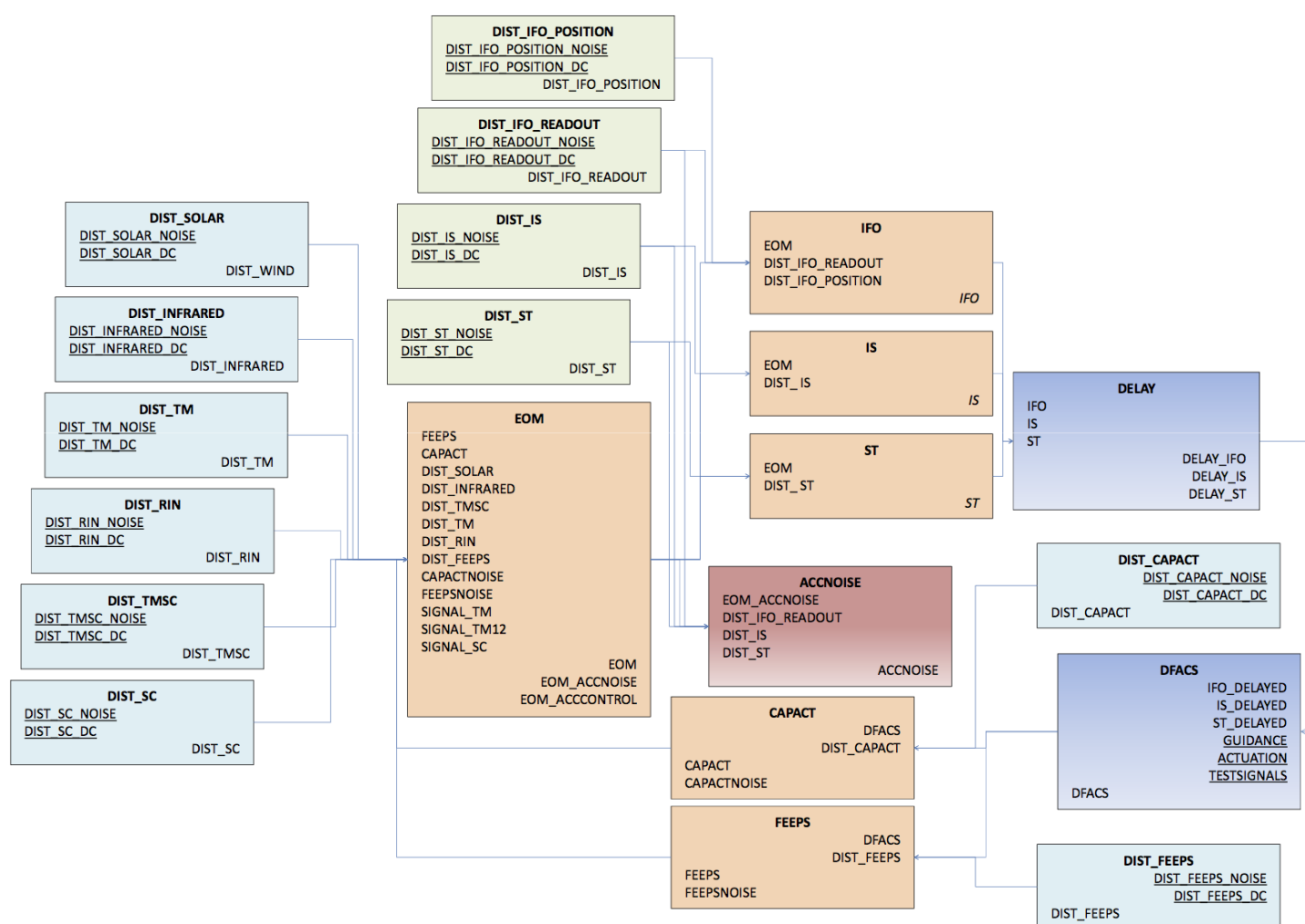
All noise models begin with the prefix `DIST_*`, and they are inputs to the corresponding model. The

noise models typically have one output block, whose name is the same as the model itself.
Each submodel has many parameters which all have default values. In section 7 you will learn how to change these parameter values at both build time and run time.

As well as having many parameters, each sub-model has many inputs and outputs. In the process of assembling the full LTP (or LPF) model, some of the inputs disappear (because they are connected to outputs). However, many do not.

The following diagram shows the full LTP model indicating many of the inputs and outputs you have access to.
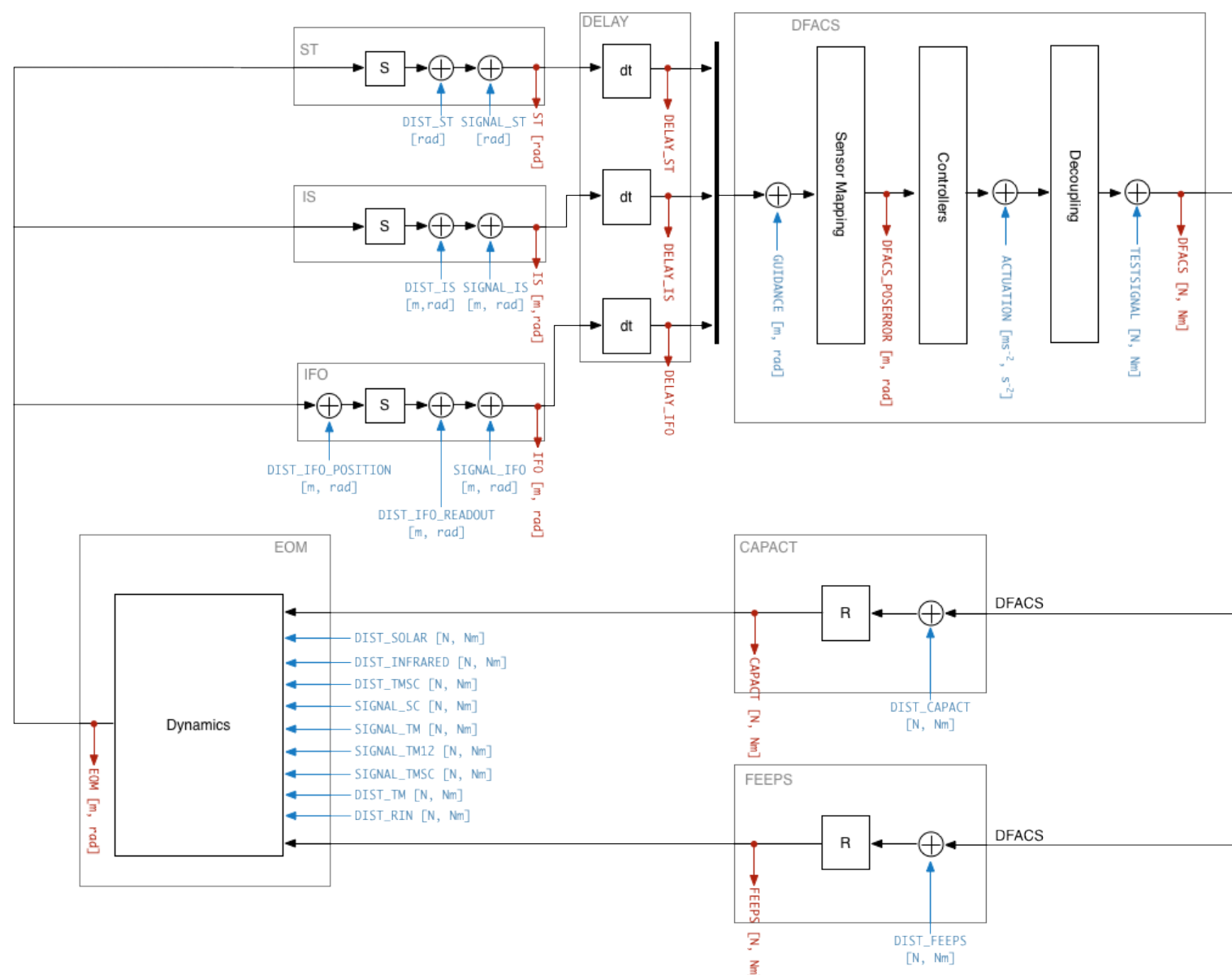
Any input named DIST_* will not appear in the full LPF model because the various noise models will connect to these. However, the noise models themselves also have inputs where you can inject noise/signals. These are described below.

LPF Noise Models

The LTP model (and its submodels) have many inputs labelled DIST_*. These inputs are designed for inputting noise. The output block has two ports, one named *_DC for the DC term, and one named *_NOISE for the fluctuating term. To simplify this process, we have created a set of noise models (noise shaping filters) which, when you input unit variance white noise, inject a particular coloured noise into the LTP model. The first figure above shows the various noise models.

©LTP Team

Building an LTP model

Building the LTP model is straightforward. It is simply another built-in model of the `ssm` class. Here's an example of building the default version of the LTP model:

```
% Build the default version of the LTP model
ltp = ssm(plist('built-in', 'LTP'));
```

To see what other versions of the LTP model are available, and the parameters that can be adjusted, you can look at the model documentation:

```
help ssm_model_LTP
```

then click on the 'Model Information' link to open a documentation window like the one shown here:

Model Report for ssm/ssm_model_LTP

Model Report for ssm/ssm_model_LTP

A built-in model that constructs a model of the LTP.

This model is a closed model of the LTP.

Model Versions	Description
Standard	This version combines various LTP models in to a closed loop. If the model is built continuous, the ACCNOISE model is not combined. This following versions are used: **System / Version** DFACS / CDR CAPACT / Standard DELAY / Standard FEEPS / Standard EOM / Standard IFO / Standard IS / Standard ST / Standard ACCNOISE / Standard
MDC3	This version combines various LTP models in to a closed loop. This following versions are used: **System / Version** DFACS / MDC3 CAPACT / Standard DELAY / Standard FEEPS / Standard EOM / Standard IFO / MDC3 IS / Standard ST / Standard
TN3045	This version combines various LTP models into a continuous closed loop. This following versions are used:

As you can see, there are various 'Versions' of the model available. Each version uses different versions of the various submodels, and different default values of the parameters.

The default version is always the first one!

In addition, you can scroll to the details section for a particular model version and see what configuration parameters are available to you.

Since the LTP model is assembled from sub-models, it is possible, for each version of the LTP model, to click on the name of the sub-models to access the documentation window associated with sub-model itself.

For example, we could build a version of the LTP model which is 1D and has DFACS set to Science Mode 2.2. To do that, you would do:

```
        % Plist defining the LTP model we want to build
        modelPlist = plist(...
                    'built-in', 'LTP', ...
                    'Version', 'Standard', ...
                    'DIM', 1, ...
                    'DFACS Mode', 'SCI2.2 M1' ...
                    );

        % Build the defined LTP model
        ltp = ssm(modelPlist);
```

To look at the details of the model you've built, you can either enter the variable name on the MATLAB terminal without a terminating semicolon and hit enter. Or, more effectively, you can use the `ssm/viewDetails` method which will display details of the model you've constructed in a documentation window:

```
        ltp.viewDetails
```

◀ Introduction to the LPF state-space models in LTPDA

Introduction to the various LPF noise models ▶

©LTP Team

Introduction to the various LPF noise models

As with the system models, the various noise models also come in different versions. You can build these noise models individually and inspect their responses. Let's look at a couple of examples.

Capacitive actuation Noise Model

For example, let's build the noise model used for generating force and torque noise of the GRS actuation. The model is called `DIST_CAPACT`.
We start by inspecting the model to see what versions are available, and, within each version, the parameters available for setting. Start with opening the model documentation:

```
help ssm_model_DIST_CAPACT
```

and click on the 'Model Information' link. As we can see, there are 5 versions available. All of them allow the user to choose a parameter for the dimension of the model, so to operate only along x (1D), along x, y and phi (2D), or along all degrees of freedom (3D). In addition, you can scroll to the details section for a particular model version and see what configuration parameters are available to you. For instance, in this case we have different pole–zero models that characterize the noise shaping filters that color unit–variance white noise to mimick the force and torque noise associated with the electrostatic actuation provided bt the GRS and the FEE. After deciding the version we want to build, and the values of the parameters we want to set, we can build it by doing:

```
% Build the default version of the GRS CAPactive ACTuation noise model
GRScapactNoise = ssm(plist('built-in', 'DIST_CAPACT'));
```

Now that we've built the object, we can use the `ssm/viewDetails` method to inspect the various inputs and outputs of this model:

```
GRScapactNoise.viewDetails
```

so to understand how the input blocks are called, how many and which ports we have for each input, and how many and which outputs are available.

IFO readout Noise Models

As another example, let's build the noise model used for generating readout noise of the interferometer. The model is called `DIST_IFO_READOUT`.
Again, start by inspecting the model to see what versions are available, and, within each version, the parameters available for setting. As before, open the model documentation with:

```
help ssm_model_DIST_IFO_READOUT
```

As we can see, there are 4 versions available. All of them allow the user to choose a parameter for the dimensions of the model, so to operate only with the longitudinal degrees of freedom (1D), or also with angular ones (2D and 3D). In addition, you can scroll to the details section for a particular model version and see what configuration parameters are available to you. For

instance, in this case we have different pole-zero models that characterize the noise shaping filters that color unit-variance white noise to mimick the readout noise for the different IFO components. After deciding the version we want to build, and the values of the parameters we want to set, we can build it by doing:

```
% Build the default version of the IFO Readout noise model
IFOreadoutNoise = ssm(plist('built-in', 'DIST_IFO_READOUT'));
```

Now that we built the object, we can use the `ssm/viewDetails` method to inspect the various inputs and outputs of this model:

```
IFOreadoutNoise.viewDetails
```

so to understand how the input blocks are called, how many and which ports we have for each input, and how many and which outputs are available.

This noise source is added after the modelling of the actual conversion of the TMs position into an IFO reading, so it represents noise sources such as shot noise in the beams, electronics noise (additive, digitization, ...).

Other noise sources that act 'common mode' on all IFO channels, such as laser frequency fluctuations, are modelled by the `DIST_IFO_POSITION` noise model, that can be built by doing:

```
% Build the default version of the IFO so-called position noise model
IFOfrequencyNoise = ssm(plist('built-in', 'DIST_IFO_POSITION'));
```

Now that we built the object, we can use the `ssm/viewDetails` method to inspect the various inputs and outputs of this model:

```
IFOfrequencyNoise.viewDetails
```

so to understand how the input blocks are called, how many and which ports we have for each input, and how many and which outputs are available.

◀ Building an LTP model   Building an LPF model ▶

©LTP Team

Building an LPF model

Building an LPF model is done in just the same was as building the LTP and the noise models. For example, to build the LPF model version 'Best Case June 2011' you can do:

```
% Build the desired version of the LPF model
lpf = ssm(plist('built-in', 'LPF', 'Version', 'Best Case June 2011'));
```

To see what other versions of the LPF `ssm` model are available, and how to configure them, use the model documentation:

```
% Get help on the model
help ssm_model_LPF
```

◀ Introduction to the various LPF noise models   Simulating noise ▶

©LTP Team

Building an LPF model (LTPDA Toolbox)

Simulating noise

Simulation Basics

Running time-domain simulations of `ssm` models is done using the `ssm/simulate` method. It has three main operational modes:

1. Simulate using internal noise generation
2. Simulate with injected signals
3. Simulate with both injected signals and noise

Whichever mode you are operating in, you have to tell simulate:

- which inputs you want to use;
- what noise level you want to have (or what signal you want to inject);
- which outputs you would like to simulate.

In the following we will explore this via a few examples:

1. [Simulating harmonic oscillator noise](#)
2. [Simulating capacitive actuation noise](#)
3. [Simulating LPF noise](#)

◀ Building an LPF model   Simulating harmonic oscillator noise ▶

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Simulating harmonic oscillator noise

Basic system simulations

Let's begin with a simple example, namely the harmonic oscillator. The system can be built easily, specifying the values for the key parameters (mass, spring constant, viscous damping coefficient):

```
% Build a Harmonic Oscillator model
% Physical parameters
m     = .1;     % mass: 0.1 kg
k     = 1e-2;   % spring constant: 1e-2 N / m
vbeta = 1e-3;   % viscous damping coefficient: 1e-3 N s / m

% Definition plist
buildPlist = plist(...
                'built-in', 'HARMONIC_OSC_1D', ...
                'm', m, ...
                'vbeta', vbeta, ...
                'k', k ...
                );

% Build the ssm model
harm_osc = ssm(buildPlist);
```

Now, a call to ssm/viewDetails is telling us that the system has two input blocks, namely 'COMMAND' and 'NOISE', representing respectively the force acting on the particle and the readout additive noise. Each of the input blocks has a single port, respectively 'COMMAND.force' and 'NOISE.readout', where data needs to have the proper physical units.

```
%% Look at the model details
harm_osc.viewDetails
```

The system is continuous, i.e., it has a time-step of zero. We can verify this by inspecting its timestep property:

```
%% Check if the model is continuous
harm_osc.timestep
```

Now we want to calculate the system transfer function of the input to the outputs, in order to estimate the effect of the sources. In order to do that, we can call the ssm/bode method, specifying the input port, the output port, and the frequency range. Here is an example, where we look at the response of the harmonic oscillator mass position to the application of an external force, in the range from 0.1 mHz t 1 Hz:

```
%% Calcolate the bode response from force to displacement for the continuous system
forceBodePlist = plist(...
  'inputs', 'COMMAND.force', ...
  'outputs', 'HARMONIC_OSC_1D.position', ...
  'f', logspace(-4, 0, 1000) ... % from 0.1mHz to 1Hz
  );

bode_output            = bode(harm_osc, forceBodePlist);
harm_osc_cont_resp_force = bode_output.unpack();
```

> NOTE: The output of bode is a matrix object. The single response we want (1 input to 1 output) is represented by the single ao inside the output matrix. So we unpack that single object from the matrix.

Similarly, we could look at the response of the harmonic oscillator mass position to the additive readout noise, in the range from 0.1 mHz t 1 Hz:

```
%% Calculate the bode response from readout noise to displacement for the continuous system
readoutBodePlist = plist(...
  'inputs', 'NOISE.readout', ...
  'outputs', 'HARMONIC_OSC_1D.position', ...
  'f', logspace(-4, 0, 1000) ... % from 0.1mHz to 1Hz
  );

bode_output            = bode(harm_osc, readoutBodePlist);
harm_osc_cont_resp_readout = bode_output.unpack();
```

Discretizing the system

In order to simulate the ssm models, we need to discretize them. This is done by setting the time-step to a non-zero value.

```
%% Discretize the system to be simulated at 10 Hz
timestep = 0.1;
harm_osc_discrete = harm_osc.modifyTimeStep(timestep);
```

The discretization of a continuous system is a delicate step, because it involves a change from s-domain to z-domain representation. In this case, we want to evaluate the impact of discretizing our harmonic oscillator at 10 Hz. To do so, we go ahead and calculate the transfer function of the discretized system, for both inputs, and compare them with the continuous ones.

```
% Calculate the bode response from force to displacement for the discrete system
bode_output            = bode(harm_osc_discrete, forceBodePlist);
harm_osc_disc_resp_force = bode_output.unpack();

% Calculate the bode response from readout noise to displacement for the discrete system
bode_output              = bode(harm_osc, readoutBodePlist);
harm_osc_disc_resp_readout = bode_output.unpack();

% Compare the transfer functions for the discrete and continuous case
plot_plist = plist('linestyles', {'-', '--'});

iplot(harm_osc_cont_resp_readout, harm_osc_disc_resp_readout, plot_plist);
iplot(harm_osc_cont_resp_force, harm_osc_disc_resp_force, plot_plist);
```
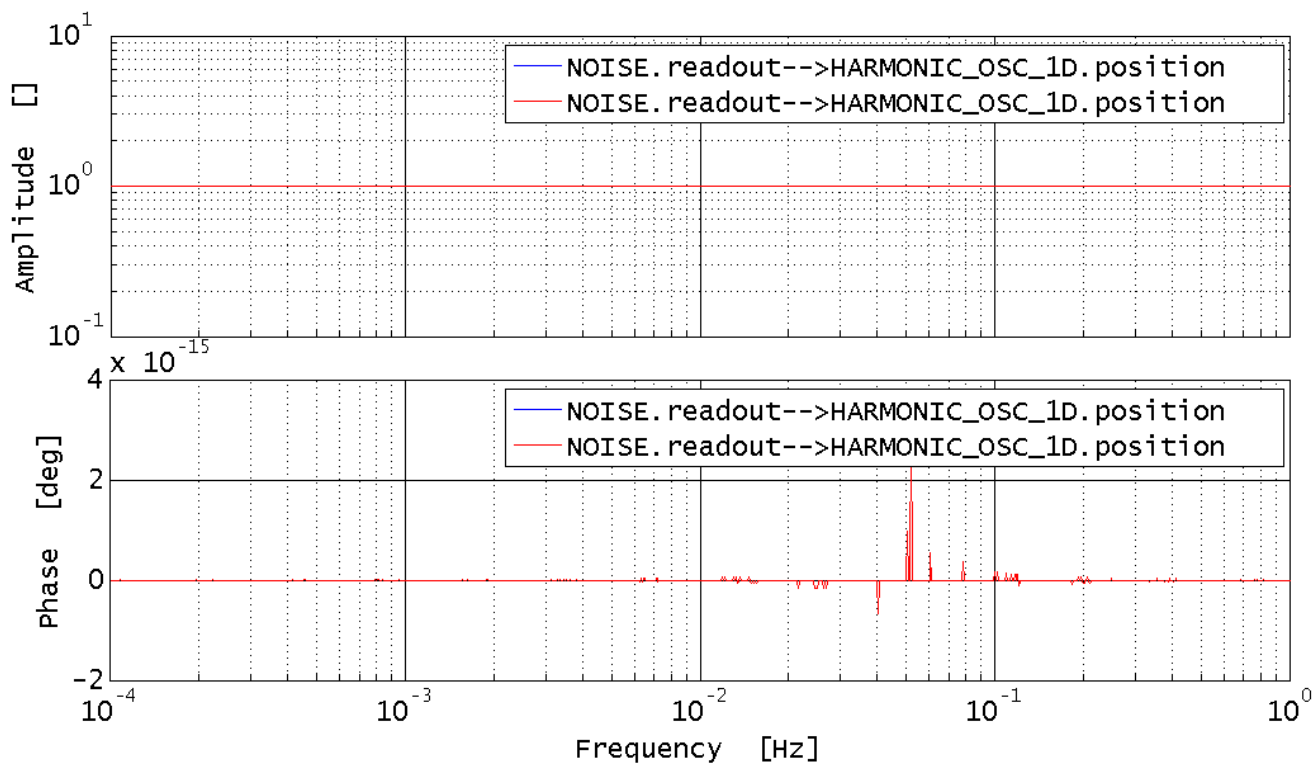
We expect to see two plots similar to the following ones, showing that the effect of discretizing at this rate is very tiny, and as expected it impacts only at very high frequencies:
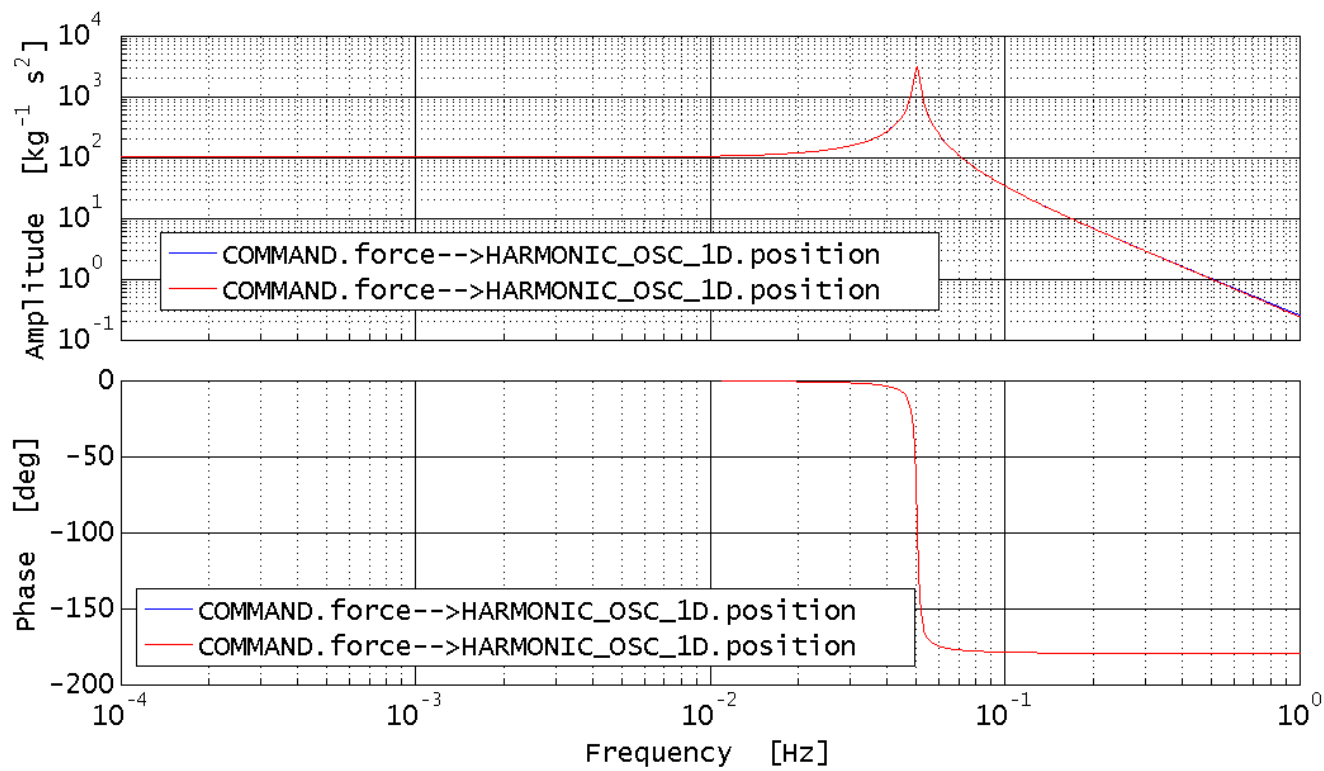
Simulating the system

Let's assume we only want to simulate the effect of the external force noise, so we simulate a single noise source. We can do that by specifying a single input name and a value for the CPSD of that noise source:

```
%% Simulate the system behavior: only force noise
% Simulation configuration plist
simPlist_force = plist(...
    'CPSD Variable Names', 'COMMAND.force', ...
    'CPSD', 1e-6, ...
    'Return outputs', {'HARMONIC_OSC_1D.position'}, ...
    'Nsamples', 100000 ...
    );

% Launch the simulation
sim_output = harm_osc_discrete.simulate(simPlist_force);

% Extract the AO with the postion data
out_force = sim_output.unpack();

% Plot the results
iplot(out_force)
```

You should see a plot similar to the following one:
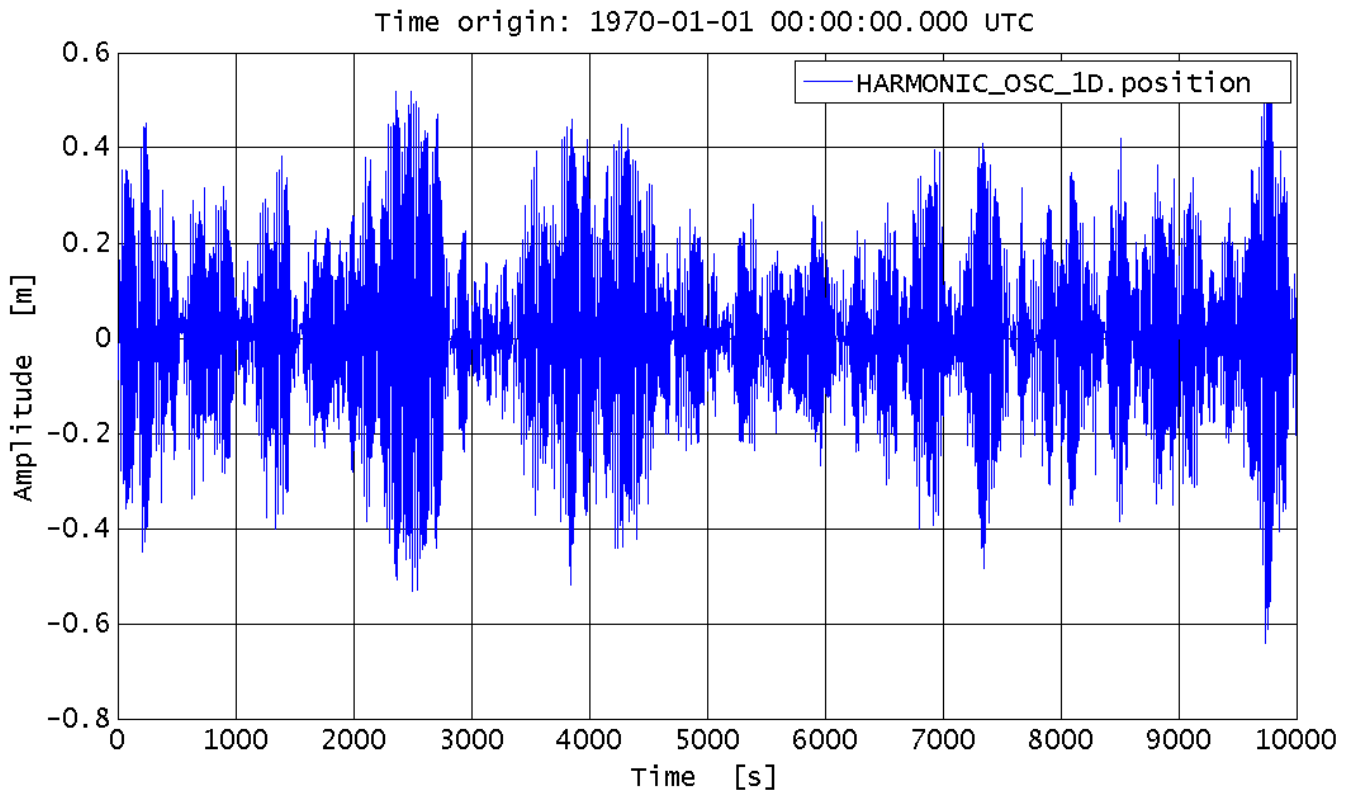
Similarly, let's assume we only want to simulate the effect of the readout noise, so we simulate a single noise source. We can do that by specifying a single input name and a value for the CPSD of that noise source:

```
%% Simulate the system behavior: only readout noise
% Simulation configuration plist
simPlist_readout = plist(...
    'CPSD Variable Names', 'NOISE.readout', ...
    'CPSD', 0.01, ...
    'Return outputs', {'HARMONIC_OSC_1D.position'}, ...
    'Nsamples', 100000 ...
    );

% Launch the simulation
sim_output = harm_osc_discrete.simulate(simPlist_readout);

% Extract the AO with the postion data
out_readout = sim_output.unpack();

% Plot the results
iplot(out_readout)
```

You should see a plot similar to the following one:
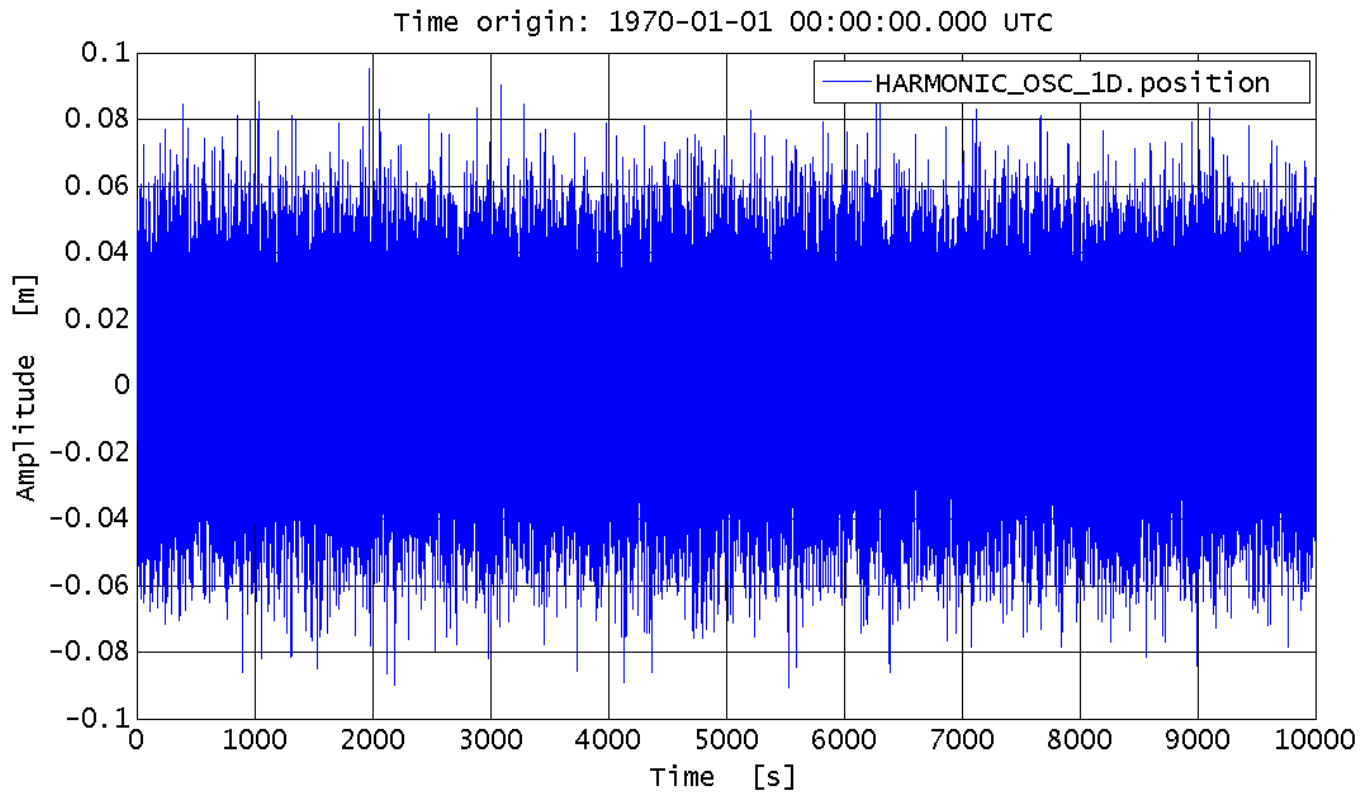
You can verify that by changing the CPSD quantity, the noise level changes accordingly.

Estimating the PSD

Eventually, we want to estimate the PSD of the displacement in the two cases. Just for curiosity, we change the number of averages in the two cases.

```
%% Estimate the PSD
% PSD estimation configuration plist
psdPlist = plist(...
    'scale', 'PSD', ...
    'order', 1, ...
    'win', 'BH92' ...
    );

% Estimate the PSD
S_out_force   = out_force.psd(psdPlist.pset('navs', 16));
S_out_readout = out_readout.psd(psdPlist.pset('navs', 100));

% Plot the ASD (sqrt(PSD))
iplot(sqrt(S_out_readout));
iplot(sqrt(S_out_force));
```

We expect to see two plots similar to the following ones:
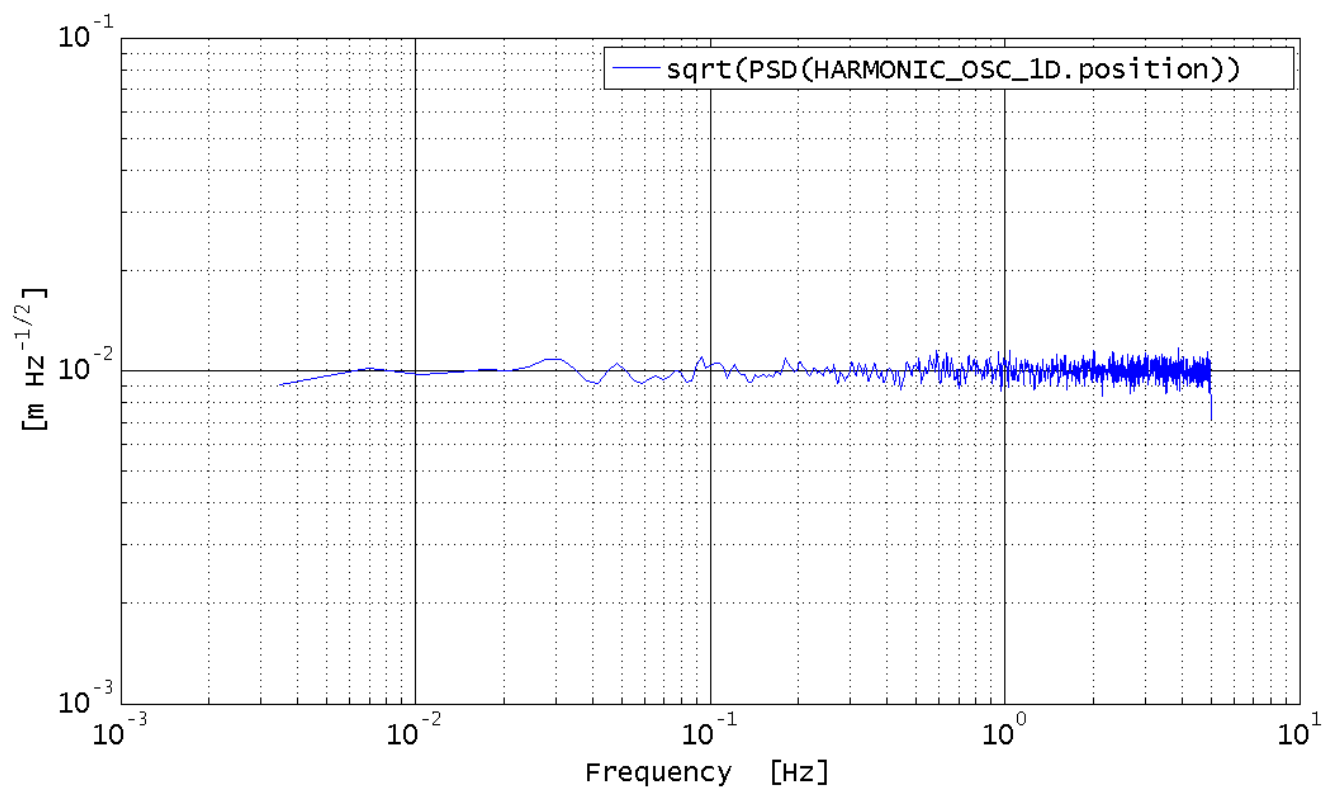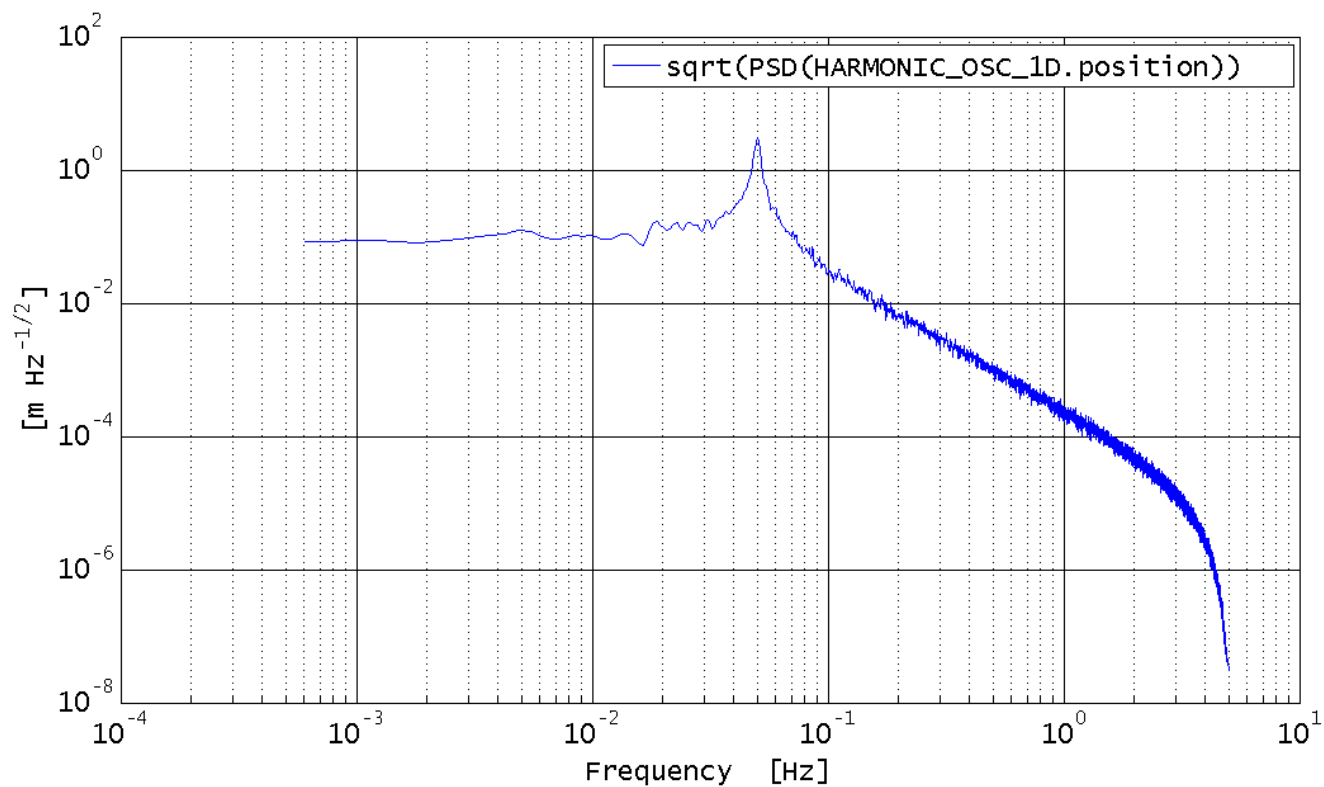
Again, you can verify that by changing the CPSD quantity, the noise levels change accordingly.

©LTP Team

Simulating capacitive actuation noise

LPF noise source simulations

In this section we want to investigate how the DIST_* models behave, by simulating them. Let's begin with the DIST_CAPACT model, that provides the force/torque noise associated with the electrostatic actuation in the LPF TMs. We build the standard version of the system:

```
% Build the default version of the GRS CAPactive ACTuation noise model
GRScapactNoise = ssm(plist('built-in', 'DIST_CAPACT'));
```

We just recall that is it possible to obtain more informations about the system by:

```
%% Get help on the GRS CAPactive ACTuation noise model
help ssm_model_DIST_CAPACT
```

and:

```
%% Obtain more details about this model
GRScapactNoise.viewDetails
```

Now, as we can see from the call to `ssm/viewDetails`, the system has two input blocks, namely 'DIST_CAPACT_NOISE' and 'DIST_CAPACT_DC', representing respectively the DC forces/torques acting on the TMs and the fluctuating part (the noise). Each of the input block has 12 ports, one for each degree of freedom of the two test-masses.
We also see that the system has one output block, namely 'DIST_CAPACT' representing the total forces/torques acting on the TMs. The output block has 12 ports, one for each degree of freedom of the two test-masses.
Now we want to calculate the system transfer function of the input to the outputs, in order to estimate the effect of the sources. In order to do that, we can call the `ssm/bode` method, specifying the input port, the output port, and the frequency range. Here is an example, where we look at the response of the force noise along x acting on TM2 to the noise in input for the same quantity. Practically speaking, we charaterize the transfer function of the noise shaping filter.

```
%% Calculate the bode response of the noise coloring filter for the continuous system, tm2_x port
tm2_xNoisePlist = plist(...
                        'inputs', 'DIST_CAPACT_NOISE.tm2_x', ...
                        'outputs', 'DIST_CAPACT.tm2_x', ...
                        'f', logspace(-4, log10(0.5), 1000) ... % from 0.1mHz to 0.5Hz
                        );

bode_out                      = bode(GRScapactNoise, tm2_xNoisePlist);
GRScapactNoise_tm2_x_cont_resp = bode_out.unpack();
```

> Remember: the output of bode is a `matrix` object. The single response we want (1 input to 1 output) is represented by the single `ao` inside the output matrix. So we unpack that single object from the matrix.

Discretizing the system

In order to simulate the `ssm` models, we need to discretize them, by setting the time-step to a non-zero value.

```
%% Discretize the system to be simulated at 1 Hz
timestep = 1;
GRScapactNoise_discrete = GRScapactNoise.modifyTimeStep(timestep);
```

The discretization of a continuous system is a delicate step, because it involves a change from s-domain to z-domain representation. In this case, we want to evalue the impact of discretizing our capacitive actuation noise model at 1 Hz. To do so, we go ahead and calculate the transfer function of the discretized system, for both inputs, and compare them with the continuous ones.

```
%% Calculate the bode response of the noise coloring filter for the discrete system, tm2_x port
bode_out                      = bode(GRScapactNoise_discrete, tm2_xNoisePlist);
GRScapactNoise_tm2_x_disc_resp = bode_out.unpack();

%% Compare the transfer functions for the discrete and continuous case
iplot(GRScapactNoise_tm2_x_cont_resp, GRScapactNoise_tm2_x_disc_resp);
```

We expect to see a plot similar to the following one, showing that the effect of discretizing at this rate is very tiny, and as expected it impacts only at very high frequencies:
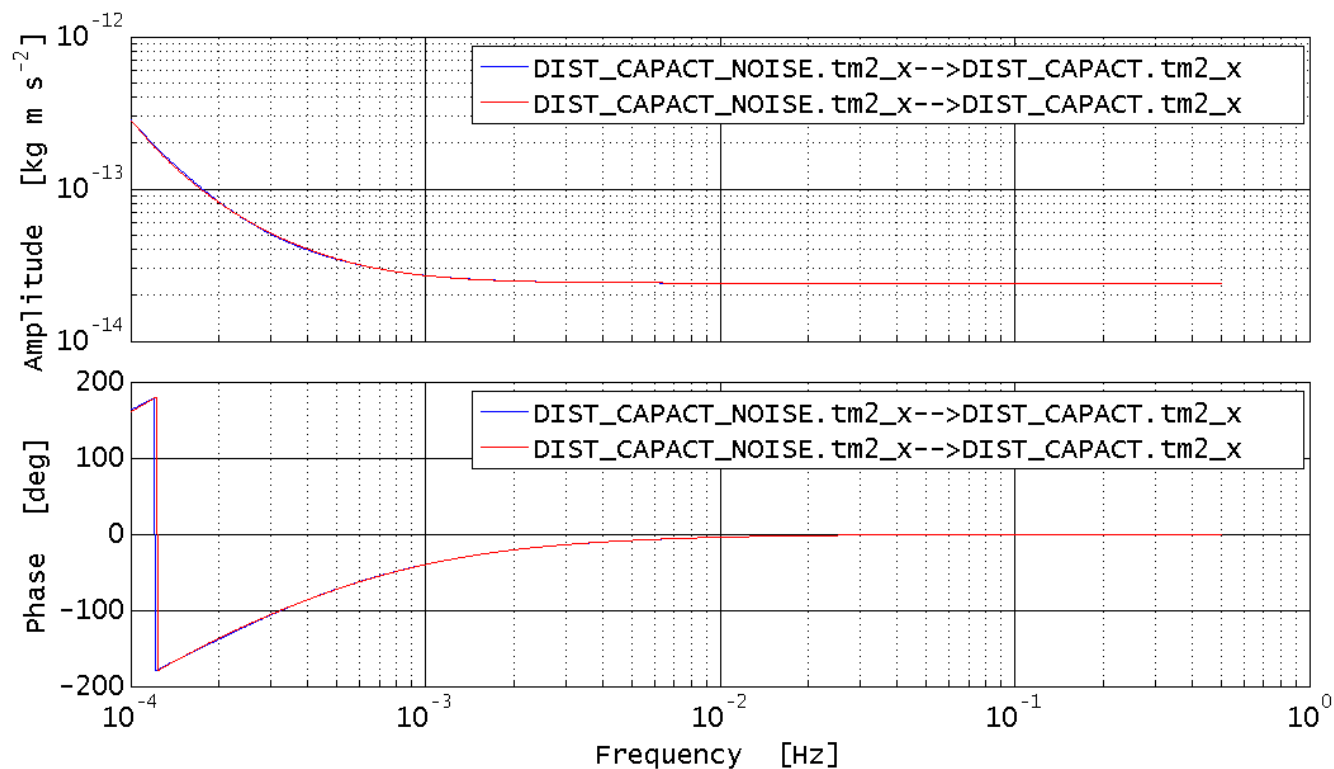
Simulating the system

Let's assume we only want to simulate the effect of the force noise acting on TM2 along x, so we simulate a single noise source. We can do that by specifying a single input name and a value for the CPSD of that noise source:

```
%% Simulate the system behavior: only tm2_x force noise
% Simulation configuration plist
simPlist_noise = plist(...
    'CPSD Variable Names', 'DIST_CAPACT_NOISE.tm2_x', ...
    'CPSD', 1, ...
    'Return outputs', {'DIST_CAPACT.tm2_x'}, ...
    'Nsamples', 1e6 ...
    );

% Launch the simulation
sim_output = GRScapactNoise_discrete.simulate(simPlist_noise);

% Extract the AO with the force data
out_tm2_x = sim_output.unpack();

% Plot the results
iplot(out_tm2_x)
```

The output of `ssm/simulate` is always encapsulated inside a a `matrix` object, so we can handle multiple output variables (blocks and ports) together. If we want to extract the individual `ao` object with the individual noise realizations for each port, we use the `matrix/unpack` method. We expect to see a plot similar to the following one:
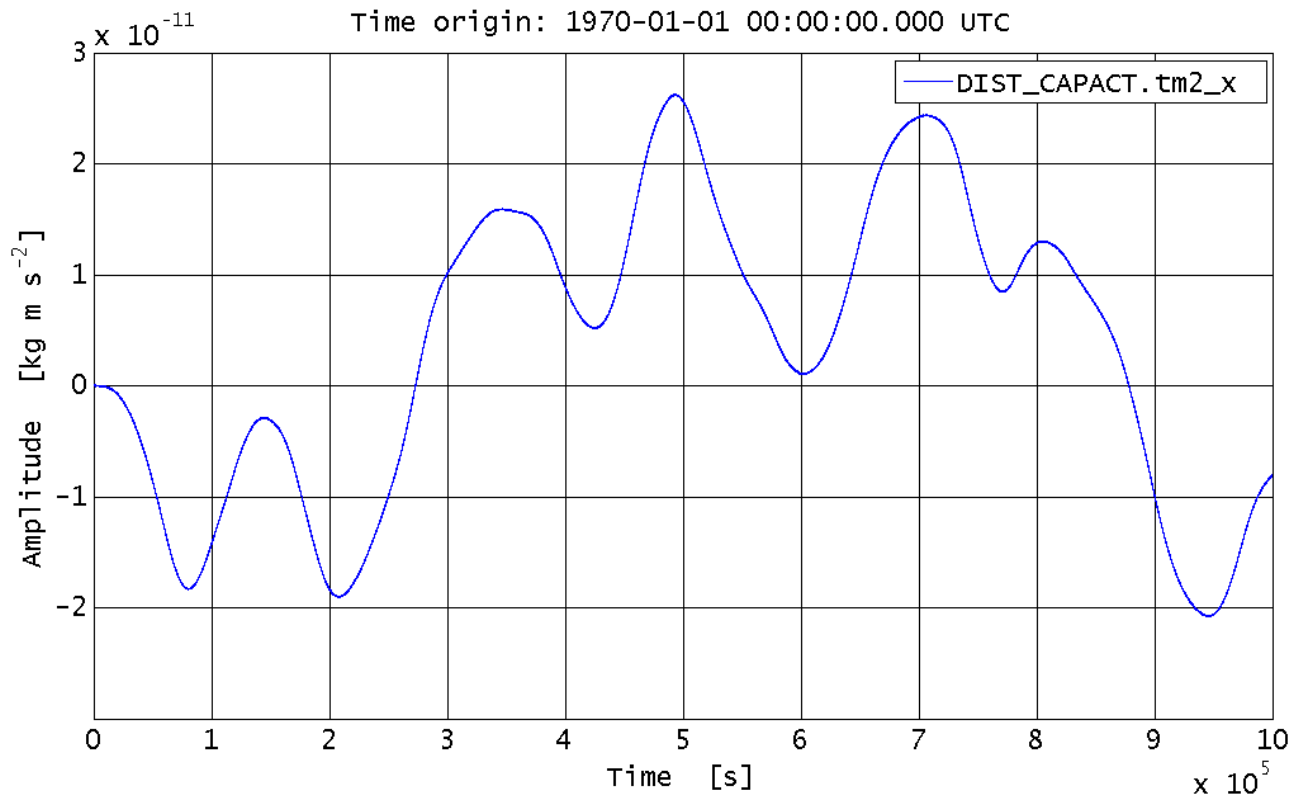
You can verify that by changing the CPSD quantity, the noise level changes accordingly.

Estimating the PSD

Eventually, we want to estimate the PSD of the force. We also want to compare the estimated PSD with the expected output of the coloring filter; in order to do that, we have to change the units, so to account for the fact that the input CPSD is expressed in Hz^-1.

```
%% Estimate the PSD
% PSD estimation configuration plist
psdPlist = plist(...
    'scale', 'PSD', ...
    'order', 1, ...
    'win', 'BH92', ...
    'navs', 25 ...
    );

% Estimate the PSD
S_out_tm2_x = out_tm2_x.psd(psdPlist);

% Plot the PSD and compare with the expected noise shape
% We have to account for the unit of the input to the coloring filter
GRScapactNoise_tm2_x_disc_resp.setYunits(sqrt(S_out_tm2_x.yunits));
iplot(sqrt(S_out_tm2_x), abs(GRScapactNoise_tm2_x_disc_resp));
```
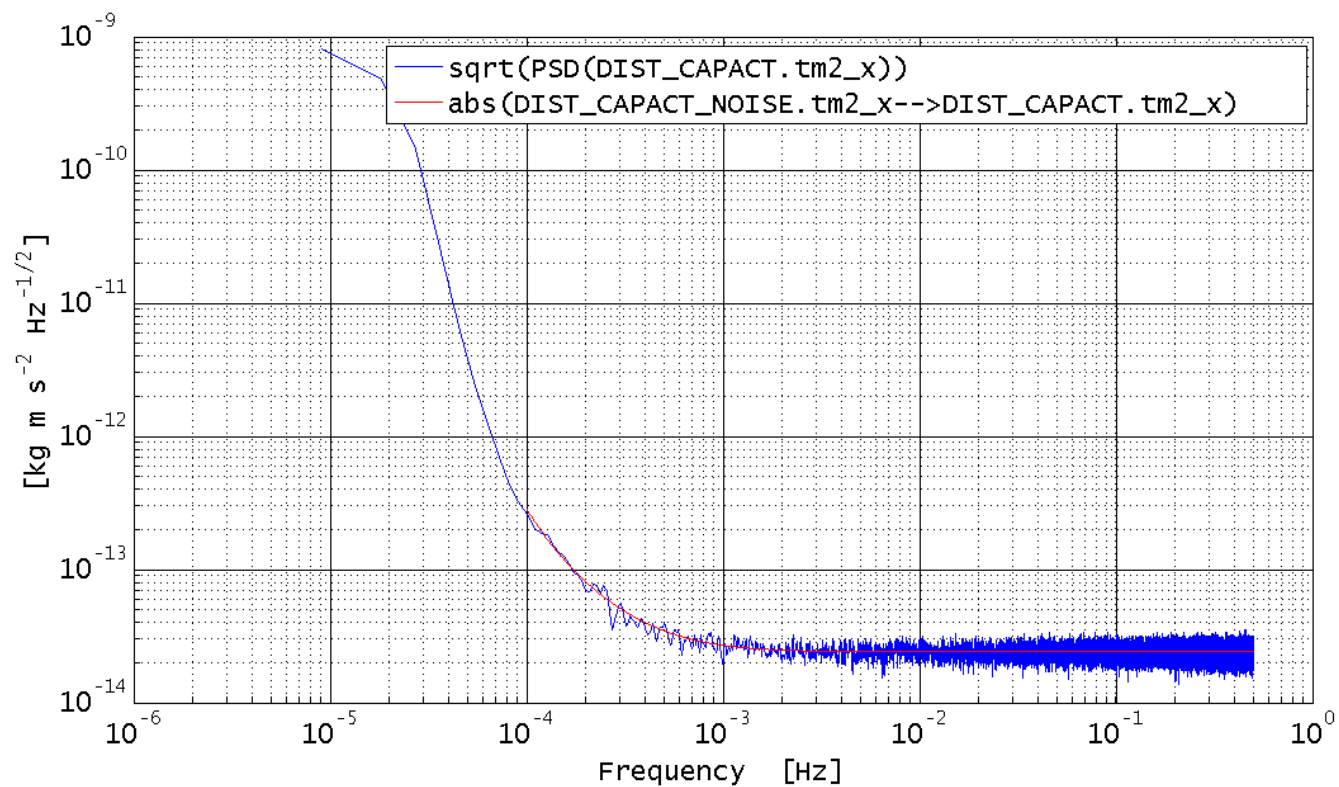
We expect to see a plot similar to the following one:

Again, you can verify that by changing the CPSD quantity, the noise levels change accordingly.

©LTP Team

Simulating LPF noise

Here we will simulate the LPF system using the internal noise generator. In Topic 4 you will see how to inject signals into the simulation.

LPF noise simulations: single noise source

You can set up the simulation to simulate a single noise source for LPF by specifying a single input name and a value for the variance of that noise. Let's suppose we want to simulate the effect of interferometer readout noise on the `x1` interferometer and look how that appears in the two longitudinal interferometer outputs. Here's how to do it:

```
% Build the default version of the LPF model
lpf = ssm(plist('built-in', 'LPF'));

% Define the simulation plist for a 10000 sample simulation
simPlist = plist(...
            'CPSD Variable Names', 'DIST_IFO_READOUT_NOISE.x1', ...
            'CPSD', 1, ...
            'Return outputs', {'IFO.x1', 'IFO.x12', 'DIST_IFO_READOUT.x1'}, ...
            'Nsamples', 10000 ...
            );

% Run the simulation
out = simulate(lpf, simPlist);
```

The output of the `ssm/simulate` method is a `matrix` object containing a series of time-series analysis objects, one for each requested output. Please notice that we requested as output also the quantity 'DIST_IFO_READOUT.x1' which is an (internal) input to the system, but can also be passed in output.
You can plot the two time-series by doing:

```
% Plot on subplots
iplot(out, plist('arrangement', 'subplots'));

% Or, unpack them and plot individually
[o1, o12, noise] = unpack(out);
iplot(o1)
iplot(o12)
iplot(noise)
```

It's also interesting to look at the spectra of these two outputs. Since we do not plan to combine the spectra later on, we delegate to the spectral estimator `ao/psd` the calculation of the square root of the spectra themselves, so to ease the plotting steps. Here's how:

```
% Plist for configuring the PSD estimator
psdPlist = plist(...
            'navs', 16, ...
            'scale', 'ASD', ...
            'order', 1, ...
            'win', 'BH92' ...
            );

% Estimate the PSD of each output
[S_o1, S_o12, S_noise] = psd(o1, o12, noise, psdPlist);

% Plot the results
iplot(S_o1, S_o12, S_noise);

% Note: you can also use the matrix/psd method
S_out = psd(out, psdPlist);

% Unpack them and plot individually
[S_o1, S_o12, S_noise] = unpack(S_out);
iplot(S_o1, S_noise)
iplot(S_o12)
```

Covariance or CPSD?

The `ssm/simulate` method has two possible noise configuration inputs. Both use the internal noise generator. The difference is simply in the scaling:

- When specifying the 'CPSD', the noise is scaled according to the sample rate of the simulation so to preserve the signal energy content.
- When specifying the 'COV', the scaling is not performed.

As such, most of the time you will want to use the 'CPSD' inputs.

LPF noise simulations: two correlated noise sources

Simulating with more than one noise source requires setting of the inputs and (co-)variance just as in the single noise case. Suppose we want to activate the readout noise in both the `x1` and `x12` IFO. And further suppose that these two readout noises should be correlated. To simulate this, we should do:

```
% Define the simulation plist for a 100000 sample simulation
simPlist = plist(...
                'CPSD Variable Names', {'DIST_IFO_READOUT_NOISE.x1', 'DIST_IFO_READOUT_NOISE.x12'}, ...
                'CPSD', [1 0.5; 0.5 1], ...
                'Return outputs', {'IFO.x1', 'IFO.x12', 'DIST_IFO_READOUT.x1', 'DIST_IFO_READOUT.x12'}, ...
                'Nsamples', 100000 ...
                );

% Run the simulation
out = simulate(lpf, simPlist);
```

We can now extract the outputs and look at their spectra and cross-spectra, as well as the coherence between the measured `x1` and `x12` signals:

```
% Unpack the signals from the simulation output matrix
[o1, o12, x1noise, x12noise] = unpack(out);

% Estimate PSD of the measured outputs
[S_o1, S_o12] = psd(o1, o12, psdPlist);

% Estimate CPSD of the measured outputs
S_o1o12 = cpsd(o1, o12, psdPlist);

% Estimate COHERENCE of the measured outputs
C_o1o12 = cohere(o1, o12, psdPlist);

% Plot results
iplot(S_o1, S_o12, abs(sqrt(S_o1o12)));
```

The warnings shown on the terminal are associated with the parameters that are peculiar only to the `ao/psd` method, and are not recognized by the other spectral estimators. We can ignore them, and take advantage of re-using the plist for the common parameters (window, number of averages, detrending order). This should result in a plot something like the one below:
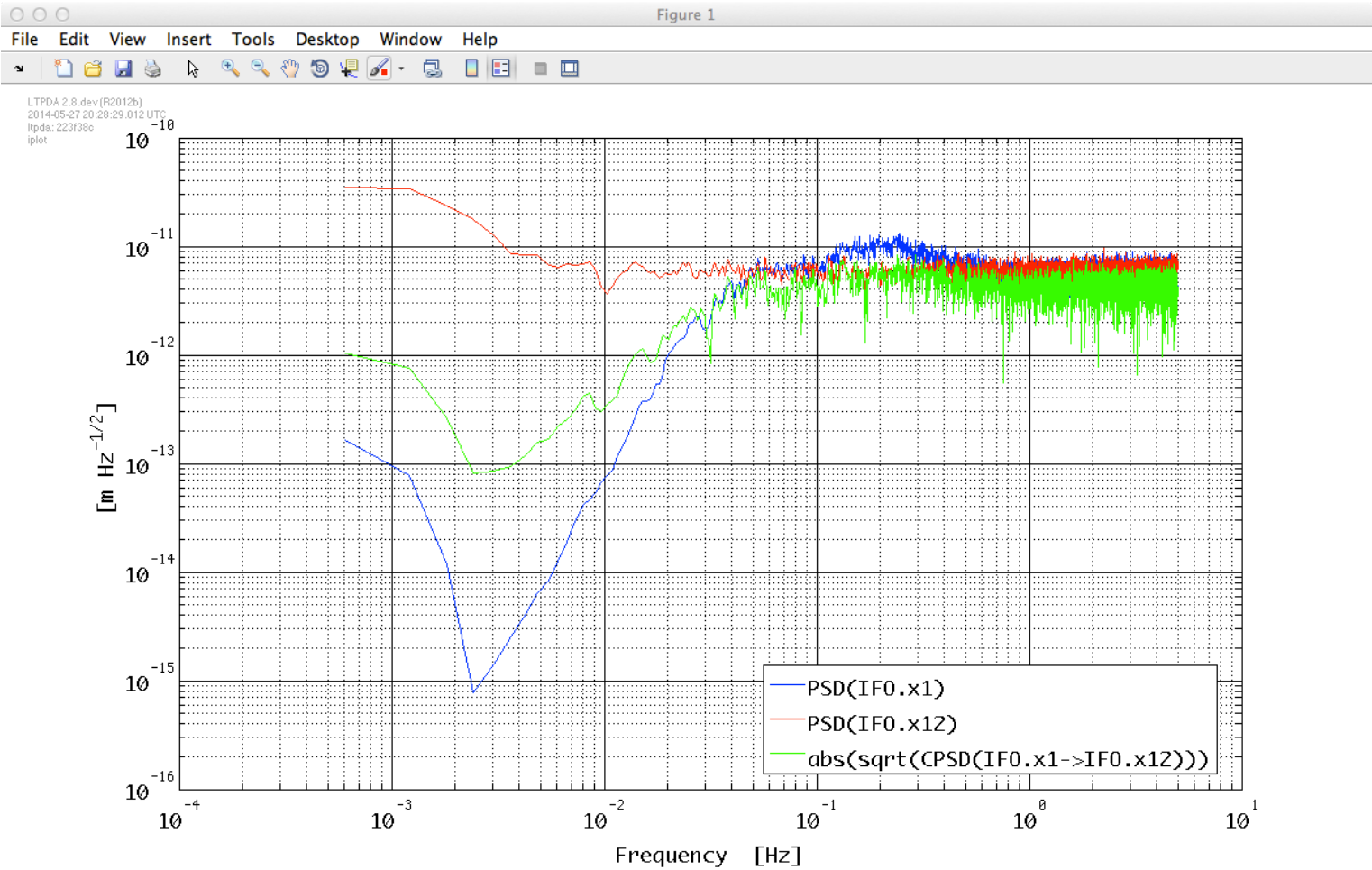
LPF noise simulations: all noise sources on

Simulating with all the noise sources on works just the same as the above examples. However, there are 95 noise inputs to the LPF model and specifying each of them would be quite painful. So, we have a shortcut for doing this. There is an `ssm`

method called `generateCovariance` which creates a suitable covariance matrix for you as well as a list of the all the noise input port names. The covariance matrix it generates is diagonal and as such all input noises are specified as uncorrelated. To use this method in conjunction with `ssm/simulate` you can follow the example. The covariance matrix is included in a `plist` object, together with other informations, and as such it must be extracted from the plist to be passed as a numeric parameter.

```matlab
% Create a covariance matrix and port list sized for our LPF model
cov_matrix = lpf.generateCovariance();

% Define the simulation plist for a 100000 sample simulation
simPlist = plist(...
            'CPSD Variable Names', cov_matrix.find('names'), ...
            'CPSD', cov_matrix.find('cov'), ...
            'Return outputs', {'IFO.x1', 'IFO.x12', 'DFACS.sc_x', 'DFACS.tm2_x'}, ...
            'Nsamples', 100000 ...
            );

% Run the simulation
out = simulate(lpf, simPlist);
```

In this case, we are asking to report at the output the reading of the `x1` IFO, as well as the forces that the DFACS controller calculates and commands to the FEEPs, so to control the spacecraft along `x`, and to the FEE connected to the EH of the TM2, so to control TM2 itself along `x`.

We can look at the time-series and spectra in the usual way:

```matlab
% Unpack the signals from the simulation output matrix
[o1, o12, F_cmd_X, F_cmd_x2] = unpack(out);

% Plot the IFO outputs and commanded forces along x
iplot(o1, o12);
iplot(F_cmd_X, F_cmd_x2);

% Estimate PSD of the measured outputs
[S_o1, S_o12, S_F_cmd_X, S_F_cmd_x2] = psd(o1, o12, F_cmd_X, F_cmd_x2, psdPlist);

% Plot results
iplot(S_o1, S_o12);
iplot(S_F_cmd_X, S_F_cmd_x2);
```

©LTP Team

Changing system parameters

The statespace models in LTPDA have two types of parameters: configuration parameters, like 'DIM', 'VERSION', which are common to all `ssm` models, and system parameters which are model specific. These *system parameters* represent symbolic entries in the underlying statespace matrices. By default, when you build a `ssm` object, the system parameters are replaced by their numerical values and the model is returned in a fully numeric state. The numerical values that were substituted are stored in the 'numparams' field of the object. This field is a `plist` where each key represents the parameter name, and the corresponding value is the parameter value.

For example, suppose we build the LTP model like this:

```
% Build the default version of the LTP model
ltp = ssm(plist('built-in', 'LTP'));
```

Then firstly you will see that the model is numeric:

```
% Check if the model is numeric
ltp.isnumerical
```

We can inspect the numerical parameters that were substituted during the building process by looking at the field 'numparams' (numerical parameters):

```
% List the numerical parameters
ltp.numparams
```

As you can see, the model has many parameters (351). A more useful way to view them is:

```
% View the 'numparams' plist in MATLAB's documentation browser
ltp.numparams.tohtml
```

Since this field is a `plist`, if you know the name of a parameter, you can search for its value like this:

```
% Get the TM1 stiffness along x
w1 = ltp.numparams.find('EOM_TM1_STIFF_XX');
```

Hint: if you don't know the name of the parameter, you can search using the `plist/regexp` method. For example:

```
% Search for all stiffness values
subsetPlist = ltp.numparams.regexp('stiffness');
```

You can even chain these searches together:

```
% Search for all stiffness values
subsetPlist = ltp.numparams.regexp('stiffness').regexp('TM1');
```

Setting system parameters at build time

You can override the default system parameters at build time by specifying the parameter name and a value in the configuration plist used for building the model. For example, let's set the stiffness of TM1 to a different value when building our LTP model:

```
% Build the default version of the LTP model
modelPlist = plist(...
                'built-in', 'LTP', ...
                'EOM_TM1_STIFF_XX', 1e-6 ...
                );
% Build the LTP model with our configuration plist
ltp = ssm(modelPlist);
```

> Note: In the current version of the LPF models, stiffness terms are specified as positive. This will likely change in future versions.

Confirm that your value for the stiffness was indeed used:

```
% Get the TM1 stiffness along x
```

```
wlUsed = ltp.numparams.find('EOM_TM1_STIFF_XX')
```

Creating symbolic models

The default is to build models completely numeric, but you can also build symbolic models. One limitation of doing this is that the model needs to be built continuous rather than discrete. This imposes further constraints on the models you are able to build, especially when it comes to the version of the DFACS model that you build because we don't have continuous versions of the DFACS for all versions.

To build the 'Standard' version of LTP continuous and with some symbolic parameters, you can do:

```
% Build the default version of the LTP model
modelPlist = plist(...
                'built-in', 'LTP', ...
                'SYMBOLIC PARAMS', {'EOM_TM1_STIFF_XX', 'EOM_TM2_STIFF_XX'}, ...
                'continuous', true ...
                );
% Build the symbolic LTP model with our configuration plist
ltp = ssm(modelPlist);
```

If you build the LTP model like this you will see, firstly, that it is not numeric, and secondly that it has a time-step of zero, i.e., it is continuous.

```
% Check if the model is numeric
ltp.isnumerical

% Check if the model is continuous
ltp.timestep
```

You can also see that the 'params' field contains two parameters, the ones you specified when constructing the model. Like the 'numparams' field, the 'params' field is also a `plist`.

```
% List the symbolic parameters
ltp.params
```

Setting system parameters at run time

Once you have a symbolic model, you can change the values of those symbolic parameters by using the `ssm/setParameters` method. For example, let's set the two stiffness values we kept symbolic in the model we built above:

```
% Set the values of the two stiffness parameters
ltp.setParameters('EOM_TM1_STIFF_XX', 1e-6, 'EOM_TM2_STIFF_XX', 3e-6);

% Check the values were set
ltp.params

% View the 'params' plist in MATLAB's documentation browser
ltp.params.tohtml
```

Note: by calling `setParameters` with no outputs, we modify our existing model. If we did give an output variable, then the original model would be copied, the parameters set on the copy, and then the copy returned, leaving the original model untouched.

Simulating with a symbolic model

Before we can simulate with this symbolic model, we have to make the model discrete. Before we can do that, we have to make the model numeric. Making the model numeric means substituting all symbolic parameters with their numeric values. To do that, we use the `ssm/subsParameters` method like this:

```
% Substitute all symbolic parameters with their numeric values
numericLTP = ltp.subsParameters;

% Confirm there are no symbolic parameters remaining
numericLTP.params
```

The model is now numeric but still continuous. We can already look at the response of the model in the frequency domain using `ssm/bode`. For example, we can look at the transfer function from differential force applied to the TMs to the differential displacement sensed by the IFO:

```
% Define a plist for bode
bodePlist = plist(...
                  'inputs', 'SIGNAL_TM12.x', ...
                  'outputs', 'DELAY_IFO.x1', ...
                  'f', logspace(-4, log10(0.5), 1000) ... % from 0.1mHz to 0.5Hz
                  );

% Compute system response
out = numericLTP.bode(bodePlist);

% Plot response
iplot(out);
```

Now we can go ahead and discretize the model using `ssm/modifyTimeStep`. On-board LPF the DFACS controllers are discretized at 10Hz, so we'll do the same here:

```
% Modify the time-step of the model to 0.1s (10Hz)
dLTP = numericLTP.modifyTimeStep(0.1);

% Confirm the model has this timestep
dLTP.timestep
```

Again, we can compute the same bode response as above and explore the effect of discretizing the model:

```
% Compute system response
dout = dLTP.bode(bodePlist);

% Get the continuous response and set its name
contResp = out.unpack;
contResp.setName('Force->displacement (Continuous)');

% Get the discrete response and set its name
discResp = dout.unpack;
discResp.setName('Force->displacement (Discrete)');

% Compare the two bode plots
iplot(contResp, discResp)
```
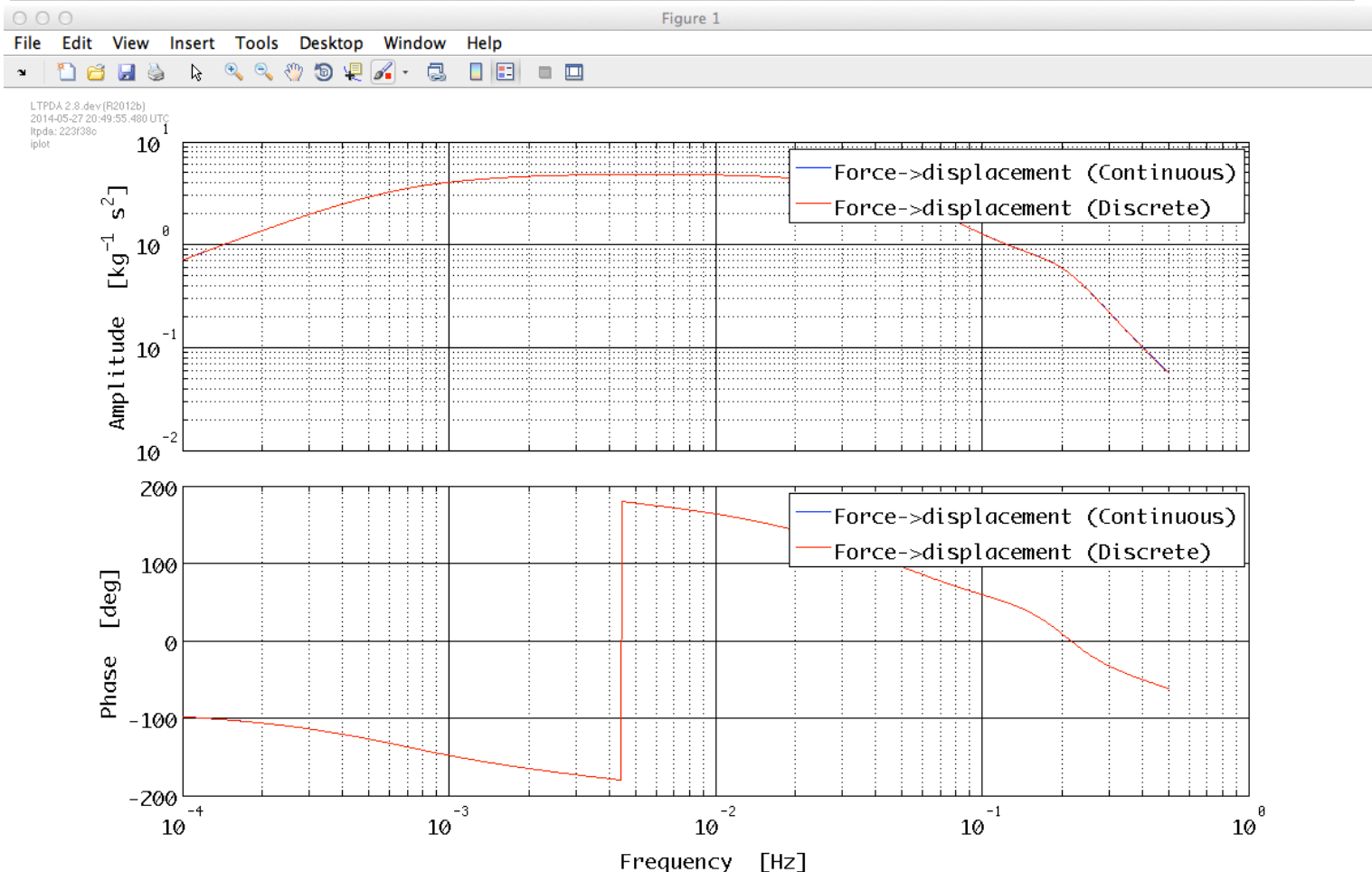
Discretizing a model is not an exact mapping, and should be used with care. It is always better, where possible, to build a discrete model from the outset. When doing that, each submodel is built discrete before assembling. Doing it this way round means that each model is built continuous, then assembled, and the final full system is discretized.

Since the model is now numerical and discrete, we can simulate the noise of the system just as we did earlier.

Simulate LPF with matched stiffness

Simulating LPF with matched stiffness involves bringing together the details of the previous section. The following code snippet builds two versions of the LPF model: one with matched stiffness, one with unmatched stiffness.

```matlab
% Configuration plist for building our model with matched stiffness
modelPlistMatched = plist(...
                          'built-in', 'LPF', ...
                          'EOM_TM1_STIFF_XX', 2e-6, ...
                          'EOM_TM2_STIFF_XX', 2e-6 ...
                          );

% Configuration plist for building our model with unmatched stiffness
modelPlistUnmatched = plist(...
                          'built-in', 'LPF', ...
                          'EOM_TM1_STIFF_XX', 1e-6, ...
                          'EOM_TM2_STIFF_XX', 5e-6 ...
                          );

% Build the LPF model with our configuration plists
lpfMatched   = ssm(modelPlistMatched);
lpfUnmatched = ssm(modelPlistUnmatched);
```

We can simulate both of these models with all noise on:

```matlab
% Create a covariance matrix and port list sized for our LPF model
cov_matrix = lpfMatched.generateCovariance();

% Define the simulation plist for a 100,000s simulation
simPlist = plist(...
                 'CPSD Variable Names', cov_matrix.find('names'), ...
                 'CPSD', cov_matrix.find('cov'), ...
                 'Return outputs', {'IFO.x1', 'IFO.x12', 'DFACS.sc_x', 'DFACS.tm2_x'}, ...
                 'Nsamples', 1000000 ...
                 );

% Run the simulations
outMatched   = simulate(lpfMatched, simPlist);
outUnmatched = simulate(lpfUnmatched, simPlist);
```

Now we have two simulated data sets. We can look at the effect of matching the stiffness by looking at the coherence of the force commanded on the spacecraft with the differential displacement of the TMs measured by the IFO:

```matlab
% Compare the coherence between force on SC and the diff. displacement

% Unpack the outputs from the simulations
[o1m, o12m, F_cmd_Xm, F_cmd_x2m] = unpack(outMatched);
[o1u, o12u, F_cmd_Xu, F_cmd_x2u] = unpack(outUnmatched);

% Compute the coherence
coherePlist = plist(...
                    'navs', 16, ...
                    'order', 1 ...
                    );

Cm = cohere(F_cmd_Xm, o12m, coherePlist);
Cm.setName('SC Jitter to X12 coherence (matched stiffness)');

Cu = cohere(F_cmd_Xu, o12u, coherePlist);
Cu.setName('SC Jitter to X12 coherence (unmatched stiffness)');

% Bin the data to do some frequency-domain averaging
Cm_binned = Cm.bin_data;
Cu_binned = Cu.bin_data;

% Plot the results with error bars
iplot(Cm_binned, Cu_binned)
```

You should see a plot like the following, showing a clear decrease in coherence in the matched stiffness case:

©LTP Team
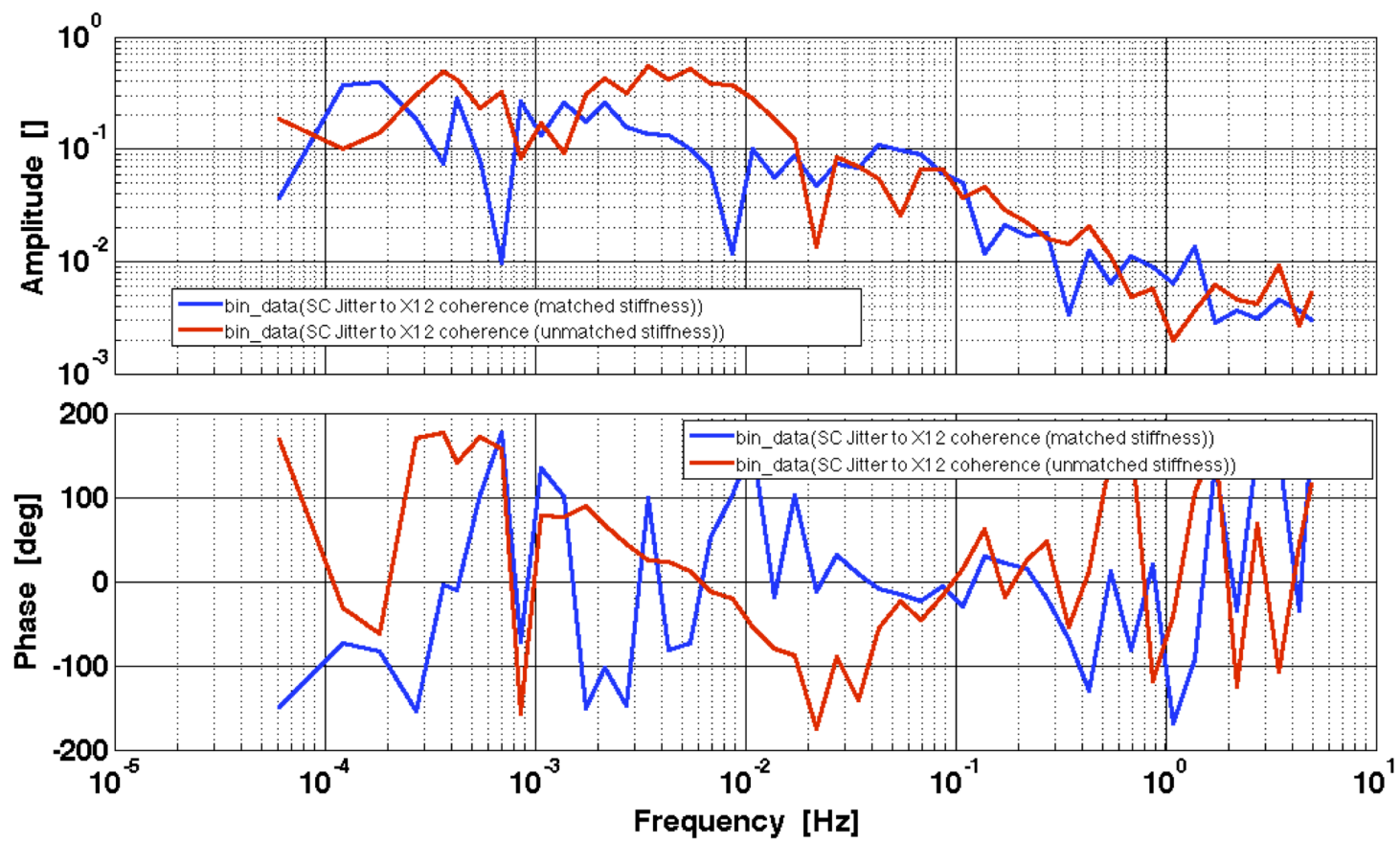
# Topic 3 – Estimation of equivalent acceleration.

Topic 3 will introduce the principles involved in estimating the equivalent acceleration of the two test-masses from the observed relative positions of the three bodies. We will introduce the tools in LTPDA that can be used to estimate the equivalent residual differential acceleration of the test-masses as well as the equivalent residual acceleration of the SC. These tools can then be used to estimate the accelerations from the simulated data produced in the previous topic.

1. Principles and theory
2. Tools for estimating the equivalent acceleration in LTPDA
3. Estimate equivalent accelerations from simulation data

©LTP Team

Topic 3 - Estimation of equivalent acceleration. (LTPDA Toolbox)

Principles and theory

LTP along the principal measurement axis (x) is represented schematically by the diagram reported in figure below.
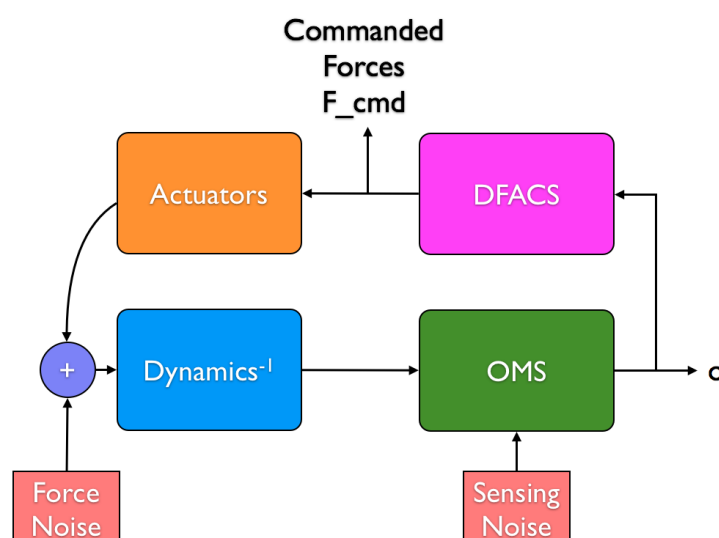
Schematic representation of LTP along the x axis.

The available signal is the displacement at the output of the optical metrology system (OMS). In order to extract the acceleration noise from the displacement signal we have to process the data accordingly to the scheme reported in figure below.
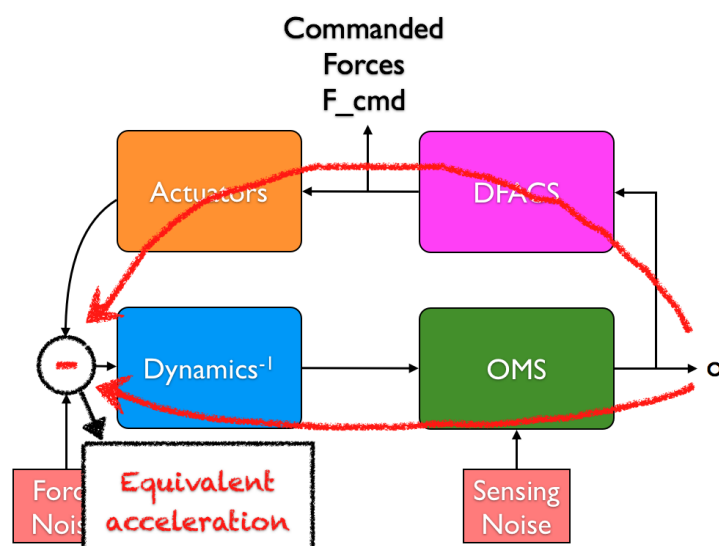
Schematic representation of the conversion to acceleration process.

In case the commanded forces are available, the scheme for the conversion to acceleration is reported in the diagram.
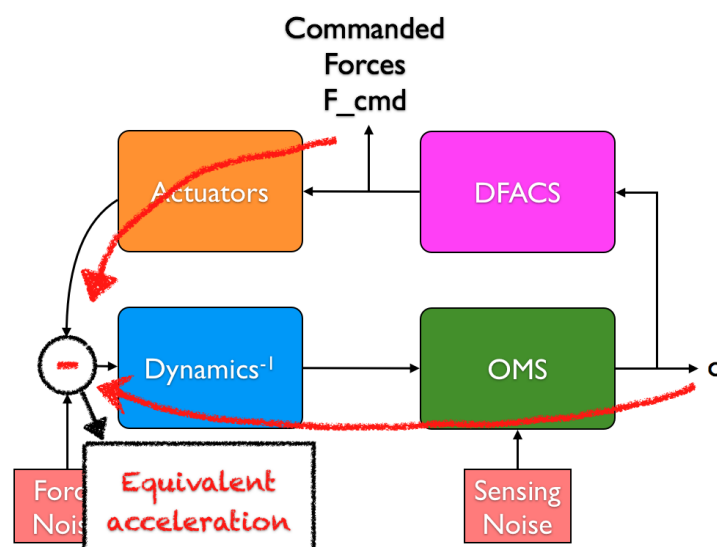
Schematic representation of the conversion to acceleration process when commended forces are available.

Since displacement data contains interferometer sensing noise, the reconstructed acceleration is the combination of the proper acceleration noise and a contribution from the sensing noise (shaped by the inverse of the dynamics). For such reason the reconstructed signal is indicated with the term equivalent acceleration.

LPF can be considered as a three body controlled system composed of the two TMs and the spacecraft (SC). Its equations of motion along the measurement axis can be written as:

$$m_1 \ddot{x}_1 + m_1 \ddot{x}_{sc} + m_1 \omega_1^2 x_1 = f_1$$
$$m_2 \ddot{x}_2 + m_2 \ddot{x}_{sc} + m_2 \omega_2^2 x_2 = f_2 + f_{c2}$$
$$m_{sc} \ddot{x}_{sc} - m_1 \omega_1^2 x_1 - m_2 \omega_2^2 x_2 = f_{sc} - f_{c2} + f_{csc}.$$

In order to avoid confusion, TM1 is the free-fall reference and TM2 is the actuated TM.

Moving to the Laplace domain, substituting for the spacecraft dynamics and substituting for the differential coordinate, the equation of motion can be rewritten as:

$$s^2 x_1 \omega_1^2 \left(1 + \frac{m_1}{m_{sc}}\right) x_1 + \omega_2^2 \frac{m_2}{m_{sc}} x_1 + \omega_2^2 \frac{m_2}{m_{sc}} x_\Delta =$$
$$= \frac{f_1}{m_1} - \frac{f_{sc}}{m_{sc}} + \frac{1}{m_{sc}} H_2 o_\Delta - \frac{1}{m_{sc}} H_{sc} o_1$$
$$s^2 x_\Delta + \left(\omega_2^2 - \omega_1^2\right) x_1 + \omega_2^2 x_\Delta =$$
$$= \frac{f_2}{m_2} - \frac{f_1}{m_1} + \frac{1}{m_2} H_2 o_\Delta.$$

Introducing matrix notation.

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_\Delta \end{pmatrix}$$

$$\mathbf{o} = \begin{pmatrix} o_1 \\ o_\Delta \end{pmatrix}$$

$$\mathbf{f} = \begin{pmatrix} f_1 \\ f_2 \\ f_{sc} \end{pmatrix}$$

$$\mathbf{D} = \begin{pmatrix} s^2 + \omega_1^2 + \frac{m_1}{m_{sc}}\omega_1^2 + \frac{m_2}{m_{sc}}\omega_2^2 & \frac{m_2}{m_{sc}}\omega_2^2 \\ \omega_2^2 - \omega_1^2 & s^2 + \omega_2^2 \end{pmatrix}$$

$$\mathbf{G} = \begin{pmatrix} \frac{1}{m_1} & 0 & -\frac{1}{m_{sc}} \\ -\frac{1}{m_1} & \frac{1}{m_2} & 0 \end{pmatrix}$$

$$\mathbf{C} = \begin{pmatrix} -\frac{1}{m_{sc}} & \frac{1}{m_{sc}} \\ 0 & \frac{1}{m_2} \end{pmatrix}$$

$$\mathbf{H} = \begin{pmatrix} H_{sc} & 0 \\ 0 & H_2 \end{pmatrix}$$

The equation of motion can be rewritten:

$$\mathbf{D} \cdot \mathbf{x} = \mathbf{G} \cdot \mathbf{f} + \mathbf{C} \cdot \mathbf{H} \cdot \mathbf{o}$$

The output displacement o corresponds to the measurement of x provided by the sensing system:

$$\mathbf{o} = \mathbf{S} \cdot \mathbf{x} + \mathbf{o_{rn}}$$

The dynamics can be now rewritten in terms of the known output o:

$$\mathbf{D} \cdot \mathbf{S}^{-1} \cdot \mathbf{o} - \mathbf{C} \cdot \mathbf{H} \cdot \mathbf{o} = \mathbf{G} \cdot \mathbf{f} + \mathbf{D} \cdot \mathbf{S}^{-1} \cdot \mathbf{o_{rn}}$$

The quantity on the right-hand side represents the target force-per-unit-mass acting on the test masses combined with the sensing noise. This is the only quantity that can be estimated from the displacement output by means of the left-hand side of the equation above. The tool described in the following section performs the calculation reported in the left-hand side of the equation.

Notation table.

Symbol	Description
x_1 and x_2	TMs coordinates along the sensitive axis. They are relative coordinates in the spacecraft reference frame.
x_{sc}	Absolute spacecraft coordinate along the sensitive axis.
m_1, m_2 and m_{sc}	Masses of the two TMs and of the spacecraft.
ω_1^2 and ω_2^2	Stiffnesses coupling the TMs and the spacecraft.
f_1, f_2 and f_{sc}	Forces acting on TMs and the spacecraft respectively.
f_{c2} and f_{csc}	Control forces on the second TM and the spacecraft respectively.

o_1 and o_Δ	Output displacement signals as provided by the interferometer readout system.
H_2 and H_{sc}	Transfer functions of the control systems on TM2 and the spacecraft. The force applied by the controllers is calculated on the basis of the output displacement, therefore $f_{c2} = H_2 o_\Delta$ and $f_{csc} = H_{sc} o_1$.

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Tools for estimating the equivalent acceleration in LTPDA

In previous page we have discovered as the equivalent acceleration can be in principle estimated from the interferometer displacement signal. LTPDA provides a dedicated tool for the purpose. The tool name is 'ltp_ifo2acc', it is a method of the class 'ao' and it is provided with the external module 'LPF_DA_Module'. If the module is correctly installed 'ltp_ifo2acc' help can be obtained with the command

```
    help ao/ltp_ifo2acc
```

```
 ltp_ifo2acc convert ifo data to acceleration
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 DESCRIPTION: ltp_ifo2acc converts interferometer data to acceleration

 CALL:                   bs = ltp_ifo2acc(o1,od,pl);
                         [a1,ad] = ltp_ifo2acc(o1,od,pl);


 INPUTS:
       - o1, channel 1 interferometer output
       - od, differential channel interferometer output
       - pl, plist containing parameters

 OUTPUTS:
       - a1, Force per unit of mass on TM1 in the spacecraft reference
       frame
       - ad, Differential force per unit of mass between TMs

 Parameters Description

 REFERENCES:


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

With a click on the 'Parameters Description' link, we obtain a list of method's parameters together with a description.

In this training session we will concentrate only on 7 parameters of the full list:

LSS v4.9.2			
Key	**Default Value**	**Options**	**Description**
GDF	1	*none*	thrusters actuation gain
GSUS	1	*none*	suspension actuation gain
W1	-1.3e-06	*none*	The parasitic stiffness per unit of mass on TM1 (s^-2)
W2	-	*none*	The parasitic stiffness per unit of

	1.9999999999999999e-06		mass on TM2 (s^-2)
SD1	0.0001	*none*	Cross-talk channel 1 to diff channel
OMS DELAY O1	0.4000000000000002	*none*	OMS processing delay on channel 1 [s]
OMS DELAY OD	0.4000000000000002	*none*	OMS processing delay on differential channel [s]
HDF	-1	*none*	Drag-free controller model. You can choose between 3 options: • -1, Uses the default builtin model (). • 0, calculation with Hdf will be skipped, acom1 = 0. • Input model, input your own model.It can be a miir, mfir, pzmodel, smodel or an ao with the commanded force
HSUS	-1	*none*	Electrostatic suspension controller model. You can choose between 3 options: • -1, Uses the default builtin model (). • 0, calculation with Hsus will be skipped, acomD = 0. • Input model, input your own model.It can be a miir, mfir, pzmodel, smodel or an ao with the commanded force

In particular, parameters 'HDF' and 'HSUS' can be used to input the commanded forces to the method. The method can also calculate automatically the expected commanded forces on the basis of a built-in model for the controllers. Unfortunately, since such a model has a pole at zero, the resulting output is affected by huge transients and is not completely representative of the 'true' commanded forces provided by the control system (one should know the initial state of the DFACS to perform the correct calculation). For this reason it is always advisable to make use of the true commanded forces if available. OMS DELAY O1 and OMS DELAY OD should instead be set at 0.3 s in order to match current 'ssm' default values.

Here is a table matching current parameters names with method parameters names

Current name	Method name
FEEPS_XX	GDF
CAPACT_TM2_XX	GSUS
IFO_X12X1	SD1
-1 * EOM_TM1_STIFF_XX	W1

| −1 * EOM_TM2_STIFF_XX | W2 |

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Estimate equivalent acceleration from simulation data

This exercise starts with the generation of simulated noise data.

```matlab
% set output names
outNames = {...
  'DELAY_IFO.x1'   ...      % o1 IFO
  'DELAY_IFO.x12'  ...      % o12 IFO
  'DFACS.sc_x', ...         % Commanded force F_cmd_X
  'DFACS.tm2_x' ...         % Commanded force F_cmd_x2
  };

% Define parameters and nominal values
params = {...
  'FEEPS_XX', ...           % FEEPs actuation gain
  'CAPACT_TM2_XX', ...      % Capacitive actuation gain
  'IFO_X12X1', ...          % The coupling of the x position of TM1 to the estimated x position of TM2
  'EOM_TM1_STIFF_XX', ...   % Total stiffness of TM1 along X
  'EOM_TM2_STIFF_XX' ...    % Total stiffness of TM2 along X
  };

% set parameters values for the conversion to acceleration
FEEPS_XX        = 1;
CAPACT_TM2_XX   = 1;
IFO_X12X1       = 1e-4;
EOM_TM1_STIFF_XX = 1.3e-6;
EOM_TM2_STIFF_XX = 2.0e-6;

% actual values for the parameters
values = [...
  FEEPS_XX, ...
  CAPACT_TM2_XX, ...
  IFO_X12X1, ...
  EOM_TM1_STIFF_XX, ...
  EOM_TM2_STIFF_XX];

% Create LPF SSM model
pl = plist('built-in', 'LPF', ...
  'DIM', 1, ...                       % We use a one dimensional model
  'CONTINUOUS', false, ...            % The model is discrete
  'param names', params, ...          % Parameter names
  'param values', values, ...         % Parameter values
  'VERSION', 'Best Case June 2011'); % Model version

LPF = ssm(pl);

% Create default noise covariance, assumes independent noise sources
cov = LPF.generateCovariance;

% Create a plist to configure the simulation
plsim = plist(...
          'return outputs', outNames,...
          'cpsd variable names', cov.find('names'), ...
          'cpsd', cov.find('cov'), ...
          'Nsamples', 1e6);

% Run the simulation
out = LPF.simulate(plsim);

% Unpack the signals from the simulation output matrix
[o1, o12, F_cmd_X, F_cmd_x2] = unpack(out);
```

Since the method for the conversion to acceleration uses a different naming convention for the parameters, it is convenient to set the convention mapping for the parameters.

```matlab
Gdf  = FEEPS_XX;
Gsus = CAPACT_TM2_XX;
SD1  = IFO_X12X1;
w1   = -1*EOM_TM1_STIFF_XX;
w2   = -1*EOM_TM2_STIFF_XX;
```

Note that in the method for the conversion to acceleration the stiffness is negative!

'ltp_ifo2acc' likes to work with SI units and in particular the force units have to be expressed in 'kg m s^(-2)'.

```matlab
F_cmd_X.yunits.toSI;
F_cmd_x2.yunits.toSI;
```

Now we can set the parameter list for the conversion to acceleration.

```matlab
pli2a = plist(...
    'SD1', SD1,...
    'Gdf', Gdf,...
    'Gsus', Gsus,...
```

```
        'w1', w1,...
        'w2', w2,...
        'OMS delay o1', 0.3, ...
        'OMS delay oD', 0.3, ...
        'Hdf', F_cmd_X*(-1),...
        'Hsus', F_cmd_x2*(-1)...
      );
```

Note the −1 multiplying the commanded forces, this is need because of a difference of convention between 'ssm' and 'ltp_ifo2acc' on the controllers outputs.

Now the conversion to equivalent residual acceleration of the interferometer displacement signals.

```
    M = ltp_ifo2acc(o1, o12, pli2a); % M is a matrix object containing a1 and a12
```

Calculate the square root of the power spetral density in order to check the resuls.

```
    plpsd = plist(...
      'order', 1, ...
      'SCALE', 'ASD' ...
      );

    Mxx  = M.lpsd(plpsd);
```

Remove the first 6 samples (lowest frequencies bins) since they are corrupted by the window function.

```
    plsp = plist(...
      'samples', [7 inf] ...
      );

    Mxxs  = Mxx.split(plsp);
```

Finally, plot the result.

```
    plplot = plist(...
    'Linecolors', {'k', 'r'}, ...
    'LineStyles', {'--', '-'}, ...
    'LineWidths', {3, 3}, ...
    'Legends', {'a_1', 'a_{\Delta}'}, ...
    'XLABELS', {'All', 'Frequency'}, ...
    'YLABELS', {'All', 'PSD^{1/2}'} ...
      );

  iplot(Mxxs, plplot)
```

©LTP Team

LTPDA Toolbox™   [contents](#)   ◄ ►
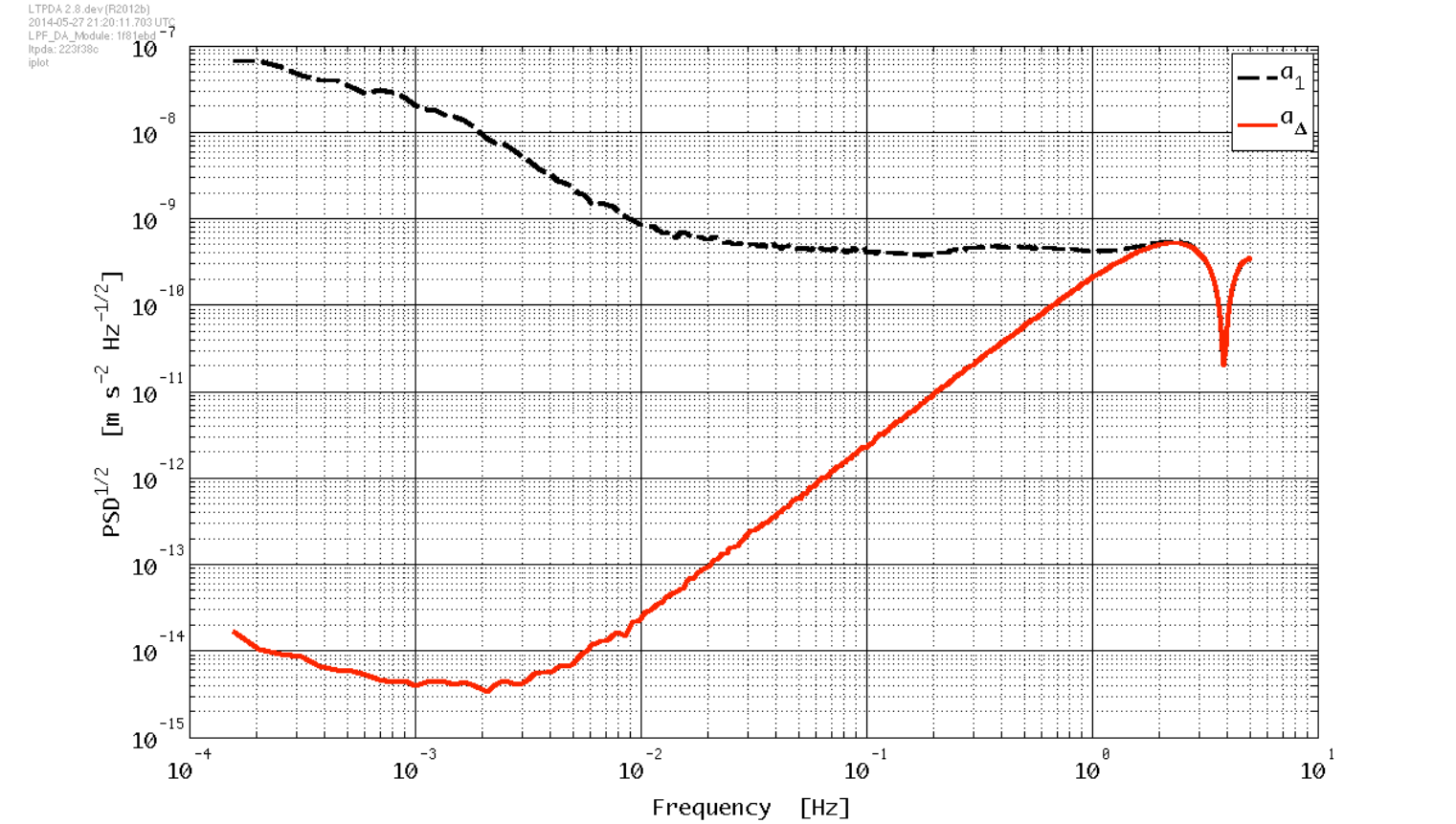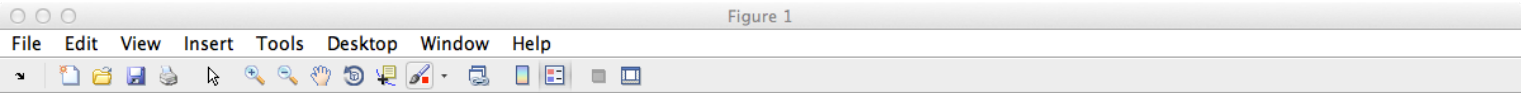
Topic 4 – Simulating LPF with injected signals.

Topic 4 introduces the techniques needed to inject signals into the LPF simulations. We will review the available inputs of the LPF model and how to inject signals during the simulation.

1. LPF model inputs
2. Building signals
3. How to inject signals
4. Simulate LTP with injected signals (no noise)
5. Inject noise signals to LTP
6. Estimate transfer functions from simulated signals, compare with Bode estimates
7. Simulate LPF with injected signals

◄ Estimate equivalent acceleration from simulation data   LPF model inputs ►

©LTP Team

LPF model inputs

The state-space model of LPF in LTPDA is extensive and has a total of 294 input ports spread over 32 input blocks. The majority of these inputs are intended as noise inputs, but some are inputs to the control loops of the system.

Guidance inputs

If you inspect the model using `viewDetails` you will see an input block called `GUIDANCE` which has 21 input ports, one for each possible sensor input. These inputs act as set points for the different control loops. For example, injecting a 1nm, 1mHz sine-wave signal on `GUIDANCE.ifo_x1` whilst in science mode 1 will cause the SC to oscillate around TM1 by 1nm (or very close to). This is because at these frequencies the loop which controls the relative x postion of TM1 and the SC has very high gain.

The following figure shows schematically the guidance input for the LPF closed-loop system:
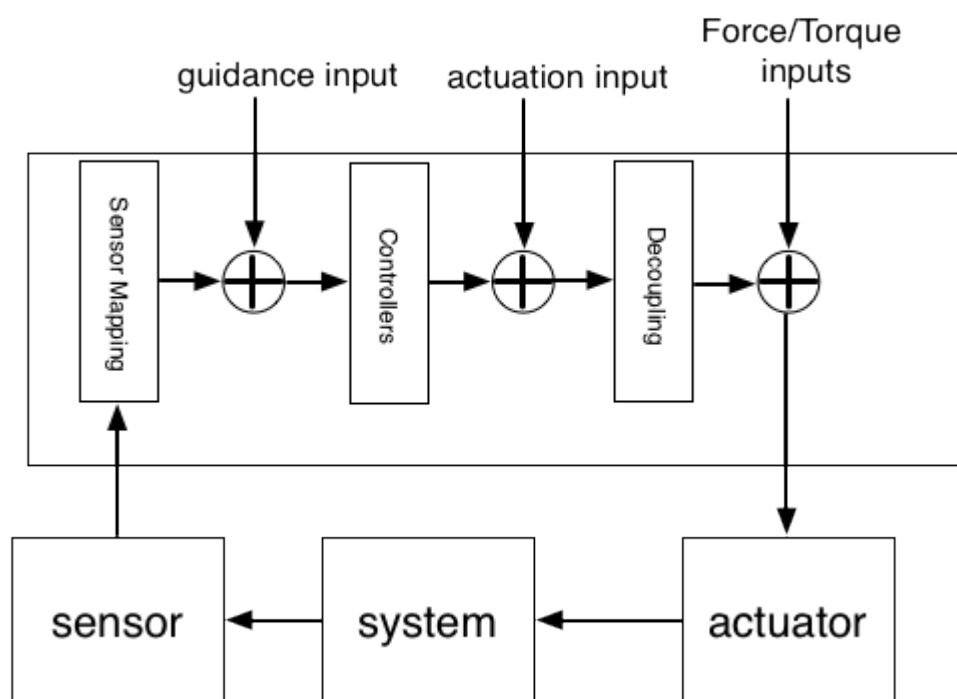
> Note: In the current version of the LPF models, the guidance signals are added before the sensor mapping matrix. For all practical purposes, this makes little difference, but this will be corrected in a future version.

Actuation inputs

The LTP (or LPF) model also has a set of inputs which we term 'Actuation' inputs. These are commanded forces and torques per unit mass or inertia, and are added to those commands coming from DFACS before the decoupling matrix is applied.

The figure above also shows the actuation inputs in relation to the other inputs discussed here.

Force/Torque inputs

After the decoupling matrix, there is another input where we can inject forces and torques per degree-of-freedom on the different actuators. These force/torque inputs are also shown on the figure above.

◀ Topic 4 – Simulating LPF with injected signals.	Building signals ▶

©LTP Team

Building signals

Introduction

Building signals in LTPDA can be done in various ways. Perhaps you already have data in a text file, or you have a formula for a particular time-series. These can all be accommodated by the AO constructor.

Another useful way to build template signals is by using so-called 'built-in models'. These built-in models serve as kind-of user-defined constructors. They typicall output objects of a particular type, with the exact details of the object being governed by a few configuration parameters. For example, you've already used a built-in model to build a particular kind of ssm object -- the LPF model. We also have built-in models for some of the signal templates indended for use in some of the LPF system identification experiments.

If you launch the LTPDA model browser, you can browse the various built-in models available to LTPDA on your system. To launch the browser, do:

```
>> LTPDAModelBrowser
```

You can also request a textual list of the available models for a particular LTPDA user class by doing, for example,

```
>> ao.getBuiltInModels
```

Building signals from AO built-in model

The battery of signals prescribed in S2-UTN-TN-3045 for the first system identification experiment can be built using the following command:

```
>> sig = ao(plist('built-in', 'signals_3045_1_1'));
```

If you plot this AO you should see a plot like this:
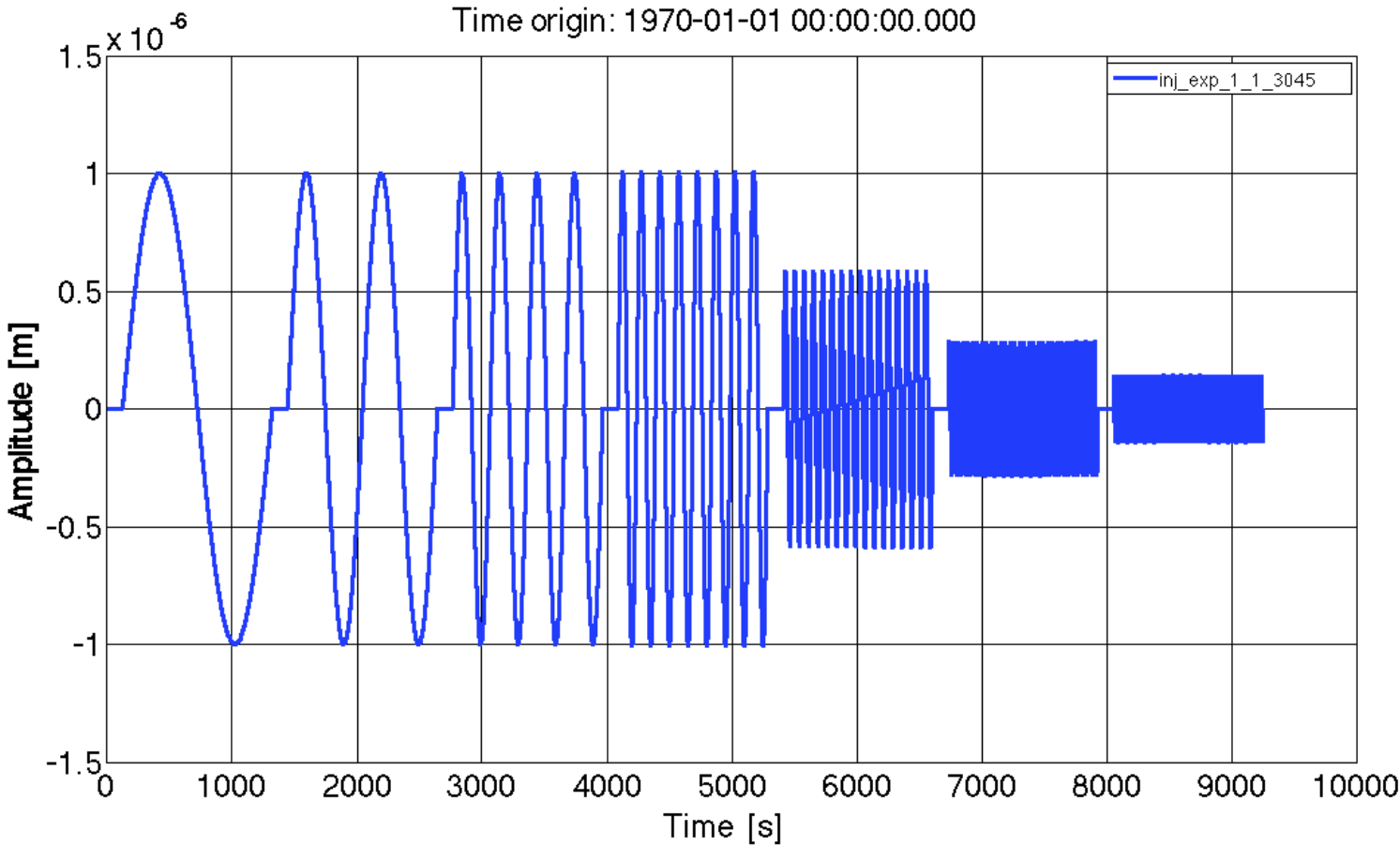
There are a number of ao built-in models included in the LPF_DA_Module which reproduce the signals for the TN-3045 experiments. All 6 of these models have a single configurable parameter. To see the help/description for this model, either use the model browser, or from the terminal do:

```
>> help ao_model_signals_3045_1_1
```

This pattern is always the same in LTPDA, to get help for any built-in model, you need to know its class, then just do:

```
>> help [class]_model_[model name]
```

The help usually includes a short description of the model and a link called "Model Information". If you click on that you will see a window which gives more information about the model and the parameters you can configure at build-time. For example, for the model we built above, you can also specify the gap you want to have between each of the 7 finite duration sine-waves. The duration of the sine-waves is specified by the experiment and so is not a parameter. To change the gap between all signals to 1000s, build the model like this:

```
>> sig = ao(plist('built-in', 'signals_3045_1_1', 'gaps', 1000));
```

©LTP Team

How to inject signals

Injecting signals in to `ssm` models is done at the time of simulating. In Topic 2 you learned how to simulate an LPF model which generated its noise internally. But you can also inject time-series data into the simulation in the form of AOs.

The two key parameters for `ssm/simulate` are:

Key	Description
AOS	An array of AOs to be input to the simulation. Specify one per "AOS VARIABLE NAMES".
AOS VARIABLE NAMES	An cell-array of strings which give the names of the input ports you want to inject the AOs into.

The input ports should be specified using the "[block].[port]" notation. For example, to inject a guidance signal on the x1 control coordinate of our LPF model, you would do:

```
        % The following supposes you already have an LPF SSM model called 'lpf' in your
MATLAB workspace.

    % Create some time-series analysis object to inject
    aSignal = ao(plist('tsfcn', '1e-6*sin(2*pi*0.1*t)', 'fs', 10, 'nsecs', 1000));

    % Create the plist to configure simulate
    sim_pl = plist('AOS', aSignal, 'AOS Variable Names', 'GUIDANCE.ifo_x1', 'return
outputs', 'DELAY_IFO.x1')

    % Run the simulation
    out = simulate(lpf, sim_pl);
```

To inject multiple signals into different input ports, you would do something like:

```
        % The following supposes you already have an LPF SSM model called 'lpf' in your MATLAB
workspace.

    % Create some time-series analysis object to inject
    fs    = 1/lpf.timestep; % Get the sample rate from the model's timestep
    nsecs = 1000; % Create signals 1000s long
    sig1 = ao(plist('tsfcn', '1e-6*sin(2*pi*0.1*t)', 'fs', fs, 'nsecs', nsecs));
    sig2 = ao(plist('built-in', 'signals_3045_1_1', 'gaps', 1000));

    % Create the plist to configure simulate
    sim_pl = plist('AOS', [sig1 sig2], ...
                   'AOS Variable Names', {'GUIDANCE.ifo_x1', 'GUIDANCE.ifo_x12'}, ...
                   'return outputs', {'DELAY_IFO.x1', 'DELAY_IFO.x12'})

    % Run the simulation
    out = simulate(lpf, sim_pl);
```

There are a number of rules which need to be followed for this all to work:

1. The order of the AOs must match the order of the input port names.
2. The time-series AOs must all have the same sample rate and that sample rate must match the time-step of the discretized `ssm` model.
3. The length of the simulation will be governed by the shortest input AO. Specifying the number of samples with the 'NSamples' key for `ssm/simulate` will be ignored when input AOs are given.

How do I know which ports are available?

One of the difficulties of working with very large models like the LPF model, is that it's sometimes difficult to see which inputs and outputs are available to you. To aid in this process, we have two tools which can be useful. The first presents details of a given model in a documentation window. For example, suppose we have our LPF model in the MATLAB workspace, then you can do

```
lpf.viewDetails
```

which will present a window like this one:
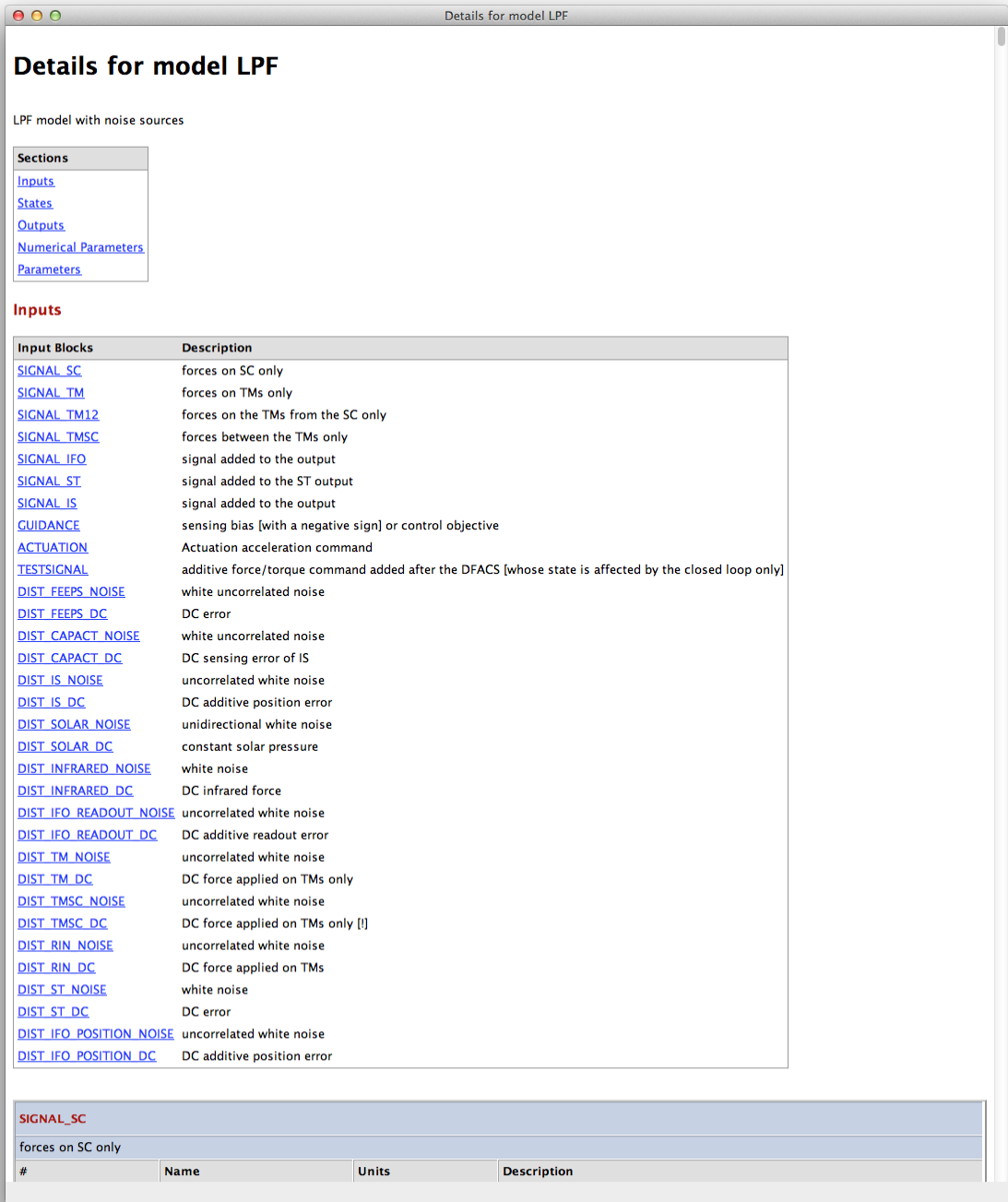
You can also use the `ssm` method `getPortNamesForBlocks`. This will return an cell-array of names for a given block. You can also configure what kind of blocks (inputs, outputs, or states) to return results for. For example, suppose we want to get a list of all guidance input ports, we could do

```
portNames = lpf.getPortNamesForBlocks('GUIDANCE');
```

This returns only input ports because the LPF model we are using only has one block called 'GUIDANCE' and it is an input block. This need not necessarily be true, and there could be an input block with the same name as an output block. In that case, you would need to specify which block types to search like this:

```
lpf.getPortNamesForBlocks(plist('blocks', 'guidance', 'type', 'inputs'))
```

The list returned from this method can be directly used as input to `ssm/simulate`.

◀ Building signals   Simulate LTP with injected signals (no noise) ▶

Simulate LTP with injected signals (no noise)

In Topic 2 you learned how to create an LTP model. Recall, the LTP model is the same as the LPF model but without all the noise-shape filters assembled. As a reminder, you can build this model like this:

```
ltp = ssm(plist('built-in', 'LTP', 'Version', 'Standard'))
```

Inject a signal into guidance of the LTP model

In the previous section we saw how to inject signals in to an `ssm` model. Here we will inject a signal into the LTP model. In particular, we'll modulate the set-point of the drag-free loop along x by injecting a sine-wave in the guidance input for that control coordinate. We'll ask the simulator for the two main IFO outputs: the position of SC relative to TM1 and the position of TM2 relative to TM1.

```
% Create some time-series analysis object to inject
aSignal = ao(plist('tsfcn', '1e-6*sin(2*pi*0.01*t)', 'fs', 10, 'nsecs', 1000));

% Create the plist to configure simulate
sim_pl = plist('AOS', aSignal, 'AOS Variable Names', 'GUIDANCE.ifo_x1', ...
               'return outputs', {'DELAY_IFO.x1', 'DELAY_IFO.x12'})

% Run the simulation
out = simulate(ltp, sim_pl);
```
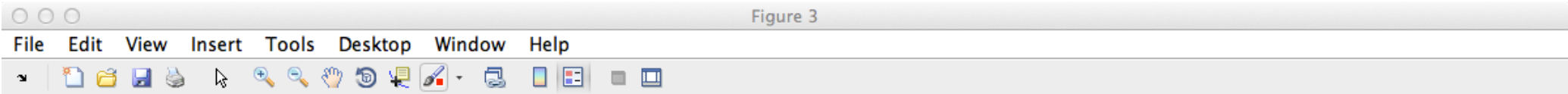
You can then plot the output signals by doing either of the following:

```
% Create a plist to configure iplot to plot with subplots
plot_pl = plist('arrangement', 'subplots');

% Use matrix/iplot to plot the internal objects
iplot(out, plot_pl)

% First extract the internal AOs and then plot
[o1, o12] = unpack(out);
iplot(o1, o12, plot_pl);
```
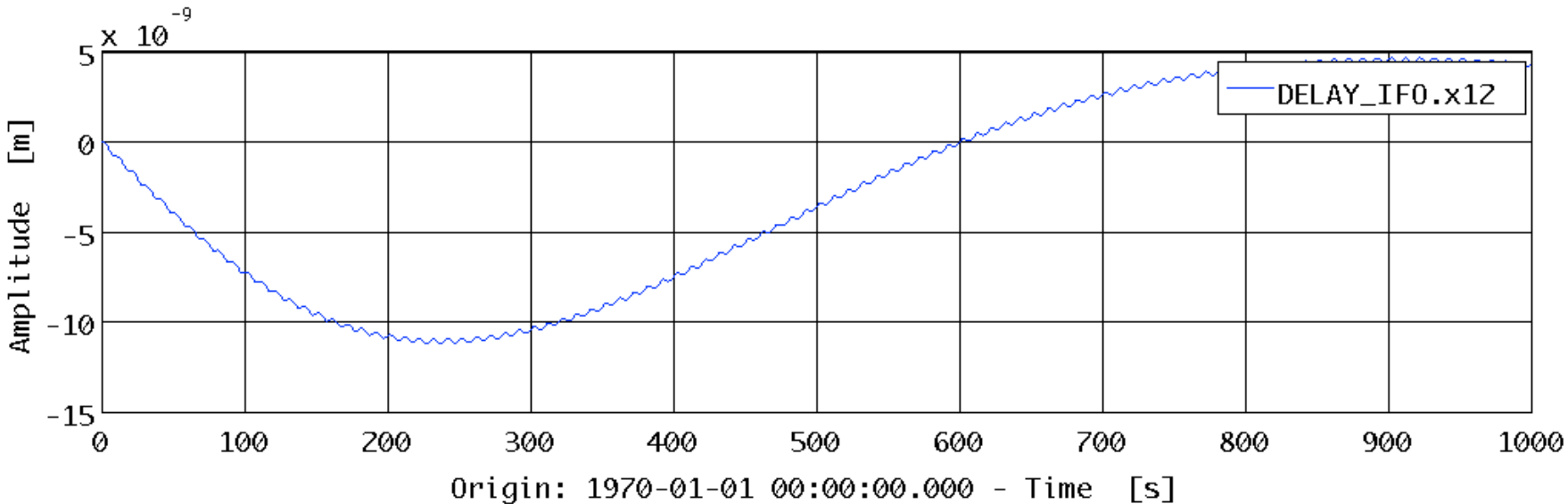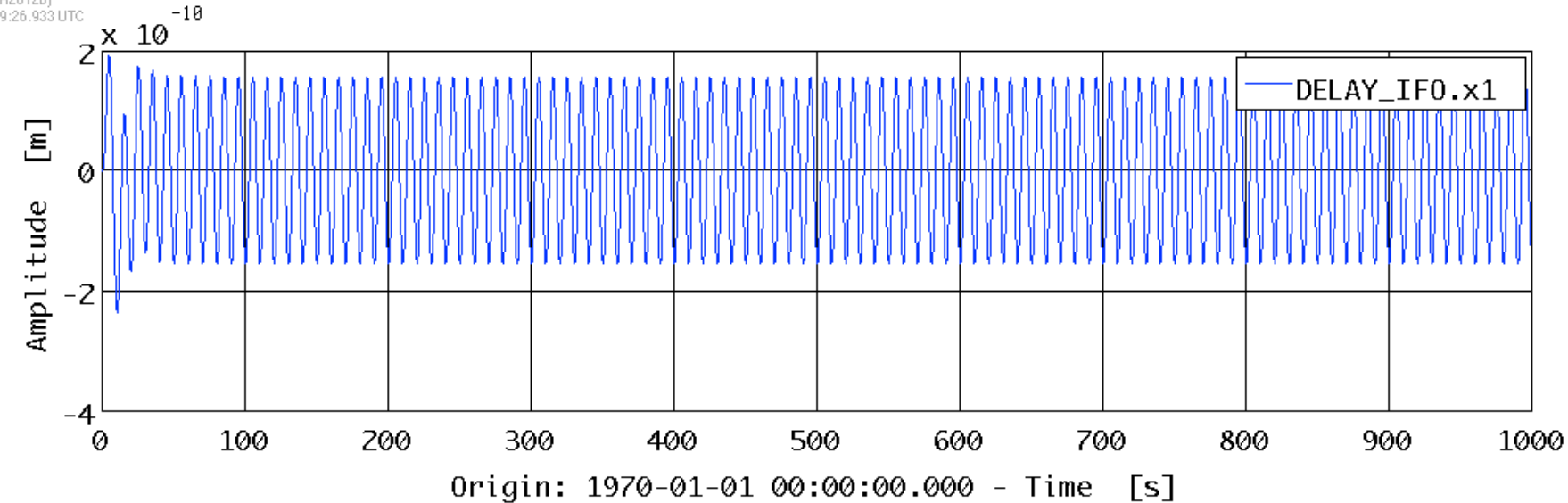
You should then see a plot something like:
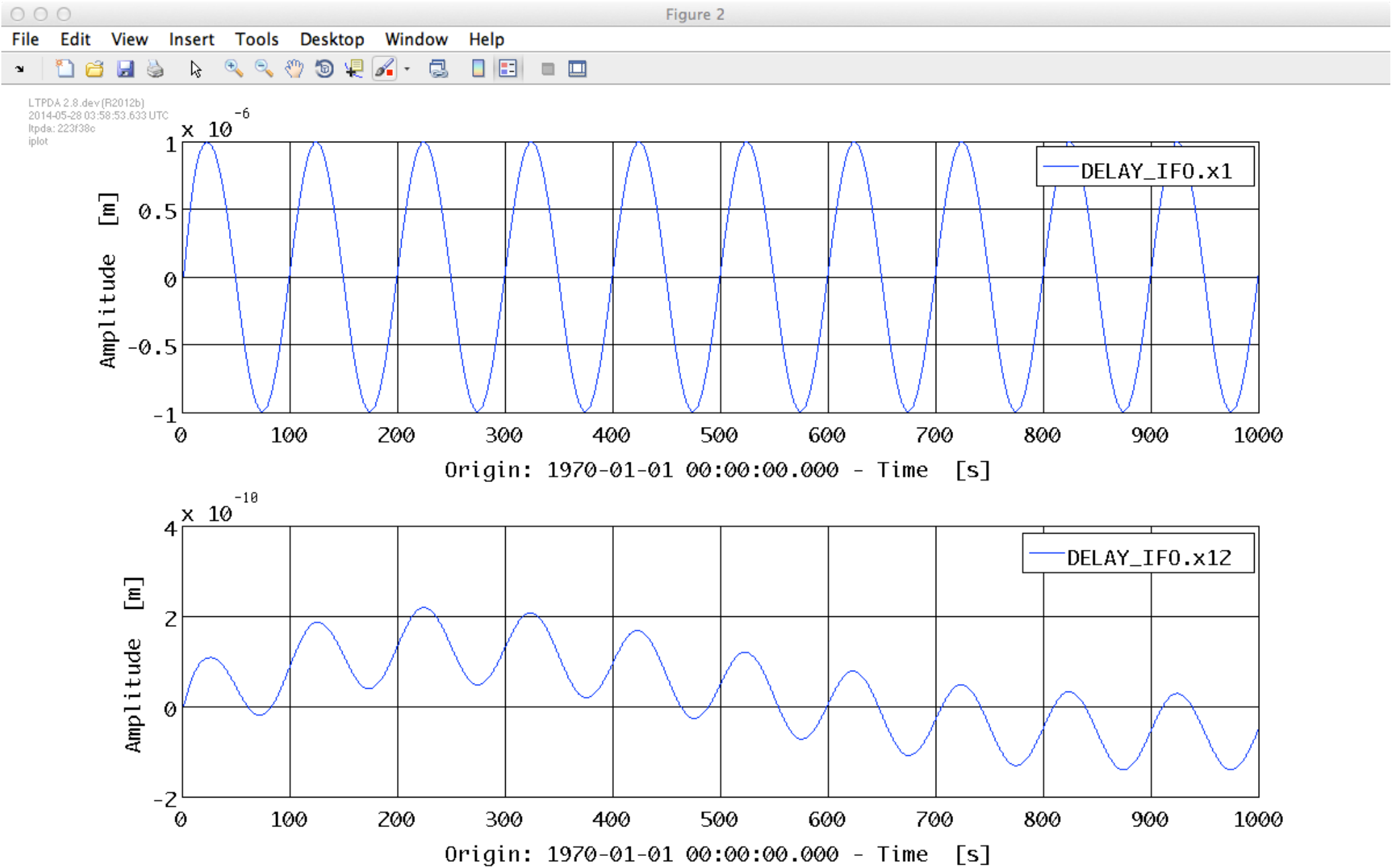
Command a force on TM1 of the LTP model

This time we'll apply a modulated force on TM1. We'll ask the simulator for the all IFO outputs and all IS output. These force inputs are called 'TESTSIGNAL' in the current models. We'll create a sine-wave signal of 0.1nN at 10 mHz and apply it to TM1 along x:

```
% Create some time-series analysis object to inject
aSignal = ao(plist('tsfcn', '1e-10*sin(2*pi*0.1*t)', 'yunits', 'N', 'fs', 10, 'nsecs', 1000));

% Generate a list of outputs we want from the simulator
outputs = ltp.getPortNamesForBlocks(plist('blocks', {'DELAY_IFO', 'IS'}, 'type', 'outputs'));

% Create the plist to configure simulate
sim_pl = plist('AOS', aSignal, 'AOS Variable Names', 'TESTSIGNAL.tm1_x', 'return outputs', outputs)

% Run the simulation
out = simulate(ltp, sim_pl);
```

Note: although the input (and output) ports of the `ssm` models have assigned units, and your injected signals may have units, these are not currently checked to be consistent. This will likely change in a future release.

You can then plot the output signals by doing the following:

```
% Create a plist to configure iplot to plot with subplots
plot_pl = plist('arrangement', 'subplots');

% First extract the internal AOs we want and then plot
o1  = out.getObjectAtIndex(1);
o12 = out.getObjectAtIndex(4);
IS_tm1_x = out.getObjectAtIndex(7);
IS_tm2_x = out.getObjectAtIndex(13);

% Then plot the IFO and IS signals together
iplot(o1, o12, plot_pl);
iplot(IS_tm1_x, IS_tm2_x, plot_pl);
```
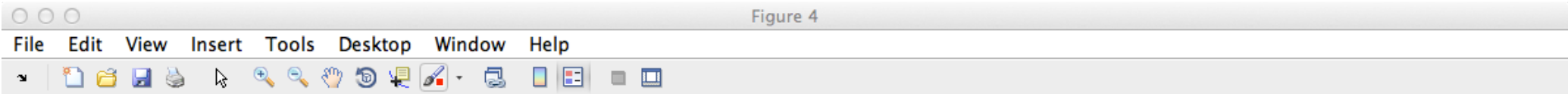
You should then see plots like:

Simulate LTP with injected signals (no noise) (LTPDA Toolbox)

Simulate LTP with injected signals (no noise) (LTPDA Toolbox)

Simulate LTP with injected signals (no noise) (LTPDA Toolbox)

©LTP Team
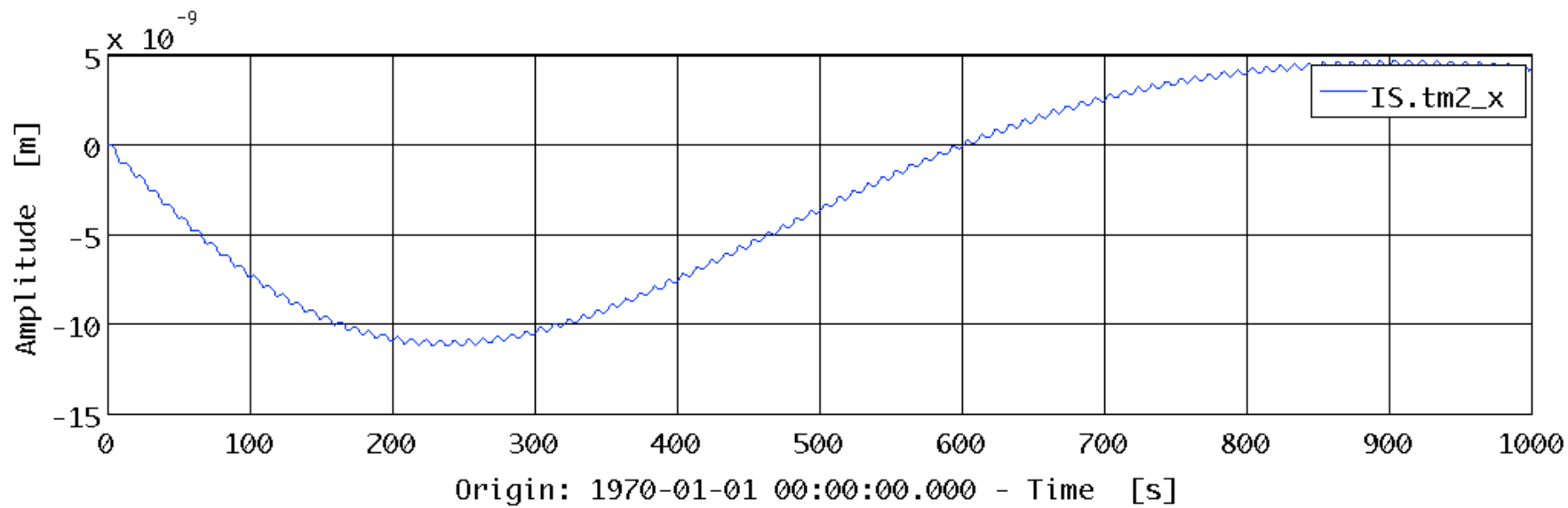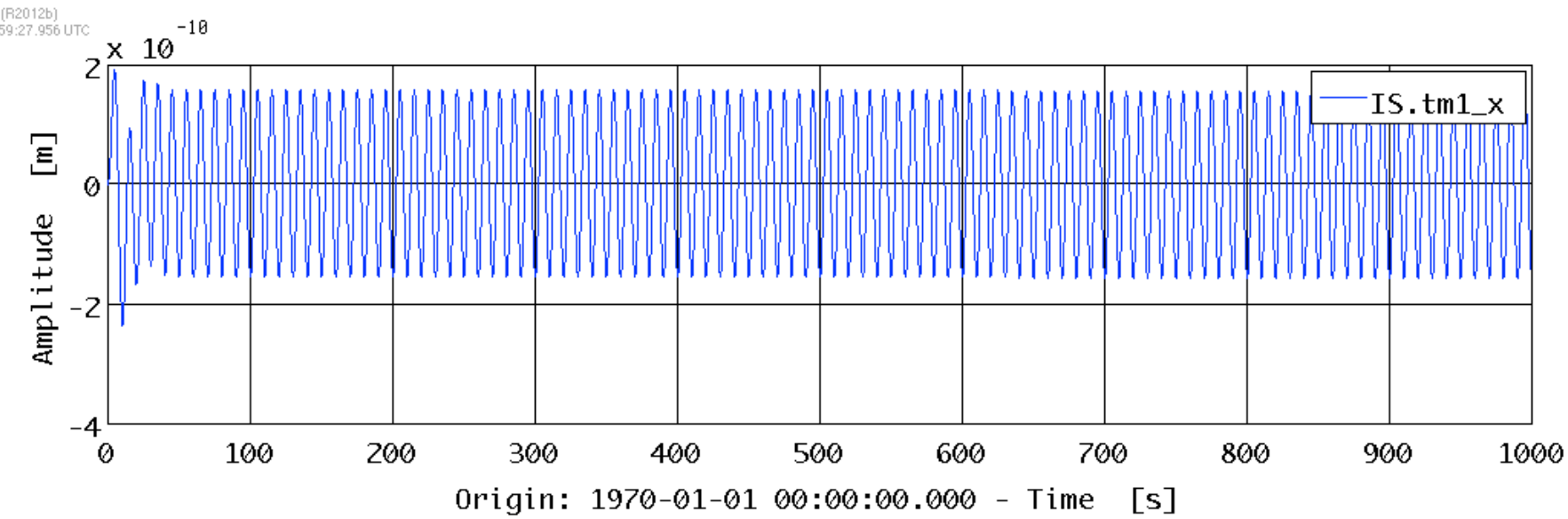
Inject noise signals to LTP

We can inject any type of signal in to the simulations. In the previous section we injected sinusoidal siganls. Here we inject noise.

Let's start by making some random noise time-series data:

```
      % Create a noise time-series analysis object to inject
      fs    = 1/ltp.timestep;
      nsecs = 1000;
      noise = ao.randn(nsecs, fs);
      noise.setName;

      % Generate a list of outputs we want from the simulator
      outputs = ltp.getPortNamesForBlocks(plist('blocks', {'DELAY_IFO', 'IS'}, 'type',
'outputs'));

      % Create the plist to configure simulate
      sim_pl = plist('AOS', noise, 'AOS Variable Names', 'TESTSIGNAL.tm1_x', 'return
outputs', outputs)

      % Run the simulation
      out = simulate(ltp, sim_pl);
```

You can then plot the output signals by doing the following:

```
      % Create a plist to configure iplot to plot with subplots
      plot_pl = plist('arrangement', 'subplots');

      % First extract the internal AOs we want and then plot
      o1  = out.getObjectAtIndex(1);
      o12 = out.getObjectAtIndex(4);
      IS_tm1_x = out.getObjectAtIndex(7);
      IS_tm2_x = out.getObjectAtIndex(13);

      % Then plot the IFO and IS signals together
      iplot(o1, o12, plot_pl);
      iplot(IS_tm1_x, IS_tm2_x, plot_pl);
```

©LTP Team

Inject noise signals to LTP (LTPDA Toolbox)

Estimate tranfser functions from simulated signals, compare with Bode estimates

---

We can estimate transfer functions of the system in two ways: by injecting noise and using spectral estimator methods, or by computing the Bode response based on the system matrices.

Measuring a transfer function

Let's start by injecting a white noise time-series in to the guidance input of the y1 control coordinate:

```matlab
% Create a noise time-series analysis object to inject
fs    = 1/ltp.timestep;
nsecs = 10000;
noise = ao.randn(nsecs, fs);
noise.setYunits('m'); % We want to inject [m]. Then the estimated TF later will have the right units.
noise.setName;

% Generate a list of outputs we want from the simulator
outputs = ltp.getPortNamesForBlocks(plist('blocks', {'DELAY_IFO', 'IS'}, 'type', 'outputs'));

% Create the plist to configure simulate
sim_pl = plist('AOS', noise, 'AOS Variable Names', 'GUIDANCE.is_tm1_y', 'return outputs', outputs)

% Run the simulation
out = simulate(ltp, sim_pl);
```

You can then plot the output signals by doing the following:

```matlab
% Create a plist to configure iplot to plot with subplots
plot_pl = plist('arrangement', 'subplots');

% First extract the internal AOs we want and then plot
IS_tm1_y = out.getObjectAtIndex(8);
IS_tm2_y = out.getObjectAtIndex(14);

% Then plot the IS y signals together
iplot(IS_tm1_y, IS_tm2_y, plot_pl);
```

To estimate the transfer function between the injected noise and the signal recorded from the IS of TM1, we can do

```matlab
tfe_pl = plist('navs', 16, 'order', 1, 'win', 'BH92');
Tn2is1y = tfe(noise, IS_tm1_y, tfe_pl);
```

To plot the estimated transfer function, you can do:

```
iplot(Tn2is1y);
```

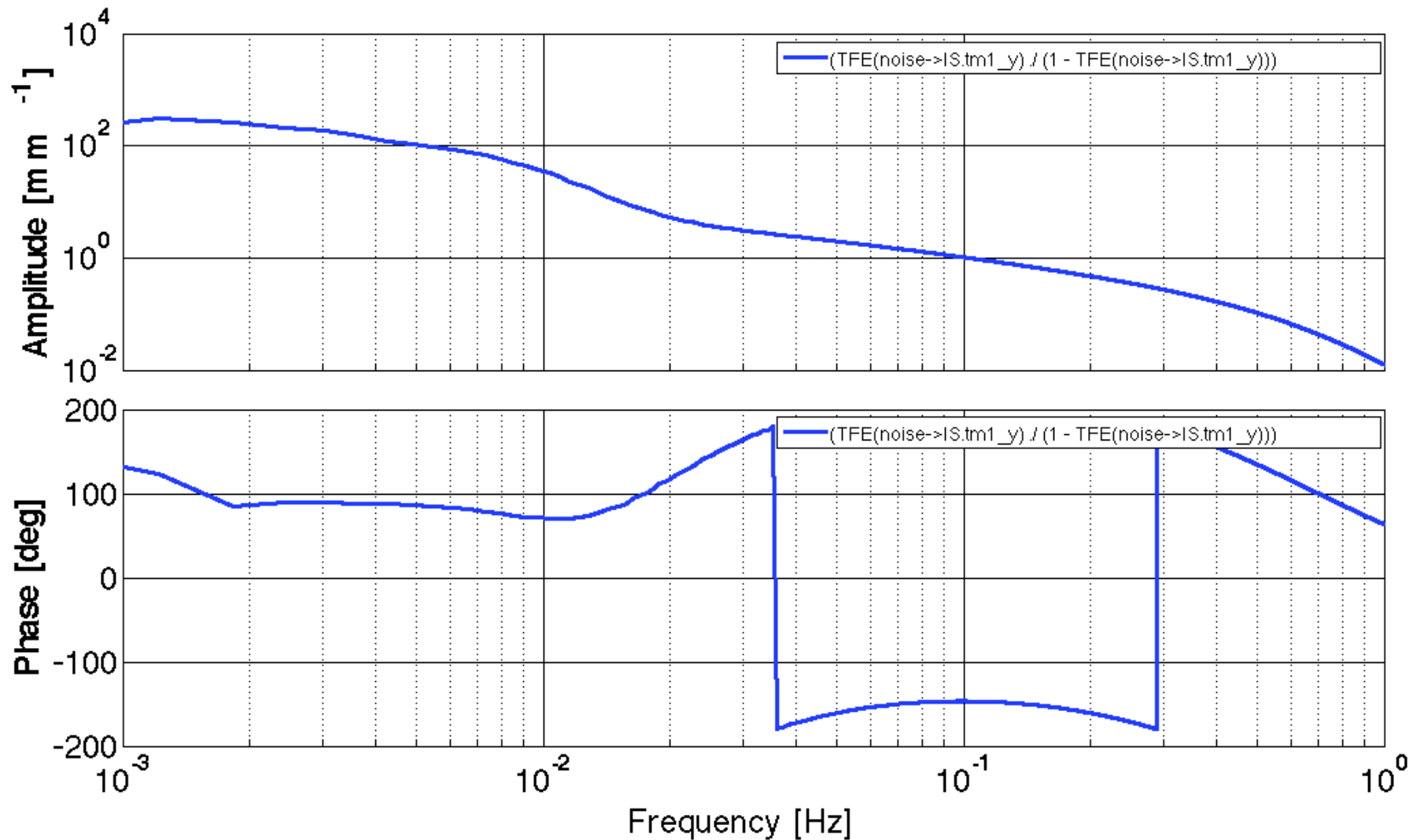This transfer function is the complementary sensitivity transfer function defined as:

$$T(s) = \frac{O(s)}{1 + O(s)}$$

where `O(s)` is the open loop gain of the system. We can calculate `O(s)` as

```
O = Tn2is1y / (1 - Tn2is1y)
```

which looks like this when plotted:

```
iplot(O, plist('xranges', {'all', [1e-3 1]}));
```

Bode response of the system

We can also extract transfer functions from the system directly using the `ssm/bode` function. The usage of the `bode` method is very similar to simulate. You have to specify the input and output ports, and a set of frequencies to compute the response at, and `bode` returns one frequency-series for each possible input to output transfer function.

Using the same LTP model we've been using all along, you can extract the same transfer function as the one measured above by doing:

```
% Create a plist to configure bode. Evaluate the response at the frequencies of the measured TF from above.
bpl = plist('inputs', 'GUIDANCE.is_tm1_y', 'outputs', 'IS.tm1_y', 'f', Tn2is1y.x);
```
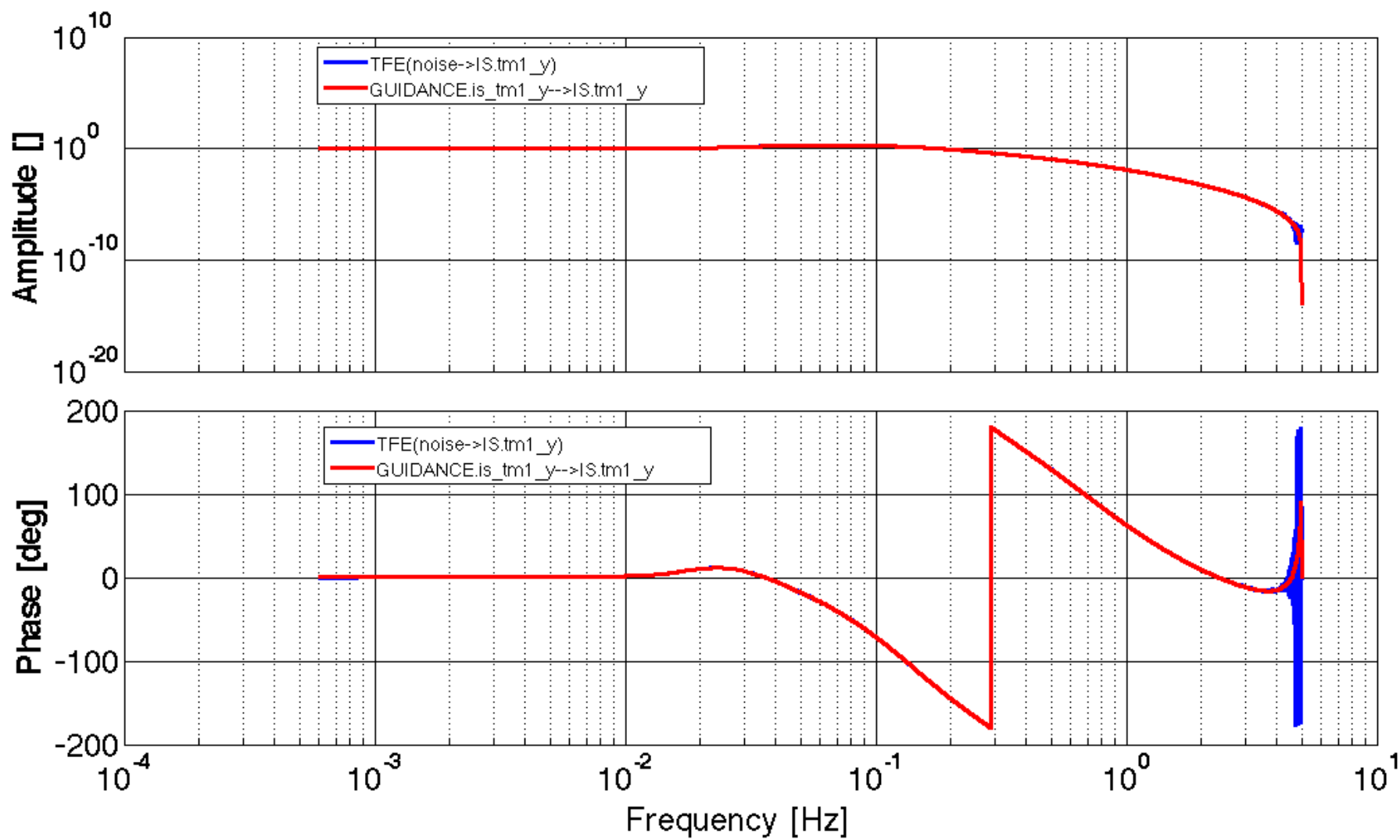
```
        % Compute the response
        out = bode(ltp, bpl);
```

You can plot the response, together with the one estimated above by doing:

```
        Tb = out.getObjectAtIndex(1);
        iplot(Tn2is1y, Tb);
```

You should see a plot something like:

©LTP Team

Simulate LPF with injected signals

We can, of course, do the same injections but with the full LPF model and with noise switched on.

In Topic 2 you already saw how to simulate an LPF model with all the noises switched on. If we combine that with the signal injections we've been doing in this topic, then a simulation with noise and injected signals can be done like this:

```matlab
    % Create a standard LPF model
    lpf = ssm(plist('built-in', 'LPF'));

    % Generate suitable covariance matrix for all inputs
    cov = lpf.generateCovariance;

    % Create some time-series analysis object to inject
    aSignal = ao(plist('tsfcn', '1e-7*sin(2*pi*0.005*t)', 'fs', 1/lpf.timestep, 'nsecs', 10000));

    % Create the plist to configure simulate
    sim_pl = plist('AOS', aSignal, 'AOS Variable Names', 'GUIDANCE.ifo_x1', ...
                   'return outputs', {'DELAY_IFO.x1', 'DELAY_IFO.x12'}, ...
                   'CPSD variable names', cov.find('names'), ...
                   'CPSD', cov.find('cov'));

    % Run the simulation
    out = simulate(lpf, sim_pl);
```
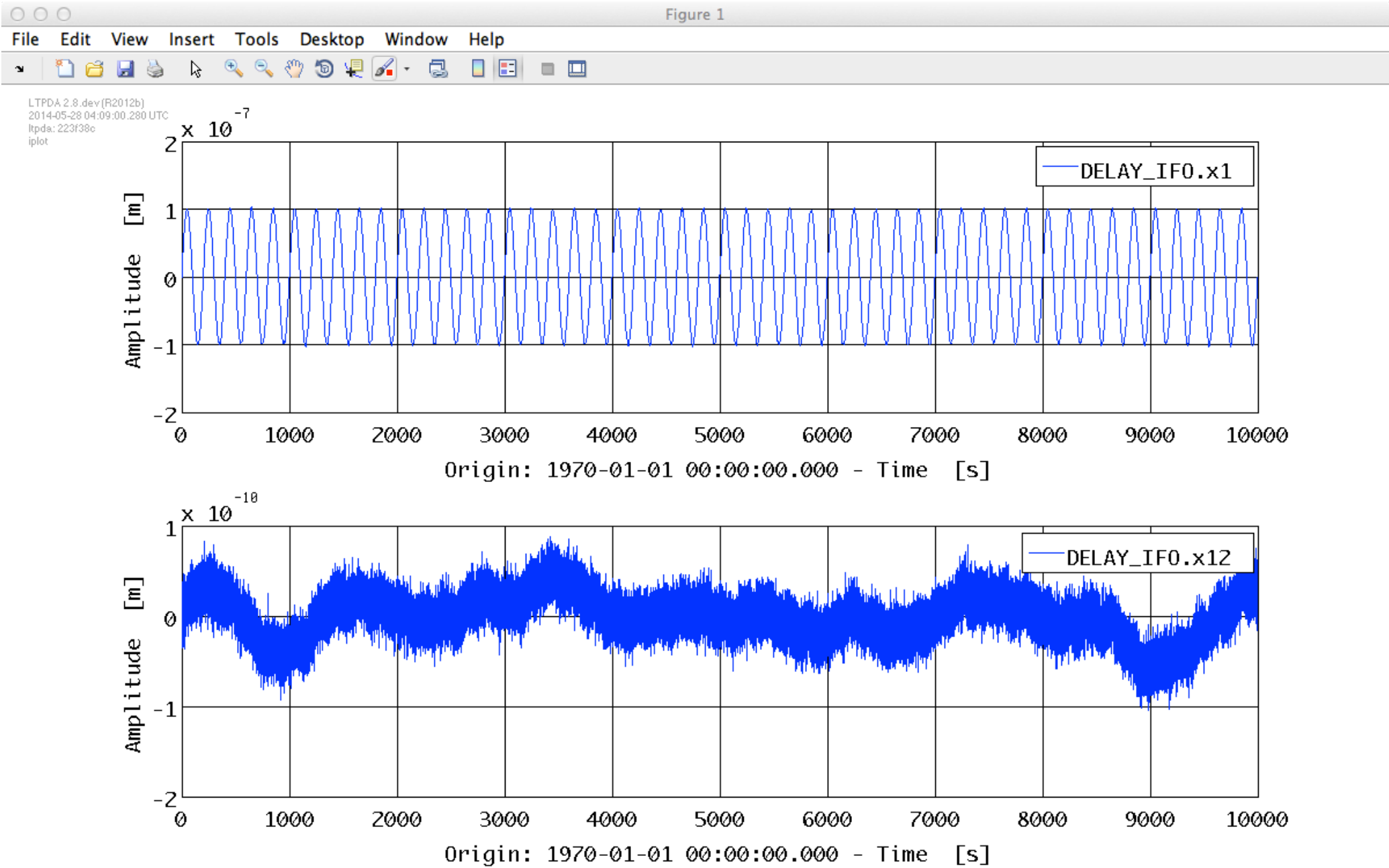
We can then plot the two IFO outputs by doing:

```matlab
    % Unpack the outputs
    [o1, o12] = unpack(out);

    % Plot on subplots
    plot_pl = plist('arrangement', 'subplots');
    iplot(o1, o12, plot_pl);
```

This should result in a plot like this:

Simulate LPF with injected signals (LTPDA Toolbox)

File   Edit   View   Insert   Tools   Desktop   Window   Help

LTPDA 2.8.dev (R2012b)
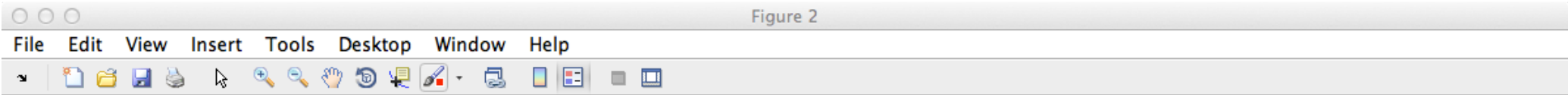2014-05-28 04:09:00.280 UTC
ltpda: 223f38c
iplot

We can estimate the spectral densities of these two signals and we will should a signal at 5mHz:

```
% Create a PSD plist with 16 averages, linear segment-wise detrending
```

```
psd_pl = plist('navs', 16, 'order', 1, 'win', 'BH92');

% Estimate PSDs
[o1_xx, o12_xx] = psd(o1, o12, psd_pl);

% Plot the ASDs
iplot(sqrt(o1_xx), sqrt(o12_xx));
```

This should result in a plot like this:

Simulate LPF with injected signals (LTPDA Toolbox)

Figure 2

©LTP Team

LTPDA Toolbox™   [contents]   ◀ ▶

Topic 5 – Introduction to system identification of LPF.

Topic 5 introduces system identification in the context of LPF. We will describe a simplified experiment, introduce some fitting tools, and go ahead and estimate some parameter values of a simulated experiment.

1. Introduction to LTPDA's fitting tools (theory, implementation, usage)
2. A simplified LPF system identification experiment
3. Create simulated experiment data sets
4. Build state-space models for system identification
5. Calculated expected errors of the parameters (FIM)
6. Perform system identification to estimate desired parameters
   - Parameter Estimation with MCMC
   - Linear Parameter Estimation with Singular Value Decomposition
7. Results and comparison
8. Use parameter estimates to estimate residual differential acceleration

◀Simulate LPF with injected signals   Introduction to LTPDA's parameter estimation tools ▶

©LTP Team

Introduction to LTPDA's parameter estimation tools

In this topic we will rum some parameter estimation analysis on LTP simulated data. Although the analysis will run on `ssm` objects, the basics of the analysis and tools used can be understood in the following terms. Assume a system described (in frequency domain) as follows,

$$o(\omega) = G(\theta;\omega)\,s(\omega) + n(\omega)$$

where we have an applied input (s) that goes through a system defined by a set of transfer functions (G) to produce a measured output (o), that is usually contaminated by a random process (n). Notice that in our exercise we use two LTP measurement, x1 and x12 and, therefore in reality we will be dealing with a 2D problem. Our system decomposed in components (and omitting dependences) is:

$$\vec{o} = \begin{pmatrix} o_1 \\ o_{12} \end{pmatrix}, \quad \vec{s} = \begin{pmatrix} s_1 \\ s_{12} \end{pmatrix} \quad \vec{n} = \begin{pmatrix} n_1 \\ n_{12} \end{pmatrix}, \quad \mathbf{G} = \begin{pmatrix} G_{11} & G_{12} \\ G_{21} & G_{22} \end{pmatrix}$$

Our aim is to obtain the parameters via the dependence on the transfer function. We will use two different methods for that.

Linear Parameter Estimation with Singular Value Decomposition

The first approach is based on least-squares. If we expand the transfer function and keep it to the first order,

$$o(\omega) = G(\theta_1^0, \theta_2^0, \ldots \theta_N^0; \omega) \cdot s(\omega) + \sum_i \frac{\partial G(\theta_1, \theta_2, \ldots \theta_N; \omega)}{\partial \theta_i} \Delta\theta_i \cdot s(\omega) + \ldots$$

If we do not consider 2nd order terms, we get a constant zeroth order term and the parameters enter only with a linear dependence. This equation define a linear system that can be solved analytical.

Whitening filter
It is worth mentioning that to correctly apply the least-squares approach, the noise must be uncorrelated and therefore the method will need to whiten (or uncorrelate) the data. In a 1D case, the process requires to fit the square root of the noise power spectrum density with a digital filter

$$h(z^{-1}) = \sum_i^N \frac{r_i}{1 - p_i\,z^{-1}}$$

and use the inverse of that filter to 'whiten' the data. In 2D systems, like the ones treated here the procedure is more involved, implying the eigendecomposition of the cross-power spectrum matrix.

Implementation

In LTPDA, the previous scheme is implemented in different methods that the user can tune and apply. In particular, for this analysis we will use a method to fit a frequency domain object

```
    % Fit frequency domain (continuous)
     out = sDomainFit(in, pl_fit);
    % Fit frequency domain (discrete)
     out = zDomainFit(in, pl_fit);
```

A method that builds the whitening filters

```
    % Build whitening filter
   out = buildWhitener1D(in, plw);
```

An the method to solve the linear system and get the parameters

```
    % Run Linear Fit
   params = linfitsvd(in, plist);
```

Markov Chain Monte Carlo

The second apporoach that we will use is a statistical method based on a random sampling of the parameter space. In this method, new samples are generated using a Markov Chain mechanism and in each jump (or step) in the parameter space, the likelihood is calculated. According to our previous description, the likelihood reads as

$$\log \Lambda \propto -\frac{1}{2} \left[o(\omega) - G(\theta; \omega)\, s(\omega) \right]^T \Sigma^{-1} \left[o(\omega) - G(\theta; \omega)\, s(\omega) \right]$$

where the C stands here for the covariance matrix between channels, that we build computing the cross−power spectrum matrix. This term plays the role here of the whitening filter in the previous method. The MCMC will sample the parameter space looking for the parameters that maximize the likelihood, eventually reaching the maximum likelihood estimates. Since the method will draw random samples during this process, we will be able to trace the shape of the likelihood surface during the process from where we will be able to derive the posterior probability distribution of the parameters.

Fisher matrix
An important tool in this section will be the Fisher matrix, usually defined as

$$F_{ij} = E\left[\left(\frac{\partial \log \Lambda}{\partial \theta_i} \right)^* \left(\frac{\partial \log \Lambda}{\partial \theta_j} \right) \right]\Bigg|_{\theta}$$

which can be used to derive a lower bound for the variance of an unbiased estimator. This result is known as the Cramer−Rao bound,

$$\mathrm{cov}[\theta] \geq \mathbf{F}^{-1}$$

We will use this in two different steps. First, to get an estimate of the expected errors that we should get for each parameter (and correlations between them). In this sense, the Fisher matrix is directly related with the design of our experiment. Second,we will use the inverse of the Fisher matrix as an input to the MCMC. It is a

standard procedure in MCMC algorithms to use a proposal distribution which encodes the information of this covariance matrix to draw new samples in the Markov Chain process. In that way we increase efficiency by 'jumping' into the right directions. In our case the proposal distribution will be a multivariate gaussian with the covariance given by the inverse of the Fisher matrix

Implementation
Methods to implement this analysis scheme are, first, the one to compute the inverse of the Fisher matrix

```
    % Compute inverse of Fisher matrix
    out = crb(in, pl_crb);
```

An one implementin the MCMC

```
    % Run MCMC
    out = mcmc(in, pl_mcmc);
```

References

http://prd.aps.org/abstract/PRD/v82/i12/e122002

- F Ferraioli et al., Calibrating spectral estimation for the LISA Technology Package with multichannel synthetic noise generation, Phys. Rev. D 82, 042001 (2010).
- M Nofrarias et al., Bayesian parameter estimation in the second LISA Pathfinder mock data challenge, Phys. Rev. D 82, 042001 (2010).
- F Ferraioli et al., Quantitative analysis of LISA pathfinder test-mass noise, Phys. Rev. D 84, 122003 (2011).
- M Nofrarias et al., Parameter estimation in LISA Pathfinder operational exercises arXiv: gr-qc:1111.4916 (2011).

©LTP Team

A simplified LPF system identification experiment

In this section we give an overview of the main elements we will use in the following to simulate some experiments in the LTP and estimate system parameters from them. In order to capture the main concepts of the analysis we perform a limited analysis based on the following constraints:

- We use 1D models of the LTP.
- We restrict our analysis to 5 unknown parameters.
- We only consider as injections (fake) interferometer signals.
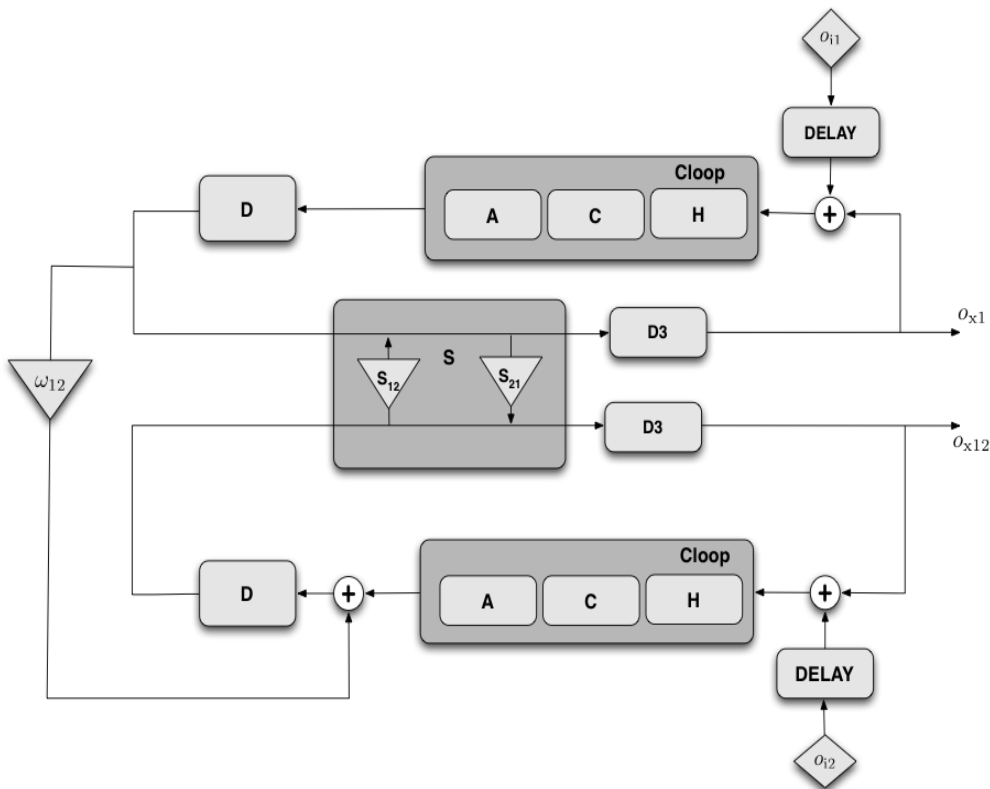
Model used

Fig. Scheme of the LTP as a closed loop system. The triangles represent cross-couplings and rhombus the injections to the system.

The scheme above show the main elements in the model: **D** stands for the dynamics of the test mass; **S** is the sensor (interferometer in our case); the control is split in the controller transfer functions (**H**), the decoupling matrix (**C**) and the actuators (**H**). **DELAY** and **D3** are delay boxes that we will not use in our exercise. In the exercise we will use two `ssm` models implementing the previous scheme: one to generate the data (which) include noise sources and a second used for parameter estimation (where we do not need the noise inputs). These are the following:

Model name	Comment
LTP	noise sources **not** included (type: help ssm_model_LTP)

LPF	noise sources included (type: help ssm_model_LPF)

Parameters used

The parameters relevant for our analysis are described in the table below. The names are according to the notation in the ssm models

Parameter	Description
FEEPS_XX	FEEPs actuation gain in X direction
CAPACT_TM2_XX	Capacitive actuation in TM2 in X direction
IFO_X12X1	Interferometer cross-coupling X1 -> X12
EOM_TM1_STIFF_XX	TM1 stifness in X direction (squared)
EOM_TM2_STIFF_XX	TM2 stifness in X direction (squared)

Injections signals

We will apply two signals to the system. These follow the prescription described in the technical note S2-UTN-TN-3045. Since these are quite used, the LTPDA contains two built-in models to create them in a single line. The first experiment will inject a sinusoid sequence to the x1 channel and the second experiment will inject a different sinusoid sequence to the x12 channel. Hence, the experiments we will study can be summarized in the following table:

Experiment name	Signal applied	Injection port
Inv0001	built-in model: 'signals_3045_1_1'	GUIDANCE.IFO_x1
Inv0002	built-in model: 'signals_3045_1_2'	GUIDANCE.IFO_x12

Measured quantities

In this simplified analysis, our analysis of the displacement (or acceleration) measured on the test masses will only come from the X component of optical read-out for the two channels, x1 and x12. Hence, discarding the remaining degrees of freedom. We will add two quantities that will, as described in Topic 3, will be useful when translating displacement into acceleration noise. The summary of measured quantities (for each experiment) used in the analysis is the following:

Output port	Description
DELAY_IFO.x1	Interferometer x1 measurement
DELAY_IFO.x12	Interferometer x12 measurement
DFACS.sc_x	Commanded force on spacecraft
DFACS.tm2_x	commanded force on test mass #2

A simplified LPF system identification experiment (LTPDA Toolbox)

©LTP Team

Create simulated experiment data sets

In this section of the exercise we show how to creat simulated experiment data sets. First of all, we set a reference time for our experiment. The `t0` can be defined as follows.

```matlab
clear all
% Define a reference time for the experiments
t0 = time('2012-01-17 17:04:11.529 UTC');
```

The two experiments are in fact two separate injections of frequency sweeps in the first and the differential channel. This means that we create a signal of sinusoidals and a signal of zeros for each experiment. These injections are already built-in and can be constructed with the equivalent input key of a plist.

Create first experiment (Inv0001) injection

For the injection signal to the first channel we can use the following:

```matlab
%%% First channel (x1)

% Create input signal from built-in model
i1_1 = ao(plist('built-in', 'signals_3045_1_1'));

% Adding reference time
i1_1.setT0(t0);
```

We can call the zeropad function and add some zeros before and after the actual signal. These chuncks will be used to extract the simulated noise of the LTP system. The `'N'` key of the plist defines the number of samples and with the `'position'` key (`'pre'` or `'post'`) we can define where to add the zeros; before of after the built-in signal constructed.

```matlab
% Add zeros before and after
i1_1.zeropad(plist('N', 200000, 'position', 'pre'));
i1_1.zeropad(plist('N', 20000, 'position', 'post'));

% Set Name
i1_1.setName('Inv0001_x1_inj');
```

The input of zeros to inject to the differential channel is easier to obtain. We construct an `ao` object with an input plist. The `'Yvals'` would be all zeros and the number of samples the same as `inj_1_1`.

```matlab
%%% Second channel (x12)

% Input signal for second channel is zero
% We build this signal by forcing it to lay on the same time values as the i1_1
i1_2 = ao(plist('Yvals', zeros(size(i1_1.x)), 'fs', 10, 't0', t0, 'Yunits', 'm'));
i1_2.setToffset(i1_1.toffset);
i1_2.setName('Inv0001_x12_inj');
```

Plotting the injection for the two channels in the first experiment produces the figure below.

```matlab
% Plot injection signals
iplot(i1_1, i1_2)
```
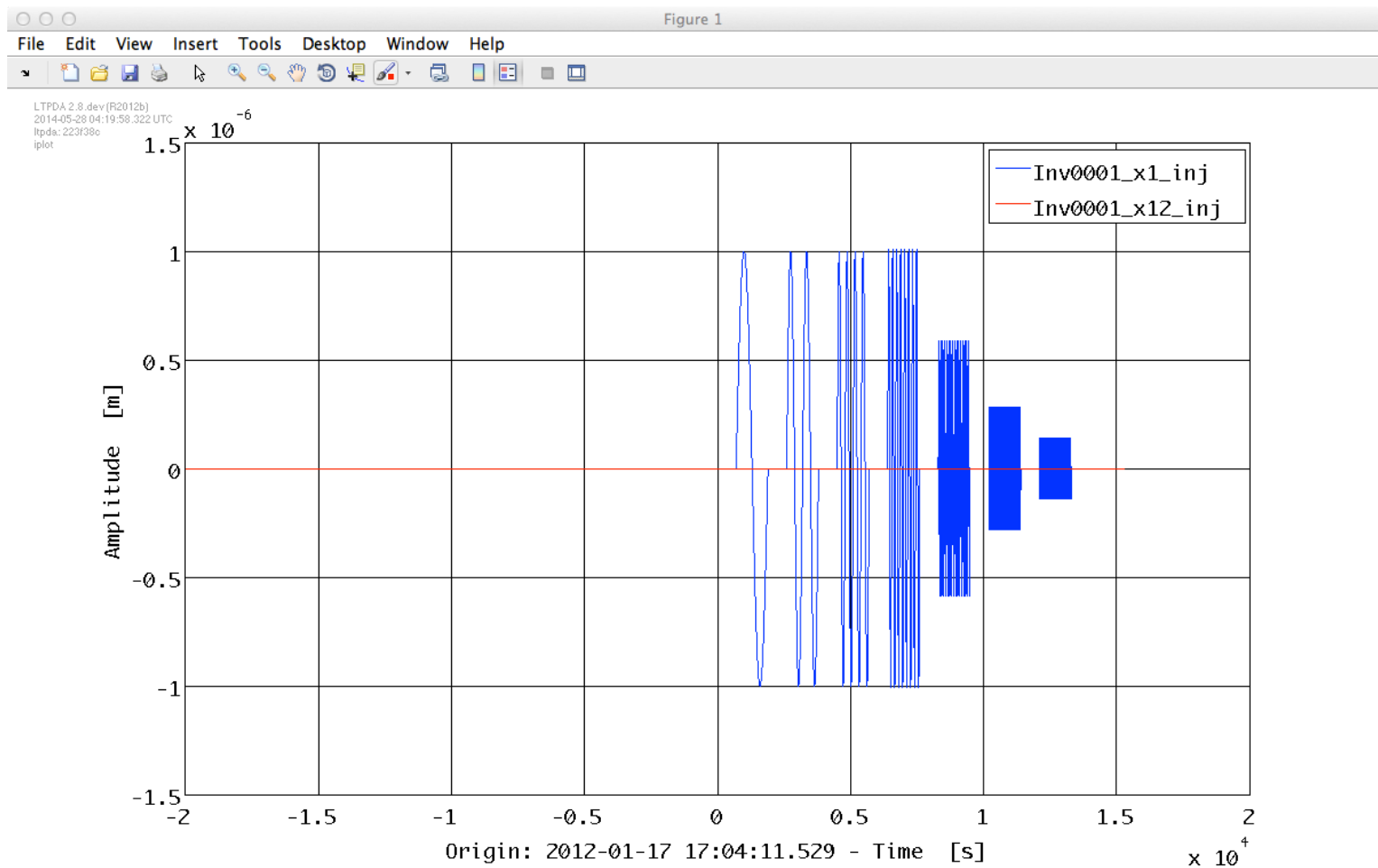
## Create second experiment (Inv0002) injection

The same would apply for the second experiment with the exception that:

1. The injection is applied now to the second channel.
2. The injected signal is different from the previous. We use the built-in signa `signals_3045_1_1`.

```
%% Generate data for experiments Inv0002

%%% Second channel

i2_2 = ao(plist('built-in','signals_3045_1_2'));
i2_2.setT0(t0);
i2_2.zeropad(plist('N', 200000,'position', 'pre'));
i2_2.zeropad(plist('N', 20000,'position', 'post'));
i2_2.setName('Inv0002_x1_inj');

%%% First channel

i2_1 = ao(plist('Yvals', zeros(size(i2_2.x)), 'fs', 10, 't0', t0));
i2_1.setToffset(i2_2.toffset);
i2_1.setName('Inv0002_x12_inj');

iplot(i2_1, i2_2)
```
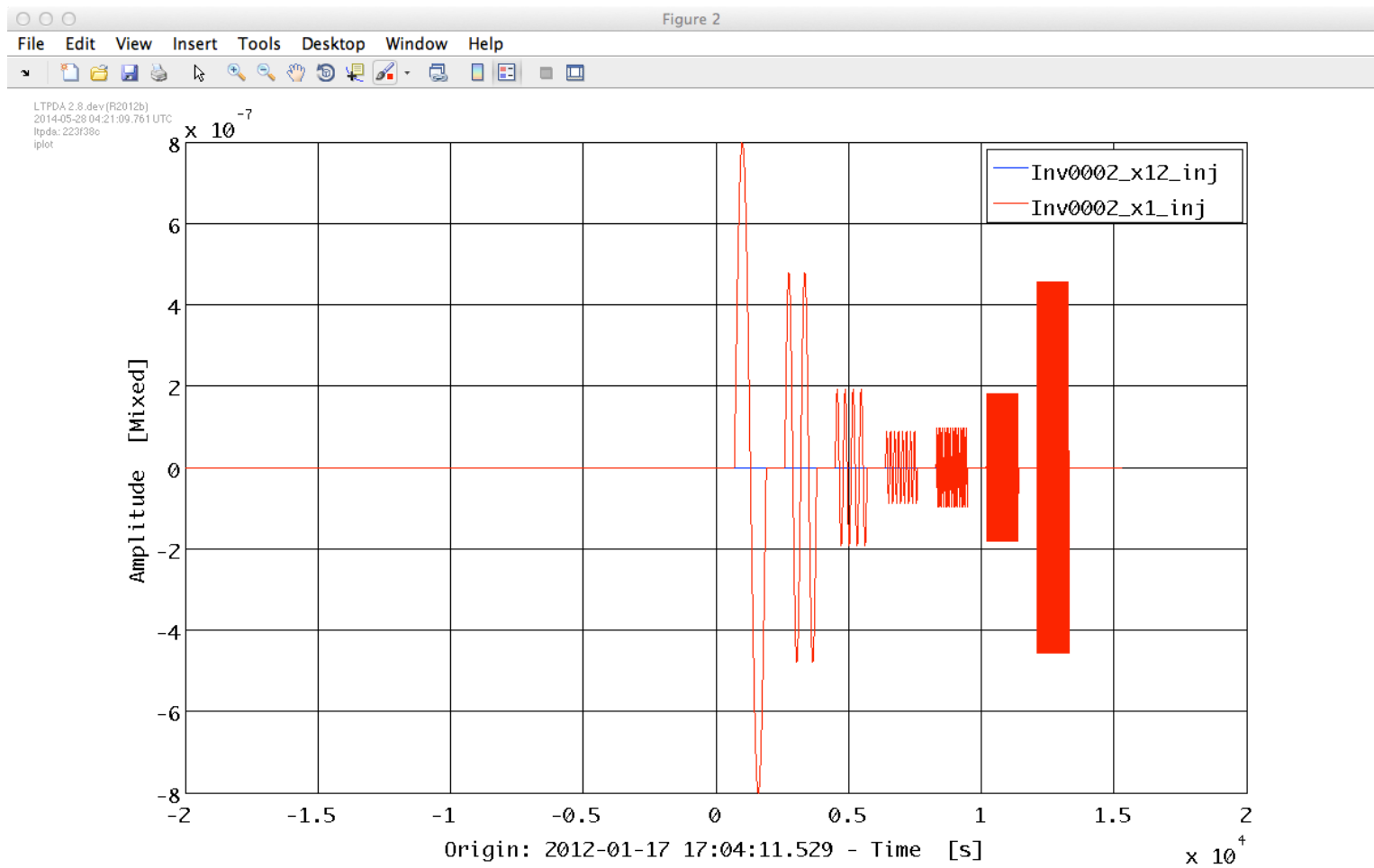
Create the model for data generation

To generate our data sets we will use the built-in model `LPF` which assemble in a single object all the subsystems together with a default profile of noise inputs. For more information on the model type

```
help ssm_model_LPF
```

We will build this model at the same time that we set some parameters to a certain numerical value. These are the parameters that we will be estimating in later in this exercise and are described in the following table

Key	Value	Description
'FEEPS_XX'	0.82	FEEPs actuation gain in X direction
'CAPACT_TM2_XX'	1.08	Capacitive actuation in TM2 in X direction
'IFO_X12X1'	0.0004	Interferometer cross-coupling X1 -> X12
'EOM_TM1_STIFF_XX'	1.3e-6	TM1 stifness in X direction (squared)
'EOM_TM2_STIFF_XX'	1.9e-6	TM2 stifness in X direction (squared)

On our implementation, we first define a string `params` with the name of the parameters, an array `values` with the numerical values previously defined and then we input them to the plist that defines our model.

```
        % Define parameters and nominal values
        params = {...
                'FEEPS_XX', ...             % Coupling of the commanded force along X to the applied force along X
                'CAPACT_TM2_XX', ...        % Actuaction cross-coupling of TM2 force along X to TM2 force along X
                'IFO_X12X1', ...            % The coupling of the x position of TM1 to the estimated x position of TM2 w.r.t. TM1
                'EOM_TM1_STIFF_XX', ...     % Total stiffness of TM1 along X when moving along X.
                'EOM_TM2_STIFF_XX' ...      % Total stiffness of TM2 along X when moving along X.
                };
        values = [0.82 1.08 0.0004 1.3e-6 1.9e-6];

        % Create plist defining how we want to build the model
        simPlist = plist('built-in','LPF', ... % From built-in models
```

```
                          'DIM', 1, ...                        % We use a one dimensional model
                          'CONTINUOUS', false, ...             % The model descrete
                          'param names', params, ...           % Parameter names
                          'param values', values, ...          % Parameter values
                          'VERSION', 'Best Case June 2011');   % Version of the LPF

        % Create the LPF model.
        LPF = ssm(simPlist);

        %% save model
        save(LPF, 'generation_LPF_model.mat');
```

Generate data for the experiments

To produce a set of data we first need to define how the different noise sources are correlated, i.e. the noise covariance. The `ssm` class provides a method that provides this information (although the user is free to provide his own covariance matrix)

```
        %% Generate noise covariance for experiments 1 & 2
        cov = LPF.generateCovariance;
```

Next, define the ports where the inputs are injected abd where the outputs are measured

```
        % Define input and output channels
        inNames = {...
                'GUIDANCE.IFO_x1'  ...    % x1 control coordinate, read by o1 IFO
                'GUIDANCE.IFO_x12' ...    % x12 control coordinate, read by differential IFO
                };

        outNames = {...
                'DELAY_IFO.x1'  ...       % o1 IFO
                'DELAY_IFO.x12' ...       % o12 IFO
                'DFACS.sc_x'    ...       % commanded force on SC
                'DFACS.tm2_x'   ...       % commanded force on TM2
                };
```

And add this information in a `plist` and use it to call the method `simulate`

```
    pl_sim_Inv0001 = plist(...
                'aos variable names', inNames(1),...
                'aos', i1_1,...
                'return outputs', outNames,...
                'cpsd variable names', cov.find('names'), ...
                'cpsd', cov.find('cov'), ...
                'displayTime', true,...
                't0', t0);

    % Inv0001: run the simulation
    % In this investigation, the guidance is applied on the x1 control coordinate, read by o1 IFO
    pl_sim_Inv0001 = plist(...
                'aos variable names', inNames(1),...        % Names of the inputs
                'aos', i1_1,...                             % injection signals
                'return outputs', outNames,...              % Names of the outputs
                'cpsd variable names', cov.find('names'), ...
                'cpsd', cov.find('cov'), ...
                'displayTime', true,...
                't0', t0);

    % Generate data
    o_0001 = LPF.simulate(pl_sim_Inv0001);
```

We `unpack` to recover all the different output ports we ask for in the `plist`

```
        % The output Analysis Objects are grouped in a matrix object, so we need to index them
        [o1_0001, o12_0001, Fsc_0001, Ftm2_0001] =  o_0001.unpack;
```

And repeat the same operation to produce the data set for the second experiment. Notice that in the plist we change the input port (in order to inject the signal in the differential channel) and the injected signal is now `i2_2`

```
        % Inv0002: run the simulation
        % In this investigation, the guidance is applied on the x12 control coordinate, read by differential IFO
        pl_sim_Inv0002 = plist(...
                    'aos variable names', inNames(2), ...
                    'aos', i2_2, ...
                    'return outputs', outNames, ...
                    'cpsd variable names', cov.find('names'), ...
                    'cpsd', cov.find('cov'), ...
                    'displayTime', true, ...
                    't0', t0);

        % Generate data
        o_0002 = LPF.simulate(pl_sim_Inv0002);

        % The output Analysis Objects are grouped in a matrix object, so we need to unpack them
        [o1_0002, o12_0002, Fsc_0002, Ftm2_0002] =  o_0002.unpack;
```

Build matrix objects for the analysis

The methods that we will use in the analysis work, for simplicity, with `matrix` objects. These objects allow to put several time series together to refelct the fact that our analysis requires many channels. In this particular case only two: x1 and x12. The objects are built as follows:

```matlab
%% Build matrix objects

% Group the guidance injection signals of both experiments in the same matrix model
inj_signal(1) = matrix(i1_1, i1_2, plist('shape', [2 1]));
inj_signal(2) = matrix(i2_1, i2_2, plist('shape', [2 1]));

% Group the measured outputs of both experiments in the same matrix model
meas_signal(1) = matrix(o1_0001, o12_0001, plist('shape', [2 1]));
meas_signal(2) = matrix(o1_0002, o12_0002, plist('shape', [2 1]));

% Group the commanded forces of both experiments in the same matrix model
cmd_forces(1) = matrix(Fsc_0001, Ftm2_0001, plist('shape', [2 1]));
cmd_forces(2) = matrix(Fsc_0002, Ftm2_0002, plist('shape', [2 1]));
```

Once these objects are built, we can easily split our original data into the different segments of interest: we take a short segment where the input was applied for the `in` and `out` object, and a long segment without injected signal for the `noise`.

```matlab
% Split in times to get input, output, noise and commanded forces chunks
in    = split(inj_signal, plist('times', [-2000 inf]));
out   = split(meas_signal, plist('times', [-2000 inf]));
noise = split(meas_signal, plist('times', [-inf -2000]));
Fcmd  = split(cmd_forces, plist('times', [-inf -2000]));
```

Each of these contains two `matrix` objects (one for each experiment), with two time series (for x1 and x12) within. We can take a look at the objects for the second experiment:

```matlab
% Plot injected signal in 2nd experiment
in(2).iplot
```
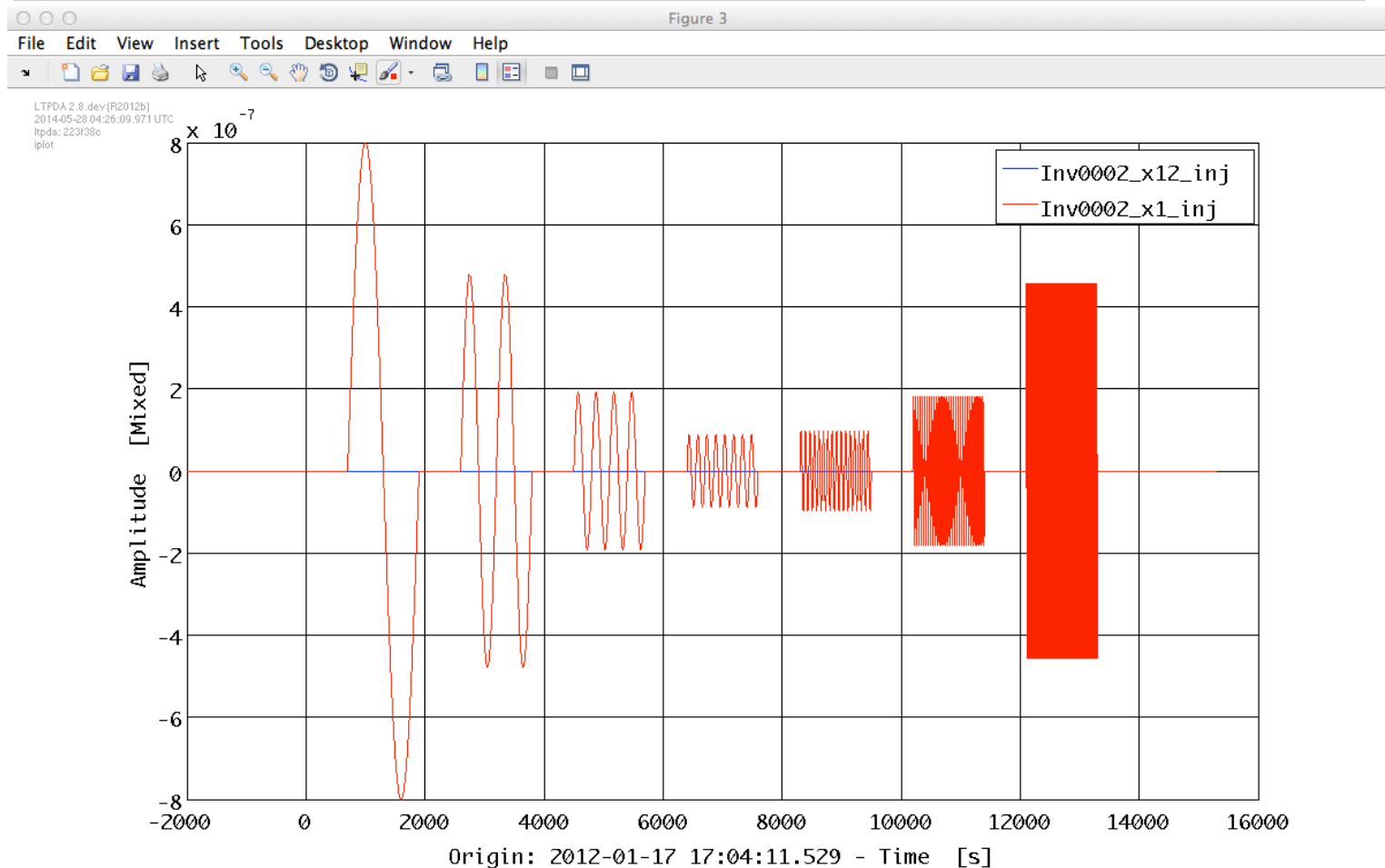
```
% Plot measured signal in 2nd experiment
out(2).iplot
```
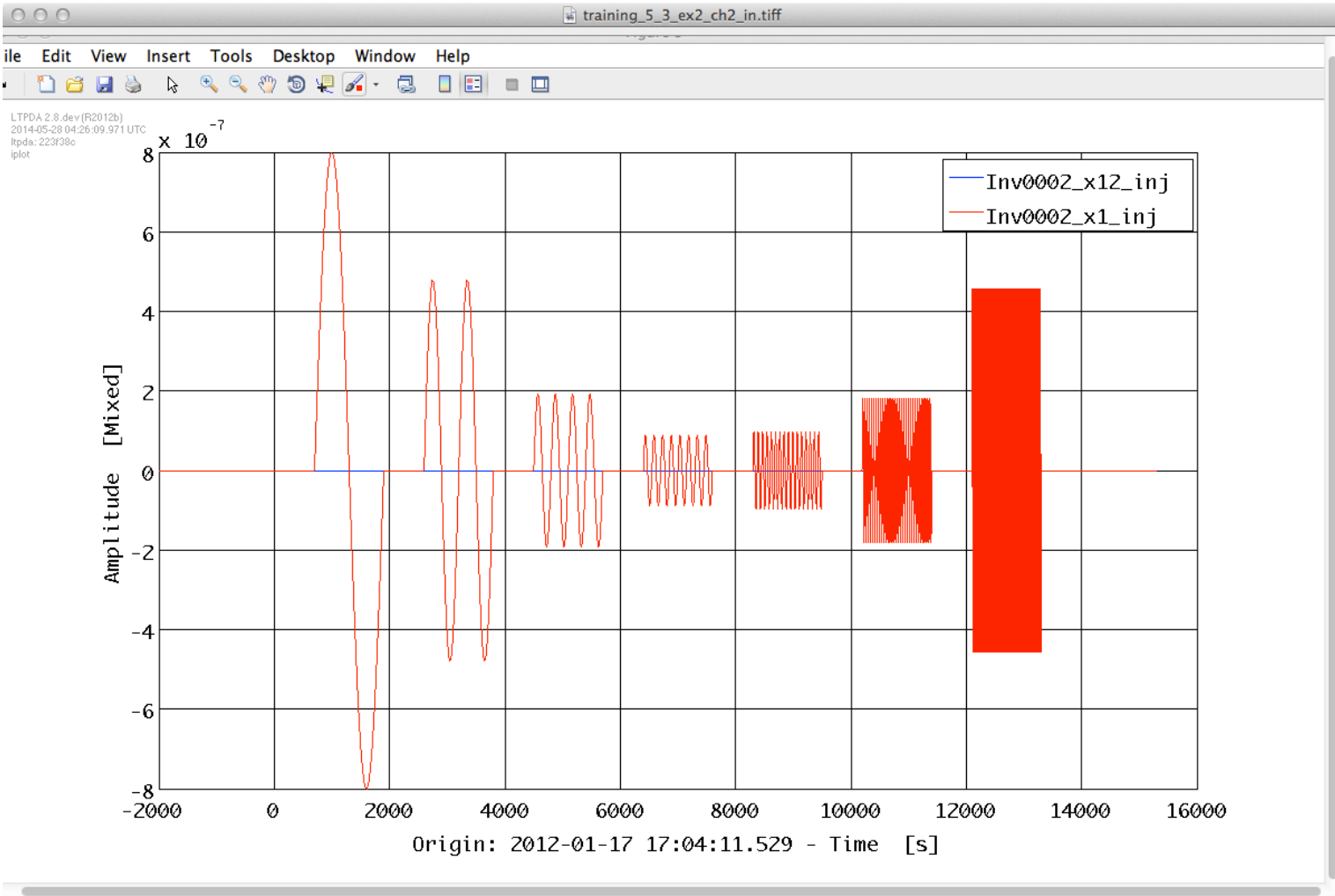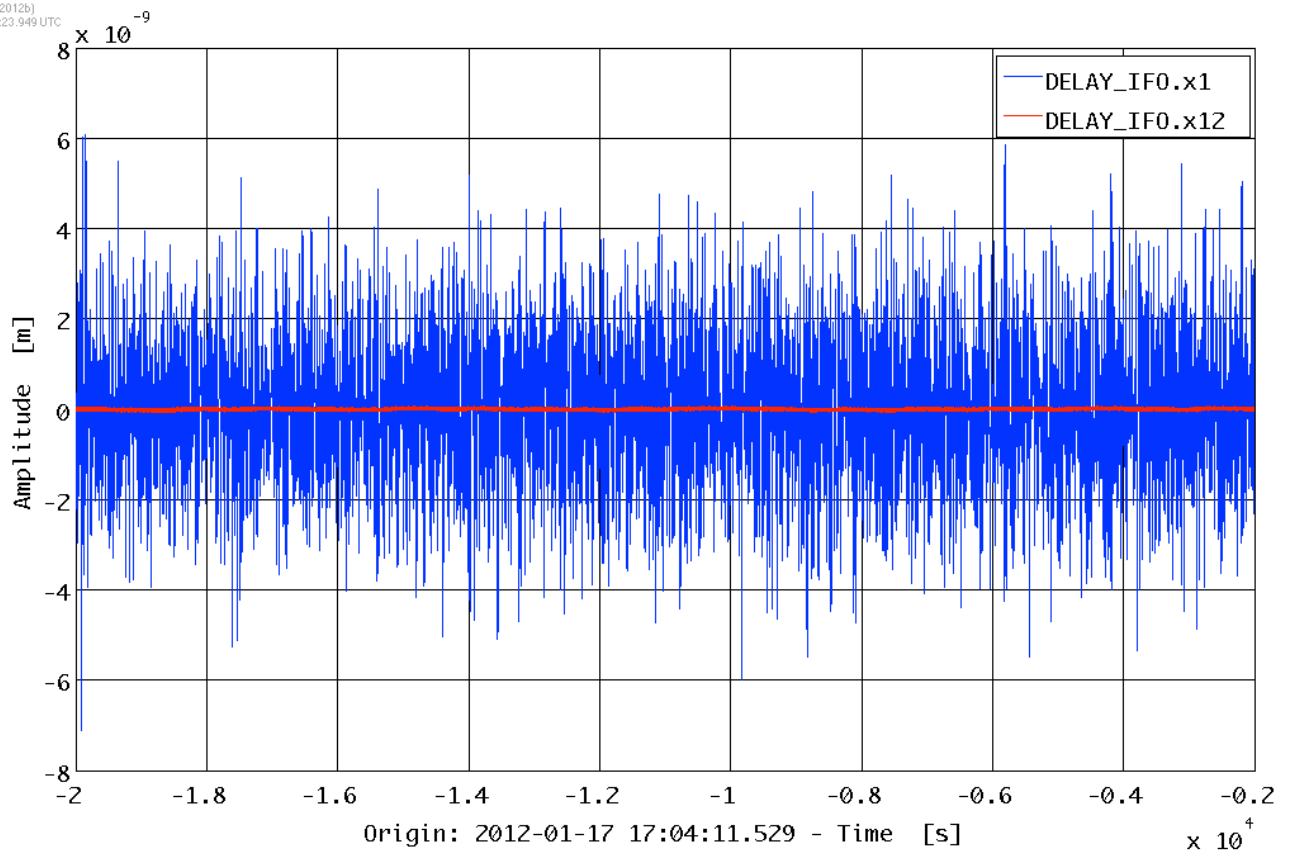


```
% Plot initial noise segment in 2nd experiment
noise(2).iplot
```
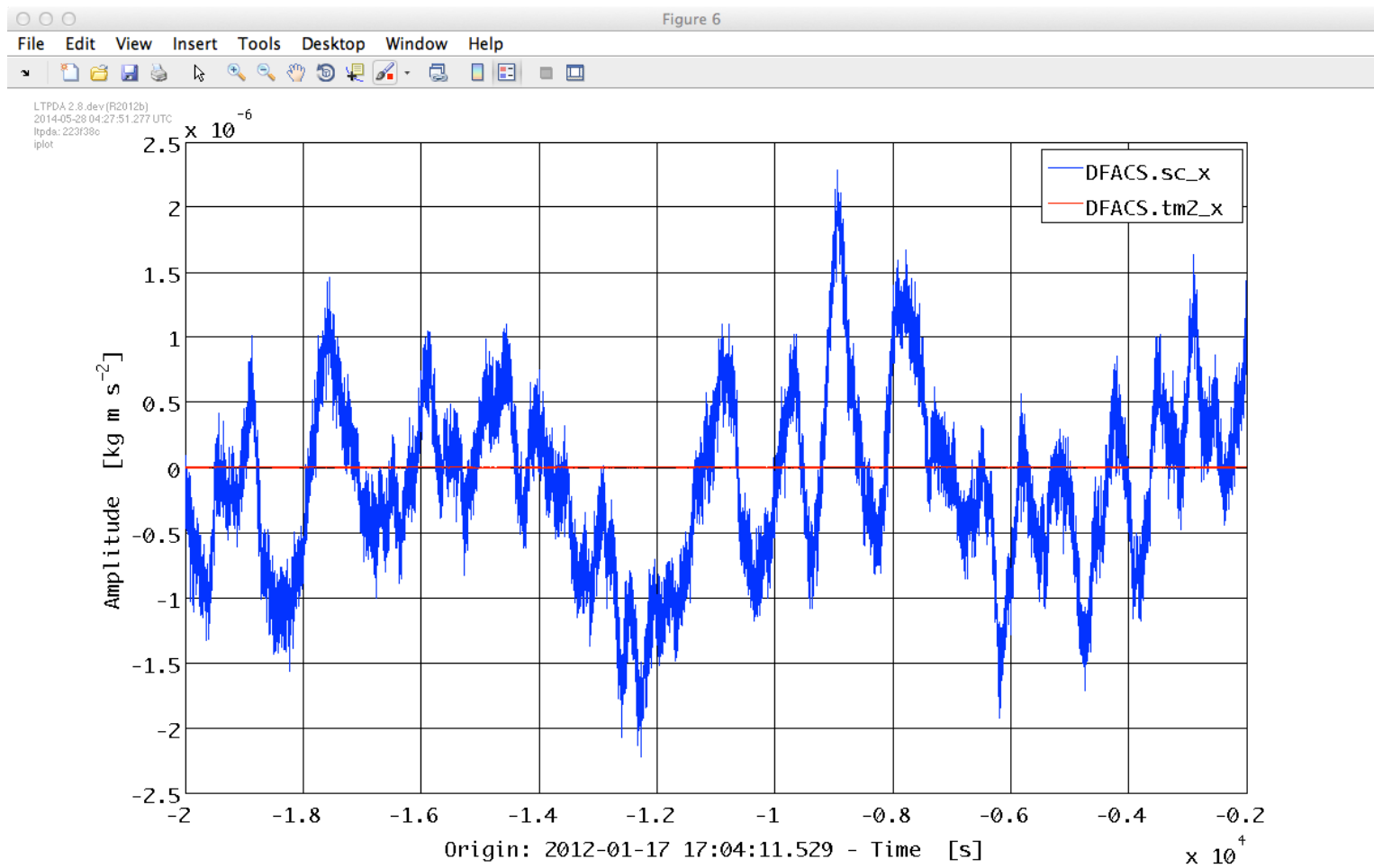
```
% Plot commanded forces in 2nd experiment
Fcmd(2).iplot
```

Finally, we just need to save the objects we have just created. We will use them later for parameter estimation.

```
%% save objects

save([o_0001 o_0002], 'full_signal.mat');
save(inj_signal, 'inj_signal.mat');
save(in, 'input.mat');
save(out, 'output.mat');
save(noise, 'noise.mat');
save(Fcmd, 'cmdForces.mat');
```

©LTP Team

Build state–space LTP models for system identification

The model to use for the estimation of the parameters can be constructed in a similar way as we did before when we simulated the signals. The difference is that the parameters to be estimated must remain symbolic. This is achieved with the `'SYMBOLIC PARAMS'` key of the plist. The model must be continuous.

```
% cell array contains the names of the parameters.
params = {'FEEPS_XX', 'CAPACT_TM2_XX', 'IFO_X12X1', 'EOM_TM1_STIFF_XX', 'EOM_TM2_STIFF_XX'};

%% built ssm model
model = ssm(plist('built-in', 'LTP', ...
    'DIM', 1, ...                       % dimensions of LPF model
    'CONTINUOUS', 1, ...                % Continuous or descrete
    'SYMBOLIC PARAMS', params, ...      % symbolic parameters to estimate
    'VERSION', 'Standard'));            % Version of the LPF

%% save fitting model
save(model, 'fitting_model.mat');
```

We can view all available information about the model if we type `model.viewDetails` in the command window. First of all, the structure of the SSM LTP model is presented and then the available inputs and outputs of the system.

```
>> model.viewDetails
```
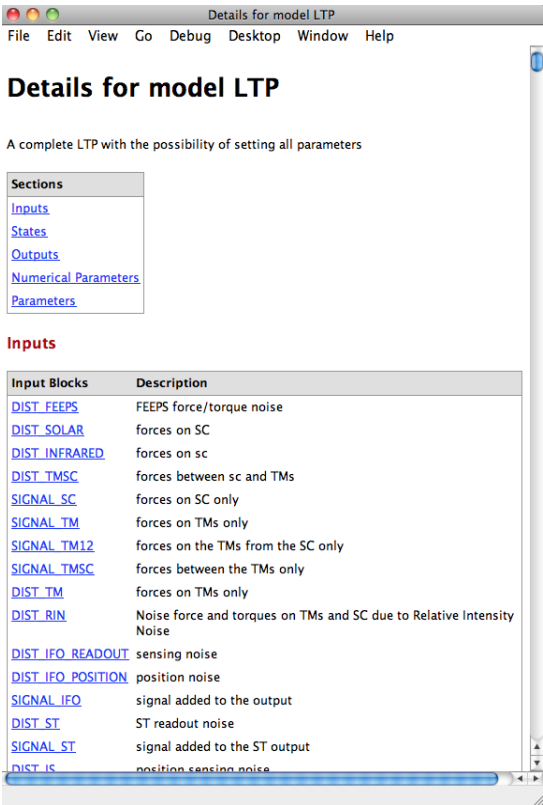
©LTP Team

©LTP Team

Calculate expected covariance of the parameters (FIM)

Once we have defined a models and some injected signals, we can proceed to estimate the errors we can expect on the estimation of our parameters. To do so we compute the inverse of the Fisher Information Matrix (FIM) as shown below.

Our first step is to recover the objects created in the previous section.

```
%% Load objects

in    = matrix('input.mat');
out   = matrix('output.mat');
noise = matrix('noise.mat');
mdl   = ssm('fitting_model.mat');
```

Next step will be to define input/output ports for the `ssm` model and the numerical value for the parameters under analysis. Notice that we evaluate the Fisher matrix for a given set of numerical values, here we will use the ones used to generate data so we will get the optimal expected errors. This is also known as the Cramer–Rao (lower) Bound (CRB).

```
%% Input & Output names
inNames  = {'GUIDANCE.IFO_x1'  'GUIDANCE.IFO_x12'};
outNames = {'DELAY_IFO.x1' 'DELAY_IFO.x12'};

params = {'FEEPS_XX','CAPACT_TM2_XX', 'IFO_X12X1', 'EOM_TM1_STIFF_XX',
'EOM_TM2_STIFF_XX'};
values = [0.82 1.08 0.0004 1.3e-6 1.9e-6];
```

Our `ssm` models are not analytical models which means that to derive them with respect our parameters we need to set a differentiation step. The LTPDA allows to compute it, but we will not enter here in this detail. For this exercise we will use the following values:

```
% Differentiation steps
steps  =  [1e-11 2.3e-07 1e-15 8.4e-11 2.4e-12];
```

And now we can call the `crb` method with the `plist` provided in the example below

```
%% Estimate Covariance matrix
pl_crb = plist('FitParams', params,...
           'paramsValues', values,...
           'diffStep', steps,...
           'ngrid', 5,...                          % Number of points in the grid to
compute the optimal differentiation step for ssm models
           'inNames', inNames,...
           'outNames', outNames,...
           'input', in,...
           'f1', 1e-4,...                          % Initial frequency for the analysis
           'f2', 1,...                             % Final frequency for the analysis
```

```
                    'model', mdl,...
                    'Navs', 5,...                      % Force number of averages
                    'pinv', false);                    % Use the Penrose-Moore pseudoinverse

        mcrb = crb(noise, pl_crb);

        % Save object
        save(mcrb, 'crb_5params_ssm.mat');
```

The method returns an `ao` with the inverse of the Fisher matrix, i.e. the estimated covariance matrix for our parameters. The square root of the diagonal of this object tell us the error standard deviation we can expect for these parameters and the non−diagonal terms, the correlation between them. In our case, for instance:

Parameter	Std. deviation
FEEPS_XX	0.0002
CAPACT_TM2_XX	3.0e−06
IFO_X12X1	1.1e−07
EOM_TM1_STIFF_XX	1.7e−10
EOM_TM2_STIFF_XX	1.6e−10

We will later compare this estimated standard deviations to the ones obtained with our parameter estimation methods.

©LTP Team

Perform system identification to estimate desired parameters

In this section we perform system identification to estimate desired parameters. We follow two approaches. The Marcov Chain Monte Carlo and the Linear Parameter Estimation with Singular Value Decomposition. For both approaches, some definitions are needed. First we load all the objects created for this simulation.

```
clear all

%% Load Objects (signals and model)
in    = matrix('input.mat');
out   = matrix('output.mat');
noise = matrix('noise.mat');
mdl   = ssm('fitting_model.mat');
```

Again we will have to construct cell arrays that will contain the names of the inputs and the outputs of the system.

```
% Input & Output names
inNames  = { 'GUIDANCE.IFO_x1'  'GUIDANCE.IFO_x12'};
outNames = {'DELAY_IFO.x1' 'DELAY_IFO.x12'};
```

The last common action for both methods is to again define the parameters to estimate and also the initial values of the parameters.

```
% Define the parameters to estimate and the starting values.
params = {'FEEPS_XX', 'CAPACT_TM2_XX', 'IFO_X12X1', 'EOM_TM1_STIFF_XX', 'EOM_TM2_STIFF_XX'};
values = [1 1 0.0001 1.e-6 1.e-6];
```

◀ Calculate expected covariance of the parameters (FIM)   Parameter estimation with MCMC ▶

©LTP Team

Perform system identification to estimate desired parameters (LTPDA Toolbox)

Parameter estimation with MCMC

A) Markov Chain Monte Carlo

At this point we have all the inputs necessary to use the Markoc Chain Monte Carlo. We have the injected signals, the simulated output signals, the covariance matrix of the parameters and the LTP model. First, of course, we load all the objects we created before: the signals and the fitting model.

```
in    = matrix('input.mat');
out   = matrix('output.mat');
noise = matrix('noise.mat');
mdl   = ssm('fitting_model.mat');
```

Secondly, we arrange again the parameters to estimate in a cell array and define a vector with their initial values. We also create cell arrays containing the inputs and outputs of the system.

```
inNames  = { 'GUIDANCE.IFO_x1'  'GUIDANCE.IFO_x12'};
outNames = {'DELAY_IFO.x1' 'DELAY_IFO.x12' };

% Also define again the parameters to estimate and our starting values
params = {'FEEPS_XX', 'CAPACT_TM2_XX', 'IFO_X12X1', 'EOM_TM1_STIFF_XX',
'EOM_TM2_STIFF_XX'};
values = [1 1 0.0001 1.e-6 1.e-6];
```

Now, we have to estimate the expected errors of the parameters based on our first guess of the parameters (`[1 1 0.0001 1.e-6 1.e-6]`). The covariance matrix extracted from the `crb` function will be used as the covariance matrix for the proposal distribution of the metropolis algorithm. Just like in section 5.5:

```
    % as in section 5.5 we provide the optimal differentiation step
    steps =  [1e-11 2.3e-07 1e-15 8.4e-11 2.4e-12];

    % We define the plist to input to the crb fuction
    % to estimate the errors of the parameters.
    pl_crb = plist('FitParams', params, ...
                'paramsValues', values, ...
                'diffStep', steps, ...
                'ngrid', 5 ,...                         % Number of points in the grid to compute
the optimal differentiation step for ssm models
                'inNames', inNames, ...
                'outNames', outNames, ...
                'input', in, ...
                'f1', 1e-4, ...                         % Initial frequency for the analysis
                'f2', 1, ...                            % Final frequency for the analysis
                'model', mdl, ...
                'Navs', 5, ...                          % Force number of averages
                'pinv', false);                         % Use the Penrose-Moore pseudoinverse

    % Estimate the CRB
    mcrb = crb(noise, pl_crb);

    % Save object
    save(mcrb, 'crb_5params_fitting.mat');
```

The next step is to call the `mcmc` method of the `LTPDA` after we define the input `plist`. The `plist` will contain all the settings for the metropolis sampling. There are two parameters which set the main characteristics ot the metropolis algorithm:

- `h` is a tempering coefficient that will prevent the chains of getting stuck in localities during the search phase. Increasing it implies 'heating' the chain.
- `Tc` defines the profile of the cooling down, i.e. the duration of the annealing where the 'heat' is on.

We show schematically in the picture below the meaning of these two parameters.
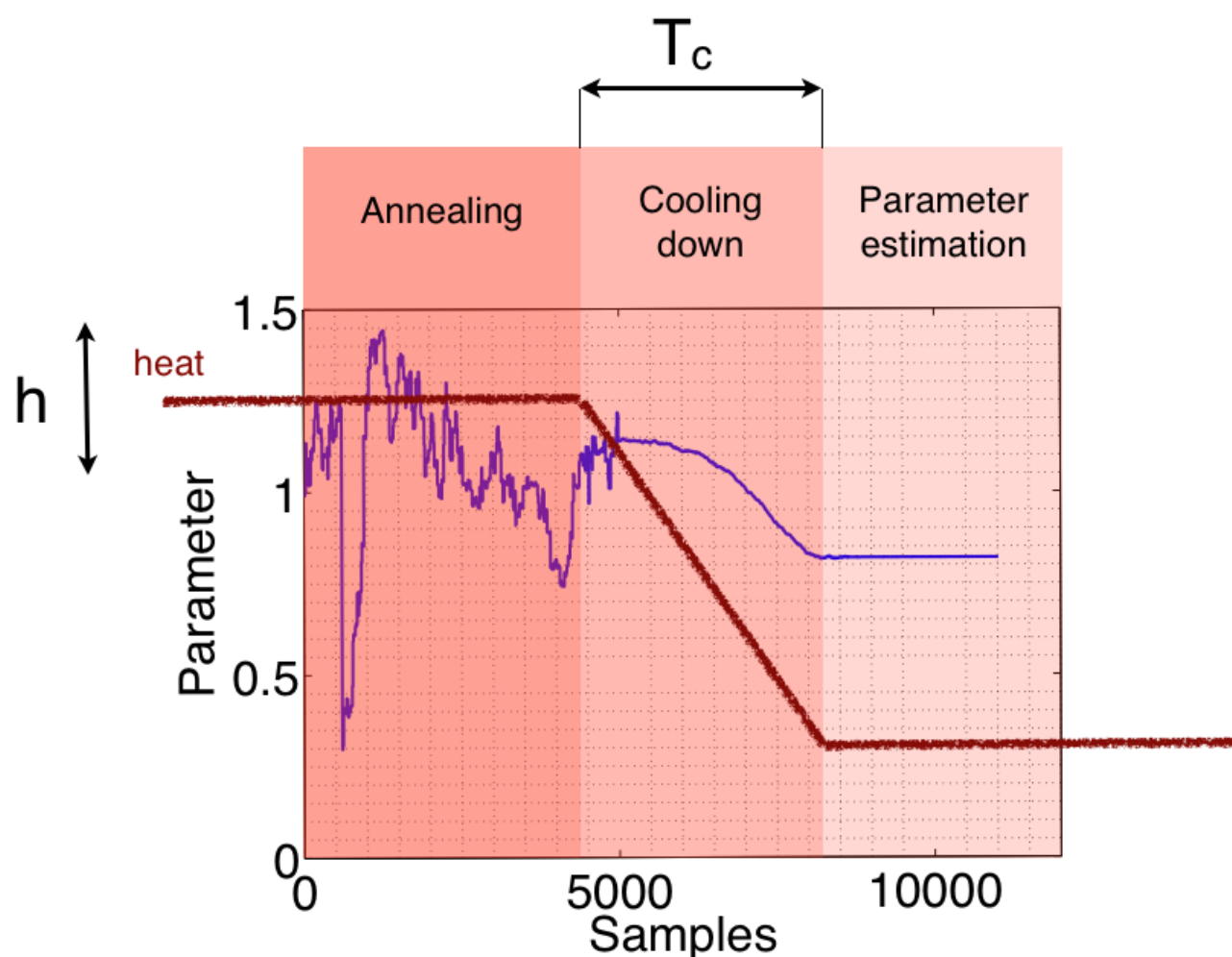
Other parameters to be set are: `fsout`, the desired sampling frequency to resample the input time series or a cell array of `ranges` for the parameters, meaning the upper and lower bound to search for each one.

```
% Now we define some parameters for the mcmc plist

h = 2;              % heat index for simulated annealing
Tc = [100 1000];    % An array of two values setting the initial
                    % and final value for the cooling down. We set it to
                    % these values because we use the simplex algorithm
                    % before the sampling and we get close to the real
                    % values. If simplex was set to false then we should set
                    % Tc = [5000 7000] and 'N' to 9000.

numsampl = 2000;    % Total number of samples

% Upper and lower bounds to search for the parameters
```

```
ranges = {[0 2], [0 2], [0 1], [0 1e-4], [0 1e-4]};
```

Filling the plist:

```
% defining the mcmc plist
pl_mcmc = plist('N', numsampl, ...        % Total number of samples
            'J', 1, ...
            'cov', mcrb, ...
            'range', ranges, ...
            'FitParams', params, ...
            'input', in, ...
            'noise', noise, ...
            'model', mdl, ...
            'inNames', inNames, ...
            'outNames', outNames, ...
            'frequencies', [1e-4 0.5], ...
            'Navs', 5, ...
            'search', true, ...
            'x0', values, ...
            'Tc', Tc, ...
            'heat', h, ...
            'jumps', [2e0 1e2 1e3 1e4], ...
            'simplex', true, ...     % Set simplex to true for speed!
            'plot', [1 2 3 4 5 6], ...
            'debug', false);
% start the mcmc sampling and create a new pest object b
b = mcmc(out, pl_mcmc);
```

Now we can save the `pest` object.

```
%% save mcmc output
save(b, 'fit_mcmc.mat');
```

©LTP Team

Linear Parameter Estimation with Singular Value Decomposition

Topic 5.6 will introduce the principles involved in performing parameters estimation with model linearization in terms of the required parameters. This procedures assumes that noise corrupting the data is white, therefore we will also provide an introduction to data whitening.

The complete procedure can be summarised as:

1. Model linearization. The model linearized in terms of the fit parameters. Which correspond to substitute the model with its Taylor series expansion at the first order.
2. Data whitening. The noise on our data is non-white. Since least-squares methods can be rigorously applied only if the noise corrupting the data is white and uncorrelated, we need a noise whitening step before the fitting operation.
3. Generate whitened templates. In order to correctly perform the fit, both the nominal response and the fit basis need to be whitened. The nominal response is the model response when parameters are set to their nominal values. Fit basis is the response of the first order term of the Taylor series expansion of the model.
4. Basis change. It often happens that the fit basis is composed of linearly dependent elements since several parameters are physically indistinguishable. The system cannot be solved in such conditions, therefore we perform a change of basis with the singular value decomposition (SVD) algorithm. The SVD ensures that we deal with a linearly independent fit basis. The procedure is suited to perform the estimation of parameters value from the combination of the knowledge provided by different experiments on the same system. This ensure, in principle, that the value of each parameter can be obtained with proper accuracy.
5. Fit. The system of equations for the t are nally solved in order to get the best estimation for the parameters.
6. Check convergence and goto 3. The process is iterated until the convergence of the parameters is reached. At each step the nominal values of the parameters are updated with the current knowledge.
7. Change basis back to physical parameters. At this stage we revert the change of the fit basis and recover the values for the Physical parameters with their corresponding errors.

Next sections cover:

1. Building whitening filters
2. Linear Parameter Estimation

Building whitening filters

Whitening filters can be constructed from power spectral density (PSD) of noise only data series. The procedure can be summarized in few steps:

1. Get noise only data
2. Etimate PSD
3. Obtain a smoothed version of your PSD estimation
4. Use the smoothed PSD to calculate the whitening filter

The process starts with the doenload of noise only data.

```
noise = matrix('noise.mat');
```

Assuming the noise is stationary we calculate whitening filter from the noise only part of the first experiment.

```
[n1, n2] = noise(1).unpack;
```

Whitening filter will be calculated from the noise spectra. We assume that the correlations between the two data series are so low that they cannot be properly identified by a cpsd estimation. Therefore we calculate two independent whitening filters for the two output channels. This is equivalent to assume uncorrelated data.
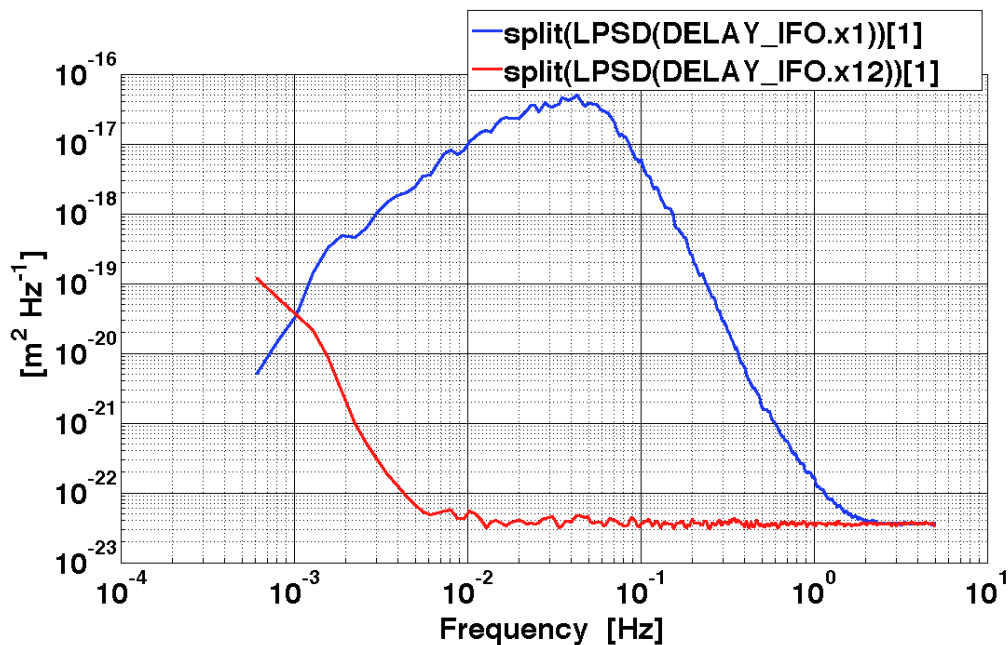
```
n1x = n1.lpsd;
n2x = n2.lpsd;
```

We are going to split in order to remove low frequency window corrupted data-points,

```
n1xx = split(n1x, plist('samples', [6 numel(n1x.y)]));
n2xx = split(n2x, plist('samples', [6 numel(n2x.y)]));
```

and finally plot to check results.

```
iplot(n1xx, n2xx)
```

Since the amount of data is limited, we do not properly cover low frequency with the estimated spectrum. We need to exted low frequency part in order to avoid that the fit would introduce unwanted features in a sensible frequency range (e.g. if we skip this part we end-up with a notch on channel 1 that will be clearly visible in the whitened time series).

```
xt = [logspace(-5, log10(4.5e-4), 60)]';

level1 = n1xx.y(1) .* 0.90;
yt1 = ones(numel(xt),1) .* level1;

level2 = n2xx.y(1) .* 3.00;
yt2 = ones(numel(xt),1) .* level2;
```
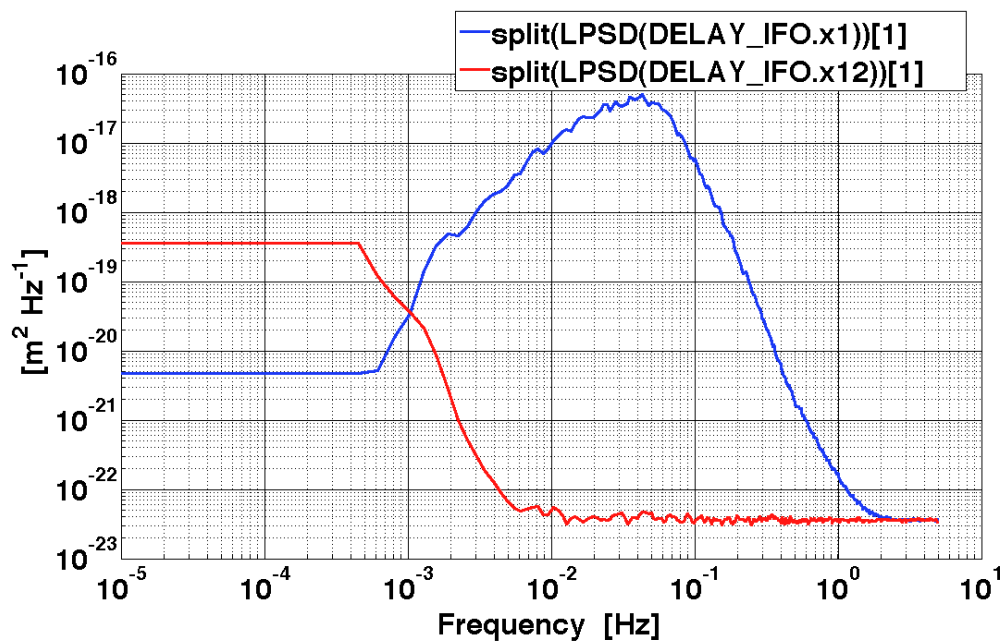
and generate an AO from them.

```
yunit  = n1xx.yunits;
fs     = n1xx.fs;
ext1   = ao(plist('XVALS', xt, 'YVALS', yt1, 'TYPE', 'FSDATA', 'YUNITS', yunit, 'FS',
fs));
ext2   = ao(plist('XVALS', xt, 'YVALS', yt2, 'TYPE', 'FSDATA', 'YUNITS', yunit, 'FS',
fs));
```

Original data and extension can be joint now.

```
nn1xx = join(ext1, n1xx);
nn2xx = join(ext2, n2xx);
```

The resulting spectrum che be checked with a plot.

```
iplot(nn1xx, nn2xx)
```

In order to obtain a smooth version of the calculated PSD we perform a frequency domain fit with the 'ao' class method 'sDomainFit', which output a 'parfrac' object that can be used to define the input for our whitener.

```
        plfit = plist(...
                'AUTOSEARCH', 'on', ...
                'STARTPOLESOPT', 'clog', ...
                'maxiter', 30, ...
                'minorder', 4, ...
                'fittol', 1e-2, ...
                'msevartol', 5e-1, ...
                'plot', 'off' );
        s1 = sDomainFit(nn1xx,plfit);


        plfit = plist(...
                'AUTOSEARCH', 'on', ...
                'STARTPOLESOPT', 'clog', ...
                'maxiter', 30, ...
                'minorder', 3, ...
                'fittol', 1e-2, ...
                'msevartol', 5e-1, ...
                'plot', 'off' );
        s2 = sDomainFit(nn2xx,plfit);
```

Frequency series 'aos' can be constructed from the 'parfrac' objects. They will be used as input to 'buildWhitener'.

```
    plr  = plist('f',logspace(-5, log10(5), 5000)); % define plist with frequency grid

    rs1  = s1.resp(plr); % get frequency response from parfarc object
    mod1 = abs(rs1); % this is a model for a psd, so it must be real

    rs2  = s2.resp(plr); % get frequency response from parfarc object
    mod2 = abs(rs2); % this is a model for a psd, so it must be real

    iplot(mod1,mod2)
```
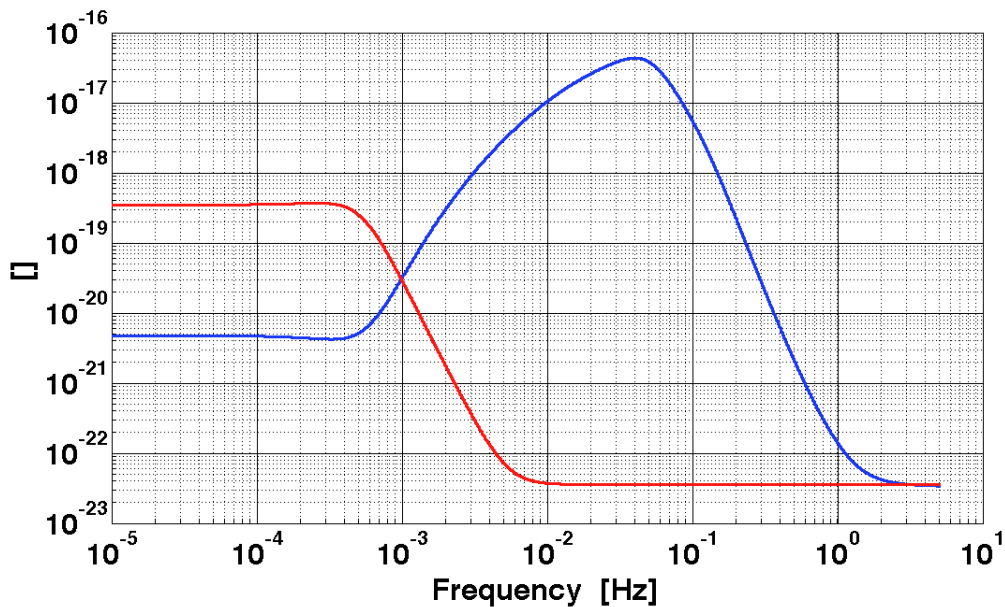
We are now ready to build the whitening filters for the first and differential channels. 'buildWhitener', build a whitening filter from a frequency series input and a 'plist'. The frequency series 'ao' should be representative of the one-sided psd of the noise we want to whiten. 'buildWhitener' perform a fit in the z-domain to the data in order to identify the coefficients (poles and residues) of a digital filter.

```
plw = plist(...
       'fs', fs,...
       'FITTOL', 1e-3);

w1 = buildWhitener1D(mod1, plw);
w2 = buildWhitener1D(mod2, plw);
```

If you want more information about 'buildWhitener options, then type
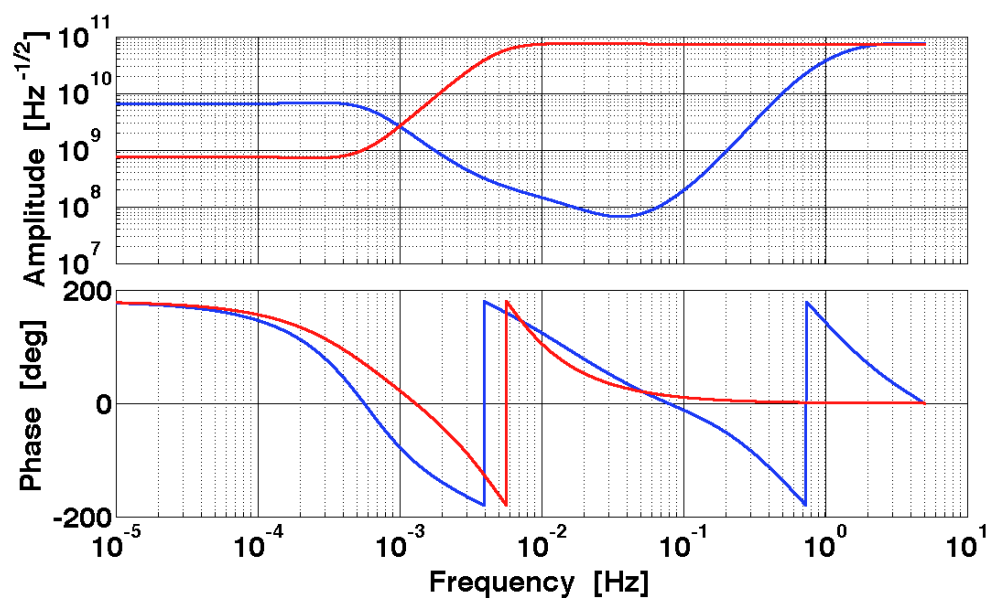
```
help ao/buildWhitener1D
```

in MATLAB command window.

Once we have the whitening filters, we can check their frequency response. It is good practice to ceck filters response on a frequency grid tighter than that used for buildind the filter.

```
plr = plist('f', logspace(-5, log10(5), 10000));

rw1 = w1.resp(plr);
rw2 = w2.resp(plr);

iplot(rw1,rw2)
```
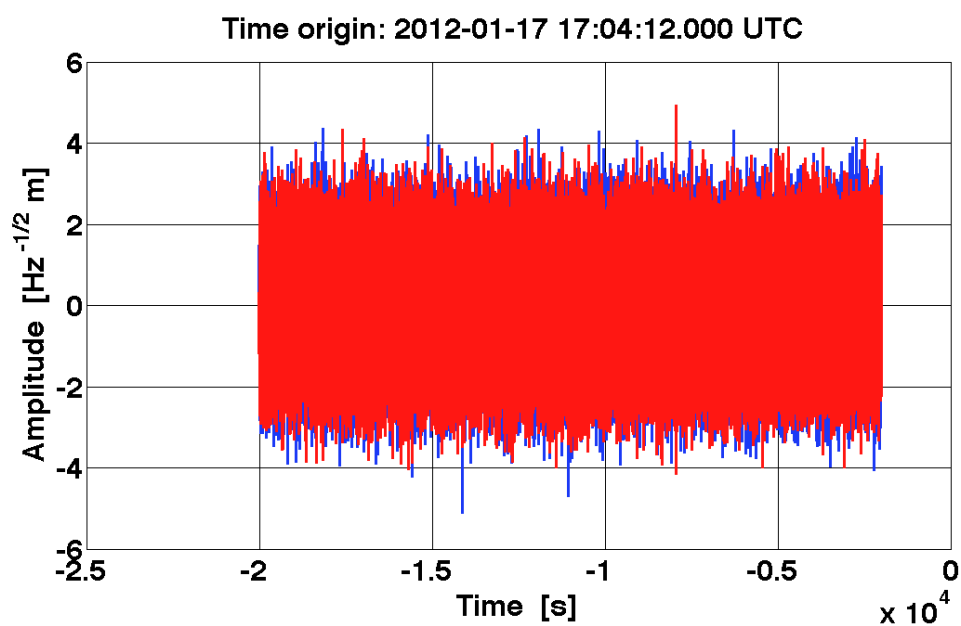
It is also interesting to check the behavior of the whitening filters on the noise time series.
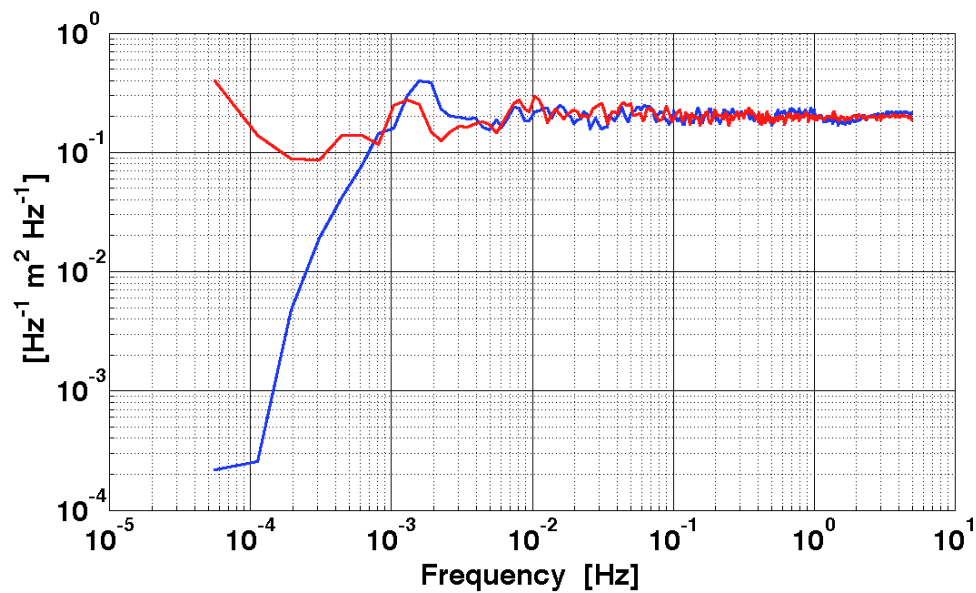
```
nw1 = filter(n1, w1);
nw2 = filter(n2, w2);

iplot(nw1, nw2)
```



```
nw1xx = lpsd(nw1);
nw2xx = lpsd(nw2);

iplot(nw1xx, nw2xx)
```

As can be seen, the whitening is not perfect expecially at low frequency but enough to perform a meaningful fit.

Once we are happy with our whitening filters, we can pack them in a matrix (diagonal) and save it. In order to have a diagonal filter matrix we will put empty filter objects out of the diagonal.

```
% out of diagonal filter
fil = filterbank(miir());

% build a matrix with whitening filters elements
wf = matrix(w1, fil, fil, w2, plist('shape', [2 2]));

wf.save('whitening_filter.mat');
```

◀ Linear Parameter Estimation with Singular Value Decomposition   Linear Parameter ▶ Estimation

©LTP Team

Linear Parameter Estimation

At this point we have all the data we need to perform a linear fir to the IFO output data. We start by loading the data we need that we have produced in the previous sections. In particulr:

1. IFO signals
2. Inputs for the experiments
3. Whitening filters
4. Fit model

```matlab
odat = matrix('output.mat');
idat = matrix('input.mat');
wf   = matrix('whitening_filter.mat');
H    = ssm('fitting_model.mat');
```

Then we define input ports, output ports and parameters names.

```matlab
% define input port-names for the different experiments
InputNames = {{'GUIDANCE.ifo_x1'}, {'GUIDANCE.ifo_x12'}};

% define output port-names for the different experiments
OutputNames = {{'DELAY_IFO.x1', 'DELAY_IFO.x12'}, {'DELAY_IFO.x1','DELAY_IFO.x12'}};

% parameters names
params = {'FEEPS_XX', 'CAPACT_TM2_XX', 'IFO_X12X1', 'EOM_TM1_STIFF_XX',
'EOM_TM2_STIFF_XX'};
```

Then we can define the input 'aos' for the different experiments and the step that will be used for numerical differentiation. Such step is chosen as the 1% of the nominal parameters values.

```matlab
% Input signals
is1ao = idat(1).getObjectAtIndex(1); % extract the AOs
is2ao = idat(2).getObjectAtIndex(2);
iS    = collection(is1ao, is2ao); % put inputs inside a collection

% set numerical derivative step as 1% of the nominal values
diffStep = [1, 1, 1e-4, 1.935e-06, 2.0e-6] .* 0.01;
```

We have all the data we need for starting the fit. We start defining a 'plist' with all the parameters we need for the current fit session.

```matlab
plfit = plist(...
        'FitParams', params ,...
        'diffStep', diffStep, ...
        'Model', H, ...
        'Input', iS, ...
        'INNAMES', InputNames, ...
        'OUTNAMES', OutputNames, ...
        'WhiteningFilter', wf, ...
        'tol', 1, ...
        'Nloops', 10, ...
        'Ncut', 1e3);
```
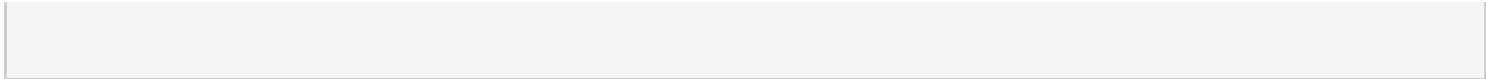
Ready to fit now!

```
fpars = linfitsvd(odat, plfit);
```

If you want more information about 'linfitsvd' you have to type in MATLAB command line:

```
help matrix/linfisvd
```

With a click on the link you get the full list of parameters and a brief explanation. In particular, for the parameters we are currently interested the list read:

Default			
no description			
Key	**Default Value**	**Options**	**Description**
MODEL	[]	*none*	System model. It have to be parametric. A matrix of smodel objects or a ssm object
INNAMES	{} [0x0]	*none*	A cell array containing cell arrays of the input ports names for each experiment. Used only with ssm models.
OUTNAMES	{} [0x0]	*none*	A cell array containing cell arrays of the output ports names for each experiment. Used only with ssm models.
FITPARAMS	{} [0x0]	*none*	A cell array with the names of the fit parameters
INPUT	[]	*none*	Collection of input signals
WHITENINGFILTER	[]	*none*	The multichannel whitening filter. A matrix object of filters
NLOOPS	1	*none*	Number of desired iteration loops.
NCUT	100	*none*	Number of bins to be discharged in order to cut whitening filter transients
TOL	1	*none*	Convergence threshold for fit parameters
DIFFSTEP	[]	*none*	Numerical differentiation step for ssm models

In particular the parameter 'TOL' is the variable controlling fit convergence. 'TOL' define the ratio between the square of the parameters increment and corresponding parameters variance.

Where parameter increment is the difference between a parameter value after and before the fit. If all ratios are lower than 'TOL' then fit convergence is declared and loop is stopped. 'linfitsvd' provides at the output the parameter set that is minimizing Mean Squared Error over the current loop. Output is packed in a 'pest' object.

We are now ready to save our fit results.

```
fpars.save('fit_linear.mat');
```

◀ Building whitening filters   Results and Comparison ▶

©LTP Team

Results and Comparison

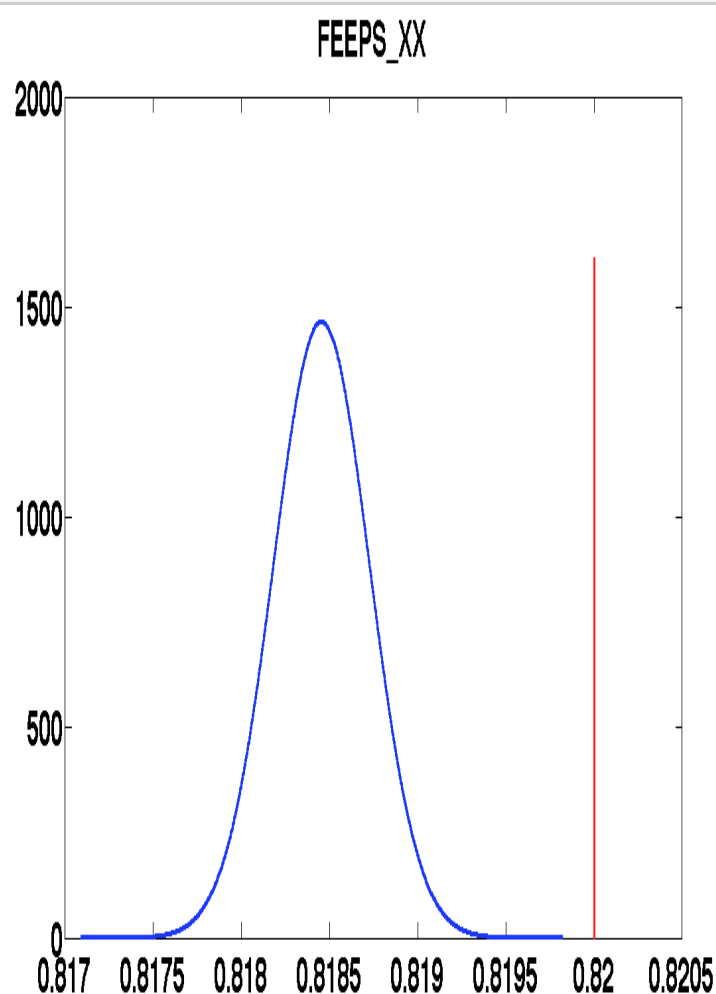## Results for the Linear Parameter Estimation

Results of linear parameter estimation can be visualized with the method 'linfitsvdPlot'
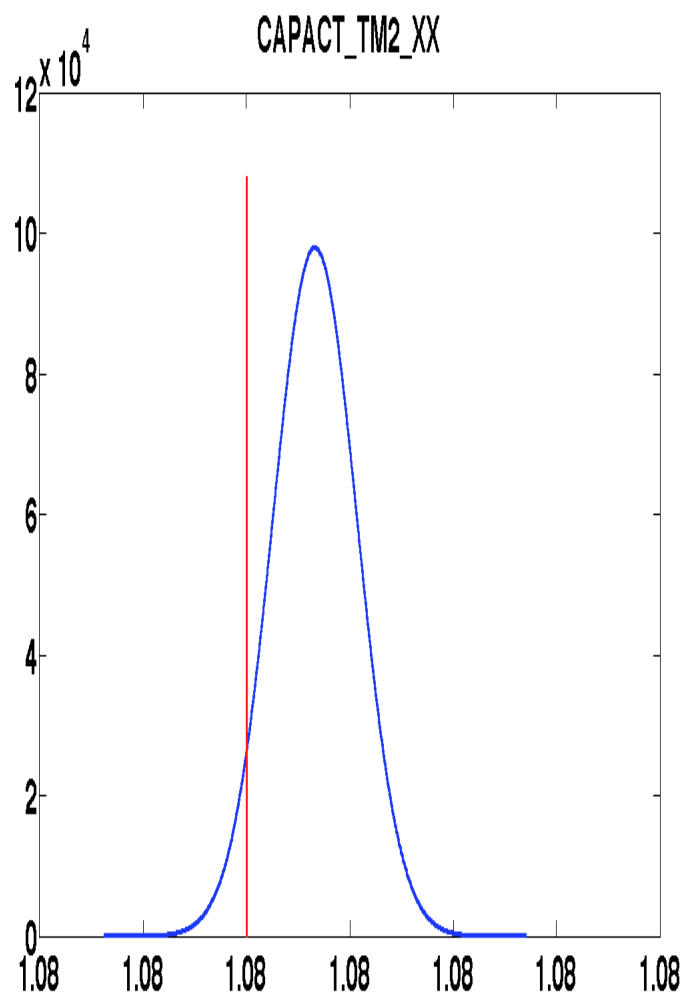
```
params = {...
      'FEEPS_XX', ...
      'CAPACT_TM2_XX', ...
      'IFO_X12X1', ...
      'EOM_TM1_STIFF_XX', ...
      'EOM_TM2_STIFF_XX'};

values = [0.82 1.08 0.0004 1.3e-6 1.9e-6];

plshow = plist(...
      'FitParams', params,...
      'FitParamsValues', values);

linfitsvdPlot(fpars, plshow);
```
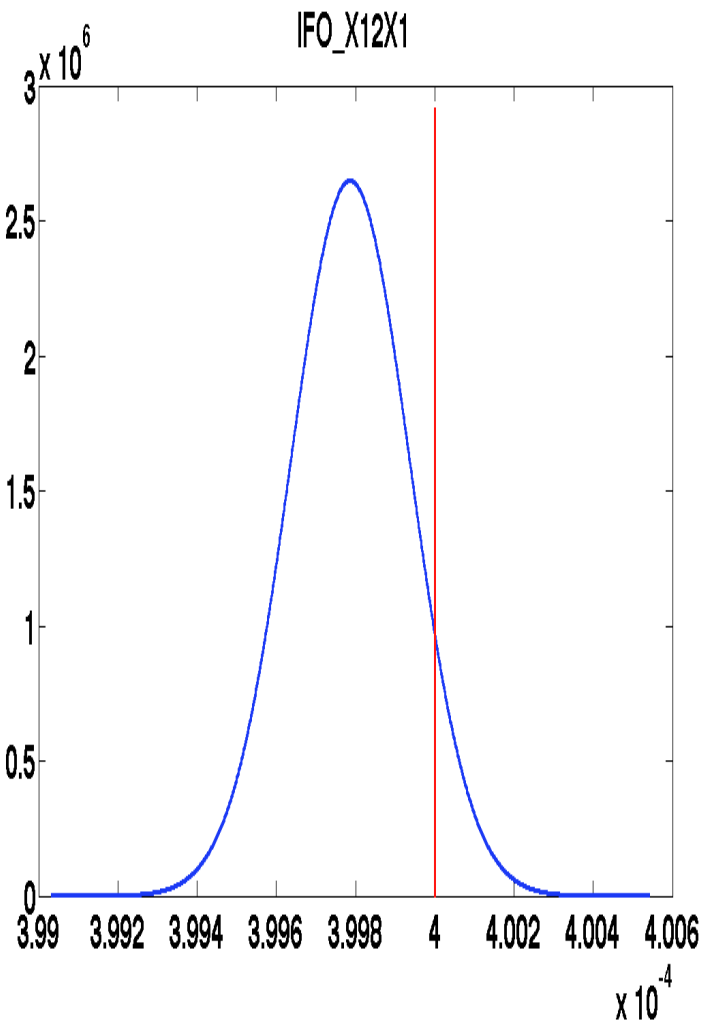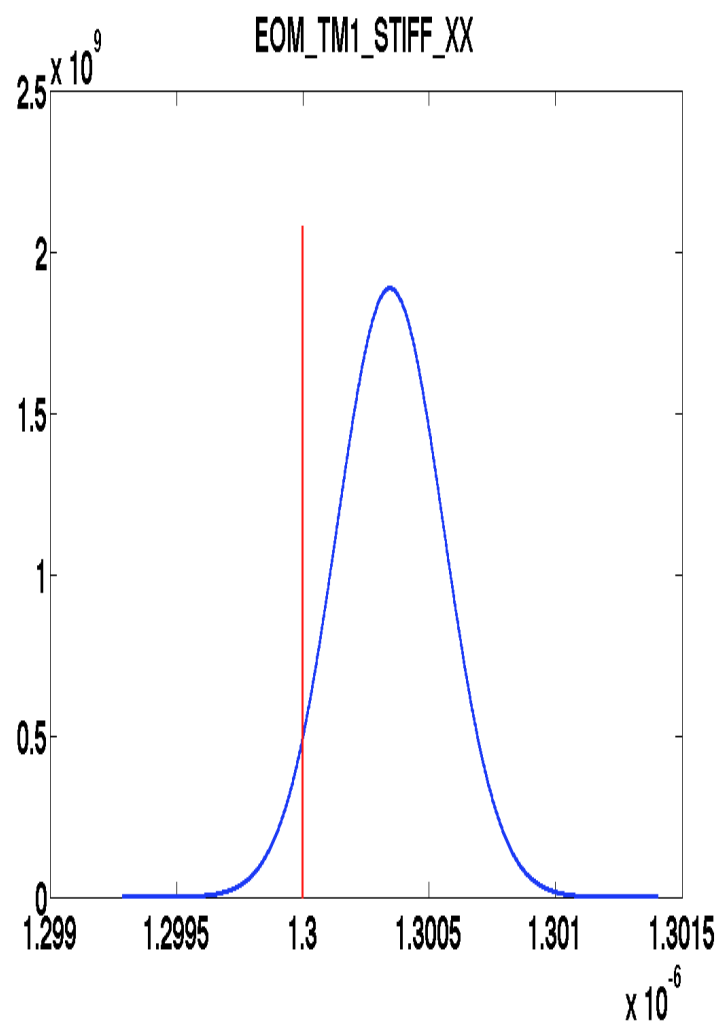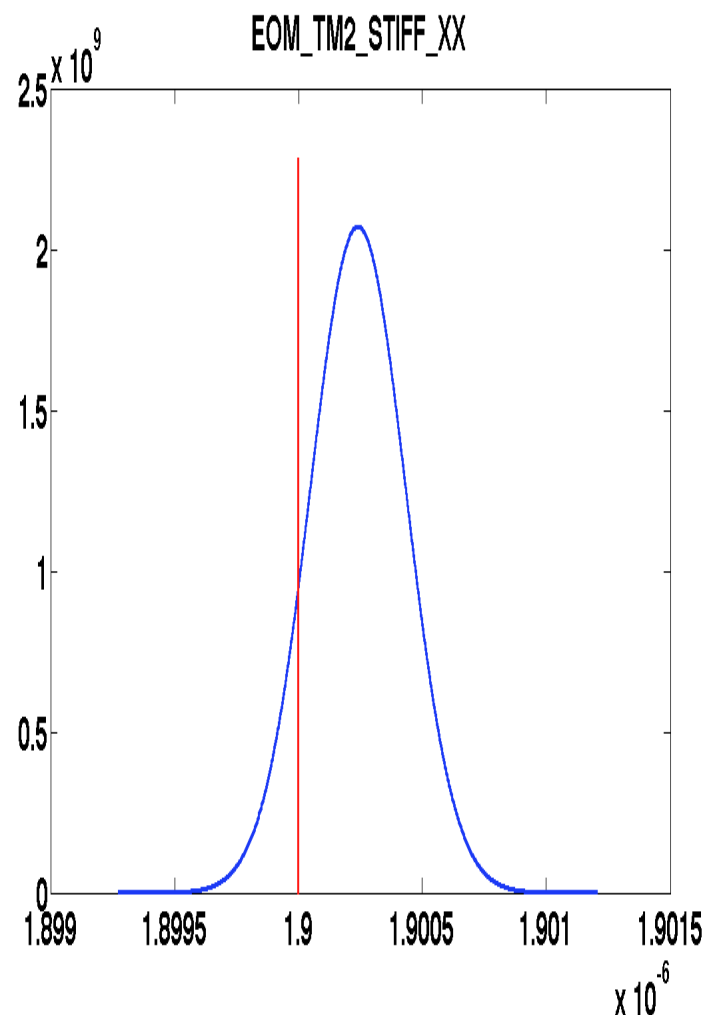
CAPACT_TM2_XX

IFO_X12X1

EOM_TM1_STIFF_XX

Results for the MCMC

If our `pest` object obtained with MCMC is `b`, then we can plot the results, or print them to screen with the following piece of code:

```
pl = plist(...
        'burnin', 1000, ...                    % The number of samples to be discarded from
the chain
        'pdfs', true, ...                      % Plot the pdfs of the parameters
        'chain', 0 ,...                        % Do not plot chains
        'separate pdfs', [1 2 3 4 5], ...      % Plot the pdfs in separate figures
        'nbins', 20, ...                       % Number of bins for the histograms
        'colormap', summer, ...                % choose colormap for cosmetics!
        'results', true);                      % Print results to screen

mcmcPlot(b,pl)
```

Note: Here we assumed that we used the `simplex` algorithm before the mcmc sampling as in section 5.6.1. The PDFs of the parameters will look like the following:

CAPACT TM2 XX
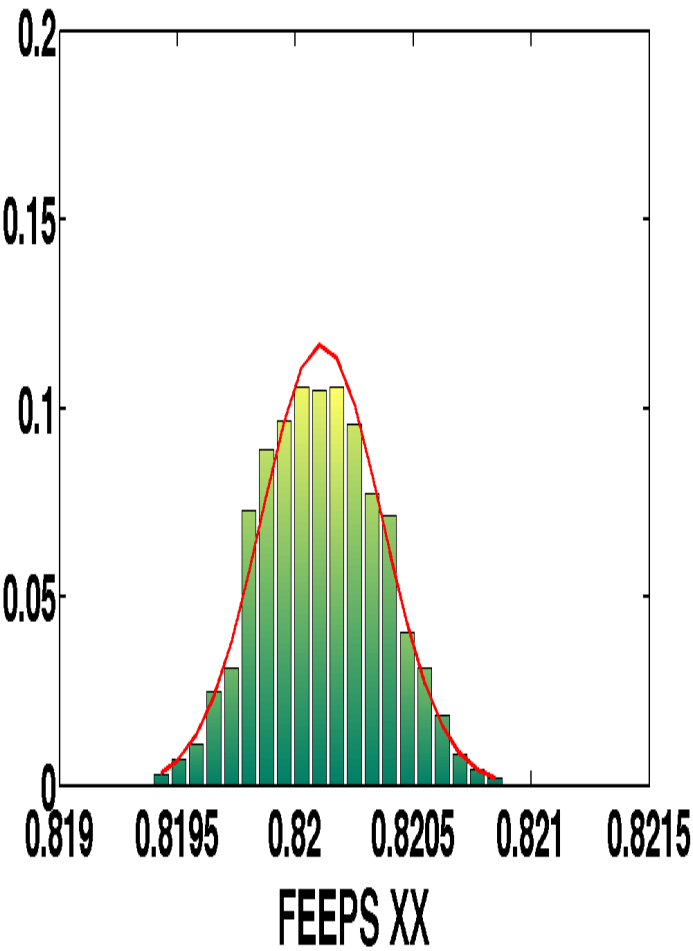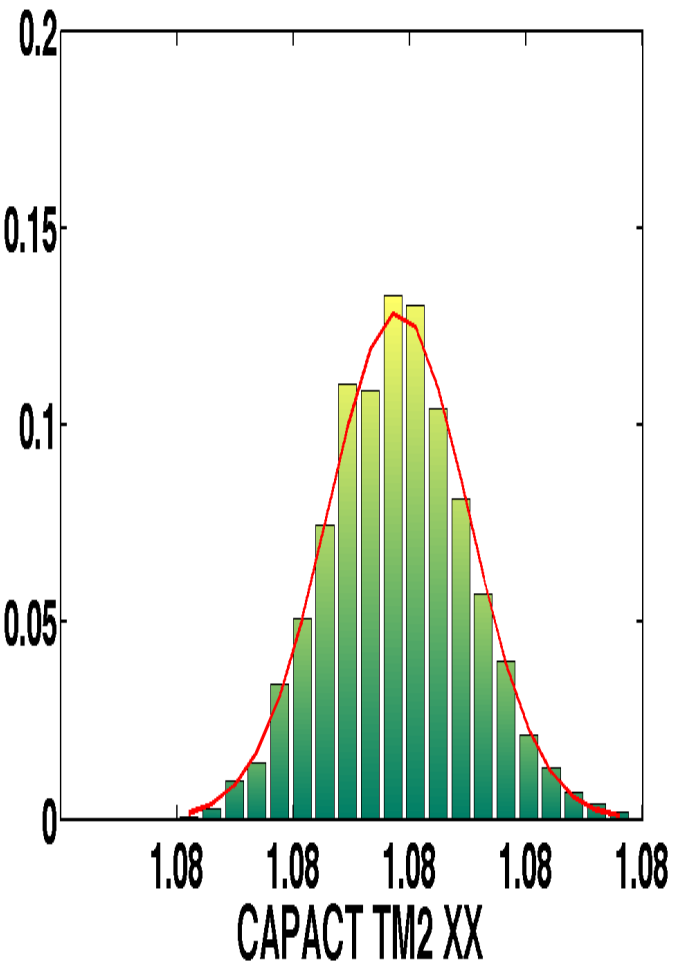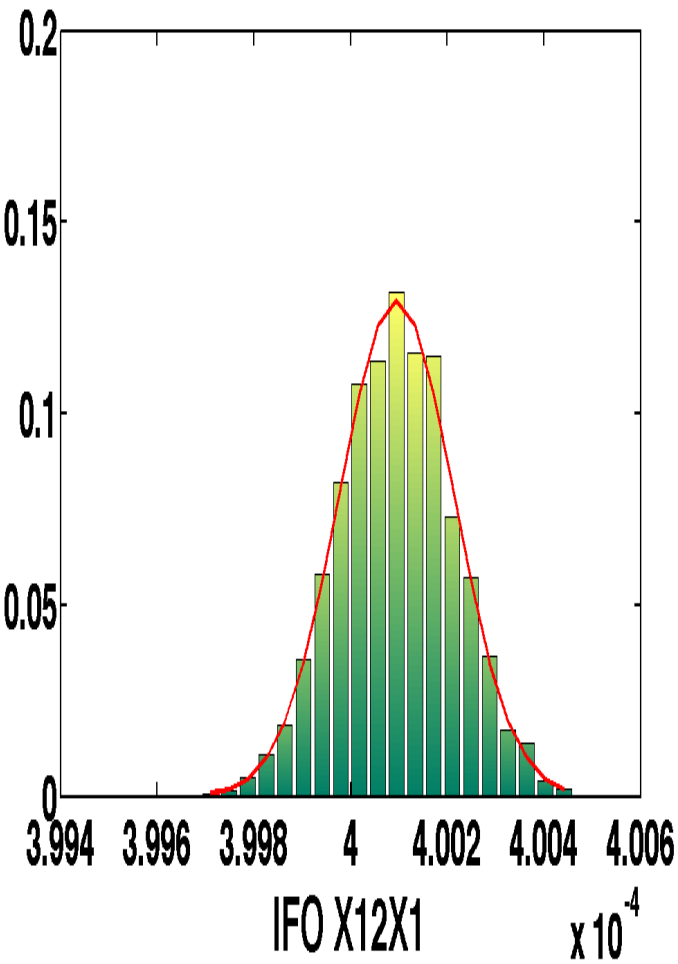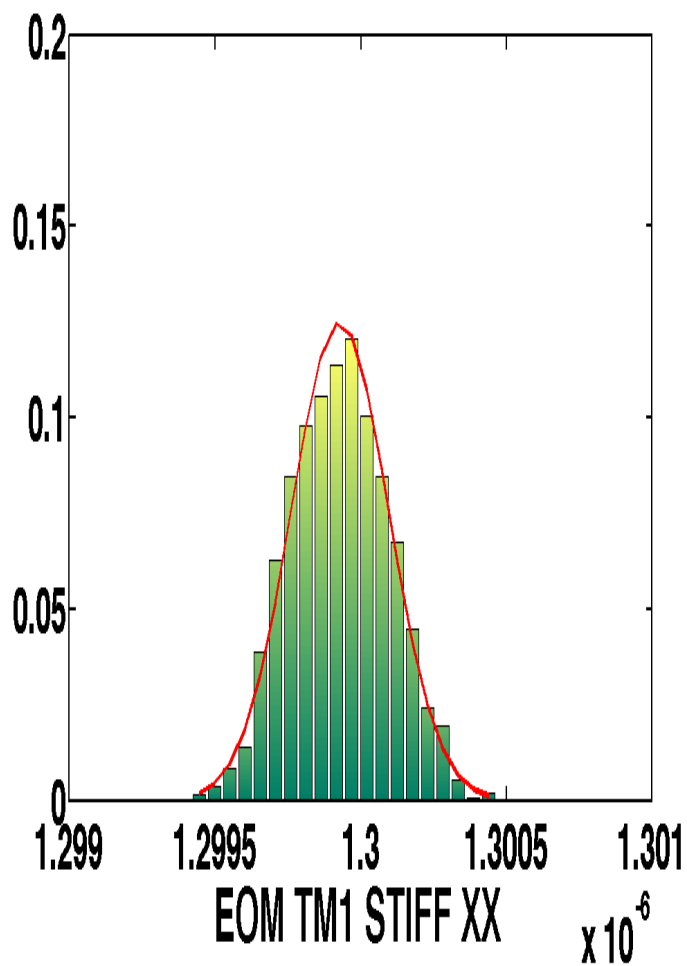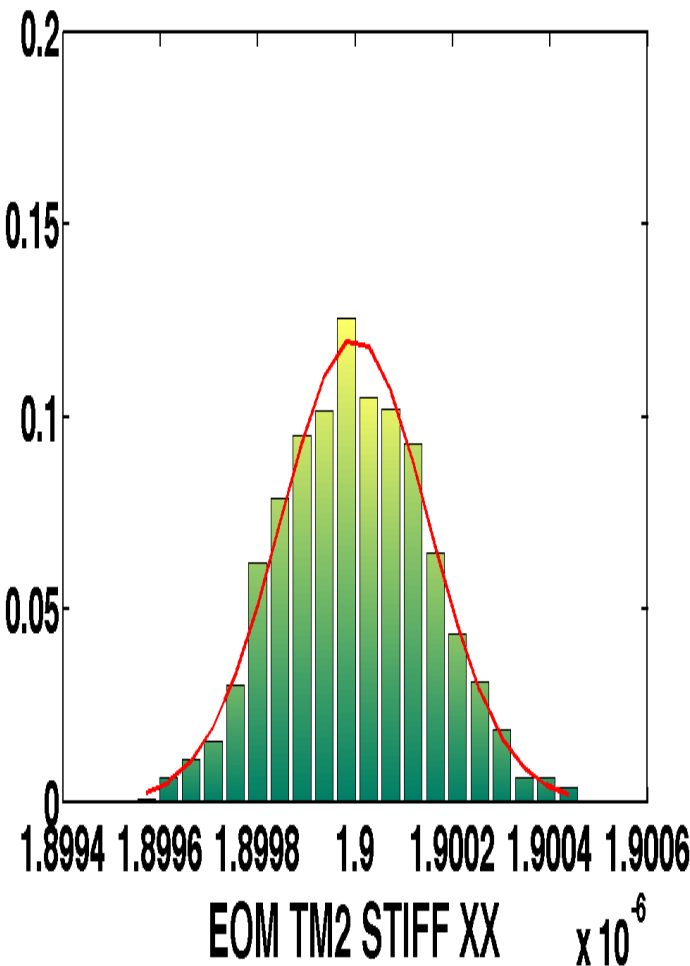
Comparison

The table below show the comparison of the results obtained with the two methods. We compare as well the error on the parameter with the optimal error expected from the Fisher matrix analysis

Parameter	True value	Exp. error	Linear	MCMC
FEEPS_XX	0.82	0.0002	0.8185 ± 3e-4	0.8201 ± 2.5e-04
CAPACT_TM2_XX	1.08	3.0e-06	1.080007 ± 4e-06	1.080004 ± 3e-06
IFO_X12X1	0.0004	1.1e-07	3.998e-04 ± 1.5e-07	4.000e-04 ± 1e-07
EOM_TM1_STIFF_XX	1.3e-6	1.7e-10	1.3004e-06 ± 2.0e-10	1.2999e-06 ± 1.6e-10
EOM_TM2_STIFF_XX	1.9e-6	1.6e-10	1.9002e-06 ± 2.0e-10	1.8999e-06 ± 1.5e-10

Use parameter estimates to estimate residual differential acceleration

Our last step once we have recovered the correct parameters for our system is to translate the interferometer displacement noise to accelerometer noise, as previously shown. We will need first to load the objects that we have previously created. In this exercise, we will take the initial noise segment and translate into acceleration noise.

```
%% Load objects

noise  = matrix('noise.mat');
mdl    = ssm('fitting_model.mat');
Fcmd   = matrix('cmdForces.mat');
params = pest('fit_mcmc.mat');
```

Since our objects contain two experiments, we will focus on one of them, the second experiment to do this analysis.

```
%% Focus on noise for 2nd experiment

[o1_0002, o12_0002]  = unpack(noise(2));
[Fsc_0002, Ftm2_0002] = unpack(Fcmd(2));
```

As shown in Topic 3, the <>ltp_ifo2acc<> method requires a mapping of parameters because a change of notation.

```
%% Parameters mapping

% Define new variables with the name of the parameters used in ltp_ifo2acc
Gdf  = params.find('FEEPS_XX');
Gsus = params.find('CAPACT_TM2_XX');
SD1  = params.find('IFO_X12X1');
w1   = -1*params.find('EOM_TM1_STIFF_XX'); % use a different convention from ssm so the -
1
w2   = -1*params.find('EOM_TM2_STIFF_XX'); % use a different convention from ssm so the -
1

%% Conversion to acceleration

% set the input plist for conversion to acceleration
pli2a = plist(...
        'Gdf', Gdf, ...
        'Gsus', Gsus, ...
        'SD1', SD1, ...
        'w1', w1, ...
        'w2', w2, ...
        'Hdf', Fsc_0002, ...
        'Hsus', Ftm2_0002);
```

Now this `plist` allows us call the method to get the acceleration noise

```
%% Convert to acceleration
```

```
        % ifo2acc with commanded forces
        [a1, a12] = ltp_ifo2acc(o1_0002, o12_0002, pli2a);
```

We will remove some samples from the last part of the acceleration noise time series to avoid a transient that appears in the x1 channel.

```
        %% set paremeters for lpsd
        plpsd = plist('times', [-inf -2010]);

        a1s  = split(a1, plpsd);
        a12s = split(a12, plpsd);

        iplot(a1, a1s)
```
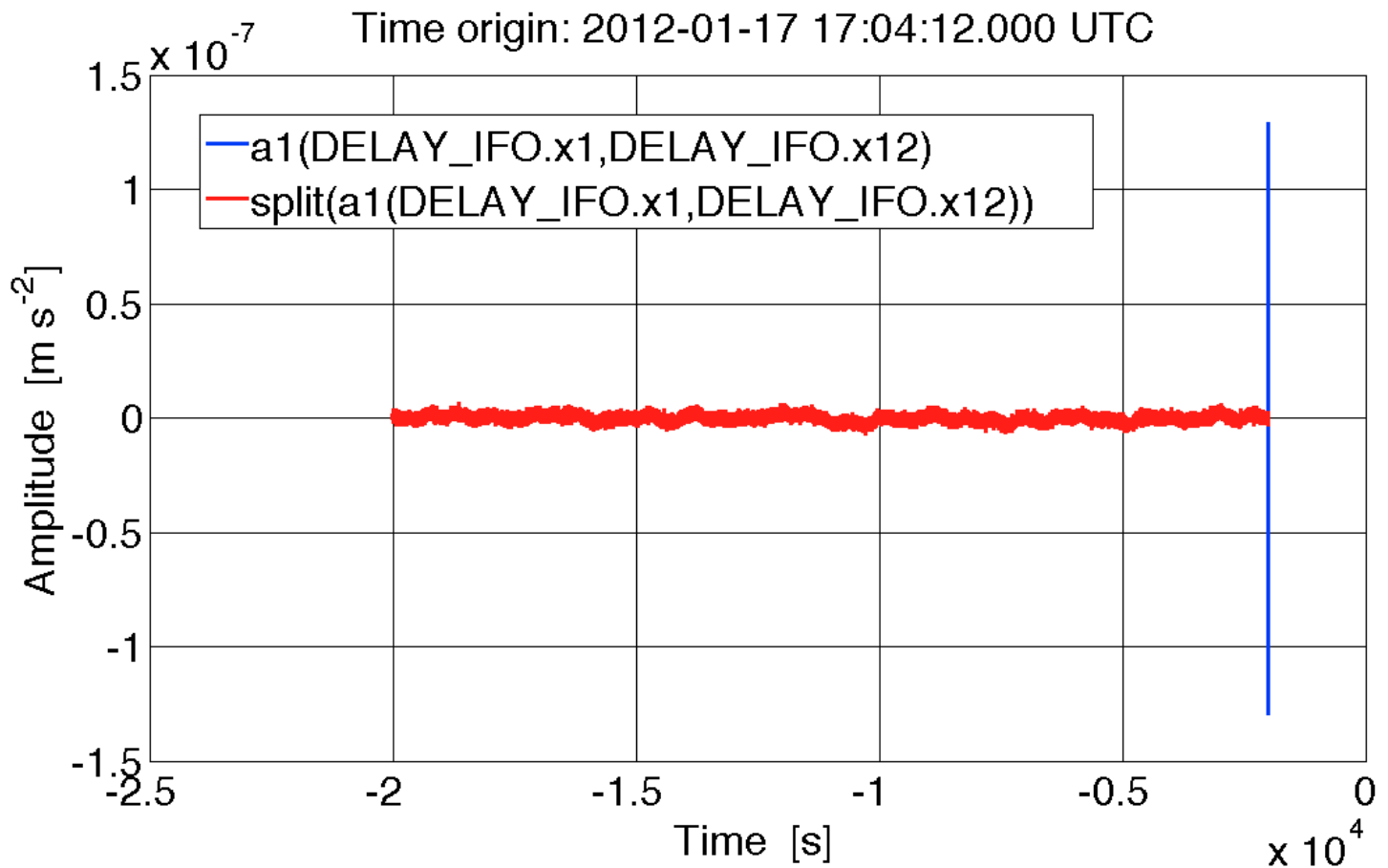

And compute the Amplitude Spectral Density (ASD) for the remaining time series.

```
        %% Calculate Amplitude Spectral Density

        % set paremeters for lpsd
        plpsd = plist(...
                'order', 1, ...
                'SCALE', 'ASD');

        a1xx  = lpsd(a1s, plpsd);
        a12xx = lpsd(a12s, plpsd);
```

Which leads to our final result: the perfomance of the instrument in acceleration noise.

```
        %% Plot results
        plplot = plist(...
```

Use parameter estimates to estimate residual differential acceleration (LTPDA Toolbox)

```
                       'Linecolors', {'k', 'r'}, ...
                       'LineStyles', {'-', '--'}, ...
                       'LineWidths', {3, 3}, ...
                       'XLABELS', {'All', 'Frequency'}, ...
                       'YLABELS', {'All', 'Spectral Density'});

          % Plot results
          a1xx.setName('accel(DELAY_IFO.x1)')
          a12xx.setName('accel(DELAY_IFO.x12)')

          iplot(a1xx, a12xx, plplot)
```
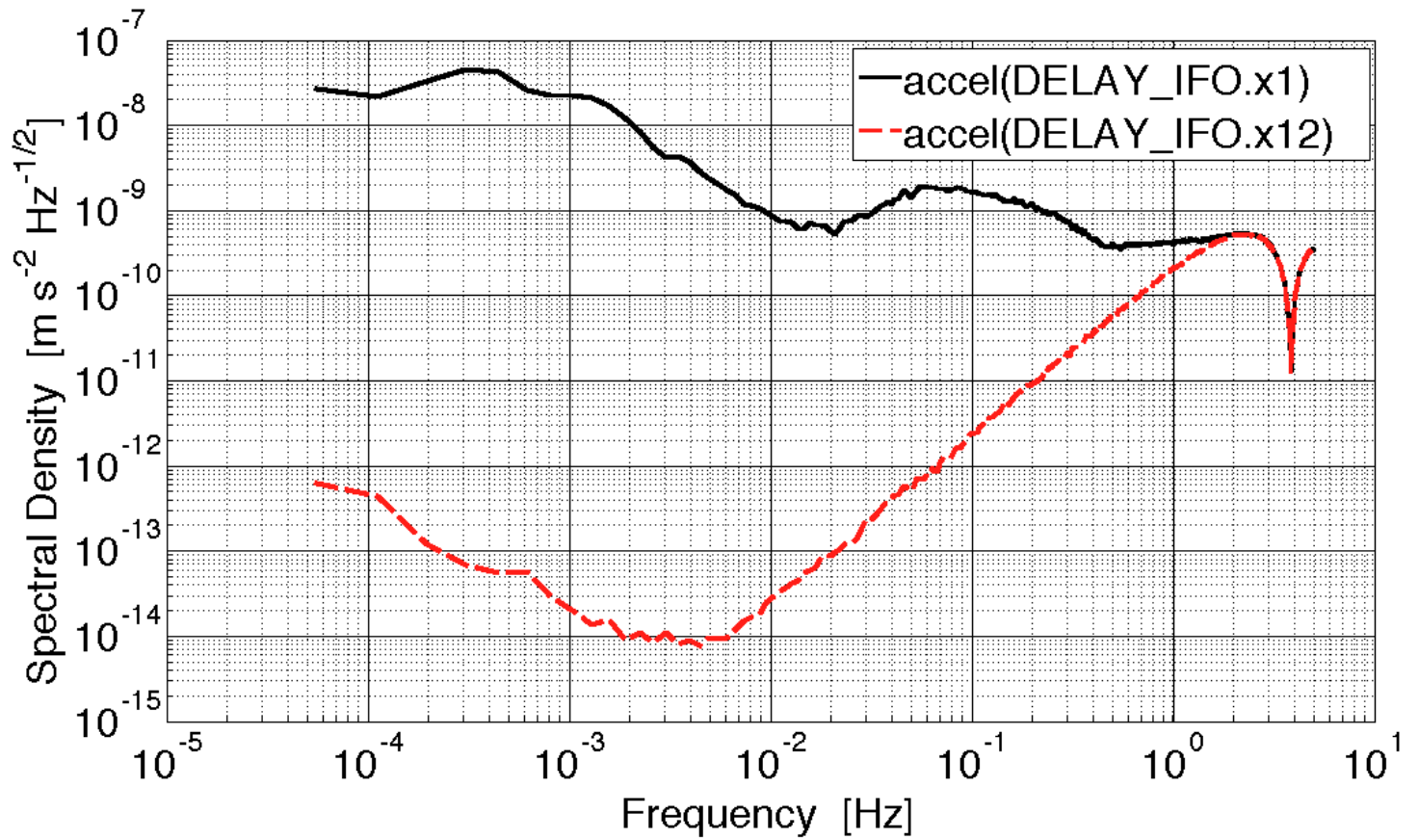
©LTP Team

Examples

General

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

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

Constructor examples

[Constructor examples of the AO class](#)
[Constructor examples of the MFIR class](#)
[Constructor examples of the MIIR class](#)
[Constructor examples of the PZMODEL class](#)
[Constructor examples of the PARFRAC class](#)
[Constructor examples of the RATIONAL class](#)
[Constructor examples of the TIMESPAN class](#)
[Constructor examples of the PLIST class](#)
[Constructor examples of the SPECWIN class](#)

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

See the Constructor Helper GUI documentation for more information.

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Release Notes

Summary by Version

This table provides quick access to what's new in each version. For clarification, see Using Release Notes.

Version (Release)	New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Latest Version LTPDA V2.8	No Details	No	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.7.1	No Details	No	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.7	Yes Details	No	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.6.1	No Details	No	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.6	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.5.5	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.5.4	Yes Details	No	Bug Reports at Web site	Printable Release Notes: Not available
LTPDA V2.5.3	Yes Details	No	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.5.2	Yes Details	No	No	Printable Release Notes: PDF
LTPDA V2.5.1	Yes Details	No	No	Printable Release Notes: PDF
LTPDA V2.5	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF

LTPDA V2.4	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.3.1	Yes Details	No	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.3	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.2	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.1	Yes Details	Yes Summary	Bug Reports at Web site	Printable Release Notes: PDF
LTPDA V2.0.1	Yes Details	No	No	Printable Release Notes: PDF
LTPDA 2.0	Yes Details	No	Bug Reports at Web site	Printable Release Notes: PDF

Using Release Notes

Use release notes when upgrading to a newer version to learn about:

- New features
- Changes
- Potential impact on your existing files and practices

Review the release notes for other MathWorks™ products required for this product (for example, MATLAB$^{®}$ or Simulink$^{®}$). Determine if enhancements, bugs, or compatibility considerations in other products impact you.

If you are upgrading from a software version other than the most recent one, review the current release notes and all interim versions. For example, when you upgrade from V2.5.4 to V2.6, review the release notes for V2.5.5 and V2.6.

◀ Examples   Version 2.8 LTPDA Toolbox Software ▶

©LTP Team

Version 2.8 LTPDA Toolbox Software

This table summarizes what's new in Version 2.8:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [Known Issues](#)
- [Main Changes](#)
  - [New Methods](#)
  - [Changes to methods](#)
- [MANTIS Issues Resolved](#)
- [Detailed changes since v2.7.1](#)

Introduction

This version of LTPDA is 2.8. This document lists the changes since V2.7.1.

This version requires MATLAB 2011a or above.

Known Issues

On Mac OS X versions which use java 1.7 (on newer versions of MATLAB) there is an issue with the LTPDAprefs GUI when selecting paths of extension modules. There is a bug in the underlying java file-chooser widget which means there is an 'Open' button instead of a 'Choose' button. The result is that you can't select a directory but must, instead, select a file (for example the .xml file in the extension) then delete the filename from the path before pressing the + button to add the extension.

Main Changes

This version of LTPDA contains many under-the-hood improvements, particularly in the area of speed, and also some changes to the underlying data structures for analysis objects.

In addition, a number of deprecated GUI tools (including the workbench) have been removed from this version. These have been unsupported for a few versions now, and continued maintenance is no longer possible. The user manual, and in particular, the training sessions, have been updated to reflect this. Some of the functionality pertaining to the repository interaction, has been grouped together into a single GUI called **LTPDARepositoryQuery**.

New Methods

- **ao/polyfitSpectrum** A new tool which offers an alternative way to fit a smooth model to a spectrum. The method works by fitting a polynomial to the log of the x and y data in the input spectrum.

- **timespan/tohtml** displays an array of time-span objects in an HTML table.
- New setter methods for AOs: **ao/setEnbw**, and **ao/setNavs**.
- There is a new AO built-in model for building pulse-trains (**help ao_model_pulsetrain**).
- **ao/edgedetect** is a new method which detects edges in a binary pulse train.
- **ao/pad** allows padding an AO with a particular value.
- **ao/cast** allows you to cast the values in an AO to another data type.
- **matrix/double** returns a matrix of doubles created from the internal objects.
- **ao/simplifyZunits** simplifies the z-units of the AO.
- **ao/simplifyXunits** simplifies the z-units of the AO.
- **timespan/merge** can merge multiple timespans into one.
- **ao/detectOutliers** detects outliers in a time-series.

Changes to methods

- **ao/iplotyy** has been rewritten to allow for many new features, including plotting of multiple objects on each of the two axes.
- In all AO constructors, you can now specify a 'timespan' plist key, passing a timespan object which will then be used to split the data before returning.
- Reading from data files with the AO constructor now allows for some header lines to be skipped.
- AO linear fitting methods have improved error estimates.
- **pest/jtable** displays the values in the parameter estimate object in a table window.
- **pest/table** displays a **pest** object at the MATLAB command prompt.
- We have new methods to set the marker face and edge colour on all user objects. We also have a method to set if the marker shouldd be filled or not.
- **ao/iplot** subplots are now linked when zooming.
- **ao/dft** now estimates errors based on the frequencies surrounding the target frequency.

MANTIS Issues Resolved

Issue	Type	Decription
0000748	Bug Report	iplot fails for bar plots
0000747	Change Request	giving a name to a figure
0000744	Change Request	An option in ao/consolidate not to be truncating to the shortest data set
0000737	Change Request	skip the printout of the error and units titles when these are empty.
0000732	Change Request	Giving a 'name' to the window of an iPlot
0000731	Bug Report	iplot not showing titles in fsdata plot
0000727	Change Request	A search method for collection objects
0000726	Change Request	A method to join/merge timespan objects
0000722	Change Request	toSI converts [Hz] into [s]
0000720	Bug Report	warnings from bin_data

0000719	Change Request	Remove unused/old/broken GUIs
0000715	Bug Report	built-in model information only contains 1st version
0000714	Change Request	showErrors field in the plotinfo objects is ignored by ao.iplot
0000713	Bug Report	The plotinfo constructor via plist is incomplete
0000708	Change Request	iplotyy should accept a parameter to change the axis range
0000707	Bug Report	iplotyy should accept a vector of two objects
0000706	Change Request	ao.lincom should be implemented as a wrapper of ao/plus and ao/times
0000705	Bug Report	Bug division and multiplication uncertainty propagation
0000704	Bug Report	ao/linfit does overwrites dy in input aos
0000703	Change Request	Lack of GRS electrostatic built-in model parameters
0000702	Bug Report	HetSegsFreqs does not fully support heterodyne with cuts
0000701	Bug Report	Phi torque estimation uses wrong default parameters for stiffness
0000700	Bug Report	ao/spikecleaning leaves ao one sample short
0000699	Change Request	addvalue method missing
0000698	Change Request	An utility to fit spectra with polynomial low
0000697	Bug Report	Review issue with ao/split
0000696	Change Request	timespan.toHtml method needed
0000695	Change Request	ao/split to match AOs
0000694	Change Request	missing ao/simplifyXunits
0000692	Change Request	ao class supporting specification via object supporting char
0000691	Change Request	ao.split should support an array of timespan objects
0000688	Bug Report	stoc.misc.readExecutedTimeline crashes while reading 20131115_319_executed_timeline.txt
0000687	Bug Report	heterodyne and pipe.heterodyne show inconsistent behaviour
0000684	Change Request	Remove LTPDAWorkbench message handling from main toolbox
0000683	Change Request	utils.const.msg function with no \newline needed
0000681	Bug Report	parameter list help on plotIntervals do not work

0000680	Change Request	Improve utils.helper.warn and .err
0000678	Change Request	ao.linfit could be implemented as a particular case of ao.polynomfit
0000677	Bug Report	ao.polynomfit errors for the case of no errors provided
0000071	Change Request	Correct behavior of 'procinfo'

©LTP Team

LTPDA Toolbox™ [contents] ◀ ▶

Version 2.7.1 LTPDA Toolbox Software

This table summarizes what's new in Version 2.7.1:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	No	Printable Release Notes: PDF

- Introduction
- MANTIS Issues Resolved
- Complete CVS Changes

Introduction

This version of LTPDA is 2.7.1. This document lists the changes since V2.7.

This version requires MATLAB 2010a or above

Mostly changes are bug fixes due to the 4rd STOC Simulation.

◀ Version 2.8 LTPDA Toolbox Software   Version 2.7 LTPDA Toolbox Software ▶

©LTP Team

LTPDA Toolbox™ [contents](#) ◀ ➡

Version 2.7 LTPDA Toolbox Software

This table summarizes what's new in Version 2.7:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [Main Changes](#)
  - [Copying Objects](#)
  - [History context](#)
  - [Function handle objects](#)
  - [New Methods](#)
  - [Changes to methods](#)
  - [LTPDA Repository Interaction](#)
- [MANTIS Issues Resolved](#)
- [Detailed changes since v2.6.1](#)

Introduction

This version of LTPDA is 2.7. This document lists the changes since V2.6.1.

This version requires MATLAB 2010a or above.

Main Changes

This version of LTPDA contains many under-the-hood improvements, particularly in the area of speed, and also some changes to the underlying data structures for analysis objects. In addition, a new test infrastructure is under development which includes a significant number of new unit tests. As a result a large number of bug fixes have been made.

Copying Objects

In order to make deep copies of objects, users should now use the copy() method rather than the old (and confusing) copy constructor. So, instead of

```
a = ao.randn(10,1)
b = ao(a)
```

users should now do

```
b = copy(a)
```

History context

When tracking processing steps the 'context' in which the current command is executed is captured in the history structure. For example, it may capture the name of the script or

pipeline being executed. In a later release this may be used to group history steps in the history tree displays.

Function handle objects

Function handle objects have a new property 'transform' where you can specify a single or a list of functions that should be applied to the input objects prior to evaluating the function.

New Methods

- the **ao** class has a new method **subsData** which can be used to replace segments of a data series in different ways.
- the **ao** class has a new method **cgfill** which can replace segments of a data series with constrained gaussian noise generated from an input spectral model.
- the function handle class (**mfh**) has a new bounded non-linear least squares search (**fminsearchbnd**). This can be considered a beta version as it doesn't yet return error estimates.
- the **ao** class has 4 new convenient filtering methods: **lowpass**, **highpass**, **bandpass** and **bandreject**.
- there is a new **ao** built-in model which computes a set of notches in the frequency domain (**ao_model_notch**)
- the **matrix** class now supports element-wise power operations. For now it only supports for **X = matrix.^scalar**.

Changes to methods

- **ao/polyfit** now returns error estimates (assuming uncorrelated inputs)
- in **ao/iplot** the provenance info patches can now be disabled in the plist
- **ao/setPlotinfo** now accepts an empty array to allow the user to clear the **plotinfo**
- the built-in **ao** model **retrieve_in_timespan** has a new **dryrun** parameter which just reports on whether the request will result in any objects.
- **mfh/fminsearch** now returns error estimates if the original data model is input in the plist.
- **timespan/plot** has been improved to allow the boxes representing the timespans to be transparent so they can be overlaid on other data plots.
- **ao/iplotyy** has been updated to work properly with the plotinfo objects.

LTPDA Repository Interaction

This version of the toolbox has some new utilities for querying LTPDA repositories from within MATLAB.

- **utils.repository.report** dumps the records of a database to a file
- **utils.repository.existObjectInDB** checks if a given name exist in the database table objmeta.name. It is also possible to specify for AOs with time-series a time-span.

MANTIS Issues Resolved

Issue	Type	Decription
0000510	Change Request	A possibility to specify the dy field of an ao without knowing its length
0000559	Change Request	Give warning if negative samping rate is found.
	Change	

0000625	Request	More matrix methods
0000642	Change Request	Power operator for matrix objects
0000643	Bug Report	ao.setXunits rejects certain units
0000641	Bug Report	iplot returns error if objects have no dependency information
0000658	Bug Report	Iplot does not work as documented / plotinfo objects overrule manual plot settings
0000656	Change Request	ao/search should warn of failure
0000660	Change Request	A report method for get_data step of pipelines
0000662	Change Request	Warnings about telemetry parameters alias
0000668	Bug Report	Applying Yranges on multiple iplots
0000666	Bug Report	Manual is not accessible online
0000647	Change Request	Test mode in retrieve in timespan
0000665	Bug Report	ao.polyfit does not calculate errors for the estimated coefficients
0000661	Change Request	An easy way to access names of LTPDApipeline telemetry parameters
0000659	Change Request	Release notes are not accessible/file is damaged
0000650	Bug Report	Mantis doesn't allow to select product version 2.6
0000664	Change Request	pest class needs a y method
0000655	Bug Report	iplot for aos from (older version) files is broken
0000653	Bug Report	simplifyYunits does not like kg and does not scale errors

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ▶

Version 2.6.1 LTPDA Toolbox Software

This table summarizes what's new in Version 2.6.1:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	No	Printable Release Notes: **PDF**

- Introduction
- MANTIS Issues Resolved
- Complete CVS Changes

Introduction

This version of LTPDA is 2.6.1. This document lists the changes since V2.6.

This version requires MATLAB 2010a or above

Mostly changes are bug fixes due to the 3rd STOC Simulation.

◀ Version 2.7 LTPDA Toolbox Software   Version 2.6 LTPDA Toolbox Software ▶

©LTP Team

Version 2.6.1 LTPDA Toolbox Software (LTPDA Toolbox)

Version 2.6 LTPDA Toolbox Software

This table summarizes what's new in Version 2.6:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as [Deprecated and removed features](#) in descriptions of changes, below.	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [Main Changes](#)
    - [Function Handles](#)
    - [Algorithm Classes](#)
- [Other Changes](#)
- [New Methods](#)
- [Deprecated and removed features](#)
- [MANTIS Issues Resolved](#)
- [Detailed changes since v2.5.5](#)

Introduction

This version of LTPDA is 2.6. This document lists the changes since V2.5.5.

This version requires MATLAB 2010a or above.

Main Changes

This version of LTPDA includes a vast number of speed improvements and bug fixes. In addition, the following significant changes are included in this release.

Function Handles

This release of LTPDA contains a first implementation of a new user-level class for creating LTPDA function handles. This class is called mfh and is somewhat similar to MATLAB function handles except that history is tracked.

Here are some example usages:

```
%% Simple function of x
f1 = mfh(plist('func', 'x.^2', 'inputs', {'x'}));
x1 = ao(12);
o1 = f1(x1);
```

```
%% Simple function of x and y
f2 = mfh(plist('func', 'x^2 + y^2', 'inputs', {'x', 'y'}));
x = ao(2);
y = ao(3);
o2 = f2(x, y)
```

```
%% chain functions
x = ao(2);
y = ao(3);
o4 = f2(f1(x), y)
```

```
%% Use constants
a = ao.randn(10,10);
c = ao(2);
f6 = mfh(plist('func', 'x + c.*x', 'inputs', 'x', ...
'constants', {'c'}, 'constant objects', c));
o6 = f6(a)
```

```
%% Nest functions
c = ao(2);
h1 = mfh(plist('name', 'myFunc', 'func', 'x.^2', 'inputs', {'x'}));
f5 = mfh(plist('func', 'myFunc(x) + c.*x', ...
'inputs', 'x', ...
'subfuncs', h1, ...
'constants', {'c'}, 'constant objects', c));
o5 = f5(a)
```

Algorithm Classes

There has been some work to implement what we call Algorithm Classes. That is, classes which are not associated with data, but instead implement an algorithm. You then create an instance of the algorithm and use it to process data. This is particularly sensible for complex algorithms because it allows us to keep the various utility functions and tests associated with the algorithm together in one class. This version of the toolbox includes the base class for these algorithm classes, **ltpda_algorithm**, and a new implementation of the MCMC algorithm as one of these classes. We maintain backwards compatibility for the most part by creating an using an MCMC object with the **ao/mcmc** method. Nonetheless, this was a large structural change so please report any strange behaviour.

Other Changes

- The state of the Franklin number generator used in the **ao(pzmodel)** constructor is now returned in the **procinfo** of the output ao, and can be input when constructing further AOs to continue the sequence.
- You can now add and subtract **timespan** objects.
- The **ssm/bode** method no longer throws an error if the system is not stable. Instead a warning is thrown.
- The x ranges on an **ao/iplot** call can now be specified using **timespan** objects.
- The equality operators **ao/eq** and **ao/ne** now return AOs containing logical values to match the behaviour of **ao/ge** and **ao/gt** etc.
- The **ao/fft** method now supports other input data types (cdata and xydata). See commit dd86b5b below for details.
- The user object **submit** and **bsubmit** methods now return a plist containing the IDs etc of the submitted objects.
- The ao waveform constructor now has an 'offset' parameter.
- The **ao/find** method can now be configured to return the indices of the samples found as the main output.
- The ao data file reading constructor can now ignore header lines (via the 'ignorelines' plist key)
- The **ao/split** method has a new way to split data: into segments of a defined length.
- The **ao** constructor can now build 'from xyz' values.
    - **a = ao('*.mat')**
- You can now specify the vertices in **ao/interp** by passing an AO.
- The **ao/delay** method now has a 'mode' parameter which has a default of 'samples' which keeps the old sample shifting behaviour. A new mode 'fftfilter' performs fft filtering using

a simple delay model (constructed from a pzmodel) to apply the delay. Another mode 'timedomain' which does a time-domain fractional delay filtering of the data.

New Methods

- The **ao** class has two new static convenience constructors, **ao.zeros** and **ao.ones**.
- The matrix class now have a **sqrt** method.
- There is a new ao method called **powerFit** to fit a power-law to a frequency series using a log-likelihood approach.
- The **ao** class has a new method, **subsData**, which can replace the values at the specified indices in a number of different ways.
- The **ao** class has a new method called **iacf** which computes the inverse-auto-correlation function of a spectrum. There is also a method **ao/cgfill** which uses **iacf** then **subsData** in constrained gaussian filling mode to fill a data series containing gaps. These methods should be considered as beta releases.
- The **miir** and **mfir** classes have new convenience static constructors for creating standard filters: **miir.lowpass**, **miir.highpass**, **miir.bandpass**, **miir.bandreject**.
- You can create 3D AOs from a sequence of 2D AOs (by forming a matrix from the y values) using the new **ao/scatter3D** method.
- The **matrix** and **collection** classes have new methods: **uminus**, **lpsd**, **abs**, **consolidate**, **detrend**, **diff**, **heterodyne**, **polyfit**, **timeaverage**, **downsample**, **interp**, **interpmissing**, **removeVal**.
- The **matrix** class gets other new methods: fftfilter,
- The **collection** class gets other new methods: cohere, cpsd, tfe, filter, filtfilt

Deprecated and removed features

The following methods and features have been deprecated:

- The LTPDAWorkbench is no longer supported but will kept in the toolbox as long as it continues to function. Once it fails to comply with the status of the toolbox, we will remove it from future releases.

The following methods and features have been removed:

- Some constructors historically supported a call like this:
  a = ao(plist('plist', plist('vals', 1:10)))

  These nested **plist** calls are no longer supported.

MANTIS Issues Resolved

Issue	Type	Decription
0000615	Bug Report	Capacitive actuation models descriptions are wrong
0000606	Change Request	Improvement to pipeline chainability
0000604	Bug Report	Bug in quicklook pipeline steps regarding the 'consolidated' subdir
0000619	Change Request	Pipelines should be allowed to retrieve data from multiple repositories/databases
0000631	Change Request	eliminating Running simulate complete when display time is 'false'

0000635	Change Request	ao/split only accepts a single timespan
0000627	Change Request	ao/heterodyne should accept a 'split' / 'times' option
0000640	Change Request	ao.diff should support order higher than 1 for the '3point' case
0000639	Bug Report	Hash files for extension modules
0000598	Bug Report	AO constructor from mat file doesn't set xunits
0000603	Bug Report	ao/plotSpectrogram not available
0000597	Change Request	ao/interp should accept an ao for the vertices
0000600	Change Request	ao/interp should accept AOs in input plist
0000596	Change Request	Searching in plists
0000622	Change Request	Display plist parameter properties
0000629	Bug Report	ao.linfit does not deal properly with data with no errors
0000616	Bug Report	linfitsvd.m is not dealing with units properly
0000614	Bug Report	submit/bsubmit methods shoul also return UUID
0000620	Change Request	Introducing windowing for FFT
0000623	Bug Report	Integrate does not look after units properly
0000621	Bug Report	test_ltp_iof2acc does not work ...
0000618	Change Request	Suppress output in ao/plot
0000602	Bug Report	3rd party extensions return error during ltpda_startup
0000611	Bug Report	Timespan objects should be accepted as x-range specifier
0000613	Bug Report	Vectors of time objects: issue with minusv
0000612	Bug Report	Timespan class needs a minus method
0000599	Change Request	Bode plots for unstable SSM systems
0000605	Bug Report	Clicking help on STOCEstimateAcc/estimate_acc raises error
0000617	Change Request	Accessing param properties

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Version 2.5.5 LTPDA Toolbox Software

This table summarizes what's new in Version 2.5.5:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as [Deprecated and removed features](#) in descriptions of changes, below.	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [Main Changes](#)
  - [Parameter lists](#)
  - [Plotting](#)
  - [Improved model documentation for SSM models](#)
- [Other Changes](#)
- [New Methods](#)
  - [AO Static Constant Constructors](#)
- [Deprecated and removed features](#)
- [MANTIS Issues](#)
- [Plotting in LTPDA – the new way](#)
  - [Introduction](#)
  - [Plot Styles](#)
  - [Plot information](#)
  - [Plotting AOs](#)
  - [Plotting Collections](#)
  - [Plotting Matrix Objects](#)
  - [Combining collection and matrix objects](#)
- [Detailed changes since v2.5.4](#)

Introduction

This version of LTPDA is 2.5.5. This document lists the changes since V2.5.4.

This version requires MATLAB 2010a or above.

Main Changes

This version of LTPDA includes a vast number of improvements and bug fixes. In addition, the following significant changes are included in this release.

Parameter lists

Default plist values are applied in all methods now and a check is done whether or not the user input keys are supported by the algorithm. You can choose the action that results if a input key is not found in the default plist (choose from 'None', 'Warning', or 'Error') in the LTPDA preferences (**LTPDAprefs**).

Parameters in parameter lists now support multiple key names. This allows us to change the names of parameter keys while remaining backwards compatible with existing scripts.

Plotting

A new way of plotting has been introduced into LTPDA which leverages the figure, axes and line handles of MATLAB. You can set up your default line styles in the LTPDA preferences GUI now. One structural change here is that the 'plotinfo' property of a user object is no longer a plist, but is now a plotinfo object. More details are included below in the section [Plotting in LTPDA – the new way](#).

Improved model documentation for SSM models

For models which support it, the model overview generated when clicking the 'Model Information' documentation link will contain lists of inputs, states, outputs and parameters.

Other Changes

- The mfir constructor now supports specifying the window used to design the filter.
- The **ao/spectrogram** can now produce normalised spectrograms via the new **plist** option 'normalise'.
- The toolbox has a new data type (**tfmap**) for storing time–frequency maps. This is a combination of the **data3D** and **tsdata** types. The **ao/spectrogram** method now returns AOs with this data type.
- **ao/interp** now supports all interpolation methods that MATLAB's **interp1** supports.
- **timespan** objects now have a property 'nsecs' which just returns the length of the timespan. We already have an 'interval' but this returns a string.

New Methods

- The **matrix** class has a wrapper of the **ao/dsmean** method
- The **matrix** class now has spectral methods **cohere**, **cpsd**, **lcohere**, **lcpsd**, **ltfe**, and **tfe**
- The **collection** class now has a **psd** method
- The **ao** class has a new static method **R** which returns an AO containing the gas constant. See table below for other constants.
- New AO method which converts the units of an AO to SI units (**ao/toSI).**

AO Static Constant Constructors

A number of static constructors for the ao class have been added to LTPDA. To create an ao with the value of the speed of light, for example, do

```
>> ao.c
----------- ao 01: C -----------

       name: C
       data: 299792458
             -------- cdata 01 ------------
                  y:  [1x1], double
                 dy:  [0x0], double
             yunits:  [m s^(-1)]
             ------------------------------

       hist: ao / ao / 976fad6449ed51fd421ad39004d225dd468e2ede
description: Speed of light in vacuum
       UUID: 83da778e-b004-46cb-8690-52381120a429
----------------------------
```

Key	Default Value

c	convenient constructor of an AO containing the value of the speed of light in vacuum
G	convenient constructor of an AO containing the value of the universal constant of gravitation
R	convenient constructor of an AO containing the value of the gas constant
e	convenient constructor of an AO containing the value of the elementary charge
epsilon0	convenient constructor of an AO containing the value of the vacuum permittivity
h	convenient constructor of an AO containing the value of the Planck constant
kB	convenient constructor of an AO containing the value of the Boltzmann's constant
mu0	convenient constructor of an AO containing the value of the vacuum permeability

Deprecated and removed features

The following methods and features have been deprecated:

None

The following methods and features have been removed:

- The various math options for ao/iplot which were deprecated in an earlier version of LTPDA have now been removed.
- The function ao/whiten1D doesn't support any more the value 2 for the parameter 'Weights'. Use the string '1/abs' instead.

MANTIS Issues

For detailed information please visit the webpage: **Mantis Bug Tracker**

Issue #	Status	Description
568	resolved	Can save a filter that does not load with miir('<filename>')
572	resolved	ao/detrend doesn't recognise parameter 'N'
590	resolved	Why does the whitened data have using m / sqrt(Hz) ?
594	new	Split for xydata
593	new	scale to SI prefix
592	closed	ao.iplot changes needed
588	closed	Method to retrieve objects for pipeline
580	resolved	We need a method to 'tune' the length of timeseries objects
579	resolved	An ao.toSI method is needed
578	assigned	matrix and collection should subclass ltpda_container

577	resolved	LTPDA Preferences: an option to choose error/warning for plist keys/values
576	resolved	LTPDA Plotting Preferences: 'Markersize' within plot styles
575	assigned	consolidate function terminates with an error message
574	assigned	ao from pzmodel doesn't respect the original output units

Plotting in LTPDA – the new way

Introduction

This document, and the work behind it, is in response to various user requests to have a more powerful way of plotting data. Generally, iplot is fine, but it covers only, let's say, 70% of use cases. In the past, if you wanted to do anything more powerful you had to drop back down to basic MATLAB figure/plot construction. While that is possible, it's quite painful to extract all the information from LTPDA objects and get it all on the plot the way you want. So now there's a new way, and that is the subject of this document.

Once this new way of plotting settles down and solidifies, we can write a new section of the user manual to capture the contents of this document.

Plot Styles

The first new feature is the ability to define yourself the default plot colors and styles that will be used. This is done under the LTPDA preferences panel (LTPDAprefs). On the tab entitled "Plot Styles" you can now specify a list of styles that will be used, in order, if no other information is provided. You can also import and export these styles (to an XML file). That even allows you to edit this list by hand (which may be easier). If you choose to do that, you should be aware that the color is specified in the RGB format of java – a single negative number. To determine a number for your favourite color, you can do the following:

```
% Translate a MATLAB color to a Java color
c1 = utils.prog.mcolor2jcolor([0.565 0.247 0.667]);
c2 = utils.prog.mcolor2jcolor('r');
% Determine the RGB value
v1 = c1.getRGB()
v2 = c2.getRGB()
```

Plot information

In previous times, the 'plotinfo' field of all ltpda_uoh objects was intended to contain a plist which could set values for how the object/data would be presented on plots. For example, you could set values for things like 'color', 'linestyle', and 'linewidth'. These values would be read by iplot and used when creating a plot.

Now, in the current CVS version of LTPDA, the plotinfo field is always a plotinfo object – a new class of object which encapsulates these various display properties. The following table lists and describes the properties:

plotinfo	
Property	Description
style	Contains a (java) line-style object. See table below.
includeInLegend	A flag to indicate whether or not the owning object should be

	included in the legend when plotted.
showErrors	A flag to indicate whether or not the errors (should they exist) of the owning object should be plotted as errorbars on the plot.
figure	A handle to the figure on which the owning object should be plotted.
axes	A handle (or handles) on which the owning object should be plotted.

plot style [mpipeline.ltpdapreferences.PlotStyle]	
Property	**Description**
linestyle	One of the valid linestyles (see utils.plottools.allowedLinestyles)
linewidth	The linewidth to use when plotting the data contained in the object which owns the plotinfo object.
color	A valid MATLAB color.
marker	A valid MATLAB marker (see utils.plottools.allowedMarkers)

Creating a plotinfo object (should you wish to) can be done in the following ways:

```
p = plotinfo(); % creates a plotinfo with the next default style
p = plotinfo(2); % creates a plotinfo with the 2nd style defined in the prefs
p = plotinfo(pl) % creates a plotinfo from a plist
p = plotinfo(linestyle, linewidth, color, marker); % create with these properties
p = plotinfo(linestyle, linewidth, color, marker, ...
includeInLegend, showErrors, axes, figure);
```

Normally, creating a plotinfo object is not necessary. Instead, if you have an LTPDA user object (for example, an AO) you can use convenient methods to set properties. When using the following methods, if the object doesn't already have a defined plotinfo object, then one is created.

Methods for setting plotinfo fifields	
Method	**Description**
setPlotLineStyle	Set the line style of the plotinfo.
setPlotLinewidth	Set the linewidth of the plotinfo.
setPlotColor	Set the color of the plotinfo.
setPlotMarker	Set the marker of the plotinfo.
setShowsErrors	Set the showErrors flag.
setPlottingStyle	Set one of the defined plotting styles.
setPlotAxes	Set the axes of the plotinfo.
setPlotFigure	Set the figure of the plotinfo.

Examples of how to use plotinfo objects and these associated methods will be presented in the following sections.
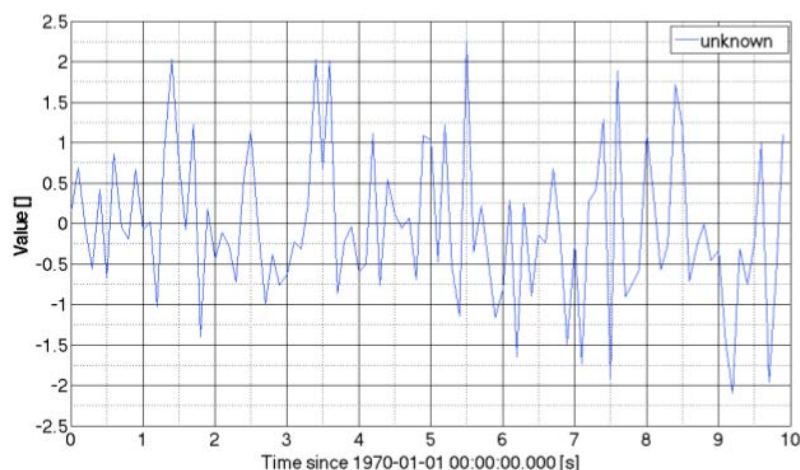
Plotting AOs

Plot a single AO

Given a single AO, you can tell it to plot itself by doing the following:

```
% build an AO
a = ao.randn(10,10);
% plot it on a new set of axes on a new figure using the next default plot style
a.plot
```

This will produce a plot like the following:

Note that if you plot the object again, the next defined plot style will be used. If you want to use the same style again, you have to set the plot style to a defined state before plotting, either by resetting the internal counter (plotinfo.resetStyles()) before plotting, or by setting the style before plotting (using, for example, a.setPlottingStyle(1)).

Plot an AO with a particular style

You can plot an AO with one of you defined styles by doing the following:

```
% build an AO
a = ao.randn(10,10);
a.setPlottingStyle(4);
% plot it on a new set of axes on a new figure using the next default plot style
a.plot
```
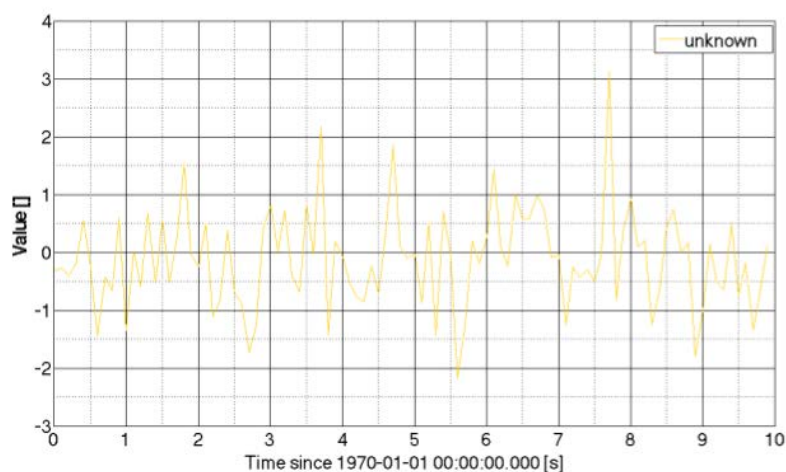

Alternatively, you can specify simply the color, or any of the other properties:

```
% build an AO
a = ao(1:10);
a.setPlotColor('m');
a.setPlotLineStyle('--');
% plot it on a new set of axes on a new figure using the next default plot style
a.plot
```
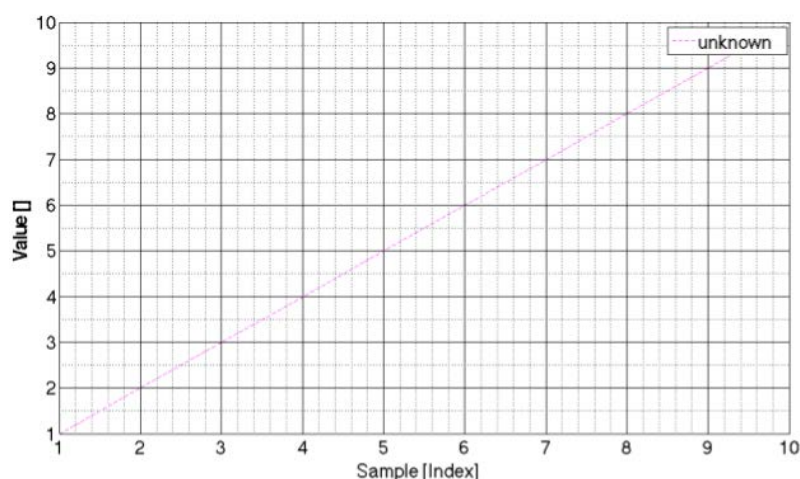

You can, if you so wish, build a plotinfo object and use it:

```
% build an AO
a = ao.randn(10,10);
pi = plotinfo('-', 3, 'k', 'none');
a.setPlotinfo(pi);
% plot it on a new set of axes on a new figure using the next default plot style
a.plot
```
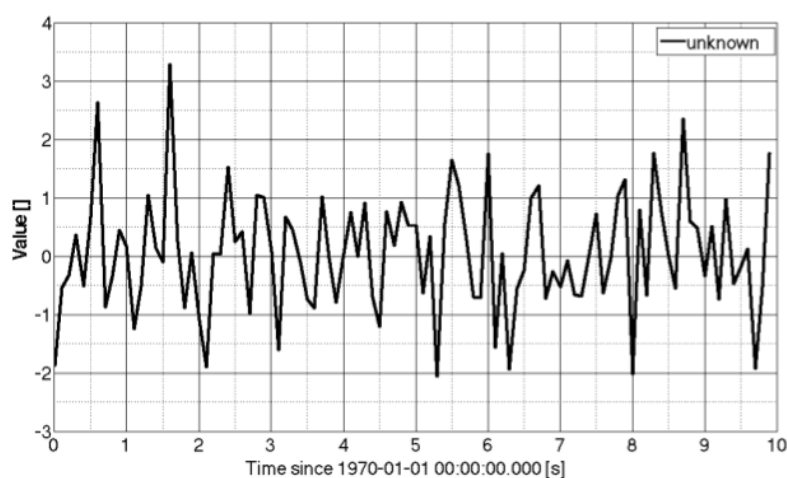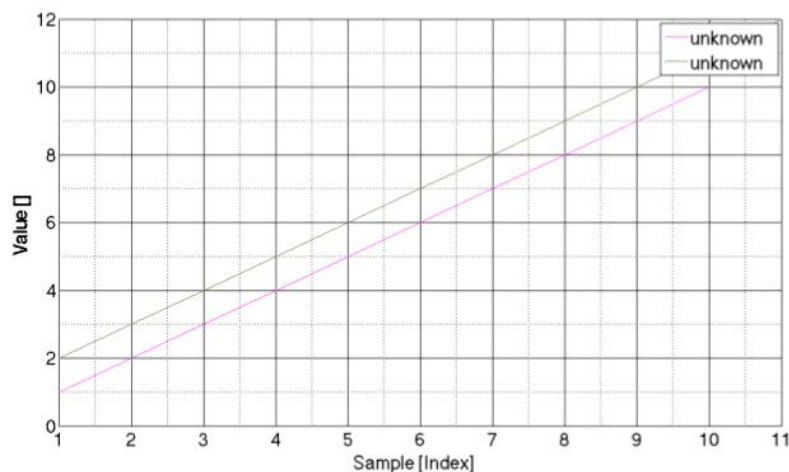

Plotting multiple AOs on the sample axes

By default, if you tell ao/plot to plot multiple objects, it will try to plot them on the same set of axes. Here's an example of this:

```
% build two AOs
a = ao(1:10);
b = ao(2:12);
% Set their colors
a.setPlotColor('m');
b.setPlotColor([0.3 0.5 0.24]);
% Plot them together
plot(a,b)
```

This will produce a plot like this:

You can also chain together plotting events. Suppose you plot an object then later wish to plot an object on the same axes, you can do:

```
% reset the plotting styles counter
plotinfo.resetStyles();
% build two AOs with particular start times
a = ao.randn(10,10);
a.setName();
a.setT0('2012-04-04 12:00:00')
a.setToffset(5)
b = ao.randn(10,10);
b.setName();
b.setT0('2012-04-04 12:00:20')
% Plot them together, and get the objects back with their plotinfo fields filled
[a, b] = plot(a,b)
% Create a third object
c = ao.randn(15,10);
c.setName;
c.setT0('2012-04-04 12:00:10')
% Set the plot axes for this object to be the same as for the first object we plotted
c.setPlotAxes(a.plotinfo.axes);
% Plot the third object
c.plot
```

This will yield a plot like this:
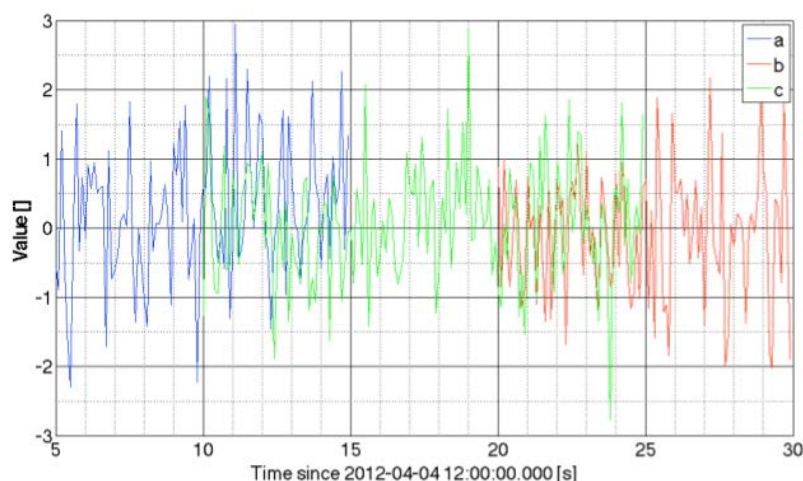
The key here is that you tell the object 'c' where it should plot itself (on which axes).

Plotting multiple AOs on subplots

As stated above, the default is to plot multiple objects on the same set of axes. Suppose you want to plot objects on subplots. Then you first create the subplot axes and tell each object where it should plot itself. Here's an example:

```
% Reset the plot styles counter
plotinfo.resetStyles();
% Create two AOs
a = ao(1:10);
b = ao(2:12);
% Make a new figure with two subplots
figure
ah(1) = subplot(121);
ah(2) = subplot(122);
% Tell each AO on which subplot axes it should plot itself
a.setPlotAxes(ah(1));
b.setPlotAxes(ah(2));
% Plot the objects
plot(a,b)
```

This should yield a plot like this:
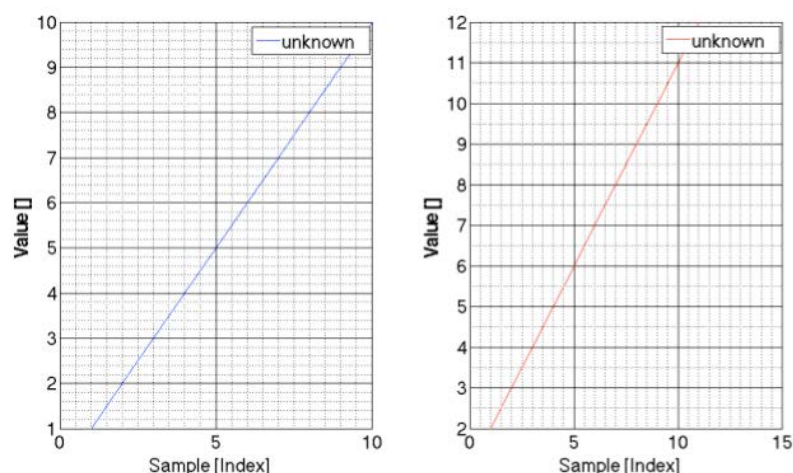
Plotting complex valued AOs

If the AO you wish to plot has complex values, the default is to plot the absolute value of the data. If you provide two axes handles, however, then the real and imaginary (for cdata or xydata) or absolute value and phase (for fsdata) will be plotted. Here's an example:

```
% Reset the plot styles counter
plotinfo.resetStyles();
% Create a complex valued cdata AO
a = ao(complex(1:10, 1:10));
% plot magnitude
a.plot
% plot real and imag
figure
ah(1) = subplot(211);
ah(2) = subplot(212);
a.setPlotAxes(ah);
a.plot
```
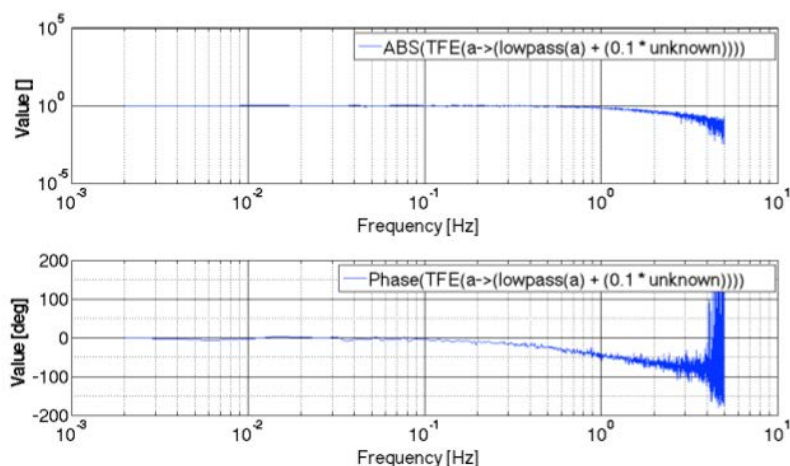
This will produce two plots like this:
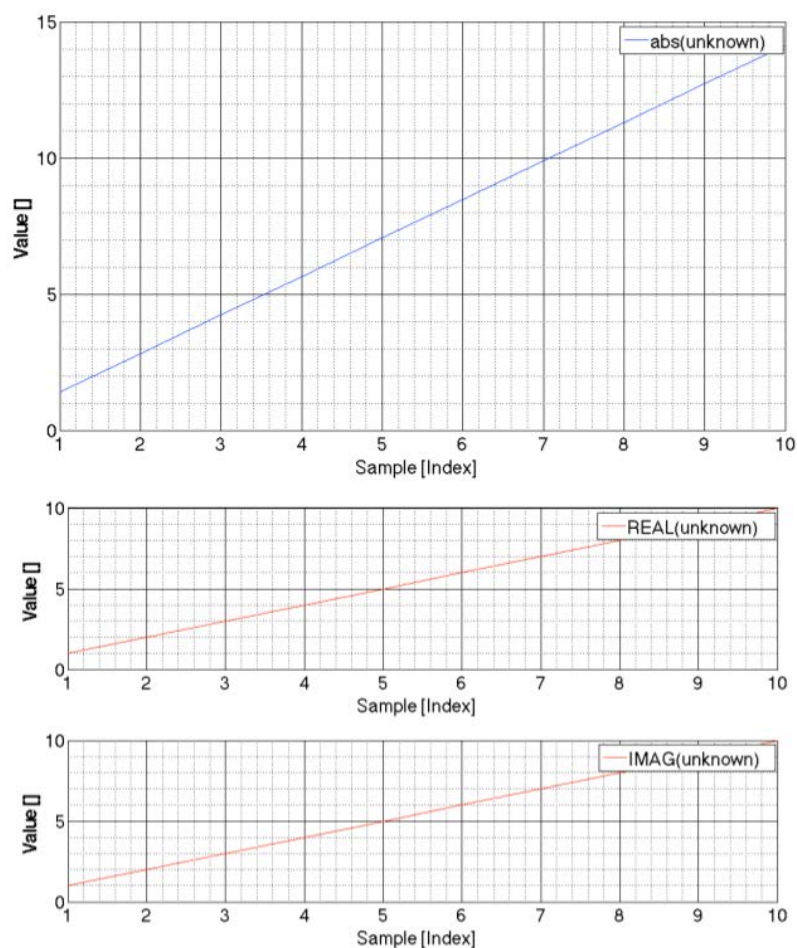
Here's an example of a complex valued fsdata AO:

```matlab
% Reset the plot styles counter
plotinfo.resetStyles();
% Create a complex valued fsdata AO
a = ao.randn(1000,10);
f = miir(plist('type', 'lowpass', 'fc', 1, 'fs', 10));
af = a.filter(f);
out = af + 0.1 * ao.randn(1000,10);
T = tfe(a, out, plist('navs', 4));
% Create two subplots
figure
ah(1) = subplot(211);
ah(2) = subplot(212);
% Set the axes
T.setPlotAxes(ah)
% Plot the transfer function
T.plot
```

This gives a plot like this:

Here's an example of a complex valued fsdata AO:

Plotting other kinds of objects

In order to plot objects other than AOs, there needs to be a plot method defined. So far we have a plot method for all subclasses of ltpda_tf (pzmodel, mfir, miir, rational, parfrac). So you can now do:

```
% Reset the plot styles counter
plotinfo.resetStyles();
% Create a pzmodel
p = pzmodel(1, 1, 100);
% Plot mag and phase
figure
ah(1) = subplot(121);
ah(2) = subplot(122);
p.setPlotAxes(ah);
p.plot
```

Which yields:
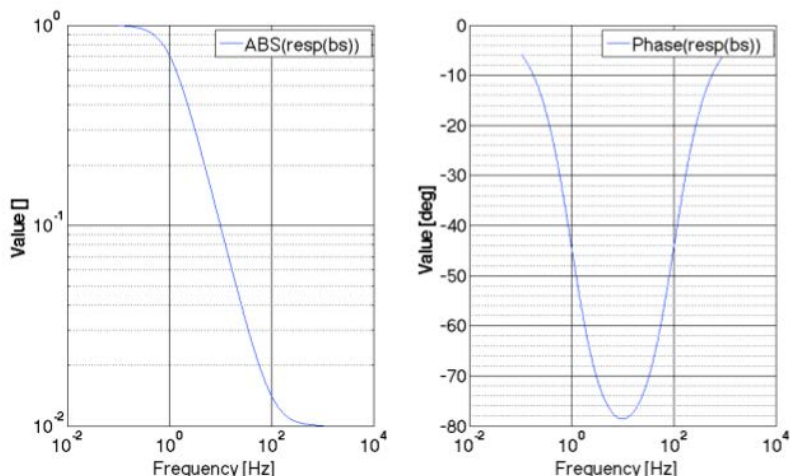
Creating more complex plots

Now that you can specify the axes on which objects should appear, you can do very complex plots. Here's an example where the axes are set out and then populated.

```
% Reset the plot styles counter
plotinfo.resetStyles();
% Create two AO time-series
a1 = ao.randn(100, 10);
a1.setName();
a2 = ao.randn(100, 10);
a2.setName();
```

```
% compute PSDs
[S_a1, S_a2] = psd(a1, a2);
% make figure
figure
topLeft = subplot('Position', [0.1 0.6 0.35 0.35]);
topRight = subplot('Position', [0.55 0.6 0.35 0.35]);
bottom = subplot('Position', [0.1 0.12 0.8 0.35]);
% set axes
a1.setPlotAxes(topLeft);
a2.setPlotAxes(topRight);
S_a1.setPlotAxes(bottom);
S_a2.setPlotAxes(bottom);
% plot objects
plot(a1, a2, S_a1, S_a2)
```

This will produce a figure like this:
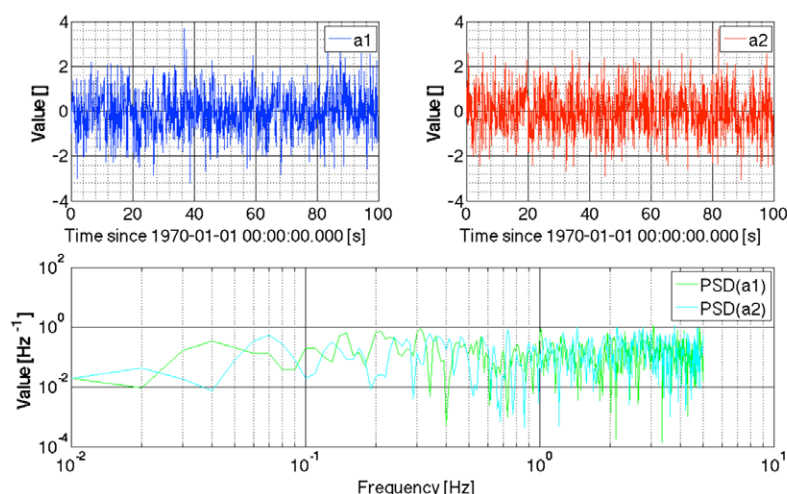
Plotting Collections

A plot method is defined on the collection class. The way it works is to try to plot all objects in the collection on the same set of axes. If the collection object has no plotinfo, or if no axes are defined in the plotinfo, then a new set of axes are created on a new figure. If the collection does have a defined set of axes in its plotinfo, then all objects are plotted on that set of axes. This works by setting the axes to each object in the collection, then telling the object to plot itself. This relies on there being a defined plot method for each class of object in the collection.

Here's an example of plotting a collection containing two AOs on a single set of axes:

```
% Reset the plot styles counter
plotinfo.resetStyles();
% Create two AO time-series
a = ao.randn(10,10);
b = ao.randn(10,10);
% Create a collection containing these AOs
c = collection(a, b);
% Plot the collection
c.plot
```

This should produce the following figure:
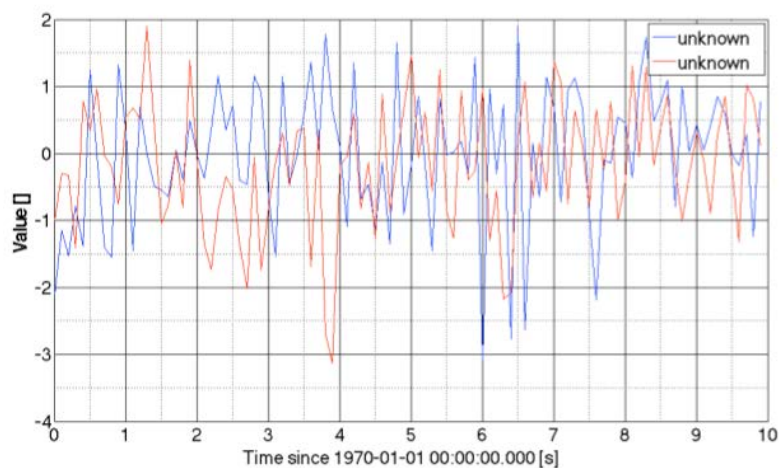
You can also plot collections on subplots:

```
% Reset the plot styles counter
plotinfo.resetStyles();
% Create 4 AOs
a = ao.randn(10,10);
b = ao.randn(10,10);
c = ao.randn(10,10);
d = ao.randn(10,10);
% Collect them into 2 collections
c1 = collection(a, b);
c2 = collection(c, d);
% Create two subplots
figure
ah(1) = subplot(211);
ah(2) = subplot(212);
% Set the collection axes
c1.setPlotAxes(ah(1));
c2.setPlotAxes(ah(2));
% Plot the collections
plot(c1, c2)
```

## This will give a plot like this:
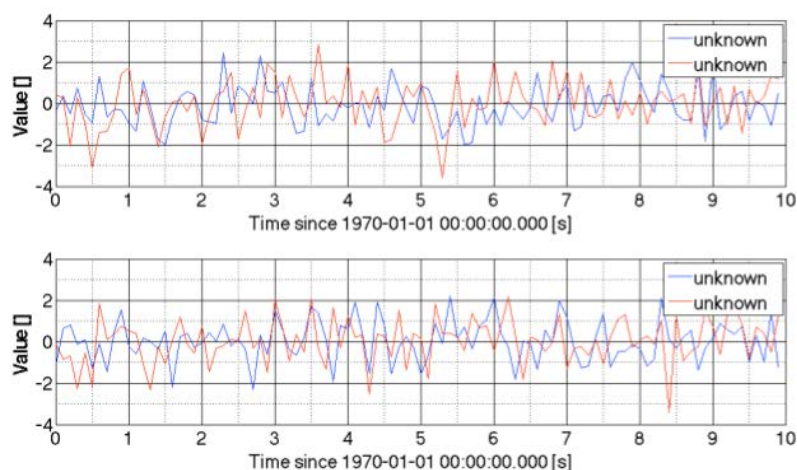
Plotting Matrix Objects

In order to create complex arrays of subplots, you can use matrix objects. The matrix class has a plot method defined on it, allowing you to layout your objects in any way you wish.

Plotting a 2x1 matrix

Here is an example of plotting two AOs side-by-side using a matrix object:

```
% Create two AOs
a = ao.randn(10,10);
b = ao.randn(10,10);
a.setName;
b.setName;
% Collect them into a matrix
m = matrix(a, b)
% Plot the matrix
m.plot
```

This will produce a figure like this:
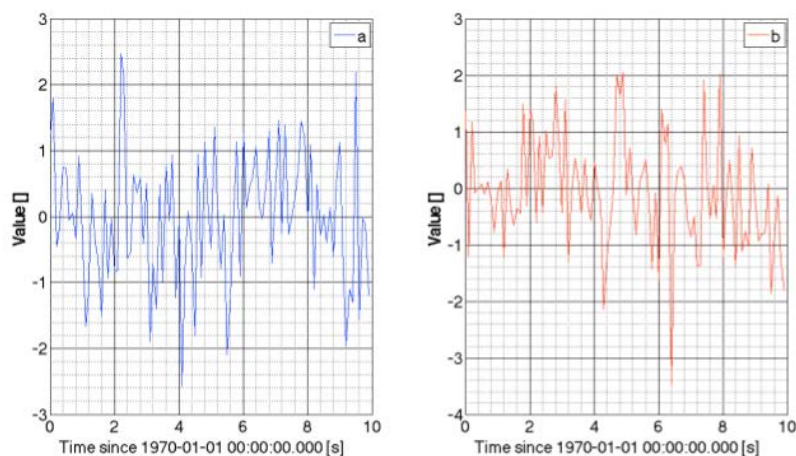
If you want the plots the other way around, you can just make a matrix of a different shape:

```
% Create two AOs
a = ao.randn(10,10);
b = ao.randn(10,10);
a.setName;
b.setName;
% Collect them into a matrix
m = matrix([a; b])
% Plot the matrix
m.plot
```
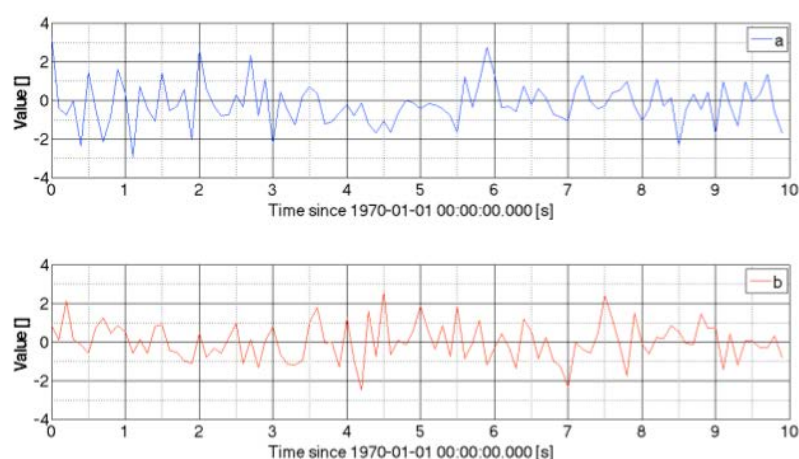
Producing a plot like this:

Plotting a 2x2 matrix

```
% Create two AOs
a1 = ao.randn(10,10);
a1.setName;
a2 = ao.randn(10,10);
a2.setName;
a3 = ao.randn(10,10);
a3.setName;
```

```
a4 = ao.randn(10,10);
a4.setName;
% create a 2x2 matrix
m1 = matrix([a1 a2; a3 a4])
m1.plot
```
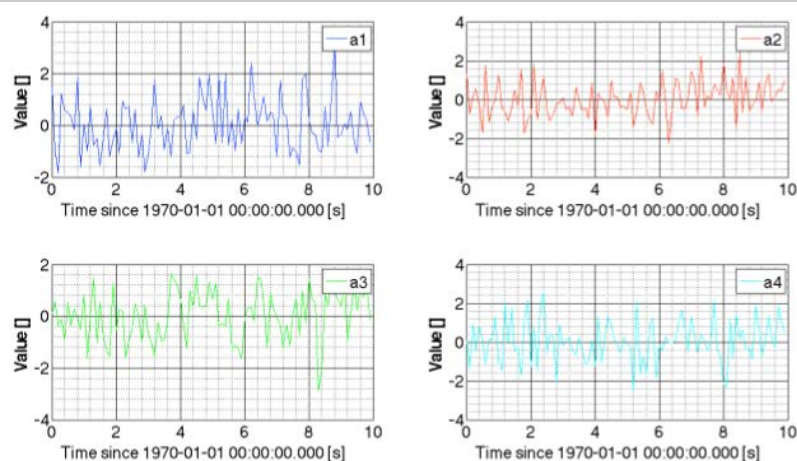

Combining collection and matrix objects

By using a combination of collections and matrix objects, you can create more complex arrangements:

```
a1 = ao.randn(10,10);
a1.setName;
a2 = ao.randn(10,10);
a2.setName;
a3 = ao.randn(10,10);
a3.setName;
c1 = collection(a1)
c2 = collection(a2, a3);
m = matrix([c1; c2])
m.plot
```



```
a1 = ao.randn(10,10);
a1.setName;
a2 = ao.randn(10,10);
a2.setName;
a3 = ao.randn(10,10);
a3.setName;
c1 = collection(a1)
c2 = collection(a2, a3);
m = matrix([c1; c2])
m.plot
```
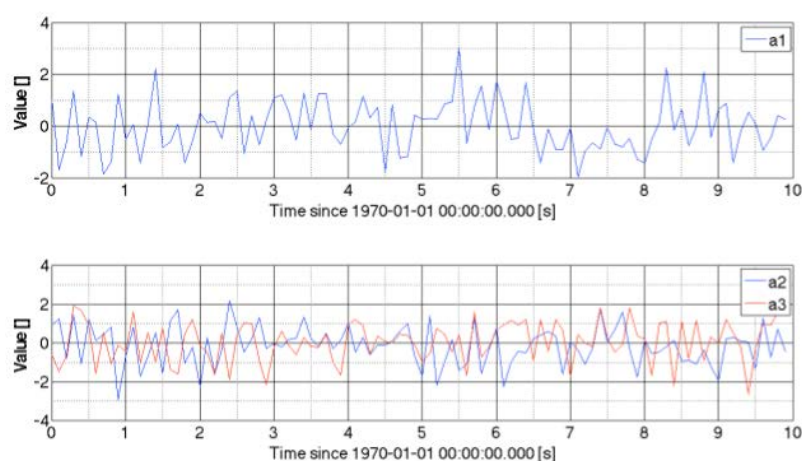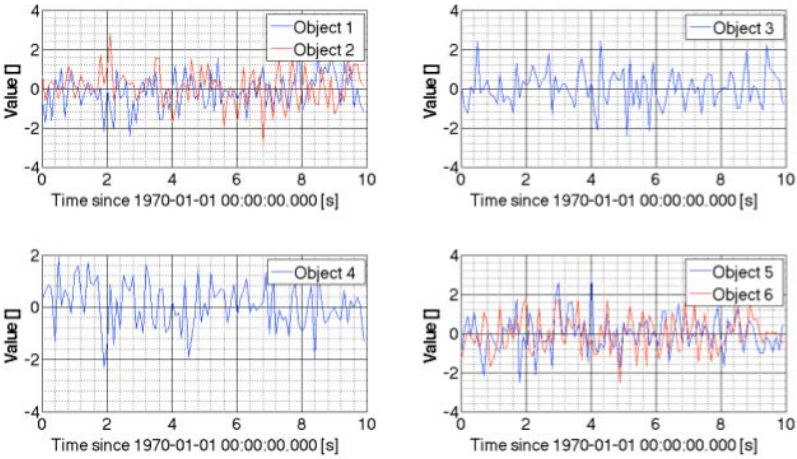
©LTP Team

Version 2.5.4 LTPDA Toolbox Software

This table summarizes what's new in Version 2.5.4:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	[Bug Reports](#) at Web site	Printable Release Notes: Not available

- [Introduction](#)
- [MANTIS Issues Resolved](#)

Introduction

This version of LTPDA is 2.5.4. This document lists the changes since V2.5.3.

This version requires MATLAB 2010a or above.

This is a minor update on V2.5.3. Mostly changes are minor changes and bug fixes which are collected in the STOC-LPF Simulation 1 meeting.

Version 2.5.4 LTPDA Toolbox Software (LTPDA Toolbox)

LTPDA Toolbox™   [contents](#)   ◀ ➡

Version 2.5.3 LTPDA Toolbox Software

This table summarizes what's new in Version 2.5.3:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [Main Changes](#)
  - [The collection class](#)
- [Other changes](#)
- [MANTIS Issues Resolved](#)
- [Complete CVS Changes](#)

Introduction

This version of LTPDA is 2.5.3. This document lists the changes since V2.5.2.

This version requires MATLAB 2010a or above.

This is a minor update on V2.5.2. Mostly changes are minor changes and bug fixes due to the STOC–LPF Simulation 1 meeting.

Main Changes

The collection class

The collection class has new behaviour similar to MATLAB's **struct** class. This means that the objects in the collection can now have names assigned to them, and can be referenced via these names. Here is some example usage:

```
% Build collection using default reference names
a = ao(1);
b = ao.randn(10,10);
c = collection(a,b);
a1 = c.obj1;
b1 = c.obj2;
```

```
% specify the reference names during construction
c = collection('a', a, 'b', b);
a1 = c.a;
b1 = c.b;
```

```
% Add an object to the collection
c = collection('a', a, 'b', b);
c.d = ao(4);
```

Other changes

- The **ao** methods **gt**, **lt**, **ge**, **le** now return **cdata** AOs.

- The time class now has implementations of **ge** and **le**.
- The **ao** class has a new **logical** method.
- The **ao** class now has **all** and **any** methods which return **cdata** AOs containing logical values.
- The **ao** class has a new parametric PSD estimator (**ppsd**) – still in beta.
- The **timespan** class has a new method, **inTimespan**, which checks if the given time is contained in the **timespan** object.
- The **pest** class has a new display method, **viewResults**, to show its contents in nicely formatted html tables.

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Version 2.5.2 LTPDA Toolbox Software

This table summarizes what's new in Version 2.5.2:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	No	Printable Release Notes: PDF

- Introduction
- MANTIS Issues Resolved
- Complete CVS Changes

Introduction

This version of LTPDA is V2.5.2.

This version requires MATLAB 2010a or above

This is a very minor update on V2.5.1. Mostly changes are bug fixes due to the training session 2.

◀ Version 2.5.3 LTPDA Toolbox Software   Version 2.5.1 LTPDA Toolbox Software ➡

©LTP Team

Version 2.5.2 LTPDA Toolbox Software (LTPDA Toolbox)

Version 2.5.1 LTPDA Toolbox Software

This table summarizes what's new in Version 2.5.1:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	No	Printable Release Notes: PDF

- Introduction
- MANTIS Issues Resolved
- Complete CVS Changes

Introduction

This version of LTPDA is V2.5.1.

This version requires MATLAB 2010a or above

This release is essentially a documentation update to include the second LTPDA training session content in the user manual.
There are also a number of bug fixes in this release.

◀ Version 2.5.2 LTPDA Toolbox Software   Version 2.5 LTPDA Toolbox Software ➡

©LTP Team

Version 2.5.1 LTPDA Toolbox Software (LTPDA Toolbox)

LTPDA Toolbox™	[contents](#)	⬅ ➡

Version 2.5 LTPDA Toolbox Software

This table summarizes what's new in Version 2.5:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as **Removed methods** in descriptions of changes, below. See [Removed Methods](#).	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New features and major changes](#)
    - [Changes to time-series data handling](#)
    - [New class methods](#)
    - [Methods removed](#)
- [Other Minor Changes of Note!](#)
- [MANTIS Issues Resolved](#)
- [Complete CVS Changes](#)

Introduction

This version of LTPDA is 2.5. This document lists the changes since V2.4.

This version requires MATLAB 2010a or above

The main focus of this release has been on improvements in time-series handling, bug fixing and underlying code improvements. In addition, the LTPDA repository underlying database structure has changed in this release, and the client-side functionality has been adapted to work with both the old repositories and new repositories. There have also been some fairly major changes to the toolbox, so keep reading to find out what they are! In addition, a significant number of bugs and change requests have been addressed.

New features and major changes

- A check is made on the plist parameters passed to methods now. If the user specifies a key which is not in the default plist described in the method documentation, a warning is issued. In addition, for parameters which have a specified set of possible values, an error will be thrown if the user tries to use a value not in the set.
- The repository connection manager has had a major overhaul with new settings in the LTPDA preferences for setting password expiry times and default repositories.
- A new utility function is available for creating new methods in extension modules. See help on **utils.modules.makeMethod** for details.
- The **matrix/mcmc** method can now handle priors and can do annealing with a thermostat.
- The **ao/filtfilt** method can now do frequency-domain filtering.
- The matrix class has a number of new methods which simply call the same method on the internal objects. For example:
    m = matrix(ao.randn(10,10), ao.randn(10,10));
    p = psd(m)
calls **ao/psd** on each of the internal AOs in the matrix.
- The LTPDA user-manual now has a Functions section which lists all class methods, organised by class and category.
- The **ao/split** method supports now a parameter offset whose value is an array of start/stop **offsets** to split by. Positive offsets are relative to the first sample. Negative offsets are taken from the end of the vector. For example [10 -10] removes 10 seconds from the beginning and end of the vector. An end time of 0 indicates the end of the vector.
- The rules for adding and subtracting objects were updated:
    - for xydata objects, the x axes should match
    - for tsdata objects, the nsecs and fs should match (evenly sampled data)
    - for tsdata objects, the x axes should match modulo the toffset (unevenly sampled data)

- It is now possible to create a cdata AO using an existing AO as input. Here are some examples:
    a1 = ao.randn(2,2);
    a2 = ao(plist('vals', a1)) % a2.y == a1.y
    a3 = ao(plist('vals', a1, 'axis', 'x')) % a3.y == a1.x
- The matrix and collection classes can now be nested. For example, you can do:
    m1 = matrix(ao.randn(10,10), ao.randn(10,10));
    m2 = matrix(ao.randn(10,10), ao.randn(10,10));
   **m = matrix(m1, m2);**
  In previous versions of LTPDA, these calls would just copy the input matrix objects. Now these 'copy constructors' are removed and building of nested matrix and collection objects works.
- LTPDA includes a useful function for listing all the plist keys for constructors. For example:
   **keys('ao')**
- For the curious user, you can generate a class diagram (assuming you have graphviz installed) by doing:
    utils.helper.make_class_diagram(false) % don't show class methods
    utils.helper.make_class_diagram(true) % show class methods
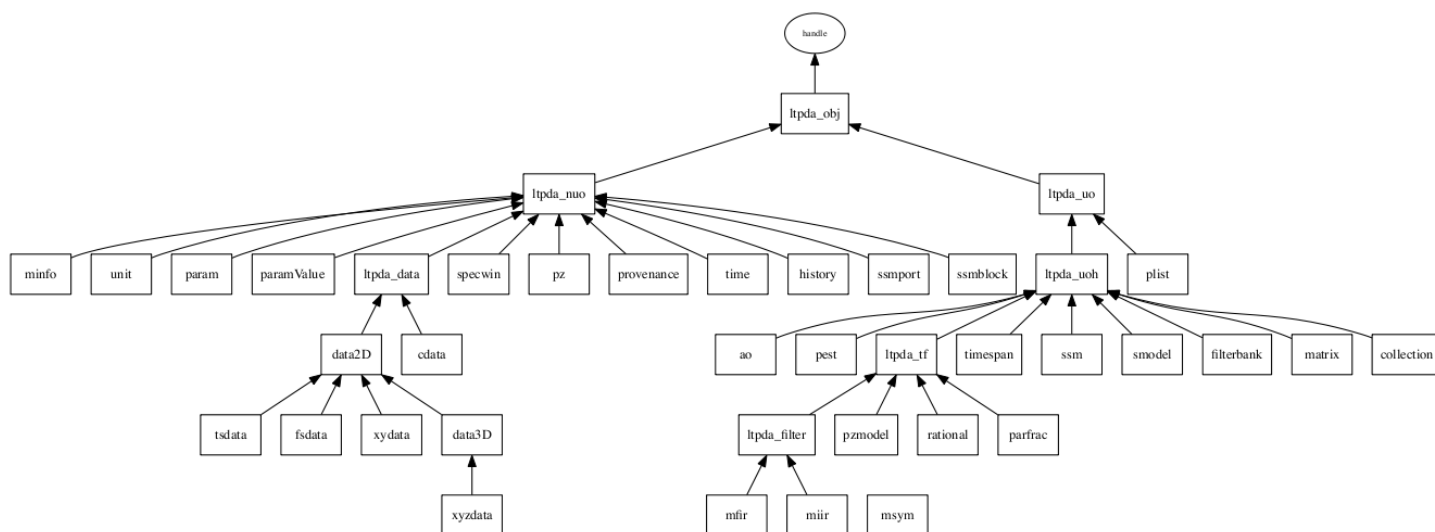
Changes to time–series data handling

In previous versions of LTPDA, time–series AOs (which contain a **tsdata** object) had two degrees-offreedom for describing the absolute position of the data in time. These were the 't0' property which was used to specify the absolute time of the first sample, and then value of the first x sample (for evenly sampled data, this was always 0). In this version of LTPDA the property 't0' has a new meaning. Now it refers to a reference time and a new property has been added to the **tsdata** class which encodes the time between this reference time and the first x sample. As such, it is now possible to have two time–series which have the same reference time (the same 't0') but which have their data actually starting at different absolute times. An example is depicted in the following diagram:
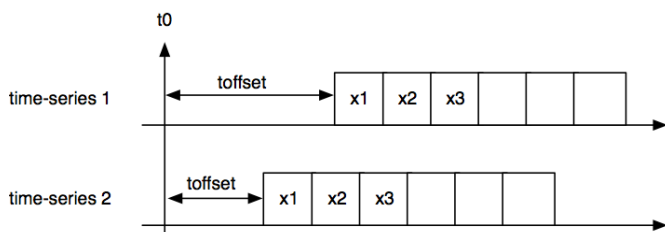
This **toffset** property is mostly handled automatically by the toolbox, but it is possible for the user to set this value using the **ao/setToffset** method. The following situations can occur:

- Setting **toffset** via **ao/setToffset** means the data will be shifted in time.
- Setting the reference time via **ao/setReferenceTime** means the data stays where it is, the reference time (**t0**) is changed, and the **toffset** is updated accordingly.
- Changing the reference time vi **ao/setT0** means the data moves in time because **t0** is update but **toffset** retains its value.

To adapt to this change, the **ao/timeshift** method no longer behaves as it did. Instead, now it adjusts the **toffset** by the relative amount given. Note: This method does no fancy interpolation it just shifts the start

time of the first sample by the given amount relative to t0.

For backwards compatibility, we do the following steps on saving and loading of objects:
– before saving, t0 = t0 + toffset
– after loading, t0 = t0 – toffset

This means objects created with older versions of the toolbox are handled as if the toffset is zero, and older versions of the toolbox should be able to read objects created with toolbox versions >= 2.5.

New class methods

- **matrix/dispersion** computes the dispersion function of an (**smodel**) system given the matrix of input signals.
- **ao/qqplot** is a method to do the quantile–quantile plot with confidence intervals.
- **matrix/modelselect** is method to compute the Bayes factors for an array of input models (ssm or matrix models are supported).
- **ssm/getPortNamesForBlocks** returns a list of port names for a given block. This can be useful for preparing plists for methods like **ssm/simulate** and **ssm/bode**.
- **ssm/viewDetails** shows a documentation page containing details of the given state–space model. It lists the inputs, outputs, states and parameters.
- **matrix/unpack** can be used to conveniently extract the objects from within a matrix object. Internally this calls **matrix/getObjectAtIndex** to ensure proper history tracking.
- The ao class now has some useful static constructors. You can do:
  kb = ao.kb; G = ao.G; c = ao.c; e = ao.e;
  e0 = ao.epsilon0; h = ao.h; mu0 = ao.mu0;

Methods removed

- **ssm/getMatrixSelection** was deprecated since Aug 2010.
- **ssm/noiseSpectrum** was deprecated since Aug 2010.
- **ssm/setBlockDescriptions** was deprecated since Aug 2010.
- **ssm/setBlockNames** was deprecated since Aug 2010.
- **ssm/setBlockProperties** was deprecated since Aug 2010.
- **ssm/setPortUnits** was deprecated since Aug 2010.
- **ssm/setPortNames** was deprecated since Aug 2010.
- **ssm/setPortDescriptions** was deprecated since Aug 2010.
- **matrix/MultiChannelNoise** was removed and replaced by **matrix/mchNoisegenFilter**.
- The matrix class constructor 'from CPSD' was removed and replaced by **matrix/mchNoisegen**.

Other Minor Changes of Note

- **ssm methods CPSD and PSD were renamed to cpsd and psd.**
- **ssm/keepParameters is now deprecated and will be removed in a future release. Use ssm/subsParameters instead.**
- **ao/filter and ao/filfilt no longer support the additional output for returning the filter with its history filled. Instead the filter is returned in the procinfo.**
- **The syntax utils.const.physics.c to obtain values of frequently used fundamental physical constants now returns an AO object, with units and descriptions**
- **Spectral windows are now treated (from user's point of view), merely via their name and length. All the other properties are calculated internally, when needed, based on the window definition.**
- **Operations like adding and subtracting objects with different units, but equivalent to the same SI units, are now supported. For example:**
  a1 = ao(1, plist('yunits', 'N'));
  a2 = ao(2, plist('yunits', 'kg m s^-2'));
  a3 = a1 + a2;
  a4 = a2 + a1;

MANTIS Issues Resolved

Issue	Type	Decription
0000555	Bug Report	Robot does not generate plot for ao with data type 'fsdata'.
0000561	Bug Report	Plot not generated for some of the fsdata type objects
0000536	Change	matrix wrappers for common ao methods

	Request	
0000534	Bug Report	iplot automatic ranging failing with two transfer functions
0000538	Bug Report	setFs does not update nsecs
0000552	Bug Report	Gnuplot interface uses Linux specific code to execute gnuplot, does not work on windows.
0000551	Bug Report	The wrong columns are plotted for data with error bars in gnuplot interface.
0000554	Bug Report	Missing install.php
0000540	Bug Report	split treats aos with and without .data.x vector differently
0000535	Bug Report	ao/convert doesn't retain units etc
0000548	Bug Report	iplotyy() mixes up left and right y-axis labels/yunits
0000537	Bug Report	iplotyy does not work
0000549	Bug Report	linewidths key for iplot not working
0000558	Bug Report	Erratic behaviour of split and consolidate
0000541	Bug Report	consolidate does not work as expected
0000556	Bug Report	Default value for marker size is too small
0000545	Change Request	ao(file,plist('type','tsdata','fs',fs,...) should automatically set xunits to s

◀ **Version 2.5.1 LTPDA Toolbox Software**　　　　　　　　**Version 2.4 LTPDA Toolbox Software** ▶

©LTP Team

LTPDA Toolbox™　　　　　　　　[contents](#)　　　　　　　◄　►

Version 2.4 LTPDA Toolbox Software

This table summarizes what's new in Version 2.4:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as **Deprecated methods** and **Removed methods** in descriptions of changes, below. See [Deprecated](#) or [Removed](#).	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New features and major changes](#)
  - [Extension modules](#)
  - [Saving and loading AOs](#)
  - [Preferences](#)
  - [Logical AOs](#)
  - [Plotting AOs](#)
  - [Error propagation](#)
  - [Matrix constructors](#)
  - [New class methods](#)
  - [Removed methods](#)
  - [Multiplying matrices of AOs](#)
  - [Method outputs](#)
  - [Changes to evaluating smodel and pest object](#)
  - [Smodel class has new properties](#)
  - [Built-in model format](#)
  - [SSM class user-interface](#)
  - [Detrending](#)
- [Other Minor Changes of Note](#)
- [MANTIS Issues Resolved](#)

Introduction

This version of LTPDA is 2.4. This document lists the changes since V2.3.1.

This version requires MATLAB 2010a or above

The main focus of this release has been on code optimisation to improve execution speed and to aid in maintainability. There have also been some fairly major changes to the toolbox, so keep reading to find out what they are! In addition, a significant number of bugs and change requests have been addressed.

New features and major changes

Extension modules

LTPDA now supports 3rd party extension modules. See the LTPDA user manual for details how to build and install extension modules.

Note: this is now the supported way to add your own-built in models to LTPDA. The old scheme of adding directories in the LTPDA preferences is no longer supported and has been removed. Users with their own built-in models should build an extension module and move the models in to the extension module. The user manual explains where to put them.

Saving and loading AOs

- **ao/save**If the user doesn't specify the filename then **save** saves the object(s) as a MAT file using the variable name in the current folder
- The AO constructor for loading data from ASCII files now makes fewer assumptions about the content of the file. Instead the user is expected to pass more information in the plist. The only assumption we make is that the data files contains at least one column of data. As such, the call
  a = ao('mydata.txt')
  will result in a cdata AO containing the all columns of data in the file stored as a matrix.
- When constructing AOs from files, all file extensions other than 'xml' and 'mat' are treated as if they contain ASCII data.

Preferences

LTPDA preferences has new panel for configuring the default look of plots

Logical AOs

AOs now accept logical values for construction and we have some new logical logical operators (see below).

Plotting AOs

ao/iplot now has a new plist key for overriding the default legend font size **'LegendFontSize'**. You can also include object descriptions in the legend by setting the corresponding preference is set in LTPDAprefs.

ao/iplot The use of the **'xmaths'**, **'ymaths'**, and **'zmaths'** parameters is now deprecated and will be removed in a future release.

Error propagation

More AO methods now propagate errors: **ao/scale**, **ao/power**, **ao/mtimes**, **ao/mrdivide**.

Matrix constructors

We have some new constructors for the matrix class:

matrix(plist('values', ..., 'names', ..., 'yunits', ...))
matrix(doubleArray)
matrix(doubleArray, cellArray)

New Class Methods

- **smodel/hessian** computes the hessian matrix for **smodels**
- **ao/intersect** forms the intersection of two AOs
- new logical binary operators for AO class: **ao/and, ao/or**
- **ao/bicohere** computes the bicoherence of two input time-series. The result is a complex frequency-frequency-complex coherence map.

- **ao/average** is a method to average AOs point-by-point. For each point, an average is taken over all the input objects.

- New AO factory constructors:
  1. **ao.randn(nsecs, fs)** – produces a time-series of random numbers
  2. **ao.randn(nsamples)** – produces a cdata AO of random numbers
  3. **ao.sinewave(nsecs, fs, f0, phi)** – produces a time-series of a sine-wave
  4. **[o1, o2, ...] = ao.load(filename)** – load multiple objects from a file. You need to know how many objects are in the file so that you can specify the correct number of output objects.

- **ltpda_uoh/requirements** lists the required extensions for rebuilding an object.
- **ao/zunits** returns the zunits of an xzydata AO
- **ao/setZunits** sets the units of the z-data for an xyzdata AO
- We added a new AO constructor from a parameter contained in a plist. Suppose we have a plist containing a key **'a'**, then we can make a cdata AO with the value of **'a'**. Other properties are also used in constructing the AO (at the moment only properties 'unit' and 'units' are supported). All properties are added to the procinfo.
  e.g. **a = ao(plist('parameter', pl, 'key', 'a'))**

Removed methods

Some LTP-specific methods have been removed and moved in to extension modules.

The following were all moved to the LPF_DA_Module extension module:

- **ao/ltp_ifo2ac**
- **ao/smallvec_coef**
- **ao/smallvector_lincom**
- **ao/smallvectorfit**
- **ao/mdc1_ifo2acc_inloop**
- **ao/mdc1_ifo2cont_utn**
- **ao/mdc1_cont2act_utn**
- **ao/mdc1_ifo2control**
- **ao/mdc1_ifo2acc_fd_utn**
- **ao/mdc1_x2acc**
- **ao/mdc1_ifo2acc_fd**
- **pest/LTPimperf2physParams**

Other methods removed which were deprecated in previous releases:

- **ao/pwelch** – use ao/psd instead
- **ao/curvefit** – use ao/xfit or ao/tdfit instead
- **ao/straightLineFit** – use ao/linfit instead
- **ao/timedomainfit** – use ao/lscov instead
- **pest/toAO** – use pest/find instead
- **ao/ltpda_fitChiSquare**
- **ao/hist_gauss** use ao/hist instead

Multiplying matrices of AOs

Given two matrices of AOs, the inner multiplication operator has changed from a matrix multiplication to an element multiplication.

M1 = [a1 a2; a3 a4]; % a# is an AO
M2 = [b1 b2; b3 b4]; % b# is an AO
M = M1*M2

then **M(1) = a1.*b1 + a2.*b3**

Method outputs

The output of some methods are now matrix objects and no longer vectors of AOs to make rebuilding work:

- **ssm/simulate**
- **ssm/CPSD**
- **ssm/PSD**
- **ssm/bode**
- **ssm/kalman**
- **ssm/resp**

ao/rotate no longer supports the multiple output function call.

To retrieve the AOs from inside the matrix object in a 'history safe' way, use **matrix/getObjectAtIndex**.

matrix/det no longer returns a matrix object, but instead an object of the same class as the inner objects.

Smodel class has new properties

The **xvals** field is now always a cell–array of vectors

The properties **aliasNames** and **aliasValues** can be used to defines variable aliases which are then used in the model expression.

The **trans** field usage is depracted in favour of setting an **alias**.

Changes to evaluating smodel and pest object

The smodel/double method has been harmonized with Matlab double: the values for the independent variables x must now be vectors of double, and they must be set before the call to the method, that now has no parameters.

The smodel/eval method has been changed to build the output AO is built from the smodel, based on these parameters:

- 'output type' to choose the output data type
- 'output x', to choose the X values for the output data ao. In case of a double vector, the result is a cdata AO. In case of an ao, the output is a copy of this object but the "y" field is calculated from the model.

As a consequence, the pest/eval method has been modified. If the user input the independent variable values XDATA within aos, their 'y' field will be used if not otherwise specified. This is different from previous versions of the Toolbox. Furthermore, if the user input the independent data XDATA within aos, the output data type will be of the same type.

Built-in model format

The format of built-in models was completely written to allow a more flexible versioning of models. You can build new-style built-in models using the utility **utils.models.makeBuiltInModel()**. See the documentation for more details.

ssm class user-interface

The user interface of some ssm class methods was simplified and changed. As such, some old usages may no longer work. Please consult the relevant method help if you encounter any problems.

Detrending

Default detrending order in **ao/detrend** was changed from 0 to 1 to match the corresponding MATLAB detrend behaviour.

Other Minor Changes of Note

- The verbose level for many output messages was increased to make LTPDA a bit quieter.
- User objects now have an empty default name.
- **ao/search** now properly tracks history
- constructing AOs from data files (**'from ASCII'** constructor) works with arbitrary delimiters now. The delimiter can be set in the plist.
- **ao/fft** and **ao/ifft** support a scaling option (**'scale'** plist key) which scales the output by the sampling rate if the AO is a time-series.
- ssm class speed improvements for bode and simulate
- **ao/select** now returns an AO with the x-field filled, even if the result is evenly sampled. The idea here is that selecting samples from a vector (especially a time-series) is not well defined in terms of the resulting sample rate. So we leave the x data alone and let the user decide what to do next.

MANTIS Issues Resolved

Issue	Type	Decription
0000529	Bug Report	LTPDAPrefs GUI not reading the .dot file location
0000528	Bug Report	Rebuilt library does not become live until the workbench is restarted
0000527	Bug Report	inconsistency in AO cdata/xydata class for handling logicals.
0000525	Bug Report	Error message: ??? Operands to the || and && operators must be convertible to logical scalar values. Error in ==> param.getVal
0000518	Bug Report	pzmodel help/documentation
0000516	Bug Report	Problem with default preferences being written on first use
0000515	Bug Report	ao/transpose not working for cdata aos
0000513	Bug Report	ao/join with data not in sequence
0000512	Change Request	FFT/IFFT unit conventions and scaling
0000508	Change Request	Make the parameter overview table editable.
0000507	Change Request	FIlter by key the parameter overview table

0000506	Bug Report	Length of the pipeline name is not updated upon pipeline renaming
0000503	Change Request	Don't limit the loading of ASCII files to .txt and .dat
0000502	Bug Report	Loading single column ascii files
0000501	Bug Report	Arithmetic operations on frequency-series
0000498	Bug Report	LTPDAworkbench/reset has a bug in V2.3
0000497	Bug Report	Remove all mention of non-user classes from the user manual
0000496	Bug Report	Specifying multiple values for 'Xunits' key of ao/iplot
0000495	Bug Report	ltpda_filter/impresp has fixed sample rate.
0000488	Bug Report	Problem with ltpda_startup and LTPDARepositoryManager
0000487	Bug Report	Java error choosing repository
0000486	Bug Report	Original wb filename is stored inside pipelines
0000485	Bug Report	Resize the library tree when gui is maximized
0000481	Bug Report	ao/submit fails to create meta data.
0000480	Bug Report	The Web interface on objmeta table shows: validation, validation date and author , the datail page shows only 2
0000478	Bug Report	Retrieving objects from database, when not specifying completely hostaname, database, username
0000476	Bug Report	Cannot change the size of the pipeline window
0000474	Bug Report	ssm/kalman: parameter 'select' is of wrong type.
0000473	Bug Report	ssm/simulate: field AOS default value lacks a left square bracket.
0000470	Change Request	The databases 'tsdata' table should also store the timezone of the 't0'
0000465	Bug Report	Keyboard shortcut CTRL+O does not work.
0000460	Bug Report	no plot is available with frequency series with 1 point size
0000459	Bug	no plot is available with time series with 1 point size.

	Report	
0000458	Bug Report	Web repository interface generate wrong iplot.
0000458	Bug Report	Web repository interface generate wrong plot
0000457	Bug Report	ao.table doesn't show third column with xyz data.
0000447	Bug Report	ltpdareporobot error
0000444	Bug Report	Submitting 1 object with proc5 verbose i get submitting 2 objects to repository.
0000443	Bug Report	After deleting a object the submit button is still enable
0000442	Bug Report	Interpolation methods 4 resample and downsample.
0000438	Bug Report	Error changing model during execution
0000437	Bug Report	Annoying parameter editing issue
0000434	Bug Report	Annoying zoom problem
0000433	Bug Report	mysql-connector-java-5.1.6-bin.jar is installed in two copies.
0000427	Bug Report	Up arrow and down buttons don't work in execution plan form
0000426	Bug Report	Hitting cancel in the execution Plan panel the modified parameters are stored
0000424	Bug Report	NullPointerException using PEST constructor.
0000421	Bug Report	delay prameter missing in pzmodel plist and is present in mfir from pzmodel
0000420	Bug Report	AO built-in FreeDyn_1: suspicius error message.
0000417	Bug Report	Pole/Zero model editor: problem reading the value of a modified gain.
0000416	Bug Report	Issues reading LISO files.
0000412	Bug Report	Web documentation: downsample, etc.
0000411	Bug Report	SSM: some built-in models generate errors.
0000409	Bug Report	Pzmodel from parfrac conversion problem.

0000403	Bug Report	Plot of different xunits.
0000401	Bug Report	Some Ghost subsystem when iconize all command
0000390	Bug Report	Warning should be given if plist contains parameters that are not applicable.
0000389	Bug Report	Java error when opening the Char prefs of the annotation into the LTPDAWorkbench preferences
0000387	Bug Report	The relational operators do not mention the "exceptions" parameters in their help
0000381	Bug Report	miir.resp and filterbank.resp have different plists.
0000375	Bug Report	Error using the "include in legend" checkbox.
0000370	Bug Report	Setting plotinfo marker stops execution and throws exception
0000360	Change Request	saving the submit info into a file
0000355	Change Request	We have two positions in the cvs where we store the jar files -> remove one.
0000354	Bug Report	Two different plots swapping inputs (sum function on x axis).
0000349	Bug Report	Wrong pipe creation when trying to create a pipeline from an input port and an object
0000339	Bug Report	Problem with subsystem execution
0000333	Bug Report	Wrong error message when constructing a miir with fc [0.1 0.6] and fs 1.
0000327	Bug Report	Parfrac constructor ignores a parameter and does not show errors/warnings
0000301	Bug Report	We should not store the password of a connection that was refused because of wrong credentials
0000280	Bug Report	Documentation: some classes do not provide examples or information.
0000277	Bug Report	WEB INTERFACE: graph shows wrong x-axis
0000258	Change Request	Provide a way to load submission info fields from a simple text file.
0000245	Bug Report	iplot labels/legends with real and complex objects
0000244	Bug Report	Problem handling PORT choice
0000220	Bug	the operation remove output to MUX block or input to Demux Block are

	Report	undoable
0000201	Change Request	The class filterbank needs a response method.
0000133	Bug Report	Coefficients reported by ao/detrend are wrong
0000053	Change Request	change zDomainFit so that it returns the best paramters and not the last iteration.
0000027	Change Request	Units conversion to SI method.

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Version 2.3.1 LTPDA Toolbox Software

This table summarizes what's new in Version 2.3.1:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New toolbox features](#)
- [Changes, New Algorithms and Functions](#)

Introduction

This version of LTPDA is 2.3.1. This document lists the changes since V2.3.

New toolbox features

As well as a large number of bug fixes, version 2.3.1 of LTPDA has various significant changes and new features.

- The user can now open multiple instances of the LTPDA workbench.
- The history graph of objects contained in plists inside the history of other objects is now displayed when using **viewHistory**.
- It is now possible to run **LTPDAworkbench** files without opening the GUI: **LTPDAworkbench.run('foo.lwb');**

Changes, New Algorithms and Functions

- **ao/timeaverage** – A method to average time series intervals and return a reduced timeseries where each point represents the average of a stretch of data
- **matrix/crb** – Computes the inverse of the Fisher Matrix for the given, signals, noisespectra and models in the matrix objects
- **matrix/mcmc** – Compute log–likelihood for matrix objects
- **ao/buildWhitener** – Builds a whitening filter for the input AO
- **ao/zeropad** – Can now prepad as well as post–pad
- **ao/cos, ao/acos, ao/asin, ao/atan** – Now include proper error calculation

◀ Version 2.4 LTPDA Toolbox Software   Version 2.3 LTPDA Toolbox Software ➡

LTPDA Toolbox™   [contents](#)   ◀ ➡

Version 2.3 LTPDA Toolbox Software

This table summarizes what's new in Version 2.3:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as **Deprecated methods** in descriptions of changes, below. See also [Summary](#).	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New toolbox features](#)
- [New Algorithms and Functions](#)
- [Changes to the LTPDA Workbench](#)
- [Updates to the Statespace Modelling class](#)
- [Deprecated methods](#)

Introduction

This version of LTPDA is 2.3. This document lists the changes since V2.2.

New toolbox features

As well as a large number of bug fixes, version 2.3 of LTPDA has various significant changes and new features.

- The viewing of object history via the DOT interpreter now does a smarter job for shared sections of the history tree.
- The saving of LTPDA objects to XML files has been completely rewritten. The resulting XML files are now significantly smaller and the reading and writing is much faster.
- We no longer convert LTPDA objects to structures when saving to MAT files. This used to be necessary due to a bug in MATLAB R2008a, but that bug is now fixed and we can skip this step. This results in much faster saving and loading, and in smaller MAT files.
- The **time** class now supports arithmetic operations: plus and minus.
- **pest**/**eval** has been modified to allow evaluation of **pest** & **smodels** with multiple x (independent variables)
- The **smodel** class supports now multiple 'xvar', 'xvals', and 'xunits' values. This allows the evaluation of the outputs of multidimensional fit routines (e.g. lscov, bilinfit)

New Algorithms and Functions

- **ao**/**removeVal** – a method to remove unwanted values from a data set
- **ao**/**rotate** – applies a rotation factor to the input AOs
- **ao**/**buildWhitener** – builds a whitening filter for the input AO

Changes to the LTPDA Workbench

- The workbench now has a "Parameter Overview" dialog which shows a list of all parameters of all blocks on the active canvas. This is useful to compare the different parameters for different blocks. The table can be sorted by block name, parameter key, parameter value.
- The workbench now has a console showing messages, warnings, and errors. The console can be opened and brought to the front when the pipeline is executed. This behaviour is configurable in the workbench preferences.
- The layout of the controls and various panels has been changed to maximise the canvas area. Now all controls and tables are arranged in the tab panel on the left leaving the whole of the right of the screen for the canvas.
- The commenting-out of blocks is now more under the user's control. To support this, the pipeline variables are no longer cleared from the MATLAB workspace when the pipeline is executed. That means that blocks which were previously executed and are then commented out, still have values in the workspace which can be used.
- ctrl-o (cmd-o) now offers to load a workbench from disk. This means that the current workbench will be reset (the user is prompted if the workbench is not saved) and then the workbench from disk will be loaded.
- ctrl-shift-o offers the user to import the pipelines from a workbench on disk in to the current workbench.
- Closing a pipeline (with the little cross on the pipeline window) no longer deletes it from the workbench. Double clicking the pipeline in the pipeline list makes the window appear again.
- To remove a pipeline from the workbench, you can right click in the pipeline list and select "remove pipeline" or select "Pipeline -> Remove from Workbench".
- The "File -> Export Active Pipeline As..." has now moved to "Pipeline -> Export As..."

Updates to the Statespace Modelling class

- The **ssm** property **mmats** was removed, as it was not used.
- The block names are now upper-case, port names lower case. The characters ' ' and '.' are now forbidden.
- The function **resp** returns the step/impulse response of an **ssm** for specified i/o.
- The function "**bode2**" is now "**bode**", and does not need the control toolbox.
- The function **bode** now has default values for the frequency vector.
- **cpsd** and **psd** are new functions. They return the output theoretical noise spectrum given an input psd/cpsd. **cpsd** accepts cross spectral inputs. **psd** takes only diagonal noise inputs, but returns the individual noise contributions as well as the total contribution, **cpsd** returns only the total contribution.
- The input indexes for **kalman, simulate, bode, steadyState, psd, cpsd, resp** was modified. Now are accepted : the strings 'ALL', 'NONE', or a cellstr with mixed port and block-names. Port names may be specified as 'portName ' or 'blockName.portName', 'blockName_portName' for a faster detection. The outputs are now returned in the order provided by the user in the cellstr.
- The function "**reorganize**" was introduced. It does a pre-processing of the system's matrices for the functions **kalman, simulate, bode, steadyState, psd, cpsd, resp**. If these functions are called multiple times, it allows for a faster execution.
- The noise inputs to **simulate** and **kalman** were modified to match each other, and split into two options ((two sided) 'cpsd' and 'variance') so the user can scale the input white noise according to his needs.
- **simulate, kalman, psd, cpsdn bode** and **resp** now call **iplot** if no output variable is provided by the user.
- The **ssm** built-in model facility looks for upper-case m-files names.
- The function **parameterDiff** provides a numerical differentiation of the system's outputs regarding some selected parameters.

- The function "**append**" allows to merge multiple **ssm**s without assembling the input/output blocks. Assemble now works with a single system as an input.
- The conversion from/to a **pzmodel** bug is corrected, gain is now correct.
- The conversion from a **parfrac** is now enabled.

Deprecated methods

The following ao methods are deprecated

- timedomainfit
- straightLineFit
- curvefit
- evaluateModel

◀ Version 2.3.1 LTPDA Toolbox Software   Version 2.2 LTPDA Toolbox Software ▶

©LTP Team

Version 2.2 LTPDA Toolbox Software

This table summarizes what's new in Version 2.2:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as **New repository manager** in descriptions of changes, below. See also [Summary](#).	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New toolbox features](#)
- [New linear fitting tools in LTPDA](#)
- [New repository manager](#)

Introduction

This version of LTPDA is 2.2. This document lists the changes since V2.1.

New toolbox features

Version 2.2 of LTPDA has various new features.

- A reworking of the way connections to LTPDA repositories are handled. Now all connections are managed by the new "LTPDA Repository Manager" object which is created when **ltpda_startup** is executed. *All users should review their use of repository connection code in any scripts and make the appropriate changes to work with the repository manager* ([see below](#)).
- A significant amount of work has been done on fitting tools ([see below](#)).
- A new class called pest (parameter estimation) has been included which aims to capture details of the various fitting procedures in LTPDA. *Objects of this class are now output from all LTPDA fitting function, so some changes to existing analysis scripts and pipelines may be necessary.*

New linear fitting tools in LTPDA

The linear fitting tools are specialised **ao** methods, built around MATLAB's **lscov**. All of them return a **pest** (parameters estimate) object where the fields are containing:

1. Fit parameters
2. Uncertainties on the fit parameters (given as standard deviations)
3. The reduced CHI2 of the fit
4. The covariance matrix
5. The degrees of freedom of the fit.

The following methods are implemented:

- **ao**/**linfit** solves an equation in the form $Y = P(1) + X * P(2)$ for the fit parameters **P**.
- **ao**/**bilinfit** solves an equation in the form $Y = X(1) * P(1) + X(2) * P(2) + … + P(N+1)$ for the fit parameters **P**.
- **ao**/**polyfit** overloads **polyfit()** function of MATLAB for Analysis Objects. It finds the coefficients of a polynomial P(X) of degree N that fits the data Y best in a least-squares sense: $P(1)*X^N + P(2)*X^{(N-1)} +…+ P(N)*X + P(N+1)$
- **ao**/**polynomfit** solves an equation in the form $Y = P(1) * X^{N(1)} + P(2) * X^{N(2)} + …$ for the fit parameters **P**. It handles arbitrary powers of the input vector and uncertainties on the dependent vector Y and input vectors X.
- **ao**/**lscov** is a wrapper for MATLAB's **lscov** function. It solves a set of linear equations by performing a linear least-squares fit. It solves the problem $Y = HX$ where X are the parameters, Y the measurements, and H the linear equations relating the two.

New repository manager

Until the main documentation is updated, the following a some quick tips to using the new repository manager.

Only a single instance of the repository manager exists. It is created when you run **ltpda_startup**. To access the repository manager, you can do:

```
>> rm = LTPDARepositoryManager
---- Repository Manager -----
connections: 0
connected: 0
----------------------------
```

The Repository Manager has both a scripting interface and a graphical user interface. To launch the GUI:

```
>> rm.showGui
```
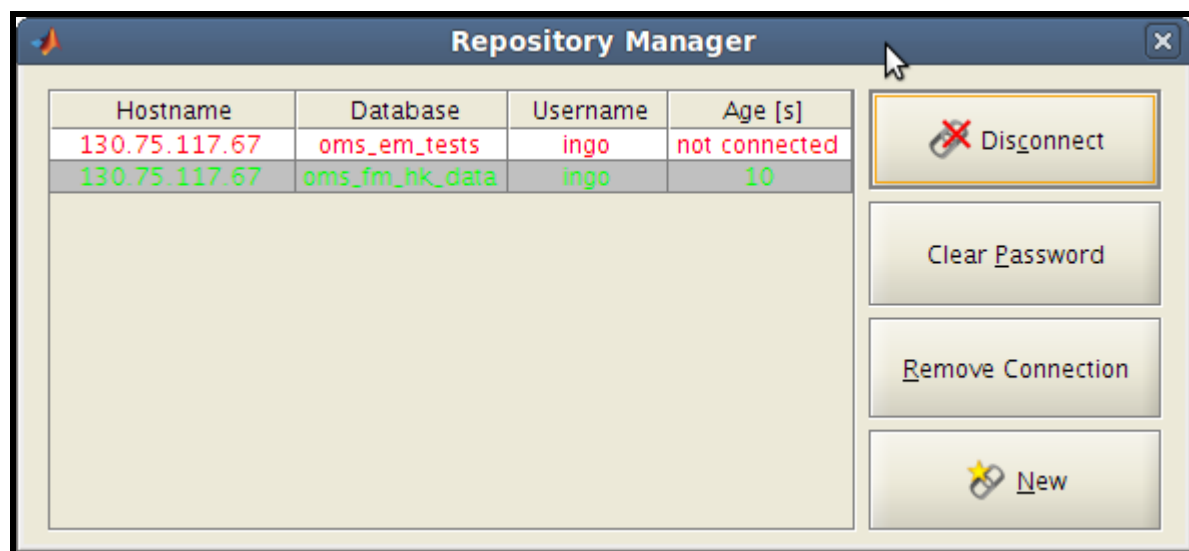

Using the scripting interface, you can create new connections and search for existing connects:

```
>> rm.newConnection('localhost', 'ltpda_test', 'hewitson')
```

specifying the hostname, the database and the username. You can also give the password in plain text to the newConnection command, but that's clearly not advisable.

Making new connections like this allows you to set up your typical connections in your

startup.m file (after executing **ltpda_startup**).

A couple of words about time-outs. The repository manager uses two timers to maintain two aspects of each connection.

The first timer handles whether or not the connection is actually connected to the server. This timer is not user configurable and is set to fire every few seconds so that connections are typically in the disconnected state (you can see this state in the 'age' column of the repository manager gui). Whenever the user tries to use a valid existing connection, the connection is opened and then locked until it is not needed anymore. This is transparent for the user and is handled automatically in methods like **submit**, **retrieve** and **update**.

The second timer is user configurable via the LTPDA preferences. This timer defines when the password for each connection expires. This creates a level of safety so that connections from one user can not easily be abused by other uses. When this password fires, it checks how long each connection has been idle, and clears the password if it is older than the setting in the preferences. Connections with their password cleared are red on the repository manager gui. The next time the user attempts to use a connection whose password has been cleared, they will be prompted to re-enter the password.

©LTP Team

Version 2.1 LTPDA Toolbox Software

This table summarizes what's new in Version 2.1:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	Yes — Details labeled as **Deprecated methods** in descriptions of changes, below. See also [Summary](#).	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New toolbox features](#)
- [New Workbench features](#)
- [New AO methods](#)
- [Main Changes](#)
- [Deprecated methods](#)

Introduction

This version of LTPDA is 2.1. This document lists the changes since V2.0.1.

New toolbox features

Version 2.1 of LTPDA has various new features.

- Analysis Objects have two new properties, **dx** and **dy**, for storing errors/uncertainties on the **x** and **y** data vectors. Currently, the spectral estimation tools (**psd, cohere**, etc) are calculating sample variances and assigning the values to **dy**. In this release, errors are not properly propagated through all additional functions; this is sometimes left to the user to handle. In a future release, the aim is to properly handle error propagation in all functions, where possible. Often this will mean deleting these vectors, or calculating new errors based on the input AOs.
- All user objects now have the following new fields:
    - description: a string that can be assigned to describe the object
    - procinfo: a **plist** containing additional information produced during processing by an algorithm. The contents of the **plist** depend on the algorithm being applied.
- Documentation (>>**help** method) has been significantly simplified. Now the details of the parameters for configuring a method are dynamically created and presented to the user in the MATLAB help browser after clicking on the 'Parameter Sets' link in the help text.
- New user classes:
    - **smodel**: defines parametric models of an X variable which can be combined and evaluated to AOs.
    - **matrix**: a container class which can be used to group together other user objects and perform some basic matrix operations on the group of objects, not necessarily on the data in the objects.
    - **filterbank**: represents a bank of digital filters. It can be either a 'parallel' or 'serial'

filter bank.

- All user classes now support built-in models.
- LTPDA now provides direct access to NDS (version 1) servers using the AO constructor 'From NDS Server'. This requires the (3rd party) NDS client mex files to be available on the MATLAB path.
- The repository interactions are now done through purpose written java code so that no database toolbox is required. Also, the repository GUI (repogui) has been removed and the various features are replaced by individual GUI components which can be launched either via the workbench or via the new workspace browser (**workspaceBrowser**).
- A new GUI, the Workspace Browser (workspaceBrowser) allows fast access to various toolbox features as well as showing the current LTPDA user objects in the workspace.

New Workbench features

Apart from various cosmetic improvements, the workbench has the following new features:

- The workbench now has a 'recent files' list under the File menu
- The workbench now has the concept of an 'Execution plan'. Any or all of the pipelines in the workbench can be entered in the execution plan. The full plan can then be executed in the order given. To edit the plan: File->Plan->Edit. To run the plan: File->Plan->Run.
- New block types:
    - 'From Pipeline' block allows the user to link together pipelines in the current workbench. Double clicking the 'From Pipeline' block allows the user to select the source to be from the output of any block on any other pipeline in the workbench. This goes along with the new ' Plan' concept.
    - Mux/Demux blocks allow objects to be put in to, or removed from, arrays.
    - 'To Workspace' block pushes the objects to the MATLAB workspace.
- A graphical representation of the current pipeline can be exported as a JPEG image.
- The workbench has a new panel called the 'Shelf'. Users can store subsystems on the shelf and pull them off anytime in the future to include them in a different pipeline. The Shelf supports nested categories, exporting and importing of categories, and adding of subsystems from the canvas or from the shelf context menus.
- The pipelines of the current workbench are now shown in a tree-like structure, where the subsystems of each pipeline can also be seen.
- The block-property table has been re-coded to more nicely present the options to the user.
- The parameter list table now supports:
    - activating and deactivating individual parameters
    - double-click in-place editing of the key and value
    - selection of parameters with options via drop-down menus
    - selection of boolean parameters via check-boxes
    - editing of parameter values in special editors via the edit button ('...')
    - changing the type of a parameter (double/char/boolean) via the context menu
- The workbench has the following new block button commands:
    - saves the outputs of selected objects to disk.
    - displays the data of the selected objects in tables.
    - present a report on the selected objects.
- New menu entries for
    - show quickblock (ctrl-b)
    - edit canvas info (ctrl-i)
    - search workbench (ctrl-f)
- The element searching now has new features, like the ability to choose whether to search for block names, algorithms, or both. Info about the block **plist** is now shown as a tooltip on the results table, and double clicking a result highlights that block in the workbench.

The workbench now supports plain text documents.
- The workbench output files have seen some optimisation and are much smaller than in earlier versions.
- A new special editor for keys: 'xunit', 'xunits', 'yunit', 'yunits', 'unit'.
- A new special editor for assembling a list of colors that can, for example, be passed to **iplot**. So far this will activate for any parameter with key 'colors' or 'linecolors'.
- A new special editor for a cell-array of line-styles, like that used in configuring iplot.
- A new special editor for building a list of markers. Activated for key 'markers'.
- A new special editor for the key 'plotinfo'. Allows to edit the line style, line width, color and marker for AOs.
- The workbench can now be attached to any or all output objects from all blocks on a pipeline. The workbench can then be loaded from the object, whether it's on disk or in the MATLAB workspace. To attach the workbench to an object, select the block and set the property 'Attach Workbench' in the property table. To import a pipeline from an existing object, 'File->Import Pipelines->From LTPDA Object...'.
- Blocks can be 'Commented Out' (deactivated) on the pipeline. These 'commented out' blocks will not be considered during pipeline execution.
- The workbench now supports some direct interaction with an LTPDA Repository, avoiding the need to go through the repository GUI. For example, it is possible to:
  - (dis-)connect to a repository via the (dis-)connect ( ) button(s)
  - To submit the output of the selected blocks to the repository (pipeline must have been executed first) using the submit ( ) button.
  - Query a repository using the query button ( ). Results of the query can be used to automatically build constructor blocks on the current pipeline.
- The workbench now has an autosave feature. This can be activated in the preferences ('File->Preferences...') and the interval can be set.

New AO methods

- **ao/table**: display the data in the AO in a graphical table.
- **ao/normdist**: computes a normal distributed pdf based on the mean and variance of the data in the AO.
- **ao/corr**: computes the sample correlation matrix using Pearson's product-moment method.
- **ao/linSubtract**: subtract linear contributions to a data set.
- **ao/filtSubtract**: subtracts a frequency dependent noise contribution from an input ao.
- **ao/hypot**: robust computation of square-root of the sum of the squares.
- **ao/xfit**: fits a function of x to a data set using simplex non-linear fitting routine.
- **ao/tdfit**: fits a model to a set of input and output data. The parametric model (**smodel**) should represent the system(s) which gives the outputs for the given inputs.

Main Changes

- All user-created objects now have a Universal Unique Identifier (UUID) which is reset whenever the object is modified (a history step is added).
- Cross-spectral estimators (**ltfe, cohere**, etc) now only accept two inputs and return a single output.
- **ao/lscov** now returns a vector of AOs, each containing a single parameter, with units and errors.
- The statespace modelling class has been heavily rewritten and as such behaves differently than in previous versions. Review the LTPDA documentation for the new behaviour of this class.
- On Windows, Microsoft's Visual C++ compiler version 2008 is used to compile the mex files. That means that the appropriate runtime environment must be installed. The

installation files for 32-bit (**vcredist_x86.exe**) and 64-bit (**vcredist_x64.exe**) are included in the ltpda/src directory.
- updated versions of **fft** and **ifft**.
    - **fft** – added 'plain' option which output the standard matlab **fft** results. Available parameters are now 'type' = 'plain', 'one' (onesided), 'two' (twosided) – frequency bins now are correctly calculated
    - **ifft** – option twosided is no more necessary, the algorithm now is capable to distinguish between 'plain' fft, twosided fft, onesided fft with odd nfft or onesided fft with even nfft
- **ao/iplot** now supports the **plotinfo** field of an AO. This means that each AO can be given specific plotting attributes. The **plotinfo** field can be set using **ao/setPlotinfo**. It is a parameter list of properties and values; see help of **ao/iplot** for supported properties. The values in **plotinfo** override all other inputs to **iplot** for that particular AO.
- The AO methods **polyfit, zDomainFit** and **sDomainFit** now return matrix objects. This change is not backwards compatible.

Deprecated methods

The following ao methods are deprecated

- **ao/curvefit**: has been replaced by **ao/xfit** and as such will no longer be maintained.
- **ao/pwelch**: is still deprecated. Use **ao/psd** instead.

©LTP Team

LTPDA Toolbox™   [contents](#)   ◀ ➡

Version 2.0.1 LTPDA Toolbox Software

This table summarizes what's new in Version 2.0.1:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	No	Printable Release Notes: [PDF](#)

- [Introduction](#)

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.

◀ Version 2.1 LTPDA Toolbox Software   Version 2.0 LTPDA Toolbox Software ➡

©LTP Team

Version 2.0.1 LTPDA Toolbox Software (LTPDA Toolbox)

LTPDA Toolbox™   [contents](#)   ◀ ▶

Version 2.0 LTPDA Toolbox Software

This table summarizes what's new in Version 2.0:

New Features and Changes	Version Compatibility Considerations	Fixed Bugs and Known Problems	Related Documentation at Web Site
Yes Details below	No	[Bug Reports](#) at Web site	Printable Release Notes: [PDF](#)

- [Introduction](#)
- [New toolbox features and changes](#)

Introduction

There has been a significant amount of work done since V1.9.3.

New toolbox features and changes

The highlights are:

- A new Graphical Programming interface for the construction of signal processing pipelines (LTPDAworkbench)
- New data whitening methods are included, **ao/whiten1D** and **ao/whiten2D**.
- Most methods of the AO class should now support multiple outputs, in addition to the vector output. For example, the following are supported:
  - **a = lpsd(b1,b2,b3)**
  - **[a1,a2,a3] = lpsd(b1,b2,b3)**
- Model fitting tools:
  - **curvefit** for doing non-linear least-squares, now including weights
  - **lscov** for doing linear least-squares fitting
  - **straightLineFit** is a wrapper of **lscov** for fitting data with a straight-line
- Operators, +-.*./*/, all conform to more standard MATLAB type behaviour. For example, adding a single AO to a vector of AOs results in a vector of AOs where the single AO is added to each element of the vector.
- Digital filtering with IIR filters now has two possible ways of applying a bank of filters. By specifying the '**bank**' parameter, the filters will be applied as a '**serial**' bank or a '**parallel**' bank.
- Two new noise-generators are included, **noisegen1D** and **noisegen2D** (which can produce correlated data streams).
- AOs can now be built from a set of built-in models. Users can also write their own models and point LTPDA to a directory (or more than one) containing those model files.
- The user preferences of LTPDA are now controlled via a graphical user interface. To adjust the user preferences, run **LTPDAprefs**.
- Two new transfer function representations are present as user classes of this version of the toolbox. The class **parfrac** allows the user to build transfer function models that are a series of partial fractions; the class **rational** allows the user to build a transfer function as a rational polynomial in s. Some converters also exist to go between these and the existing transfer function models, **pzmodel** (pole/zero models), **miir** (IIR filters) and **ssm** (state-space models).

- A new method, **ao/heterodyne**, to heterodyne data at a given frequency.
- All transfer function representations and digital filters now can be assigned input and output units. This means that units are now propagated through fully through any analysis.
- A new **rebuild** method to rebuild objects from their self-contained history trees.
- New graphical user interfaces
  - modelViewer – a viewer for the built-in ao and ssm models
  - constructor – a helper for writing object constructors (rewritten)
- New convenient AO methods to apply gain and to add an offset to data (**scale, offset**).
- Transfer function estimators now return error terms in the **procinfo** field of the result **ao**.
- Units are not automatically simplified. Instead the user can now do that on demand with the **simplifyYunits** method. There are also methods to convert between Hz and s: **HzToS** and **sToHz**.
- Digital IIR filters are now initialised to a state based on the input data. This should reduce start-up transients.
- Data in AOs can be converted from one type to another using the new **ao/convert** method.
- Pole/zero models (**pzmodel**) now support a delay parameter.