avg_q manual

Dr. Bernd Feige mailto:bernd.feige@uniklinik-freiburg.de

Version 6.0.0

December 1, 2014

1 Introduction

The approach chosen to implement a data analysis tool was to write a single, user configurable main program (called **avg_q**) instead of many single utilities that would have to exchange large amounts of data via files. The main idea, as it presented itself originally, is contained in the name: **avg_q** stands for 'average-queue' which means that there is a processing queue applied sequentially to epochs of incoming data, and the processed epochs are averaged. The configurable processing queue consists of a sequence of 'methods' (algorithms), each of which works on the result of the previous. Averaging was only the first data reduction method the program featured; today, some more are available.

avg_q is configured by means of a script file containing the name of one method per line together with its options and arguments. After configuration, the methods are executed from top to bottom with an implicit loop over all epochs yielded by the first method. Comments may be placed anywhere after a hash ('#') sign, just as in Unix shell scripts. Empty lines are ignored. Method options must precede any required arguments and start with a hyphen ('-'). For your reference, figure 1 gives a graphical overview of the syntactic elements of an avg_q script.

It must be noted that the script syntax is not intended as a general programming language. It only serves the purpose of configuration. There is no equivalent to variables as in programming languages, conditional execution or explicit looping. Therefore, it is not possible to implement any thinkable analysis within an **avg_q** script, without generating intermediate data files and without the help of external programs.

For example, to perform the same analysis on multiple input files or triggers or with a number of different analysis settings, it is of great advantage to have a command interface or batch interpreter at hand providing variables and looping, like the Unix shells or the widely available interpreted languages *perl* or *python*. Since **avg_q** can read the configuration script from its standard input stream, looping can be performed as in the following example script for the Unix shell *sh*. This script calculates the event-related field (ERF) on trigger 1 for any MFX[1] file in the current directory:

```
#/bin/sh
channels=A1-A37,E5
triggercode=1
pre_trigger=1s
post_trigger=2s
for mfx_file in *.mfx ; do
avg_q stdin <<EOF
# First line of avg_q script
get_mfxepoch $mfx_file $channels $pre_trigger $post_trigger $triggercode
average
Post:
writeasc $mfx_file.EF
# Last line of avg_q script
EOF
done
```

[1] MFX: 'Münster File Exchange' binary MEG/EEG data format

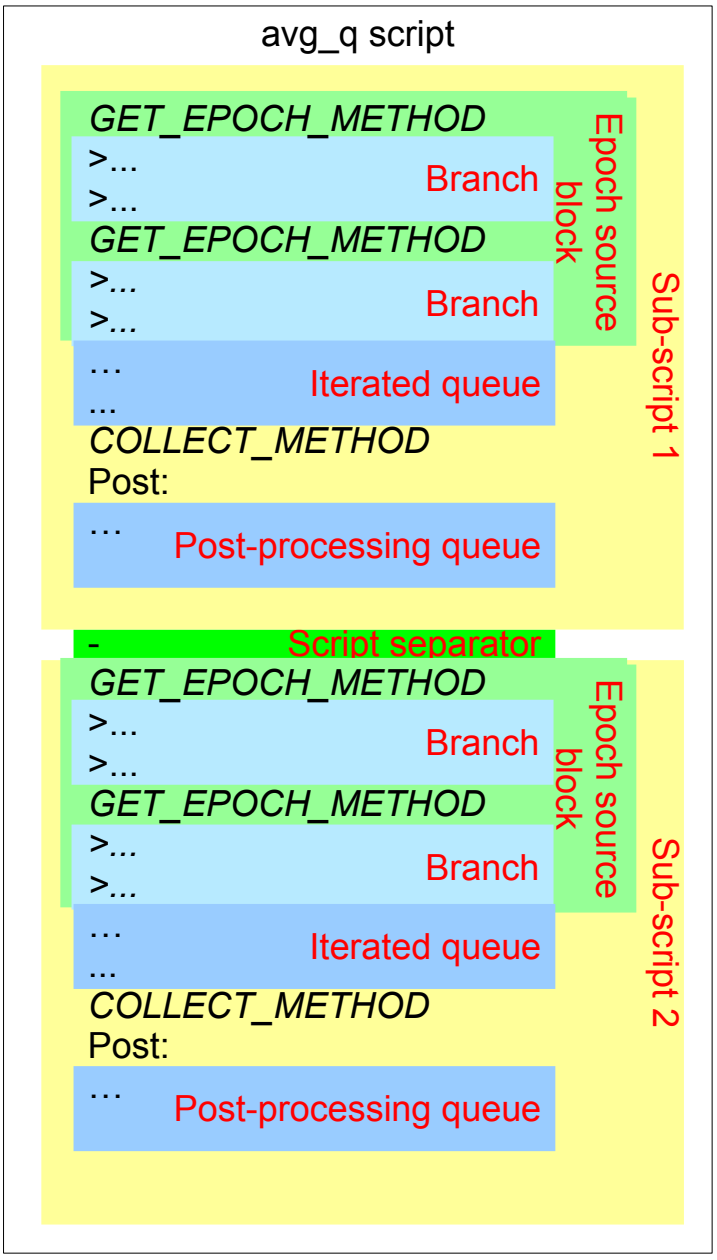

Figure 1: Structure of an avg_q script with two sub-scripts. Words written in *slanted* type are placeholders for a specific method type; "..." can be replaced by any number of TRANSFORM_METHODs, REJECT_METHODs or PUT_EPOCH_METHODs (or none); At least one GET_EPOCH_METHOD and one COLLECT_METHOD are mandatory. If there is a post-processing queue to process the end result of the COLLECT_METHOD, the keyword "Post:" must be present on the line following that method. In branches (processing specific to the epochs yielded by one GET_EPOCH_METHOD), all lines begin with the character ">". Additionally, lines beginning with the character "#" are comments.

The following is the general structure of an **avg_q** script:

1. The first method must be a 'Get_Epoch_Method' providing data epochs to work on.
2. An arbitrary number of 'Reject_Methods', 'Transform_Methods', and 'Put_Epoch_Methods' may follow. Note that the method type 'Transform_Methods' is very broad; there are 'Transform_Methods' that display data or start external programs but do not actually modify the data. 'Put_Epoch_Methods' may be used to write each epoch at any processing stage in any of the available data formats.
3. A 'Collect_Method' that takes all processed data epochs and yields a single data set after the last input epoch has been processed. This method ends the 'iterated queue', i.e. the processing queue that is applied to every single input epoch generated by the 'Get_Epoch_Method'.
4. An optional keyword 'Post:' on a line by itself, followed by an arbitrary number of methods as in (2) above. These methods represent the 'post–processing queue' which is applied to the single data set generated by the 'Collect_Method'. It is usually desirable to either display the result or to write it to a file, since there is no 'default' handling of the result and it will be lost when the script terminates.

Multiple Get_Epoch_Methods and branching

Since May 1996, part (1) was extended to allow multiple 'Get_Epoch_Method's, each on a line by itself. Of such a 'pool' of 'Get_Epoch_Method's, the next method is initialized and used to obtain more epochs as one method ceases to yield epochs. With this feature it is possible to treat multiple files like a single stream of data, for example if files have been split because of mass storage limitations. Another common application is to create grand-averages or to concatenate sub-averages from different files.

In some applications, one may wish to perform different types of preprocessing on data read from different files, or even compare the results of different processing on the same file, and yet see the result as one sequence of epochs. This is implemented by 'branching', which means that a number of transform methods following a 'Get_Epoch_Method' are executed only for epochs from this 'Get_Epoch_Method'. Such methods are marked in the script by preceding them with a '>' sign. During execution, these methods are executed or skipped together with the preceding 'Get_Epoch_Method', thus creating a 'branch' that is executed only on epochs coming from a given 'Get_Epoch_Method'. An example:

```
# First line of avg_q script
readasc file1.asc
>fftfilter 0.1 0.2 1 1
>sliding_average 1 5
readasc file2.asc
>fftfilter 0.4 0.5 1 1
>sliding_average 1 2
average
Post:
writeasc average.asc
# Last line of avg_q script
```

This script resamples all epochs in the first file to one fifth and all epochs in the second to one half of their respective original sampling frequencies and then averages them. Obviously this 'branch skipping' feature should be used with due diligence because it is easy to produce epochs with varying geometry in the output; Some methods cannot handle the case in which the number of channels varies across epochs and others (like the average method) cannot handle varying epoch lengths. Another point to be aware of is that although the syntactic parsing of all lines is done before any execution of the script is started, the actual initialization of a branch takes place only after the prior branches are exhausted and thus possibly after a lot of processing has already been done; for example, '`file1.asc`' in the script above would be processed even if '`file2.asc`' was missing, failing with an error message only when trying to access '`file2.asc`'.

One more advanced feature is 'disguising' a transform method as 'Get_Epoch_Method' by prepending an exclamation mark ('!'). The reason for this construct is that there is sometimes the need to output messages or to run programs safely *between* the processing of two branches. Message output can be used by a subsequent

program to separate data resulting from different files within an output stream - for example when multiple input files are measured using the **write_crossings** method writing to standard output. It is obvious that only methods like **echo** and **run**, which in contrast to most transform methods do not try to access the epoch data in any way, should be disguised as 'Get_Epoch_Method's, since all data pointers are initialized to NULL, causing an exception when accessed. Clearly, a 'disguised' method will only be called once and never yield any valid epoch – if a transform method does not touch the data, pointers will remain NULL, indicating to the executor that the method has no epoch to deliver. An example:

```
# First line of avg_q script
!echo Hello, I'm about to open file1.asc!\n
readasc file1.asc
>fftfilter 0.1 0.2 1 1
>sliding_average 1 5
!echo Finished with averaging file1.asc, proceeding with file2.asc!\n
readasc file2.asc
>fftfilter 0.4 0.5 1 1
>sliding_average 1 2
!echo Finished with averaging file1.asc and file2.asc.\n
# For example, the input files could now be removed:
!run /bin/rm -f file1.asc file2.asc average
Post:
writeasc average.asc
# Last line of avg_q script
```

Multiple sub-scripts in one script file

Since August 1997, multiple scripts as described above ('sub-scripts') can be appended to a single text file (which is still called a 'script'). The sub-script separator is a line starting with a dash '-'. The rest of this line is ignored, so that the separator line can be extended to a full line of dashes to make the separation visually clearer, and to contain other characters and comments. Sub-scripts are independently executed from top to bottom. No global syntax check takes place before execution begins; in fact, the next script is not even read from the file before the execution of the current sub-script is finished. This is done in order to allow a controlling program to start **avg_q** only once and request operations on demand by writing more sub-scripts to it (i.e., **avg_q** reads scripts from a so-called *pipe*). Multiple sub-scripts are also helpful since arbitrary sequential operations necessary for a task can be specified in a single script. The **avg_q** program itself (the 'user interface', see below) also allows to select one sub-script to execute, so that a script file can also be used as a library of operations. This is more useful for 'standalone' versions of the program with built-in script than for the version capable of executing arbitrary scripts.

Protecting white space in arguments by backslashes

Since August 2000, it is possible to protect white space (space and tab characters) in scripts by preceding them with a backslash ('\'). This is necessary because otherwise, white space is taken to separate arguments. The most obvious uses of backslash protection are to select channels by name whose names contain blanks, and to read or write files with names containing blanks. An actual backslash can be specified by a sequence of two backslashes. In order to avoid 'backslash orgies', **avg_q** leaves backslashes not followed by either white space or another backslash intact - channel selection, for example, uses backslashes to protect hyphens from being interpreted as range indicators.

Remarks about the program design

The 'configurable data analyzer' approach has several advantages: The analysis modules implementing the 'methods' are independent of the data source or output formats, which means that testing a data analysis procedure is possible without compiling the methods into a test version of the program and without having to generate test data externally, because Get_Epoch_Methods are available that generate test data. The program can also be easily maintained because there is no need to keep track of several single utilities and versions; and it is easy to add more methods anytime, steadily increasing the abilities of the program without sacrificing any of the former functionality. For example, only input and output methods have to be added for full support of other file formats. Finally, selected methods can also be incorporated in any fixed program (like **ascaverage**, see section 7) from a library, without the configurable queue.

There are, on the other hand, also disadvantages: Most users are familiar with a "document-centric" user interface design, which displays a "document" (in our case a multichannel data file) and shows menus to perform processing steps on it. **avg_q** does not follow this standard, therefore users have to familiarize with the inherent concepts in order to use the program. To many users, for example, it is not obvious why **avg_q** cannot "look back" into features of the original file while processing goes on. The reason is of course the design decision of modularity and independence of the processing methods. Thus, other methods besides Get_Epoch_Methods only see whatever data is stored in memory to represent the epoch. This limits the scope of tasks for which **avg_q** can be used, but on the other hand surprisingly few tasks have surfaced which could not be solved with **avg_q** – Since no programming is necessary, at least a few important and otherwise cumbersome processing steps can always be implemented as an **avg_q** script.

One more annotation appears necessary: Today, a large number of methods have been written for **avg_q**. Some of them, like **posplot** or the sound I/O methods, add significantly to the size of the program because they are built on top of graphics/sound libraries that are large by themselves. This is not a problem on Unix systems, since **avg_q** is usually compiled using shared libraries. However, if size is a problem and a version of **avg_q** will be used only for a limited number of tasks, it is very easy to build versions of **avg_q** containing any subset of the available methods. This means that a compiled version of **avg_q** may or may not include all of the methods described below. A quite detailed description of the included methods is available in every **avg_q** binary and can be requested from the program as described in the next section.

2 avg_q command line interface

2.1 Arguments

The command line version of **avg_q** is simply invoked as

```
avg_q [options] scriptfile
```

Any number of additional arguments can be given to **avg_q**. These arguments replace strings of the form '$*number*' in the pre-parsed script in a way similar to the access to command-line parameters in many script languages, i.e. '$1' expands to the first additional argument and so on. It is an error if less such script arguments are available than requested in the script, but only a warning results if more arguments are available than requested. Note that the script argument expansion is *not* done prior to parsing the script, but rather the selected method parameters are replaced within the 'precompiled' version. Therefore it is not possible to replace a method name or an option switch with a '$*number*' construct. This was designed so that argument expansion could work in exactly the same fashion in precompiled 'standalone' versions of **avg_q** (see below), for which it was primarily intended.

2.2 Options

The available command line options of the **avg_q** program (starting with version 2.5) are:

-s scriptnumber: Execute only this sub-script (counting from 1).

-l: List all available methods

-H: Describe all available methods

-h methodname: Describe method methodname

-t tracelevel: Set the global trace level. Default is 0, higher integers cause more messages to be displayed. A trace level of 1 suffices for most informative messages. Trace level 6 and above output the method name, script line and action (init, exec, exit) for each method being executed.

-D: Dump C source code to the standard output that contains a version of the script that can be directly compiled with the **avg_q** libraries to produce a 'standalone' version. This is only useful for the developer in order to provide people with customized, small programs after testing the scripts with the full **avg_q** version.

The user interface version avg_q_ui has two more options:

-i: Start with the main window iconified. This keeps the script window from cluttering the desktop when batches are run or if only the posplot window needs to be seen.

-I: Interactive, i.e. only load but don't run the script.

2.3 Standalone versions of avg_q

After the main program was used successfully for data analysis for some years, it became obvious that using a configurable general-purpose system was an 'overkill' for many small, conversion-type applications. Even a complicated analysis may call for a fixed monolithic command if its execution belongs to the laboratory routine. For such purposes, a way was devised to produce compiled **avg_q** binaries with a built-in script. The compilable script, consisting of C data structures, is dumped into a source file using a normal version of **avg_q**. Dumping in this fashion is necessary in order to detect syntactic problems in the script and to include only the methods and support modules necessary for the given script. Standalone versions of **avg_q** only support the **-s scriptnumber** and **-t tracelevel** options and a **-D** option that reconstructs the built-in script from the compiled data structures, so that it is always possible to see which actions a standalone version will perform. Otherwise, standalone versions only take script arguments, i.e. arguments that will be passed to methods as defined in the script. Obviously and just as desired, users of standalone versions will generally not need this manual.

3 GUI (Windowing) interface: avg_q_ui

From **g**raphical **u**ser **i**nterfaces such as X-Windows, Microsoft Windows or Apple Macintosh desktops, a command-line interface as implemented by **avg_q** can only be used in conjunction with a terminal emulation program and a text editor. Information about available methods must be requested from the help function of the command line version or looked up from a manual. A native application for these environments should present the user with a window that allows to interactively select the actions to perform. For **avg_q**, this includes the construction of processing queues from the available methods as well as loading, saving, starting and interrupting such queues. The single main reason for implementing a GUI version (named **avg_q_ui**) was that the interactive data plotting method **posplot** could only work reasonably under MS Windows (opening a new window) if the main application was itself a native Windows application. The functionality of **posplot** is essential for **avg_q**, because it allows direct scrutiny of the data at any processing step.
An interactive application turned out to be also quite helpful for the initial creation of scripts (less so for production runs, which are done in 'batch mode' most of the time), avoiding the loop between consulting the help function built into **avg_q**, editing the script using a text editor and running **avg_q**. First, a graphical representation of methods as 'icons' was considered. But it was desirable to use **avg_q**-compatible scripts as the save and restore format to be able to use such scripts interchangeably with the command line version. If all aspects of **avg_q** scripts such as comments, multiple sub-scripts and multiple Get_Epoch_Methods with 'branch processing' should be supported, the clearest design appeared to be to display the script file in a text editor window with special features, such as changing arguments and options of a method via interactive form dialogs. This has the advantage of allowing comments in a natural way and also allowing changes to be done directly in the script wherever this is more comfortable to the user.

avg_q_ui and **avg_q** work with the same method modules. Both only have a list of methods which they can ask for more information, such as method type, argument descriptors and the method description text. Error-handling and tracing use callbacks to the main program. In this way it is guaranteed that a change to a method (or the addition of a method, for that matter) is immediately propagated to all user interfaces and that scripts always execute identically.

As described, **avg_q_ui** was implemented as a text editor with special features. Besides the functionality of a simple text editor, it is possible to insert a new method (*before* the current line) by selecting a method from the menu. There is one menu for each of the method types. A dialog with all options and arguments for the selected method will appear, allowing a successful termination only after all formally required fields are filled out: This dialog is actually a view into the precompiled (configured) ready-to-execute method structure. Each method usually has reasonable defaults for some or all of the fields. Options have a check box in front which indicates whether or not the option is activated, while required arguments just present the text, number or filename input. Filename input fields have a 'select' button to the right side which causes a file selector box to appear. When the 'OK' button is pressed, the text representation of the form just filled out is inserted into the script.

By pressing the **right mouse button** or the **TAB key** within the text window, the current method line is displayed as a dialog box with the current values filled in. When the first word in the line is not a recognized method name, **avg_q_ui** tries to find a method starting with that word. Then, the method is configured using the rest of the line as arguments; if this configuration fails (wrong number of arguments, unrecognized options, etc.), default argument values are substituted just as for methods selected from the menu. The values can be changed as above, and the 'OK' button will replace the old line with the new one in the script. Finally, only the script text is relevant; the conversion between text and form dialog, i.e. compiling and de-compiling, only serves the purpose of specifying method lines correctly.

Variables ($1, $2 ...) can be used in scripts and receive their startup values from the command line in the same way as for **avg_q**. However, the question arises in which way their values can be modified or added interactively. The approach chosen for **avg_q_ui** is to show the *contents* of the addressed variable in the method configuration dialog box. If, for example, a filename is replaced by $1 in a script and the command line does not give $1 a value, the dialog box will display an empty field. Any value entered in this field will be written into the variable $1 ready to be used by the script, instead of inserting it in place of the $1. In this way the dialog box always shows the values that will actually be used but not the variable identifiers. It is implied that variable identifiers must sometimes be entered using the text editor instead of the method dialog box, since not all input elements that can be set to a variable have input fields suitable to enter an identifier (e.g. choice boxes).

The following keyboard shortcuts are defined in combination with the 'CTRL' key in the version using the V GUI library - for the GTK version, the GTK text widget functions are used for copying/pasting text and other keys are menu shortcuts that can be seen in the menu.

d Delete (cut) the current line (and save it to the copy buffer).

c Copy the current line to the copy buffer.

v Paste (insert) the line in the copy buffer before the current line.

f Find a string.

g Find the next occurrence of a string.

l Load a file as a script.

s Save the current script under the last remembered name (or 'avg_q_ui.script').

q Quit the program.

r Run the script.

t Run only *this* sub-script, i.e. the sub-script in which the cursor is currently placed.

x Interrupt a running script.

While a script runs, the name of the method currently executed is shown on the status line of the window together with the script number and line. In case of a configuration or run-time error, the cursor will be placed on the line at which the error occurred, with the status line containing the error message. Trace messages and warnings are written to a second text window that appears only after the first line is written to it. The trace level can be set via the menu as well as a simple single-step mode which causes a dialog box to appear before

each method is executed. The **posplot** method opens a third window which makes some **posplot** commands available in its own menu; the rest of the interaction uses keyboard and mouse as described in section 6 (page 43). A feature that should be noted is that all editing operations in the script window, including the use of configuration dialogs, can be performed while a script runs. It is not possible to start another script, however. In any case, the running script executes from the compiled, internal representation of the text window contents at the time at which it was started, so that editing has no influence on the (sub-)script currently executed.

Multithreading: The GUI can usually only process commands and update the windows within special callbacks before and after methods are executed, while trace messages are output, or within methods which are interactive by themselves such as **posplot**. This means that the GUI is non-responsive for the time in which a single method runs without issuing trace messages. The natural approach to this problem is to start the script as a separate 'thread'. Such multithreaded versions of **avg_q_ui** have two status lines, the left one for the GUI and the right for the script processor. Also, with these versions it is possible to either 'stop' the script the way the single-threaded version did, namely waiting for the current method to finish its operation; or, which is more insecure, to 'cancel' the script by killing the script thread. In both cases, however, the program will attempt to execute the normal exit operations of each method in the queue.

Besides presenting a graphical user interface, **avg_q_ui** understands the same options and arguments given on the command line as **avg_q**, including the script name and values for script variables. If the name of a script is given, the default behavior is to immediately execute it and terminate after successful completion just as **avg_q** would. Interactive mode is only entered in case of an error or if the user interrupts the script. It is possible to force **avg_q_ui** to start in interactive mode showing the given script by using the option **-I**.

Finally it should be mentioned that dumping a script as compilable C code is also supported by **avg_q_ui**, that this output is the same as for **avg_q**, and that corresponding 'standalone' versions of **avg_q_ui** with compiled-in script can be produced. Such standalone versions take values for variables from the command line as usual, but if variables are missing or the middle or right mouse button is pressed in interactive mode, dialog boxes are shown in which the variable values can be entered or modified.

avg_q_ui currently runs under the X-Window system (X11R6) on virtually all UNIX versions and under Microsoft Windows 95/NT.

4 General conventions

4.1 Readasc file format

To store epochs in a way that is very close to the representation used internally by all processing methods, a special file format is used that will subsequently be referred to as 'asc'. There are two versions: an ASCII text version that can be easily read by statistical or spreadsheet programs, and a binary version that is faster to read and write and should normally be used for intermediate storage. The two versions contain identical information. They are automatically recognized when they are read; special options are only needed to request that binary format is written instead of the default ASCII. The file format has the following capabilities:

- The file header stores any number of 'feature=value' pairs on individual lines.
- Any number of epochs with the following attributes:
- An epoch descriptor text (comment) that usually contains information about the experiment the data came from and the names of the processing steps applied to it. Additionally, a textual z axis label and a numerical 'z value' (like 'Lat[ms]=50') may be contained in this descriptor. The comment may be preceded by 'feature=value;' pairs containing properties local to the current epoch.
- Each epoch has its own list of channel names and positions as well as its own data size descriptors. The 'channel name' associated with the additional 'x-value' column usually specifies the name and unit of the x data as mentioned above for the z axis label.
- The data in each epoch is organized as a data table with the channels as the columns and any number of rows that are associated with x-values. On each channel × x-value field, multiple data entries ('items') may exist, forming a data tuple. The tuple size is fixed for the whole epoch. Multiple items are used e.g. to store real and imaginary parts of spectral coefficients or statistical information for each data point.

The 'feature=value' pairs mentioned above are used to store information that might be used for processing or ignored; e.g. the total number of epochs that lead to this result (the 'nrofaverages' property used for statistical analysis: Nr_of_averages=#) or the number of items in each field that are protected from normal numerical processing (such as counts: Leaveright=#). Also, epoch trigger information is written, if present, as a Triggers= field.
When data is written using **writeasc**, most attributes are written to the epoch header, since they usually differ across epochs.

4.2 ARRAY_DUMP file format

The 'ARRAY_DUMP' format is an ASCII-only format that is used to output single data maps to external programs like contour-plot software or mathematical tools such as MatLab. It does not contain the channel description facilities of the asc format, but is suitable to represent numerical vectors or matrices. The first line, starting with a hash sign ('#'), identifies the format (ARRAY_DUMP ASCII or MatLab), then the number of columns and rows are given and the matrix data follows. The MatLab version, for which only the header differs, can be read directly by MatLab.

4.3 Trigger file format

Some Get_Epoch_Methods process continuous files from which epochs could be read around arbitrary time points. A simple ASCII trigger file format is supported by **avg_q** in order to represent lists of such time points or triggers, each of which can have an integer number or 'trigger code' assigned to it. In trigger files, lines beginning with a hash '#' are comments and skipped; otherwise, each line should contain a pair of values of the form 'point code' giving the point number around which to read and the trigger code assigned to this point. Any further columns appearing in the file ('description') are read and written transparently, and displayed by posplot, but not used for epoch selection. This feature is, for example, used by **write_crossings -E**, which creates a third column containing the measured value of the extremum. The point number is the sampling point number relative to the first point in the file, starting with 0. It can be given in time units by the general convention below (4.4).

A trigger code of 0 ('point' being irrelevant) signals the end of the trigger list just as the End of File does. This is useful when, instead of an actual file name, 'stdin' is used as a trigger file name. This will read trigger lines from standard input. Usually, no End of File condition is desired on the standard input stream; instead, the end of the trigger list is marked by a zero trigger code.

This might be a good moment to note that representation of triggers or markers is one of the big incompatibility issues between data formats. **avg_q** alone cannot solve these issues completely. While triggers represented as lists (most often appended to the end of the raw continuous data) can be handled relatively efficiently and are even internally kept with each epoch and written to some output formats, especially the surprisingly common storage in dedicated 'marker channels' is tedious to deal with. The detection of steps or peaks in such channels is a signal processing task in its own right rather than an aid for analysis. Of course, dedicated **avg_q** methods for a given file format handle their specific trigger format (one or several trigger 'channels', binary trigger lists appended to the file or stored separately and so forth); When dealing with random access to data segments, however, extra steps may be needed to produce an ASCII trigger list first.

4.4 Entering point numbers or time values

Wherever a time value (duration, window length etc) is required in **avg_q** - be it in a method configuration or a trigger file, a number without any suffix is treated as a number of sampling points. By appending the unit shortcut (without intervening space), **avg_q** uses the known sampling frequency value ('sfreq') to compute the corresponding number of points. Unit shortcuts are 'us' (microseconds), 'ms' (milliseconds), 's' (seconds), 'min' (minutes) and 'h' (hours).

4.5 Channel selection by name

Many of the methods available in **avg_q** can take a list of channel names as an argument. Such channel lists have several features and are generally denoted as 'channelnames' in the argument lists. The channel names in the list must be separated by commas; white space is not allowed. A range of channels ending in consecutive numbers, such as 'E2,E3,E4' may be abbreviated using a hyphen, as 'E2-E4'. If hyphens or commas belong to a channel name, they must be protected by preceding them with a backslash ('\').
Normally, it is a fatal error if any of the named channels does not exist. This behavior can be relaxed by preceding the whole list with a question mark. For example, '?M12,E30-E128' selects any of the named channels that are found.

The expansion of numbers in channel names zero-pads each output number to the length of the first number, so that 'E001-E200' yields the expected result. 'E001-E1000' is zero-padded only up to length 3 but not to length 4, just as 'E1-E29' is not padded.

An exclamation mark prefix '!' negates the selection: '!?M1-M11,E1-E29' selects all channels that do not bear one of the names in the list. At this point, a few remarks about channel order and channels with equal names should be made. The 'positive' list (without the exclamation mark) determines an order of the channels: For example, if channels should be copied (**remove_channel -k**), the order in which they will appear in the output multichannel data set is the order in which channels are mentioned in the list. However, with negative lists and within sets of channels with equal names, the order is determined by their sequence in the original data set. For example, if a data set contains channels $M1_a$, $M2_a$, $M1_b$ and $M2_b$ in this order, 'M2,M1' will select $M2_a$, $M2_b$, $M1_a$ and $M1_b$.

A special value for channelnames is 'ALL', which selects all channels present in the file.

4.6 Metadata handling

avg_q uses a number of metadata entries with each epoch. Obviously, the data dimension is defined by the numbers of channels, points per channel and items per point. The sampling rate (sfreq) is mandatory. Additionally, channel name and position information is mandatory (positions being defined in three dimensions but often using only the first two dimensions for a flat screen layout). All Get_Epoch_Methods generate this data; if it is not explicitly available in the file format, default routines are used to create numbered channel names ("1", "2", ...) and a square flat channel layout.

xdata, a special "channel" holding x axis values, is not mandatory; However it is generated when needed using a standard routine. The metadata involved is sfreq, beforetrig and aftertrig. These three values are sufficient to create a default epoch-local time axis. A heuristic based upon the sampling rate is used to determine the time unit (µs, ms, s, min or h, the same as in 4.4) and the x axis name constructed as "Lat[*unit*]". If the epoch contains frequency-domain data, the x axis name will be "Freq[Hz]" and the values constructed using nroffreq and basefreq properties of the epoch. 'set xdata' can be used to specify the unit explicitly.

5 Methods within avg_q

5.1 Methods to read or generate data: Get_Epoch_Methods

Get_Epoch_Methods determine which data is subsequently processed. They are asked for epochs of data until they are 'exhausted'. When this happens and no other Get_Epoch_Method is available, the Collect_Method of the script is asked for an end result, which may then be postprocessed in the 'Post:' queue before the script terminates.
One convention pertains to all available Get_Epoch_Methods: The options -f and -e, specifying the starting epoch and the number of epochs to be (at most) delivered, apply to the epochs 'pre-filtered' by an eventual trigger code and other constraints. This means that -f and -e select from the stream of epochs that would be processed if these two options were omitted, *not* from the stream of physical epochs (or triggers) in the file. For example, it is not possible to specify something like 'from the first 50 epochs, select only those with trigger code 5', since the order of options is irrelevant in **avg_q**.

Since a complete data set in memory always includes channel names and positions but not all supported file formats contain this information, the corresponding methods construct reasonable defaults; as last resort, the channel names are set to numbers starting with 1 and the positions are set to a regular grid in the x-y plane at z=0.

dip_simulate: Simulation method to simulate dipolar sources with defined time behaviour

Arguments: sampling_freq nr_of_epochs beforetrig aftertrig sim_module_name [(args)]

Modules:

eg_source: Example source description, no options or arguments. This can be used as a simple intrinsic data source for testing. Three dipoles are simulated, two harmonic and one with noise time course.

var_random_dipoles: Simulates any number of dipoles with configurable location and (maximum) momentum, each dipole either showing bursts of harmonic activity of a specific frequency or a randomly fluctuating amplitude. Each dipole is defined by seven arguments; if they are missing, they are set to fixed default values or determined using a scheme specified by keywords. The 'frequency' values used here are relative to half the sampling frequency, i.e. 0.2 would mean two fifth of the sampling frequency.

Arguments: n_harmonics n_noisedips [freq1 pos1x pos1y pos1z moment1x moment1y moment1z freq2 ...].

Optional keywords: FREQ_DEFAULT freq; RANDOM_POS radius; RANDOM_MOMENT amplitude

The keywords and their arguments may be specified anywhere between two of the 7-item dipole descriptor blocks or at the end of the descriptors. RANDOM_POS and RANDOM_MOMENT generate random 3-vectors on a sphere with the specified radius. This can be used to simulate 'brain noise' by a sheet of noise dipoles.
Each harmonic dipole will burst with a smooth envelope of between 0 and 10 cycles random duration, the quiet times between two bursts having the same duration characteristics. The starting phase of the wave that is modulated by the envelope is also randomized. Noise dipoles are not switched on and off but are active all the time. The 'freq' parameter is ignored for them.

ramping_dipoles: While var_random_dipoles generates a continuous stream of burst events that is packaged into epochs by dip_simulate, the events generated by ramping_dipoles are locked to specified latencies within each epoch.

Arguments: n_harmonics n_noisedips [freq1 start_at1 duration1 risetime1 falltime1 pos1x pos1y pos1z moment1x moment1y moment1z freq2 ...].

All 'times' specified here are in time points. Every dipole is started (the harmonic contents with random phase as above) at point 'start_at' with a total duration of 'duration' points. Within this duration, the starting ramp (a factor varying linearly between 0 and 1) is 'risetime' points wide and the ending ramp 'falltime' points. The only difference between harmonic and noise dipoles here is the time series that is subjected to this envelope.

get_mfxepoch: Get-epoch method to read epochs from an MFX ('Münster File eXchange') file. Epochs can either be selected by values in the trigger channel recorded parallel to the data (where each sampling point contains a trigger code and a 'valid' flag), or by a fixed epoch length specified in the file header.

Arguments: Inputfile channelnames beforetrig aftertrig trigcode

Options:

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The actual trigger position is shifted by offset

-t trigname: The channel named trigname is used as trigger channel instead of 'TRIGGER'. 0=epoch mode

Only the channels specified by the channelnames channel list (cf. section 4.5 on page 10) are read from the file.
If trigcode=0, any valid trigger code is accepted to specify an epoch. If aftertrig=0, the length of the returned epoch is determined by the distance between the current and the next trigger. An example to read complete epochs in epoch mode is 'get_mfxepoch -t 0 Inputfile channelnames 0 0'.

null_source: Null get-epoch method. This method generates all-zero data sets.

Arguments: sampling_freq nr_of_epochs nr_of_channels beforetrig aftertrig

Options:

-I Items: Specify the number of items per channel and sampling point (default: 1)

readasc: Get-epoch method to read epochs from asc files.

Arguments: Inputfile

Options:

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

-c: Close and reopen the file for each epoch. This can be useful if external programs are started from the queue using **run**.

read_brainvision: Read Brain Vision (tm) files. This format consists of at least three files with endings .vhdr, .vmrk and .eeg. The .vhdr file is the main header file explicitly containing the names of the other two files, therefore this file must be specified as Inputfile. Markers are contained in the .vmrk file, while the actual data is in the .eeg file. The other files are opened with the same path as the .vhdr file. Note that when the files are renamed from whatever was specified during saving, the names in the .vhdr text file must be changed as well.
Also, markers or event types are stored as text labels rather than with numerical codes as required by avg_q; read_brainvision has a built-in list to convert known labels into codes. Unknown labels appear as code -1.

Arguments: Inputfile beforetrig aftertrig

Options:

-c: Continuous: Read a continuous file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_freiburg: Get-epoch method to read data in the binary ERP and sleep EEG format created at the Psychiatric University Clinic in Freiburg. The variant of this format without .coa file (see below) does not include channel names or positions, and codes the data as short integer values. However, **read_freiburg** supplies channel names automatically depending on the number of channels employed. Channel positions can then be set using "set_channelposition =PSG". If no setup for the given number of channels is known, the channels are named by number and arranged on a grid.
If the Inputfile argument specifies the name of a directory, (compressed) single trials are read from separate files arranged in its subdirectories. The names of the subdirectories are constructed from the name of the directory by appending numbers $00-99$, and each subdirectory holds up to 100 trials whose names are made up of the (lowercase) last letter of the directory name, the digits of the subdirectory and two more digits $00-99$.

If the Inputfile argument is the name of a file, then this file is read as an average (uncompressed single epoch). The nr_of_channels argument is only used for average files, since the information is missing from these. In this case, the number of points in the epoch is computed from the file length. The nrofaverages property is correctly read from averaged files.
If the option -c is given, an epoch length must be specified instead of nr_of_channels, and the file is treated as a continuous (sleep) file (cf. **write_freiburg**). First, the method looks for Inputfile; if it finds it, it is opened as an OS/9 sleep file. If not, it looks for Inputfile.co, which is opened as an MSDOS sleep file. If the (text) file Inputfile.coa is also found (cf. **write_freiburg**), then channel names and sensitivities are read from this file and the data will have the unit microvolts. If it is not found, a warning message is issued and the output consists of the raw integer values just as in the case of the OS/9 variant. Sleep files are different from the single trial format in that all data sections are written consecutively to a single file and represent a continuous recording.
File repair issues: Due to erroneous storage media or programs, the files in the ERP format variant were often damaged. Quite often, the sampling frequency is not correctly set in the header; it can be set with option -s. Option -z is a workaround for some files that had zeros inserted as every 513th data point. Such points in which all channels are zero are tested for and skipped if this option is given; a warning message is issued if any points were actually omitted. And if data must be read that lacks a header at all, the number of channels must be given explicitly with option -C and a file offset to start at with -R. The latter must usually be adjusted by trial and error to have the channels on their correct positions...

Arguments: Inputfile {nr_of_channels | epochlength}

Options:

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

-s sampling_freq: Explicitly specify the sampling frequency in Hz

-C Channels: Explicitly specify the number of channels (continuous only)

-R Repair_offset: Don't use the header, start at this file offset (continuous only)

-c: Continuous mode (for sleep files)

-z: Check for and eliminate data points with all channels zero

read_generic: Generic get-epoch method to read only the data part of files in which data is stored in successive elements of a known type. Since no header is read, all meta-information must be given by arguments or options. data_type can be uint8, int8, int16, int32, float32, float64 or string. Here, 'uint8' is an unsigned character. If aftertrig is set to zero in continuous mode, then the whole file is read as a single epoch.
The 'string' data type provides some ability to read data from text files. The numbers are read from the file successively. Any character that cannot belong to a number (as [+-0-9.eEdD]) is a valid separator. One or more consecutive separators separate two numbers. The end-of-line character is only special in that a line beginning with a hash sign ('#') is skipped. Triggering and 'fromepoch' selection do not work on text files, because they would have to be completely processed in order to determine which data point starts where in the file. If triggering is needed, the text file should first be read continuously with this method and stored in some binary format. The last issue with the 'string' data type is that the number of values in the file must be exactly a multiple of the epoch size; otherwise a fatal read error will result in the last epoch (this does not occur for the binary types because here the number of complete epochs can be calculated from the file size). This read error can still be avoided by limiting the number of epochs attempted to be read using the '-e' option.
read_generic and **write_generic** can both handle multiple items per channel and sampling point. Items always form a tuple of adjacent values, independent of whether channels or points vary fastest in the file.

Arguments: Inputfile beforetrig aftertrig data_type

Options:

-S: Swap byte order relative to the current machine

-P: Points vary fastest in the input file ('nonmultiplexed')

-c: Continuous: Read the file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3. This is only useful in conjunction with the -R option.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

-s sampling_freq: Specify the sampling frequency in Hz (default: 100)

-x xchannelname: Read the x axis data as the first 'channel' and give it this name. Note that the analogy to a 'channel' of data is not perfect since the x axis data always represents only a single item.

-C Channels: Specify the number of channels (default: 1)

-I Items: Specify the number of items per channel and sampling point (default: 1)

-O File_offset: Skip this many bytes (with the 'string' data type: lines) at the start (default: 0).

-B Block_gap: Skip this many bytes between 'blocks'. The 'blocks' meant here are the contiguous areas belonging to one sampling point (or, with the -P option, to one channel). This option allows to skip fields that can or should not be read.

read_hdf: Get-epoch method to read epochs from HDF (Hierarchical Data Format) files. This is a data format and data access library from the National Center for Supercomputing Applications (NCSA <softdev@ncsa.uiuc.edu>) that is, together with netCDF, a standard for the cross-platform data representation of scientific data. The HDF 4.0 specification is used that is converged with Unidata netCDF, i.e. netCDF files can be read transparently. The data is stored as floats, and **write_hdf** writes meta-information like the sampling frequency, channel names/positions and the trigger list as annotations. Therefore, HDF is quite close to the asc file format in terms of conservation of **avg_q**'s internal epoch information as far as epoched data is concerned. In addition, it can also contain data sets with unlimited first dimension, corresponding to "continuous" data in other formats.
read_hdf will read one, two and three-dimensional data arrays. Any arrays of higher dimension are silently skipped. If beforetrig and aftertrig are given, only this many points of each data set are read. For epoched data written by write_hdf, both can usually be omitted and all epochs will be restored as written. Continuous data will, as with the other get-epoch methods, be read around triggers either stored within the HDF file or supplied via an external trigger file. To read continuous data in chunks of a given size, the -c option must be supplied. Caution: If beforetrig and aftertrig are omitted, the whole file will be read as a single epoch in this case.

Arguments: Inputfile [beforetrig [aftertrig]]

Options:

-c: Continuous: Read a continuous file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_kn: Get-epoch method to read epochs from files in the binary Konstanz format (by Patrick Berg). This format does not include channel names or positions, and codes the data as short integer values and a short integer divisor for each channel (see **write_kn**). The channels are named by number and arranged on a grid by default. If other positions are desired, they must be set using **set_channelposition**.

Arguments: Inputfile

Options:

-t trigger_list: Restrict epochs to those marked with a 'condition' code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3

-T string: Specify how trigger codes are formed from the 3 condition and 5 marker values available in a trial, e.g. 'm5c2' would mean that the lowest byte of the trigger code will be the first 'condition' value and the higher byte will be the fifth 'marker' value. Default is 'c1', i.e. only the first 'condition' entry is used.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_labview: Get-epoch method to read epochs from LabView (TM) files. Currently, only a special Mannheim (U.Ebner) variant is supported which stores data in fixed "epochs" of 1s. You need to write this in a supported continuous format in order to get triggered access etc.

Arguments: Inputfile

Options:

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

read_neurofile: Get-epoch method to read the NeuroFile II format by Nihon Kohden. This is a binary format using one byte per sampled value that codes a difference to the preceding value. The binary data is stored in a file whose name ends in '.eeg', while descriptive data is stored in another file with the extension '.dsc'. Thus, the 'Inputfile' argument to **read_neurofile** should be given without extension. Note that **read_neurofile** always uses the last values in the preceding epoch as starting point for the difference procedure for the next epoch, which is only correct if the data is read continuously instead of using a trigger file. The uncertainty about the level of the first data point, however, is introduced by the data format.
Besides the 'fast' channels whose data is stored in the '.eeg' files, there may also be some 'slow' channels sampled at a fixed rate of 256/30 Hz whose data is stored in a file with extension '.cpl'. If desired, these files can be read using **read_generic** with data type 'int16', Intel byte order.

Arguments: Inputfile beforetrig aftertrig

Options:

-c: Continuous: Read the file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3. This is only useful in conjunction with the -R option.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_rec: Get-epoch method to read data in the binary REC sleep data format. This format has been designed by a number of sleep laboratories for data exchange, as described by Kemp et al. (1992). It is also called EDF (European Data Format). The format consists of file and channel headers in pure ASCII and data in binary, Intel-ordered short (16-bit) integers. In REC files, each channel has its own name, scaling and offset, but no position. **read_rec** thus arranges the channels on a grid. Data is internally stored in blocks called records in order to accomodate different sampling rates for different channels. avg_q uses the maximum sampling rate present in the file for all channels, and values sampled slower are extended by replication. **read_rec** also transparently detects and reads the 24-bit variant called BDF. avg_q reads EDF+ or BDF+ (Kemp and Olivan, 2003) annotation markers as triggers; a trigger code 1 is assigned

to the start and for events with non-zero duration a second trigger with the same annotation text as description and a trigger code -1 is assigned to start+duration. Time-keeping annotations are discarded. As usual, triggers can also be supplied using a trigger file.[2]

Arguments: Inputfile beforetrig aftertrig

Options:

-c: Continuous: Read the file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3. This is only useful in conjunction with the -R option.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_sound: Get-epoch method to read data from any of the sound formats supported by the SOX (SOund eXchange) package (Maintainer: Lance Norskog <thinman@netcom.com>, freely available from various FTP servers). This library is interfaced on a very high level, so that all capabilities and most of the behavior of **avg_q**'s sound methods are directly due to this package; see the description of the **write_sound** method. The file type is recognized primarily from the file extension and only then from the file contents. **avg_q** enumerates the channels and arranges them on a grid, because channel names and positions are not usually specified in sound files. The 32-Bit integer sound representation internal to SOX is not changed by **avg_q**, which means that the amplitudes read are generally of the magnitude $\pm 2 \cdot 10^9$. Just as the **write_sound** method can output directly to an audio device, **read_sound** can record directly from an audio device (extension .ussdsp or .sunau).
An epochlength of 0 points causes **read_sound** to try to read the whole file as one epoch. As the SOX library provides no means to determine the number of samples beforehand, this automatic determination of the epoch length involves reading up to the end of file and then seeking back to the start for the actual read. Therefore, this feature cannot be applied on non-seekable input, e.g. when recording.

Arguments: Inputfile epochlength

Options:

-H: Help. Query the SOX library for supported formats and exit.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_synamps: Get-epoch method to read epochs from files in the binary NeuroScan formats continuous (.CNT), epoched (.EEG) or averaged (.AVG). The method infers the actual file type from the file header, so that the file name does not matter. It must be noted that there are presently four different continuous formats that I know of ('CONT0', '100/330 kHz', 'DCMES' and 'SynAmps') and that are supported by **read_synamps**. The main difference is in the way triggers are stored: 'CONT0' and 'DCMES' have special trigger traces (2 and 1, respectively), while the other formats have an event table. In continuous and epoched format files, the data is coded as short integer values (16 bit resolution) and a float sensitivity for each channel; for the averaged format, it is coded as floats. Irrespective of the internal format differences, **read_synamps** reads all variants transparently.

Channel names and channel positions (x-y screen arrangement) are read from the file. For the continuous formats, epochs are read around each trigger present in the trigger trace(s) or included in the event table. The codes for different NeuroScan event types ('StimType', 'KeyBoard', 'KeyPad' and 'Accept') are mapped to a single value in the following way: 'StimType' codes are positive between 1 and 255, 'KeyPad'

[2] Note that read_rec arguments have changed in version 4.5, adding full support for triggered reading.

codes are negative between −1 and −15 and 'KeyBoard'+1 codes negative multiplied by a factor of 16[3]. The key F2 pressed on the Scan 3 EEG recording system corresponds to $-(3*16)$ while it is $-(1*16)$ if inserted by Scan 4. In addition, the somewhat ill-defined 'Accept' field can have the value ACCEPT or REJECT, in which case it codes an attribute of the current trigger interpreted accordingly if option -r is given, or it may represent an event in its own right if no other codes are set, namely a Start/Stop event (translated to code 256), a DC offset correction event (translated to 257), or the start and end of a reject block (translated to 258 and 259, respectively). A Start/Stop event is present at the end of each continuous file and at the points at which acquisition was interrupted. The inclusion of these special events in the stream of normal triggers means that using the option -t is always recommended with continuous files in order to restrict triggering to the intended events. Combinations of codes cannot occur because simultaneous events of different type are split into distinct events by **read_synamps**.

If the option -R is given, triggers occurring in the continuous input file itself are ignored; instead, triggers are read from an ASCII trigger file as described in section 4.3, page 9.

If beforetrig and aftertrig are zero, then **read_synamps** tries to automatically determine the epoch size to be read: From epoched files, epochs are read in the size they are stored, and from continuous files with option -c, the *whole* file is regarded as a single epoch. In continuous files without option -c, an error message results because there can be no reasonable default for the epoch size.

The (possibly remapped, cf. option -K) condition code and an optional list of other triggers occurring within the read epoch (option -T) are passed with the epoch and can be used by successive methods like **write_kn**. In epoched or averaged formats, no event table is available, epochs have a limited length and a defined time zero within the epoch. beforetrig and aftertrig times count from this point backward and forward, respectively, a value of zero being interpreted as request to extend the epoch to the corresponding limit in the file. Since an averaged file only contains a single epoch, any epoch-restricting options (-f, -e, -t) are ignored for such files. The nrofaverages property is correctly read from averaged files.

Arguments: Inputfile beforetrig aftertrig

Options:

-t trigger_list: Restrict epochs to those marked with a 'StimType' value contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9

-B: Do not load channels marked as Bad

-r: Do not load epochs marked as rejected (epoched or SynAmps files only)

-c: Continuous: Read a continuous file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-K: Konstanz remapping of trigger_list values to $1 \ldots n$

read_tucker: Get-epoch method to read binary files in the 'Tucker' format used with the EGI 'Geodesic Net' EEG system. Currently 'simple binary' export format versions 2, 4 and 6 are supported (short integer, float and double samples). Triggers can be contained in specialized channels, with each channel coding for only one event type (or condition). The rising ramp of a trigger signal in trigger channel 1 is interpreted as an event with condition code 1 and so on. No channel name or position information is available.

Arguments: Inputfile beforetrig aftertrig

Options:

-c: Continuous: Read a continuous file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

[3] On February 12, 2004 the fact that Scan 4 introduced all-zero events to represent F2 while F1 is reserved to the help system lead to the change that now 'KeyBoard'+1 is coded here; Codes previously seen as −16 are now −32 and so forth. The previous mapping of Scan 4 F2 events to −256 does no longer occur.

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

read_vitaport: Get-epoch method to read binary files in the 'vitaport' format used by the portable multichannel amplifier/data acquisition system with that name. Data is stored as either bytes or short integers with scale and offset information, and different channels may have different sampling rates. **read_vitaport** mainly supports the so-called 'reconfigured Vitaport II format' in which certain header extensions are present and in which the data is stored non-interlaced, i.e. with the data from each channel stored separately. Raw Vitaport II format (the format in which the data is originally stored on the PCMCIA cards of the portable system) can only be read in continuous mode. All channels are automatically resampled to the highest sampling rate present in the file. Channel name information is available. A special channel with the name 'MARKER' is used for the triggers, and the trigger code is the level to which the flank raises.

Arguments: Inputfile beforetrig aftertrig

Options:

-c: Continuous: Read a continuous file in adjacent chunks of the given size disregarding triggers

-T: Transfer a list of triggers within the read epoch

-M: Use the VITAGRAPH off-line marker table at the end of the file rather than the MARKER channel for triggers. No trigger codes are stored in the table; there is, however, the convention that markers at even millisecond offsets are 'start' and at uneven offsets 'end' markers. This is expressed as code 1 and 2, respectively, for **avg_q**.

-R trigger_file: Read trigger points and codes from file trigger_file in the format described in section 4.3, page 9.

-t trigger_list: Restrict epochs to those marked with a condition code contained in the trigger_list. The list consists of comma-separated integer values, for example: 5,2,3.

-f fromepoch: Specify start epoch beginning with 1

-e epochs: Specify maximum number of epochs to get

-o offset: The zero point 'beforetrig' is shifted by offset

5.2 Methods to accept or reject data sets: Reject_Methods

assert: Rejection method to assert conditions for values of various system variables associated with the given epoch (cf. **query** to query and **set** to manipulate most of these variables). var_name can be any of the following words:
sfreq nr_of_points nr_of_channels itemsize leaveright length_of_output_region beforetrig aftertrig nroffreq nrofaverages accepted_epochs rejected_epochs failed_assertions condition z_label z_value comment channelname xchannelname nr_of_triggers
comparison is any one of: == != < <= > >= = ! (equal, not equal, less than, less or equal, greater, greater or equal, matching, not matching)
value is the value to compare with; comparison follows the type of the variable, i.e. alphabetical with string variables, as integer with integer numerical variables and so on. Matching can be read as 'contains' for string variables; for numerical values, it is the same as equality.
For example, this method can be used to assert that all processed epochs have a certain number of channels or consist of at least a given number of averages when creating grand averages.
The 'channelname' assertion is special in that the condition is evaluated on every single channel. In order to succeed, the comparison must be met by at least one channel for the == comparison and on all channels for all other comparisons.
The 'failed_assertions' variable can be used to create complex assertions by counting how many previous assertions failed.

Arguments: var_name comparison value

Options

-E: Create an error (terminate the script) when the assertion fails, instead of just rejecting the current epoch.

-S: Stop the iterated queue when the assertion fails. The current epoch is rejected and additionally, the get_epoch_methods are not asked for further epochs. The postprocessing queue is executed normally. This can be used to collect epochs until a certain condition is met.

reject_bandwidth: Rejection method to reject epochs in which any channel exceeds a given bandwidth (max-min value). This is most often needed to exclude raw data epochs with large-amplitude artifacts from further processing. However, the choice can also be inverted using the option -I, thus selectively choosing epochs with high bandwidth. Using the option -m, it is also possible to use the absolute maximum as the criterion rather than the bandwidth.
The option -C specifies that the rejection should act not on the whole epoch, but rather only on the channels exceeding their specified thresholds, causing these channels to be deleted from the epoch. In this case, the option -I also inverts the selection, i.e. all channels *not* exceeding their thresholds are deleted. The whole epoch is only rejected if no channel would remain. It should be clear that changing numbers of channels are somewhat exceptional; the further processing steps have to be carefully chosen. For example, using the **average** method without the option -M will result in an error message (option -M tells it to match channels by name). Most output data formats cannot handle changing numbers of channels (the Readasc format does). One of the meaningful postprocessing steps would be to use **collapse_channels** without arguments to average across the remaining channels, producing a single channel in any case.

Arguments: [number] [filename], where

- number is a bandwidth to use as default for all channels; if number is left out, the default is 3500e-15 (3500 fT).
- filename is the name of a file containing explicit assignments of bandwidths for a number of channels of the form:
  channel_number bandwidth_limit ..., or (with option -n):
  channel_name bandwidth_limit ...
  Lines starting with a hash are regarded as comments.

Options:

-m: Maximum only. Reject where the maximum of a channel exceeds a threshold, rather than the difference between maximum and minimum.

-C: Channel mode. Remove individual channels due to the criterion.

-I: Invert. Reject the epochs that would otherwise have been accepted and vice versa.

-n: Select channels by name rather than by number in the bandwidth file.

-i nr_of_item: reject on this item number ($\geq$0; default: 0)

reject_flor: Rejection method to reject any epoch containing the Tübingen error marker 9999

Arguments: NONE

5.3 Methods to perform operations on data sets: Transform_Methods

add: Method to add to all points in the the incoming epochs either a constant value (if no command but a value is given), the negative of the map mean (command='negmean'), of the map minimum ('negmin') or map maximum ('negmax'). Variations that work on channels rather than on maps are called 'negpointmean', 'negpointmin' and 'negpointmax'. As an example, **add negpointmin** followed by **scale_by invpointmax** will map all the data, for each channel individually, to the interval $[0, 1]$. Things like this are often needed for threshold detection tasks, e.g. using the **write_crossings** method. Similarly, an arbitrary quantile can be added using command='negquantile' with a value set to the quantile desired ($[0, 1]$). If command='noise', random values evenly distributed in the interval $[-\text{value}, \text{value}]$ are added to

each input value[4] command='gaussnoise' adds zero-centered Gaussian noise with value as the standard deviation. Finally, if command='triggers', then the trigger code is added to all (selected) channels at the latency indicated by each respective trigger. The most obvious use for this feature is to add (one point wide) trigger spikes to an all-zero channel to serve as a trigger channel for software which needs triggers in such a form.

Arguments: [command] [value]

Options:

-i nr_of_item: act only on this item number (≥0)
-n channelnames: Act only on the given subset of channels.

add_channels: Transform method to add channels, points or items from the asc file add_channels_file. The number of points and items (resp.: channels and items or channels and points) in add_channels_file must be identical to that in the epoch currently in memory. If channels are added, the new channels are imported together with their names and positions. All file information like sampling rate, baseline offset etc. is taken from the epoch in memory. Appending sampling points or items rather than channels is done by **add_channels** as well (options **-p** or **-i**) because these are technically very similar operations on the data matrix. Option **-l** causes a similar operation to **append -l**: The epoch from add_channels_file is linked in memory at the end of the current list or stack of data sets and will be displayed as separate data set by **posplot**.
By using the **-n** switch, an arbitrary combination of channels from the add_channels_file can be specified for addition (this option has no effect in combination with **-l**, though). As usual, the channels are treated in the order given in the channelnames list and channels can be mentioned multiple times. If adding points or items, the number of chosen channels must equal the number of channels in the current epoch.
By default, the epoch from add_channels_file is not advanced when processing multiple epochs; this means that the same data is appended to each incoming epoch. This behavior was chosen to be similar to that of the **subtract** method and can be changed with the option **-e**.

Arguments: add_channels_file

Options:

{ -c -p -i -l }: Add channels (default), points, items, or link the new epoch in memory (append as the last data set)
-f fromepoch: Start with epoch number fromepoch (≥1)
-e: Advance the add_channels_file epoch for each input epoch processed
-n channelnames: Only add the named channels

add_zerochannel: Transform method to add a single channel with all-zero data to the current epoch. This is useful directly before rereferencing EEG data for the first time, in order to explicitly add the data of the normally implicit reference channel. After rereferencing, this channel will obtain useful values.

Arguments: channelname x y z

baseline_divide: Transform method to divide the data by the baseline mean.

Arguments: NONE

baseline_subtract: Transform method to subtract the baseline mean from the data.

Arguments: NONE

calc: transform method to apply elementwise transformations to the input data. The behavior of the functions acting on single floating-point values is to apply the function on all items of each input element except the 'leaveright' last items. Functions acting on multiple items (like the complex functions) are applied only once per element. The -i option specifies the number of the first item to use in this case. In no case does the **calc** method change the item count of the current epoch.

[4] **add noise** superseded the special method **add_noise** in 08/1999.

As usual, 'exp' denotes the exponential to the natural basis e. 'logdB' calculates the amplification ($dB > 0$) or attenuation ($dB < 0$) of power in Decibels. Because **avg_q** cannot know whether the values represent amplitude or power, the latter is assumed (note that logdB is just 10·log10). 'inv' calculates the inverse, i.e. $1.0/x$.
'ceil', 'floor' and 'rint' round to the next higher, lower or the nearest integer, respectively.
The complex functions returning real values (abs2, square2, phase) follow the notion of complex-valued output, i.e. they set the second item (the imaginary part) to 0. **extract_item** can be used to make the real-part item the only one if this is desired. The absandphase function outputs absolute value and phase in the two items. The coherence function returns the same value in the first item as abs2, but assumes that the input is a complex coherency and calculates the probability α of the resulting coherence Z as $\alpha = (1-Z)^{L-1}$ (cf. Rosenberg et al. 1989), where L is the number of independent sections from which the coherency was formed by averaging and is assumed to be $2 \cdot nrofaverages$, which is correct if the **fftspect** overlap parameter was 1. The norm2 function normalizes the complex values.
The functions working on three or four statistical items are intended to be used on data as created by either the **average** or the **subtract** method with the options -t and -u or similarly by the **ascaverage** command (see page 48). They neither use nor modify the first (average) item. The second and third items, together with the 'nrofaverages' parameter, are used by 'ttest' to calculate t and two-tailed p values for the t-test against zero, and by 'stddev' to calculate the empirical standard deviation of the original ensemble and the standard error of the average (with $N-1$ degrees of freedom). The nrofaverages parameter may be given on a by-point basis in a fourth item as output by the **average -M -t -u** method. In each case, the two calculated values replace the second and third item in the output.

Arguments: function_name
Available functions are: log, log10, logdB, exp, exp10, expdB, atanh, sqrt, square, neg, abs, inv, ceil, floor, rint
Functions using two successive items as a complex value: abs2, square2, phase, coherence
Functions using three successive items as average, sum and sum of squares: ttest and stddev

Options:

-i nr_of_item: act only on this item number ($\geq$0)

-n channelnames: Act only on the given subset of channels.

calc_binomial_items: transform method adding probability items calculated from +/− counts. Leaveright must be 2. The two rightmost items of each data point are interpreted as numbers $n_+$and $n_-$of incidences of a binary decision (e.g. , spectral power value was higher/lower than baseline). The old items are preserved and three items added: the binomial probability p, the probabilistic logarithmic change measure $-\log_{10} p \cdot \text{sign\_of\_change}$ and the relative gain $G_r = (n_+ - n_-)/(n_+ + n_-)$ (cf. section MAIN:3.4). The latter value has the advantage of being comparable between experiments with different numbers of trials.

Arguments: NONE

change_axes: changes (shifts and scales) the 'x' and 'z' axes and sets the axis labels if told so. A '*' as label will leave the label unchanged. If no x axis exists, a new one is built from the point numbers $(0 \ldots N-1)$. The offset is added first (which makes it easy to control the final zero point). Since changing the 'y' axis would mean to perform an operation on the actual data values (possibly including multiple items), regular transform methods like scale_by should be employed for that purpose. For example, change_axes can be used to change the axis units or to shift the latency values obtained for spectral coefficients (fftspect), which depend on definition.

Arguments: x_offset x_factor x_newlabel z_offset z_factor z_newlabel

collapse_channels: transform method to create a number of output channels by performing some operation across subsets of existing channels. After each 'channelnames' channel list of the usual format (section 4.5, page 10), the name of the channel to hold their collapsed value follows after a colon. If no arguments are given, this is equivalent to the single argument 'ALL:collapsed', i.e. all channels are collapsed pointwise and a single output channel named 'collapsed' results. If averaging is selected (default), then the Leaveright items are still only summed instead of averaged across the selected channels.

Arguments: [channelnames1:name1 [channelnames2:name2 ...]]

Options:

-{asMhl}: Collapse channels by averaging (-a), summation (-s) or by choosing the highest (-h) or the lowest (-l) value within each subset of channels.

convolve: Method heavily based upon **sliding_average**, used to build simple detectors of a given wave shape. **Convolve** reads waveform(s) from convolve_file; the length of the resulting filter is therefore given by the number of points in that file. Then, a pointwise multiplication is done, summed across the filter (corresponding to a scalar product) and then divided by the number of filter points. The number of points by which the filter window is shifted between steps by sliding_step. The output x value corresponds to the middle of the window. At the data boundaries, the window size reduces to half of sliding_size. The number of output points is equal to inpoints/sliding_step. sliding_step may be fractional and may also be entered as time value by appending time units as usual (4.4). There may be either only one channel in convolve_file or as many channels as in the current epoch; in case of one channel, the same wave form is used for all input channels, otherwise each input channel is convolved with the corresponding convolve_file channel. Note that x-axis data, if available, is not convolved but a simple sliding window average is performed.

Arguments: convolve_file sliding_step

Options:

-f fromepoch: Start with epoch number fromepoch ($\geq$1)

-e: Advance the convolve_file epoch for each input epoch processed

correlate: This method implements linear operations on incoming time courses, treating the data for each channel as a vector in a space with as many dimensions as there are time points. The **project** method does the same for maps. The general procedure consists of two parts: A scalar multiplication between each data vector (time course) and each of a number of projection time courses read from an asc file (correlate_file), and subsequently, if requested, a reconstruction of time courses of the original length using the coefficients obtained in the first step. The time courses are read from the specified channels of correlate_file, possibly from multiple epochs (by default only from the first epoch). Thus, the number of correlation time courses, and also the number of obtained coefficients, is (correlate_file channels·epochs). The number of points in the correlate_file must equal that in the current epoch.
The operation carried out by default is a scalar multiplication between the input time courses I and the normalized correlate_file time courses $N_i = P_i/|P_i|$: $O_i = I \cdot N_i$. The output coefficients O_i are stored in successive output points. In subspace mode (option -s), the reconstruction step mentioned above takes place. The projection time courses are summed up weighted with the coefficients from the first step, yielding an output with the same number of points as the input had. Similarly, a subtraction of this projection from the raw data can be done (which is equivalent to the projection onto the orthogonal space of the subspace) to suppress components with a known time course.

Arguments: correlate_file channelnames

Options:

-s: Subspace mode: Incoming time courses are projected onto the correlate_file subspace. This means that the output contains as many time points as the input and is formed as $\sum_i (I \cdot N_i) N_i$.

-S: Subtract-Subspace mode: The projected time courses formed as above are subtracted from the input time courses I, effectively implementing a projection onto the orthogonal space of the P_i.

-m: Multiply mode. This is essentially just a matrix multiplication with the same result as the subspace option above, but here the projection step is skipped and the current epoch assumed to contain the weights O_i, just as after a previous call to **project**.

-c: Correlation. Each correlate_file time course is demeaned first. If the input time course are demeaned (using **add negpointmean**) and normalized (using **scale_by invpointnorm**), the output is the correlation coefficient between I and P_i.

-n: Do NOT normalize the time courses P_i.

-o: Orthogonalize the time courses P_i first. Recommended for the Subspace modes!

-D vectorfile: Dump correlate_file vectors, after preprocessing, to an ASCII (matlab) file. Preprocessing includes demeaning, orthogonalization and normalization (in this order!).

-f fromepoch: Start reading maps from correlate_file at epoch number fromepoch (≥1)

-e epochs: Number of epochs to read from the correlate_file (default: 1)

-i nr_of_item: Read P_i from this item number in correlate_file (≥0; default: 0)

demean_maps: Method to subtract the map mean from each incoming map, i.e. an average across all channels is subtracted from all channels for each time point. For EEG data, this corresponds to rereferencing to the 'common average reference'. This method is now kept mainly for compatibility with older versions, since the same effect can be achieved by either using **add negmean** or **rereference ALL**.

Arguments: NONE

detrend: Method to de-trend the given data set; This is done by subtracting a linear regression line from each time course. It is possible to specify any number of ranges to omit from the fit by start and end times relative to the epoch trigger, e.g. : `detrend 0ms 500ms`. This is useful if, for example, slow potentials are expected in part of the trace and should be excluded from the linear fit.
If the option -o is given, no actual detrending is done, but the constant value found at the given latency is subtracted from all points.
The option -0 is useful because the ability to specify arbitrary ranges to omit makes this method much more flexible than **baseline_subtract**.
Note that **detrend** does accept input epochs of varying lengths; however, x axis values (as well as the channel list) are only evaluated on the first epoch, so that unexpected behavior may result when processing epochs with varying size using the channel name and x axis options.

Arguments: omit_start1 omit_end1 omit_start2 ...

Options:

-0: Fit a 0-Order polynomial, i.e. demeaning is done instead of detrending.

-I: Interpolate, i.e. subtract a line connecting the first and the last point.

-o latency: Subtracts the constant value found at that latency from all points

-i nr_of_item: act only on this item number (≥0)

-n channelnames: Act only on the given subset of channels.

-x: The two values given for each range are the x axis values xstart and xend instead of point numbers relative to 'beforetrig'. Exactly as in the **trim** method, it is possible to enter a point offset to the position closest to the given value in the form xvalue+offset or xvalue-offset.

differentiate: transform method generating the derivative – defined by the difference between two successive data points. The first point of the result is not modified. This means that **integrate** will exactly revert the action of **differentiate**.

Arguments: NONE

Options:

-e: Epoch mode: Start anew for each epoch. Normally the last value of the previous epoch is carried over to yield a continuous differentiation.

-i nr_of_item: act only on this item number (≥0)

-n channelnames: Act only on the given subset of channels.

dip_fit: Method to calculate the best fitting dipole

Arguments: time

echo: Method to echo a string to the trace stream or a file. This will be mostly used to indicate progress within a script. '\n' indicates a newline, '\t' a tab.

Arguments: String to output

Options:

-F Output file: Output to this file.

export_point: transform method printing a 'point' - the channel positions and values at the specified point - to an ARRAY_DUMP file

Arguments: point_number outfilename

Options:

-{0123}: Chooses number of channel coordinates to display (default: 3)

-a: Opens the output file for append instead of overwriting it

-c: Closes and reopens the output file for each epoch. This is useful if an external program processes the newly created file for each epoch, cf. **run**

-m: Output MatLab format instead of ARRAY_DUMP ASCII

-x: The 'point_number' argument specifies an x axis value belonging to the point to export. Actually, the point with the closest x value is exported.

extract_item: transform method to extract items from tuple data. The output epoch will consist of the given items of the original epoch in the order of the argument list. It is possible to have multiple copies of an input item in the output. In this way, it is possible to duplicate a signal and apply different operations to the two copies, for example: **extract_item** 0 0 followed by **recode** -i 0 -Inf 0 0 0 and **recode** -i 1 0 Inf 0 0 splits the signal up in all positive values in the first and all negative values in the second item.

Arguments: item_number1 item_number2 ...: Item numbers starting at 0 (e.g. : Re=0, Im=1)

fftfilter: transform method to filter the signals by performing a Fourier transform, suppressing specified ranges of the Fourier coefficients and transforming the signals back into the time domain. Since a filter should usually not be infinitely steep in the frequency domain, the frequency ranges to suppress are not just specified by their starting and ending frequencies but by two values for each end of the range that tell at which frequency the change in the suppression coefficient should start to 'fade in' and at which frequency the target suppression should be reached. The interpolation was chosen to be linear. An arbitrary number of suppression blocks may be given; Each block contains four ascending numbers between 0.0 and 1.0 that specify frequencies relative to half the sampling frequency: start, zerostart, zeroend, end. Frequencies can also be entered in Hz by appending the string 'Hz', e.g. 48.2Hz. The transmission x between start and zerostart is 0 by default but can be specified. Between start and zerostart the transmission varies linearly from 1 to x and between zeroend and end from x to 1. If the suppression should start at 0Hz (i.e., the first block represents a high pass), zerostart and zeroend can both be set to 0; similarly, the last block can extend to half the sampling frequency (i.e., represent a low pass) by setting both zeroend and end to 1. These are usually the only cases in which an infinitely steep ramp should be chosen. In all other cases, it is desirable to let the ramps span at least two Fourier coefficients, which means that the difference between the start and end frequencies should at least be $2/T$ Hz, where T is the epoch length in seconds. For examples, see figure 2.

Arguments: block1 [block2 ...]

Options:

-i nr_of_item: act only on this item number ($\geq$0)

-n channelnames: Act only on the given subset of channels.

-V: Verbose output of filtering diagnostics

-x where x is a floating-point number between 0 and 1: May occur at the start of any block. If specified, the frequency range between zerostart and zeroend of this block is multiplied by x instead of by zero.

fftspect: Method to analyze the data in the frequency domain. The 'epoch' resulting from this operation is special in that it has both latency and frequency axes. When **writeasc** writes an epoch of type FREQ_DATA, it forms multiple output epochs containing spectra and assigns *latencies* to the individual epochs as 'z values'. (Note that the same is temporarily done if **posplot** needs to display FREQ_DATA.) These latencies correspond to the ***end*** of the analysis window used to evaluate the spectra. If the latency measures of the final results should refer to the middle of the analysis window instead, the **change_axes** method can be used in another **avg_q** run to shift the z values or, after a run of the **swap_xz** program, the x

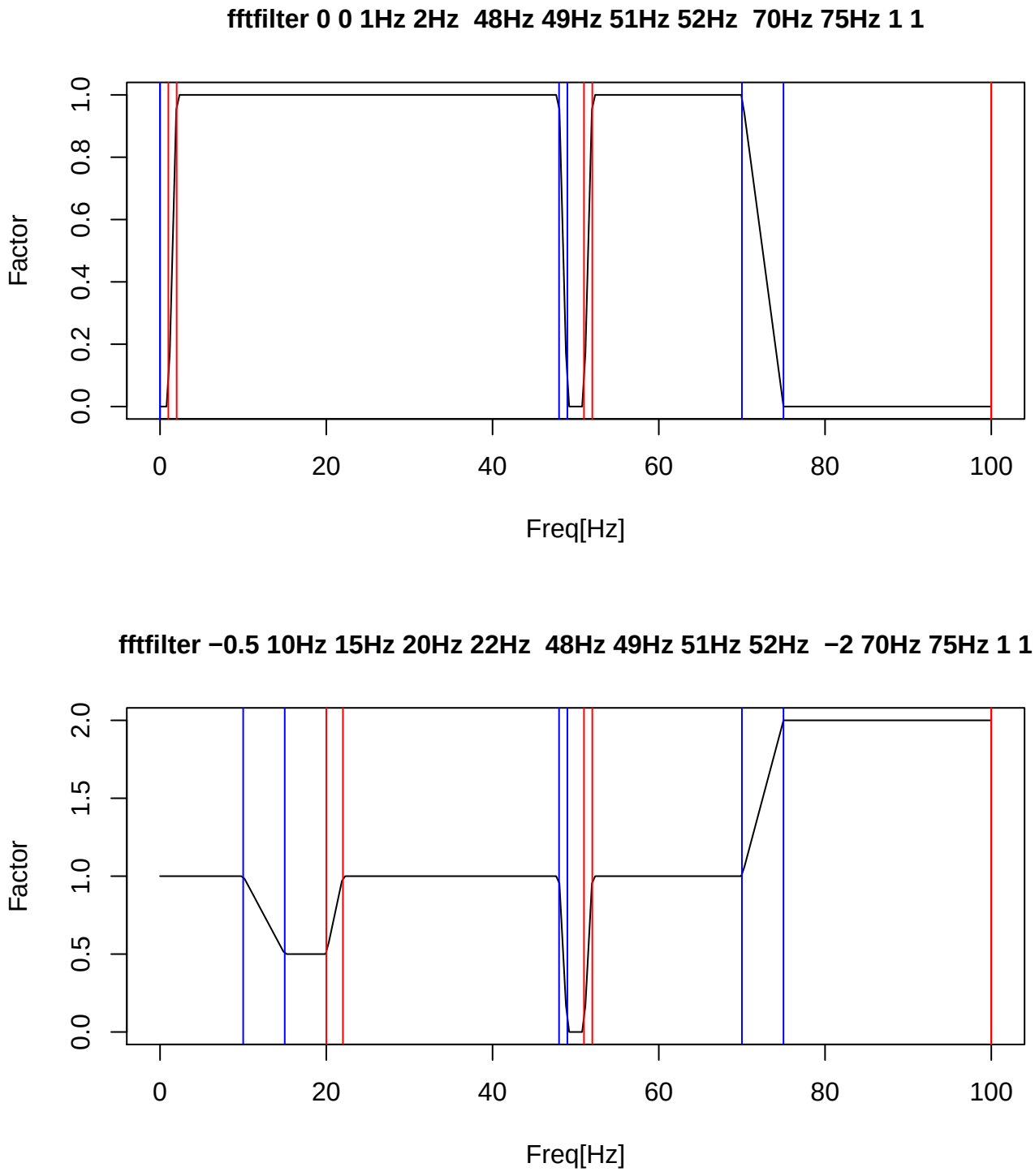

Figure 2: fftfilter examples. Sampling frequency is 200Hz, Nyquist frequency 100Hz (=1). Start and end of blocks are marked with blue and red vertical lines, respectively. The second example shows how factors can be arbitrarily set for each block by using a number prepended with "-" (-x). If omitted, it defaults to 0.0 (full suppression) for each block. Note that these plots are not schematic; they show fftfilter -V diagnostic output for the given configurations.

values back by half the window width.
Spectral analysis is performed after demeaning, detrending and applying a Welch taper to each single FFT window.

Arguments: windowsize nrofshifts overlaps

Options:

-c[{c,s}] refchannel: Use complex spectra phase aligned to refchannel.
c: Output coherence only (the above divided by the channel amplitude)
s: Output the cross-spectrum (don't divide by either amplitude)
refchannel="ALL": Build complete channel x channel matrix

-p padto: Zero-pad the window to padto points before FFT

The 'windowsize' argument defines the length of the window from which a single spectral estimate is obtained. 'nrofshifts' requests a number of linearly spaced window shifts (latencies) within the epoch and 'overlaps'·2 is the number of single FFT windows to use for each estimate. The FFT windows overlap by 50 %. Since the number of (positive) frequencies $nfreq$ obtained from an FFT is half the number of input points, the formula to calculate the analysis window size is ('overlaps'·2 + 1) · $nfreq$. For example, a 'windowsize' of 100 points at 'overlaps'= 1 will result in a warning message that the window size was shortened to 96 points= 3 · 32 points, because the FFT only operates on data sizes that are a power of 2 (in this case, 64).
If the option -p is given, then the 'padto' parameter defines the analysis window size as above and the 'windowsize' parameter (less than 'padto') defines how many points of actual data are used. The rest of the analysis window is filled (padded) with zeroes.
If -c[{c,s}] ALL is given, 'nrofshifts' must be 1, because the dimension of the output table normally containing spectra for multiple window shifts (latencies) now contains the spectra for multiple reference channels.

icadecomp This is the **avg_q** incorporation of the Independent Component Analysis method developed at the Salk Institute, San Diego. See http://www.cnl.salk.edu/~tewon/ica_cnl.html for an overview and http://www.cnl.salk.edu/~enghoff/ for the 'standalone' code, implemented by Sigurd Enghoff, which was actually incorporated. We output the resulting component maps in adjacent 'points'. The activations can be calculated using the 'project' method. The argument n_components is the number of PCA components to which to reduce the data before the analysis.

Arguments: n_components

Options:

-e n: Perform an 'extended ICA' with PDF estimation every n blocks (default: 1)

import_point: transform method loading a 'point' - the channel positions and values at the specified point - from an ARRAY_DUMP file

Arguments: point_number infilename

Options:

-{0123}: Chooses number of channel coordinates to import (default: 3)

-c: Closes and reopens the input file for each epoch

-x: The 'point_number' argument specifies an x axis value belonging to the point to import. Actually, the point with the closest x value is imported.

integrate: Transform method generating the integral – defined by the cumulative sum of the data points

Arguments: NONE

Options:

-e: Epoch mode: Start anew for each epoch. Normally the last value of the previous epoch is carried over to yield a continuous integration.

-i nr_of_item: act only on this item number (≥0)

-n channelnames: Act only on the given subset of channels.

invert: Transform method to calculate the inverse or, if the data matrix is not of maximum rank, the pseudoinverse (in the least squares sense) of the data. The result has the same number of channels and points as the input. The inverse is done in such a way that a **project -n -C -p 0 orig.asc 0**, with orig.asc being the original data, will result in the identity matrix. This is, for example, the operation necessary to transform the weights (or spatial filter) matrix derived by **icadecomp** into the corresponding set of topographies (maps).

Arguments: NONE

Options:

-i nr_of_item: act only on this item number ($\geq$0)

laplacian: Transform method to perform the Laplacian operation on the incoming epochs. The algorithm is roughly constructed after Le et al. (1994): The electrode set is triangulated and those points around which a closed path can be constructed are used for a local planar surface. The Taylor- expansion parameters (the planar derivatives up to second order) are then estimated by solving a set of linear equations by SVD backsubstitution. The sum of the second-order derivatives is output as the Laplacian estimation. If P denotes the potential, the result is an estimation for $d^2P/dx^2 + d^2P/dy^2$ at each electrode. For electrodes at the border, no Laplacian can be estimated; these electrodes will be missing from the output. Note that bad electrodes (containing noise or deviating in sensitivity from the surrounding) will have large impact on all surrounding electrodes, so that it is better to remove such channels first.

Arguments: NONE

Options:

-{NRD}: Choose the mode of operation. **-N** (default) selects actual laplacian as described above; **-D** does the laplacian but writes all derivatives as single output items for test purposes (dP/dx, dP/dy, d^2P/dx^2, d^2P/dy^2). Note that the local coordinate system is different for each channel and thus 'x' and 'y' will usually denote different directions for different channels. **-R**, finally, selects local average reference operation: Each channel is rereferenced to the mean of the surrounding channels. This operation does not depend upon the exact distances between the channels and can therefore also be applied if channel positions are inexact or if only a 2-D channel layout is available.

-M matfile: Output the transformation matrix in MatLab format. matfile can be 'stdout' to write the matrix to standard output. There are as many columns in this matrix as original channels and as many rows as output channels. This matrix can be used in general linear processing programs (e.g. with the **project** method or with the NeuroScan 'linear derivation' facility) to yield the same result as the **laplacian** method. This feature is intended for testing and to utilize the matrix for different programs, but the output format must generally be adapted (using a text editor, for example) for other programs than MatLab. At the start of the file, the names of the input and output channels are listed to help with such editing.
Note that the option -n described below is correctly processed (i.e. the named channels are excluded from the triangulation but added as extra rows containing single unity columns at the bottom of the matrix), but the option -D has no effect on the matrix obtained using -M. The normal processing is not affected by this option.

-n channelnames: Act only on the given subset of channels. The channels not mentioned in this list are just copied from the input to the output. One example would be `laplacian -n !EMG,vEOG,hEOG` to perform the Laplacian derivation on all but the mentioned bipolar channels.

link_order: Transform method to reorder the linked datasets in memory. Since transform methods always operate on the first of a linked list of datasets in memory (cf. **push**, **pop**, **append -l** etc.), it is useful to be able to 'move a dataset to the front' to apply some transformation to it. When only one dataset number is given (starting with 1), **link_order** makes this the first dataset, keeping the order of the other datasets. As a generalization, it is possible to define any wanted order of datasets; those data sets not mentioned will appear at the end in their original order. Since **link_order** never alters the dataset count, any dataset may be mentioned only once.
A second use of **link_order** is to make the datasets appear in any desired order within **posplot**.

Arguments: dataset_no_1 [dataset_no_2 ...]

normalize_channelbox 'transform method' to rotate the channel positions (for systems covering only a smaller part of the scalp) into a normal position suitable for displaying: the main plane of the channel positions becomes to the x-y plane, with the topmost channel highest in y and the frontmost (relative to the subject) highest in x, so that the head is viewed from the right. This assumes the BTi coordinate system: x is posterior → anterior, y is right → left. If the epoch descriptor starts with 'Roma ', then the CNR/IESS coordinate system is automatically used: x is left → right, y is posterior → anterior.

Arguments: NONE

Options:

-d: The frontmost channel will be lowest in x if the y coordinate of the center of the xyz-box containing the channel arrangement is positive, i.e. if the dewar was on the left side of the head. This conforms to the usual convention to always view the channel arrangement from 'outside'.

orthogonalize Transform method to successively orthogonalize the current data vectors, leaving the first vector unmodified. By default, time courses are orthogonalized.

Arguments: NONE

Options:

-m: Orthogonalize maps instead of time courses.

posplot: Method to display data traces at positions corresponding to the 3-dimensional sensor locations. The sensor layout can be viewed from any point in space, and the positions themselves can be manipulated within a special mode. Another display mode shows the channels layed out vertically.
This interactive data browser can also be run by the stand-alone program **do_posplot**. A summary of the keyboard and mouse commands that can be used within posplot can be found in section 6 on page 43. Since **posplot** can be exited by either pressing 'q' or 'Q' and the latter marks the current epoch as rejected, it could also be called an interactive rejection method ;-)
posplot is able to display multiple epochs simultaneously. This feature is used automatically if time-frequency distributions are plotted. To collect single epochs so that they can be displayed together, the epochs can either be saved to an asc file to display using **do_posplot**, or they can be collected in memory using the **append -l** collect method and displayed by placing **posplot** in the post-processing queue.
A note about event replay files: These are simple text files that record a number of key presses within **posplot** (via the 'r' key) and can be called as 'macros' via the 'R' key or using the options below. They can also easily be manipulated using a text editor: The format ignores newlines, so that commands can be grouped. A 'return' key press is coded as '\n', a backslash as '\\' and mouse clicks as '\L', '\M' and '\R' for left, middle and right plus the x and y coordinates.

Arguments: NONE

Options:

-q: Quit at the point where otherwise user interaction would begin. This means that plotting is performed as usual but the **avg_q** script continues directly afterwards. For this to be useful, e.g. so that the processed epochs can be seen 'flying by', this will require the initial view to be set using a few commands from a replay file.

-R: Run the event replay file named 'posplot.rec'.

-r eventfile: Run the named event replay file.

project: This method implements linear operations on incoming maps, treating the data for each time point as a vector in a space with as many dimensions as there are channels. The **correlate** method does the same for time courses. The general procedure consists of two parts: A scalar multiplication between each data vector (map)I and each of a number of projection maps P_i read from an asc file (project_file), and subsequently, if requested, a reconstruction of maps of the original dimensionality using the coefficients obtained in the first step. The maps read from project_file start at a specified data point (nr_of_point) and continue through successive points and then epochs as requested. Thus, the number of projection maps, and also the number of obtained coefficients, is (epochs·points). The number of channels in the

project_file should equal that in the current epoch.
The operation carried out by default is a scalar multiplication between the input maps I and the normalized project_file maps $N_i = P_i/|P_i|$: $O_i = I \cdot N_i$. By default the output coefficients O_i are stored in successive items and the output will be reduced to a single channel named '1'. If there are multiple items in the input already, the output contains one block of consecutive items for each i.
In subspace mode (option -s), the reconstruction step mentioned above takes place. The projection maps are summed, weighted with the coefficients in the first step, yielding an output with the same number of channels as the input had. Even more often used than the projection onto a known subspace is the subtraction of this projection from the raw data (which is equivalent to the projection onto the orthogonal space of the known subspace unless option -N is given) to eliminate components with a known topography such as the EOG. For EOG correction, -N can be used with subtract-subspace in order to let only a limited number of channels determine the coefficient but yet subtract the the whole template weighted with the coefficient (this correction method is equivalent to the one implemented by NeuroScan if only the EOG channel is in the -N channel list). Thus, by itself it does NOT mean that only the mentioned channels are modified. To protect channels like the EMG from modification, their coefficients in the project_file can be set to zero in addition to excluding them from the -N list, or equivalently the -z option can be used. However, the built-in orthogonalization step will always work on all channels, so that additional steps may be required if orthogonalization is necessary.
project can also be used for the calculation of new channels as arbitrary linear combinations of input channels. This can be achieved by switching the P_i normalization off (option -n) and requesting that the scalar product is output as one *channel* per P_i instead of one *item* per P_i (option -C).

Arguments: project_file nr_of_point

Options:

-s: Subspace mode: Incoming maps are projected onto the project_file subspace. This means that the output contains as many channels as the input and is formed as $\sum_i (I \cdot N_i) N_i$.

-S: Subtract-Subspace mode: The projected maps formed as above are subtracted from the input maps I, effectively implementing a projection onto the orthogonal space of the P_i.

-m: Multiply mode. This is essentially just a matrix multiplication with the same result as the subspace option above, but here the projection step is skipped and the current epoch assumed to contain the weights O_i, just as after a previous call to **project**. This is useful with methods like ICA, where the spatial filter used to derive the weights is different from the map of the estimated generator. **project -m** can be used to construct the multichannel data accounted for by any combination of the ICA sources.

-c: Correlation. Each project_file map is demeaned first. If the input maps are demeaned (using **demean_maps**) and normalized (using **scale_by invnorm**), the output is the correlation coefficient between I and P_i.

-C: Different channels are used for scalars instead of different items. This option has no effect in subspace mode (-s or -S). The resulting channels are numbered starting with 1 and arranged on a grid.

-n: Do NOT normalize the maps P_i.

-o: Orthogonalize the maps P_i first. Recommended for the Subspace modes!

-z: Set project_file channels not in the -N list to zero. In conjunction with Subtract-Subspace mode, this protects these input channels from modification if -c and -o are NOT used.

-N channelnames: Restrict scalar product and normalization to these channels. This is mainly of interest for Subtract-Subspace applications in which the coefficients for the maps to subtract should come from only a few channels, e.g. the vertical EOG channels to subtract the vertical EOG maps.

-D vectorfile: Dump project_file vectors, after preprocessing, to an ASCII (matlab) file. Preprocessing includes channel zeroing, demeaning, orthogonalization and normalization (in this order!).

-f fromepoch: Start reading maps from project_file at epoch number fromepoch ($\geq$1)

-e epochs: Number of vectors to read from the project_file (default: 1)

-p points: Number of vectors to read from each epoch (default: 1)

-i nr_of_item: Read P_i from this item number in project_file ($\geq$0; default: 0)

push: 'Transform' method to duplicate the current data set and store it as the second within the current linked list of data sets; cf. the 'linking' in memory which can be done by **append -l**, leading to multiple data sets shown by **posplot**. The main use of **push** and **pop** is to save an epoch at some point in time, apply one set of transformations, possibly saving the result or using it to apply rejection criteria, and then restore the epoch and apply some other transformations. This can be useful if reading or transforming the data up to the current stage is time consuming; another important use is in conjunction with Reject_Methods: The rejection criterion can be applied to the spectra of certain channels, for example, and then the original data set can be restored if the criterion is met, restricting further analysis to those epochs meeting the frequency-domain criterion.

Arguments: NONE

pop: 'Transform' method to free the current data set and replace it by the second one, such as previously stored by the last **push** method.

Arguments: NONE

query: 'Transform' method to query epoch properties, writing the result to a text file or stream. var_name can be any of the following words:
sfreq nr_of_points nr_of_channels itemsize leaveright length_of_output_region file_start_point points_in_file beforetrig aftertrig nroffreq nrofaverages accepted_epochs rejected_epochs failed_assertions condition z_label z_value comment datetime xchannelname xdata channelnames channelpositions triggers triggers_for_trigfile triggers_for_trigfile_s triggers_for_trigfile_ms filetriggers_for_trigfile filetriggers_for_trigfile_s filetriggers_for_trigfile_ms CWD
These are the names of various bits of information internally available for each epoch. The value is written to Output_file[5] (which can be 'stdout' as a special case) followed by a carriage-return character. If the option -N is given, the value will be preceded by 'var_name=', which makes it easier to write multiple values to a file and scan that file afterwards. In this way, **avg_q** can be used to determine, for example, how many channels are in a file or with was the sampling frequency without the need for specialized tools for the specific (often binary) file format.
'file_start_point' is set to the starting point of the current epoch in the current file by most Get_Epoch_Methods reading from continuous input.
'points_in_file' gives the total number of points (samples) in the input file. This is a special informational value provided by some Get_Epoch_Methods, in particular those capable of continuous reading. No effort is made to calculate the total number of points for epoched data formats, and the value will be set to 0.
'datetime' outputs the date and time of the recording in the format mm/dd/yyyy,hh:mm:ss as specified within the comment in the same format (where both 2- and 4-digit forms of the year are recognized).
'channelpositions' outputs a line for each channel containing the space-separated channel name and x, y and z position values, corresponding to the format read by **set_channelposition**[6].
'triggers' outputs the list of internally stored triggers for the current epoch - trigger positions are relative to the start of the epoch. The first value output is the point offset of the epoch start within the file, where supplied by the Get_Epoch_Method. If an x axis is defined, x values are reported instead of point offsets.
'triggers_for_trigfile' aims at writing the epoch triggers as absolute file triggers suitable as a trigger file. Note that for the correct positions to be calculated for a given output file, the query method must be at a point in the processing queue where it sees all epochs in the shape as they are written to the file. The variants of this variable with "_s" and "_ms" at the end store the trigger positions in seconds and milliseconds, respectively.
'filetriggers_for_trigfile' is special in that it does not access epoch-specific data but rather looks at the trigger list provided by many Get_Epoch_Methods. Therefore querying it will make sense for a single, possibly very small continusous data epoch (1 sampling point), while forcing the method to read the trigger list using -T (trigtransfer). It was added in order to be able to leverage the functionality of the Get_Epoch_Methods, which necessarily "know" how to read the trigger or event lists of each respective format, to make these lists available from within avg_q. The variants of this variable with "_s" and "_ms" at the end store the trigger positions in seconds amd milliseconds, respectively.

[5] Note that until April 2007, values were *appended* to Output_file if it already existed; for consistency with the rest of avg_q, an existing Output_file is now overwritten.
[6] **query channelpositions** replaced the special method **write_channelpositions** in 02/2000.

'CWD', as you might expect, outputs the current working directory, as this can be changed using '**run** chdir'

Arguments: var_name Output_file

Options:

-a: Append to Output_file if it exists (Default: Overwrite)

-N: Output as 'name=value' pair

-t: Write a TAB after the entry instead of a NEWLINE. This is useful to group multiple data items on an output line. Note that also for the variables containing multiple elements (like channelnames, xdata), TABs are written instead of newlines.

raw_fft: Transform method to perform a raw Fourier transform on the input data. This is a method for experimenting with raw complex spectra and not really meant as a production analysis tool; the methods **fftfilter** and **fftspect**, for example, use Fourier transforms as well but perform additional operations to suit their special functions. It is advisable to do a **detrend** before **raw_fft**. The result is complex-valued if the input was real time-domain data. If applied on compatible complex-valued spectra, the inverse operation is performed. This means that calling **raw_fft** twice will yield the original data, padded to a power of two data points. It is also possible to perform operations on the spectrum before reconverting it; note, however, that the sampling frequency value will be wrong after the size of the spectrum is changed, so that it must be **set** correctly before the inverse transform.
Notes on phase calculations: Since $z(\varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$, phase angles (as calculated by **calc absandphase** and the **posplot** phase display) are measured relative to a *cosine*. Positive phase shift is counted toward the right, i.e. a higher latency corresponds to a larger phase as would be expected. Therefore, a normal sine wave has a phase shift of $+90°$ (or correspondingly, $\pi/2$).

Arguments: NONE

recode: Transform method to linearly map input to output value ranges. Each 'block' of arguments defines an interval of data values [fromstart, fromend] and an interval [tostart, toend]to which data values falling into the first interval are linearly mapped. Each 'block' consists of the four values fromstart, fromend, tostart and toend separated by white space. While fromend must be greater than or equal to fromstart, tostart and toend can also be descending numbers. In any case, a value of fromstart is mapped to tostart, fromend is mapped to toend and values in-between to according values in-between. If either fromstart = fromend or tostart = toend, values are mapped to tostart. The special value 'Inf' can be used to denote infinity, and the special value 'NaN' can be used to denote the undefined value. Since terms involving NaN obviously cannot be interpolated in any way, from-start and from-end must both be NaN and to-start and to-end identical when recoding NaN, or to-start and to-end both be NaN if recoding into NaN.
For each incoming value, the blocks are checked in the order in which they appear in the arguments, and only the first match is used.
Note that this can be used to achieve transformations excluding an interval boundary:
recode 0 0 0 0 0 Inf 1 1
will map only positive values to 1 and leave 0 at 0.

Arguments: block1 [block2 ...]

Options:

-i nr_of_item: act only on this item number ($\geq$0)

-n channelnames: Act only on the given subset of channels.

remove_channel: Transform method to remove a single channel or ranges of channels from the input data. Channel numbers start with 1 and refer to the internal channel order. Multiple channel ranges can be given as ascending pairs. Using the -k (keep) option, it is possible to remove all but the channels with the given names. In this case, no channel_number arguments may follow, and the named channels are copied in the order in which they appear in the channelnames list. In this way, it is possible to change the physical order of channels in memory and even to replicate channels (if a single name appears in the list more than once). For the format of 'channelnames' lists, see section 4.5 on page 10.

Arguments: channel_number_start1 [channel_number_end1 channel_number_start2 ...]

Options:

-k channelnames: Keep channels by name (remove all but these channels).

-n channelnames: Remove channels by name.

rereference: Transform method to subtract the mean of a number of channels from each incoming map. For EEG data, this is known as rereferencing to the given (averaged) reference. Note that due to the conventions for 'channelnames' lists, '**rereference** ALL' results in the same transformation that is performed by the **demean_maps** method.

Arguments: ref_channelnames

Options:

-E: Exclude all non-reference channels from the operation. This is often used for 'common average reference' calculations where bipolar channels are excluded from the whole rereferencing process.

-e channelnames: Exclude these channels from the operation. They may still be included in forming the reference average but are not modified.

run: 'Transform method' to run an external program. The argument is simply passed to the C *system()* call. The only exception is made if program_path is 'chdir', in which case **avg_q** will try to change the directory to the path specified in the argument. While **avg_q** does not try to determine whether the *system()* call was successful, it is an error if 'chdir' fails (e.g. because the target directory does not exist).

Arguments: program_path args...

scale_by: Method to scale (multiply) the incoming epochs either by a constant factor (any value other than '1.0', which is a deprecated former alias to 'invnorm'), by the inverse of the map vector length (normalization, command='invnorm'), by the inverse of the square map vector length (i.e. of the sum of squares; command='invsquarenorm'), by the inverse of the map sum (command='invsum'), by the inverse of the map maximum ('invmax'), by the inverse of the map maximum absolute value ('invmaxabs'), by arbitrary quantile ('invquantile', with factor set to the desired quantile $[0, 1]$), by the x axis value ('xdata') or by its inverse ('invxdata'). Variations that work on channels rather than on maps are called 'invpointnorm', 'invpointsquarenorm', 'invpointsum', 'invpointmax', 'invpointmaxabs' and 'invpointquantile'. The special constants 'pi' and 'invpi' are also allowed, and the following commands involving dataset properties: 'sfreq', 'invsfreq', 'nr_of_points', 'invnr_of_points', 'nr_of_channels', 'invnr_of_channels', 'nrofaverages', 'invnrofaverages', 'sqrtnrofaverages', 'invsqrtnrofaverages'.
Note that nr_of_points is the number of frequencies for frequency (spectral) data.
A hint for one of the several uses: Since **scale_by invpointmaxabs** ensures that the data in each channel does not exceed the interval $[-1, +1]$, it can be used to maximally exploit the dynamic range of some integer output format, provided that a second **scale_by** operation multiplies the data with the maximum allowable value for that format, e.g. **scale_by 32767** for 16-Bit signed integers.

Arguments: [command] [factor]

Options:

-i nr_of_item: act only on this item number (≥ 0)

-n channelnames: Act only on the given subset of channels. This does not (yet?) work with those special operations that are targeted on maps.

set: transform method to set epoch properties just as **query** queries them. var_name can be any of the following words:
sfreq sfreq_from_xdata leaveright beforetrig aftertrig beforetrig_at_xvalue file_start_point nrofaverages accepted_epochs rejected_epochs failed_assertions condition xchannelname xdata posdata z_label z_value FREQ_DATA basefreq trigger triggers_from_trigfile
'sfreq' sets the sampling frequency used to calculate back and forth between times and sampling point numbers.
'sfreq_from_xdata' calculates the sampling frequency from the first two values of xdata ("value" is ignored).

'leaveright'[7] is the number of items of each point that are to be neglected during calculations.
'beforetrig' and 'aftertrig' are the number of points before and after the zero point (beforetrig points are used for the baseline) and setting any of these two also modifies the second so that their sum corresponds to the total number of points in the epoch. As a special case, beforetrig_at_xvalue sets beforetrig to the index of the point (≥0) with x value closest to this value; therefore, the points before this value will form the baseline (x axis data must be available for this operation).
'file_start_point' sets the absolute point number at which the epoch started in an input file, normally set by most Get_Epoch_Methods and used for absolute xdata computation (see below).
'nrofaverages' sets the number of averages for the current epoch, which can be used, for example, for weighted averaging.
'accepted_epochs' and rejected_epochs' are the respective epoch counters.
'failed_assertions' is the counter increased by the **assert** method whenever the assert condition is not met.
'condition is the condition code of the epoch (if set by the Get_Epoch_Method).
'xchannelname' sets the name of the x axis (e.g. 'Lat[ms]').
'xdata' either deletes the current explicit x axis information (value==0)[8] or creates an explicit x axis information anew. For time domain data, value can be one of the known units of time: us (=µs), ms, s, min, or h. For any other value, it falls back to use the standard heuristic. Normally this calculates the latency within the current epoch, using the variables beforetrig and sfreq. If 'abs_' is prepended (no space!) before the unit name, absolute time is computed using file_start_point, which is set to the starting point of the current epoch in the file by most Get_Epoch_Methods reading from continuous input.
'xdata_from_channel' copies the data of the named channel to xdata, allowing for example to plot channels as Lissajous figures.
'posdata' uses **avg_q**'s channel name and position default routine to create new channel positions (value==0, channels are arranged on a square 2-D grid), new channel names (value==1, simply enumerate channels starting with 1), a grid with given number of columns (value>1) or both (value<0, number of columns is abs(value)).
'z_label' is the label and 'z_value' the value which localizes the epoch on the 'z axis' as explained in section 4.1. z_label can be set to the NULL pointer (meaning 'no z axis information') by specifying 'NULL' as value. 'z_value' can be set to the current value of either 'nrofaverages', 'accepted_epochs', 'rejected_epochs' or 'condition' by specifying these names as value instead of a numeric constant.
'FREQ_DATA' will convert the data set to frequency-domain data if value!=0 and to time-domain data if value==0. The conversion to time-domain data will only work for single-shift data (see **fftspect**).
'basefreq' sets the value of the internal frequency resolution variable, which is used for x axis calculations in frequency domain data.
'trigger' will insert a trigger into the epoch's trigger memory. The value is of the form [x=]point[:code[:description]], where point is the offset in points, counted from the epoch start (can be given in time units as usual, 4.4) or, if preceded by x=, the x value of the point on which to set the trigger, and code is the trigger code. If the colon and code part is left out, code is set to 1. Value can also be DELETE, which will clear the epoch trigger memory. A trigger code of 0 will not actually be inserted into the trigger list but sets the 'file position' of the trigger list, intended to contain the offset of the current epoch relative to the input file.
'triggers_from_trigfile' accepts the name of a standard **avg_q** trigger file and adds epoch triggers according to that file. Note that the trigger positions are counted from the start of all points that the **set** method has seen. This behavior is symmetric with that of **query triggers_for_trigfile**.

Arguments: var_name value

set_channelposition: 'Transform' method to set the position values for selected channels or to set channel names and positions for all channels. Channel names and positions can also be read from the lines of an ASCII file as written by **query channelpositions** (tab delimited, no header, columns: name x y z). Multiple channels of the same name are set to the same position by default.
Builtin sets currently available are grid, PSG and EEG. "grid" arranges the channels on a rectangular grid as in other parts of avg_q. The other sets provide fixed mappings of channel names to positions and are useful with data formats which include channel names but lack channel position information, such as

[7] **set leaveright** superseded the special method **set_leaveright** in 10/1996.
[8] Note that after removing xdata with 'set xdata 0', **avg_q** will rebuild the x axis using default heuristics whenever it is needed.

from read_freiburg (set PSG) or read_brainvision (set EEG). Channels with unknown names are arranged horizontally outside the main set, making unknown channels visible to the user.

Arguments: *either* channelname1 posx posy posz [channelname2 ...]
or @pos_filename
or =builtin_set

Options:

-s: Set channel names and positions rather than identify channels by names. Setting starts with the first channel in the epoch and continues through as many channels as specified, up to the number of channels in the epoch. This option is not useful with builtin sets since the order of channels in those sets will usually not correspond to your data file.

set_comment: 'Transform' method to set or append to the data set comment string. '\n' indicates a newline, '\t' a tab.

Arguments: comment

Options:

{ -a -p }: Append or prepend to rather than replace the existing comment.

show_memuse: 'Transform' method to show memory usage and other process statistics for the analysis program (using the Unix program 'ps'). This is mainly used to decide whether the program handles memory correctly between processing steps.

Arguments: NONE

sliding_average: Sliding average or median method (block filter) for smoothing and resampling. The number of adjacent input points for each step is determined by sliding_size, and the number of points by which the averaging window is shifted between steps by sliding_step. The output x value corresponds to the middle of the window. At the data boundaries, the window size reduces to half of sliding_size. The number of output points is equal to inpoints/sliding_step. sliding_size must be an integer value, while sliding_step may be fractional. This means that any target sampling frequency can be obtained (only limited by the fact that the output epoch must contain an integer number of points...) and upsampling is possible as well as downsampling. Both values may also be entered as time values by appending the time units as usual (4.4). In this way, the target sampling interval can be entered independently of the input sampling frequency: e.g. if sliding_step is specified as '5ms', the output will have a sampling frequency of 200 Hz. Note that the **trim** method can also perform averaging or summation across adjacent data points, and there the point ranges to sum can be exactly and individually specified for each output point. Thus, **trim** can be used where an operation other than regular resampling is intended.

Arguments: sliding_size sliding_step

Options:

-M: Use sliding median instead of average.

subtract: Method to subtract a given asc-file epoch from all incoming epochs. Alternatively, the asc-file epoch may also be added to the current epoch, the mean or weighted mean of the two epochs can be calculated (setting the output nrofaverages to 2 or to the sum of the input nrofaverages, respectively), or the current epoch can be multiplied or divided by the asc-file epoch, all in the same point-, channel and itempart-wise manner. For scalar products, see the **project** method; However, the special case of complex multiplication and division is also supported by **subtract**. Note that for complex multiply and divide operations, the values from subtract_file are automatically conjugated to form the proper metric product $c_1 \cdot (c_2)^*$.
In files with multiple items and 'leaveright'>0, the 'leaveright' right-most items are actually only summed in mean or weighted mean mode, replicating the behavior of the **average** method. This is useful to combine averages containing the accumulated sums and sums of squares (for computing variance and t tests) as second and third item. In such averages, 'leaveright' is automatically set to 2.
With the option **-t** in subtraction mode, the sum and sum of squares are used to calculate t comparisons and two-tailed p values for the difference of the means of independent samples. This assumes that the

second and third items of both files contain the sum and the sum of squares of the samples, respectively, and that a nrofaverages property is specified in both files, as provided by **average -t -u**. Alternatively, the nrofaverages may be given on a point-by-point basis in a fourth item as output by the **average -M -t -u** method. Note that the correct number of averages to use for a t test between groups is the actual number of values averaged to obtain the given mean and variance measures, not the sum of the weights (number of included epochs) as both `ascaverage` and the **average** method output by default. Be sure to use the option -N with either of them to force them to output the number of cases if this is desired. **subtract -t** replaces the second and third items of the processed epoch with the t value for the difference test for the means of the two samples and the corresponding two-tailed p value. When computing t tests with 3-item data, the output nrofaverages is set to n1+n2-1 allowing correct results in subsequent degrees of freedom calculations (in tests assuming equal variance).
With the option **-u** in addition to **-t**, **subtract** will output sums and sums of squares in the second and third output items. The output nrofaverages setting explained above has the effect of enabling a subsequent `calc ttest` to compute the p and t values. This means that the output values contain variance and df information for the difference in a form that can be used for further pooling or comparison.

Arguments: subtract_file

Options:

{ -a -mean -wmean -s -m -d -cm -cd }: Select the basic pointwise action: Add, average, weighted average (using the respective nrofaverages as weight), subtract (default), multiply, divide, complex multiply or divide.

-f fromepoch: Start with epoch number fromepoch ($\geq$1)

-e: Advance the subtract_file epoch for each input epoch processed

-C,-P: Recycle channels (respectively points) of the subtract_file epoch when its number of channels (points) is smaller than that of the current epoch. Default behavior is to run over the minimum number of channels and points present in both epochs. The normal use of these options is to subtract the same value from each channel or point by using a subtract_file containing a single channel or point.

-t: Perform t-tests as explained above.

svdecomp: This method performs a singular value decomposition (SVD) of the channels$\times$points matrix, yielding a number of items corresponding to different component waveforms; within each item, the waveforms of all channels are equal, only the amplitude is different. The sum across all items is equal to the original data set if all components are used. This view on the SVD result is somewhat 'exploded', because a single representation of the waveform together with an amplitude for each channel would suffice, but useful, because the result has the same form as the input. As usual with SVDs, the components are sorted by the magnitude of the singular value and the waveforms as well as the topographies of different components are orthogonal.
The argument n_components is the number of SVD components (items) to output.

Arguments: n_components

Options:

-m: The components are the maps rather than the waveforms. If there are more points than channels, this results in more computation time, and the results are usually the same; there are also not really more components available, because the rang of the input matrix is fixed. However, this option may be tried for comparison, or to reduce the computational effort if the number of points is less than the number of channels.

-M: Return only the maps for later projection as 'points' of the output. The time-course projection can later be obtained by using the **project** method. This has the advantage that the SVD maps of one data set can be used to project other data sets. This is also compatible with the default action of the **icadecomp** method.

-c: Treat item pairs as complex entities. Normally, multiple items in the input data are analyzed separately and stored in successive output items as item1comp1 item2comp1 ... item1compN item2compN .... With this option, two successive input items are treated, for the decomposition, as if they would constitute a single vector of twice the length. The output has the identical form as without this option.

swap_fc: 'Transform' method to swap, usually in a data set of type FREQ_DATA with a single channel, frequencies for channels. The new 'channels' are named by the frequencies they contain and arranged on a grid. The number of points in the resulting epoch of type TIME_DATA will equal the number of shifts in the FREQ_DATA epoch. One application of this method is to frequency analyze a continuous single channel (only 1 analysis window 'shift') and write out a continuous file with each channel containing the spectral coefficients for one frequency. If the incoming epoch is of type TIME_DATA, it is treated like FREQ_DATA with only a single shift, i.e. a TIME_DATA epoch containing a single point will result.
swap_fc also performs the reverse operation, i.e. creating a single channel with multiple points if the input consists of multiple channels with a single point.

Arguments: NONE

swap_ic: 'Transform' method to swap the items and channels of the incoming data. Successive channels will become items and vice versa. One application is to create the full product matrix between all channels on one file and all channels of another using **subtract -m**: **subtract** is able to recycle channels from the "subtract file" with option -C, so that every channel will be multiplied if the "subtract file" contains only one channel. Multiple original channels in the "subtract file" can be converted to items using **swap_ic** and the input data made to contain the corresponding number of items by replicating items with **extract_item**. Note that channel names will be lost, since items do not have corresponding name tags.

Arguments: NONE

swap_ix: 'Transform' method to swap the item and x axes of the incoming data. Successive time or frequency points will become items and vice versa. Usually this will be interesting if there is only one item but multiple points, e.g. created using **trim -s**, which will be made into a single point with multiple items. Such single points can in turn be collected into a single epoch using **append**. Note that x data will be lost, since items do not have corresponding axis tags.

Arguments: NONE

trim: The first use of this transform method is to trim all channels to a specified length. This can be useful in cases where the get-epoch method does not provide a way to influence the epoch length (like **readasc** or **read_kn**) or where the epoch length might even vary (like in epochs read by **read_kn**).
Inclusion of data points starts at point offset (zeros are prepended if offset is negative) and proceeds through the specified length, appending zeros if less than length points are available in the source data starting at offset. This copying process is done for all ranges specified, appending all data in the output. This means that arbitrary sections of all channels (and the x data vector, if available) can be cut out and concatenated in any order. Ranges can also be specified by the x axis values of the first and last points to include (option -x; obviously, no extension with zeros will happen since boundaries are always within the data length), or, as a special extension, defined very dynamically by the times during which values of a given channel fall into given ranges. Epoch triggers are correctly transferred if available.
Note that when **trim** is used to select (concatenate) multiple sections of the current epoch, the absence of explicit x axis data has the consequence of a contiguous time axis being constructed later (eg by posplot). If the intention is to retain the original x values for each selected point, x axis data should be created explicitly before **trim** (set xdata 1).
By giving one of the collapse options, each of the specified ranges is reduced to a single output data point using the corresponding function. Thus, the output will have as many data points as ranges were specified. Corresponding ranges in the x data vector, if available, are always averaged. The nominal sampling frequency (sfreq) is not adjusted, because no general rule for such an adjustment exists. If a proper resampling with block averaging is intended, the **sliding_average** method should be used. **trim** should instead be used if exact control of the average points or varying range sizes and spacings across the epoch are needed.
A 'length' of 0 is replaced by the length of the rest of each incoming epoch starting at offset, which makes it possible to collapse all data points without specifying the epoch length explicitly in the script.

Arguments: offset1 length1 [offset2 length2 . . .]

Options:

-{asMhl}: Collapse ranges by averaging (-a), summation (-s) or by calculating the median value across each range (-M); by choosing the highest (-h) or the lowest (-l) value within each range.

-x: The two values given for each range are the x axis values xstart and xend instead of offset and length. **trim** finds the points with x values closest to xstart and xend within each epoch and calculates offset and length of the range so that both points are included. Note that the x axis search just looks for the value in the stored x axis vector (which is created if not already present), thus it makes no sense to use unit specifiers like 's' or 'Hz' in this case.
Because it is sometimes not desired to include both end points, it is possible to enter a point offset to the position closest to the given value in the form xvalue+offset or xvalue-offset. For example, **trim -s -x 0 5 5+1 10** sums up all points from the one closest to $x = 0$ to the one closest to $x = 5$, and then sums up all points from (1+ the one closest to $x = 5$) to the one closest to $x = 10$. In this way, two non-overlapping but adjacent regions are guaranteed to be used independently of the concrete x values present in the file.

-n channelname: The two values given for each range are *value* ranges for the given channel. **trim** selects point ranges in which the given channel value falls into the range. Selected point ranges from all value ranges are appended as with the other selection types. Internally, every consecutive point range in which the value remains within the given interval is added as one normal range, thus collapsing works across these consecutive ranges (one value per such range, not per range argument).

write_crossings: 'Transform' method finding the positions at which each specified channel crosses a given threshold value, or at which a channel has a local maximum≥threshold or a local minimum≤threshold (option -E). The output by default represents a valid avg_q trigger file: The point number at which the threshold was crossed, counted across all points seen by the method, is written to the text file outfile one per line. The second column contains a trigger code that is either 1 if the slope at the crossing was positive or −1 if it was negative. – This default behavior is most useful for continuous files read with the -c option, because in this case the cumulative point numbers indicate the position of the crossing relative to the start of the continuous file. To be fully operational as a continuous detector, **write_crossings** preserves state between epochs by default (i.e., the last point(s) of the previous epoch are considered).
write_crossings writes extended information into the output file as 'comment' lines beginning with a hash character ('#'): The number of points per epoch, as well as the sampling frequency and the threshold value, are output at the top of the file so that it is possible to calculate the epoch number from the output if necessary (see also option -x below). Also, the epoch number is output explicitly if option -x is given, and within each epoch the name of the current channel precedes the crossings data for that channel.
Using option -E, the point numbers and values at the local extrema in the given channel are output with trigger code 1 for maxima and −1 for minima, and a third column contains the value measured in the extremum. In this case, the threshold value serves to reduce the number of applicable points by the constraint that only maxima ≥ threshold and minima ≤ threshold are output. Note that by making this value negative it is possible to allow local minima with positive values and maxima with negative values.
While the standard output format of **write_crossings** is oriented at detection tasks in continuous files such as EOG candidate detection, option -e is for 'epoched' data. In this epoch mode, the detector is restarted anew for each epoch and point numbers relative to the start of the epoch are reported. With option -x, x-axis values are reported by [x-label=value] pairs. Note that options -e and -x are independent but usually -x will make most sense together with -e. With either option, epochs are separated by comments noting epoch number, z_label and z_value (if available). This is for example useful to measure event-related potentials or to describe peaks in the latency–frequency plane.
As outfile argument, the name 'triggers' is special besides the usual 'stdin' and 'stderr' values: It indicates that the points found are written into the internal epoch trigger list instead of to a file. The trigger code used is ±channelnumber. Epoch triggers can be shown by **posplot**, queried by **query** or written to an event-aware (continuous) output format together with the data. The -x option is meaningless for writing 'triggers' but -e should be used for epoched data.

Arguments: channelnames threshold outfile

Options:

-E: Write extrema rather than threshold crossings.

-e: Epoch mode - Restart detector for each new epoch. This is recommended whenever consecutive input epochs do not form continuous time series.

-x: Report x-axis values rather than continuous point numbers.

-i nr_of_item: act on this item number ($\geq$0)

-R Refractory_period: The detection algorithm is forced to remain silent for this many points after the last issued event. This can be used in EKG or fMRI artifact detection, for example. Note that this option can lead to unbalanced positive and negative crossings.

-o offset: The offset will be added to each output point number. Since the reported point numbers can only be based upon the epochs seen by write_crossings, the output file cannot be directly used as trigger file if the Get_Epoch_Method started reading in the middle of the (usually continuous) data file. A properly computed offset can be used to compensate for this.

zero_phase: Method to set, for each incoming complex map, the phase of one channel to zero by applying the appropriate phase factor to all channels. If channel_name is MAXAMP, for each map the channel with maximum amplitude is found and used as the zero_phase channel. If channel_name is NYQUIST, fit a linear regression line to each map in the Nyquist plane and define the zero phase by its angle. This means that an elongated point cloud in the Nyquist plane is rotated towards the real axis.

Arguments: channel_name

5.4 Methods for data reduction: Collect_Methods

append: Collect method to assemble all incoming epochs to a single epoch *in memory* by appending the time signals for each channel (default) or by adding channels or items. This is very similar to the **add_channels** method, the difference being that points, channels or items are added successively from the iterated queue instead of once from a file. If points are appended, the number of channels and items in the incoming epochs must be equal in all incoming epochs, but not necessarily the number of sampling points; if channels are added, the number of points and items must be equal but the number of channels may vary, and for adding items, the number of items may vary. Since this method performs no data reduction at all, it is mainly useful in situations where all data is known to fit into memory but cannot be read as one epoch from the start. For example, if a spectral analysis of a single channel from a continuous file is performed and the frequencies distributed into separate channels using **swap_fc**, the single points resulting from each run through the iterated queue can be collected into one epoch using **append**, yielding time courses across the length of the continuous file for each frequency.
Alternatively (option **-l**), the epochs can be collected into a linked list of epochs to be able to display them together via the **posplot** method. This can be used to visually compare a number of epochs or to produce a script that is equivalent to the standalone program **do_posplot**. This script would read all epochs from an ASC file, append them into a linked list using **append -l** and call posplot in the post-processing queue. Note that most methods, including put-epoch methods, will only recognize the first epoch within a linked list, so that generating this representation is presently only useful for **posplot**.

Arguments: NONE

Options:

{ -c -p -i -l }: Add channels, points (default), items or link epochs in memory (for displaying with **posplot**)

average: Collect method to average all incoming epochs (which should, of course, have equal types and sizes). The 'leaveright' rightmost items are summed (non-weighted, even in the presence of the -W option) instead of averaged.

Arguments: NONE

Options:

-M: Match the channels of successive epochs by name. In this case, the averaged epochs may have different numbers of channels, such as when forming a grand average with deleted bad channels. Unless one of the sign test methods is specified, an additional output item is created containing the number of averages or sum of weights on a by-point basis. Note that this method will give unexpected results with epochs containing multiple channels with the same name.

-N: forces the output nrofaverages property to be the number of averaged epochs; this useful if weights are used with -u, since the correct degrees of freedom for the t-test can be determined from the number of averaged files, not the sum of the weights. This way, a grand average of high quality (using weights) can be obtained simultaneously with the correct degrees of freedom of the t test.

-W: Use the nrofaverages property of the incoming epochs to perform a weighted average. This is only useful if the epochs come from multiple averaged files (readasc or NeuroScan format) with different nrofaverages values.

-s: Perform a sign test against baseline, i.e. count − and + incidences. Warning: the comparison against averaged baseline produces spurious 'significances' in the baseline on data with asymmetric distribution. -ss is recommended instead (cf. section MAIN:3.2).

-ss: As -s, but average +/− counts against all single baseline values. This means that one comparison is computed for each baseline value. If the current value itself is in the baseline, then the test against itself is skipped. The averaged test result $r \in [0 \ldots 1]$ is added to the n_+ counter, the complement $1 - r$ to the n_- counter.

-t: Perform a t-test of each value against 0 (note that this is equivalent to the option -d of **ascaverage** !). By using the **baseline_subtract** method before the average, this becomes a t-test against averaged baseline. The items added to the output for each input channel, point and item are the resulting t and two-tailed p values.

-u: (issued together with -t) Output the accumulated parameters for the t test (sum and sum of squares) instead of t and p. The parameters can subsequently be used for a paired t-test, for example using the **subtract -t** method.

minmax: Collect method to collect minimum and maximum of the received values. By default, the resulting epoch will be twice the size of each input epoch, because two adjacent items result for minimum and maximum of each input item. Optionally, the extrema can be calculated over all points (one output point with two items results) and over all channels (one output channel named 'm_collapsed' results).

Arguments: NONE

Options:

-c: Collapse over all channels

-p: Collapse over all points

histogram: Collect method to calculate a histogram of the values received from each incoming epoch. The histogram boundaries are the same for all points and channels and must be specified explicitly. Different input points or frequencies are mapped onto different output items (mapping into a single output item can be requested using option -p), and the x axis will be the amplitude bin on output. The input may not contain multiple items per point. The histograms may be calculated separately for each channel (default) or for all channels combined (option -c), in which case only one channel descriptor named 'h_collapsed' is output. The processed input x range can be restricted in order to avoid output with a large number of items. The amplitude values and unit will be properly noted on the output x axis and eventual latency values on the z axis, but since no values are associated with the item axis, no reference to the point or frequency values corresponding to output items is available in the output.

Arguments: hist_min hist_max nr_of_bins [minx [maxx]]

Options:

-c: Collapse over all channels

-p: Collapse over all points

-o: Assign the two ends of the histogram to values outside the mapped y range

null_sink: Collect method to just discard each incoming epoch and close the epoch loop, used if all desired processing takes place within the iterated queue. For example, if all input epochs are needed in the output, perhaps in a processed form or in a different data format, a Put_Epoch method would be present in the iterated queue.

Arguments: NONE

5.5 Methods to output data sets: Put_ Epoch_ Methods

writeasc: Put-epoch method to write epochs to an ASCII (asc) file.

Arguments: Outfile

Options:

-a: Append data if file exists

-b: Write binary format

-c: Close the file after writing each epoch (and open it again next time)

-L: Write all linked datasets

write_brainvision: Put-epoch method to write data in Brain Products (TM) "Brain Vision Data Exchange" format (cf. **read_brainvision**). This format consists of three files, "xx.vhdr", "xx.vmrk" and "xx.eeg". The VHDR file, with text-format metadata including data format and channel specifications, should be given as 'Outfile' argument. Triggers are written to the VMRK file and the binary EEG data is written to the EEG file.
data_type can be INT_8, INT_16, INT_32, IEEE_FLOAT_32 or IEEE_FLOAT_64.

Arguments: Outfile data_type

Options:

-a: Append data if file exists

-c: Close the file after writing each epoch (and open it again next time)

-S: Swap byte order relative to the current machine. This would be used when writing data on a machine with a different endianness than the one on which the data is going to be read.

-P: Points vary fastest in the output file ('nonmultiplexed', called "VECTORIZED" in the format). This only makes sense for writing a single epoch of data.

write_freiburg: Put-epoch method to write epochs to a binary Freiburg continuous file, as used for sleep data (cf. **read_freiburg** -c). To the name given in Outfile, '.co' is appended for the data file and '.coa' for the info file, which is a text file holding the lengths of the compressed sections in the data file 'Outfile.co' along with the channel names and sensitivities. Note that to use the compressed section lengths (**read_freiburg** does not use them), the reader must know the number of points stored per epoch (usually 512). Also, some sleep analysis programs require such a block to correspond to 5 s of recording. Data is written as shorts. By default, no sensitivity factor is applied. Using the option -s, such a sensitivity can be specified, assuming that the incoming data is in microvolts. Therefore, rescaling may be necessary using the **scale_by** method.

Arguments: Outfile

Options:

-a: Append data if file exists

-c: Close the file after writing each epoch (and open it again next time)

-s sensitivity: Use this sensitivity (in nV/Bit). Assuming that the incoming data is in microvolts, the data is scaled appropriately in the output to achieve this resolution, and the sensitivity value is written to the .COA channel table file.

write_generic: Generic put-epoch method for writing headerless files with data stored in successive elements of one of various types. Since no header is written, all meta-information is lost. data_type can be uint8, int8, int16, int32, float32, float64 or string; cf. the **read_generic** method. If the output file is to contain a header, then the header section must be prepared by some other program and the -a option can be used to append the data to it. The 'string' data type output is a text tab-delimited floating-point format with data points (or channels, with the -P option) separated by newlines. It can of course be read by the corresponding data type option of **read_generic** . If the string data output should be integer, **calc rint** can be used before this method.
Multiple items per channel and sampling point are silently written as adjacent single values, which will

make the output file appear to have a corresponding multiple of channels or, with the option -P, points. **read_generic** can, however, properly redistribute the values across a specified number of items. If only one item is to be written, the **extract_item** method should be applied first.
The options -z and -C add columns to the output that usually have a high level of redundancy; however, there are many multidimensional visualization and statistics packages which accept the data in such an 'exploded', non-hierarchical view.

Arguments: Outfile data_type

Options:

-a: Append data if file exists

-c: Close the file after writing each epoch (and open it again next time)

-x: Prepend the x axis data as an additional 'channel'. Note that the analogy to a 'channel' of data is not perfect since the x axis data always represents only a single item.

-z: Prepend the z value as an additional column (i.e. as 'channel' or 'point' whichever varies fastest).

-C: Similarly, prepend the comment as an additional column. This is a special case because the comment is a text field. For the 'string' data type, the comment itself is output, while the comment is silently converted to a numerical value for the numerical output types (yielding 0 when the comment does not start with a valid floating-point number). For the 'string' data type, the user should assure that the comment is properly formatted for the output file format to make sense.

-N: Prepend the channel name as an additional 'point'. The text/number issue is treated as with -C above.

-S: Swap byte order relative to the current machine

-P: Points vary fastest in the output file ('nonmultiplexed')

-s epochsep: Write this string at the beginning of each new epoch except at the beginning of the output file. Note that stdout and stderr have no beginning. Within the epochsep string, ' \ n' indicates a newline.

write_hdf: Put-epoch method to write epochs to an HDF file. The compression options are provided by HDF 4.0; Deflation does not yet work correctly. See **read_hdf** for more information about the library.
If option **-c** is given, a continuous (also called 'record') data set is written along the points dimension. If option **-a** is also selected to append to an existing file, a data set to append to is searched that has the same dimensionality as the data set to be written. If none is found, the method falls back to creating a new data set in the file as without the option **-c** .

Arguments: Outfile

Options:

-a: Append data if file exists

-c: Continuous output (unlimited first dimension)

{ -rle -nbit -skphuff -deflate }: Choose output compression

write_kn: Put-epoch method to write epochs to a binary Konstanz format file (by Patrick Berg). Since data is coded as short integers, a conversion factor is needed to project the data to a 16 bit-codeable range. This factor is stored as a short with the data and taken into account on reads, so that its value only changes the data representation but not the data values. Since it is stored as short, however, large factors like $1e15$ should be applied beforehand if necessary (e.g. by **scale_by**).

Arguments: Outfile conv_factor

Options:

-a: Append data if file exists

-c condition: Set condition code

-p: Pack (compress) data

-s subject: Set subject number

-T string: Specify how trigger codes are distributed into the 3 condition and 5 marker values available in a trial, e.g. 'm5c2' would mean that the lowest byte of the trigger code will become the first 'condition' value and the higher byte will become the fifth 'marker' value. Default is 'c1', i.e. only the first 'condition' entry is set (cf. **read_kn**).

write_mfx: Put-epoch method to write epochs to a 'Mün ster File eXchange' file (cf. **get_mfxepoch**). The conversion factor is stored in the file header and automatically taken into account when the data is read, so that it changes only the data representa tion but not the data value (disregarding rounding effects).

Arguments: Outfile conv_factor

Options:

-a: Append data if file exists. The channels are matched by name, not position. This means that all channel names in the epochs to be written must be present in the existing MFX file. Any additional channels in the MFX file are filled with zeros.

-c: Continuous output: The resulting MFX file will contain a single 'epoch' only.

-T: Add an empty trigger channel to the output. This has no effect while appending to an existing file.

write_rec: Put-epoch method to write epochs to a binary REC (EDF) sleep data format as described by Kemp et al. (1992) (cf. **read_rec**). Note that each epoch is written as one block (record) of the output file; if it is necessary to use a given block size on output after processing with a different epoch length, the file should be written to a temporary file and then re-read with an epoch size corresponding to the block size needed. There is not yet an option to write different channels with different sampling rate, offset and sensitivity, as the format would support. Data is written as shorts. By default, the resolution is 1.0 and the number of bits (that define the range of digital and, via the resolution, physical values) is 16.

Arguments: Outfile

Options:

-a: Append data if file exists

-c: Close the file after writing each epoch (and open it again next time)

-b bits: Pretend this number of digitization bits (default: 16)

-r resolution: Digitize in steps of this size (default: 1.0)

-P Patient: Set the 'patient' field value

-R Recording: Set the 'recording' field value

-S reServed: Set the 'reserved' field value

write_sound: Put-epoch method to write epochs to any of the sound formats supported by the SOX ('SOund eXchange') package (see **read_sound**). This method is mainly here to make it possible to listen to EEG or MEG data, and thus to hear the frequency content and the effect of transform methods like filters.
avg_q uses the SOX library verbatim, just writing data to it and redirecting fatal errors, warnings and reports (tracelevel 2) to its own error and trace handling system. The output file type is determined from the file extension. For example, 'a.wav' will write a Windows RIFF file, 'a.voc' a Creative Labs VOC file, 'a.ussdsp' will write directly to the USS sound system audio output /dev/dsp (on Linux), 'a.sunau' will write directly to the Sun Microsystems compatible /dev/audio interface. The latter two will not actually create Outfile. Of course, not all output formats (and especially not all sound processing software) will be able to handle more than 2 or 4 channels, so that it will often be advisable to use the **remove_ channel** method to select the channels to be written.
The incoming data must be scaled so that the **highest** 8, 16 (or 32...) bits of a 32-bit integer representation of it contain some data, or the output will consist of zeros only. This means that writing data with about unity amplitude to an 8 bit file will need some method like **scale_by 4e9** before **write_sound** .

Arguments: Outfile

Options:

-H: Help. Query the SOX library for supported formats and exit.

{ -8 -16 -32 }: Select number of output bits. Default: 16.

{ -u -s -ul -al }: Select the output coding style: Unsigned, Signed, ULaw or ALaw. The default depends upon the format written.

write_synamps: Put-epoch method to write epochs to a binary NeuroScan epoched (.EEG), continuous (.CNT, option -c) or 'averaged' (.AVG, option -A) file. For the .EEG and .CNT formats, data is coded as short integers, and the conversio n factor conv_factor is used to project the data to a 16 bit-codeable range. This factor is stored with the data as a 32-bit floating-point value (inversely coded into the channel property 'sensitivity', which is actually an amplitude-p er-bit measure) and taken into account on reads, so that its application does not result in an actual data scaling. If available, the trigger code for the epoch is written into the epoched file in a way compatible with **read_synamps** , so that stimulus and response codes are retained when epochs were read from a NeuroScan file. The continuous type created with the -c option is '100/330 kHz' (cf. **read_synamps**); the event table will usually contain only the start/stop marker at the end, but if trigger tables were present in the incoming epochs (as with **read_synamps** -T), these triggers will be written to the event table as well. This makes it easy to manipulate, resample or append .CNT files while properly maintaining the events. The .AVG format is provided mainly because various source modeling programs can read it; data is stored as 32-Bit floating point values and the conv_factor is ignored. Since it only supports a single epoch, trying to write a second epoch or trying to append epochs to an existing file is an error with this format.

Arguments: Outfile conv_factor

Options:

-a: Append data if file exists

-L: Write all linked datasets

{-E -c -A}: Select the output format: EEG, CNT or AVG (default: EEG)

write_vitaport: Put-epoch method to write epochs to a binary 'vitaport' file. As explained for **read_vitaport**, this format stores the channels separately, which is why appending to such a file is not feasible. For the same reason, **write_vitaport** collects the data in separate temporary files, one per channel; the actual target file is only built after all epochs have been processed in this manner. This two-stage process also allows **write_vitaport** to automatically choose scale factors for storing the channel data as short integers. Any channel named 'MARKER' is stored with conversion factor 1.0 in any case, because markers are read as raw integers. Currently, only the 'reconfigured Vitaport II format' is written, all channels are coded with 16 bits and with the same sampling frequency. One big problem with this format is that the sampling frequency is not simply stored as a floating-point value, but rather as a divisor of a base frequency of 76800 Hz. For example, it is not possible to precisely code a sampling rate of 500 Hz. The nearest value chosen when such a file is written is 498.701 Hz, which will render trigger times incorrect that are in time units, such as the VITAGRAPH markers that are in milliseconds. It thus can be necessary to resample a file (e.g. using the **sliding_average** method) to a rate that can be exactly represented before writing Vitaport data. As with **write_synamps**, any triggers passed with incoming epochs are written to the event table at the end of the file.

Arguments: Outfile

6 User interface of the posplot method

Data organization:

The posplot module can display data with up to 5 dimensions:

- A number of data sets ('z axis')
- A number of channels
- Multiple y values (items) for each x in each of the above (e.g. complex data)

Note that there are quite elaborate precautions to display data sets with different numbers of data points or channels in a useful manner. Different numbers of data points are handled by always plotting the data against the corresponding x axis values, therefore allowing data with different sampling rates but equal x data ranges to be superimposed just as data spanning different x ranges. Posplot version 2.98 (March 2001) introduced the use of a 'channel map', a list of channel names and positions initialized from all available data sets, which guarantees that channels with the same name and position always appear at the same plotting position irrespective of the order they have in the different data sets (of special importance for overlaying data sets in which channels have been removed because of artifacts). The *order* of channels defined by this list is mostly important for the vertical display mode. Note that some functions using or modifying channel positions, such as dipole fitting, triangulation and gridding, cannot use posplot's channel map but work on the channels as defined by the first or first selected data set. The interactive 'sensor position manipulation', however, does work on the channel map and modifies the position of the current channel in all data sets synchronously. Channels with identical names but different positions are displayed with a number in braces appended to their name, without modifying their actual names in the data set. When channels are selected in posplot by entering their name, the unique name shown by posplot should be used. If multiple channels with the same name and position occur within one data set, the value 0.123 is added to the x position of the second channel in order to maintain a unique mapping between (name, position) in each data set and in the channel map.

Selection of data sets:

To show/unshow a data set, left click into the data set bar at the right. By default, no data set is selected and all data sets will appear in white. Shown (selected) data sets appear in green. Note that data traces will only appear after plotting is toggled on by using the 'p' key (see below in the 'commands' section).

Data sets can also be "fixed" by a right click, i.e. their display status will not be changed by the up/down movement or 'toggle all' commands. Fixed selected data sets appear in yellow, fixed deselected data sets in red. Fixing data sets was designed for situations where one specific data set should always remain shown for comparison, while the other data sets are successively reviewed (e.g. grand average remains displayed, individual averages are cycled through).

Keyboard:

1-9,0 Equivalent to a left click at dataset 1-10

, Toggle selection of a data set using the value last entered with '='. If the string starts with '#', the n'th data set is selected; else, the data set with z-value closest to the input value is selected. If the '=' argument ends with an 'F', the fixed status of the data set is toggled instead.

(,) Shift the data set selections up (down) by one. If only one data set is selected, the previous (next) data set will be selected instead; multiple selections move up or down in synchrony.

***** Toggle all data sets between shown and not shown.

S Swap between x-y (z) and z-y (x) display (dataset axis in parentheses).

Selection of channels to display:

To toggle a channel on/off, point to it with the mouse and press the <backspace> key. Channels can also be toggled by name or number by entering the name, or the number with a hash '#' prepended, as an argument with '=' (see below) before pressing <backspace>.

To toggle displaying a single channel, point to it with the mouse and press the <enter> key or the right mouse button. As with <backspace>, a channel name or number to make the only one displayed can be given; in this case, there is no toggling.

Changing the channel layout:

_ (underscore): Cycle through three 'position modes': position layout of channel plots, 'vertical mode' (i.e. channels stacked vertically) and 'overlay mode' (i.e. all channels are plotted on top of each other).
In vertical mode, the keys J and K can be used to move the selected channel down or up in the channel map and therefore down and up on the screen. The selected channel is moved directly before or after the first or last shown channel.
A fourth, 3-D 'sensor position manipulation' mode can be entered by typing "change" as an argument with the '=' command (see below) before the underscore. While all other keys continue to work as usual, the uppercase rotation keys H, J, K and L are redefined to move the currently selected channel on the screen to the left, down, up and right, respectively. This is done by adjusting the actual 3-D sensor positions in the plane perpendicular to the current line of sight. Note that while all other rotations etc. in posplot only change the view on the data, this changes the actual sensor position values in the current data set as they are seen by subsequent methods. The normal position mode is entered by typing underscore again.

G Arrange the channels on a grid. This overwrites the current positions. Gridding actually works only on the first or first selected data set; this data set is then used to initialize the channel map (cf. the channel map initialization done by 'C'). Therefore, channels not available in this data set will retain their positions. If a '=' argument is available, it will be used as the number of colums of the channel grid to create.

-,+ Change viewing distance to sensors

/, Increase/decrease the field of view

<,> Change size of the channel plot frames (one step smaller/larger)

h,j,k,l Rotate sensors; H,J,K,L: in larger steps

t,T Twist: rotate sensors around the viewing axis left/right

Z Set the last selected channel to be the central 'pivot' channel

c Reset distance, rotation and 'pivot channel' settings.
If an input value is available, this command sets the condition code of the current epoch. This is probably the preferable technique for reviewing epochs and classifying them interactively, since epochs can later be filtered depending on condition codes by using the **assert** method5.2.

s Standard position of the sensor array for BTI 37-channel or Rome MEG. The topmost channel is rotated to the top, the frontmost channel either to the right or to the left depending on the side of the head on which the sensor was placed. This side is indicated by a small head outline faced in the appropriate direction.

Changing the range and transformation of displayed data:

b Toggle background/foreground colors. By default, background is black and "foreground" (i.e., coordinate system axes etc.) white. Note that the background is never printed to postscript, therefore the default leads to white on white plotting for the comment, coordinate sytem and eventual curves plotted in white. By choosing black on white plotting before generating postscript plots (**o**, **O**), these elements are plotted in black.

u cycle between using differently dashed lines for different datasets (default), using different colors and plotting them with the same (green, solid) lines.

U Plot the topmost selected dataset with normal line style

v Lock the vertical scale (lower and upper boundary values). If no argument is entered using the '=' function or if the given argument starts with 'i', an 'incremental' vertical scale mode is started, i.e. the range only expands as necessary but doesn't shrink. If a '=' argument is available, it must have the form lower_bound upper_bound and the scale is fixed to these values. From incremental scale mode, pressing 'v' switches to fixed mode with the current values; from fixed mode, pressing 'v' reactivates autoscaling.

~ Toggle plotting of negative side down (default)/up

[,] Let the currently selected x value be the left (right) border. If the selection already is at the border, the border is set to the left (right) edge of the available data. This allows to toggle between restricted and full view. If a '=' argument is available, this is used to set the corresponding border.

C Reset the x range selection and the channel map. Initialization of the new channel map starts with the first selected data set, thus setting the order in which channels are displayed to the order in which they appear in that data set.

@ Set the subsampling step: Data is displayed using this step for speed, but is processed and can be selected at the original rate. In September 2004, "auto-subsampling" was introduced and enabled by default based upon the number of displayed points per available display points in the x dimension. Auto subsampling is switched off once a sampling step is specified by hand. '@' without argument toggles between fixed sampling step and auto subsampling; the fixed step width used is 1. Since August 2007, a line is drawn between the minimum and maximum data value in the skipped points; this effectively avoids aliasing problems (seeing lower frequencies than actually present in the data) and shows spikes where they are present.

{,} Decrement (Increment) the itempart (current item) to show

' Switch functions below from regarding multiple items as complex numbers to just acting on the current item alone

P Select/deselect Power function (for complex values; square for scalars)

a Select/deselect Abs function (for complex or scalar values)

A Select/deselect Phase function (for complex values)

e Cycle through exp functions to various bases (exp, exp10, exp_dB)

E Cycle through log functions to various bases (log, log10, log_dB)

The following commands actually modify the data in memory:

D Detrend the selected datasets (on tuples, each row is detrended separately)

$ Subtract the value at the marked (or leftmost displayed) point from each curve (= detrend -o lastselected)

% Differentiate the data (pointwise)

& Integrate the data (pointwise)

Selection of x values (setting the marker):

Mouse: To select a specific x value and channel, click into the plot frame of a channel.

i,I increment (decrement) the currently marked x-value in steps of the subsampling as specified by the '@' command. If no x value is selected, the mark enters from the left (right) of the currently visible x range. The mark will be cleared if it moves outside the visible x range. – If epoch triggers are shown (see key 'M'), advance the marker to the next (previous) trigger.

. Set the current x-value selection from the string entered with '='. If the string starts with '#', the x value of the n'th data point is selected. With '#', the entered value can also end in s or ms; in this case, the value is converted to an integer using the sampling rate.

Commands:

= Input a string. String input ends with the return key; use backspace to delete the last character. The string is used by the '.', ',', '@', 'v' and <backspace> commands if available.

p switch plotting (data curves) on/off

x switch coordinate system on/off

m switch vertical bar at selected x position (marker) on/off

M switch vertical bars at trigger positions on/off

n switch channel names on/off

d switch heightfield grey display on/off. Grey levels normally vary from black for the topographical minimum of the currently selected point to white for the maximum; if negative values are plotted upward, this association is reversed. If the vertical scale is fixed, the global minimum and maximum values are taken instead; therefore, the grey scale will remain fixed as well, which is desirable for viewing time sequences as a 'movie'.

? switch epoch information display on/off. This shows the contents of some epoch variables such as nrofaverages which are otherwise not evident from within posplot.

N Create a new trigger at the current marker position. The trigger code is 1 by default but can be set using a '=' argument. If the marker is exactly on an already defined trigger, that trigger is deleted instead.

CTRL-L Redraw screen

q,ESC,Q,V Quit from viewing the current epoch. 'Q' signals that the current epoch should be rejected (if posplot is called from the iterated queue of an **avg_q** script). 'V', finally, also signals that the iterated queue should be ended, just as in **assert -S**5.2. This can be used, for example, in programmed batches where the script should continue to run but no further epochs from the current file(s) should be considered.

X eXit. This command terminates the execution of the whole script from which posplot was called. It is nicer than pressing Ctrl-C to terminate the **avg_q** non-GUI version and necessary to close the window and return to the GUI if a script was started within a **avg_q** Graphical User Interface.

Y Output all channel names to stdout with x and y screen position and the values of the data selection

y Output a simple list of x and y screen positions and the values of the data selection, ready for plotting

r Enter record mode: a new record file is created and the key/mouse events stored in this (ascii) file until 'r' is entered again.

R Replay the events stored in the record file.

For both recording and playback, the default file name is "posplot_rec". If an argument was entered using '=' then this is used instead. The format of this file has changed in posplot version 2.33 (May 1997). It used to be a two-column list of numbers, 'device' and 'value', directly taken from the corresponding values of the GL library used. Because it can be very helpful to edit the file by hand, the key presses are now stored as single characters, '\n' represents the RETURN key, '\\' the backslash character itself, and '\L *xpos ypos*' a left mouse click at screen coordinate *xpos*, *ypos*. This works correspondingly with '\M' and '\R' for middle and right mouse clicks. Note that use of mouse events, although supported, is not recommended for recording, because the screen layout may be different at the time of playback; also, when editing the file, it is hard to guess what a mouse click was meant to do. Newline characters are ignored and may be used to structure the file when editing by hand.

Postscript output (VOGL version only):

o,O Output the current drawing in postscript format. The output file is named "posplot_out.ps" by default. If an argument was entered using '=' then this is used instead. o: landscape, O: portrait

Saving data to an asc file:

W Write all datasets (if none is selected) or all selected datasets to a (binary) asc file. The output file is named "posplot_out.asc" by default. If an argument was entered using '=' then this is used as file name instead. The data are written in the form presently in memory, including modifications by differentiation or integration, channel position editing, and trigger editing. If the output file already exists, the new data is appended.

Mouse buttons:

Left Select a channel and set the marker to the x-position within that channel

Middle Output value at selected position of this channel to stderr; also selects this channel, but doesn't redraw the screen (see CTRL L)

Right Make the channel closest to the cursor the only displayed channel. A second click with the right button switches the other channels on again

7 Averaging across asc files: ascaverage

Because it provides relevant, especially statistical, functionality in addition to **avg_q**, the program **ascaverage** should be briefly described. Its basic function is to read a number of asc files (all with the same number of channels, items and epochs, as they result from an analysis applied to multiple subjects) and to average the values at corresponding positions across files. Options allow the additional computation of the sum and sum of squares across the averaged sample and the calculation of the corresponding t and p values for an appropriate test. The output, which is an asc file with the same size parameters as the input files (normally) or grown by two items (for the statistical options), is written to `stdout` if no output file is specified by the -o option. The option -o must be used, however, on some systems (like MSDOS) because these systems perform text translation on stdout, which would garble the data.

In combination with the support for multiple get-epoch methods and statistical options of the **average** method, **avg_q** itself now provides much of the functionality of the ascaverage program. Since preprocessing such as baseline-normalization can be performed using other methods prior to averaging, the **average** method needs much less options while still providing more flexibility. However, each file processed by **ascaverage** can have multiple epochs, e.g. for the different latencies or 'shifts' resulting from spectral analysis, and averages the corresponding epochs for all input files.

Arguments: ascdatafile1 ascdatafile2 ...

Options:

-B: writes output in binary format.

-s: outputs the square root of the averaged squares. This is useful if the files contain spectral data stored as the square root of the spectral power.

-t: performs a t-test against average baseline for each value.

-T: does the same with the log values (although the original values are used for the average)

-d: performs a t-test of each value against 0. This is useful if the values themselves are already differences or in conjunction with **-u**.

-D: does the same with the log values.

-u: outputs the accumulated parameters for the t test (sum and sum of squares) instead of t and p. This option is only useful in conjunction with one of the t-test options. The parameters can subsequently be used for a paired t-test, for example with the **subtract -t** method of **avg_q**.

-N: forces the output nrofaverages property to be the number of files; this useful if weights are used with -u, since the correct degrees of freedom for the t-test can be determined from the number of averaged files, not the sum of the weights. This way, a grand average of high quality (using weights) can be obtained simultaneously with the correct degrees of freedom of the t test.

-o outfile: Write the result to file outfile rather than to `stdout`.

-S sum_only: Just sums the sum_only rightmost items, but averages the remaining items (useful if the rightmost items contain counts).

-w weight_filename: Instead of simple averaging with equal weight, file names to process and associated weights are read from the given weight file, one per line. No additional asc files may be specified on the command line. The operation performed is $(\sum_i x_i w_i) / (\sum_i w_i)$, where i runs across files, x_i is the value from and w_i the weight for file i.

-W: uses the nrofaverages property of the specified files as weights. nrofaverages is set correctly by the **average** method of **avg_q** and also by **ascaverage** itself (namely, to $\sum_i w_i$ if **-w** or **-W** are used), so that averaging files with **ascaverage -W** is usually equivalent to averaging with equal weight all data that was averaged for the input files.

8 Extracting spectra for plotting: extspec

The spectral data resulting from **avg_q** spectral analysis is a large array of values that generally must be output in single channel pieces for plotting. This is the application for the **extspec** program. It outputs the 3-D data (latency × frequency × spectral coefficient) that is contained in the epochs (≡ latencies) of an asc file, in the ASCII text form read by various plotting programs: The frequency in column 1, the latency in column 2, and the data items in the following columns. All frequencies for the one latency are output on successive lines, and a blank line separates each latency block from the next.

The **extspec** program is also responsible for calculating statistical parameters from +/− counts that are present as additional items in the data, as produced by the `average -s` method. If the option -binomial is given, **extspec** computes three additional columns: the binomial probability p, the probabilistic logarithmic change measure $-\log_{10} p \cdot \text{sign\_of\_change}$ and the relative gain $G_r = (n_+ - n_-)/(n_+ + n_-)$. This corresponds to the values that can be calculated for a whole epoch in **avg_q** by the method **calc_binomial_items**.

Arguments: ascfile channelname [minx maxx]
The optional arguments minx and maxx restrict the x (frequency) range to output.

Options:

-zrange minz maxz: Restrict the z (latency) range that is output

-binomial: Calculate probability from binomial distribution.

References

Kemp B, Olivan J (2003): European data format 'plus' (EDF+), an EDF alike standard format for the exchange of physiological data. *Clin Neurophysiol* 114:1755–1761 5.1

Kemp B, VÃ€rri A, Rosa AC, Nielsen KD, Grade J (1992): A simple format for exchange of digitized polygraphic recordings. *Electroencephalogr Clin Neurophysiol* 82:391–393 5.1, 5.5

Le J, Menon V, Gevins AS (1994): Local estimate of surface Laplacian derivation on a realistically shaped scalp surface and its performance on noisy data. *Electroencephalogr Clin Neurophysiol* 92:433–441 5.3

Rosenberg JR, Amjad AM, Breeze P, Brillinger DR, Halliday DM (1989): The Fourier approach to the identification of functional coupling between neuronal spike trains. *Prog Biophys Mol Biol* 53:1–31 5.3

Index