Elekta Neuromag

# Signal Processor

## Reference manual

**Software version 2.94**

October 2008

ELEKTA

Manufacturer:

Elekta Neuromag Oy
Siltasaarenkatu 18-20 A
FIN-00 530 Helsinki, Finland
Tel: +358 9 756 2400
Fax: +358 9 756 24011
Web: www.elekta.com



Printed in Finland.

| **Printing History** | **Neuromag p/n** | **Software** | **Date** |
|---|---|---|---|
| 1st edition | DA10240-4A | 2.92 | October 1996 |
| 2nd edition | NM10240A-A | 2.93 | December 2004 |
| 3rd edition | NM10240A-B | 2.94 | 2008-10-09 |

## CONTENTS

# Part 1

## Module Reference

**Introduction**

This part of the manual contains variable reference information. It starts with reference of utility LISP modules delivered with the program. The following modules are described:

- **batch-processor**: A module that provides an easy way to perform some procedures automatically on several files and measurements.
- **channel-selection:** Enables easy definition of channel sets and creation of channel selection menus.
- **classify-by-table**: Classifier function for off-line averager which allows category definitions through a relatively simple configuration file.
- **pca:** This module performs principal component analysis on data.
- **ssp**: Signal space projection utility, that can be used to manage vectors that describe sub-spaces for projection operations.
- **std-selections:** Utility for easy selection of standard channel sets to the main window.
- **std-selections-nm122:** Standard channel selections for Neuromag122.
- **std-selections-vv:** Standard channel selections for Elekta Neuromag.
- **templates**: This is a utility that manages a template filter to ease template matching operations.
- **trigger-utils**: Set of utilities to ease trigger and event handling.
- **xfit**: Utility that enables easy transfer of data from Graph to Xfit.
- **xplotter**: Utility that enables easy data transfer from Graph to Xplotter.

# 1

**Changes to the reference**

The reference has been updated for Graph version 2.94.

Batch processor interface images updated to correspond latest appearance.

Added reference pages classify-by-table.

Added reference pages for trigger-utils.

Ssp module now adds channel names to the saved files.

Xfit has channel selection controls.

Xplotter has possibility to overlay data.

Added module version history to several modules.

## 1.1 BATCH-PROCESSOR 1.4

**General Description**

Batch-processor is a module that enables you to apply predefined processing to multiple data files. It essentially takes a list of files and processes them to produce some output files. It provides an easy to use window based interface to define the files and processing to be applied, in addition to built-in name generation for the output files. Processing definitions allow also multiple runs on each file with the data processing system tuned to slightly different settings. These routines also create a log file that records proceeding of the batch processes and possible error and log messages delivered.

**Installing the module**

The batch processor module is supplied with the Graph version 2.92 and later in the standard module library /neuro/lisp/graph-<version>/lib and to load the code into the system you need to give lisp command

```
(require "batch-processor").
```

After this command the functionality of the module should be available. Note however, that the setup files, that are used to save Graph state, are capable to load the code automatically if the module was present when the setup was made. Is such case loading the setup will load the batch processor automatically.

**Components of a batch definition**

A complete definition of a batch process consists of four parts:

- 1) List of files to be processed, possibly with some extra arguments
- 2) Output directory
- 3) A set of signal processor state definitions
- 4) A description of process to be applied

The file list is simply a list of file names. However there can be a set of arbitrary extra arguments attached to each file. Same file can not be entered twice with same arguments. The arguments are passed to the process code and they are process dependent and documented elsewhere for each process.

Output directory is the name of the directory where the output files should be saved. One can use special value `:source` to denote that the output files should be placed in the same directory where the corresponding source file is.

The definition for the states of the signal processor that should prevail during processing is given by its name in the batch dialog. The definition itself should reside in file

<*user-lisp-root*>`/batch-states/`<*name*>`.lsp`,

where <*name*> matches to the name of the state definition and <*user-lisp-root*> denotes the root directory of users private lisp code. One definition file can define arbitrary number of signal processor states, each of which is realized before the actual processing routine is invoked. It can define for example three different states which cause averaging process to average triggers one, two, and three. Each file is then averaged three times producing three output files, one for each trigger. The exact format of the state definition files is given in “State description syntax” on page 6.

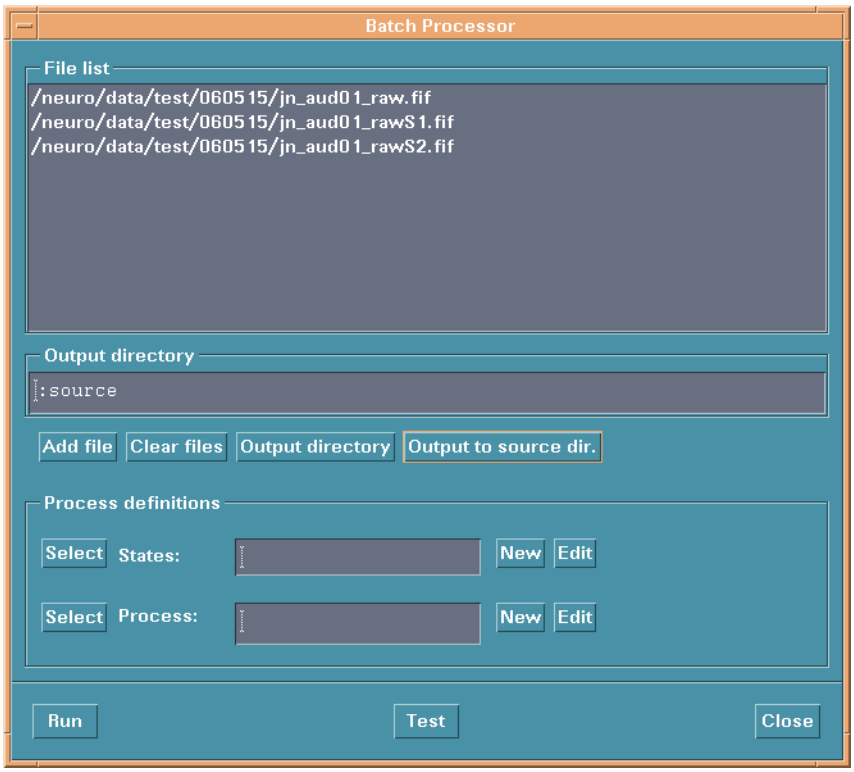
The definition of the process, that is to be applied for each file in each state of the processing system, is also defined by its name in the batch dialog. The actual code that is executed should be in

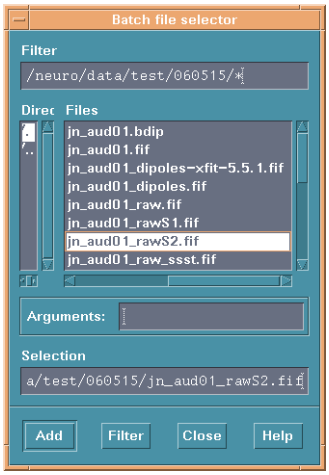
<*user-lisp-root*>`/batch-processes/`<*name*>`.lsp`,

in similar manner as a the state file. Process definition is essentially a LISP program, that has to obey some formal constraints. The syntax of the file is given in “Process definition syntax” on page 7.

## User interface

The batch processor has a friendly interface dialog that can be popped up from button **Batch...** in the **Commands** menu. The dialog is shown in figure below.




At the top of the dialog, there is a list showing all files currently selected to be processed. To add a file into this list, click the **Add file** button below the file list. This pops up a file selection box. Use the selection box to pick up the desired file. For details about selection boxes, see “Using file selection boxes” of “Signal Processor Users Manual”. You may give the extra

arguments to be passed to the batch process with the file in the text box above the file name field of the selection box. Pressing **Add** button will not remove the dialog, so that you can easily enter multiple files. To remove it, use the **Close** button.

In order to delete erroneous entries, select the desired file names by cursor, and press the **Clear files** button. Pressing the button when no files are selected clears all entries.

Below the file list is a text field for output directory. Pressing the **Output directory** button pops up a file selection box that allows you to select the directory easily. You may enter the directory also using keyboard. Pressing button **Output to source dir** sets the output directory to `:source`, so that output files will be created to the same directory as the corresponding source file.

The lower part of the batch dialog contains text fields for names of state definition and the process definition. You can enter them either by keyboard or by pressing the **Select** button in front of them. Pressing it pops up a file selection box viewing the correct directory. If the batch code directories do not exist, program asks user whether it can create them. Answer yes, if you want them to be created. The directory will be empty, except of a directory file that is a symbolic link to the directory containing example code provided with the system.

You can create a new definition file by clicking the **New** button. This pops up a file selection box where you should enter the name of the file to be created. After you have given the name, the file will be created, and a template description will be written into it. Comments in template code describe also the syntax of the file, so it is sort of self documenting. If you give name of an existing file, the program asks (somewhat misleadingly) if it is OK to write over the existing file. If you answer “yes”, an editor will be opened editing the file, and you can make changes to it.

You can edit the process or state definition file entered into a text field by pressing the **Edit** button on the right side of the text field.

### Testing batches

After all the above mentioned four fields have been filled, you can run the batch by pressing **Run** button at the bottom of the dialog. It is however recommended that you first run a test run, by clicking the **Test** button instead. The test button runs a bare bones process that just checks that all input files can be opened (test procedure uses the default-disk-file, and assumes that the inputs should be normal “.fif” files), that none of the output files already exist, that the output files can be created, and that they will be created only once (so that some process steps will not write over previous results). Test routine also realize all the different states of the processing system, so that obvious mistakes in state definitions can be detected (that is, those that cause errors).

### Running batches

To run a batch, press the **Run** button at the bottom of the dialog. The dialog will disappear and a log window will pop up. Then the batch process starts. You can not interact with the Graph program during the batch, unless the batch process asks you to do something, which is not recommended, since then the batch would need continuous attention. You can however still scroll the log window, since this runs independently of the main program.

To view the log file use the scroll bar on the right side of the text area. When the batch starts, the log dialog shows the end of the log and scrolls automatically forward when new log messages appear. However if you use the scrollbar to view the file. The automatic scrolling is then suppressed so that you can examine the text. To activate the automatic scrolling again, move the view to the end of the file. Note that the color of the trough of the scrollbar changes when the automatic scrolling is on.

Running a batch also creates a log file <*user-lisp-root*>`/batch-logs/Batch`<*date*><*time*>`.log` where date is in format yymmdd and the time in hhmmss. The date and the time tell the time when the batch was started. The log files are never deleted automatically, so you should clean up the log directory ever now and then.

### Restrictions

The batch processor expects currently that all input and output files are fiff-format files having extension `.fif` in their names. Using files having some other naming convention is possible, but the automatic test utility is not able to anticipate such an action. Same applies to cases where processing code produces output files having different name than the one that was given to it, or if it produces multiple files.

### State description syntax

State descriptions should be in directory

<*user-lisp-root*>`/batch-states.`

The name of the batch file should be <*name*>`.lsp` where <*name*> is the name of the batch. A batch may have same name as some previously defined function since the batch names are kept in a separate name space. Syntax of a state description is as follows:

```
(define-batch-states <state definition name>
 <base setup name>
 <modification> <modification>...)
```

Where the <*state definition name*> should be a symbol that matches the name of the file where the definition is saved. The <*base setup name*> should be a string defining a Graph setup file name compatible with LISP function `load-settings`. If the state file is available in your personal setup file directory, the base name of the setup is enough, for example `"basic"`. If not, complete file path is needed. Extension can be left out if it is `.setup`.

Modification entries describe how the signal processor state created by the setup routine should be modified to reach a specific processing state. The modifications can have two possible forms. First:

```
<simple modification> :=
(<name extension> [<lisp form> <lisp form>]...)
```

The *<name extension>* is a string to be appended to the file name of the output file when this modification is applied to the base setup. This term is always needed. The rest of the definition is a sequence of lisp forms (commands) that are evaluated to modify the base setup. Typically these forms alter resources of some widgets, or values of user parameters affecting the processing. Typical example would be to set averager trigger channel or tune filter settings.

Second modification entry type is a notation to form combinations of simple modifications. Its syntax is:

```
<combination modification> :=
 (combine (<sm> <sm>...)(<sm> ...) ...)
```

where *<sm>* stands for a *<simple modification>*. This form takes several lists of simple modifications. This syntax corresponds writing all combinations of modifications so that one simple modification is taken from each of the lists and these are merged together. The merging is done so that the file extensions are concatenated to form a new combined extension, and forms are concatenated to apply all the modification commands.

If the definition is for example

```
(combine
 (("A" (resource (G-widget "filter")
                 :pass-band '(band-pass 8 12)))
  ("B" (resource (G-widget "filter")
                :pass-band '(band-pass 15 25))))
 (("S1" (setu trigger-source '("threshold" 0)))
  ("S2" (setu trigger-source '("threshold" 1)))))
```

Then this is equivalent of giving the simple modifications:

```
("AS1" (resource...(... 8 12))(setu... 0))
("AS2" (resource...(... 8 12))(setu... 1))
("BS1" (resource...(... 15 25))(setu ... 0))
("BS2" (resource...(... 15 25))(setu ... 1))
```

**Process definition syntax**

The process definitions are essentially normal function definitions. They just have to adhere to some specific conventions. The syntax is as follows:

```
(define-batch-process <name>(&keys file output args)
 [<Comment string>]
 <form>...)
```

These functions are defined by define-batch-process instead of `defun`. The argument list must match conventions used by the batch processor, but otherwise the code can do what ever is needed. The arguments passed to the process during batch are following:

- `file`: String containing the full pathname of the input file.
- `output`: String Containing the full pathname of the output file.
- `args`: List of arguments that were given with the input file.

**Example process definition**

There is a sample averaging batch process in directory examples, under your batch process directory. The code of the routine is listed below.

```
(define-batch-process average(&key file args output)

  "Standard averaging batch process. This simply calls
  routine average with possible given arguments. The
  arguments   for average default to 10 and 10000
  (start and end of the span)."

  (if (> (length args) 2)
    (warn "batch:average: Too many arguments ~s" args))

  (let ((start (or (first args) 10))
        (end (or (second args) 10000)))
    (open-diskfile file)
    (average start end)
    (save-average output)
   )
)
```

This batch routine is a good example what the process code should do. As with all function definition, the first form may be a document string. It is a good idea to include all relevant information that is needed to use the code.

The second form

```
(if (> (length args) 2)
    (warn "batch:average: Too many arguments ~s" args))
```

just checks that there are not too many arguments. In the let construction, suitable local variables are set using the argument values, or to default values if the argument is not available:

```
(let ((start (or (first args) 10))
        (end (or (second args) 10000)))
    <code>
   )
```

Then finally we get to the actual code:

```
(open-diskfile file)
(average start end)
(save-average output)
```

This code does the following things. 1) it opens a diskfile defined by the argument `file`. 2) It executes the averaging command using the given or defaulted start and end time value. 3) It saves the result of averaging to file defined by the parameter `output`.

**Module version history**

Main new features / modifications in the module.

**Version 1.4**

Removed direct references to directory /neuro. This allows usage also when software is installed to a nonstandard location.

**Version 1.3 (package rev 1.2.2)**

Modified filename handling to make module to work in Linux systems.

**Version 1.2**

Added edit buttons to edit configuration files easily.

Allows using nil as the setup name and thus preventing loading any setup. This is useful if one wants to run the batch with different setups.

**Version 1.1**

Initial version.

## 1.2 CHANNEL-SELECTION 2.0

**General Description**

The package “channel-selection” provides an library utility for easy selection of standard channel sets to a display. It supersedes the functionality that used to be provided by module ‘std-selections’, which is now rewritten to use this module.

Code can add a new submenu into desired menu. This menu contains one button for each predefined selection (channel set), and a submenu that has set of channels sets. When a new channel set is selected from the menu, the channel list is inserted into a pick widget defined at the creation time of the menu. The pick widget is then linked into a display defined also at the creation time of the menu.

Note that the menu is a tear-off menu, so that it can be lifted on the workspace for handy operation.

**Installing the module**

The channel-selection module is supplied with the Graph version 2.93 and later in the standard module library /neuro/lisp/graph-<version>/lib, and to load the code into the system you need to give LISP command

```
(require "channel-selection").
```

After this command the functionality of the module should be available. Note however, that the setup files, that are used to save Graph state, are capable to load the code automatically if the module was present when the setup was made. Is such case, loading the setup will load the module, and the require command is not necessary.

**Functionality**

In addition to routine for creating a channel selection menu, it contains also routine for defining channel sets and some utility routines that can be used from lisp code control the utility. Following functions are exported:

**clear-channel-selections (&optional device)**

Clear channel selection definitions. Removes all current channel definitions for given device type. If no type is defined, all definitions will be cleared.

**define-channel-selections (devtype &rest specs)**

Define some channel selections. Usage:

`(define-channel-selections devtype <entry> <entry>…)`

Each entry is a list starting with the name of the channel set followed by pick-widget selection strings.

**make-selection-menu (menu-bar name accelerator pick display)**

Make a new selection menu. Make a selection menu into given ‘menu-bar’. This menu controls pick widget ‘pick’ and display ‘display’. These may be either names or widgets.

**current-selection-name**

This variable contains the name of latest selection made. Note that using this directly works only if you have only one selection menu.

**selection-default-device**

This variable contains the default device for selection menus. This can be set in the environment to choose the default set.

**use-selection (device-type name &optional pick display)**

Activate a view selection This routine sets the selection of pick widget pick to a predefined set of channels. It takes the name (string) of the desired device type and the name of the selection. The selection must be earlier defined by using define-channel-selections. Command uses given pick and display widgets.

The pick widget defaults to widget 'pick' and the display defaults widget 'display'.

## Module revision history

**Version 2.0.2 (2008-03-28)**

Technical repackaging, no changes.

**Version 2.0.1 (not released)**

Moved std-selections.lsp to sub-directory ‘config’ since it really is more like a configuration file.

channel-selection.lsp: Set default device to Vectorview.

**Version 2.0.0**

std-selections.lsp: This now contains only requires of NM122 and Vectorview selections. Bumped to version 2.0 to denote that this is now official Vectorview version.

channel-selection.lsp (use-pick-selection): Display can be now nil: allows calling without any specific display widget. Added a hook function, to allow later definitions of actions to be performed when a selection is made. Useful to update name labels etc.

std-selections-vv.lsp, std-selections-204.lsp: Corrected channel pair ordering from prototype order to production version. std-selections-204.lsp: Removed definition for nm122. Corrected error in left-parietal.

**Version 1.0**

Initial version

## 1.3 CLASSIFY-BY-TABLE 1.2

**General Description**

Classify-by-table provides an utility that allows relatively easy way to classify epochs. It is intended to be used with the off-line averager utility.

The module consists of a single LISP routine (classify-by-table) and an example classification table. The routine attempts to classify an epoch at the time point given as a parameter.

This documentation assumes that the user knows the basic operations and scripting needed for using the signal processor program. For details of such operations refer to *Graph Users Guide.*

**Installing the module**

This module is available in graph-2.94 and later, or it can be provided separately. It is dependent on module “trigger-utils”, which must be present. You can test its presence by trying command `(require "trigger-utils")`, and see if it succeeds or fails. The module itself is also loaded with `(require "classify-by-table").`

**Setup files provided**

No example setup files are provided.

**User interface**

Variable *event-tag-table-name* is added into the **Parameters** menu parameter class classification.

**Technical details**

The routine has an interface that matches the calling convention of the classifier function of the off-line averager. It can thus be used as such. It takes only a single parameter, which is the trigger time (in x-units, typically seconds) of the epoch. All other settings are given indirectly through a table on the disk.

**How the averager uses the classifier**

When the averager parameter *average-classify* is set, the value is assumed to be a name of a function. When a trigger point is found, this classifier function is called with the trigger time, and the return value is used to tag the epoch. The average-widgets contain a field “tag”, which defines the event tag to which it responds. If the tag does not match, no averaging is performed. So, by writing a suitable classifier one can easily average epochs belonging to some particular class.

The tag can be any LISP object, but using strings is preferred.

**What classify-by-table does**

The operation of the routine is defined by a table file, which is a LISP file having fixed structure. The file contains a list of definitions; one for each class. These definitions contain the tag and information that is used to define the class.

When the routine is called, it examines the trigger channels for a trigger combination. In order to make this work, the setup must contain a channel which carries the combination information. In systems that do not provide binary coded trigger combination channels (dacq-3.3 and earlier), easiest way to create such a channel is to use encode-widget.

**Syntax of the event tag table**

The following paragraphs describe the format of the table. It is actually read indirectly as a lisp object, so it resembles a function definition.

The file should contain one list starting with symbol event-tag-table. That should be followed by the entries, which also are lists. The first element of the entry defines the “tag”. Second element describes the trigger combination that must be present on the trigger line. Here the combination is assumed to be encoded into the height of the trigger signal as the binary combination of the trigger bits.

Rest of the elements define extra conditions that must prevail. Currently there are provided two combinations:

`(signal <`*widget name*`> <`*channel number*`> <`*begin*`> <`*end*`>)`

`(no-signal <`*widget name*`> <`*channel number*`> <`*begin*`> <`*end*`>)`

These definitions cause function to examine the signals at specified widget / channel / time range. The channel count starts from 1 and the epoch is relative to the trigger point.

Below is an example file:

```
;;
;; This is a sample event tag table
;; Comments can be entered in normal lisp fashion.
;;

(event-tag-table

 ;; Most specific entries first
 ;; Note that the second element is trigger combination,
 ;; but the third element in 'signal' is channel number.

 ("S3ab" 4   (signal "stim" 1 1.0 2.0)
             (signal "stim" 2 -1.0 -0.05))
 ("S3a"  4  (signal "stim" 1 1.0 2.0))
 ("S3b"  4  (signal "stim" 2 -1.0 -0.05))
 ("S1"  1)
 ("S2"  2)
 ("S3"  4)
 ("S4"  8)
 ("S5"  16)
 ("S6"  32)
 ("other"  :any)

 )
```

The first three classes are conditioned by presence of signals in the outputs of widget "signal". The records are examined one by one until a match is found. Therefore ones need to put the most specific conditions first.

One could imagine, that the example, the channels of "stim" contains e.g. thresholded response signals. Then the event is classified as "S3ab" if there is a response both before and after the event. Second class would be the epochs when the response is only either before or after the trigger point.

The entries from fourth to ninth are to classify the ordinary triggers where they are delivered to one trigger line at a time. The last entry matches to anything, and is recommended.

**Developing a setup**

In order to have a functional system, we need a setup, which has trigger combinations available. You can start from a simple basic averaging setup and replace the "threshold" widget by a "encode" widget. The output of the encode has the trigger channels combinations as data values. If you analyze Neuromag 122 data, and you need to condition the tagging to specific trigger lines, you also need to have the threshold widget present to create clean singe trigger lines. In Vectorview this is not necessary, since the triggers are not analog signals. The triggers used in conditioning can be created from any source by thresholding properly (maybe with some preprocessing).

Verify that the setup works with these simple settings. It should average everything into one category. Then add `classify-by-table` into the **Parameters>averager>average-classify**, and change the tag in the average-widget to match something that is known to exist on the data. Then try again.

The name of the event tag table is setup specific, and is set through the Parameters menu. The file is not reloaded automatically if you make changes to it. After changes, you should issue

```
(load "complete filename")
```

or

```
(load* filename-with-no-.lsp)
```

The second simpler forms requires that the directory where the table is, is in the search paths of the program.

## Debugging

There is a handy way to debug the working of the classifier. Since it simply takes one argument, it can be easily tested on command line. Simply call the function with some time:

```
(classify-by-table <time>)
```

and see what is the opinion of the routine. Even handier is use

```
(classify-by-table (first (x-selection)).
```

Then you can use mouse to point the time points.

## Module revision history

Changes to the module:

**Version 1.2.1**

Technical repackaging, no changes to the code.

**Version 1.2.0**

classify-by-table.lsp: Removed the dependency to module trigger-utils by copying the get-stimulus-combination into the file (as cbt-get-stimulus-combination).

## 1.4 PCA

**General Description**

The package “pca” provides code for applying principal component analysis on widget data. It exports function `pca-on-widget` which does the task. It also co-operates with the module SSP so, that the menu **Actions** in the SSP dialog will contain buttons for easy access to PCA vectors.

This package is capable to operate on data in any G-widget, so that it does not depend on the setup. However, it creates a new matrix-source widget “pca-fields”, where it stores the PCA vectors representing the “field” expansion.

**Installing the module**

The pca module is supplied with the Graph version 2.92 and later in the standard module library /neuro/lisp/graph-<version>/lib, and to load the code into the system you need to give lisp command

```
(require "pca").
```

After this command the functionality of the module should be available. Note however, that the setup files, that are used to save Graph state, are capable to load the code automatically if the module was present when the setup was made. Is such case loading the setup will load the pca code too, and the require command is unnecessary.

**pca-on-widget**

The function `pca-on-widget` has the following syntax:

```
(pca-on-widget <widget> <start-time> <end-time>)
```

The parameter <*widget*> defines the source from which the data should be taken into the analysis. It can either be a G-widget or its name (string). The parameters <*start-time*> and <*end-time*> define the data span on which the analysis is performed. They are expressed in real x-units, typically in seconds.

Function calculates the singular vectors of the data matrix corresponding to field shapes. The singular vectors representing the time variation of the fields are not calculated. If you need to know them, use the SSP module with the PCA field vectors to display the field amplitudes as a function of time.

Two variables are exported. They contain the results of the calculation:

- `pca-fields`: This variable is set to contain a matrix holding the field vectors.
- `pca-svalues`: This variable is set to contain the squares of the singular values of the data matrix.

**Add PCA fields menu**

When PCA module and the SSP module are present simultaneously, a menu button **Add PCA fields** is added to the **Actions** menu of the SSP dialog. See “ssp” on page 18. This button opens a submenu that contains buttons to add 1, 2, 3, 5, 8, or 12 first “singular fields” into the vector pool of the SSP dialog.

## 1.5 SSP

**General Description**

The package “ssp” provides a handy interface for managing vectors for a suppressor widget. Its basic purpose is to make easy to project away external disturbances from measured signal. It can be used for any other projecting problem as well. Vectors can be easily assembled both into a suppressor widget, so that requested space can be removed, and into a linear widget to show the time dependence of each vector, providing effectively a linear “multi source model”.

The subspace that we are interested in, is defined by a set of vectors or matrices. These vectors define the subspace that they span. Vectors need not to be orthogonal and can have arbitrary length. The lengths have no effect in projections, but in the “multi source model” they are taken into account. Matrices are handled the same way as a list of column vectors. They provide, however, a handy way to group related vectors, so that they can be used as a single object.

When loaded, the module creates a suppressor widget called “ssp”, unless it already exists. All operations apply automatically to this widget, so to use the module in sensible way, it should be used with a setup that contains the “ssp” widget in reasonable environment. The data connected into the “ssp” widget should contain only the channels that used in the projections. Typically this means that all MEG channels are coupled in it.

When selected space is activated, a related matrix is installed automatically to two other widgets too, provided that they exist. They should be linear widgets. Into widget “removed” code assembles a set of orthonormal basis vectors that span the defined subspace. Use how you like. In widget “timevar” it installs the solution matrix to calculate the amplitudes related to the original space definition vectors, so that multiplying the vectors with these amplitudes would produce vector that matches best (in sense of an euclidean norm) to the input vector. So this widget has in its output the time variations of the sources in a “multi source model” where sources are defined by given field shapes. These last two widgets are not created automatically. If they are needed, they should be created by a suitable setup.

**Installing the module**

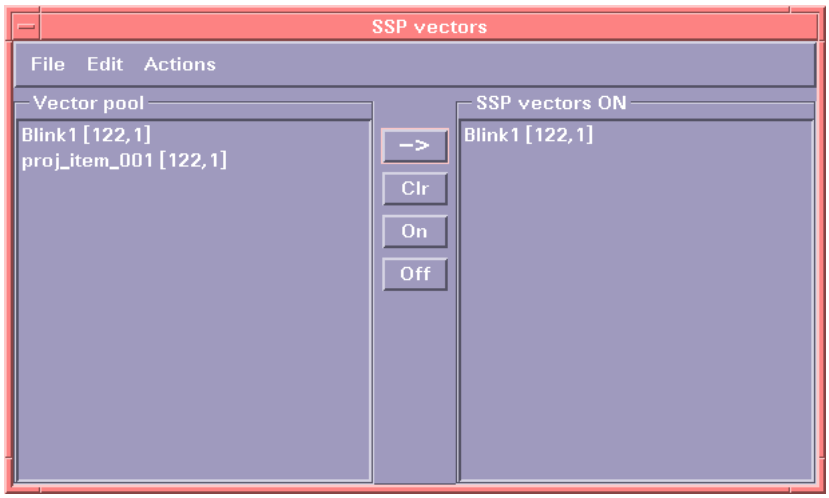
This module is available in standard module library of Graph, so that you can load it by command

```
(require "ssp")
```

Note that setup files are also capable to load this module, so that if your current setup provides the module, you don’t need to enter the command.

### Using ssp dialog

When the module is loaded, it creates button **SSP dialog...** in the **Commands** menu. Selecting this menu button will pop up a dialog for projection manipulation. To close the dialog, use menu button **Close** in the **File** menu of the SSP dialog. A picture of the dialog is shown below.




On top of the dialog there is a menu bar that allows user do execute some commands. Below the menu bar are two selection boxes with some buttons in between. The left box, the “Vector pool”, lists a set of space definitions that can be easily used in the operations, but which are not “active”, that is, being used to define the subspace used. On the right side, in the “SSP vectors”, are represented the definitions that are currently being used. Each space definition has a name. The text in the selection boxes state the names and sizes of the matrices that define a subspace. Note that the length of the definition (column) vectors must match the number of input channels fed into “ssp” widget. Otherwise the widgets do nothing. You can see the description of the pool vectors (if any) by double clicking the name of the projection. To enter a description, see “Edit menu” on page 21.

### Activating vectors

One can activate a definition in the vector pool by selecting it and the pressing the button between the selection boxes that has a right arrow. Selecting a definition is achieved by clicking its name in the selection box. You may also select multiple entries at a time by dragging the cursor, or by pressing SHIFTkey while selecting. Pressing the SHIFTkey makes selection state to toggle when an item is clicked. Activating an item causes it to appear on the active vector selection box, and the new set of active vectors is installed to widgets.

**Clearing active vectors**

To clear the current active vectors, press the **Clr** button. Your confirmation for the clearing will be asked before the actual deletion. Note that normally you don't lose the vectors if you clear the active vectors, since when the pool vectors are activated, they are copied rather than moved to the active vectors. You loose definitions only when you delete pool vectors.

**Deactivating projections temporarily**

The transformations can be set temporarily off by pressing the **Off** button between the selection boxes. The active vectors are kept in the "ssp vectors", but the title on top of the box changes from "SSP vectors ON" to "SSP vectors OFF", to show that the vectors are not used just now. To activate the vectors again, press the **On** button.

**Description of the menus**

The **File** menu contains commands that are related to saving and loading projections. In addition it contains the **Close** button which closes the dialog. The **Edit** menu contains commands that are related to manipulating the projections and the **Actions** menu contains mainly commands that are used to create new vectors. Note that other modules may add buttons into this menu. These additions are not documented here. (See for example "pca" on page 16). Following paragraphs describe what each menu button does.

**File menu**

**Default directory**

To set the default directory for the projection files, use menu button **Default directory** in the **File** menu. This pops up a file selection box for easy setting of the directory. The directory is defined by user variable `ssp-default-directory`, so that the value is saved into setup files, and you can alter the value also using lisp commands.

**Save**

To save selected or all (if nothing is selected) pool vectors, use the **Save** button in the **File** menu. This will pop up a file selection box, and you should enter into in the file name where you wish to save the vectors. The file name must have either extension ".fif" or ".lsp". The extension is used to decide in which format the files are saved. Normally it is best to use fiff-files, since both Xfit and Graph can read them.

**Load**

To load a file containing projections, select **Load** in the **File** menu of the ssp dialog. A file selection box will pop up. Use this box to select which file you want to load. Program is capable to load projections both from a LISP file and from a fiff-file.

**Close**

Activating this button will close the ssp dialog.

**Edit menu**

**Delete selected**

Activating this button will delete all selected entries from both vector pool and active vectors. It will ask confirmations before deleting. **When you select items to be deleted, check that there are no items selected on the other selection box**.

**Add to SSP**

When this button is pressed, the currently selected vectors in vector pool are activated, that is, they are added into the current projections. The vectors will also be copied to the selection box showing the active vectors.

**Clear SSP**

Selecting this button will clear all active ssp vectors.

**Explode**

When this button is activated all selected projections in the pool are “exploded”. This means, that if the dimension of the projection space is larger than 1, which means that the definition is a matrix rather than a vector, this matrix is replaced by set of separate vectors that are the column vectors of the matrix. For the reverse operation, see the **Implode** button.

**Implode**

Activating this button will take all selected spaces in vector pool, and replaces them with a single matrix, that contains all the basis vectors in the selected spaces. The reverse operation is available using the **Explode** button.

**Edit name**

Activation this button will open a dialog that enables you to give a new name to the selected vector. Enter the new name into the text box and press **OK**.

**Edit description**

Activation this button will pop up a simple text editor with the current description text of the vector. You can change the text freely. The description need not be a single line. Remember that the vector definition and the comment are lost unless you save the entry into a file.

**Actions menu**

This menu is reserved for all kinds of commands to obtain new vectors. By default it contains only one button **Pick selected field**, but other modules may create more buttons into it.

**Pick selected field**

Before activation this button, you should select a time point from one of the displays. The length of the time should zero, that is click a single line into the display. For making time selections, see “Selecting time intervals” in “Signal Processor User’s Manual”. When the button is activated, a sample vector is taken from the current output of the “ssp” widget, at the time specified by the current selection. This vector will be added into both pool and active vectors, and if the projections are temporarily off, they will be turned on again. Before adding the vector. User will be prompted to give a name to this new vector. Vector names need not to be unique, but having multiple vectors with same name is likely to case confusion.

## Module revision history

**Version 1.7.0**

ssp-remove-space: Moved list refresh to the beginning to ensure the list is updated also when errors occur.

ssp.lsp (ssp-save-file): Uses now add_proj_namelist to add the channel names. Requires that the utility exists. Wrapped add_proj_namelist with (utility program) to make it installation root independent.

(ssp-load-file, ssp-vector-dir-template): Handle missing /neuro/ssp directory cleanly.

## 1.6 STD-SELECTIONS

**General Description**

The package “std-selections” provides an interface for easy selection of standard channel sets to the main display. It adds a new submenu **Selections** into the **Displays** menu. This menu contains a button for each predefined selection. Note that the menu is a tear-off menu, so that it can be lifted on the workspace for handy operation.

This package uses a pick widget called “pick” to select the channels, and links it automatically to widget “display”. This means that it is only useful with setups that contain these two widgets in suitable places. The input to the pick widget can be freely chosen. However the output of the widget is linked into widget “display” every time the selection is changed. The basic setup provided in the examples directory provides an example setup, that works with the module.

**Installing the module**

The std-selections module is supplied with the Graph version 2.92 and later in the standard module library /neuro/lisp/graph-<version>/lib, and to load the code into the system you need to give LISP command

```
(require "std-selections").
```

After this command the functionality of the module should be available. Note however, that the setup files, that are used to save Graph state, are capable to load the code automatically if the module was present when the setup was made. Is such case, loading the setup will load the module, and the require command is not necessary.

**Menu description**

This module uses module channel-selection, std-selection-nm122, std-selection-vv to create the menu. See the reference of these modules for details of the selections.

## 1.7 STD-SELECTIONS-NM122

**General Description**

The package “std-selections-nm122” provides definitions of standard channel sets suitable to be used with Neuromag122 device. It contains essentially data and provides no functions or variables. See also channel-selection.lsp and std-selections.lsp.

**Installing the module**

The module is supplied with the Graph version 2.93 and later in the standard module library /neuro/lisp/graph-<version>/lib, and to load the code into the system you need to give LISP command

```
(require "std-selections-nm122").
```

After this command the functionality of the module should be available. Note however, that the setup files, that are used to save Graph state, are capable to load the code automatically if the module was present when the setup was made. Is such case, loading the setup will load the module, and the require command is not necessary.

**Selections**

The following channel definitions apply to channel names. This means that the channel numbering presented here starts from1, not from 0 as the channel location numbers in selectors.

**vertex**

MEG channels: 13 14 15 16 49 50 51 52 53 54 61 62 63 64 65 66 73 74 89 90 99 100 117 118 119 120 121 122.

**left-temporal**

MEG channels: 43 44 45 46 47 48 49 50 55 56 57 58 59 60 61 62 67 68 69 70 71 72 73 74.

**right-temporal**

MEG channels: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 111 112 113 114 115 116 117 118 119 120.

**left-parietal**

MEG channels: 47 48 49 50 51 52 59 60 61 62 63 64 71 72 73 74 79 80 87 88 89 90.

**right-parietal-channels**

MEG channels: 5 6 11 12 13 14 15 16 97 98 99 100 109 110 115 116 117 118 119 120 121 122

**occipital-channels**

MEG channels: 75 76 77 78 79 80 81 82 83 84 85 86 91 92 93 94 95 96 101 102 103 104 105 106 107 108 109 110.

**frontal**

MEG channels:17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42.

**EEG-1**

EEG channels: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16.

**EEG-2**

EEG channels: 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32.

**Rack-A**

MEG channels: 7 8 3 4 1 2 9 10 5 6 29 30 25 26 21 22 17 18 13 14 27 28 23 24 19 20 15 16 11 12.

**Rack-B**

MEG channels: 32 31 36 35 40 39 44 43 48 47 34 33 38 37 42 41 46 45 50 49 52 51 56 55 60 59 54 53 58 57.

**Rack-C**

MEG channels: 66 65 70 69 62 61 64 63 68 67 74 73 78 77 82 81 86 85 90 89 72 71 76 75 80 79 84 83 88 87.

**Rack-D**

MEG channels: 107 108 103 104 99 100 95 96 91 92 109 110 105 106 101 102 97 98 93 94 117 118 113 114 119 120 115 116 111 112 121 122

## 1.8 STD-SELECTIONS-VV

**General Description**

The package “std-selections-vv” provides definitions for standard channel sets suitable to be used with Elekta Neuromag and Vectorview devices. It contains essentially data and provides no functions or variables. See also channel-selection.lsp and std-selections.lsp.

**Installing the module**

The module is supplied with the Graph version 2.93 and later in the standard module library /neuro/lisp/graph-<version>/lib, and to load the code into the system you need to give lisp command

(require "std-selections-vv").

After this command the functionality of the module should be available. Note however, that the setup files, that are used to save Graph state, are capable to load the code automatically if the module was present when the setup was made. Is such case, loading the setup will load the module, and the require command is not necessary.

**Selections**

The following channel definitions apply to channel names.

**left-temporal**

MEG channels 223 222 212 213 133 132 112 113 233 232 242 243 1513 1512 142 143 1623 1622 1612 1613 1523 1522 1542 1543 1532 1533.

**right-temporal**

MEG channels [1313 1312 1322 1323 1443 1442 1422 1423 1343 1342 1332 1333 2613 2612 1432 1433 2413 2412 2422 2423 2643 2642 2622 2623 2632 2633.

**left-parietal**

MEG channels 632 633 423 422 412 413 712 713 433 432 442 443 742 743 1823 1822 1812 1813 1833 1832 1842 1843 1633 1632 2012 2013.

**right-parietal**

MEG channels 1042 1043 1113 1112 1122 1123 722 723 1143 1142 1132 1133 732 733 2213 2212 2222 2223 2243 2242 2232 2233 2443 2442 2022 2023.

**left-occipital**

MEG channels 2043 2042 1913 1912 2112 2113 1922 1923 1942 1943 1642 1643 1933 1932 1733 1732 1723 1722 2143 2142 1742 1743 1712 1713.

**right-occipital**

MEG channels 2033 2032 2313 2312 2342 2343 2322 2323 2432 2433 2123 2122 2333 2332 2513 2512 2523 2522 2132 2133 2542 2543 2532 2533.

**left-frontal**

MEG channels 522 523 512 513 312 313 342 343 123 122 823 822 533 532 543 542 323 322 612 613 332 333 623 622 643 642.

**right-frontal**

MEG channels 812 813 912 913 922 923 1212 1213 1222 1223 1413 1412 943 942 933 932 1233 1232 1012 1013 1022 1023 1242 1243 1033 1032.

**left-temporal-mags**

MEG channels 221 211 131 111 231 241 1511 141 1621 1611 1521 1541 1531.

**right-temporal-mags**

MEG channels 1311 1321 1441 1421 1341 1331 2611 1431 2411 2421 2641 2621 2631.

**left-parietal-mags**

MEG channels 631 421 411 711 431 441 741 1821 1811 1831 1841 1631 2011.

**right-parietal-mags**

MEG channels [1041 1111 1121 721 1141 1131 731 2211 2221 2241 2231 2441 2021.

**left-occipital-mags**

MEG channels 2041 1911 2111 1921 1941 1641 1931 1731 1721 2141 1741 1711

**right-occipital-mags**

MEG channels 2031 2311 2341 2321 2431 2121 2331 2511 2521 2131 2541 2531.

**left-frontal-mags**

MEG channels 521 511 311 341 121 821 531 541 321 611 331 621 641.

**right-frontal-mags**

MEG channels 811 911 921 1211 1221 1411 941 931 1231 1011 1021 1241 1031.

## 1.9 TEMPLATES

**General Description**

The package “templates” provides an interface for managing a template filter. The template filter it uses must have name “template”. Loading the module will also create the widget if it does not exists or has wrong type. It provides a menu for easy creation and manipulation of the templates but it does not define what to do with the template output itself. For evaluation of this module, a setup called “template-search” is provided in the examples directory. The menu provided is created as a submenu of the Commands menu. Note that the menu is a tear-off menu, so it can be lifted on the desktop. See “Using menus” in “Graph User’s Manual’.

**Installing the module**

This module is available in standard module library of Graph, so that you can load it by command

```
(require "templates")
```

Note that setup files are also capable to load this module, so that if your current setup provides the module, you don’t need to enter the command.

**User interface**

Descriptions of the actions provided by the menu are listed below. Note also that the **Parameter** menu will contain an editor for several user parameters defined by this module. The name of the parameter group is “Template search”.

**Set Template**

This button provides a submenu that contains several ways to set the active template in the template filter “template”. In the beginning there are two buttons in it, namely **From selection** and **From average**. The buttons are described below.

**From selection**

Before selecting this button, you should select a time span from a the display. This selection defines the template we are going to install. Activating this button will take data from widget defined by user parameter template-source, and installs it into the template widget. Typically this user parameter should contain the name of a widget that carries filtered MEG data.

**From average**

Activating this button will take the current average (which has to be stored in widget “average”) and installs it into template widget. You can also use only a portion of the average data by selecting part of it in a display that is connected to it

### Saved template

If a template has been saved during a session, a new button to load and activate this template is created into this menu. These buttons have the name of the template on them.

### Save template

Activating this button will save the current template. That is the template that was last activated. Program will pop up a file selection box that allows you to save the template in fiff-format file. Note, however, that the extension used with templates is not “fif” but “tpl”. The extension will be added automatically if you don’t provide it explicitly. When you save a template, a new button will be added into the **Set template** menu. This button will automatically load and install the template.

### Load template

Selecting this button you will pop up a file selection box that enables you to load and activate a template. Note that the template files have name extension “tpl”.

### Set search point

Before using this button, set a zero length x-selection into a suitable place. Activation the button will set this time into the search pointer for template searches. Next search will then start from this point.

### Next match

This action will scan the channel defined by user variable template-output, until it finds a positive pulse. It sets the search pointer to the time of the first local maximum after the start of the pulse. It also alters the x-selection and the current view of the main display so that the trigger point is visible and selected.

### Threshold up 10%

This button multiplies the current threshold value in unary widget defined by user variable template-threshold by 1.1.

### Threshold up 2x

This button multiplies the current threshold value in unary widget defined by user variable template-threshold by 2.0.

### Threshold down 10%

This button multiplies the current threshold value in unary widget defined by user variable template-threshold by 0.9.

### Threshold down 2x

This button multiplies the current threshold value in unary widget defined by user variable template-threshold by 0.5.

**Show current**

Activation this button will select using x-selection of the main display the time span of the origin of the current template. It will also move the view to show it.

## 1.10 TRIGGER-UTILS 1.9

**General Description**

The package “trigger-utils” provides utilities for managing events. It provides basic functions to create events, access their properties, and to scan widget data to find events. Using event manipulation routines in this module are recommended in order to keep the internal representation of event descriptions coherent in all modules. This is purely scripting utility. No user interfaces are provided.

**Installing the module**

This module is available in standard module library of Graph, so that you can load it by command

```
(require "trigger-utils")
```

Note that setup files are also capable of loading this module, so that if your current setup provides the module, you don’t need to enter the command.

**Lisp interface**

This section describes briefly what functionality is available. For detailed information on the functions, use on-line descriptions of the functions.

Events are currently described by association lists, that is, lists that contain property-value pairs. However, the event data should not be accessed directly. Using the routines here allows later changes to the event objects without rewriting the code that uses the event data. For basic creation and manipulation of events, the following functions are provided:

- **make-event**: Create an event.
- **event-time**: Return time of an event.
- **event-class**: Return event class of an event.
- **event-length**: Return length of an event (if any).
- **event-set-property**: Set an arbitrary property of an event.
- **event-get-property**: Return an arbitrary property of an event.

To scan data and create event objects from events in the data, following utilities are available. First, following variables define the default values that are used when no explicit value is given to the functions.

- **default-trigger-source**: Trigger channel to be scanned for events. In normal averaging applications this is usually a channel which have the triggers encoded as numbers, but it can also contain analog signals, if the triggering modes are set suitably.
- **default-classifier**: Default classifier function. Classifiers functions are used to classify events. They are similar to ones used in the averager and they take a trigger time and return the class of the event.

Second, following functions can be used to search events from data:

- **find-event**: Scan data until a particular kind of an event occurs. Return an event object describing the event. This routine has various options defining what events are included, and how triggers are handled.
- **map-events**: Scan events in a given time span and apply a given function to each of the events.

Module contains also a set of low level functions that are used by the above described functions and may be useful in other contexts too:

- **classify-event**: Classifies signals at particular time moment to deduce event class of an event. This relies completely on given or default classifier.
- **channel-exists-p**: Check that given widget exists and that given channel number is within existing channels of that widget.
- **time-span-available-p**: Check that given time span is available from a given widget.
- **signal-present**: Check if the signal in given widget/channel/time-span is nonzero.
- **get-stimulus-combination**: Get data value from given/default trigger channel at sample which follows given time.

**Module revision history:**

Changes to the module:

**trigger-utils-1.9**

- Changed the calling convention in map-events.
- Modified the also the way find-event behaves when no tags are defined.

**trigger-utils-1.8**

- Modified event structure. Events now can contain properties which are dynamically defined.

**trigger-utils-1.6**

- Added re-trigger option.
- Added get-length key to find out the length of the trigger.
- Routine make-event now accepts length of the event.

**trigger-utils-1.5**

- Added context variable to map-events. This allows recursive calls.
- Map-events now shows the event tags in working dialog.
- Classifier can be passed to map-events.

**trigger-utils-1.4**

Minor internal modifications.

**trigger-utils-1.3**

Minor internal modifications.

**trigger-utils-1.2**

- Added routine time-span-available-p
- Added key polarity to map-events

**trigger-utils-1.1**

Initial version.

## 1.11 XFIT 1.5

**General Description**

The package “xfit” provides an interface for managing a slave Xfit program. It provides a menu to make data transfers from Graph to Xfit easy. Note that the **xfit** menu is a tear-off menu, so it can be lifted on the desktop.

**Installing the module**

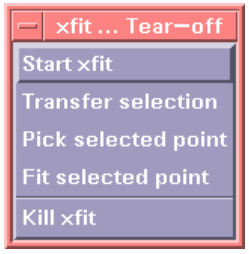
This module is available in standard module library of Graph, so that you can load it by command

```
(require "xfit")
```

Note that setup files are also capable to load this module, so that if your current setup provides the module, you don’t need to enter the command.

**User interface**

When loaded, the this module creates a submenu called **xfit** into the **Displays** menu. The actions performed by the buttons in the **xfit** menu are described below.




**Start Xfit**

This button starts a Xfit program for viewing data, unless it is already started. Only one Xfit can be managed using the menu.

**Transfer selection**

Before using this button, user should select some time interval on some of the displays. See “Selecting time intervals” in the “Signal Processor Users Manual”. Activating this button will transfer data in widget defined by the user variable default-data-source within the selected time span to the Xfit program. It also performs an autoscale operation.

**Pick selected point**

Before using this button, user should select a single time point on some of the displays. Activating this button will transfer a range of data around the selected point into the Xfit process, and Xfit is told to select the same time point. The length of the data transferred is defined by user variable xfit-region-width that is available also through **Parameters** menu button **Misc defaults**. If the selected point is within data that is currently transferred into Xfit, no data is transferred, only the new time point is picked.

**Fit selected point**

Before using this button, user should select a single time point on some of the displays. Activating this button will transfer a range of data around the selected point into the Xfit process, and Xfit is commanded

to fit a dipole to this particular time point. The details how the fit is actually done depend on settings of the Xfit program. The length of the data transferred is defined by user variable xfit-region-width that is available also through **Parameters** menu button **Misc defaults**. If the selected point is within data that is currently transferred into Xfit, no data is transferred, only a fitting is performed.

**Kill xfit**

Activating this button will stop the slave Xfit program.

**Lisp interface**

In addition to the **xfit** menu, this module exports several functions that are useful in operating with Xfit. Here is a list of functions and parameters currently provided. For detailed information on the functions, use the on line descriptions of the functions.

- `kill-xfit`: Function that stops the Xfit process.
- `xfit`: Function that starts a xfit process.

  Variable that contains the stream used in communication with the Xfit used in menus.
- `xfit-command`: Function that sends a command to a Xfit.
- `xfit-fit-selected-point`: Function that fits current selected time point using Xfit. See menu button **Fit selected point** above.
- xfit-fit-span: Function that makes multiple single dipole fits over the defined time span. Channels used in fits can also be selected.
- `xfit-load`: Function that loads a file into a Xfit.
- `xfit-menu`: Function that creates the **xfit** menu.

  Variable containing the **xfit** menu.
- `xfit-region-width`: Variable defining the length of the data transferred into Xfit when functions `xfit-pick-selected-point` and `xfit-fit-selected-point` are used.
- `xfit-program`: Variable containing the start-up command used to start a Xfit process.
- `xfit-pick-selected-point`: Function that shows current selected time point using Xfit. See menu button **Pick selected point above**.
- `xfit-timeout`: Variable defining the timeout value of the communication link.
- `xfit-transfer-data`: Function that shows arbitrary data using a Xfit.

Manipulating xfit channel selections is unfortunately somewhat complicated due to usage pattern of Xift, which is oriented towards manual operations. Selections to use must be first defined and the taken into use. Further complication is that selection are done using logical channels.

Following functions are available

- `xfit-selection-clear`: Clear Xfit channels selection.

- `xfit-selection-add`: Add selection into collection of channel definitions.
- `xfit-selection-use`: Use a predefined channel definition, or define and use a new one.
- `xfit-selection-omit`: Make Xfit to forget one selection definition.
- `xfit-selection-set`: Set xfit channel selection. As a side effect, the selection is added to the selection definitions in Xfit.
- `xfit-loadsel`: Load a selection from a file.

**Module revision history**

Summary of changes

**xfit-1.5**

Maintenance changes to remove explicit references to directory /neuro.

**xfit-1.4**

Added routines to manage xfit channel selections.

Fit command did not function if selection was not restricted.

**xfit-1.3**

Fitting can be restricted to use only a set of channels.

Current working directory of graph is transferred to xfit after data transfer. Allows easier saving to the source directory.

## 1.12 XPLOTTER 1.2

**General Description**

The package “xplotter” provides an interface for managing slave Xplotter programs. It provides a menu to make data transfers from Graph to Xplotter easy. It also provides functions that enable a command widget to behave as if it was a display widget having Xplotter functionality. Note that the xplotter menu is a tear-off menu, so it can be lifted on the desktop.

**Installing the module**

This module is available in standard module library of Graph, so that you can load it by command

```
(require "xplotter")
```

Note that setup files are also capable to load this module, so that if your current setup provides the module, you don’t need to enter the command.

**User interface**

When loaded, the this module creates a submenu called **xplotter** into the **Displays** menu. The actions performed by the buttons in the **xplotter** menu are described below.

**Start xplotter**

This button starts a Xplotter program for viewing data, unless it is already started. So only one Xplotter can be managed using the menu.

**Show selection**

Before using this button, user should select some time interval on some of the displays. See “Selecting time intervals” in “Signal Processor User’s Guide”. Activating this button will transfer data in widget defined by the user variable default-data-source within the selected time span to the Xplotter program. It also performs an autoscale operation.

**Kill xplotter**

Activating this button will stop the slave Xplotter program used to view data by this menu.

**Lisp interface**

In addition to the **xplotter** menu, this module exports several functions that are useful in operating with Xplotters. This module is for example needed by the command widgets created by the **Xplotter** button in the **Widgets** menu. Here is a list of functions and parameters currently provided. For detailed information on the functions, use on line descriptions of the functions.

- `kill-xplotter`: Function that stops a Xplotter process.

- `xplotter`: Function that starts a xplotter process.

  Variable that contains the stream used in communication with the Xplotter used in menus.
- `xplotter-command`: Function that sends a command to a Xplotter.
- `xplotter-load`: Function that loads a file into a Xplotter. Both replacing the existing data and overlay are supported.
- `xplotter-menu`: Function that creates the **xplotter** menu.

  Variable containing the **xplotter** menu.
- `xplotter-program`: Variable containing the start-up command used to start a Xplotter process.
- `xplotter-show-data`: Function that shows arbitrary data using a Xplotter. Both replacing the existing data and overlay are supported.
- `xplotter-show-selection`: Function that shows current selection using default Xplotter.
- `xplotter-timeout`: Variable defining the timeout value of the communication link.

**Module revision history:** Changes to the module:

**xplotter-1.2**

Xplotter interface for loading data now includes option overlay. It allows overlaying old data instead of replacing it.


Elekta Neuromag Oy
Siltasaarenkatu 18-20
FI-00530 Helsinki, Finland
Tel: +358 9 756 2400
Fax: +358 9 756 24011
Web: www.elekta.com