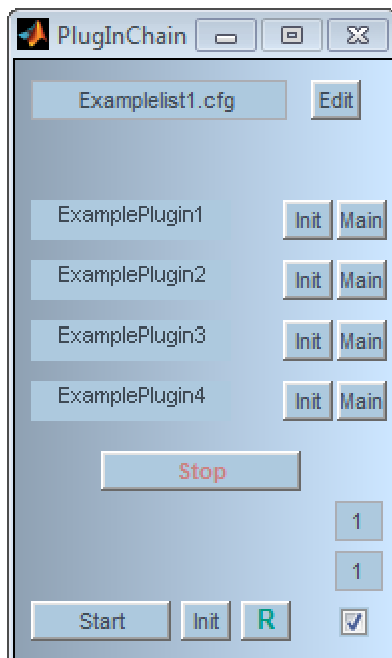
PlugInChain documentation

PlugInChain for Mathwork's MATLAB
Version 1.00.0
June 30, 2011

Medizinische Physik
Carl-von-Ossietzky University of Oldenburg

Marion Wirschins, Thomas Rohdenburg, Tobias May, Stephan D. Ewert

Corresponding author: stephan.ewert@uni-oldenburg.de



Copyright © 2011, Medizinische Physik, Carl-von-Ossietzky Universität Oldenburg.
Some rights reserved.

Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-nc-nd/3.0/

1 Introduction ... 3
1.1 Directory structure ... 4
1.2 Basic concept ... 4
1.3 Graphical User Interface (GUI) ... 5
1.2.1 Pluginlist panel ... 6
1.2.2 PlugInChain Framework ... 7
1.2.3 Plugins ... 7
1.2.4 Signal processing control ... 8
1.2.5 Batch processing ... 8
1.4 Command line functions ... 9
1.4.1 Initialization of plugins ... 9
1.4.2 Run processing ... 10
1.5 Examples of starting the PlugInChain ... 10
2 The PlugInChain framework ... 11
2.1 Basic structure ... 11
2.1.1 ConfigStruct ... 11
2.1.1.1 GlobalSettings ... 11
2.1.1.2 Plugin ... 11
2.1.2 Signal ... 12
2.2 Plugin structure ... 14
2.2.1 Initialization ... 14
2.2.2 Block size ... 15
2.2.3 _init function ... 15
2.2.4 _main function ... 16
2.3 Plugin lists ... 16
2.3.1 Use existing plugin lists ... 17
2.3.2 Create new plugin lists ... 17
2.3.3 Assessing the history of loaded plugin lists ... 18
3 Plugins ... 19
3.1 Use plugins ... 19
3.2 Change plugins ... 19
3.3 Write new plugins ... 19
3.3.1 Example 1 - IIR low-pass filter ... 20
3.3.2 Example 2 - Adaptation loops ... 22
3.3.3 Useful features ... 25
3.4 PlugInChain rules ... 25
3.5 GlobalSettings of the pluginchain ... 26
3.5.1 Audiotory Profile ... 26
3.5.2 Performance parameters ... 26
4 Quickstart ... 28

1 Introduction

The ‘PlugInChain’ is a modular signal processing framework for research and development in MATLAB. It offers the functionality to build and to configure processing chains of individual signal-processing modules which are termed ‘plugins’ in the following. A plugin may contain fundamental or advanced signal processing, e.g., a linear filtering. A chain of individual plugins is defined by a simple text file, a ‘pluginlist’, and can be edited and executed via a graphical user interface or can be directly executed via the MATLAB command window. The user can refer to existing signal-processing functionality in the form of plugins and pluginlist and can built more complex signal-processing chains without knowing all the implementational details and without necessarily programming new own plugins.
The PlugInChain offers frame-based signal-processing with arbitrary frame size. It provides a well defined interface between the plugins and the easily accessible and extendable structure of the plugins. Plugins can have configuration parameters that can be controlled via the pluginlist. In this way, it is possible to initialize a plugin with different settings without changing its source code. In order to store complete signal processing chains, only the pluginlist file (including the related configuration parameters) is necessary.
Users are permitted to create custom plugins and plugin lists or derive them from included examples and to distribute their own creations in any form under any license independent of the PlugInChain. The plugin and pluginlist structure and the included examples are therefore licensed under Creative Commons Attribution 3.0 Unported (CC BY 3.0) while the remaining PlugInChain is licensed under Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported (CC BY-NC-ND 3.0).

The PlugInChain package is available at http://medi.uni-oldenburg.de/pluginchain or on demand per email from the corresponding author stephan.ewert@uni-oldenburg.de.
After downloading this zip-archive, unzip the included folder ‘pluginchain’ to your harddisk, e.g., your Matlab working directory. Alternatively, create a directory for the PlugInChain with a name of choice and unzip the contents of the folder ‘pluginchain’ in this zip-archive there (the folder structure of ‘pluginchain’ is shown in Fig. 2.2). Third-party or custom plugins should be placed in the subfolder ‘pluginchain/plugins’.

This section will give a general overview about the functionality of the PlugInChain. The plugin structure and the use of plugin lists will be explained in more detail in section 2. Section 3 presents some more detailed information about how to use and to create plugins. The PlugInChain is, in principle, suited to process any type of data but is commonly used by the authors to process audio material. This documentation thus often refers to signals or audio data, without implying a limitation of the PlugInChain.

This work was supported by the Bundesministerium für Bildung und Forschung (BMBF) “Modellbasierte Hörsysteme”.

1.1 Directory structure

After extracting the PlugInChain package you should find a folder structure as presented in Fig. 2.2

In the working directory[1] ('pluginchain'), the starting routines (*pluginchain.m* and *pluginchaingui.m*) for the PlugInChain, the additional help function *makelist.m*, some pictures for the GUI and a 'ReadMe.txt' file are located. The main routines of the PlugInChain are located in the subfolder 'base'. All available plugins are stored in the subfolder 'plugins' (some plugins need extra databases or files that usually are stored in their own subfolder within the folder 'plugins'). All general tools which may be used by the core routine or by several plugins are stored in 'pluginchain_tools'. The folder 'pluginlists' contains the plugin lists configuration files for using plugins or special signal processing chains of plugins with the PlugInChain. Additional audio files may be stored in the folder 'waves'. Some special demo files for a special use of the PlugInChain can be found in 'demo'. The documentation files are located in the folder 'docs'. In the subfolder 'html' (beside some pictures for the html documentation file) some MATLAB-generated documentation of m-files are located.

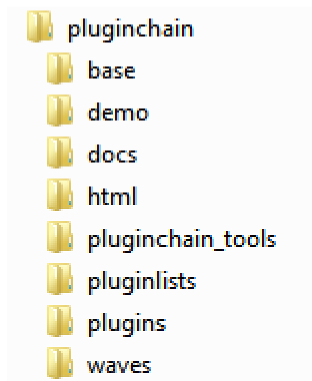


Figure 2.2: Folder structure of the PlugInChain package.

1.2 Basic concept

The functionality of the PlugInChain structure is outlined in Fig.1.1. The main function coordinates the processing without having any knowledge about the particular plugins. The processing steps of the PlugInChain can be divided into three phases, namely

1. Initialize pluginchain main file with loaded plugin list
2. Initialize all plugins
3. Run (frame-based) processing routines of all plugins.

[1]The PlugInChain working directory can be visualized by the MATLAB command `pwd`

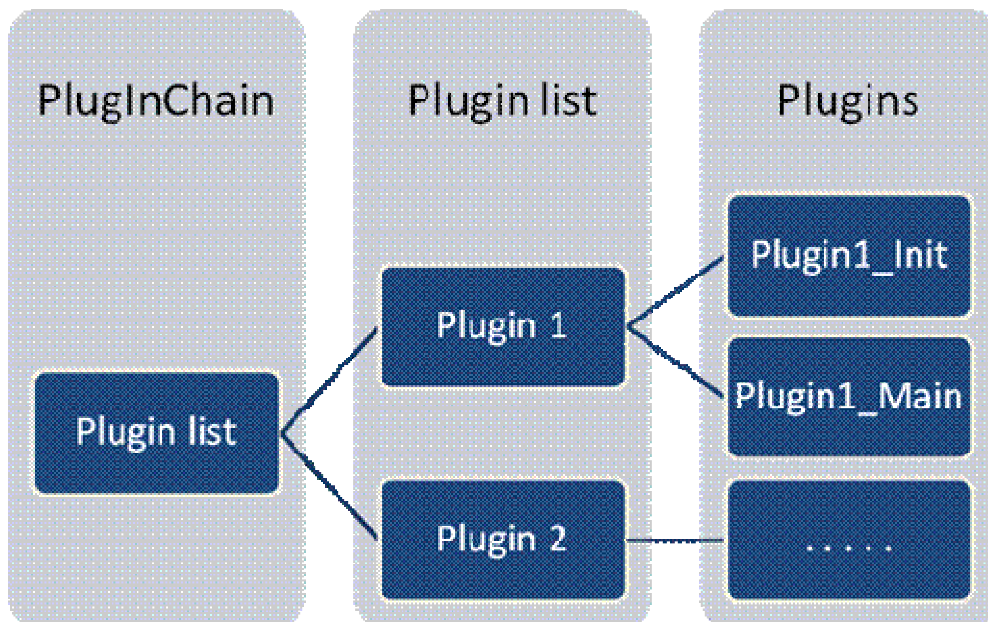


Figure 1.1: PlugInChain structure.

The first step initializes the PlugInChain by loading the desired plugin list and building up the graphical user interface, if required. In the plugin initialization phase, all relevant variables are initialized and stored in the plugin 'Settings' structure. After every plugin is initialized successfully, the actual (frame-based) processing of the input signal is accomplished. Each plugin works independent of the other plugins, it is just necessary that the algorithm of a plugin can work with signal type produced by the prior plugin. The incoming data is managed by the framework and stored in separate arrays, so that each plugin can run with an arbitrary frame size. The PlugInChain can be controlled either from a Graphical User Interface (GUI) or from the command line.

1.3 Graphical User Interface (GUI)

The Graphical User Interface was designed to make the use of the PlugInChain more comfortable. It has two different modes – the usual User mode for using and programming plugins and a developer mode that adds some functionality for editing the files of the PlugInChain framework itself.

The GUI-based version of the PlugInChain is started by typing

```
pluginchaingui('pluginlist','ListName.cfg');
```

in the MATLAB command window (from the folder where the PlugInChain is installed). This command automatically initializes the PlugInChain and starts the GUI with the specified pluginlist 'ListName.cfg'. The appearing graphical user interface is shown in Fig. 1.2 and is divided into three different control panels.

1. Plugin list
2. Plugins
3. Signal processing items (Reset, Initialize, batchprocessing)

If the function `pluginchaingui` is called without an input argument or if the specified pluginlist does not exist, the GUI is opened with a default pluginlist or the last loaded list displaying a warning message.

It is possible to start `pluginchaingui` in developer mode:

```
pluginchaingui([…],'Developermode',1).
```

This requires the source code (m-files, not only executables) of the PlugInChain and will add additional buttons to the GUI to open the m-files of the framework directly in the MATLAB editor. The buttons are arranged in an additional control panel directly beneath the Pluginlist panel.

In the following, the different panels will be described in more detail.

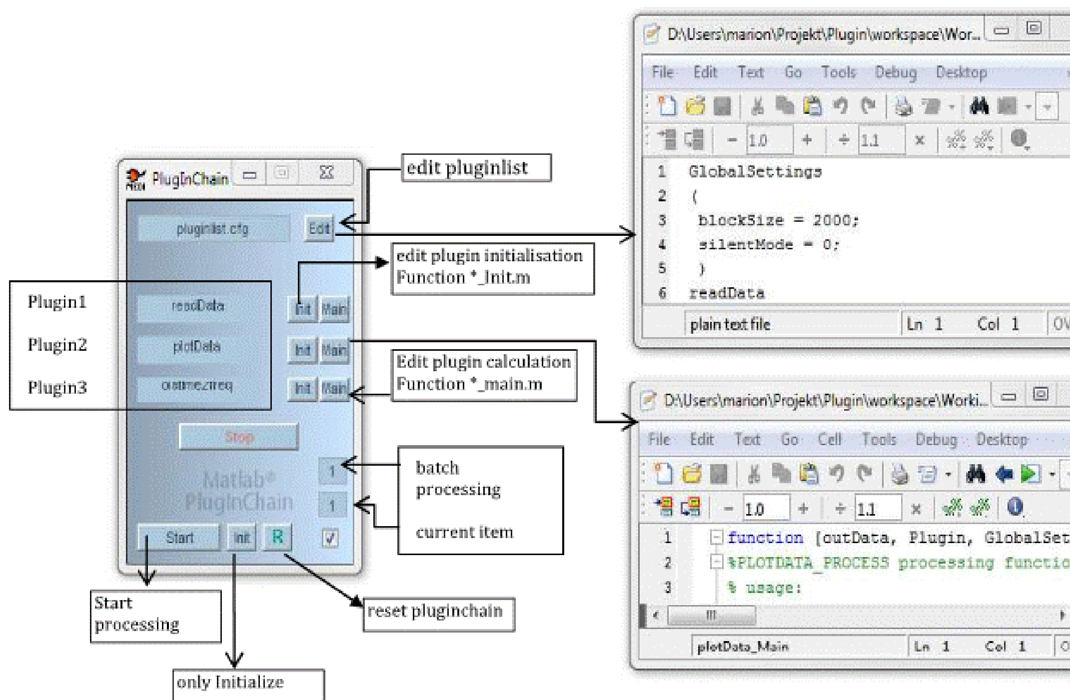


Figure 1.2: Graphical User Interface (GUI) of the PlugInChain.

1.2.1 Pluginlist panel

The first panel shows the loaded pluginlist, which defines any number of cascaded plugins and optional configuration parameters for them. A pluginlist can be loaded by *clicking with the right mouse button* on the plugin list name *display*. The current plugin list can be edited with the adjacent button Edit and is shown on the upper right-hand side of Fig. 1.2.

The first uncommented line of the pluginlist contains the string 'GlobalSettings'. After that, all plugins are listed (in every line only one plugin name). The signal processing flow is de-

termined by the order of the plugins. Optionally, configuration parameters for the plugins are specified in the pluginlist, otherwise the default values defined in the plugin itself are taken.

To comment out a line the usual MATLAB sign '%' is used.

To define configuration parameters for a plugin or the 'GlobalSettings', the following syntax has to be used:

- plugin name (or GlobalSettings), new line
- Line with only opening bracket '(', new line
- expression for the configuration of the plugin e.g. 'blocksize=1024;', new line (new line for each parameter)
- Line with only closing bracket ')'.

This is demonstrated in the following example.

```
GlobalSettings
(
% blockSize = 1024;
)
readData
(
command = 'wavread';
targetname = 'mywavfile.wav'
)
Filterbank
Resampler
plotData
```

The first uncommented line in the file comprises `GlobalSettings`, the following user parameter `blockSize` in line 3 is commented out. For the first plugin in the list `readData` there are two user parameters set. They define that the reading option is `'wavread'` and specify the source for signal file to be loaded. For the last three plugins no configuration parameters are defined.

It is also possible to use plugins multiple times. For example, the pluginlist may use the plugin 'plotData' two times, first after loading the audio data and second after transforming the signal. Every 'instance' of a plugin gets its own configuration parameters, so that every time the plugin 'plotData' appears in the list, it may have other configuration parameters.

1.2.2 PlugInChain Framework

In developer mode, the second section of the GUI contains the buttons for editing the PlugInChain files. It is recommended not to edit the files of the PlugInChain framework, except if it is necessary to define new parameters in the `GlobalSettings` of the PlugInChain.

1.2.3 Plugins

All plugins which belong to the loaded pluginlist (also named 'configuration file') are described in detail in section three. One can access the configuration parameters of a plugin by

pressing the edit button for the pluginlist. This will open the file ‘Pluginlistname.cfg’ in the MATLAB editor. In the current example, ‘Pluginlist.cfg’ is shown in the editor in the upper right-hand side of Fig. 1.2. It was accessed by pressing the button edit just right of the field displaying the loaded pluginlist.
The buttons on the right hand side of the listed plugins enable control over the initialization file **_init.m* and the processing main file **_main.m*. They are only needed for developing new plugins and to provide quick access for editing the plugin files. In the current example, the main file of the plugin *‘plotData_main.m’* is shown on the lower right-hand side of Fig. 1.2.. It was accessed by pressing the button Main right of the field displaying the plugin ‘plotData’.

The plugin structure and corresponding files will be discussed in more detail in section 3.

1.2.4 Signal processing control

The button Start can be used to start the signal processing of the PlugInChain. First, all plugins are initialized by calling the function *pluginname_init.m* of all plugins from the frameworks base function *pluginchain_init.m*. The first plugin ‘readData’ is reading during initialization phase the needed parameters for building up the ‘signal’ structure (e.g. from the header of a wav-file). The processing of the input signal is then performed by calling the function *pluginname_main.m* of each plugin from the frameworks processing file *pluginchain_process.m,* whereas the function *readData_main.m* reads the signal data and stores it in the ‘signal’ structure of the PlugInChain. The execution of the PlugInChain can be stopped by pressing the button Stop. Note that under MATLAB this is only possible between different processing steps and will not break a long calculation.

1.2.5 Batch processing

Each plugin is initialized with loading the configuration parameters from the configuration file (see lower right-hand side of Fig. 1.2). So far, a parameter was assigned exactly one value. If the PlugInChain should process data with several different values for a specific configuration parameter, a ‘batch variable’ can be used. A batch variable can be defined by appending the character # at the end of the parameter name. For example the configuration parameter `myVariable=0.3` can be defined as a batch variable `myVariable# = {0.3,0.4,0.5}`, whereby its multiple values (alphanumerical) need brackets ‘{}‘ and are stored in a cell array.
It's also possible to use MATLAB code to initialize parameters. The batch processing can, e.g. be really useful if several audio files should be processed. Assume that the text file ‘batch.txt’ contains a list of audio files as shown in Fig. 1.3.

```
1: ./waves/PeterHahne_kurz.wav
2: ./waves/PeterHahne.wav
3: ./waves/TestMeddis.wav
4: ./waves/testsinmod.wav
```

Figure 1.3: Text file batch.txt containing a list of audio files.

Typing the following line in the configuration file

```
targetname# = textread('batch.txt',[char(37),'s'],'delimiter','\n');
```

loads the file `batch.txt` into the batch variable `targetname#.`

The current iteration can be controlled in the GUI. Batch processing only takes place, if the checkbox (at the bottom right in Fig. 1.2) is activated. The batch processing number (upper field for batch processing in Fig. 1.2) in the GUI for the 'current batch processing item" is set automatically to the length of the batch variable. If no batch variable is defined it is set to 1. If more than one configuration parameter is defined as batch variable, they must all have the same number of iterations.

1.4 Command line functions

The PlugInChain can be also executed from MATLAB'S command line. There are several ways to do this. The recommended way is to use the starting function

```
pluginchain('pluginListname.cfg');
```

from the folder where you installed the PlugInChain. Note, that the command line mode (without GUI) function always needs a valid plugin list file as input argument. Only the function *'pluginchaingui.m'* allows the user to start it without input argument, as long as a default plugin list exists.

The following three commands that are called by the `pluginchain` command can also be directly started from the directory 'base':

```
[Signal, ConfigStruct] = pluginchain_main('pluginListname.cfg');

[Signal, ConfigStruct] = pluginchain_init(Signal, ConfigStruct);

[Signal, ConfigStruct] = pluginchain_process(Signal, ConfigStruct);
```

The first input parameter `'pluginListname.cfg'` contains the desired plugin list (string). The output parameters are two MATLAB structures, namely the `Signal` and the `ConfigStruct` structure. The signal structure is meant to contain the data which is processed. After initializing the PlugInChain, the `Signal` structure contains just the signal header describing the four different signal domains (time, channel frequency and modulation frequency domain). The `ConfigStruct` structure contains as the first sub-structure the `GlobalSettings`, and as second a structure for the plugins which are overwritten when initializing the plugins. The `ConfigStruct` contains e.g. information about the current plugin list and parameters that are globally and must be available for the plugins. The hierarchical design of these MATLAB structures will be described in more detail in Sec. 2.1.2 and Sec. 2.1.1, respectively.
The direct execution of *pluginchain_main.m, pluginchain_init.m, pluginchain_process.m* can be useful, if direct access to the `Signal` and the `ConfigStruct` structure is required, e.g., to edit or overload the data in the `Signal` structure or to retrieve the data from the `Signal` structure for further use.

1.4.1 Initialization of plugins

In order to initialize the PlugInChain, the two structures are passed to the initialization function

```
[Signal, ConfigStruct]= pluginchain_init(Signal, ConfigStruct);.
```

All settings which are necessary to process the audio material are initialized and stored for every plugin in the `Plugin` structure as a sub-structure of `ConfigStruct`. The PlugInChain `GlobalSettings` stores information about the plugin list, paths and some other needed control parameters.

1.4.2 Run processing

The block-based processing of the input signal can be started by

```
[Signal,ConfigStruct]=pluginchain_process(Signal, ConfigStruct);.
```

Note that the input arguments of the function *pluginchain_process.m* require the initialized `Signal` and the initialized `ConfigStruct` structure (contains the plugins as well as the 'GlobalSettings').
If no block processing is needed it is recommended to process the whole signal in one, because it's much faster and therefore is chosen as the default constellation.
The actual progress is reported by a waitbar showing the percentage of completed processing. It can be turned off by redefining the value in the `GlobalSettings` as
`'showProgessbar = false'.`
The processed signal is stored in the `Signal` structure. But depending on the last plugin of the actual plugin list, the output also can be

- stored to a wavefile,
- stored to a file,
- plotted into a figure

just to name a few examples.

1.5 Examples of starting the PlugInChain

In the directory where all files and folders for the PlugInChain are stored, two functions are available to run the PlugInChain without or with GUI:
'pluginchain.m' (without GUI) and
'pluginchaingui.m'. (with GUI)
For the *pluginchaingui* function most parameters are optional. If used, they always must appear in pairs of variable name and variable value. In the following the possible commands are shortly described:

1. `pluginchaingui();`
   Starts the PluginChain with GUI in the directory 'base' with the default plugin list.

2. `pluginchaingui ('pluginlist','pluginlistname.cfg')`
   Starts the PlugInChain with GUI using the specified plugin list 'pluginlistname.cfg', as long as not specified `GlobalSettings.guidefault = false`.

3. `pluginchaingui ('pluginlist','pluginlistname.cfg','gui','true')`
   Starting the PlugInChain with GUI loading the plugin list 'pluginlistname.cfg', the last two arguments are optional.

4. `pluginchaingui('pluginlist','pluginlistname.cfg','Developermode',1)`
   Starts the PlugInChain with GUI in 'Developer mode' with the plugin list 'pluginlistname.cfg'.

5. `pluginchain('pluginlistname.cfg')`
   Starts the PlugInChain without GUI with the plugin list 'pluginlistname.cfg'.

Note, that the command line mode (without GUI) function always needs a valid plugin list file as input argument. Only the function *'pluginchaingui'* allows the user to start it without input argument, as long as a default plugin list exists.

2 The PlugInChain framework

2.1 Basic structure

The PlugInChain framework relies on two structures in the MATLAB workspace: `ConfigStruct` and `Signal`.

2.1.1 ConfigStruct

`ConfigStruct` is a structure that contains all required information for the processing except for the data that is to be processed. Besides the home directory path, it contains a substructure `GlobalSettings` and a `Plugin` substructure for each plugin.

2.1.1.1 GlobalSettings

Here all information required by the framework itself or information that has to be available for several (not specified) plugins is stored. Beside some processing parameters and control flags, it contains the plugin list, all required paths and the auditory profile (if used). Furthermore, the parameters for frame-based and batch processing and GUI control are defined here. Some of these values can change during processing.

2.1.1.2 Plugin

For each plugin in the plugin list, a substructure `Plugin(k)` exists in `ConfigStruct`. The index 'k' numbers the substructures in the order as they appear in the plugin list. Each `Plugin(k)` substructure consists of the following five fields:

```
Plugin(k).name          = 'pluginname'
         .version       = '2'
         .Settings      = [1x1 struct]
         .delay         = 0
         .userSettings  = [1x1 struct]
```

The plugin name corresponds to the name of the main-file of the plugin and is read out automatically from the plugin list. The sub-field version is intended to keep track of the development stage of a plugin. An appropriate history located in the initialization file of the plugin should document important changes. The structure Settings contains the information and parameters for the plugin. This can be plugin-specific parameters like a sampling rate, a block size or filter coefficients. All relevant variables which are required for the signal processing step of each plugin are stored in the Settings structure. During processing, only the data, the GlobalSettings and the Settings structure of the corresponding plugin are passed to the processing function of the plugin. The field delay determines if the plugin introduces a time delay. In the last field userSettings, the plugin specific configuration as defined in the currently loaded plugin list is stored.
After the plugin initialization phase, a structure Plugin(k) with an individual Settings substructure exists for each plugin in the plugin list. For the plugin list 'pluginlist.cfg ', shown in the upper right-hand side of Fig. 1.2, the ConfigStruct structure contains the following six elements

```
          home     = '….\Plugin\workspace\WorkingCopyMain\base'
           name    = 'ConfigStruct'
        version    = '2'
          delay    = 0
GlobalSettings     = [1x1 struct]
         Plugin    = [1x3 struct]
```

Where the Plugin structures look like as follows

```
      name         = <1x1 cell>
      Settings      = <1x1 struct>
      version      = '2.0'
      delay        = 0
      userSettings = [1x1 struct]
```

2.1.2 Signal

The Signal structure contains the data itself and metadata, e.g. about the domain. The convention of storing data assumes the following four dimensions (which can be partly singular):

```
Signal.data = [Time x Audio channel x Frequency channel x Modulation frequency channel]
```

The metadata corresponding to each of the signal dimension is stored in the four dimensional structure Signal.Dimension with the following fields:

```
Signal.Dimension(1).name   = 'Time'
                   .length = 2048
                   .argvec = {1/fs 0}
```

The field name describes the actual domain. The other fields length, argvec are domain-dependent and specify the size and the sampling rate. If one uses the plugin 'readData' it initializes the signal with a further field units for each domain.
For the default plugin list 'pluginlist.cfg' which is shown on the upper right-hand side of Fig. 1.2, the Signal structure after the plugin initialization phase looks like this:

```
Signal.Dimension(1).name = 'Time'
                    .length = 1
                    .argvec = {[0.0938] [0]}
                    .units = 's'
Signal.Dimension(2).name = 'Channel'
                    .length = 1
                    .argvec = {[]}
                    .units = []
Signal.Dimension(3).name = 'Frequency'
                    .length = 4096
                    .argvec = {[3.9063] [0]}
                    .units = 'Hz'
Signal.Dimension(4).name = ''
                    .length = 0
                    .argvec = 0
                    .units = []
```

The field `argvec` of the signal domain `'Time'` consists of two values. The first value represents the sampling period and the second value determines the current (processing) position within the signal in seconds. This value is still zero as no data is loaded during the initialization phase. Note that the `Signal` structure contains in `Signal.data` at every point in time only the data of the actual domain (demonstrated in the following example). After loading the first audio frame to the `Signal` structure, the structure `dimensions` would become

```
   Signal =  data: [150012x1 double]
             Dimension: [1x4 struct]
```

with the following domain descriptions

```
Signal.Dimension(1).name   = 'Time'
                    .length = 150012
                    .argvec = {[6.2500e-005] [9.3758]}
                    .units  = 's'
Signal.Dimension(2).name   = 'Channel'
                    .length = 1
                    .argvec = {[]}
                    .units  = []
Signal.Dimension(3).name   = 'Frequency'
                    .length = 1
                    .argvec = {[3.9063][0]}
                    .units  = 'Hz'
Signal.Dimension(4).name   = ''
                    .length = 1
                    .argvec = 0
                    .units  = []
```

After appending a plugin that, e.g., transforms a time signal into the frequency domain (and can only be further processed by a frequency-based plugin), the four-dimensional signal structure would look like this:

```
Signal =  data: [1x1x4096 double]
          Dimension: [1x4 struct]
```

with the following domain description

```
Signal.Dimension(1).name   = 'Time'
                   .length = 1
                   .argvec = {[6.25e-005] [9.3758]}
                   .units  = 's'
Signal.Dimension(2).name   = 'Channel'
                   .length = 1
                   .argvec = {[]}
                   .units  = []


Signal.Dimension(3).name   = 'Frequency'
                   .length = 4096
                   .argvec = {[3.9063] [0]}
                   .units  = 'Hz'
Signal.Dimension(4).name   = ''
                   .length = 0
                   .argvec = 0
                   .units  = []
```

Now the actual domain of the signal structure is no longer `Time` but `Frequency`, therefore the size of the frequency domain is set to the number of frequency bins and the size of the time domain is set to one.

2.2 Plugin structure

A plugin minimally consists of the following two MATLAB files:

1. Initialisation function *plugInName_init.m*
2. Main (processing) function *plugInName_main.m*

2.2.1 Initialization

All variables needed for the processing step are initialized or calculated in the initialization function *plugInName_init.m*. The function is called from the framework's initialization function with the following syntax

```
[Signal, Plugin, GlobalSettings] =plugInName_init(Signal, Plugin, GlobalSettings);
```

Whereas

- `Signal` is the four dimensional Signal structure (see 2.1.2)
- `Plugin` is the plugin-number-specific substructure `Plugin(k)` (see 2.1.1)

General information about the current data, e.g. the sampling rate `fs` or the block size `blockSize` is stored in the settings substructure `GlobalSettings` of `ConfigStruct`.

2.2.2 Block size

Each plugin of the PlugInChain can theoretically run on a different block size. The block size must be defined in the initialization function. If a plugin requires a specific block size, the variable `Settings.blockSize` should be initialized accordingly. Otherwise, the global block size should be used as a default value

```
Settings.blockSize = GlobalSettings.blockSize;
```

Frame based processing is deactivated by using a block size of zero. Accordingly, any plugin that cannot handle frame-based processing, has to have a `Settings.blockSize = 0` statement. The necessary framing and buffering of the data is managed by the PlugInChain framework itself. If the data length is not compatible with one appearing block size, there remains some unprocessed part of the signal in the storage and the output will be shorter than the input. Sometimes it may therefore be helpful to work with zero padding if block sizes and the whole signal size do not fit. In this case the default value

```
GlobalSettings.zeroPadding = false;
```

has to be changed to `'true'` by defining it in the plugin list. With activated zero padding, the processing framework adds zeros to the input signal. The length of this is determined by the block size of the first plugin following the 'readData'-plugin.

2.2.3 _init function

To control the whole parameter set of a plugin without changing the source code in the corresponding plugin, the configuration file ('pluginListName.cfg') can be used to overwrite the specific values for each plugin. Fig.2.1. shows a source code snippet of the _init file. The default settings of the plugin are defined in line 4 - 7. Afterwards, the parameter from the configuration file is loaded (line 11 - 12). The variable `Settings.dBFS2SPL` may be changed in *.cfg-file, therefore `Settings.scale_fac` needs to be calculated (or recalculated if set) in line 17-18. It should be noted that only variables which are defined in the _init function are considered to be valid user parameters. Only these can be overwritten with values from the configuration file. If one wants to use parameters of the `GlobalSettings` in the _init function they should be defined as default values.

```
1: % -----------------------------------------------------------------------
2: % default settings
3: % -----------------------------------------------------------------------
4: Settings.blockSize = 0;
5: Settings.order = 512;
6: Settings.dBFS2SPL = 96;
7: Settings.fs = 44100;
8: % -----------------------------------------------------------------------
9: % reading the user parameters set in the configuration file
10: % ----------------------------------------------------------------------
11: Settings = applyUserSettings(Settings, Plugin.userSettings);
12: % ----------------------------------------------------------------------
13: if isfield(GlobalSettings,'dBFS2SPL')
14:    if GlobalSettings.dBFS2SPL ~= Settings.dBFS2SPL
```

```
15:         warning(['dBFS2SPL of Master is not equivalent to PlugIn!'])
16:         scale = GlobalSettings.dBFS2SPL - Settings.dBFS2SPL;
17:         Settings.scale_fac = 10^(scale/20);
18:         GlobalSettings.dBFS2SPL = Settings.dBFS2SPL;
19:     end
20: end
```

Figure 2.1: Part of the initialization file.

Furthermore, routines which check consistency between plugins or necessary conditions for the signal processing by the main file may also be located here.
If the plugin changes any signal dimension or the scaling of the signal, the initialization file has to transfer this information back to the PlugInChain by setting the new signal dimension or the new scaling in to the `GlobalSettings` or the `Signal` structure. For scaling the signal, the value of dB SPL is taken to account. If the scaling is affected by the plugin, it has to change the parameter of

```
GlobalSettings.dBFS2SPL
```

in the way, that this parameter at any time gives the value in dB SPL that equates to a signal amplitude of 1 in MATLAB.

2.2.4 _main function

The *_main* function of the plugin is called from the block processing part of the PlugInChain (*pluginchain_process.m*). The plugin's *_main* function does the actual processing of the input data. Therefore, it gets as input parameter the collected data, the `Plugin` structure which should contain all necessary variables from the main function and the `GlobalSettings`.
Usually, all signal processing is placed in the *_main* file, however, it is also possible to call further functions from here.
Routines which check consistency and monitor the proper functioning of the plugin should be located in the initialization file, so that a possible error occurs before calculation starts. But if there exists special dependencies of the actual processed numerical values (e.g. only positive or real numbers can be handled), further check routines can be located in the *_main* file.

2.3 Plugin lists

The plugin list is a text (ASCII) file which can be used to define a processing chain of plugins to yield more complex algorithms. Any number of available plugins can be concatenated whereby the listed order determines the signal flow. Note that the input signal is transformed from a plugin and passed to the next one. Thus, the signal domain of the output of a plugin must be consistent with the required input signal domain of the following plugin.
The first Line of every plugin list defines the `GlobalSettings`.

The plugin list 'pluginlist.cfg´ shown in Fig. 2.3., contains the following code:

```
1:  GlobalSettings
2:  (
3:   blockSize = 2000;
4:   silentMode = 0;
5:  )
6:  readData
7:  (
8:  )
9:  plotData
10: olatime2freq
11: (
12:  blockSize=1500;
13: )
```

Figure 2.3: File of the plugin list pluginlist.cfg.

It’s possible also to use plugins multiple times, e.g. the plugin ‘plotData’. Each plugin can have individual settings.

2.3.1 Use existing plugin lists

Existing plugin lists can be easily loaded by using either the GUI or a command line instruction (see section 1.3 and 1.4).

2.3.2 Create new plugin lists

New plugin lists can be easily derived from existing ones or can be written from scratch. It is possible to simply copy, rename and edit an existing plugin list with a conventional text editor. Alternatively, the function *makelist.m* (located in the working directory ‘pluginchain’) can be called from the MATLAB command window. The *makelist* GUI will appear (depicted in Fig. 2.4.). The function scans the PlugInChain folder ‘plugins” for all available plugins and lists them on the left side. The marked plugin can be either added or removed from the active plugin list by pressing the buttons add and remove, respectively. The name of the plugin list can be edited on the upper right side. By pressing the button create the composed plugin list can be stored to hard disc. If the desired plugin list name is already in use, the file can be overwritten by using the button overwrite.

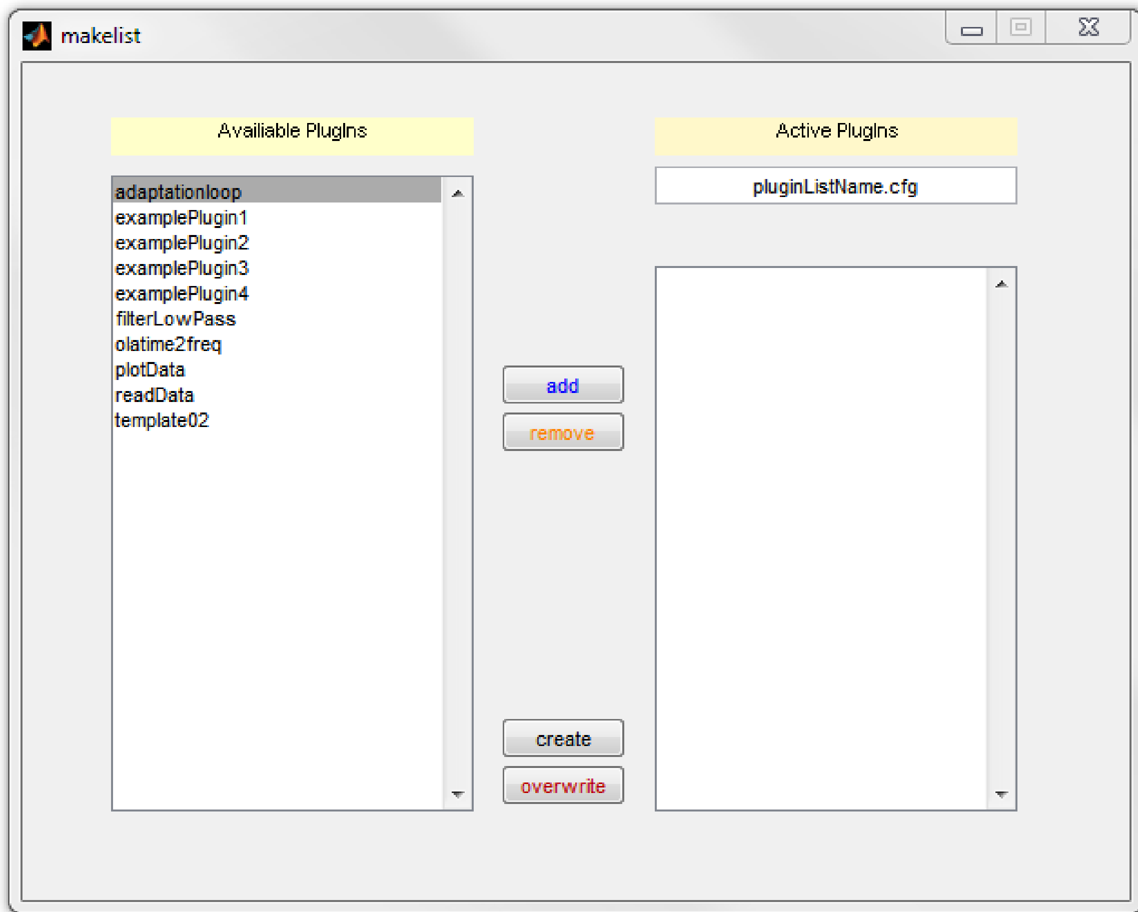


Figure 2.4: The *makelist* GUI to automatically generate new plugin lists.

Note that the current *makelist* function does not supply any consistency check about the specific plugin requirements in terms of the signal domain (time domain, frequency domain ...).

2.3.3 Assessing the history of loaded plugin lists

If the PlugInChain is started without a specified (and valid) plugin list as argument, it attempts to open the last loaded plugin list. If this is no longer available, a warning will be displayed stating that the last list was not found and that the default plugin list will be loaded. From the PlugInChain GUI (as shown in Fig. 2.5) the history of all used plugin lists can be accessed by pressing simultaneous the shift button on the keyboard and the right mouse button while pointing with the mouse on the plugin list display (*Shift + right mouse*). In this manner a plugin list also can be directly loaded by selecting it from the history list.

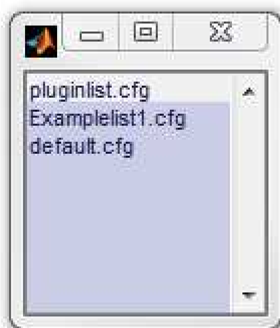


Figure 2.5: PlugInChain history of the past used plugin lists.

3 Plugins

This section gives an overview about how to work with and how to edit plugins.

3.1 Use plugins

As shown in Section 2.2 all available plugins can be used to form complex algorithms. During the initialization phase all relevant parameters are calculated for each plugin based on the default settings. All user parameter can be modified in the plugin list as described briefly in 2.2. The main advantage of configuring the plugins in the plugin list is that plugins can be used with different parameter sets without changing the source code of the corresponding plugin. This also allows the configuration of proprietary plugins that might only be available in machine readable code.

3.2 Change plugins

The user is able to add new functionality to the PlugInChain by changing and extending existing plugins. This can be especially worthwhile if available routines already provide similar tasks. In order to do so, both plugin files namely the initialization function and the main function of an existing plugin can be duplicated using the new plugin name. Afterwards, the initialization and the processing routine of the new plugin can be modified.

3.3 Write new plugins

Besides changing existing plugins, it is also possible to create a new plugin by using the template function *template2function.m*. It is located in the folder 'pluginchain_tools" and creates templates for a new plugin. This function sets up the source code for the two required plugin functions (*_init* and *_main*) with identity processing.

```
template2function('template01','newPlugInName');
```

The first input argument specifies the used template. The second input parameter determines the name of the new plugin. These new files are stored in the directory 'plugins'. If the template function is called with any third input argument, both new functions are opened in the MATLAB editor. It is also possible to use any existing plugin as template, but the user is in this case strongly advised to check if everything is correctly exported and that no variable is named like the plugin itself.
In the following, examples are given how to create new plugins by using the template function. Two plugins will be presented for different signal representations. There are two different template functions to be used in the shown manner for the different signal representations – `template01` (time domain) and `template02` (frequency domain). But they only differ in the check routines for the used signal domain.

If your plugin starts any external process that has to be stopped, your own function to stop this process, e.g. 'mystopfunc' should be defined within the file *'mypluginname_init.m'* and following setting should be defined:

```
Settings.ExitFunction =@mystopfunc;
```

The PlugInChain will call the exit function when signal processing ends or if the PlugInChain is aborted. An optional boolean flag exists for defining different behavior for usual process end and aborted processing. If aborted or stopped by occurrence of errors, the PlugInChain performs the function with optional parameter `‘true’` (abort-flag = true) and optional parameter `‘false’` otherwise (abort-flag = false). The PlugInChain checks your function for number of input arguments to decide if the flag is needed.
If your function executes different code for cases of aborted and proper end of process (or e.g. only if process aborted) you just write your ‘*mystopfunc*’ with two input parameters, whereat the second parameter is the aborted-flag. This looks like

```
function plugin =  mystopfunc (plugin, abort)
…..
      if abort
       ….
      else
       ….
      end
end
```

or if no difference for the type of processing end is needed:

```
function plugin =  mystopfunc (plugin)
…..
end
```

3.3.1 Example 1 - IIR low-pass filter

As a first example, this section will explain how to use the template function to generate an infinite impulse response (IIR) low-pass filter in the time domain. By calling

```
template2function('template01','filterLowPass',1)
```

the two files

- *filterLowPass_main.m*
- *filterLowPass_init.m*

are generated and directly opened in the MATLAB editor.

1. _init function

The actual filter design is done in the initialization function *filterLowPass_init.m*. Note that all variables which are needed for the processing step must be stored in the structure `Settings` to be available in the function *filterLowPass_main.m*.

```
1: function [Signal,Plugin,GlobalSettings]=filterLowPass_init(Signal,Plugin,GlobalSettings)
2: %FILTERLOWPASS_INIT initialization function for matlab pluginchain
3: % usage:
4: % function [Signal, Plugin, GlobalSettings] = filterLowPass_init(Signal, Plugin, GlobalSettings)
[…]
13: %
14:  Plugin.version = '1.0';
15: Plugin.delay = 0;
16: Settings.dependencies = {''}; % please enter the file dependencies manually
17: % -----------------------------------------------------------------------
18: % default settings
19: % -----------------------------------------------------------------------
20: Settings.filterOrder = 4; % filter order
21: Settings.cutoffFreq = 1000; % cutoff frequency
22: % -----------------------------------------------------------------------
23: % reading user parameter (do not edit)
24: % (user settings read from plugin list are stored in Plugin.userSettings)
25: % -----------------------------------------------------------------------
26: Settings = applyUserSettings(Settings, Plugin.userSettings);
27: % -----------------------------------------------------------------------
28: % internal settings and parameters that need to be recalculated
29: % -----------------------------------------------------------------------
30: fs = GlobalSettings.fs; % Get sampling frequency
31: chns = GlobalSettings.channels; % Get the number of channels
32: wn = Settings.cutoffFreq/fs*2; % Normalized cutoff frequency
33: % Create low-pass filter coefficients
34: [Settings.b,Settings.a] = butter(Settings.filterOrder,wn,'low');
35: % Initialize filter states
36: Settings.states = zeros(Settings.filterOrder,chns);
37:
38: % -----------------------------------------------------------------------
39: % write Settings to Plugin struct
40: % -----------------------------------------------------------------------
41: Plugin.Settings = Settings;
```
Figure 3.1: Initialisation function of the IIR low-pass filter.

The filter order `Settings.filterOrder` and the cutoff frequency of the filter `Settings.cutoffFreq` should be controllable by the user and their default values are assigned. All the parameters which are dependent on the user parameters (like the normalized cutoff frequency `wn` of the filter) are calculated after reading and applying the `Plugin.userSettings`. The filter design is done in line 34 of Fig. 3.1 by using the MATLAB function '`butter`'. After creating the filter coefficients, the memory for the inner filter states are allocated in line 36. This is required for frame-based processing which is allowed by the plugin as no specific `blockSize` is defined in the settings. In order to guarantee a proper calculation of the filter characteristics independent of the actual sampling rate of the processed file, the sampling rate is taken from the `GlobalSettings.fs`, which will be set by the signal-reading plugin to the actual value (line 30). This value corresponds to one over the sampling period which is stored in the first cell array element of the `Signal.Dimension(1).argvec.` All parameters which are required for the processing step are stored into the plugin settings structure `Plugin.Settings` for later usage.

2. _main function

The actual filtering is performed in the processing function *filterLowPass_main.m* which is shown in Fig. 3.2. Here the MATLAB function '`filter`' is used to filter the input signal data. The calculated filter coefficients from the initialization file which are stored in the settings structure of the plugin are passed by the framework of the PlugInChain to the processing function with the second input argument `Plugin`. Sometimes it might be convenient to copy

the input parameters into new variables to shorten the names, this was done here by storing `Plugin.Settings` in the new variable `Set`.
The main function is called for every chunk of data delivered as long as the complete input signal is processed. To ensure correct operation of the IIR filter across frames, the inner filter states need to be stored for the next frame.

```
1: function [Signal,Plugin,GlobalSettings]=filterLowPass_main(Signal,Plugin,GlobalSettings)
2: %FILTERLOWPASS_MAIN processing function for matlab pluginchain
3: % usage:
4: % function[Signal,Plugin,GlobalSettings]=filterLowPass_main(Signal,Plugin,GlobalSettings)
[…]
13: % insert your function here:
14: Set = Plugin.Settings; %shorten the names
15: in=Signal.data;
16: [out,Set.states] = filter(Set.b,Set.a,in,Set.states);
17: Signal.data=out;
18: Plugin.Settings=Set;
19: return
```

Figure 3.2: Main function of the IIR low-pass filter.

3.3.2 Example 2 - Adaptation loops

An example of a plugin which is working on a multidimensional signal is the adaptation loop plugin. This plugin is part of a computational model of the auditory periphery (Dau et al 1996, 1997) and mimics the temporal processing of the auditory system. Therefore, it is usually used after a 'gammatone' filterbank and a 'haircell' model.

1. **_init function**

First the default settings concerning the variables `blockSize`, `tau`, `fs`, `nStages`, `minimum` and `limit` are initialized (line 21-27 in Fig. 3.3). Afterwards, the user-dependent settings (defined in the plugin list) are loaded from `Plugin.userSettings` (line 29). Note that only settings that are defined before loading the user settings are overwritten. The adaptation loops require a specific scaling of the input signal and the value for dBFS2SPL is therefore set in line 34. If the signal was scaled differently before, rescaling is necessary (line 35-42). All internal parameters which are based on the user defined parameters are then calculated in lines 48-69 and are stored in the `Settings` structure for later access (line 73).
Finally, a check routine is provided to only enable the further processing stage if the signal header contains the correct signal domain. This is useful in order to easily detect the origin of errors during the processing. The required signal domain is for the adaptation loop plugin is `'Time'` (line 82-85). If this plugin would e.g. be used after a Fourier transformation, the plugin would set the error flag `errOccured` to `'true'` (line 83). If any error occurred during the initialization phase, the PlugInChain would stop and report the corresponding error messages.

```
1: function [Signal,Plugin,GlobalSettings]=adaptationloop_init(Signal,Plugin,GlobalSettings)
2: %ADAPTATIONLOOP_INIT initialisation function for matlab pluginchain
3: % usage:
4: % function [Signal,Plugin,GlobalSettings]=adaptationloop_init(Signal,Plugin,GlobalSettings)
[…]
13: %
15: Plugin.version = '1.1';
```

```
16: Plugin.delay = 0;
17: Settings.dependencies = {''}; % please enter the file dependencies manually
18: % -------------------------------------------------------------------------
19: % default value is taken from GlobalSettings
20: % -------------------------------------------------------------------------
21: Settings.blockSize  = GlobalSettings.blockSize;
22: Settings.tau        = [0.005; 0.05; 0.129; 0.253; 0.5];
23:                % time constants for each loop
24: Settings.fs = GlobalSettings.fs;
25: Settings.nStages = length(Settings.tau);
26: Settings.minimum = 1e-5;
27: Settings.limit = 0; % SK: taken from PEMO-Version
28: % reading user parameter (do not edit)
29: Settings = applyUserSettings(Settings, Plugin.userSettings);
30: % check level
31: % -------------------------------------------------------------------------
32: % if level of signal is not equivalent to HRTF, signal is set to HRTF for
33: % processing and afterwards rescaled.
34: Settings.dBFS2SPL  = 100; % This scaling is needed!
35: if isfield(GlobalSettings,'dBFS2SPL')
36:    if GlobalSettings.dBFS2SPL ~= Settings.dBFS2SPL
37:       warning(['dBFS2SPL of Master is not equivalent to PlugIn!'])
38:        scale = GlobalSettings.dBFS2SPL - Settings.dBFS2SPL;
39:        Settings.scale_fac = 10^(scale/20);
40:        GlobalSettings.dBFS2SPL = Settings.dBFS2SPL;
41:    end
42: end
43: % -------------------------------------------------------------------------
44: % internal settings and parameters that need to be recalculated
45: % -------------------------------------------------------------------------
48: ta = 1/GlobalSettings.fs;
49: chns = GlobalSettings.channels;
50: freq = Signal.Dimension(3).length;
51: modfreq = Signal.Dimension(4).length;
52: % Loop over the number of stages
53: for iStage = 1:Settings.nStages
54:     Settings.a1(iStage) = exp(-ta/Settings.tau(iStage));
55:     Settings.b0(iStage) = 1-Settings.a1(iStage);
56:     Settings.divisor(iStage,1:chns,1:freq) = Settings.minimum^(2^-iStage);
57:      if Settings.limit>1 % limitation enabled: calc values for exp fcn [SK]
58:         maxvalue = (1 - Settings.divisor(iStage,1,1)^2) * Settings.limit - 1;
59:         Settings.expfcn.fac(iStage) = maxvalue * 2;
60:         Settings.expfcn.expfac(iStage) = -2/maxvalue;
61:         Settings.expfcn.offset(iStage) = maxvalue - 1;
62:         clear maxvalue
63:      end
64: end
65: if size(Settings.divisor,4) == 1
66:    Settings.divisor = repmat(Settings.divisor,[1,1,1,modfreq]);
67: end
68: Settings.correction = Settings.divisor(Settings.nStages);
69: Settings.factor     = 100/(1-Settings.correction);
70: % -------------------------------------------------------------------------
71: % write Settings to Plugin struct
72: % -------------------------------------------------------------------------
73: Plugin.Settings = Settings;
74: % -------------------------------------------------------------------------
75: % check values
76: % -------------------------------------------------------------------------
77: errOccurred = false;
78: errMessage  = '';
79: % -------------------------------------------------------------------------
80: % provide check-routines here:
81: % -------------------------------------------------------------------------
82: if ~strcmp(Signal.Dimension(1).name,'Time')
83:    errOccurred = true;
```

```
84:    errMessage  = 'no time-signal detected';
85: end
86: % ----------------------------------------------------------------------
87: if errOccurred
88: %disp(repmat('-',1,80));disp(errMessage);disp(repmat('-',1,80));
89:    GlobalSettings.ok = false;
90:    GlobalSettings.errMessage = char(strcat(Plugin.name, ' : ', sprintf('%s\n'),...
91:                                   errMessage));
92:    disp(repmat('-',1,80));
93:    disp(char(strcat(Plugin.name,':',' ')));
94:    disp(errMessage);
95:    disp(repmat('_',1,80));
96: end
97: % ----------------------------------------------------------------------
98: return % go back to main function
99: % ----------------------------------------------------------------------
```

Figure 3.3: Initialization function of the adaptation loops.

2. _main function

The adaptation loops operate on the time signal (first dimension of `inData`) for any size of the other dimensions. Therefore, the size of the signal is determined in line 23. In the following, two cases are differentiated for shorten the calculation time. If the value for `Settings.limit` is <=1 an array operation is possible (line 30-35) with only two for-loops (line 29-38, line 31-36 respectively) running over the number of samples (T) and time constants (iStage) control the processing of the current data point (line 30). For limitation (`Settings.limit` > 1) a slightly different calculation (line 46-54) is needed whereas the loops in principle remain the same (line 43-58). The modified data is stored in the output variable `out` in line 59.
Although the plugin is designed to deal with multidimensional data, it can of course also be used outside the context of the PEMO model and be applied on a pure time domain signal.

```
1: function [outData,Plugin,GlobalSettings]=adaptationloop_main(inData,Plugin,GlobalSettings)
2: %ADAPTATIONLOOP_MAIN processing function for matlab pluginchain
3: % usage:
4: % function [outData,Plugin,GlobalSettings]=adaptationloop_main(inData,Plugin,GlobalSettings)
[…]
12: % some new variables to shorten names in calculation
13: outData=inData;
14: in=inData.data;
15: Set=Plugin.Settings;
16: % level correction forward (if required)
17: % ---------------------------------------------------------------------
18: if isfield(Set,'scale_fac')
19:    in = in .* Set.scale_fac;
20: end
21: % insert your function here:
22: out = in;
23: [T,C,F,M] = size(out);
24: out = max(out, Set.minimum);
25: if Set.limit<=1 % no limitation
26:    Midx = 1:M;
27:    Cidx = 1:C;
28:    Fidx = 1:F;
29:    for t=1:T
30:        tmp = out(t,Cidx,Fidx,Midx);
31:       for iStage = 1:Set.nStages
32:           tmp=tmp./Set.divisor(iStage,Cidx,Fidx,Midx);
33:           Set.divisor(iStage,Cidx,Fidx,Midx) =
34:                     Set.a1(iStage)*Set.divisor(iStage,Cidx,Fidx,Midx)...
35:                     +Set.b0(iStage)*tmp;
```

```
36:         end
37:         out(t,Cidx,Fidx,Midx) = (tmp-Set.correction)*Set.factor;
38:     end
39: else            % limitation enabled
40:     Midx = 1:M;
41:     Cidx = 1:C;
42:     Fidx = 1:F;
43:     for t=1:T
44:         tmp = out(t,Cidx,Fidx,Midx);
45:         for iStage = 1:Set.nStages
46:              tmp=tmp./Set.divisor(iStage,Cidx,Fidx,Midx);
47:              tmpGreater1 = (tmp > 1);
48:              tmpNotGreater1 = (tmp <= 1);
49:              tmp = tmp.*tmpNotGreater1 + tmpGreater1.*( Set.expfcn.fac(iStage)/...
50:                                 (1+exp(Set.expfcn.expfac(iStage)*(tmp-1)))...
51:                                 -Set.expfcn.offset(iStage) );
52:              Set.divisor(iStage,Cidx,Fidx,Midx) = ...
53:                               Set.a1(iStage)*Set.divisor(iStage,Cidx,Fidx,Midx)...
54:                               +Set.b0(iStage)*tmp;
55:         end
56:         out(t,Cidx,Fidx,Midx) = (tmp-Set.correction)*Set.factor;
57:     end
58: end
59: outData.data=out;
60: Plugin.Settings=Set;
61: return
```

Figure 3.4: Processing function of the adaptation loops.

3.3.3 Useful features

Often it is desirable to permute and use other functions to handle the multidimensional signal structure and being still able to use the conventional MATLAB functions in the common way. Due to the convention of the signal representation within the PlugInChain framework this might require rearrangements of the multidimensional signal array. The following MATLAB functions are quite useful to deal with the multidimensional signal array:

- `squeeze`
- `shiftdim`
- `permute`
- `repmat`
- `squeeze`

Please refer to the help functions of these MATLAB commands.

3.4 PlugInChain rules

In order to maintain the modularity and proper functionality of the PlugInChain, there are some rules the user should stick to.

- **Follow naming convention**
  Use an initial capital letter for MATLAB structures and a lower-case character for conventional variables and - although not conformed to by every author yet - plugin names. Persistent variables should be declared by using capital letters only. Names for Plugins should contain only alphanumerical characters.

- **Supply plugin check routines**.
  Every plugin should check, e.g., if the transferred signal dimension or signal domain matches the plugin requirements and provide corresponding error messages.

- **Support modularity**
  If the plugin changes any signal dimension, the _init should modify the signal descriptor appropriately to ensure proper initialization of the following plugins.

3.5 GlobalSettings of the pluginchain

The `GlobalSettings` contain besides general parameters like `blocksize` and path information some special variables that are used as Flags. Some of them are required for the workflow of the PlugInChain while others are user parameters to control performance.

3.5.1 Auditory Profile

Sometimes it is useful to work with an auditory profile. In this case you use the data from `GlobalSettings.Audioprofile`. There exists a 'full" version where all data for a client is loaded from a database (if existing) and a monaural version with variables

- `Audiogram` (Array contains Frequency and the corresponding hearing loss) and
- `Cat_Loudness` (Array contains Frequency and corresponding categorical loudness value).

To load these auditory profiles from a local existing MATLAB database, the value of `Global.Settings.AudioProfileFlag` has to be set to 1 and the Client identity and the type of audiogram has to be defined with

- `GlobalSettings.ClientId = 'ClientNumber' (f.e. 'TT123456')`

and

- `GlobalSettings.sAudID = 'slope'` (or other in data base defined values).

The monaural version is also available without using the database and the corresponding default values can be changed via the plugin list.
For working with auditory profiles it might be useful to use the GUI controlled tool for editing these profiles. It is available in the folder 'pluginchain_tools\audiogramtools\matlab' and started with the command

```
Clientbd();.
```

3.5.2 Performance parameters

The frame-based (block) processing is performed by the PlugInChain framework and might require some computational overhead. If the processing of individual plugins is not forcing an individual block size, it is generally recommended to set the `blockSize` in the plugin on the `GlobalSettings.blockSize`. Processing time can be considerably shortened if all plugins use the same block size. To process the whole data at once without using block processing, one should define:

```
GlobalSettings.blockSize = 0;
```

In this case, the first plugin 'readData' (this or an equivalent other plugin is required in first place to read the signal data) changes the value in this case to the size of specified input signal. The default value for the `GlobalSettings.blockSize` is set to zero. If you want to use frame-based processing with the same block sizes for all used plugins, it is useful to change this parameter in the plugin list to the required value (e.g. `GlobalSettings.blockSize=1024`;) and then define for all plugins

Settings.blockSize = `GlobalSettings.blockSize`.

In every other case of frame-based processing it is advised to use the default value.

The sampling rate and the number of channels also can be defined in the `GlobalSettings`:

```
GlobalSettings.fs = 0;
```

```
GlobalSettings.channels = 0;
```

If they (or one of them) remain on default value '0', it is set in first step by the plugin 'readData' to the sampling rate or number of channels of the signal.

If any plugin is changing the actual value it is required to change `GlobalSettings` appropriately in its _init file. If any value for sampling rate or channels is defined for the `GlobalSettings` in the plugin list, it has to agree with the used incoming signal data. Otherwise it will produce an error and processing is stopped.

WARNING:

If you write and use your own plugin to read any data instead of 'readData', your plugin has to do more than just reading data. It has to supply the PluginChain with the functionality of initializing the signal structure and setting several values in the `GlobalSettings`.

If Simulations should be performed with the PlugInChain, there exists the possibility of storing the data automatically. Therefore the variable (with default value 'false') must be set on true:

```
GlobalSettings.saveSimulation = true;.
```

There also exists the possibility to hide the progress bar while processing. To do this one has to define in the plugin list

```
GlobalSettings.showProgressBar = 0;.
```

Some plugins need to act on a specific signal level. Therefore `GlobalSettings.dBFS2SPL` defines the level of dB an amplitude of 1 will correspond to. Each plugin might have other scaling factors, but must in this case transform the signal and retransform afterwards or change the value of dBFS2SPL in the `GlobalSettings` correctly (see Example 2).

4 Quickstart

This section will give you a short description how to start working with the PlugInChain environment.
Start your MATLAB software. If you have installed all files, you should find the folder 'pluginchain'. Change your working directory in MATLAB to this folder. Type into command line

```
pluginchaingui('pluginlist','Examplelist1.cfg','gui','true')
```

Now the GUI will open with the loaded plugin list 'Examplelist1.cfg '. It should look like Fig. 4.1

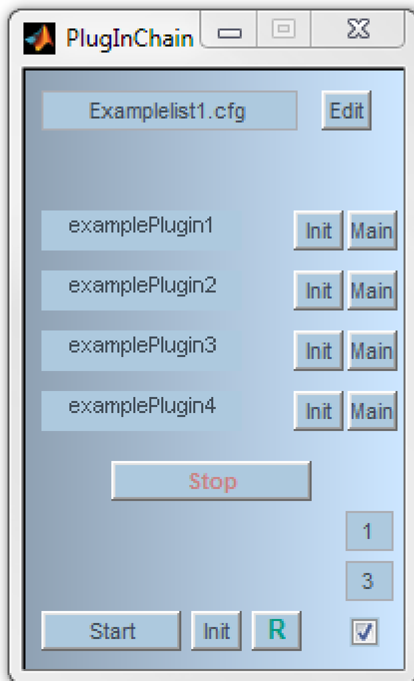


Figure 4.1: PlugInChain GUI for loaded plugin list ‘Examplelist1.cfg’.

The plugin list contains 4 plugins, named ‘examplePlugin1 -4’. The first plugin reads the signal data. The second plots it, the third is a low pass filter and the last plugin again plots the processed signal.
Start the processing by pressing the button Start in the left of the bottom of the GUI.
The plugin ‘examplePlugin2’ will open a window ‘Fig.2’ (because Fig.1 is the GUI) and display the signal and ‘examplePlugin4’ will also open a window ‘Fig.3’ and again display the actual signal data.

You now can open the plugin list ‘examplelist1.cfg’ by clicking the edit button on the right side of the displayed plugin list name in the top of the GUI. It will open the file in the MATLAB editor. It should look like shown in Fig. 4.2.
If you like you may explore its function by changing some things in the list to see what happens.

In line 9 the variable `‘targetname’` specifies the wave file from which the signal is been read. If you have any other wave file in the search path (it is recommended to store them in

the folder ‘waves’, especially because you can use files there without declaring an additional path) you may change the input file by changing the value of ‘`targetname`’.

If you want to try block processing, do just change `blockSize` of ‘examplePlugin3’ by now.

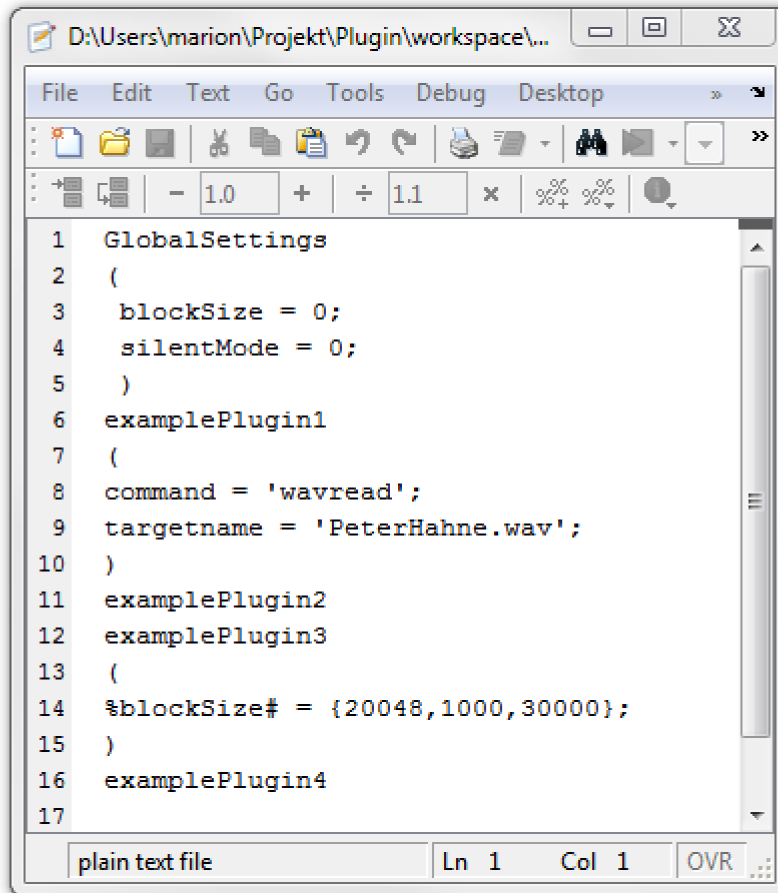


Figure 4.2: File of the plugin list ‘Examplelist1.cfg’

If you have changed anything in the list (e.g. delete examplePlugin2 from list to see what happens) you must store the file. The GUI will be refreshed when you press the button reset. Again, initialize all plugins by clicking on Init or start the processing by clicking on start. The button Init and Main in the right hand of each plugin name are only required for programming plugins.

The three buttons right sided of R in Fig. 4.1 are only required for batch processing and will not be discussed here.

The next step is now to change the loaded plugin list. You can do so by clicking with the ‘right mouse button’ onto the field where the name of the loaded plugin list is displayed (in top of the GUI). It will open up a file-open menu window where all *.cfg – files in the folder ‘pluginlists’ are displayed and can be chosen for opening. See also fig. 4.3.
At least one further plugin list ‘pluginlist.cfg’ will exist. Click on a file and then ‘open’ (depending on the language of your operating system) to load the list into the PlugInChain. The GUI will refresh and display now the plugin names of the new list. You now can use this list in the same way as in the example before.

The next step is to build a new list of your own. You surely can copy a list file and edit the content directly, but there is an easier way to do this, if you only want to use existing plugins. The function ‘makelist’ is a tool with graphical interface that gives you the possibility to choose from the existing plugins in the folder ‘plugins’. See Fig. 4.4 for an overview.

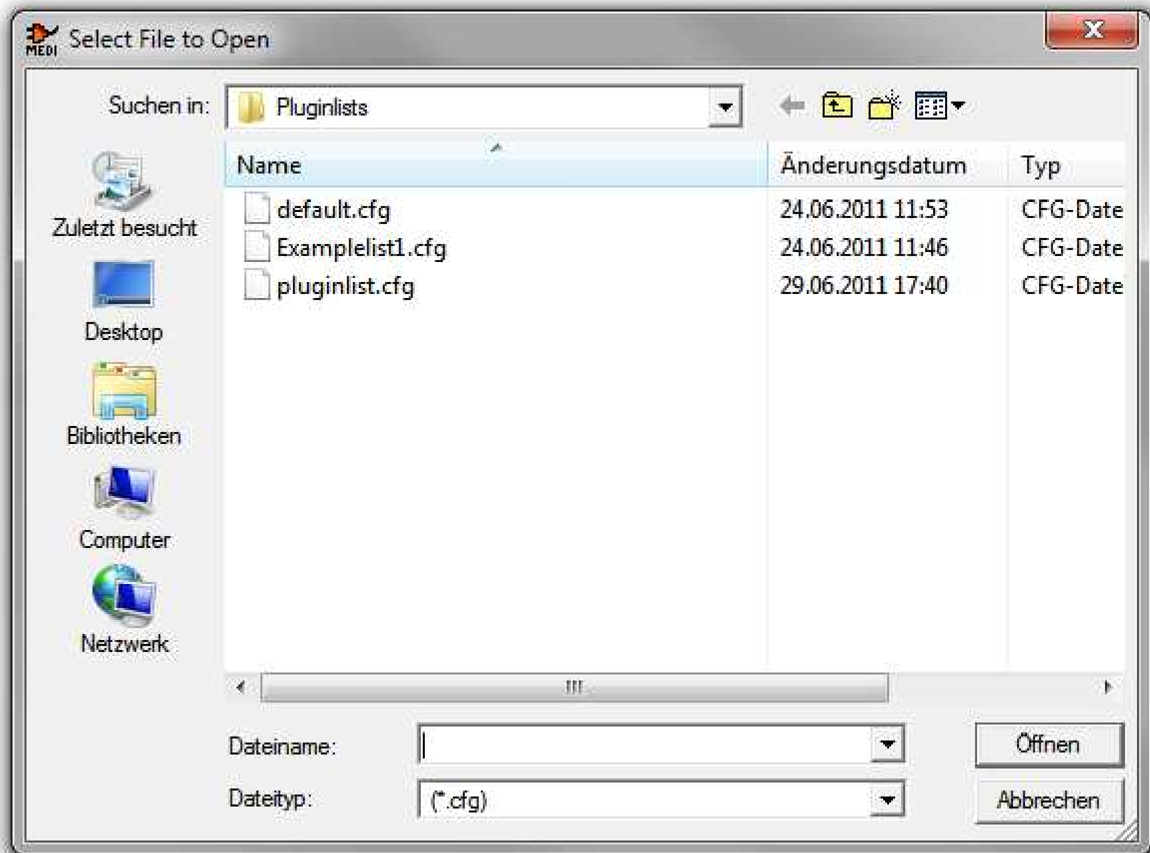


Fig. 4.3: File selection menu

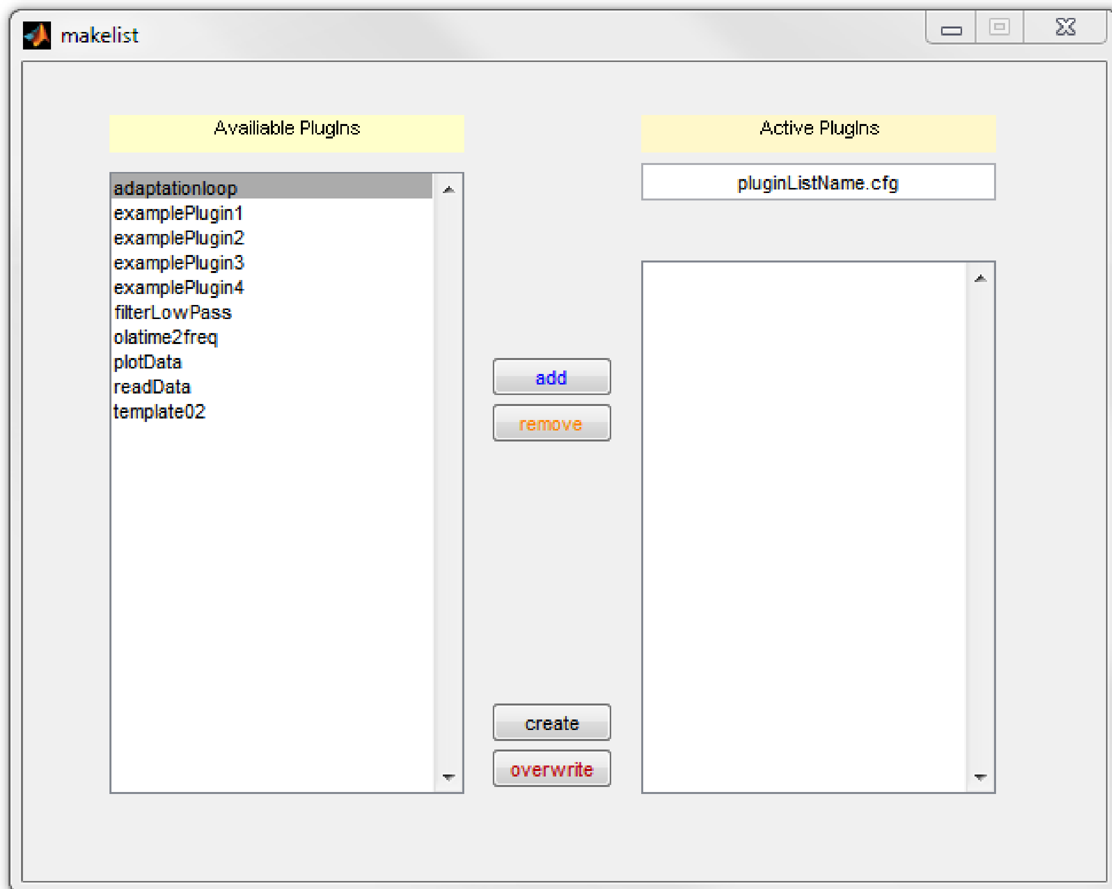


Fig.4.4: *makelist* GUI

Try with building up for example a new list where you use 'examplePlugin1'', 'examplePlugin3'', 'examplePlugin2'' and choose a name like 'Examplenew.cfg'. Load your new plugin list into the PluginChain in one of the above described ways and test it.

Available plugins

For the PlugInChain are different packages of plugins available. For an overview about the available plugins the user is referred to the documentation for these features in the folder 'docs'.