BTMHA

Boys Town Master Hearing-Aid Software Platform

Stephen Neely, D.Sc.
Director, Communication Engineering Laboratory

Boys Town National Research Hospital
4 March 2020

TABLE OF CONTENTS

Introduction

BTMHA provides a software platform for exploration of hearing-aid signal processing. The program performs three primary tasks:

(1) Sets up an input/output stream.
(2) Sets up a signal processor between the input and output.
(3) Starts the stream and maintains its operation until finished.

The inputs and outputs may be files or may be an audio device. The signal processing may be a single *plugin* that is precompiled in a separate file or may be a combination of multiple *plugins* connected by *mixers* into *chains* and *banks*.

The software design is intended to support three levels of users:

(1) Research Audiologists who want the ability to easily adjust signal-processing parameters.
(2) Application engineers who want to reconfigure existing signal-processing plugins without recompiling code.
(3) Signal-processing engineers who want to code new algorithms.

A modular design has been adopted to replacement of code with alternative implementations. The program is written entirely in C to reduce its system dependence. The installation "footprint" is kept small by restricting its software package dependence to three open-source libraries developed at BTNRH.

Installation

BTMHA is currently installed by compiling its source code. The dependence of BTMHA on other software packages is limited to three open-source libraries developed at BTNRH:

(1) ARSC – a library of functions for platform-independent access of audio devices.
(2) SIGPRO – a library of basic signal-processing functions.
(3) CHAPRO – a library of functions for simulating hearing-aid signal processing.

The source code for these libraries may be downloaded from the Boys Town GitHub repository at https://github.com/BoysTownorg/.

Source-code installation instructions

Although the source code compiles and the program runs under three different operating-systems (MacOS, Windows, and Linux), version 0.05 has only been fully tested under MacOS in a Terminal window.

Compilation in the MacOS Terminal window requires prior installation of Xcode command-line tools.

```
% xcode-select --install
```

In your preferred source-code folder, clone the three requisite library repositories.

```
% git clone https://github.com/BoysTownorg/arsc
% git clone https://github.com/BoysTownorg/sigpro
% git clone https://github.com/BoysTownorg/chapro
```

In each of the three subfolders (arsc, sigpro, chapro) build and install the library.

```
% cd <library name>
% cp makefile.mac Makefile
% make
% sudo make install
```

The final step is to install the BTMHA distribution. First, create a subfolder called btmha and download source code from audres.org. A convenient way to obtain the source code is to use wget.

```
% wget http://audres.org/downloads/btmhasc.zip
% cd btmha
% unzip ../btmhasc.zip
```

Alternatively, the zip file can be downloaded audres.org. After unzipping, build and compile the source code.

```
% cp makefile.mac Makefile
% make
% sudo make install
```

In addition to copying the btmha executable file to /usr/local/bin, the installation procedure also copies example plugins to /usr/local/lib.

Brief Tutorial

The BTMHA program has a command-line interface and accepts use input from tree sources: command-line arguments, *interactive* mode, and *list* files. The command-line option -h displays a list of options.

```
% btmha -h
usage: btmha [-options] [list_file] ...
options
-h    display usage info
-i    interactive mode
-q    quiet mode
-v    display version
```

The tutorial follows the convention that user-typed text appears in red. Besides options, command-line arguments may also specify parameter values, commands, or names of *list* files.

Interactive mode

A good way to discover what BTMHA can do is to use its interactive mode.

```
% btmha -i
```

```
==== Boys Town Master Hearing Aid ====
```

```
>>
```

The BTMHA command-line prompt is “>>”.

Any line that contains an equal’s sign is interpreted as an assignment of a value to a variable. The variable name can be any string of alphanumeric characters, but a few names are reserved for special purposes. The value may be an arithmetic expression and may contain variable names.

```
>> pi=3.1415927
pi = 3.1416
>> tpi=2*pi
tpi = 6.2832
```

Conversion of values between decibel values and scale factors is supported by log and power functions.

```
>> db=[0 3 6 9 12]
db = [0 3 6 9 12]
>> sf=10.^(db/20)
sf = [1 1.4125 1.9953 2.8184 3.9811]
>> db=20*log(sf)
db = [0 3 6 9 12]
```

Most other standard transcendental functions are also supported. Any entered line that does not contain an equal’s sign is interpreted as a command. For example, the *list* command displays the values of all currently defined variables.

```
>> list
list:
    pi                = 3.1416
    tpi               = 6.2832
    db                = [0 3 6 9 12]
    sf                = [1 1.4125 1.9953 2.8184 3.9811]
```

Appendix A contains a complete list of BTMHA commands.

List files

Any line entered at the btmha command line in interactive mode can be alternatively be place placed in a text file that is read by the program. The command for reading such files, which are called *list* files, is simply the name of the file. The default file name extension of list files, which need not be specified explicitly when list files are referenced, is *.lst*. List files are typically used to specify parameters that customize processing details. For example, the following lines, which are contained in a list file called *tst_gha.lst*, customize the processing performed by a plugin called *gha* that simulates the processing of a generic hearing aid.

```
# tst_gha.lst - test generic hearing aid example
srate        = 24000
chunk        = 32
process      = plugin
plugin       = gha
cross_freq   = [317.2 503.0 797.6 1265 2006 3181 5045]
comp_ratio   = [0.7 0.9 1 1.1 1.2 1.4 1.6 1.7]
input        = test/carrots.wav
output       = test/tst_gha.wav
datfile      = test/tst_gha.mat
```

The value assignments specified in this file include both *plugin* and *io* parameters. (For reference, complete lists of *plugin* and *io* parameters are provided in Appendices.) Processing is started by default at the end of a list file, so may be initiated by simply putting the name of the file as an argument when invoking btmha.

```
% btmha tst_gha
```

Alternatively, it is sometimes convenient to use a list file to specify the processing configuration and then enter interactive mode immediately after reading the file.

```
% btmha tst_gha -i
>> intersect
    input            = test/carrots.wav
    output           = test/gha.wav
    datfile          = test/gha.mat
    cross_freq       = [317.2 503 797.6 1265 2006 3181 5045]
    comp_ratio       = [0.7 0.9 1 1.1 1.2 1.4 1.6 1.7]
```

The *intersect* command displays the variables from the list file that have matching plugin-parameter names.

Processor Design Concepts

In general, the primary processor that is situated between the input and output streams is a composite combination of component processors. The component processors may be either computational or composite.

- Computational
  - Mixers
  - Plugins
- Composite
  - Chains
  - Banks

There are four types of component processors: mixers, plugins, chains, and banks.

Mixers

In the following example, setting mixer=1 and processor=mixer copies the input file to the output file.

```
% btmha -i
>> input=test/carrots.wav
input = test/carrots.wav
>> output=copy.wav
output = twochn.wav
>> process=mixer
process = mixer
>> mixer=1
mixer = 1
>> start
start...
  WAV input : test/carrots.wav repeat=1
  prepare_io: sr=24000 cs=32 ns=314988
 speed_ratio: (wave_time/wall_time) = (13.1/0.00106) = 1.24e+04
  WAV output: copy.wav
```

In a similar manner, a two-channel copy of the input file may be created by setting the io parameter noch=2 and specifying two outputs for the mixer.

```
>> output=twochn.wav
output = twochn.wav
>> noch=2
noch = 2
>> mixer=[1 1]
mixer = [1 1]
>> start
start...
  WAV input : test/carrots.wav repeat=1
  prepare_io: sr=24000 cs=32 ns=314988
 speed_ratio: (wave_time/wall_time) = (13.1/0.00155) = 8.45e+03
  WAV output: twochn.wav
```

The file test/carrots.wav contains a 13-second spoken passage. The speed_ratio reported at the end of processing indicates the speed of the processor relative to the natural rate of the audio file. Any speed ratio greater than one is sufficient for “real-time” processing.

Chains

A chain is a series combination of component processors. Component processors may any type of processor (i.e., mixer, plugin, chain, or bank). Each component processor must conform to the constraint that the number of its input channels must match the number of output channels of the preceding component processor. Likewise, the number of its output channels of the next processor. The number of input and output channels for the entire chain is the number of input channels of the first processor and number of output channels of the last processor.

Combining mixers into a chain is equivalent to matrix multiplication, as shown in the following example, which uses one mixer to split a single channel input into three channels with differing amplitudes. Then, a second mixer is used to recombine the three intermediate channels into a single channel with an amplitude equal to the original input.

```
% btmha -i
>> input=test/carrots.wav
input = test/carrots.wav
>> output=playback
output = playback
>> mixer1=[1 2 3]
mixer1 = [1 2 3]
>> mixer2=[1;1;1]/6
mixer2 = [1;1;1]/6
>> chain1=[mixer1 mixer2]
chain1 = [mixer1 mixer2]
>> process=chain1
process = chain1
>> start
start...
audio output: Built-in Output
  WAV input : test/carrots.wav repeat=1
  prepare_io: sr=24000 cs=3840 ns=314988
>> chain2=[mixer1 mixer2 chain1]
chain2 = [mixer1 mixer2 chain1]
>> process=chain2
process = chain2
>> start
start...
audio output: Built-in Output
  WAV input : test/carrots.wav repeat=1
  prepare_io: sr=24000 cs=3840 ns=314988>>
```

The final output of the chain, which should be identical to the original "carrots" passage, is delivered to the default audio device.

Banks

A bank is similar to a chain, expect that its component processors are connected to each other in a *parallel* arrangement instead of in series. Thus, there are no constraints on the number of input and output channels of the component processors. The number of input and output channels for the bank are the sums of the number component processor input and output channels respectively. The following example demonstrates a bank composed of two mixers that is contained within a chain.

```
% btmha -i
>> input=test/carrots.wav
input = test/carrots.wav
>> output=playback
output = playback
>> mixer1=1
mixer1 = 1
>> mixer2=[1 1]
mixer2 = [1 1]
>> mixer3=[1;1;1]/3
mixer3 = [1;1;1]/3
>> bank=[mixer1 mixer2]
bank = [mixer1 mixer2]
>> chain=[mixer2 bank mixer3]
chain = [mixer2 bank mixer3]
>> process=chain
process = chain
>> mixer
    mixer1           = [1]
    mixer2           = [1 1]
    mixer3           = [1;1;1]/3
>> bank
    bank             = [mixer1 mixer2]
>> chain
    chain            = [mixer2 bank mixer3]
>> start
start...
audio output: External Headphones
  WAV input : test/carrots.wav repeat=1
  prepare_io: sr=24000 cs=3840 ns=314988
```

Once again, the final output of the chain, which should be identical to the original “carrots” passage, is delivered to the default audio device.

Plugins

A plugin is a computational processor that is similar to a mixer in that it produces a set of output samples at each time step based on a set of input samples. A plugin differs from a mixer in having no requirement that the input-output transformation is either linear or time invariant. Plugins are implemented by writing C code that is compiled into a shared library that is dynamically loaded whenever the plugin is needed. Although plugins may, in general, have any number of input and output channels, our examples have only single-channel inputs and outputs for simplicity. A plugin parameter may be altered by assigning a new value in a list file to a variable with the same name.

The first plugin example simulates the signal processing of a generic hearing aid by combining several processing stages into a single plugin. These processing stages include feedback management, filter-bank frequency analysis, wide dynamic range compression, and frequency synthesis. The name of this plugin is “gha” and its executable file is “gha.so”. Commands for using gha as a plugin are contained the file “tst_gha.lst”.

```
# tst_gha.lst - test generic hearing aid example
srate       = 24000
chunk       = 32
process     = plugin
plugin      = gha
```

```
cross_freq  = [317.2 503.0 797.6 1265 2006 3181 5045]
comp_ratio  = [0.7 0.9 1 1.1 1.2 1.4 1.6 1.7]
afc_sqm     = 1
input       = test/carrots.wav
output      = test/tst_gha.wav
datfile     = test/tst_gha.mat
```

These commands may be executed by supplying the name of the list file as an argument when running the btmha program.

```
% btmha tst_gha
==== Boys Town Master Hearing Aid ====
start...
     process: plugin.gha
         gha: IIR+AGC+AFC
 speed_ratio: (wave_time/wall_time) = (13.1/0.164) = 80.2
  WAV output: test/tst_gha.wav
  MAT output: test/tst_gha.mat
final misalignment error = -15.90 dB
done...
```

In addition to an audio output file, this plugin also creates a file called *tst_gha.mat* to which will be written metrics for assessing the quality of adaptive feedback cancelation.

Appendix C contains a complete list of parameters for the *gha* plugin.

A second plugin example also simulates generic hearing aid processing, but splits the processing stages into two separate plugins that are combined dynamically by a chain processor. The filter-bank plugin is call iirfb and has two entry points, the first entry performs frequency analysis and the second entry performs frequency synthesis. The dynamic-range-compression (DRC) plugin is called agc and has three entry points. The first entry applies DRC to the input channel prior to frequency analysis, the second entry applies DRC to each frequency-band

```
# tst_ifsc.lst - test IIR+AGC
srate       = 24000
chunk       = 32
process     = chain
chain       = [plugin1.0 plugin0.0 plugin1.1 plugin0.1 plugin1.2]
plugin0     = iirfb
plugin1     = agc
input       = test/cat.wav
output      = test/tst_ifsc.wav
```

The commands in this list file are executed by the following line.

```
% btmha tst_ifsc
```

The third plugin example add feedback management to the previous example. The processing chain is bracketed by the input and output stages of the AFC plugin.

```
# tst_afc.lst - test IIR+AGC+AFC
srate       = 24000
chunk       = 32
process     = chain
chain       = [p2.0 p1.0 p0.0 p1.1 p0.1 p1.2 p2.1]
plugin0     = iirfb
plugin1     = agc
```

```
plugin2     = afc
cross_freq  = [317.2 503.0 797.6 1265 2006 3181 5045]
comp_ratio  = [0.7 0.9 1 1.1 1.2 1.4 1.6 1.7]
afc_sqm     = 1
input       = test/carrots.wav
output      = test/tst_afc.wav
datfile     = test/tst_afc.mat
```

Note in this example that, for convenience, processor names such as “plugin” may be abbreviated by their first letter (e.g., “p”) when specifying component processors. In the processor specification, the digit preceding the dot selects the plugin number and the digit after the dot selects the entry number within that plugin. This triple-plugin example is similar to the single-plugin example presented previously and produces similar output.

```
% btmha tst_ifsc
==== Boys Town Master Hearing Aid ====
start...
  WAV input : test/carrots.wav repeat=1
  prepare_io: sr=24000 cs=32 ns=314988
 speed_ratio: (wave_time/wall_time) = (13.1/0.162) = 81
  WAV output: test/tst_afc.wav
  MAT output: test/tst_afc.mat
final misalignment error = -15.91 dB
```

Appendix D has a list of the plugin examples included in the BTMHA distribution. These plugins are implanted as shared libraries and installed in /usr/local/lib. The section on Plugin Development below provides information about creating custom plugins.

Variable scope

Because it is possible to use the same plugin file in more than one plugin, some way is needed to limit the scope of variables, so that plugin parameters may be set to different values in each plugin. Limited scope is accomplished by preceding the variable name by the name of the plugin and a dot. For example, consider the following list file.

```
# twoplg.lst - two-plugin example
plugin1     = gha
plugin2     = gha
bank        = [plugin1 plugin2]
mixer       = [1 1]
chain       = [mixer bank]
process     = chain
noch        = 2
plugin1.dsl_ear=1
plugin2.dsl_ear=2
intersect
input=test/carrots.wav
output=twoplg.wav
```

The *intersect* command in this list file produces the following output.

```
plugin1.gha^list:
    dsl_ear          = 1
plugin2.gha^list:
    dsl_ear          = 2
```

Note that the plugin parameter *dsl_ear* has different values in the two plugins.

Plugin Processor Development

Plugin processors are written in C and compiled as shared libraries that are loaded dynamically, as needed, at run time. Every plugin processor must contain functions that are named *configure* and *prepare*.

```
    void configure(void *vlst, int *nv);
    void prepare(STA *state, void *vlst, MHA *mha, int entry);
```

The main purpose of *configure* is to return a list of settable parameters. The main purpose of *prepare* is to initialize the data used by the processor. The processor data and supporting information is called the processor "state" and is contained in structure that is manipulated by the btmha program. A special feature of the processor state is that all data is consolidated into a single contiguous block and accessed by means of a single list of data pointers. This approach facilitates the creation of composite processors by combining the states of component processors.

The implementation of the example plugin processors relies heavily on functions contained in the CHAPRO library that have been coded to conform to the data-state requirements of BTMHA plugins. However, there is no dependence of BTMHA on CHAPRO other than in the plugins and no inter-dependence of plugins with other plugins. So, plugin processors have been designed to be modular and example plugins may be replaced by custom plugins that contain novel signal-processing algorithms.

Input-Output Stream

The input-output stream connects a signal processor to either an audio device or an audio file. BTMHA current supports audio devices through the ARSC library and supports WAV format audio files. To facilitate replacement of ARSC by other audio device interfaces, all ARSC dependence is contained in a single file named stream.c. The following functions implement the input-output stream features of btmha.

```
int stream_prepare(I_O *io, MHA *mha, char *msg);
int stream_device(int io_dev);
void stream_start(I_O *, STA *);
void stream_init(MHA *mha);
void stream_show(char *line);
void stream_cleanup();
void stream_replace(MHA *mha, VAR *lvl);
void play_wave(I_O *, int);
void sound_check(double, int);
```

In the current version of btmha, replacement of the ARSC audio device interface would require writing alternative versions of these ten functions and recompiling btmha. A possible feature of some future version of btmha would be to compile these ten stream function as a shared library that would be loaded as needed. This approach could allow for support of alternate audio device interfaces without the need to recompile the entire program.

Appendix A. BTMHA commands

The following commands perform the indicated actions.

Command	*Action*
`bank`	Displays definitions of *bank* processors
`chain`	Displays definitions of *chain* processors
`clear`	Deletes all *list* variables
`device`	Displays available audio devices and indicates selected device
`help`	Displays list of BTMHA commands
`intersect`	Displays intersections of *list* variables with *plugin* variables
`stream`	Displays input/output variables
`list`	Displays all *list* variables
`mixer`	Displays definitions of *mixer* processors
`optimize`	Optimizes AFC parameters for example in *tst_gha.lst*
`play`	Streams file to audio device [default=input; @=output
`plugin`	Displays names of *plugin* processors
`prepare`	Initializes variables and state for input/output and primary processor
`quit`	Exits the program
`sndchk`	Outputs sound check
`start`	Initiates processing from input to output
`ver`	Reports version numbers of BTMHA software components

Appendix B. Input / Output Parameters

The *io* parameters customize the input and output stages. The *io* command displays the values of these parameters. When applicable, parameter units are provided in parentheses. Bracketed numbers are the default values.

Parameter	*Description*
`srate`	Sampling rate (Hz). [24000]
`chunk`	Number of samples processed at once. [32]
`device`	Device number used by ARSC functions.
`io_wait`	Streaming time-out interval inside the input/output loop (ms). [40]
`repeat`	Number of times an input file is repeated. [1]
`nich`	Number of input channels. [1]
`noch`	Number of output channels. [1]
`process`	Name of primary processor. [plugin]
`input`	Name of input file or audio device input. [capture]
`output`	Name of output file or audio device output. [playback]
`spl_ref`	RMS reference value for 0 dB SPL. [1.1219e-06]
`rms_lev`	RMS input may be rescaled to this level (dB SPL). [65]

Appendix C. GHA Plugin Parameters

The plugin called *gha* simulates "generic hearing-aid" processing. Its parameters are listed in the table below. The *pluginN* command displays the values of these parameters for plugin number N. When applicable, parameter units are provided in parentheses. Bracketed numbers are the default values. The default values may be altered by specifying the new values to a variable of the same name in a list file. One exception is *nchan*, which is a read-only parameter with a value that cannot be changed.

Parameter	*Description*
`nchan`	The numbers of input and output channels. [1 1]
`afc`	Adaptive-feedback-control toggle (0=no/1=yes). [1]
`fir`	Finite-impulse-response filter-bank toggle (0=no/1=yes). [0]
`afc_afl`	AFC adaptive filter length. [100]
`afc_sqm`	Save AFC quality metrics toggle (0=no/1=yes). [1]
`afc_rho`	AFC forgetting factor. [0.0014388]
`afc_eps`	AFC power threshold. [0.0010148]
`afc_mu`	AFC step size. [0.0001507]
`afc_fbg`	AFC feedback gain. [1]
`afc_hdel`	AFC hardware delay. [0]
`agc_attack`	AGC attack time (ms). [1]
`agc_release`	AGC release time (ms). [50]
`agc_maxdB`	AGC maximum level (dB SPL). [119]
`agc_tkgain`	AGC compression-start gain (dB). [0]
`agc_tk`	AGC compression-start knee-point (dB SPL). [105]
`agc_cr`	AGC compression ratio. [10]
`agc_bolt`	AGC broadband output limiting threshold (dB SPL). [105].
`cross_freq`	Crossover frequencies (Hz). [317.17 502.97 797.63 1264.9 2005.9 3181.1 5044.7]
`dsl_attack`	DSL attack time (ms). [5]
`dsl_release`	DSL release time (ms). [50]
`dsl_maxdB`	DSL maximum level (dB SPL). [119]
`dsl_tkgain`	DSL compression-start gain (dB). [-13.594 -16.591 -3.7978 6.6176 11.305 23.718 35.859 37.389]
`dsl_tk`	DSL compression-start knee-point (dB SPL). [32.2 26.5 26.7 26.7 29.8 33.6 34.3 32.7]
`dsl_cr`	DSL compression ratio. [0.7 0.9 1 1.1 1.2 1.4 1.6 1.7]
`dsl_bolt`	DSL broadband output limiting threshold (dB SPL). [78.767 88.2 90.7 92.833 98.2 103.3 101.9 99.8]
`dsl_nchan`	Number of channels [8]
`dsl_ear`	Ear (1=left/2=right). [0]
`input`	Input file (for reference only). []
`output`	Output file (for reference only). []
`datfile`	Data file (for reporting data). []

Appendix D. Plugin Examples

The plugin called *gha* simulates all stages of generic "generic hearing-aid" processing. The other plugin examples implement these stages separately and with some variations. In some cases, the number of input/output channels of the plugin depends on the number of bandpass filters in the filter-bank, which is denoted by *nfilt* in the table.

name	*entry points*	*channels in / out*	*Description*
`gha`	1	1 / 1	Generic hearing aid: AFC+FIRFB+AGC
`afc`	2	1 / 1	Adaptive feedback control
		1 / 1	
`firfb`	2	1 / *nfilt*	Finite-impulse-response filter-bank.
		nfilt / 1	
`iirfb`	2	1 / *nfilt*	Infinite-impulse-response filter-bank.
		nfilt / 1	
`cfirfb`	2	1 / 2·*nfilt*	Complex finite-impulse-response filter-bank.
		2·*nfilt* / 1	
`ciirfb`	2	1 / 2·*nfilt*	Complex finite-impulse-response filter-bank.
		2·*nfilt* / 1	
`agc`	3	1 / 1	Automatic gain control.
		nfilt / *nfilt*	
		1 / 1	
`icmp`	1	2·*nfilt* / 2·*nfilt*	Instantaneous compression.

Appendix E. List File Examples

These list files demonstrate several variations of hearing-aid signal-processing.

Name	*Description*
`tst_gha.lst`	Generic hearing aid (FIR+AGC+AFC) single plugin.
`tst_afc.lst`	Generic hearing aid (FIR+AGC+AFC) triple plugin.
`tst_ffio.lst`	FIR input/output demonstration.
`tst_iffio.lst`	complex FIR input/output demonstration.
`tst_cffio.lst`	complex FIR input/output demonstration.
`tst_cifio.lst`	complex IIR input/output demonstration.
`tst_ffsc.lst`	FIR+AGC input/output demonstration.
`tst_ifsc.lst`	IIR+AGC input/output demonstration.
`tst_cffsc.lst`	complex FIR+ICMP input/output demonstration.
`tst_cifsc.lst`	complex IIR+ICMP input/output demonstration.

Appendix F. Feedback-Management Parameter Optimization

To demonstrate another aspect of the flexibility of the processor-state format, the program includes a commend called “optimize” that performs optimization of three parameters of the adaptive feedback cancellation (AFC) stage of the generic hearing aid (GHA) example. Only two easy steps are required to initiate the optimization process: (1) initialize the GHA parameters and (2) invoke the “optimize” command.

```
% btmha tst_gha optimize
==== Boys Town Master Hearing Aid ====
    optimize: afc_rho afc_eps afc_mu
plugin0.gha^list:
    cross_freq       = [317.2 503 797.6 1265 2006 3181 5045]
    input            = test/carrots.wav
    output           = test/tst_gha.wav
    datfile          = test/tst_gha.mat
    process          = plugin
         gha: IIR+AGC+AFC
    # initial parameters
    afc_rho   = 0.0014388
    afc_eps   = 0.0010148
    afc_mu    = 0.0001507
......................................................................
...............................................
    # optimized parameters
    afc_rho   = 0.0014652
    afc_eps   = 0.0010149
    afc_mu    = 0.0001507
maximum error after 8 sec = -15.05 dB
 final misalignment error = -15.90 dB
done...
```

Optimization is achieved by minimization the misalignment-error of the estimated feedback impulse response over a specified period of time, which in this case extends from 8 seconds after the start until the end of the “carrots” passage.