EEGLAB Wikitorial

This tutorial was written by: Arnaud Delorme, and Scott Makeig

Many thanks to Hilit Serby, Nima Bigdely, and Toby Fernsler for additions and or edits. Thanks also to Payton Lin for capturing some images in earlier versions and to Micah Richert, Yannick Marchand, Elizabeth Milne, and Stefen Debener for their detailed comments. In addition, thanks to Devapratim Sarma for converting and updating the EEGLAB documentation to a WIKI.

Table of Contents

Getting Started **1**
What is EEGLAB? 1
Why EEGLAB? 1
Tutorial Outline 1
EEGLAB Overview 1
Distribution, documentation and support 2

I.Single subject data processing tutorial **3**

Chapter 01: Loading Data in EEGLAB **4**
Learning Matlab 4
Installing EEGLAB and tutorial files 4
Starting Matlab and EEGLAB 5
Opening an existing dataset 6
Editing event values 7
Sample experiment description 7
About this dataset 8
Scrolling through the data 8

Chapter 02: Channel Locations **13**
Importing channel location for the tutorial dataset 13

Chapter 03: Plotting Channel Spectra and Maps **16**
Plotting channel spectra and maps 16

Chapter 04: Preprocessing Tools **17**
Preprocessing tools 17

Chapter 05: Extracting Data Epochs **24**
Extracting epochs 24

Chapter 06: Data Averaging **26**
Data averaging 26

Chapter 07: Selecting Data Epochs and Comparing **33**
Selecting data epochs and plotting data averages 33

Chapter 08: Plotting ERP images **38**
Plotting ERP images 38

Chapter 09: Decomposing Data Using ICA **49**
Independent Component Analysis of EEG data 49

Chapter 10: Working with ICA components **59**
Rejecting data epochs by inspection using ICA 59
Plotting component spectra and maps 59
Plotting component ERPs 61
Plotting component ERP contributions 62
Component ERP-image plotting 64

Chapter 11: Time / Frequency decomposition **66**
Time/frequency decomposition 66

Chapter 12: Multiple Datasets **70**
Processing Multiple Datasets 70
Concluding remark on data tutorial 71

II.Multiple subject processing tutorial **72**

Chapter 01: Multiple Subject Proccessing Overview **73**
EEGLAB Studysets and Independent Component Clustering 73

Chapter 02: STUDY Creation **74**
Data organization 74

Table of Contents

Chapter 03: Working with STUDY designs **80**

Chapter 04: STUDY Data Visualization Tools **89**
Contents 89

Chapter 05: Component Clustering Tools **92**
Clustering outline 92

Chapter 06: Study Statistics and Visualization Options **108**
Contents 108

Chapter 07: EEGLAB Study Data Structures **114**
The STUDY structure 114

Chapter 08: Command line STUDY functions **124**
Creating a STUDY 124

III.Advanced Topics **128**

Chapter 01: Rejecting Artifacts **129**
Rejecting artifacts in continuous and epoched data 129
Rejection based on independent data components 141

Chapter 02: Writing EEGLAB Scripts **142**
Why write EEGLAB Matlab scripts? 142
Using dataset history to write EEGLAB scripts 142
Using EEGLAB session history to perform basic EEGLAB script writing 146
Basic scripting examples 149
Low level scripting 150

Chapter 03: Event Processing **154**
Event Processing in the EEGLAB GUI 154
Event processing from the Matlab command line 158

IV.Appendix **165**

A01: Importing Continuous and Epoched Data **166**
Importing continuous and epoched data 166

A02: Importing Event Epoch Info **175**
Importing event and epoch information 175
Exporting event information 181

A03: Importing Channel Locations **182**
Supported Electrode Data Formats 182
Importing channel location 182
Retrieving standardized channel locations 187
Importing measured 3-D channel locations information 188

A04: Exporting Data **192**
Supported Data Formats 192
Exporting data and ICA matrices 192

A05: Data Structures **195**
EEGLAB Data Structures 195

A06: Maximizing Memory **199**
Options to Maximize Memory and Disk 199

A07: Contributing to EEGLAB **201**

Open source policy **202**

Table of Contents

How to write EEGLAB functions **203**
The signal processing function 203
The associated pop_ function 203

How to write an EEGLAB plug-in **204**
The plugin function 204
Adding a sub-menu 204
Plug-ins and EEGLAB history 204
Plugin function examples 205
EEGLAB menu integration 206

Using EEGLAB GUI design functions **207**
Using the supergui function 207
Using the inputgui function 208
Advanced GUI design 210

A08: DIPFIT **211**
DIPFIT plug-in: Equivalent dipole source localization of independent components 211

A09: Using custom MRI from individual subjects **230**
Normalizing an MR head image to the MNI brain template 230

A10: MI-clust **234**
The MI-clust Plug-in: Clustering independent components using mutual information 234

A11: BESA (outdated) **243**
Equivalent dipole source localization of independent components using BESA 243
Visualizing dipole locations 246

A12: Quick Tutorial on Rejection **248**
Quick tutorial on rejecting data artifacts with EEGLAB 248

A13: Compiled EEGLAB **250**
Installation 250
Similarity between the compiled and the Matlab version 250
What is not possible to do using the compiled version 250
Frequently asked questions 250

Getting Started

What is EEGLAB?

EEGLAB is an interactive Matlab toolbox for processing continuous and event-related EEG, MEG and other electrophysiological data using independent component analysis (ICA), time/frequency analysis, and other methods including artifact rejection. First developed on Matlab 6.1 under Linux, EEGLAB runs on Matlab versions 6 (R13) and 7 (R14) under Linux or Unix, Windows, and Mac OS (Matlab v6 or current v7 recommended).

Why EEGLAB?

EEGLAB provides an interactive graphic user interface (gui) allowing users to flexibly and interactively process their high-density EEG and other dynamic brain data using independent component analysis (ICA) and/or time/frequency analysis (TFA), as well as standard averaging methods. EEGLAB also incorporates extensive tutorial and help windows, plus a command history function that eases users' transition from gui-based data exploration to building and running batch or custom data analysis scripts. EEGLAB offers a wealth of methods for visualizing and modeling event-related brain dynamics. For experienced Matlab users, EEGLAB offers a structured programming environment for storing, accessing, measuring, manipulating and visualizing event-related EEG, MEG, or other electrophysiological data. For creative research programmers and methods developers, EEGLAB offers an extensible, open-source platform through which they can share new methods with the world research community by contributing EEGLAB 'plug-in' functions that appear automatically in the EEGLAB menu. For example, EEGLAB is also being used for analysis of MEG data in several laboratories; EEGLAB plug-in functions might be created and released to perform specialized import/export, plotting and inverse source modeling for MEG data.

Tutorial Outline

This tutorial will demonstrate (hands-on) how to use EEGLAB to interactively preprocess, analyze, and visualize the dynamics of event-related EEG, MEG, or other electrophysiological data by operating on the tutorial EEG dataset *eeglab_data.set* which you may download here (4Mb). With this dataset, you should be able to reproduce the sample actions discussed in the tutorial and get the same (or equivalent) results as shown in the many results figures.

EEGLAB Overview

EEGLAB functions may be roughly divided into three layers designed to increase ease-of-use for different types of users. These three layers are described below.

The EEGLAB graphic user interface (gui)

The EEGLAB gui is designed to allow non-experienced Matlab users to apply advanced signal processing techniques to their data. However, more experienced users can also use the gui to save time in writing custom and/or batch analysis scripts in Matlab by incorporating menu shortcuts and EEGLAB history functions.

Text files containing event and epoch information can be imported via the EEGLAB menu. The user can also use the menu to import event and epoch information in any of several file formats (Presentation, Neuroscan, ASCII text file), or can read event marker information from the binary EEG data file (as in, e.g., EGI, Neuroscan, and Snapmaster data formats). The menu then allows users to review, edit or transform the event and epoch information. Event information can be used to extract data epochs from continuous EEG data, select epochs from EEG data epochs, or to sort data trials to create ERP-image plots (Jung et al., 1999; Makeig et al., 1999). EEGLAB also provides functions to compute and visualize epoch and event statistics.

New Matlab users may choose to interact only with the main EEGLAB window menu, first to import data into EEGLAB (in any of several supported formats), and then to call any of a large number of available data processing and visualization functions by selecting main-window menu items organized under seven headings:

- *File* menu functions read/load and store/save datasets and studysets.
- *Edit* menu functions allow editing a dataset, changing its properties, reviewing and modifying its event and channel information structures.
- *Tools* menu functions extract epochs from continuous data (or sub-epochs from data epochs), perform frequency filtering, baseline removal, and ICA, and can assist the user in performing semi-automated artifact data rejection based on a variety of statistical methods applied to activity in the raw electrode channels or their independent components.
- *Plot* menu functions allow users to visualize the data in a variety of formats, via (horizontally and vertically) scrolling displays or as trial (ERP), power spectrum, event-related time/frequency averages, etc. A large number of visualization functions are dedicated to the display and review of properties of scalp data channels and underlying independent data components. The user can make use of standard Matlab capabilities to edit, print, and/or save the resulting plots in a variety of formats.

- *Study* menu entries show the current *studyset* and which of its constituent *datasets* are currently loaded.
- *Datasets* menu entries list loaded *datasets*, and allows the user to swtich back and forth among them.
- *Help* menu functions allow users to call up documentation on EEGLAB functions and data structures, including function lists and scrolling function help messages.

EEGLAB structures and command history

EEGLAB uses a single structure (*EEG*) to store data, acquisition parameters, events, channel locations, and epoch information as an EEGLAB dataset. This structure can also be accessed directly from the Matlab command line. The *EEG* structure contains two key sub-structures: *EEG.chanlocs*, holding channel locations, and *EEG.event* storing dataset event information. In EEGLAB v5.0 (March, 2006), a new superordinate structure, the *STUDY* has been introduced to allow automated processing of a group of *EEG* datasets. The first such facility introduced is a set of study functions to perform and evaluate clustering of similar independent data components across subjects, conditions, and sessions.

Intermediate level users may first use the menu to perform a series of data loading, processing and visualization functions, and then may take advantage of the EEGLAB command history functions to easily produce batch scripts for processing similar data sets. Every EEGLAB menu item calls a Matlab function that may also be called from the Matlab command line. These interactive functions, called "pop" functions, work in two modes. Called without (or in some cases with few) arguments, an interactive data-entry window pops up to allow input of additional parameters. Called with additional arguments, "pop" functions simply call the eponymous data processing function, without creating a pop-up window. When a "pop" function is called by the user by selecting a menu item in the main EEGLAB window, the function is called without additional parameters, bringing up its gui pop-up window to allow the user to enter computation parameters. When the processing function is called by EEGLAB, its function call is added as a command string to the EEGLAB session history variable. By copying history commands to the Matlab command line or embedding them in Matlab text scripts, users can easily apply actions taken during a gui-based EEGLAB session to a different data set. A comprehensive help message for each of the "pop" functions allows users to adapt the commands to new types of data.

Custom EEGLAB and MATLAB scripting

More experienced Matlab users can take advantage of EEGLAB functions and dataset structures to perform computations directly on datasets using their own scripts that call EEGLAB and any other Matlab functions while referencing EEGLAB data structures. Most "pop_" functions describe above call signal processing functions. For example, the pop_erpimage() function calls signal processing and plotting function erpimage(). Since all the EEGLAB data processing functions are fully documented, they can be used directly. Experienced users should benefit from using all three modes of EEGLAB processing: gui-based, history-based, and autonomously scripted data analyses. Such users can take advantage of the data structure (*EEG*) in which EEGLAB datasets are stored. EEGLAB uses a single Matlab variable, a structure, *EEG*, that contains all dataset information and is always available at the Matlab command line. This variable can easily be used and/or modified to perform custom signal processing or visualizations. Finally, while EEGLAB "pop" functions (described above) assume that the data are stored in an EEG data structure, most EEGLAB signal processing functions accept standard Matlab array arguments. Thus, it is possible to bypass the EEGLAB interface and data structures entirely, and directly apply the signal processing functions to data matrices.

Distribution, documentation and support

The EEGLAB toolbox is distributed under the GNU General Public License (for details see http://www.gnu.org/licenses/gpl.txt). The source code, together with web tutorials and function description help pages, is freely available for download from http://sccn.ucsd.edu/eeglab/. The toolbox currently includes well over 300 Matlab functions comprising more than 50,000 lines of Matlab code. This user tutorial explains in detail how to import and process data using EEGLAB, including the derivation and evaluation of its independent components. SCCN also provides "Frequently Asked Questions (FAQ)" and "Known Bugs" web pages, a support email (eeglab@sccn.ucsd.edu), a dedicated mailing list for software updates (eeglabnews@sccn.ucsd.edu), and an email discussion mailing list (eeglablist@sccn.ucsd.edu), which currently reaches over two thousand EEG researchers.

Open-source EEGLAB functions are not precompiled; users can read and modify the source code of every function. Each EEGLAB function is also documented carefully using a standardized help-message format and each function argument is described in detail with links to related functions. We have attempted to follow recognized best practice in software design for developing EEGLAB. The source code of EEGLAB is extensively documented and is internally under the Linux revision control system (RCS), which allows us to easily collaborate with remote researchers on the development of new functions. Matlab allows incremental design of functions, so adding new features to a function can be easily accomplished while preserving backward compatibility.

(Adapted from, A Delorme & S Makeig. EEGLAB: an open source toolbox for analysis of single-trial EEG dynamics. *Journal of Neuroscience Methods* 134:9-21 (2004)).

I.Single subject data processing tutorial

The EEGLAB Tutorial is split into three parts. The first part is the Main Tutorial. In the Main Tutorial, a user is led through how to set up EEGLAB on their computer and is introduced to basic EEGLAB functions. The user is instructed through a series of Key and Exploratory Steps to manipulate a sample data set (provided). This basic tutorial should enable to user to develop a good understanding of how to use EEGLAB for their own experiments. As of 2/10/2012, there are twelve topics in the Main Tutorial.

 I.Single subject data processing tutorial

Chapter 01: Loading Data in EEGLAB

Learning Matlab

EEGLAB graphic interface is built on top of the powerful Matlab scripting language. Enjoying the full capabilities of EEGLAB for building macro commands and performing custom and automated processing requires the ability to manipulate EEGLAB data structures in Matlab. After opening the Matlab desktop, we recommend running the following demos and reading the following help sections.

In the Help Content, select menu item Help Demos and run the following demos:

- Mathematics - Basic Matrix Operations
- Mathematics - Matrix Manipulations
- Graphics - 2-D Plots
- Programming - Manipulating Multidimensional Arrays
- Programming - Structures

In the Help Content, read and practice at least the following sections:

- Getting Started - Matrices and Arrays - Matrices and Magic Squares
- Getting Started - Matrices and Arrays - Expressions
- Getting Started - Matrices and Arrays - Working with Matrices
- Getting Started - Graphics - Basic Plotting Functions
- Getting Started - Programming - Flow Control
- Getting Started - Programming - Other Data Structures
- Getting Started - Programming - Scripts and Functions

Each section or demo (if read thoroughly) should take you about 10 minutes, for a total here of about 2 hours. We encourage you to watch these demos and read these sections over several sessions.

If you do not have access to the Matlab demos, here is a short, online introduction to Matlab (recommended pages, 1 to 12).

If you are an experienced Matlab programmer, you may want to read through the EEGLAB tutorial appendices on the EEG dataset and STUDY studyset structures and their sub-structures, and on EEGLAB script writing.

Installing EEGLAB and tutorial files

First download EEGLAB (30MB) which contains the tutorial dataset (also available here (4MB) - *press the right mouse button and select* save link as *if strange characters appear*).

When you uncompress EEGLAB you will obtain a folder named "eeglabxxxx" (note: current version number varies). Under Windows, Matlab usually recommends, although this is not required, that you place toolboxes in the *Application/MatlabR14/toolbox/* folder (note: name varies with the Matlab version). In Linux, the Matlab toolbox folder is typically located at */usr/local/pkgs/Matlab-r14sp2/toolbox/*. You may also place the folder anywhere else on your path.

Start Matlab and list the eeglab folder in the Matlab search path. If you started Matlab with its graphical interface, go to menu item file and select set path, which opens the following window.

Or, if you are running Matlab from the command line, type in "pathtool", which will also call up this window. Click on the button marked *Add folder* and select the folder "eeglabxxxxx", then hit *OK* (EEGLAB will take care of adding its subfolder itself). Hit *save* in the pathtool window, which will make eeglab available for future Matlab sessions. Note that if you are

installing a more recent version of EEGLAB, it is best to remove the old version from the Matlab path (select and hit *Remove*) to avoid the possibility of calling up outdated routines.

Starting Matlab and EEGLAB

Here we will start Matlab and EEGLAB. Note that paragraphs that begin with KEY STEP are necessary to what follows in the tutorial. Paragraphs that begin with Exploratory Step are written to help you explore various EEGLAB features.

KEY STEP 1: Start Matlab.

WINDOWS: Go to Start, find Matlab and run it.
Mac Option 1: Start from the Matlab icon in the dock or in the application folder.
Mac Option 2: Start X11 and in a terminal window type "matlab" and hit enter. For added stability type "matlab -nodesktop" to run it without its java desktop.
Linux: Open a terminal window and follow the Mac Option 2 instructions above.

KEY STEP 2 (optional): Switch to the data directory (folder).

In this case the directory you switch to should be the directory containing the example data provided for this tutorial. If you are running Matlab with its java desktop, you may also browse for the directory by clicking on the button marked *"?"* in the upper right of the screen,

which opens the window below. Double-click on a directory to enter it. Double-clicking on ".." in the folder list takes you up one level. Hit *OK* once you find the folder or directory you wish to be in. Alternatively, from the command line use "cd" (change directory) to get to the desired directory.

KEY STEP 3: Start EEGLAB.

Just type "eeglab" at the Matlab command line and hit enter.

The blue main EEGLAB window below should pop up, with its seven menu headings: File, Edit, Tools, Plot, Study, Datasets, Help arranged in typical (left-to-right) order of use.

Opening an existing dataset

KEY STEP 4: Load the sample EEGLAB dataset.

> Select menu item File and press sub-menu item Load existing dataset.

In the rest of the tutorial, we will use the convention: Menu_item > Submenu_item to refer to selecting a menu choice (e.g., here select File > Load existing dataset).

Under Unix, the following window will pop up (the aspect of the window may be different under Windows):

To learn how to create EEGLAB datasets from your own data, see the tutorial on Importing data and data events.

> Select the tutorial file "eeglab_data.set" which is distributed with the toolbox (available here - press the right mouse button and select *save link as if strange characters appear -'* or in the "sample_data" sub-directory if you downloaded the full version of EEGLAB) and press *Open*.

When the dataset is loaded by EEGLAB, the main EEGLAB window shows relevant information about it -- the number of channels, sampling rate, etc...

Editing event values

The fields *type*, *position*, and *latency* have different values for each of the 154 events in the dataset.

Exploratory Step: Editing Event Values.

Select menu Edit > Event Values to call up a window where we can read and edit these values:

Scroll through the events by pressing the **>**, **>>**, **<**, and **<<** keys above.

We will now briefly describe the experiment that produced the sample dataset to motivate the analysis steps we demonstrate in the rest of the tutorial.

Sample experiment description

In this experiment, there were two types of events "square" and "rt"; "square" events correspond to the appearance of a green colored square in the display and "rt" to the reaction time of the subject. The square could be presented at five locations on the screen distributed along the horizontal axis. Here we only considered presentation on the left, i.e. position 1 and 2 as indicated by the *position* field (at about 3 degree and 1.5 degree of visual angle respectively). In this experiment, the subject covertly attended to the selected location on the computer screen responded with a quick thumb button press only when a square was presented at this location. They were to ignore circles presented either at the attended location or at an unattended location. To reduce the amount of data required to download and process, this dataset contains only targets (i.e., "square") stimuli presented at the two left-visual-field attended locations for a single subject. For more details about the experiment see Makeig, et al., Science, 2002, 295:690-694.

When using events in an EEGLAB dataset, there are two required event fields: *type* and *latency*, plus any number of additional user-defined information fields. It is important to understand here that the names of the fields were defined by the user creating the dataset, and that it is possible to create, save, and load as many event fields as desired.

Note also that *type* and *latency* (lowercase) are two keywords explicitly recognized by EEGLAB and that these fields *must* be defined by the user unless importing epoch event information (Note: If only field *latency* is defined, then EEGLAB will create field *type* with a constant default value of 1 for each event). Unless these two fields are defined, EEGLAB will not be able to handle events appropriately to extract epochs, plot reaction times, etc. The Importing data tutorial explains how to import event information and define fields.

About this dataset

Here we describe how to edit and view the text field which describes the current dataset, and is stored as a part of that dataset.

Exploratory Step: Editing the Dataset Description.

Select Edit > About this dataset. A text-editing window pops up which allows the user to edit a description of the current dataset. For the sample data, we entered the following description of the task. Press *SAVE* when done.

Scrolling through the data

Here we learn how to visualize data and reject portion of continuous data.

Exploratory Step: Scrolling data.

To scroll through the data of the current dataset, select Plot > Channel data (scroll). This pops up the eegplot() scrolling data window below. Note that the sample data file contains "faked" continuous EEG data. To reduce (your) download time, this "pseudo-continuous" EEG dataset was actually constructed by concatenating eighty separate three-second data epochs (which we will later separate again).

To the right of the plot window is the vertical scale value (unit, i.e. microvolts), which indicates the "height" of the given vertical scale bar. In this case, that value is 80 (microvolts). The same value is also shown in the lower right-hand edit box, where we can change it as explained below.

Exploratory Step: Voltage Scale.

Change the "Scale" edit-text box value to about 50, either by repeatedly clicking on the *"-"* button or by editing the text value from the keyboard, and press the *Enter* key to update the scrolling window.

Exploratory Step: Adjusting time range.

To adjust the time range shown (horizontal scale), select eegplot() menu item Settings > Time range to display, and set the desired window length to "10" seconds as shown below,

then press *OK* to display the changes.

Exploratory Step: Number of Channels to Display.

To adjust the number of channels displayed, select eegplot() menu item Settings > Number of channels to display and enter the desired number of channels to display in the screen (for instance "16").

This will return a scrolling eegplot() window with a vertical channel-set slider on the left of the plot. Use it to scroll vertically through all the channels.

Exploratory Step: Zoom.

To zoom in on a particular area of a data window, select eegplot() menu item Settings > Zoom off/on > Zoom on. Now using your mouse, drag a rectangle around an area of the data to zoom in on. The scrolling window may now look similar to the following. Click the right button on the mouse to zoom out again. Use menu Setting > Zoom off/on > Zoom off to turn off the zoom option.

Exploratory Step: Grid Lines.

To display horizontal (x) and vertical (y) grid lines on the data, select Display > Grid > x axis or Display > Grid > y axis. Repeat this process to turn off either set of grid lines.

The eegplot() window also allows you to reject (erase) arbitrary portions of the continuous data. The functioneegplot() can be called from both menu items Plot > Scroll data and Tools > Reject continuous data using the *REJECT* button on the bottom right corner.

Exploratory Step: Rejecting Data.

Close the current eegplot() window and call Tools > Reject Continuous Data by eye from the main EEGLAB window. A warning message appears, click on continue. To erase a selected portion of the data, first drag the mouse along the area of interest to mark it for rejection. It is possible to mark multiple regions. To undo a rejection mark, click once on the marked region.

(Note: Zooming must be disabled to select a portion of the data.)
Now, to erase the marked data regions, click the (lower right) *REJECT* button (above). A new dataset will be created with the marked regions removed. (Note: EEGLAB will also add new "rejection boundary" events to the new dataset event list. These insure that subsequent epoch selections will not cross non-contiguous rejection boundaries). For more details about rejecting continuous data regions and data epochs, see the data rejection tutorial.
Click *OK* (below) to create the new dataset with the marked data portions removed.

Press *OK* to create the new dataset. The EEGLAB main window now looks like:

Since we only performed this rejection for illustrative purposes, switch back to the original dataset by selecting main window menu item Datasets > Dataset 1 eeglab_data.

Exploratory Step: Deleting a Dataset from Memory.

To delete the newly created second dataset, select File > Clear dataset(s) or Edit > Delete dataset(s) and enter the dataset index, "2" as shown below, and press *OK*.

The second dataset will now be removed from the Matlab workspace. (Note: It is not necessary to switch back to the first dataset before deleting the second. It is also possible to delete several datasets at once from this window by entering their indices separated by spaces.)

Chapter 02: Channel Locations

Importing channel location for the tutorial dataset

To plot EEG scalp maps in either 2-D or 3-D format, or to estimate source locations for data components, an EEGLAB dataset must contain information about the locations of the recording electrodes.

KEY STEP 5: Load the channel locations.

To load or edit channel location information contained in a dataset, select Edit > Channel locations.

If you imported a binary data file in Neuroscan or Biosemi formats, channel labels will be present in the dataset (as of EEGLAB v4.31). When you call the channel editing window, a dialog box will appear asking you if you want to use standard channel locations corresponding to the imported channel labels (for example. Fz) from an channel locations file using an extended International 10-20 System. Otherwise, you must load a channel locations file manually.

To load a channel locations file, press the *Read Locations* button and select the sample channel locations file *eeglab_chan32.locs* (this file is located in the *sample_data* sub-directory of the EEGLAB distribution).

In the next pop-up window, simply press *OK*. If you do not specify the file format, the pop_chanedit.m function will attempt to use the filename extension to assess its format. Press button *Read help* in the main channel graphic interface window to view the supported formats.

Now the loaded channel labels and polar coordinates are displayed in the pop_chanedit.m window. To plot scalp maps only inside the head cartoon, enter 0.5 at the *Plot radius* edit box. In this case, the two eye electrodes will not be displayed nor considered computing interpolated 2-D scalp maps. If you do not see enough of the recorded field, set this dialogue box to 1.0 to interpolate and show scalp maps including all channels, with parts of the head below the (0.5) head equator shown in a 'skirt' or 'halo' outside the cartoon head boundary. (More precise separate controls of which channel locations to interpolate and plot are available from the command line in the topographic map plotting function topoplot.m. In the window below, you may scroll through the channel field values 1-by-1 using the *<* and *>* buttons, or in steps of 10 using *<<* and *>>*.

Exploratory Step: Viewing Channel Locations.

Reopen Edit > Channel locations if you closed it. To visualize the 2-D locations of the channels, press *Plot 2-D* above the *Read Locations* button. Else, at any time during an EEGLAB session you may refer to a plot showing the channel locations by selecting Plot > Channel location > By name. Either command pops up a window like that below. Note: In this plot, click on any channel label to see its channel number.

The appendix section on importing channel location contains additional information.

Chapter 03: Plotting Channel Spectra and Maps

Plotting channel spectra and maps

To begin processing the data, we recommend first scrolling the data as shown before rejecting clearly 'bad' data stretches or data epochs, then studying their power spectra to be sure that the loaded data are suitable for further analysis. Note that the Matlab Signal Processing Toolbox needs to be in your Matlab path to use these functions.

Exploratory Step: Plot Channel Spectra and Maps.

To plot the channel spectra and associated topographical maps, select Plot > Channel spectra and maps. This will pop up the pop_spectopo.m window (below). Leave the default settings and press *OK*.

The function should return a spectopo.m plot (below). Since we only sampled 15% of the data (via the *Percent data...* edit box above), results should differ slightly on each call. (Naturally, this will not occur if you enter 100% in the edit box).

Each colored trace represents the spectrum of the activity of one data channel. The leftmost scalp map shows the scalp distribution of power at 6 Hz, which in these data is concentrated on the frontal midline. The other scalp maps indicate the distribution of power at 10 Hz and 22 Hz.

The pop_spectopo.m window menu (above) allows the user to compute and plot spectra in specific time windows in the data. The *Percent data...* value can be used to speed the computation (by entering a number close to 0) or to return more definitive measures (by entering a number closer to 100). Note that functions pop_spectopo.m and spectopo.m also work with epoched data. Another menu item, Plot > Channel properties, plots the scalp location of a selected channel, its activity spectrum, and an ERP-image plot of its activity in single-epochs.

The next section deals with some data preprocessing options available via the EEGLAB menu.

Chapter 04: Preprocessing Tools

Preprocessing tools

The upper portion of the Tools menu may be used to call three data preprocessing routines:

Changing the data sampling rate

The most common use for Tools > Change sampling rate is to reduce the sampling rate to save memory and disk storage. A pop_resample.m window pops up, asking for the new sampling rate. The function uses Matlab *resample()* (in the Signal Processing toolbox--if you do not have this toolbox, it will use the slow Matlab function *griddata*). Do not use this function here, since the tutorial EEG dataset is already at an acceptable sampling rate.

Filtering the data

To remove linear trends, it is often desirable to high-pass filter the data.

KEY STEP 6: Remove linear trends.

> We recommend filtering continuous EEG data, before epoching or artifact removal, although epoched data can also be filtered with this function (each epoch being filtered separately). Filtering the continuous data minimizes the introduction of filtering artifacts at epoch boundaries. Select Tools > Filter the data > Basic FIR filter, enter *1* (Hz) as the *Lower edge* frequency, and press *OK*.

> A pop_newset.m window will pop up to ask for the name of the new dataset. We choose to modify the dataset name and to overwrite the parent dataset by checking the *Overwrite parent* checkbox, then pressing the *OK* button.

> Note that if high-pass and low-pass cutoff frequencies are BOTH selected, the filtering routine may not work. To avoid this problem, we recommend first applying the low-pass filter and then, in a second call, the high-pass filter (or vice versa).

> Another common use for bandpass filtering is to remove 50-Hz or 60-Hz line noise. The filtering option in EEGLAB, eegfilt.m, uses linear finite impulse response (FIR) filtering. If the Matlab Signal Processing Toolbox is present, it uses the Matlabroutine *filtfilt()*. This applies the filter forward and then again backward, to ensure that phase delays introduced by the filter are nullified. If the Matlab Signal Processing toobox is not present, EEGLAB uses a simple filtering method involving the inverse fourrier transform.

> A infinite impulse response (IIR) filter plug-in is also distributed with EEGLAB. See menu item Tools > Filter the data (IIR). It uses the same graphical interface as the FIR filtering option described above. Although IIR filters usually introduce different phase delays at different frequencies, this is compensated for by again applying filtering in reverse

using Matlab function *filtfilt()*. In practice, we suggest you test the use of this IIR filter, as it is stronger (and shorter) than FIR filters.

If you apply filtering and continue to work with the updated data set, check that the filter has been applied by selecting menu item Plot > Channel spectra and maps to plot the data spectra. You might notice that filtered-out frequency regions might show 'ripples', unavoidable but hopefully acceptable filtering artifacts. (Note: There is much more to be learned about filtering, and more filtering options available in Matlab itself).

Re-referencing the data

The reference electrode used in recording EEG data is usually termed the 'common' reference for the data -- if all the channels use this same reference. Typical recording references in EEG recording are one mastoid (for example, TP10 in the 10-20 System, the electrode colored red in the picture below), linked mastoids (usually, digitally-linked mastoids, computed *post hoc*, the vertex electrode (Cz), single or linked earlobes, or the nose tip. Systems with active electrodes (e.g. BIOSEMI Active Two), may record data reference-free. In this case, a reference be must be chosen *post hoc* during data import. Failing to do so will leave 40 dB of unnecessary noise in the data!

There is no 'best' common reference site. Some researchers claim that non-scalp references (earlobes, nose) introduce more noise than a scalp channel reference though this has not been proven to our knowledge. If the data have been recorded with a given reference, they can usually be re-referenced (inside or outside EEGLAB) to any other reference channel or channel combination.

Converting data, before analysis, from fixed or common reference (for example, from a common earlobe or other channel reference) to 'average reference' is advocated by some researchers, particularly when the electrode montage covers nearly the whole head (as for some high-density recording systems). The advantage of average reference rests on the fact that outward positive and negative currents, summed across an entire (electrically isolated) sphere, will sum to 0 (by Ohm's law). For example, in the figure below a tangentially-oriented electrical source is associated with a positive inward current to the left (here, blue) and an opposing outward negative current to the right (red). If the current passing through the base of the skull to the neck and body is assumed to be negligible (for instance, because of low conductance of the skull at the base of the brain), one may assume that the sum of the electric field values recorded at all (sufficiently dense and evenly distributed) scalp electrodes is always 0 (the average reference assumption).

The problem with this assumption is that true average reference data would require the distribution of electrodes to be even over the head. This is not usually the case, as researchers typically place more electrodes over certain scalp areas, and fewer (if any) on the lower half of the head surface. As a consequence, an average reference result using one montage may not be directly comparable to an average reference result obtained using another montage.

Below, we detail the process of transforming data to 'average reference'. Note that in this process, the implied activity time course at the former reference electrode may be calculated from the rest of the data (so the data acquires an additional channel - though not an additional degree of freedom!). Also note that if the data were recorded using nose tip or ear lobe electrodes, you should not include these reference electrodes when computing the average reference in (1) (below), Thus, in the example below the dividing factor (in (3)) would be 64 instead of 65. Note that in localizing sources using the EEGLAB DIPFIT plug-in, 'average reference' will be used internally (without user input).

The choice of data reference does color (literally) the plotted results of the data analysis. For example, plots of mean alpha power over the scalp must have a minimum at the reference channel, even if there are in fact alpha sources just below and oriented toward the reference channel! However, no (valid) reference can said to be wrong - rather, each reference can be said to give another view of the data. Howeever, the nature of the reference needs to be taken into account when evaluating (or, particularly, comparing) EEG results.

For ICA decomposition (covered later in the tutorial), the selection of reference is not so important. This is because changing the reference only amounts to making a linear transformation of the data (in mathematical terms, multiplying it by a fixed re-referencing matrix), a transformation to which ICA should be insensitive. In practice, we have obtained results of similar quality from data recorded and analyzed with mastoid, vertex, or nose tip reference.

We advise recording eye channels (conventionally four channels, two for vertical eye movement detection and two for horizontal eye movement detection) using the same reference as other channels, instead of using bipolar montages. One can always recover the bipolar montage activity by subtracting the activities of the electrode pairs. We term these channels 'peri-ocular EEG' channels since what they record is not exclusively electro-oculographic (EOG) signals, but also includes e.g. prefrontal EEG activities.

ICA can be used to decompose data from either average reference, common reference, or bipolar reference channels -- or from more than one of these types at once. However, plotting single scalp maps requires that all channels use either the same common reference or the same average reference. Robert Oostenveld advises that peri-ocular channel values, even in ICA component maps, may best be omitted from inverse source modeling using simple head models, since these are apt to poorly model the conductance geometry at the front of the head.

We will now describe how to specify the reference electrode(s) in EEGLAB and to (optionally) re-reference the data

Exploratory Step: Re-reference the Data.

Select Tools > Re-reference to convert the dataset to average reference by calling the pop_reref.m function. When you

call this menu item for the first time for a given dataset, the following window pops up.

The (sample) data above were recorded using a mastoid reference. Since we do not wish to include this reference channel (neither in the data nor in the average reference), we do not click the *Include current reference channel in data* check-box. (Do click this check-box when the recording reference was on the scalp itself). The box *Data are referenced to one site (default)* should remain checked.

Now, press the *OK* button: the re-referencing window below appears.

Press the *OK* button to compute the average reference. This step will then be recorded in the main EEGLAB window (not shown). As in the previous step, a dialogue box will appear asking for the name of the new dataset. Save the re-referenced data to a new dataset or hit cancel, as the new reference is not used in the following sections.

After the data have been average referenced, calling the Tools > Re-reference menu still allows re-referencing the data to any channel or group of channels (or undoing an average reference transform -- as long as you chose to include the initial reference channel in the data when you transformed to average reference).

Note that the re-referencing function also re-references the stored ICA weights and scalp maps, if they exist.

Re-referencing data can be more complicated. For instance, if you recorded data referenced to CZ and want to re-reference the data linked mastoid. Now you want to add Cz back to your data under the average reference assumption (the assumption that the average of all electrode is 0). The first step is to compute average reference and declare Cz as the reference in the channel editor. In the channel editor, references are placed after all the data channels (note that for reference the checkbox "data channel" is unchecked since these are not actual data channels). To declare a reference, go to the last channel and press the *Append* button. An empty channel is created.

Fill up the channel label (enter "Cz" in the "Channel label" edit box) and enter the position of the channel if you have it. For instance, you may enter the X, Y, Z locations and press the *XYZ -> Polar & Sph.* to convert the 3-D Cartesian coordinates to polar and spherical coordinates. If you do not have the electrode location, you may simply press the *Look up locs* button to automatically look it up based on the 10-20 channel label (note that this will look up location for all electrodes).

Then press the *Set reference* pushbutton to set the reference to all channels to Cz (Cz need to be typed into the checkbox and the channel range needs to be entered manually as well).

Press OK to validate your new reference channel, and go back to the re-referencing interface. Now click on the *Retain old reference* button.

You may now select electrode "Cz" and press OK.

Then press *OK* to actually re-reference your data. This is the first step. If you actually want to re-reference your data to linked mastoid, you will need to call the re-referencing interface once more and select both mastoids as your new reference.

The reason for this overly complex procedure is that the reference channel can have a location and that this location needs to be declared in the channel editor so it can be plotted along with other channels.

The next tutorial section deals with extracting data epochs from continuous or epoched datasets.

Chapter 05: Extracting Data Epochs

Extracting epochs

To study the event-related EEG dynamics of continuously recorded data, we must extract data epochs time locked to events of interest (for example, data epochs time locked to onsets of one class of experimental stimuli).

KEY STEP 7: Extract data epochs.

Click on the upper right button, marked *"?"*, of the resulting pop_epoch.m window, which calls up a browser box listing the available event types.

Here, choose event type *square* (onsets of square target stimuli in this experiment), and press *OK*. You may also type in the selected event type directly in the upper text box of the pop_epoch.m window.

Here, retain the default epoch limits (from 1 sec before to 2 sec after the time-locking event). If you wish, add a descriptive name for the new dataset. Then press *OK*. A new window will pop up offering another chance to change the dataset name and/or save the dataset to a disk file. At this point, it can be quite useful to edit the dataset description -- to store the exact nature of the new dataset in the dataset itself, for future reference. Do this by pressing *Description*. Accept the defaults and enter *OK*.

Another window will then pop up to facilitate removal of meaningless epoch baseline offsets. This operation is discussed in the next section.
In this example, the stimulus-locked windows are 3 seconds long. It is often better to extract long data epochs, as here, to make time-frequency decomposition possible at lower (<< 10 Hz) frequencies.

Removing baseline values

Removing a mean baseline value from each epoch is useful when baseline differences between data epochs (e.g., those arising from low frequency drifts or artifacts) are present. These are not meaningfully interpretable, but if left in the data could skew the data analysis.

KEY STEP 8: Remove baseline values.

After the data has been epoched, the following window will pop up automatically. It is also possible to call it directly, by selecting menu item Tools > Remove baseline.

Here we may specify the baseline period in each epoch (in ms or in frames = time points) -- the latency window in each epoch across which to compute the mean to remove The original epoched dataset is by default overwritten by the baseline-removed dataset. Note: There is no uniformly 'optimum' method for selecting either the baseline period or the baseline value. Using the mean value in the pre-stimulus period (the pop_rmbase.m default) is effective for many datasets, if the goal of the analysis is to define transformations that occur in the data following the time-locking events. Press *OK* to subtract the baseline (or *Cancel* to not remove the baseline).

Exploratory Step: Re-reference the Data.

This is a good time to save the epoched and baseline-removed dataset under a new name, as explained above, since we will be working extensively with these data epochs. You should also preserve the continuous dataset on the disk separately to allow later arbitrary re-epoching for various analyses. We might have saved the epoched dataset as a new file under a new filename using the *pop_newset()* window (above) (by pressing *Browse*). To save the current dataset at any other time, select File > Save current dataset or File > Save current dataset as from the EEGLAB menu. (In this case, these two menu items are equivalent, since we have not yet saved this dataset).

The file-browser window below appears. Entering a name for the dataset (which should end with the filename extension *.set*), and pressing *SAVE* (below) and then *OK* (above) will save the dataset including all its ancillary information re events, channel locations, processing history, etc., plus any unique structure fields you may have added yourself - see the script writing tutorial.

The next tutorial discusses averaging the data epochs of epoched datasets.

Chapter 06: Data Averaging

Data averaging

Previously (Makeig et al., 2002, 2004), we have discussed ways in which the event-related EEG dynamics occurring in a set of data epochs time locked to some class of events are not limited to nor completely expressed in features of their time-locked trial average or Event-Related Potential (ERP). EEGLAB contains several functions for plotting 1-D ERP averages of dataset trials (epochs). EEGLAB also features functions for studying the EEG dynamics expressed in the single trials, which may be visualized, in large part, via 2-D (potential time series by trials) ERP-image transforms of a dataset of single-trial epochs (a.k.a., epoched data). In ERP-image plots, EEG data epochs (trials) are first sorted along some relevant dimension (for example, subject reaction times, within-trial theta power levels, mean voltage in a given latency window, alpha phase at stimulus onset, or etc.), then (optionally) smoothed across neighboring trials, and finally color-coded and visualized as a 2-D rectangular color (or monochrome) image. (Construction of ERP images will be detailed in section I.8).

Plotting the ERP data on a single axis with scalp maps

Here, we will use the tutorial dataset as it was after the last Key Step, Key Step 8.

Exploratory Step: Plotting all-channel ERPs.

To plot the (ERP) average of all dataset epochs, plus ERP scalp maps at selected latencies, select Plot > Channel ERP> With scalp maps. As a simple illustration using the sample dataset, we retain the default settings in the resulting pop_timtopo.m window, entering only a plot title and pressing *OK*.

A timtopo.m figure (below) appears. Each trace plots the averaged ERP at one channel. The scalp map shows the topographic distribution of average potential at 430 ms (the latency of maximum ERP data variance). Alternatively, one or more exact scalp map latencies may be specified in the pop-window above.

Function timtopo.m plots the relative time courses of the averaged ERP at all channels, plus ?snapshots? of the scalp potential distribution at various moments during the average ERP time course. Note that to To visualize the ERP scalp map at *all* latencies -- as an ERP movie (i.e., to view the play of the ERP on the scalp), call function eegmovie.m from the command line.

Plotting ERP traces in a topographic array

Exploratory Step: Plotting ERPs in a Topographic Map.

Here we will plot the ERPs of an epoched dataset as single-channel traces in their 2-D topographic arrangement. Select Plot > Channel ERPs > In scalp array. Using the default settings and pressing *OK* in the resulting pop_topoplot.m window (below)

produces the following plottopo.m figure.

Note that if called from the command line, the plottopo.m function *geom* option can also be used to plot channel waveforms in a rectangular grid. You may visualize a specific channel time course by clicking on its trace (above), producing a pop-up sub-axis view. For example, click on the ERP trace marked *POz* (above) to call up a full-sized view of this trace (as below).

Many EEGLAB plotting routines use the toolbox function axcopy.m to pop up a sub-axis plotting window whenever the users clicks on the main plot window. Sub-axis windows, in turn, do not have axcopy.m enabled, allowing the user to use the standard Matlab mouse 'Zoom In/Out' feature.

Plotting ERPs in a two column array

Exploratory Step: Plotting ERPs in a Column Array.

To plot (one or more) average ERP data traces in two column array, select menu item Plot > Channel ERPs > In rect. array. To use the default settings in the resulting pop_plotdata.m window, simply press *OK*.

The resulting plotdata.m figure (below) appears.

As in the previous plot, clicking on a trace above pops up a full window sub-axis view.

Plotting an ERP as a series of scalp maps

Plotting a series of 2-D ERP scalp maps

Here we will plot ERP data for a series of 2-D scalp maps representing potential distributions at a selected series of trial latencies.

Exploratory Step: Plotting a series of 2-D ERP Scalp Maps.

Select Plot > ERP map series > In 2-D. In the top text box of the resulting pop_topoplot.m window (below), type the epoch latencies of the desired ERP scalp maps.
Note that in this or any other numeric text entry box, you may enter any numeric Matlab expression. For example, instead of *0 100 200 300 400 500*, try *0:100:500*. Even more complicated expressions, for example *-6000+3*(0:20:120)*, are interpreted correctly.

The topoplot.m window (below) then appears, containing ERP scalp maps at the specified latencies. Here, the plot grid has 3 columns and 2 rows; other plot geometries can be specified in the gui window above via the *Plot geometry* text box.

Plotting ERP data as a series of 3-D maps

Exploratory Step: Plotting a series of 3-D ERP scalp maps.

To plot ERP data as a series of 3-D scalp maps, go to the menu Plot > ERP map series > In 3-D The query window (below) will pop up, asking you to create and save a new 3-D head map *3-D spline file*. This process must be done only once for every montage (and proceeds much more quickly in EEGLAB v4.6-). Click *OK* to begin this process.

The window below will pop up. Here, you have two choices: If you have already generated a spline file for this channel location structure, you may enter it here in the first edit box (first click on the *Use existing spline file or structure* to activate the edit box, then browse for a datafile. If you have not made such a file, you will need generate one.

However, first your channel locations must be co-registered with the 3-D head template to be plotted. Note that if you are using one of the template channel location files, for example, the (v4.6+) tutorial dataset, the *Talairach transformation matrix* field (containing channel alignment information) will be filled automatically. Enter an output file name (in the second edit box), trial latencies to be plotted (0:100:500 below indicating latencies 0, 100, 200, 300, 400, and 500 ms) and press *OK*.

Now, the 3-D plotting function pop_headplot.m, will create the 3-D channel locations spline file. A progress bar will pop up to indicate when this process will be finished. When the 3-D spline file has been generated, select Plot > ERP map series > In 3-D. Now that a spline file has been selected, another gui window will appear. As for plotting 2-D scalp maps, in the first edit box type the desired latencies, and press *OK*. The headplot.m figure (below) will appear.

As usual, clicking on a head plot will make it pop up in a sub-axis window in which it can be rotated using the mouse. Note that to select (for other purposes) a head plot or other object in a figure that has the axcopy.m pop-up feature activated, click on it then delete the resulting pop-up window.

To plot the heads at a specified angle, select the Plot > ERP map series > In 3-D menu item. Note that now by default the function uses the 3-D spline file you have generated above. Enter latencies to be displayed and the headplot.m 'view' option (as in the example below), and press *OK*.

The headplot.m window (below) will then appear. You may also rotate the individual heads using the mouse. This is often necessary to show the illustrated spatial distribution to best effect.

We will now briefly describe the channels-to-head model co-registration process. If your dataset contains specific channel locations, for example locations that you may have measured on the subject head using a Polhemus system, and you want to use these electrode locations for 3-D plotting, headplot.m must first determine the positions of your electrode locations relative to a template head surface. A generic transformation cannot be performed because the origin ([0 0 0]) in your electrode location system does not necessarily correspond to the center of the template head (.e.g., the intersection of the fiducial points: nasion and pre-auricular points) used by headplot.m. Even if this were the case, heads have different shapes, so your scanned electrode locations might need to be scaled or warped in 3-D to match the template head mesh.

The co-registration window begins this operation. Call back the headplot.m. gui window using menu item Plot > ERP map series > In 3-D. Set the checkbox labeled *Or recompute a new spline file named:*, and then click the *Manual coreg.* push button. A window appears explaining how to perform the co-registration.

Pressing *OK* will cause the co-registration window below to open.

In the window above, the red electrodes are those natively associated with the template head mesh. Rather than directly aligning your electrode locations (shown in green) to the head mesh, your montage will be aligned to template electrode locations associated with the head mesh by scanning on the same subject's head (here shown in red). For the sample dataset, this alignment has already been performed. (Click the *Labels on* push button to display the electrode labels).

When you have finished the co-registration, simply click *OK* and a vector of 9 new channel alignment values (shift, in 3-D; rotation, in 3-D; scaling, in 3-D) will be copied back to the interactive headplot.m window. For more information about channel co-registration, see the DIPFIT tutorial.

Note that it is possible, and relatively easy, to generate custom headplot.m head meshes. Let us know by email if you are interested in learning how to do this.

The next tutorial section will demonstrate how to use EEGLAB functions to compare the ERP trial averages of two datasets.

Chapter 07: Selecting Data Epochs and Comparing

Selecting data epochs and plotting data averages

Selecting events and epochs for two conditions

To compare event-related EEG dynamics for a subject in two or more conditions from the same experiment, it is first necessary to create datasets containing epochs for each condition. In the experiment of our sample dataset, half the targets appeared at position 1 and the other half at position 2 (see sample experiment description).

Exploratory Step: Selecting Events and Epochs for Two Conditions.

Select Edit > Select epochs/events. The pop_selectevent.m window (below) will appear. Enter "1" in the textbox next to *position*, which will select all epochs in which the target appeared in position 1.

Press *Yes* in the resulting query window (below):

Now a pop_newset.m window for saving the new dataset pops up. We name this new dataset "Square, Position 1" and press *OK*.

Now, repeat the process to create a second dataset consisting of epochs in which the target appeared at position 2.

First, go back to the previous dataset by selecting Datasets > Continuous EEG Data epochs, Square. Next select Edit > Select epoch/events. In the resulting pop_selectevent.m window, enter "2" in the text box to the right of the *position* field. Press *OK*, then name the new dataset "Square, Position 2".

See the event tutorial, "selecting events", for more details on this topic.

Another function that can be useful for selecting a dataset subset is the function pop_select.m called by selecting Edit > Select data. The example below would select data sub-epochs with the epoch time range from -500 ms to 1000 ms. It would, further, remove dataset epochs 2, 3 and 4 and remove channel 31 completely.

Computing Grand Mean ERPs

Normally, ERP researchers report results on grand mean ERPs averaged across subjects. As an example, we will use EEGLAB functions to compute the ERP grand average of the two condition ERPs above.

Exploratory Step: Computing Grand Mean ERPs.

Select Plot > Sum/Compare ERPs. In the top text-entry boxes of the resulting pop_comperp.m window (below), enter the indices of datasets ?3? and ?4?. On the first row, click the *avg.* box to display grand average, the *std.* box to display standard deviation, and the *all ERPs* box to display ERP averages for each dataset. Finally *0.05* for the t-test significance probability (p) threshold. Then press *OK*.

The plot below appears.

Now, click on the traces at electrode position *FPz*, calling up the image below. You may remove the legend by deselecting it under the Insert > Legend menu.

Note: If you prefer to use positive up view for the y-axis scale, type *ydir', 1* in the *Plottopo options* field. This can be set as a global or project default in *icadefs.m*. See the Options tutorial.

The ERPs for datasets 3 and 4 are shown in blue and red. The grand average ERP for the two conditions is shown in bold black, and the standard deviation of the two ERPs in dotted black. Regions significantly different from 0 are highlighted based on a two-tailed t-test at each time point. This test compares the current ERP value distribution with a distribution having the same variance and a 0 mean. Note that this t-test has not been corrected for multiple comparisons. The p values at each time point can be obtained from a command line call to the function pop_comperp.m.

Finding ERP peak latencies

Although EEGLAB currently does not have tools for automatically finding ERP peak amplitudes and latencies, one can use the convenient Matlab zoom facility to visually determine the exact amplitude and latency of a peak in any Matlab figure.

Exploratory Step: Finding ERP Peak Latencies.

> For example, in the figure above select the magnifying-glass icon having the **+** sign. Then, zoom in on the main peak of the red curve as shown below (click on the left mouse button to zoom in and on the right button to zoom out). Read the peak latency and amplitude to any desired precision from the axis scale.

Note: It may be desirable to first use the low pass filtering edit box of the pop_comperp.m interface to smooth average data peaks before measuring latency peaks.

Comparing ERPs in two conditions

Exploratory Step: Comparing ERPs in Two Conditions.

To compare ERP averages for the two conditions (targets presented in positions 1 and 2), select Plot > Sum/Compare ERPs. In the top text-entry box of the resulting pop_comperp.m window (below), enter the indices of the datasets to compare. Click all boxes in the *avg.* column. Enter *30* for the low pass frequency and *'title', 'Position 1-2'* in the topoplot.m option edit box. Then press *OK*.

The plottopo.m figure (below) appears.

Again, individual electrode traces can be plotted in separate windows by clicking on electrode locations of interest in the figure (above). Note that here, since the two conditions are similar (only the position of the stimulus on the screen changes), the ERP difference is close to 0.

This function can also be used to compute and plot grand-mean ERP differences between conditions across several subjects, and can assess significant differences between two conditions using a paired t-test (two-tailed). To do so, load the datasets for the two conditions for each subject into EEGLAB and enter the appropriate indices in the pop_comperp.m window.

In EEGLAB 5.0b, a new concept and data structure, the STUDY, has been introduced to aggregate and process datasets from multiple subjects, sessions, and/or conditions. See the Component clustering and STUDY structure tutorials for details. The new STUDY-based functions include a command line function, std_envtopo.m that visualizes the largest or selected independent component cluster contributions to a grand-average ERP in two conditions, and to their difference.

In the following sections, we will be working from the second dataset only, and will not use datasets 3 and 4. Return to the second dataset using the Datasets top menu, and optionally delete datasets numbers 3 and 4 using File > Clear dataset(s).

Data averaging collapses the dynamic information in the data, ignoring inter-trial differences which are large and may be crucial for understanding how the brain operates *in real time*. In the next section, we show how to use EEGLAB to make 2-D ERP-image plots of collections of single trials, sorted by any of many possibly relevant variables.

Chapter 08: Plotting ERP images

Plotting ERP images

The field of electrophysiological data analysis has been dominated by analysis of 1-dimensional event-related potential (ERP) averages. Various aspects of the individual EEG trials that make up an ERP may produce nearly identical effects. For example, a large peak in an ERP might be produced by a single bad trial, an across-the-board increase in power at the same time point, or a coherence in phase across trials without any noticeable significance within individual trials. In order to better understand the causes of observed ERP effects, EEGLAB allows many different *ERP image* trial-by-trial views of a set of data epochs.

ERP-image plots are a related, but more general 2-D (values at times-by-epochs) view of the event-related data epochs. ERP-image plots are 2-D transforms of epoched data expressed as 2-D images in which data epochs are first sorted along some relevant dimension (for example, subject reaction time, alpha-phase at stimulus onset, etc.), then (optionally) smoothed (cross adjacent trials) and finally color-coded and imaged. As opposed to the average ERP, which exists in only one form, the number of possible ERP-image plots of a set of single trials is nearly infinite -- the trial data can be sorted and imaged in any order -- corresponding to epochs encountered traveling in any path through the 'space of trials'. However, not all sorting orders will give equal insights into the brain dynamics expressed in the data. It is up to the user to decide which ERP-image plots to study. By default, trials are sorted in the order of appearance in the experiment.

It is also easy to misinterpret or over-interpret an ERP-image plot. For example, using phase-sorting at one frequency (demonstrated below) may blind the user to the presence of other oscillatory phenomena at different frequencies in the same data. Again, it is the responsibility of the user to correctly weight and interpret the evidence that a 2-D ERP-image plot presents, in light of to the hypothesis of interest -- just as it is the user's responsibility to correctly interpret 1-D ERP time series.

Selecting a channel to plot

To plot an ERP image of activity at one data channel in the single trials of our dataset, we must first choose a channel to plot. Let us, for example, choose a channel with high alpha band power (near 10 Hz). Previously in the tutorial we obtained the spectopo.m plot reproduced below.

The plot above shows that alpha band power (e.g., at 10 Hz) is concentrated over the central occipital scalp.

Exploratory Step: Locating Electrodes.

> We will use the dataset as it was after the last Key Step, Key Step 8.
>
> To find which electrodes are located in this region, we can simply plot the electrode names and locations by selecting Plot > Channel locations > By name, producing the figure below. We see that electrode POz is the channel at which alpha power is largest. Click on the *POz* channel label (below) to display its number (27).

Note: It is also possible to plot electrode locations in the spectral graph by entering *'electrodes', 'on'* in the lowest text box (*Scalp map options*) of the interactive pop_spectopo.m window.

Plotting ERP images using pop_erpimage()

Now that we know the number of the channel whose activity we want to study, we can view its activity in single trials in the form of an ERP-image plot.

Exploratory Step: Viewing a Channel ERP.

Select Plot > Channel ERP image . This brings up the pop_erpimage.m window (below). Enter the channel number (27), a trial-smoothing value of *1*, and press *OK*.

An ERP image is a rectangular colored image in which every horizontal line represents activity occurring in a single experimental trial (or a vertical moving average of adjacent single trials). The figure below (not an ERP image) explains the process of constructing ERP-image plots. Instead of plotting activity in single trials such as left-to-right traces in which potential is encoded by the height of the trace, we color-code their values in left-to-right straight lines, the changing color value indicating the potential value at each time point in the trial. For example, in the following image, three different single-trial epochs (blue traces) would be coded as three different colored lines (below).

By stacking above each other the color-sequence lines for all trials in a dataset, we produce an ERP image. In the standard erpimage.m output figure (below), the trace below the ERP image shows the average of the single-trial activity, i.e. the ERP average of the imaged data epochs. The head plot (top left) containing a red dot indicates the position of the selected channel in the montage.

Note: Both of these plotting features (as well as several others) can be turned off in the pop_erpimage.m pop-up window (above). See check-boxes *plot ERP* and *plot scalp map*.

Since activity in single trials contains many variations, it may be useful to smooth the activity (vertically) across neighboring trials using a rectangular (boxcar) moving average.

Exploratory Step: Plotting a Smoothed ERP.

Again call up the pop_erpimage.m interactive window and set the smoothing width to *10* instead of *1*. Now (see below) it is easier to see the dominant alpha-band oscillations in single trials.

Note: Because of the large number of available options, parameters from the last call (if any) are recalled as defaults (though optional arguments entered via the text box are not). If you experience a problem with this feature, you may type *>>eegh(0)* on the Matlab command line to clear the history.

When plotting a large number of trials, it is not necessary to plot each (smoothed) trial as a horizontal line. (The screen and/or printer resolution may be insufficient to display them all). To reduce the imaging delay (and to decrease the saved plot file size), one can decimate some of the (smoothed) ERP-image lines. Entering *4* in the *Downsampling* box of the pop_erpimage.m</font> window would decimate (reduce) the number of lines in the ERP image by a factor of *4*. If the *Smoothing* width is (in this case) greater than *2*4 = 8*, no information will be lost from the smoothed image.

Note: To image our sample dataset, it is not necessary to decimate, since we have relatively few (80) trials.

Sorting trials in ERP images

In the ERP-image figures above, trials were imaged in (bottom-to-top) order of their occurrence during the experiment. It is also possible to sort them in order of any other variable that is coded as an event field belonging to each trial in the dataset. Below, we demonstrate sorting the same trials in order of response time event latency (reaction time).

Exploratory Step: Sorting Trials in an ERP Image.

In the pop_erpimage.m window again, first press the button *Epoch-sorting field*, and select *Latency*. Next, press the button *Event type*, and select *rt*. In the resulting ERP image, trials will be sorted by the *latency* of *rt* events (our sample data has one *rt* event per epoch if this were not the case,erpimage.m would only have plotted epochs with rt events). Enter *Event time range* of *-200 800* ms to plot activity immediately following stimulus onsets.

Note: In this and some other interactive pop-windows, holding the mouse cursor over the label above a text-entry box for a few seconds pops up an explanatory comment.

Now, the erpimage.m figure below appears. The curved black line corresponds to the latency time of the event (rt) we are sorting by.

In general, the user can sort on any event field value.

For example call back the pop_erpimage.m window, press the *Epoch-sorting Field* button, and select *position* instead of *latency*. Remove *rt* from the *Event type* box. Finally enter *yes* under the *Rescale* box. Press *OK*. In the resulting erpimage.m plot, trials are sorted by stimulus position (1 or 2, automatically normalized values to fit the post-stimulus space for display). Note that the smoothing width (10) is applied to both the single-trial data and to the sorting variable. This explains the oblique line connecting the low (1) and high (2) sorting variable regions.

Note: One can also enter a Matlab expression to normalize the sorting variable explicitly (see erpimage.m help).

Now, reselect the *latency* of the *rt* events as the trial-sorting variable (press the *Epoch-sorting field* button to select *latency* and press the *Event type* button to select *rt*). Enter *no* under *Rescale* (else, reaction times would be automatically normalized).

Use the *Align* input to re-align the single-trial data based on the sorting variable (here the reaction time) and the change time limits. The latency value given in *Align* will be used for specifying time 0.

To select the median of the trial-sorting values (here, median reaction time) for specifying the new time 0 (which will be at the response time minus the median reaction time), our convention is to use *Inf* the Matlab symbol for infinity in this box (as below). If you want to set a different value (for instance, while plotting an ERPimage for one subject, you might want to use the median reaction time you computed for all your subjects), simply enter the value in ms in the *Align* input box.

Note: Temporal realignment of data epochs, relative to one another, will result in missing data at the lower-left and upper-right corners of the ERP image. The ERP-image function shows these as green (0) and returns these values as *NaN*s (Matlab not-a-number).

The ERP image figure (below) will be created. Here, the straight vertical line at time about 400 ms indicates the moment of the subject response, and the curving vertical line, the time at which the stimulus was presented in each trial. Compare the figure below with the previous non-aligned, RT-sorted ERP image.

Plotting ERP images with spectral options

Next, we will experiment with sorting trials by their EEG phase value in a specified time/frequency window. Though *rt* values can be shown in phase-sorted ERP-image figures, we will omit them for simplicity.

Exploratory Step: Sorting Trials in an ERP by Phase Value

To do this, return to the pop_erpimage.m window from the menu. Clear the contents of the *Epoch-sorting field',* Event type *and* Align *inputs. Then, in the* Sort trials by phase *section, enter* 10 *(Hz) under* <Frequency *and* 0*(ms) under* Center window*. Enter* -200 800 *next to* Time limits (ms) *to zoom in on the period near stimulus onset, this option appear at the top of the pop window.*

We then obtain the ERP-image figure below.

Note just before the stimulus onset the red oblique stripe: this is produced by phase sorting: the phase (i.e., the latency of the wave peaks) is uniformly distributed across the re-sorted trials.

In this computation, a 3-cycle 10 Hz wavelet was applied to a window in each trial centered at time 0. The width of the wavelet was 300 ms (i.e., three 10-Hz cycles of 100 ms). Therefore, it extended from -150 ms to 150 ms. After the wavelet was applied to each trial, the function sorted the trials in order of the phase values (-pi to pi) and displayed an ERP image of the trials in this (bottom-to-top) order. The dominance of circa 10-Hz activity in the trials, together with the 10-trial smoothing we applied makes the phase coherence between adjacent trials obvious in this view.

We could have applied phase-sorting of trials using any time/frequency window. The results would depend on the strength of the selected frequency in the data, particularly on its degree of momentum (i.e., did the data exhibit long bursts at this frequency), and its phase-locking (or not) to experimental events. Phase-sorted ERP images using different time and frequency windows represent different paths to fly through complex (single-channel) EEG data. (Note: Use keyword 'showwin' to image the time window used in sorting the data for any type of data-based sorting (e.g., by phase, amplitude, or mean value).

To see the phase sorting more clearly, keep the same settings, but this time enter *50* under *percent low-amp. trials to ignore*. Here, the 50% of trials with smallest 10-Hz (alpha) power in the selected time window will be rejected; only the (40) others (larger-alpha 50%) will be imaged. Here (below), we can better see how the alpha wave seems to resynchronize following the stimulus. Before time 0, alpha phase is more or less random (uniformly distributed) and there is little activity in the average ERP. At about 200 ms, alpha activity seems to (partially) synchronize with the stimulus and an N300 and P400 ERP appears.

Our interpretation (above) of these trials as representing phase synchronization need not be based on visual impression alone. To statistically assess whether alpha activity is partially resynchronized by (i.e., is partly phase-reset by) the stimuli, we need to plot the phase coherence (or phase-locking factor) between the stimulus sequence and the post-stimulus data. This measure, the *Inter-Trial Coherence (ITC)* our terminology, takes values between 0 and 1. A value of 1 for the time frequency window of interest indicates that alpha phase (in this latency window) is constant in every trial. A value of 0 occurs when the phase values in all trials are uniformly distributed around the unit circle. In practice, values somewhat larger than 0 are expected for any finite number of randomly phase-distributed trials.

Exploratory Step: Inter-Trial Coherence.

To plot the ITC in our ERP-image figure, we choose to enter the following parameters in the pop_erpimage.m window: we omit the *Percent low-amp. of Trials to ignore* value (or enter). Under Sort trials by phase>Frequency enter *9 11* and also enter *9 11* in the Inter-Trial Coherence>Frequency box. Enter *0.01* under *Signif. level* and press *OK*.

Note that these two entries must be equal (the window actually prevents the user from entering different values). Entering a frequency range instead of one frequency (e.g., *10* as before) tells erpimage.m to find the data frequency with maximum power in the input data (here between 9 and 11 Hz).

The following window is created.

Two additional plot panels appear below the ERP panel (uV). The middle panel, labeled *ERSP* for Event Related Spectral Power, shows mean changes in power across the epochs in dB. The blue region indicates 1% confidence limits according to surrogate data drawn from random windows in the baseline. Here, power at the selected frequency (10.12 Hz) shows no significant variations across the epoch. The number *25.93 dB* in the baseline of this panel indicates the absolute baseline power level.

Note: To compare results, it is sometimes useful to set this value manually in the main ERP-image pop-window.

The bottom plot panel shows the event-related Inter-Trial Coherence (ITC), which indexes the degree of phase synchronization of trials relative to stimulus presentation. The value *10.12 Hz* here indicates the analysis frequency selected. Phase synchronization becomes stronger than our specified p=0.01 significance cutoff at about 300 ms.

Note: The ITC significance level is typically lower when based on more trials. Moreover, ITC is usually not related to power changes.

Discussion Point: Does the ERP here arise through partial phase synchronization or reset following stimulus onset?

In a 'pure' case of (partial) phase synchronization:

- EEG power (at the relevant frequencies) remains constant in the post-stimulus interval.
- The ITC value is significant during the ERP, but less than 1 (complete phase locking).

In our case, the figure (above) shows a significant post-stimulus increase in alpha ITC accompanied by a small (though non-significant) increase in alpha power. In general, an ERP could arise from partial phase synchronization of ongoing activity combined with a stimulus-related increase (or decrease) in EEG power.

It is important not to over interpret the results of phase sorting in ERP-image plots. For example, the following calls from the Matlab command line simulate 256 1-s data epochs using Gaussian white noise, low-pass filters this below (simulated) 12 Hz, and draw the following 10-Hz phase sorted ERP-image plot of the resulting data. The figure appears to identify temporally coherent 10-Hz activity in the (actual) noise. The (middle) amplitude panel below the ERP-image plot shows, however, that amplitude at (simulated) 10 Hz does not change significantly through the (simulated) epochs, and the lowest panel shows that inter-trial coherence is also nowhere significant (as confirmed visually by the straight diagonal 10-Hz wave fronts in the center of the ERP image).

```
% Simulate 256 1-s epochs with Gaussian noise
% at 256-Hz sampling rate; lowpass < 12 Hz
>> data = eegfilt(randn(1,256*256),256,0,15);

% Plot ERP image, phase sorted at 10 Hz
>> figure;
>> erpimage(data,zeros(1,256),1:256,'Phase-sorted Noise',1,1,...
 'phasesort',[128 0 10],'srate',256,...
 'coher',[10 10 .01], 'erp','caxis',0.9);
```

-->-->

Taking epochs of white noise (as above) and adding a strictly time-locked 'ERP-like' transient to each trial will give a phase-sorted ERP-image plot showing a sigmoidal, not a straight diagonal wavefront signature. How can we differentiate between the two interpretations of the same data (random EEG plus ERP versus partially phase reset EEG)? For simulated one-channel data, there is no way to do so, since both are equally valid ways of looking at the same (simulated) data - no matter how it was created. After all, the simulated data themselves do not retain any impression of how they were created - even if such an impression remains in the mind of the experimenter!

For real data, we must use convergent evidence to bias our interpretation towards one or the other (or both) interpretations. The partial phase resetting model begins with the concept that the physical sources of the EEG (partial synchronized local fields) may ALSO be the sources of or contributors to average-ERP features. This supposition may be strengthened or weakened by examination of the spatial scalp distributions of the ERP features and of the EEG activity. However, here again, a simple test may not suffice since many cortical sources are likely to contribute to both EEG and averaged ERPs recorded at a single electrode (pair). An ERP feature may result from partial phase resetting of only one of the EEG sources, or it may have many contributions including truly 'ERP-like' excursions with fixed latency and polarity across trials, monopolar 'ERP-like' excursions whose latency varies across trials, and/or partial phase resetting of many EEG processes. Detailed spatiotemporal modeling of the collection of single-trial data is required to parcel out these possibilities. For further discussion of the question in the context of an actual data set, see Makeig et al. (2002). In that paper, phase resetting at alpha and theta frequencies was indicated to be the predominant cause of the recorded ERP (at least at the indicated scalp site, POz). How does the ERP in the figure above differ?

The Makeig et al. paper dealt with non-target stimuli, whereas for the sample EEGLAB dataset we used epochs time locked to target stimuli from one subject (same experiment). The phase synchronization might be different for the two types of stimuli. Also, the analysis in the paper was carried out over 15 subjects and thousands of trials, whereas here we analyze only 80 trials from one subject. (The sample data we show here are used for tutorial purposes. We are now preparing a full report on the target responses in these experiments.)

Note: Altogether, there are five trial sorting methods available in *erpimage()* ->

- Sort by the sorting variable (default) - Sorts input data trials (epochs) by the *sortvar*, sorting variable (for example, RT) input for each epoch of the input data.
- Sort by value (*valsort*)- Here, trials are sorted in order of their mean value in a given time window. Use this option to sort by ERP size (option not available yet in the interactive window).
- Sort by amplitude (*ampsort*)-- Trials are sorted in order of spectral amplitude or power at a specified frequency and time window. Use this option to display, for example, P300 responses sorted by alpha amplitude (option not available yet in the interactive window).
- Sort by phase (*phasesort*)-- Trials are sorted in order of spectral phase in a specified time/frequency window. </font>
- Do not sort (*nosort*)-- Display input trials in the same order they are input.

Plotting spectral amplitude in single trials and additional options

There are several other erpimage.m options that we will briefly illustrate in the following example. The *Image amps* entry on the pop_erpimage.m window allows us to image amplitude of the signal (at the frequency of interest) in the single trials, instead of the raw signals themselves. Check this box. The *Plot spectrum (minHz maxHz)* entry adds a small power spectrum plot to the top right of the figure. Enter *2 50* to specify the frequency limits for this graph.

Change the *Epoch-sorting field* box back to *latency* and *Event type</font>* back to *rt*. Then enter *500* under *Mark times* to plot a vertical mark at 500 ms (here for illustrative purpose only). Finally enter *-500 1500* under *Time limits* to zoom in on a specific time window, and *-3 3* under *Amplitude limits (dB)*.

The erpimage.m figure below appears.

In the next tutorial, we show how to use EEGLAB to perform and evaluate ICA decomposition of EEG datasets.

Chapter 09: Decomposing Data Using ICA

Independent Component Analysis of EEG data

Decomposing data by ICA (or any linear decomposition method, including PCA and its derivatives) involves a linear *change of basis* from data collected at single scalp channels to a spatially transformed "virtual channel" basis. That is, instead of a collection of simultaneously recorded single-channel data records, the data are transformed to a collection of simultaneously recorded outputs of spatial filters applied to the whole multi-channel data. These spatial filters may be designed in many ways for many purposes.

In the original scalp channel data, each row of the data recording matrix represents the time course of summed in voltage differences between source projections to one data channel and one or more reference channels (thus itself constituting a linear spatial filter). After ICA decomposition, each row of the data activation matrix gives the time course of the activity of one component process spatially filtered from the channel data.

In the case of ICA decomposition, the independent component filters are chosen to produce the maximally temporally independent signals available in the channel data. These are, in effect, *information sources* in the data whose mixtures, via volume conduction, have been recorded at the scalp channels. The mixing process (for EEG, by volume conduction) is passive, linear, and adds no information to the data. On the contrary, it mixes and obscures the functionally distinct and independent source contributions.

These information sources may represent synchronous or partialy synchronous activity within one (or possibly more) cortical patch(es), else activity from non-cortical sources (e.g., potentials induced by eyeball movements or produced by single muscle activity, line noise, etc.). The following example, from Onton and Makeig (2006), shows the diversity of source information typically contained in EEG data, and the striking ability of ICA to separate out these activities from the recorded channel mixtures.

Fifteen seconds of EEG data at 9 (of 100) scalp channels (top panel) with activities of 9 (of 100) independent components (ICs, bottom panel). While nearby electrodes (upper panel) record highly similar mixtures of brain and non-brain activities, ICA component activities (lower panel) are temporally distinct (i.e. maximally independent over time), even when their scalp maps are overlapping. Compare, for example, IC1 and IC3, accounting for different phases of eye blink artifacts produced by this subject after each visual letter presentation (grey background) and ensuing auditory performance feedback signal (colored lines). Compare, also, IC4 and IC7, which account for overlapping frontal (4-8 Hz) theta band activities appearing during a stretch of correct performance (seconds 7 through 15). Typical ECG and EMG artifact ICs are also shown, as well as overlapping posterior (8-12 Hz) alpha band bursts that appear when the subject waits for the next letter presentation (white background) For comparison, the repeated average visual evoked response of a bilateral occipital IC process (IC5) is shown (in red) on the same (relative) scale. Clearly the unaveraged activity dynamics of this IC process are not well summarized by its averaged response, a dramatic illustration of the

independence of phase-locked and phase-incoherent activity.

Running ICA decompositions

KEY STEP 9: Calculate ICA Components

To compute ICA components of a dataset of EEG epochs (or of a continuous EEGLAB dataset), select Tools > Run ICA. This calls the function pop_runica.m. To test this function, simply press *OK*.

We detail each entry of this GUI in detail below.

ICA Algorithms: Note (above) that EEGLAB allows users to try different ICA decomposition algorithms. Only *runica*, which calls runica.m and *jader* which calls the function jader.m (from Jean-Francois Cardoso) are a part of the default EEGLAB distribution. To use the *fastica* algorithm (Hyvarinen et al.), one must install the fastica toolbox and include it in the Matlab path. Details of how these ICA algorithms work can be found in the scientific papers of the teams that developed them. In general, the physiological significance of any differences in the results or different algorithms (or of different parameter choices in the various algorithms) have not been tested -- neither by us nor, as far as we know, by anyone else. Applied to simulated, relatively low dimensional data sets for which all the assumptions of ICA are exactly fulfilled, all three algorithms return near-equivalent components. We are satisfied that Infomax ICA (runica/binica) gives stable decompositions with up to hundreds of channels (assuming enough training data are given, see below), and therefore we can recommend its use, particularly in its faster binary form (binica.m). Note about *jader*: this algorithm uses 4th-order moments (whereas Infomax uses (implicitly) a combination of higher-order moments) but the storage required for all the 4th-order moments become impractical for datasets with more than ~50 channels. Note about *fastica*: Using default parameters, this algorithm quickly computes individual components (one by one). However, the order of the components it finds cannot be known in advance, and performing a complete decomposition is not necessarily faster than Infomax. Thus for practical purposes its name for it should not be taken literally. Also, in our experience it may be less stable than Infomax for high-dimensional data sets.

Very important note: We usually run ICA using many more trials that the sample decomposition presented here. As a general rule, finding *N*stable components (from N-channel data) typically requires *more than kN^2* data sample points (at each channel), where N^2 is the number of weights in the unmixing matrix that ICA is trying to learn and *k* is a multiplier. In our experience, the value of *k* increases as the number of channels increases. In our example using 32 channels, we have 30800 data points, giving 30800/32^2 = 30 pts/weight points. However, to find 256 components, it appears that even 30 points per weight is not enough data. In general, it is important to give ICA as much data as possible for successful training. Can you use too much data? This would only occur when data from radically different EEG states, from different electrode placements, or containing non-stereotypic noise were concatenated, increasing the number of scalp maps associated with independent time courses and forcing ICA to mixture together dissimilar activations into the N output components. The bottom line is: ICA works best when given a large amount of basically similar and mostly clean data. When the number of channels (N) is large (>>32) then a very large amount of data may be required to find N components. When insufficient data are available, then using the 'pca' option to jader.m to find fewer than N components may be the only good option.

Supported Systems for binica: To use the optional (and much faster) *binica*, which calls binica.m , the faster C translation of runica.m, you must make the location of the executable ICA file known to Matlab and executable on your system. Note: Edit the EEGLAB icadefs.m Matlab script file to specify the location of the binica.m executable. The EEGLAB toolbox includes three versions of the binary executable Informax ica routine, for Linux (compiled under Redhat 2.4), *freebsd* (3.0) and *freebsd* (4.0) (these are named, respectively *ica_linux2.4* , *ica_bsd3.0* and *ica_bsd4.0*). The executable file must also be accessible through the Unix user path variable otherwise binica.m won't work. Windows and sun version (older version of binary ICA executable) are available here (copy them to the EEGLAB directory). Please contact us to obtain the latest source code to compile it on your own system.

Channel types: It is possible to select specific channel types (or even a list of channel numbers) to use for ICA decomposition. For instance, if you have both EEG and EMG channels, you may want to run ICA on EEG channels only, since any relationship between EEG and EMG signals should involve propagation delays and ICA assumes an instantaneous relationship (e.g., common volume conduction). Use the channel editor to define channel types.

Running *runica* produces the following text on the Matlab command line:

```
Input data size [32,1540] = 32 channels, 1540 frames.
Finding 32 ICA components using logistic ICA.
Initial learning rate will be 0.001, block size 36.
Learning rate will be multiplied by 0.9 whenever angledelta >= 60 deg.
Training will end when wchange < 1e-06 or after 512 steps.
```

```
Online bias adjustment will be used.
Removing mean of each channel ...
Final training data range: -145.3 to 309.344
Computing the sphering matrix...
Starting weights are the identity matrix ...
Sphering the data ...
Beginning ICA training ...
step 1 - lrate 0.001000, wchange 1.105647
step 2 - lrate 0.001000, wchange 0.670896
step 3 - lrate 0.001000, wchange 0.385967, angledelta 66.5 deg
step 4 - lrate 0.000900, wchange 0.352572, angledelta 92.0 deg
step 5 - lrate 0.000810, wchange 0.253948, angledelta 95.8 deg
step 6 - lrate 0.000729, wchange 0.239778, angledelta 96.8 deg
...
step 55 - lrate 0.000005, wchange 0.000001, angledelta 65.4 deg
step 56 - lrate 0.000004, wchange 0.000001, angledelta 63.1 deg
Inverting negative activations: 1 -2 -3 4 -5 6 7 8 9 10 -11 -12 -13 -14 -15 -16 17 -18 -19 -20 -21 -22 -23 24 -25 -26 -27 -28 -29 -30 31 -
Sorting components in descending order of mean projected variance ...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
```

Note: the *runica* Infomax algorithm can only select for components with a supergaussian activity distribution (i.e., more highly peaked than a Gaussian, something like an inverted T). If there is strong line noise in the data, it is preferable to enter the option *'extended', 1* in the command line option box, so the algorithm can also detect subgaussian sources of activity, such as line current and/or slow activity.

Another option we often use is the stop option: try *'stop', 1E-7* to lower the criterion for stopping learning, thereby lengthening ICA training but possibly returning cleaner decompositions, particularly of high-density array data. We also recommend the use of collections of short epochs that have been carefully pruned of noisy epochs (see Rejecting artifacts with EEGLAB).

In the commandline printout, the *angledelta* is the angle between the direction of the vector in weight space describing the current learning step and the direction describing the previous step. An intuitive view of the annealing angle (*angledelta*) threshold (see above) is as follows: If the learning step takes the weight vector (in global weight vector space) 'past' the true or best solution point, the following step will have to 'back-track.' Therefore, the learning rate is too high (the learning steps are too big) and should be reduced. If, on the other hand, the learning rate were too low, the angle would be near 0 degrees, learning would proceed in (small) steps in the same direction, and learning would be slow. The default annealing threshold of 60 degrees was arrived at heuristically, and might not be optimum.

Note: the *runica* Infomax function returns two matrices, a data sphering matrix (which is used as a linear preprocessing to ICA) and the ICA weight matrix. For more information, refer to ICA help pages (i.e. http://www.sccn.ucsd.edu/~arno/indexica.html). If you wish, the resulting decomposition (i.e., ICA weights and sphere matrices) can then be applied to longer epochs drawn from the same data, e.g. for time-frequency decompositions for which epochs of 3-sec or more may be desirable.
The component order returned by *runica/binica* is in decreasing order of the EEG variance accounted for by each component. In other words, the lower the order of a component, the more data (neural and/or artifactual) it accounts for. In contrast to PCA, for which the first component may account for 50% of the data, the second 25%, etc..., ICA component contributions are much more homogeneous, ranging from roughly 5% down to ~0%. This is because PCA specifically makes each successive component account for as much as possible of the remaining activity not accounted for by previously determined components -- while ICA seeks *maximally independent* sources of activity.

PCA components are temporally or spatially orthogonal - smaller component projections to scalp EEG data typically looking like checker boards - while ICA components of EEG data are maximally temporally independent, but spatially unconstrained -- and therefore able to find maps representing the projection of a partially synchronized domain / island / patch / region of cortex, no matter how much it may overlap the projections of other (relatively independent) EEG sources. This is useful since, apart from ideally (radially) oriented dipoles on the cortical surface (i.e., on cortical gyri, not in sulci), simple biophysics shows that the volume projection of each cortical domain must project appreciably to much of the scalp.
Important note: Run twice on the same data, ICA decompositions under *runica/binica* will differ slightly. That is, the ordering, scalp topography and activity time courses of best-matching components may appear slightly different. This is because ICA decomposition starts with a random weight matrix (and randomly shuffles the data order in each training step), so the convergence is slightly different every time. Is this a problem? At the least, features of the decomposition that do not remain stable across decompositions of the same data should not be interpreted except as irresolvable ICA uncertainty.

Differences between decompositions trained on somewhat different data subsets may have several causes. We have not yet performed such repeated decompositions and assessed their common features - though this would seem a sound approach. Instead, in our recent work we have looked for commonalities between components resulting from decompositions from different subjects.

Plotting 2-D Component Scalp Maps

To plot 2-D scalp component maps, select Plot > Component maps > In 2-D. The interactive window (below) is then produced by function pop_topoplot.m . It is similar to the window we used for plotting ERP scalp maps. Simply press *OK* to plot all components.

Note: This may take several figures, depending on number of channels and the *Plot geometry* field parameter. An alternative is to call this functions several times for smaller groups of channels (e.g., *1:30* , *31:60* , etc.). Below we ask for the first 12 components (*1:12*) only, and choosing to set 'electrodes', 'off'.

The following topoplot.m window appears, showing the scalp map projection of the selected components. Note that the scale in the following plot uses arbitrary units. The scale of the component's activity time course also uses arbitrary units. However, the component's scalpmap values multiplied by the component activity time course is in the same unit as the data.

Learning to recognize types of independent components may require experience. The main criteria to determine if a component is 1) cognitively related 2) a muscle artifact or 3) some other type of artifact are, first, the scalp map (as shown above), next the component time course, next the component activity power spectrum and, finally (given a dataset of event-related data epochs), the erpimage.m. For example an expert eye would spot component 3 (above) as an eye artifact component (see also component activity by calling menu Plot > Component activations (scroll)). In the window above, click on scalp map number 3 to pop up a window showing it alone (as mentioned earlier, your decomposition and component ordering might be slightly different).

Note: From EEGLAB v4.32 on, if the electrode montage extends below the horizontal plane of the head center, topoplot.m plots them as a 'skirt' (or halo) around the cartoon head that marks the (arc_length = 0.5) head-center plane. (Note: Usually, the best-fitting sphere is a cm or more above the plane of the nasion and ear canals). By default, all channels with location arc_lengths <= 1.0 (head bottom) are used for interpolation and are shown in the plot. From the commandline, topoplot.m allows the user to specify the interpolation and plotting radii (*intrad* and *plotrad*) as well as the radius of the cartoon head (*headrad*). The *headrad* value should normally be kept at its physiologically correct value (0.5). In 'skirt' mode (see below), the cartoon head is plotted at its normal location relative to the electrodes, the lower electrodes appearing in a 'skirt' outside the head cartoon. If you have computed an equivalent dipole model for the component map (using the DIPFIT plug-in) topoplot.m can indicate the location of the component equivalent current dipole(s). Note that the 'balls' of the dipole(s) are located as if looking down from above on the transparent head. The distance of the electrode positions from the vertex, however, is proportional to their (great circle) distance on the scalp to the vertex. This keeps the electrodes on the sides of the head from being bunched together as they would be in a top-down view of their positions. This great-circle projection spreads out the positions of the lower electrodes. Thus, in the figure below, the (bottom) electrodes plotted on the lower portion of the 'skirt' are actually located on the back of the neck. In the plot, they appear spread out, whereas in reality they are bunched on the relatively narrow neck surface. The combinations of top-down and great-circle projections allows the full component projection (or raw data scalp map) to be seen clearly, while allowing the viewer to estimate the actual 3-D locations of plot features.

The EEGLAB v4.32 topoplot.m above shows an independent component whose bilateral equivalent dipole model had only 2% residual variance across all 252 electrode locations. This binica.m decomposition used PCA to reduce the over 700,000 data points to 160 principal dimensions (a ratio of 28 time points per ICA weight).

Plotting component headplots

Using EEGLAB, you may also plot a 3-D head plot of a component topography by selecting Plot > Component maps > In 3-D. This calls pop_headplot.m. The function should automatically use the spline file you have generated when plotting ERP 3-D scalp maps. Select one ore more components (below) and press *OK*. For more information on this interface and how to perform coregistration, see the Plotting ERP Data in 3-D and the DIPFIT.

The pop_headplot.m window below appears. You may use the Matlab rotate 3-D option to rotate these headplots with the mouse. Else, enter a different *view* angle in the window above.

Studying and removing ICA components

To study component properties and label components for rejection (i.e. to identify components to subtract from the data), select Tools > Reject data using ICA > Reject components by map. The difference between the resulting figure(s) and the previous 2-D scalp map plots is that one can here plot the properties of each component by clicking on the rectangular button above each component scalp map.

For example, click on the button labeled *3*. This component can be identified as an eye artifact for three reasons:

1. The smoothly decreasing EEG spectrum (bottom panel) is typical of an eye artifact;
2. The scalp map shows a strong far-frontal projection typical of eye artifacts; And,
3. It is possible to see individual eye movements in the component erpimage.m (top-right panel).

Eye artifacts are (nearly) always present in EEG datasets. They are usually in leading positions in the component array (because they tend to be big) and their scalp topographies (if accounting for lateral eye movements) look like component 3 or perhaps (if accounting for eye blinks) like that of component 10 (above). Component property figures can also be accessed directly by selecting Plot > Component properties. (There is an equivalent menu item for channels, Plot > Channel properties). Artifactual components are also relatively easy to identify by visual inspection of component time course (menu Plot > Component activations (scroll) --- not shown here).

Since this component accounts for eye activity, we may wish to subtract it from the data before further analysis and plotting. If so, click on the bottom green *Accept* button (above) to toggle it into a red *Reject* button (note: at this point, components are only marked for rejection; to subtract marked components, see next section 'Subtracting ICA components from data'). Now press *OK* to go back to the main component property window.

Another artifact example in our decomposition is component 32, which appears to be typical muscle artifact component. This components is spatially localized and show high power at high frequencies (20-50 Hz and above) as shown below.

Artifactual components often encountered (but not present in this decomposition) are single-channel (channel-pop) artifacts in which a single channel goes 'off,' or line-noise artifacts such as 23 (the ERP image plot below shows that it picked up some noise line at 60 Hz especially in trials 65 and on).

Many other components appear to be brain-related (Note: Our sample decomposition used in this tutorial is based on clean EEG data, and may have fewer artifactual components than decompositions of some other datasets). The main criteria for recognizing brain-related components are that they have:

1. Dipole-like scalp maps,
2. Spectral peaks at typical EEG frequence is (i.e., 'EEG-like' spectra) and,
3. Regular ERP-image plots (meaning that the component does not account for activity occurring in only a few trials).

The component below has a strong alpha band peak near 10 Hz and a scalp map distribution compatible with a left occipital cortex brain source. When we localize ICA sources using single-dipole or dipole-pair source localization. Many of the 'EEG-like' components can be fit with very low residual variance (e.g., under 5%). See the tutorial example for either the EEGLAB plug-in DIPFIT or for the BESA plug-in for details.

What if a component looks to be "half artifact, half brain-related"? In this case, we may ignore it, or may try running ICA decomposition again on a cleaner data subset or using other ICA training parameters. As a rule of thumb, we have learned that removing artifactual epochs containing one-of-a-kind artifacts is very useful for obtaining 'clean' ICA components.

Important note: we believe an optimal strategy is to:

1. Run ICA
2. Reject bad epochs (see the functions we developed to detect artifactual epochs and channels, if any, in the tutorial on artifact rejection). In some cases, we do not hesitate to remove more than 10% of the trials, even from 'relatively clean' EEG datasets. We have learned that it is often better to run this first ICA composition on very short time windows.
3. Run ICA a second time on the 'pruned' dataset.
4. Apply the resulting ICA weights to the same dataset or to longer epochs drawn from the same original (continuous or epoched) dataset. For instance, to copy ICA weights and sphere information from dataset 1 to 2: First, call menu Edit > Dataset info of

dataset 2. Then enter *ALLEEG(1).icaweights* in the *ICA weight array ...* edit box, *ALLEEG(1).icasphere* in the *ICA sphere array ...* edit box, and press *OK*.

Subtracting ICA components from data

We don't always subtract whole components from our datasets because we study individual components' activity. However, when we want to remove components, we use menu Tools > Remove components, which calls the pop_subcomp.m function. The component numbers present by default in the resulting window (below) are those marked for rejection in the previous Tools > Reject using ICA > Reject components by map component rejection window (using the *Accept*/*Reject* buttons). Enter the component numbers you wish to reject and press *OK*.

A window will pop up, plotting channel ERP before (in blue) and after (in red) component(s) subtraction and asking you if you agree with the new data ERP.

Press *Yes*. A last window will pop up asking you if you want to rename the new data set. Give it a name and again press *OK*.

Note that storing the new dataset in Matlab memory does not automatically store it permanently on disk. To do this, select File > Save current dataset. Note that we will pursue with all components, so you can appreciate the contribution of artifactual components to the ERP. Go back to the previous dataset using the Dataset top menu.

Note: If you try to run ICA on this new dataset, the number of dimensions of the data will have been reduced by the number of components subtracted. We suggest again 'baseline-zeroing' the data (if it is epoched when some components have been removed, data epoch-baseline means may change). To run ICA on the reduced dataset, use the *pca* option under the Tools > Run ICA pop-up window, type *'pca', '10'* in the Commandline options box to reduce the data dimensions to the number of remaining components (here 10), before running ICA (see runica.m. If the amount of data has not changed, ICA will typically return the same (remaining) independent components -- which were, after all, already found to be maximally independent for these data.

Retaining multiple ICA weights in a dataset

To retain multiple copies of ICA weights (e.g. EEG.weights and EEG.sphere), use the extendibility property of Matlab structures. On the Matlab command line, simply define new weight and sphere variables to retain previous decomposition weights. For example,

```
>> EEG.icaweights2 = EEG.icaweights; % Store existing ICA weights matrix
>> EEG.icasphere2 = EEG.icasphere; % Store existing ICA sphere matrix
>> [ALLEEG EEG] = eeg_store(ALLEEG, EEG, CURRENTSET); % copy to EEGLAB memory
>> EEG = pop_runica(EEG);  % Compute ICA weights and sphere again using
% binica() or runica(). Overwrites new weights/sphere matrices
% into EEG.icaweights, EEG.icasphere
>> [ALLEEG EEG] = eeg_store(ALLEEG, EEG, CURRENTSET); % copy to EEGLAB memory
```

-->-->

Both sets of weights will then be saved when the dataset is saved, and reloaded when it is reloaded. See the script tutorial for more information about writing Matlab scripts for EEGLAB.

Scrolling through component activations

To scroll through component activations (time courses), select Plot > Component activations (scroll). Scrolling through the ICA activations, one may easily spot components accounting for characteristic artifacts. For example, in the scrolling eegplot.m below, component 1 appears to account primarily for blinks, components 7 and 31 for lateral eye movements, and component 25 possibly for muscle activity. Check these classifications using the complementary visualization produced by Plot > Component properties.

In the next tutorial, we show more ways to use EEGLAB to study ICA components of the data.

Chapter 10: Working with ICA components

Rejecting data epochs by inspection using ICA

To reject data by visual inspection of its ICA component activations, select Tools > Reject data using ICA > Reject by inspection (calling pop_eegplot.m. Note that marked epochs can be kept in memory before being actually rejected (if the option *Reject marked trials (checked=yes)* below is not set). Thus it is possible to mark trials for rejection and then to come back and update the marked trials. The first option (*Add to previously marked rejections (checked=yes)*) allows us to include previous trial markings in the scrolling data window or to ignore/discard them. Since it is the first time we have used this function on this dataset, the option to plot previously marked trials won't have any effect. Check the checkbox to reject marked trials, so that the marked trials will be immediately rejected when the scrolling window is closed. Press *OK*.

First, adjust the scale by entering *10* in the scale text edit box (lower right). Now, click on the data epochs you want to mark for rejection. For this exercise, highlight two epochs. You can then deselect the rejected epochs by clicking again on them. Press the *Reject* button when finished and enter a name for the new dataset (the same dataset minus the rejected epochs), or press the *Cancel* button to cancel the operation.

At this point, you may spend some time trying out the advanced rejection functions we developed to select and mark artifactual epochs based on ICA component maps and activations. For directions, see the Data rejection tutorial. Then, after rejecting 'bad' epochs, run ICA decomposition again. Hopefully, by doing this you may obtain a 'cleaner' decomposition.

Important note:what do we mean by a cleaner ICA decomposition? ICA takes all its training data into consideration. When too many types (i.e., scalp distributions) of 'noise' - complex movement artifacts, electrode 'pops', etc -- are left in the training data, these unique/irreplicable data features will 'draw the attention' of ICA, producing a set of component maps including many single-channel or 'noisy'-appearing components. The number of components (degrees of freedom) devoted to decomposition of brain EEG alone will be correspondingly reduced. Therefore, presenting ICA with as much 'clean' EEG data as possible is the best strategy (note that blink and other stereotyped EEG artifacts do not necessarily have to be removed since they are likely to be isolated as single ICA components). Here 'clean' EEG data means data after removing noisy time segments (does not apply to removed ICA components).

For this tutorial, we decide to accept our initial ICA decomposition of our data and proceed to study the nature and behavior(s) of its independent components. First, we review a series of functions whose purpose is to help us determine which components to study and how to study them.

Plotting component spectra and maps

It is of interest to see which components contribute most strongly to which frequencies in the data. To do so, select Plot > Component spectra and maps. This calls pop_spectopo.m. Its first input is the epoch time range to consider, the forth is the percentage of the data to

sample at random (smaller percentages speeding the computation, larger percentages being more definitive). Since our EEG dataset is fairly small, we choose to change this value to *100* (= all of the data). We will then visualize which components contribute the most at 10 Hz, entering *10* in the *Scalp map frequency* text box. We simply scan all components, the default in *Components to consider*. Press *OK*.

The spectopo.m window (below) appears.

In the previous window, we plotted the spectra of each component. A more accurate strategy (for technical reasons) is to plot the data signal minus the component activity and estimate the decrease in power in comparison to the original signal at one channel (it is also possible to do it at all channel but it requires to compute the spectrum of the projection of each component at each channel which is computationally intensive). To do so, go back to the previous interactive window, choose explicitly to plot component's contribution at channel *27* (POz) where power appears to be maximum at *10* Hz using the *Electrode number to analyze ...:* field, uncheck the checkbox *[checked] compute component spectra....* Set percent to *100* as before. Finally we will display *6* component maps instead of 5 (default) (note that all component spectra will be shown) and we will set the maximum frequency to be plotted at *30* Hz using the *Plotting frequency range* option in the bottom panel (below). Press *OK* when done.

The spectopo.m figure appears (as below).

The following text is displayed

Component 1 percent variance accounted for: 3.07
Component 2 percent variance accounted for: 3.60
Component 3 percent variance accounted for: -0.05
Component 4 percent variance accounted for: 5.97
Component 5 percent variance accounted for: 28.24
Component 6 percent variance accounted for: 6.15
Component 7 percent variance accounted for: 12.68
Component 8 percent variance accounted for: -0.03
Component 9 percent variance accounted for: 5.04
Component 10 percent variance accounted for: 52.08
Component 11 percent variance accounted for: 0.79
....

"Percent variance acounted for" (pvaf) compares the variance of the data MINUS the (back-projected) component to the variance of the whole data. Thus, if one component accounts for all the data, the data minus the component back-projection will be 0, and pvaf will be 100%; If the component has zero variance, it accounts for none of the data and pvaf = 0%. If a component somehow accounts for the NEGATIVE of the data, however, pvaf will be larger than 100% (meaning: "If you remove this component, the data actually get larger, not smaller!"). According to the variance accounted for output above, component 10 accounts for more than 50% of power at 10 Hz for channel POz. (Note: A channel number has to be entered otherwise component contributions are not computed).

Plotting component ERPs

After seeing which components contribute to frequency bands of interest, it is interesting to look at which components contribute the most to the ERP. A first step is to view the component ERPs. To Plot component ERPs, select Plot > Component ERPs > In rectangular array, which calls function pop_plotdata.m. Then press *OK*.

The plotdata.m window below pops up, showing the average ERP for all 31 components.

Click on the component-1 trace (above) to plot this trace in new window (as below).

As for electrodes, use menu Plot > Sum/Compare comp. ERPs to plot component ERP differences accross multiple datasets.

Plotting component ERP contributions

To plot the contribution of component ERPs to the data ERP, select Plot > Component ERPs > with component maps, which calls pop_envtopo.m. Simply press *OK* to plot the 7 components that contribute the most to the average ERP of the dataset. Note artifactual components can be subtracted from the data prior to plot the ERP using the *Indices of component to subtract ...* edit box.

In the envtopo.m plot (below), the black thick line indicates the data envelope (i.e. minimum and maximum of all channel at every time point) and the colored show the component ERPs.

The picture above looks messy, so again call the pop_envtopo.m window and zoom in on time range from *200* ms to *500* ms post-stimulus, as indicated below.

We can see (below) that near 400 ms component 1 contributes most strongly to the ERP.

On the command line, the function also returns the percent variance accounted for by each component:

```
IC4 pvaf: 31.10%
IC2 pvaf: 25.02%
IC3 pvaf: 16.92%
...
```

Component ERP-image plotting

To plot ERP-image figures for component activations, select Plot > Component ERP image (calling pop_erpimage.m. This function works exactly as the one we used for plotting channel ERP images, but instead of visualizing activity at one electrode, the function here plots the activation of one component. Enter the following parameters in the interactive window to sort trials by phase at 10 Hz and 0 ms, to image reaction time, power and Inter-Trial Coherence (see the ERP-image tutorial for more information).

For component 6 (below) we observe in the erpimage.m figure that phase at the analysis frequency (9Hz to 11Hz) is evenly distributed in the

time window -300 to 0 ms (as indicated by the bottom trace showing the inter-trial coherence (ITC) or phase-locking factor). This component accounts for much of the EEG power at 10 Hz, but for little if any of the average ERP. Overall, mean power at the analysis frequency does not change across the epoch (middle blue trace) and phase at the analysis frequency is not reset by the stimulus (bottom blue trace). Here again, the red lines show the bootstrap significance limits (for this number of trials).

Note: As scale and polarity information is distributed in the ICA decomposition (*not* lost!) between the projection weights (column of the inverse weight matrix, *EEG.icawinv*) and rows of the component activations matrix (*EEG.icaact*), the absolute amplitude and polarity of component activations are meaningless and the activations have no unit of measure (through they are *proportional to* microvolt). To recover the absolute value and polarity of activity accounted for by one or more components at an electrode, image the back-projection of the component activation(s) at that channel. Go back to the previous ERP-image window, use the same parameters and set *Project to channel #* to 27. Note that the ERP is reversed in polarity and that absolute unit for power has changed.

In the next tutorial, we show how to use EEGLAB to perform and visualize time/frequency decompositions of channel activities or independent component activations.

Chapter 11: Time / Frequency decomposition

Time/frequency decomposition

Time/frequency analysis characterizes changes or perturbations in the spectral content of the data considered as a sum of windowed sinusoidal functions (i.e. sinusoidal wavelets). There is a long history and much recent development of methods for time/frequency decomposition. The methods used in the basic EEGLAB functions are straightforward. Their mathematical details are given in a reference paper (Delorme and Makeig, in press).

Decomposing channel data

KEY STEP 10: ERSP and ITC

To detect transient *event-related spectral perturbation,* or ERSP, (Makeig, 1993) (event-related shifts in the power spectrum) and inter-trial coherence (ITC) (event-related phase-locking) events in epoched EEGLAB datasets, select Plot > Time frequency transforms > Channel time-frequency (calling pop_timef.m. Below, we enter *14* (Cz) for the *Channel number*, *.01* for the *Bootstrap significance level*, and set the optional parameter *padratio* to *16* as below (a very high over-sampling factor for frequencies, giving a smooth looking plot at some computational cost). We let the other defaults remain and press *OK*.

The timef.m window below appears. The *top image* shows mean event-related changes in spectral power (from pre-stimulus baseline) at each time during the epoch and at each frequency (< 50 Hz). To inspect these changes more closely, click on the color image. A new window will pop up. Enabling Matlab zoom allows zooming in to any desired time/frequency window. The ERSP image shows a brief but significant decrease in power at about 370 ms at 8 Hz (click on the image to zoom in and determine the exact frequency), a power increase centered at 13.5 Hz and starting at 300 ms. More spectral changes occur at 20-28 Hz. Note that the method used to asses significance is based on random data shuffling, so the exact significance limits of these features may appear slightly different.

The *upper left panel* shows the baseline mean power spectrum, and the lower part of the upper panel, the ERSP envelope (low and high mean dB values, relative to baseline, at each time in the epoch).

The *lower image* shows is the Inter-Trial coherence (ITC) at all frequencies. We previously encountered the ITC when we explained the ERP_image_plotting.m function. A significant ITC indicates that the EEG activity at a given time and frequency in single trials becomes phase-locked (not phase-random with respect to the time-locking experimental event). Here, a significant ITC appears briefly at about 10 Hz (at about the same time as the power increase in the *upper panel*), but this effect might well become insignificant using a more rigorous significance threshold. Note that the time/frequency points showing significant ITC and ERSP are not necessarily identical. In this plot, ITC hardly reaches significance and cannot be interpreted. The help message for the timef.m function contains information about all its parameters, the images and curve plotted, and their meanings.

Computing component time/frequency transforms

It is more interesting to look at time-frequency decompositions of component activations than of separate channel activities, since independent components may directly index the activity of one brain EEG source, whereas channel activities sum potentials volume-conducted from different parts of the brain.

To plot a component time-frequency transform, we select Plot > Time/frequency transforms > Component time-frequency (calling pop_timef.m. Enter *10* for the *Component number* to plot, *[-500 1000]* for the "Epoch time range", (FFT) for *Wavelet cycles*, and *.01* for the *Bootstrap significance level*. Note that Morlet wavelets are used by default although it is also possible to use sinusoidal wavelets. We change *padratio* to *16* and add the optional argument *'maxfreq', '30'* to visualize only frequencies up to 30 Hz. Again, we press *OK*.

Note: pop_timef.m decompositions using FFTs allow computation of lower frequencies than wavelets, since they compute as low as one cycle per window, whereas the wavelet method uses a fixed number of cycles (default 3) for each frequency.

The following timef.m window appears. The ITC image (*lower panel*) shows strong synchronization between the component activity and stimulus appearance, first near 15 Hz then near 4 Hz. The ERSP image (*upper panel*) shows that the 15-Hz phase-locking is followed by a 15-Hz power increase, and that the 4-Hz phase-locking event is accompanied by, but outlasts, a 4-Hz power increase. Note that the appearance of oscillatory activity in the ERP (*trace under bottom ITC image*) before and after the stimulus is not significant according to ITC.

Computing component cross-coherences

To determine the degree of synchronization between the activations of two components, we may plot their event-related cross-coherence (a concept first demonstrated for EEG analysis by Rappelsberger). Even though independent components are (maximally) independent over the whole time range of the training data, they may become transiently (partially) synchronized in specific frequency bands. To plot component cross-coherence, select Plot > Time-frequency transforms > Component cross-coherence, which calls pop_crossf.m. Below, we enter components *4* and *9* (Use any components in your decomposition), *Bootstrap significance level* to *0.01*, set *padratio* to *16*. We again press *OK*.

In the crossf.m window below, the two components become synchronized (*top panel*) around 11.5 Hz (click on the image to zoom in). The *upper panel* shows the coherence magnitude (between 0 and 1, 1 representing two perfectly synchronized signals). The *lower panel* indicates the phase difference between the two signals at time/frequency points where cross-coherence magnitude (in the *top panel*) is significant. In this example, the two components are synchronized with a phase offset of about -120 degrees (this phase difference can also be plotted as latency delay in ms, using the minimum-phase assumption. See crossf.m help for more details about the function parameters and the plotted variables).

One can also use Plot > Time-frequency transforms > Channel cross-coherence to plot event-related cross-coherence between a pair of scalp channels, but here relatively distant electrode channels may appear synchronized only because a large EEG source projects to both of them. Other source confounds may also affect channel coherences in unintuitive ways. Computing cross-coherences on independent data components may index transient synchronization between particular cortical domains.

Plotting ERSP time course and topography

Recall that spectral perturbations at a single-analysis frequency and channel or component in the single epochs (sorted by some relevant value) can be imaged using erpimage.m or by selecting Plot > Component|Channel ERP image.

Called from the command line (see EEGLAB script writing), the timef.m and crossf.m routines can return the data for each part of their figures as a Matlab variable. Accumulating the ERSP and ITC images (plus the ERSP baseline and significance-level information) for all channels (e.g., via an EEGLAB script) allows use of another toolbox routine, tftopo.m (currently not available from the EEGLAB menu).

In the next tutorial, we show more about how to import data and events into EEGLAB datasets.

Chapter 12: Multiple Datasets

Processing Multiple Datasets

Processing multiple datasets sequentially and automatically is important for analysing data. While early versions of EEGLAB exclusively relied on command line scripting for processing multiple datasets, some automated processing is now available directly from the EEGLAB graphic user interface (gui). To explore this capability, you must first select several datasets. For this section, we will use data from the STUDY tutorial.

To work on multiple datasets at once, it is first necessary to select them. To do so, use menu item Dataset > Select multiple datasets

As in the screen view above, select a few datasets. Note that the behavior of EEGLAB will differ depending on your optional settings under File > Save memory. If you allow only one dataset to be present in memory at a time (see Memory options for more details), existing datasets will be automatically overwritten *on disk*. However, if you allow all datasets to be present in memory simultaneously, only the datasets in memory will be overwritten and their copies in disk files will not be affected (you may then select menu item File > Save current dataset(s) to save all the currently selected datasets).

EEGLAB functions available through the EEGLAB menu that can process multiple datasets can be seen in the Tools menu. When there are multiple current datasets, menu items unable to process multiple datasets are disabled. Currently (v6.x-), functions that can process multiple datasets include functions that resample the data, filter the data, re-reference the data, remove channel baselines, and run ICA. If all the datasets have the same channel locations, you can also locate equivalent dipoles for independent components of multiple datasets.

All available tools process data in a similar way. Upon menu selection, a menu window pops up (identical to the single dataset window) in which you may select processing parameters that are then applied to all the datasets. If the dataset copies on disk are overwritten (under the only-one-set-in-memory option), then a warning window will appear. For example, selecting the Tools > Change sampling rate menu item pops up the following interface.

To resample all datasets at 125 Hz, enter 125 then *OK*. If the current datasets have to be resaved to disk (because at most one dataset can be present in memory), the following warning window appears:

The graphic interface for running ICA is a bit more elaborate. Select menu item Tools > Run ICA. The following window will appear.

The statistical tools in EEGLAB for evaluating STUDY condition, session, and group measure differences assume that datasets for difference conditions recorded in the same session share the same ICA decomposition, i.e. the same independent component maps. By default, pop_headplot.m will concatenate STUDY datasets from the same subject and session. For example, you may have several datasets time locked to different classes of events, constituting several experimental conditions per subject, all collected in the same session with the same electrode montage. By default (leaving the lowest check-box checked), pop_headplot.m will perform ICA decomposition on the concatenated data trials from these datasets, and will then attach the same ICA unmixing weights and sphere matrices to each dataset. Information about the datasets selected for concatenation will be provided on the Matlab command line before beginning the decomposition. </font>

In some cases, concatenating epoched datasets representing multiple conditions collected in the same session may involve replicating some portions of the data. For example, the pre-stimulus baseline portion of an epoch time locked to a target stimulus may contain some portion of an epoch time locked to a preceding nontarget stimulus event. Infomax ICA performed by pop_headplot.m and pop_headplot.m does not consider the time order of the data points, but selects time points in random order during training. Thus, replicated data points in concatenated datasets will only tend to be used more often during training. However, this may not bias the forms of the resulting components in other than unusual circumstances.

Some other blind source decomposition algorithms such as pop_headplot.m do consider the time order of points in brief data windows. The version of the pop_headplot.m function used in EEGLAB has been customized to avoid selecting data time windows that straddle an epoch boundary. To apply pop_headplot.m to concatenated datasets, however, the epoch lengths of the datasets are assumed to be equal.
If you wish (and have enough computer RAM), you may also ask pop_headplot.m to load and concatenate all datasets before running ICA. This will concatenate all the datasets in you computer main memory, so you actually need to have enough memory to contain all selected datasets. We do not recommend this approach, since it tacitly (and unreasonably) assumes that the very same set of brain and non-brain source locations and, moreover, orientations, plus the very same electrode montage exist in each session and/or subject.

After ICA decomposition of all selected datasets, you may use menu item File > Create Study > Using all loaded datasets to create a study using all loaded datasets (if you only want to use the dataset you selected, you will have to remove the other datasets from the list of datasets to include in the STUDY). Using a STUDY, you may cluster ICA components across subjects. See the multiple subjects processing tutorial for more details.

Concluding remark on data tutorial

This tutorial only gives a rough idea of the utility of EEGLAB for processing single-trial and averaged EEG or other electrophysiological data, The analyses of the sample dataset(s) presented here are by no way definitive. One should examine the activity and scalp map of each independent component carefully, noting its dynamics and its relationship to behavior, and to other component activities, to begin to understand its role and possible biological function.

Further questions may be more important: How are the activations of pairs of maximally independent components inter-related? How do their relationships evolve across time or change across conditions? Are independent components of different subjects related, and if so, how? EEGLAB is a suitable environment for exploring these and other questions about brain dynamics.

In the next tutorial, we show more about how to import data and events into EEGLAB datasets.

II.Multiple subject processing tutorial

The EEGLAB Tutorial is split into four parts. The second part is the Multiple Subject processing tutorial. In this tutorial, the user is introduced to more advanced elements of EEGLAB focused around the STUDY framework used for processing multiple subjects as once. These topics are recommended for the user to successfully analyse large scale experiments which go beyond single trials. As of 2/10/2012, there are eight topics in the Multi-subject processing tutorial.

Chapter 01: Multiple Subject Proccessing Overview

EEGLAB Studysets and Independent Component Clustering

This tutorial describes how to use a new (EEGLAB v5.0-) structure, the STUDY, to manage and process data recorded from multiple subjects, sessions, and/or conditions of an experimental study. EEGLAB uses studysets for performing statistical comparisons, for automated serial (and in future parallel) computation, and for clustering of independent signal components across subjects and sessions. It details the use of a new (v5.0-) set of EEGLAB component clustering functions that allow exploratory definition and visualization of clusters of equivalent or similar ICA data components across any number of subjects, subject groups, experimental conditions, and/or sessions. Clustering functions may be used to assess the consistency of ICA (or, other linearly filtered) decompositions across subjects and conditions, and to evaluate the separate contributions of identified clusters of these data components to the recorded EEG dynamics.

EEGLAB STUDY structures and studysets:

EEGLAB v5.0 introduced a new basic concept and data structure, the *STUDY*. Each *STUDY*, saved on disk as a *studyset* (*.std*) file, is a structure containing a set of epoched *EEG* datasets from one or more subjects, in one or more groups, recorded in one or more sessions, in one or more task conditions -- plus additional (e.g., component clustering) information. In future EEGLAB versions, *STUDY* structures and studysets will become primary EEGLAB data processing objects. As planned, operations now carried out from the EEGLAB menu or the Matlab command line on datasets will be equally applicable to studysets comprising any number of datasets.

Use of STUDY structures to process single-trial channel data:

EEGLAB studysets may be used to compute ERPs, spectrum, ERSP and other measures onto single-trial channel data across dozens or even hundreds of subjects. Missing data channels may be replaced if necessary using spherical interpolation. Parametric or bootstrap statistics may be used with correction for multiple comparisons to compare a given measure in any n x m design. The channel data may also be used to compute ICA component projections (see below).

Use of STUDY structures to cluster ICA components:

EEGLAB studysets may be used to cluster similar independent components from multiple sessions and to evaluate the results of clustering. As ICA component clustering is a powerful tool for electrophysiological data analysis, and a necessary tool for applying ICA to experimental studies involving planned comparisons between conditions, sessions, and/or subject groups, the *STUDY* concept has been applied first to independent component clustering. A small *studyset* of five datasets, released with EEGLAB v5.0b for tutorial use and available here, has been used to create the example screens below. We recommend that after following the tutorial using this small example studyset, users next explore component clustering by forming EEGLAB studies for one or more of their existing experimental studies testing the component clustering functions more fully on data they know well by repeating the steps outlined below.

Upgrades to several standard EEGLAB plotting functions also allow them to be applied simultaneously to whole studysets (either sequentially or in parallel) rather than to single datasets, for example allowing users to plot grand average channel data measures (ERPs, channel spectra, etc.) across multiple subjects, sessions, and/or conditions from the EEGLAB menu.

The dataset information contained in a *STUDY* structure allows straightforward statistical comparisons of component activities and/or source models for a variety of experimental designs. Currently, only a few two-condition comparisons are directly supported. Currently we are writing Matlab functions that will process information in more general *STUDY* structures and the *EEG* data structures they contain, potentially applying several types of statistical comparison (ANOVA, permutation-based, etc.) to many types of data measures.

Matlab toolboxes required for component clustering:

Currently, two clustering methods are available: 'kmeans' and 'neural network' clustering. 'Kmeans' clustering requires the Matlab Statistical Toolbox, while 'neural network' clustering uses a function from the Matlab Neural Network Toolbox. To learn whether these toolboxes are installed, type *>> help* on the Matlab command line and see if the line *toolbox/stats - Statistical toolbox* and/or the line *nnet/nnet - Neural Network toolbox* are present. In future, we plan to explore the possibility of providing alternate algorithms that do not require these toolboxes, as well as options to cluster components using other methods.

This tutorial assumes that readers are already familiar with the material covered in the single subject EEGLAB tutorial and also (for the later part of this chapter) in the EEGLAB script writing tutorial.

Chapter 02: STUDY Creation

This part of the tutorial will demonstrate how to create an EEGLAB STUDY. For these datasets, the measures needed for data visualization, clustering and cluster visualization have already been computed, as described below in the following section, and are stored with the datasets - download the full studyset here (~450Mb).

Description of experiment for tutorial data: These data were acquired by Peter Ullsberger and colleagues from five subjects performing an auditory task in which they were asked to distinguish between synonymous and non-synonymous word pairs (the second word presented 1 second after the first). Data epochs were extracted from 2 sec before the second word onset to 2 sec after the second word onset. After decomposing each subject's data by ICA, two EEG datasets were extracted, one (Synonyms) comprising trials in which the second word was synonymous with the first one, and one (Non-synonyms) in which the second word was not a synonym of the first. Thus the study includes 10 datasets, two condition datasets for each of five subjects. Since both datasets of each subject were recorded in a single session, the decompositions and resulting independent component maps are the same across both conditions for each subject.

After downloading the sample STUDY data, unzip it in a folder of your choice (preferably in the 'sample_data' sub-folder of the EEGLAB release you are currently using; under Linux use the 'unzip' command). This will create a sub-folder '5subjects' containing several studysets. Then open a Matlab session and run >> *eeglab*.

Data organization

The tutorial data is already optimally organized. However, when creating a new STUDY, it is preferable to organize your data before running the STUDY functions. We suggest creating one directory or folder per subject, then storing the EEG dataset (".set") files for that subject in this folder. The STUDY functions will then automatically add measure files to the same subject directories.

We also advise modifying the default EEGLAB memory options. Selecting menu item File > Memory options will pop-up the following window:

Set the first option so that no more than one dataset is stored in memory at a time. Dataset data arrays are then read from disk whenever EEGLAB requires access to the data, but without cluttering memory. This will allow Matlab to load and hold a large number of dataset structures forming a large STUDY. Also, unset the third option so ICA component activations do not get recomputed automatically. This saves time as datasets are re-saved automatically during computation of measures and clustering of ICA components.

Creating a new STUDY structure and studyset

Exploratory Step: Creating a Studyset

> To create a studyset, select menu item File > Create study > Browse for datasets. (Else, you may load into EEGLAB all the datasets you want to include in the study and select menu item File > Create study > Using all loaded datasets). A blank interface similar to the one described above will appear. In this window, first enter a name for the studyset ('N400STUDY'), a short description of the study ('Auditory task: Synonyms Vs. Non-synonyms, N400'). Here, we do not add notes about the study, but we recommend that you do so for your own studies. The accumulated notes will always be loaded with the study, easing later analysis and re-analysis. Click on the *Browse* button in the first blank location and select a dataset name. The interface window should then look like the following:

Warning: Under Matlab 7.x and Linux, do not copy and paste information into the edit boxes. Though it *appears* that this works, Matlab 7 does not correctly register such inputs. Enter all information manually (except the 'browsed' dataset names). This problem does not seem to arise under Windows. -> The fields *Subject* and *Condition* (below) have been filled automatically. This is because the datasets already contained this information. For instance, if you were to load this dataset into EEGLAB by selecting menu item Edit > Dataset info, you would be able to edit the *Subject* and *Condition* information for this dataset (as shown below). You may also edit it within the study itself, since dataset information and study dataset information may be different to ensure maximum flexibility. For instance, if you want one dataset to belong to several studies, but play different roles in each.

Note: Use the checkbox *Update dataset information...* to maintain consistency between dataset and studyset fields. However, if you check this option, datasets may be modified on disk by clustering functions.

Enter all datasets for all subjects in the STUDY, so that the STUDY creation gui looks like this:

After you have finished adding datasets to the study, press *OK* in the pop_study.m gui to import all the dataset information. We strongly recommend that you also save the STUDY as a studyset by filling in the bottom edit box in the gui, or by selecting the EEGLAB menu item File > Save study as after closing the pop_study.m window.

Important note: Continuous data collected in one task or experiment session are often separated into epochs defining different task conditions (for example, separate sets of epochs time locked to targets and non-targets respectively). Datasets from different conditions collected in the same *session* are assumed by the clustering functions to have the same ICA component weights (i.e., the same ICA decomposition is assumed to have been applied to the data from all session conditions at once). If this was not the case, then datasets from the different conditions must be assigned to different *sessions*.

Loading an existing studyset

KEY STEP 1: Loading Studysets

Either use the studyset created in the previous section or load another studyset. To load a studyset, select menu item File > Load existing study. Select the file *N400.study* in the folder *5subjects*. After loading or creating a study, the main EEGLAB interface should look like this:

An EEGLAB STUDY (or study) contains descriptions of and links to data contained in one to many epoched datasets, for example a set of datasets from a group of subjects in one or more conditions of the same task, or performing different tasks in the same or different sessions. Other designs are possible, for instance a number of different task conditions from the same subject collected during one or more sessions. The term *STUDY* indicates that these datasets should originate from a single experimental STUDY and have comparable structure and significance. When in use, studyset datasets are partially or totally loaded into EEGLAB. They thus may also be accessed and modified individually, when desired, through the main EEGLAB graphic interface or using EEGLAB command line functions or custom dataset processing scripts.

In the EEGLAB gui (above), *Epoch consistency* indicates whether or not the data epochs in all the datasets have the same lengths and limits. *Channels per frame* indicates the number of channels in each of the datasets (*It is possible to process datasets with different numbers of channels*). *Channel location* indicates whether or not channel locations are present for all datasets. Note that channel locations may be edited for all datasets at the same time (simply call menu item Edit > Channel locations). The *Clusters* entry indicates the number of component clusters associated with this STUDY. There is always at least one cluster associated with a STUDY. This contains all the pre-selected ICA components from all datasets. The *Status* entry indicates the current status of the STUDY. In the case above, this line indicates that the STUDY is ready for pre-clustering. Below, we detail what the *STUDY* terms "subjects", "conditions", "sessions", and "groups" mean.

To list the datasets in the STUDY, use menu item Study > Edit study info. The following window will pop up:

The top of the window contains information about the STUDY, namely its running name, the extended task name for the STUDY, and some notes. The next section contains information about the 10 datasets that are part of the STUDY. For each dataset, we have specified a subject code and condition name. We chose to leave the session and group labels empty, as they were irrelevant for this STUDY. (For this STUDY, there was only one subject group and data for both experimental conditions were collected in a single session, so the same ICA decomposition was used for both conditions). Uniform default values will be used by EEGLAB for those fields. The *Components* column contains the components for each dataset that will be clustered. Note that if you change the component selection (by pressing the relevant push button), all datasets with the same subject name and the same session number will also be updated (as these datasets are assumed to have the same ICA components).

Each of the datasets EEG structures may also contain subject, group, session, and condition fields. They do not necessarily have to be the same as those present in the STUDY. For example, the same dataset may represent one condition in one STUDY and a different condition in another STUDY.

- In general, we prefer the dataset information to be consistent with the studyset information -- thus we check the first checkbox above.
- The second checkbox removes all current cluster information. When cluster information is present, it is not possible to add or remove datasets and to edit certain fields (because this would not be consistent with the already computed clusters). Re-clustering the altered STUDY does not take much time, once the pre-clustering information for the new datasets (if any) has been computed and stored.
- The third checkbox allows the STUDY to be saved (or re-saved) as a studyset (for example, *MyName.std*).

Both raw data and ICA component data may be processed in STUDY. Below we focus on processing component data. If you with to skip this part and only process raw data, jump to section on STUDY data visualization tools.

KEY STEP 2: Pre-selecting components for clustering.

Simply press the *Select by r.v.* (r.v. = residual variance) push button in the gui above. The entry box below will appear. This allows you to set a threshold for residual variance of the dipole model associated with each component.
Note: Using this option requires that dipole model information is present in each dataset. Use EEGLAB plug-in dipfit.m options and save the resulting dipole models into each dataset *before* calling the study guis. Otherwise, options related to dipole localization will not be available.

This interface allows specifying that components used in clustering will only be those whose equivalent dipole models have residual dipole variance of their component map, compared to the best-fitting equivalent dipole model projection to the scalp electrodes, less than a specified threshold (0% to 100%). The default r.v. value is 15%, meaning that only components with dipole model residual variance of less than 15% will be included in clusters. This is useful because of the modeled association between components with near 'dipolar' (or sometimes dual-dipolar) scalp maps with physiologically plausible components, those that may represent the activation in one (or two coupled) brain area(s). For instance, in the interface above, the default residual variance threshold is set to 15%. This means that only component that have an equivalent dipole model with less than 15% residual variance will be selected for clustering. Pressing *OK* will cause the component column to be updated in the main study-editing window. Then press *OK* to save your changes.

Editing STUDY datasets

Selecting an individual dataset from the Datasets menu allows editing individual datasets in a *STUDY*. Note, however, that creating new datasets or removing datasets will also remove the *STUDY* from memory since the study must remain consistent with datasets loaded in memory (here, however, EEGLAB will prompt you to save the study before it is deleted).

EEGLAB (v5.0b) also allows limited parallel processing of datasets of a *STUDY* in addition to computing clustering measures during pre-clustering. You may, for instance, filter all datasets of a *STUDY*. To do so, simply select menu item Tools > Filter data. You may also perform ICA decomposition of all datasets by running ICA individually on each of the datasets.

You may also select specific datasets using the Datasets > Select multiple datasets menu item and run ICA on all the datasets concatenated together (the ICA graphic interface contains a self-explanatory checkbox to perform that operation). This is useful, for example, when you have two datasets for two conditions from a subject that were collected in the same session, and want to perform ICA decomposition on their combined data. Using this option, you do not have to concatenate the datasets yourself; EEGLAB will do it for you.

Chapter 03: Working with STUDY designs

The STUDY.design structure is a concept introduced in EEGLAB v9. It allows performing statistical comparisons of multiple trial and dataset subsets of a STUDY without creating and storing the STUDY more than once. All the statistical designs are contained in the STUDY structure as (STUDY.design) sub-structures. For instance, a STUDY might contain datasets for 5 conditions comprising a 2x2 set of conditions for each subject plus a 'novel' condition. Data from all five conditions might be used to find clusters of similar independent component sources across subjects. However, statistical comparisons might be targeted to look at the main effects and interactions among the 2x2 conditions only, at differences between the 'novel' condition and each of the other four conditions, etc. This can be handled, beginning in EEGLAB v9, by defining a single STUDY structure with multiple STUDY.design sub-structures.

The main advantages of using STUDY designs are detailed below:

- No need to have one dataset per condition. An important restriction of working with STUDY structures in EEGLAB 8.0 (prior to STUDY.design) was that to compare multiple conditions users had to generate one dataset per condition. However, to analyze the influence of various contextual information about trials, it may be relatively impractical to generate many datasets with specific sets of trials. The new 'design' scheme in EEGLAB 9.0 is backward compatible with EEGLAB 8 but allows for more flexibility, in particular allowing the STUDY functions to dynamically extract specified data trials from datasets. Each dataset may contain several conditions or several datasets can be merged, using the STUDY design facility, to form to one condition.
- No restriction to studying "groups" and "conditions" only. Any user-defined independent variable may now be used to contrast different subsets of trials or datasets.
- The ability to create and analyze different STUDY designs within the same STUDY, each selecting only a subset of independent variable values.

Example: For instance in an oddball paradigm comprising trials time locked to oddball, distractor, and standard stimuli, a user might want to contrast oddball and distractor responses, considered together, with responses to standard stimuli. One might also want to look for differences between responses to oddball and distractor stimuli. If this STUDY has two subject groups, the user might want to look at the effect of group on any of the response types, just focus an analysis on one group, or look for Group by Condition interactions. One might also want to temporarily exclude a subject from a data analysis. All of these design concepts can be processed within a single **STUDY** using multiple **STUDY.design** structures. Using multiple STUDY.design structures may also be useful for testing different signal processing options. For instance, one might create two identical STUDY designs, in one computing the time/frequency measures using FFTs in one and using wavelets in the other one. Once computed, you will be able to toggle between design results so as to compare them.

Note that the tutorial STUDY for EEGLAB v9 is the same one that has been available in the tutorial since 2006. A default design, implementing EEGLAB processing of the whole STUDY (as per EEGLAB Version 8) is automatically been created when the tutorial STUDY is first loaded. In addition, all the precomputed measure files from v8 are preserved and may be used. If you are importing an EEGLAB STUDY from a version prior to v9 and not using the STUDY design menu, EEGLAB should behave exactly as it did prior to version 9. To edit the STUDY design, simply select the second STUDY menu as shown below. Note that if you change the default design (design(1)), your precomputed measure data files (STUDY ERP, spectrum, ERSP and ITC) may be lost.

This will pop up the following interface

The three push buttons at the top may be used to add a new design ("Add design"), rename a given design ("Rename design"), or delete a given design ("Delete design"). Note that the default design (design(1)) cannot be deleted. Adding a design copies the current design and creates a new design names "Design x" (x being the index of the new design).

The "Independent variable 1" list helps define independent variables. Currently, up to two independent variables may be defined (the two left columns). The list of independent variables is automatically generated based on the STUDY definition information and also based on events from each of the datasets. For details on what information from dataset is being extracted, refer to the STUDY design structure tutorial. Once an independent variable is selected, it is possible to select only a subset of its values. All the datasets or trials not selected will simply not be included in the STUDY.design processing. For instance, in this specific example, the independent variable 'condition' may take the values 'non-synonyms' and 'synonyms'. These values may be combined by pressing the "Combine selected value" push button. In this case, since there are only two values of the independent variable "condition", this is irrelevant. The detailed example at the end of this section shows an example of combining two values. Each independent variable also has a pairing status ("paired statistics" for paired data and "unpaired statistics" for unpaired data).

The "Subject" list contains the subjects to include in a specific design. Some subjects may be excluded or included. Note that it is better to select all subjects before pre-computing all STUDY measures, and then to exclude some subjects if necessary. When two groups of subjects are included (patients versus controls, for instance), some STUDY.design instances may include only one category of subjects.

The "Select only specific dataset/trials" push button to define a list of independent variables and values to include in the design. Clicking on the push button, "Select only specific dataset/trials," pops up the following interface.

 Chapter 03: Working with STUDY designs

In this interface, you may select specific independent variables with specific values to include in the STUDY. This option is only relevant for complex STUDY designs in which some sets of trials and/or datasets are excluded. Pressing the "Add" button will add the selection to the edit box on the right of the "Select only specific dataset/trials" push button in the STUDY design interface.

Below we show a more complex STUDY design scheme including 6 designs. In this experiment, there were two groups of subjects, "control" and "nondual" (a type of meditation practice). There were also four types of stimuli ("blank", "audio", "light" and "both" (audio and light)), two sessions for each subject, and two presentation modes ("evoked" and "spontanous"). Here, the stimulus type "audio - light" is not a real stimulus type; it was obtained by selecting stimuli "audio" and "light" and pressing the "Combine selected values" push button - see design number 3 below for more details. These screen captures are courtesy of the Institute of Noetic Science.

In the first design, we contrast the two groups of subjects for the "audio" and "visual" stimuli. There are two independent variables for this design (a 3x2 design).

 Chapter 03: Working with STUDY designs

In the second design, we only consider the "nondual" group subjects and three types of stimuli ("audio", "light" and "blank"). There is only one independent variable in this design (1x3 design).

 Chapter 03: Working with STUDY designs

In the third design, we only consider "nondual" subjects and compare two types of stimuli ("blank" versus "audio - light"). The "audio - light" condition groups together both "audio" and "light" trials - i.e. the stimulus may have been of either type "audio" or "light". This design helps contrast brain activity following stimulation compared to when no stimulus was presented ("blank" condition). This is a simple 1x2 design.

 Chapter 03: Working with STUDY designs

In the fourth design, we were interested in analysing audio stimuli that had been preceded by various types of stimuli. The independent variable "prevevent" had to be defined using a script. A field was added to the event structure using a custom script - it contained the previous stimulus type for each time-locking event. Note that the edit box on the right of the "Select only specific datasets/trials" GUI contains the string "'StimulusType', { 'audio' }". It indicates to EEGLAB that it should only consider this type of stimulus in this design and ignore all trials of type "light", "blank" and "both". This is a 1x3 design.

 Chapter 03: Working with STUDY designs

In the fifth design, we compared brain responses to "light" and "audio" stimuli in sessions 1 and 2. This is a 2x2 design.

 Chapter 03: Working with STUDY designs

Finally, in the sixth design, we compared brain responses to "light" and "audio" stimuli in "evoked" and "spontaneous" conditions. This is again a 2x2 design.

 Chapter 03: Working with STUDY designs

This simple example shows that the range of possibilities for STUDY designs is large. More details about STUDY.design structure is available in the STUDY structure part of the tutorial.

 Chapter 03: Working with STUDY designs

Chapter 04: STUDY Data Visualization Tools

EEGLAB (v6-) allows visualization of data properties (ERP, power spectrum, ERSP and ITC). Originally in 2006 this interface was only available for component clusters but even when working on ICA clusters, it is important go back to the data to see if a given effect visible at the cluster level can also be observed in the (relatively) unprocessed data.

Description of experiment for this part of the tutorial: These data were acquired by Arnaud Delorme and colleagues from fourteen subjects. Subjects were presented with pictures that either contained or did not contain animal image. Subjects respond with a button press whenever the picture presented contained an animal. These data are available for download here (380 Mb). A complete description of the task, the raw data (4Gb), and some Matlab files to process it, are all available here.

We have used these data in the following two sections since the released cluster tutorial data used in previous sections are too sparse to allow computing statistical significance. However, for initial training, you might better use that much smaller example STUDY.

Contents

-->

- 1 Precomputing channel measures
- 2 Plotting channel measures

Precomputing channel measures

Before plotting the component distance measures, you must precompute them using the Study > Precompute measures menu item as shown below.

It is highly recommended that for visualizing and computing statistics on data channels you first interpolate missing channels. Automated interpolation in EEGLAB is based on channel names. If datasets have different channel locations (for instance if the locations of the channels were scanned), you must interpolate missing channels for each dataset from the command line using eeg_interp.m. Select all the measures above, or just one if you want to experiment. The channel ERPs have been included in the tutorial dataset; if you select ERPs, they will not be recomputed -- unless you also check the box *Recompute even if present on disk*.

Each measure to precompute is explained in detail in the component clustering part (where the same exact measures may be computed).

Plotting channel measures

After precomputing the channel measures, you may now plot them, using menu item Study > Plot channel measures.

Here we only illustrate the behavior of pop_chanplot.m for plotting ERPs. Spectral and time/frequency (ERSP/ITC) measure data for scalp channels may be plotted in a similar manner, as shown in the previous section on component clustering. To plot data for a single scalp channel ERP, press the *Plot ERPs* pushbutton on the left column. A plot like the one below should appear:

You may plot all subjects ERPs by pressing the *Plot ERPs* pushbutton in the left columns, obtaining a figure similar to the one below.

Finally, you may also plot all scalp channels simultaneously. To do this, simply click the push button *Sel. all* to select all data channels. Then again press the *Plot ERPs* button in the right column.

Clicking on individual ERPs will make a window plotting the selected channel ERP pop up. Many other plotting options are available in the central column of the pop_chanplot.m gui. These will be described in the next section.

More complex plotting options are demonstrated in the section dealing with visualization and statistics.

Chapter 05: Component Clustering Tools

Clustering outline

There are several steps in the independent component clustering process, some of which we have already addressed in the previous section:

1. Identify a set of epoched EEG datasets containing ICA weights to form the STUDY to be clustered.
2. Specify the subject code and group, task condition, and session for each dataset.
3. Identify the components in each dataset to cluster.
4. Specify and compute ("pre-clustering") measures to use in clustering.
5. Perform component clustering using these measures.
6. View the scalp maps, dipole models, and activity measures of the component clusters.
7. Perform signal processing and statistical estimation on the clusters.

This part of the tutorial will demonstrate how to use EEGLAB to interactively preprocess, cluster, and then visualize the dynamics of ICA (or other linear) signal components across one or many subjects by operating on the tutorial study.

Note: With only a few subjects and a few clusters (a necessary limitation, to be able to easily distribute the example), it may not be possible to find six consistent component clusters with uniform and easily identifiable natures. We have obtained much more satisfactory results clustering data from 15 to 30 or more subjects.

After following this tutorial using the sample data, we recommend you create a study for a larger group of datasets, if available, whose properties you know well. Then try clustering components this study in several ways. Carefully study the consistency and properties of the generated component clusters to determine which method of clustering produces clusters adequate for your research purposes.

Note: We recommend performing one ICA decomposition on all the data collected in each data collection session, even when task several conditions are involved. In our experience, ICA can return a more stable decomposition when trained on more data. Having components with common spatial maps also makes it easier to compare component behaviors across conditions. To use the same ICA decomposition for several conditions, simply run ICA on the continuous or epoched data *before* extracting separate datasets corresponding to specific task conditions of interest. Then extract specific condition datasets; they will automatically inherit the same ICA decomposition.

Why cluster?

Is my Cz your Cz? To compare electrophysiological results across subjects, the usual practice of most researchers has been to identify scalp channels (for instance, considering recorded channel "Cz" in every subject's data to be spatially equivalent). Actually, this is an idealization, since the spatial relationship of any physical electrode site (for instance, Cz, the vertex in the International 10-20 System electrode labeling convention) to the underlying cortical areas that generate the activities summed by the (Cz) channel may be rather different in different subjects, depending on the physical locations, extents, and particularly the orientations of the cortical source areas, both in relation to the 'active' electrode site (e.g., Cz) and/or to its recorded reference channel (for example, the nose, right mastoid, or other site).

That is, data recorded from equivalent channel locations (Cz) in different subjects may sum activity of different mixtures of underlying cortical EEG sources, no matter how accurately the equivalent electrode locations were measured on the scalp. This fact is commonly ignored in EEG research.

Is my IC your IC? Following ICA (or other linear) decomposition, however, there is no natural and easy way to identify a component from one subject with one (or more) component(s) from another subject. A pair of independent components (ICs) from two subjects might resemble and/or differ from each other in many ways and to different degrees -- by differences in their scalp maps, power spectra, ERPs, ERSPs, ITCs, or etc. Thus, there are many possible (distance) measures of similarity, and many different ways of combining activity measures into a global distance measure to estimate component pair similarity.

Thus, the problem of identifying equivalent components across subjects is non-trivial. An attempt at doing this for 31-channel data was published in 2002 and 2004 in papers whose preparation required elaborate custom scripting (by Westerfield, Makeig, and Delorme). A 2005 paper by Onton et al. reported on dynamics of a frontal midline component cluster identified in 71-channel data. EEGLAB now contains functions and supporting structures for flexibly and efficiently performing and evaluating component clustering across subjects and conditions. With its supporting data structures and stand-alone '**std_**' prefix analysis functions, EEGLAB makes it possible to summarize results of ICA-based analysis across more than one condition from a large number of subjects. This should make more routine use of linear decomposition and ICA possible to apply to a wide variety of hypothesis testing on datasets from several to many subjects.

The number of EEGLAB clustering and cluster-based functions will doubtless continue to grow in number and power in the future versions, since they allow the realistic use of ICA decomposition in hypothesis-driven research on large or small subject populations.

NOTE: Independent component clustering (like much other data clustering) has no single 'correct' solution. Interpreting results of component clustering, therefore, warrants caution. Claims to discovery of physiological facts from component clustering should be accompanied by thoughtful caveat and, preferably, by results of statistical testing against suitable null hypotheses.

Clustering Methods

There are two types of clustering methods available. One is the original (PCA) method and the other one is the new Measure Product (MP) method. Both methods produce reasonable clusters, but the new MP method requires less parameter tuning (it has only one parameter) and is therefore recommended. The PCA method is older and its implementation is more stable though. The MP method has been removed from version 9.0.3.3b for lack of stability but you may contact the author at nima@sccn.ucsd.edu to obtain a copy.

Preparing to cluster (Pre-clustering) with PCA (original) method

The next step before clustering is to prepare the *STUDY* for clustering. This requires, first, identifying the components from each dataset to be entered into the clustering (as explained briefly above), then computing component activity measures for each study dataset (described below). For this purpose, for each dataset component the pre-clustering function pop_preclust.m first computes desired condition-mean measures used to determine the cluster 'distance' of components from each other. The condition means used to construct this overall cluster 'distance' measure may be selected from a palette of standard EEGLAB measures: ERP, power spectrum, ERSP, and/or ITC, as well as the component scalp maps (interpolated to a standard scalp grid) and their equivalent dipole model locations (if any).

Note: Dipole locations are the one type of pre-clustering information *not* computed by pop_preclust.m. As explained previously, to use dipole locations in clustering and/or in reviewing cluster results, dipole model information must be computed separately and saved in each dataset using the dipfit.m EEGLAB plug-in.

The aim of the pre-clustering interface is to build a global distance matrix specifying 'distances' between components for use by the clustering algorithm. This component 'distance' is typically abstract, estimating how 'far' the components' maps, dipole models, and/or activity measures are from one another in the space of the joint, PCA-reduced measures selected. This will become clearer as we detail the use of the graphic interface below.

KEY STEP 3: Computing component measures.

Invoke the pre-clustering graphic interface by using menu item Study > Build pre-clustering array.

The top section of the pop_preclust.m gui above allows selecting clusters from which to produce a refined clustering. There is not yet any choice here -- we must select the parent datasets that contain all selected components from all datasets (e.g., the components selected at the end of the previous section).

The checkboxes on the left in the second section of the pop_preclust.m interface above allow selection of the component activity measures to include in the *cluster location measure* constructed to perform clustering. The goal of the pre-clustering function is to compute an N-dimensional cluster position vector for each component. These 'cluster position' vectors will be used to measure the 'distance' of components from each other in the N-dimensional cluster space. The value of N is arbitrary but, for numeric reasons pertaining to the clustering algorithms, should be kept relatively low (e.g., <10). In the cluster position vectors, for example, the three first values might represent the 3-D (x,y,z) spatial locations of the equivalent dipole for each component. The next, say, 10 values might represent the largest 10 principal components of the first condition ERP, the next 10, for the second condition ERP, and so on.

If you are computing (time/frequency) spectral perturbation images, you cannot use all their (near-3000) time-frequency values, which are redundant, in any case. Here also, you should use the *Dim.* column inputs to reduce the number of dimensions (for instance, to 10).

Note: pop_preclust.m reduces the dimension of the cluster position measures (incorporating information from ERP, ERSP, or other measures) by restricting the cluster position vectors to an N-dimensional principal subspace by principal component analysis (PCA).

You may wish to "normalize" these principal dimensions for the location and activity measures you select so their metrics are equivariant across measures. Do this by checking the checkbox under the *norm* column. This 'normalization' process involves dividing the measure data of all principal components by the standard deviation of the first PCA component for this measure. You may also specify a relative weight (versus other measures). For instance if you use two measures (A and B) and you want A to have twice the "weight" of B, you would normalize both measures and enter a weight of 2 for A and 1 for B. If you estimate that measure A has more relevant information than measure B, you could also enter a greater number of PCA dimension for A than for B. Below, for illustration we elect to cluster on all six allowed activity measures.

TIP: All the measures described below, once computed, can be used for clustering and/or for cluster visualization (see the following section of the tutorial, 'Visualize Component Cluster Information'). If you do not wish to use some of the measures in clustering but still want to be able to visualize it, select it and enter 0 for the PCA dimension. This measure will then be available for cluster visualization although it will not have been used in the clustering process itself. This allows an easy way of performing exploratory clustering on different measure subsets.

- Spectra: The first checkbox in the middle right of the pre-clustering window (above) allows you to include the log mean power spectrum for each condition in the pre-clustering measures. Clicking on the checkbox allow you to enter power spectral parameters. In this case, a frequency range [lo hi] (in Hz) is required. Note that for clustering purposes (but not for display), for each subject individually, the mean spectral value (averaged across all selected frequencies) is subtracted from all selected components, and the mean spectral value at each frequency (averaged across all selected components) is subtracted from all components. The reason is that some subjects have larger EEG power than others. If we did not subtract the (log) means, clusters might contain components from only one subject, or from one type of subject (e.g., women, who often have thinner skulls and therefore larger EEG than men).
- ERPs: The second checkbox computes mean ERPs for each condition. Here, an ERP latency window [lo hi] (in ms) is required.
- Dipole locations: The third checkbox allows you to include component equivalent dipole locations in the pre-clustering process. Dipole locations (shown as [x y z]) automatically have three dimensions (Note: It is not yet possible to cluster on dipole orientations). As mentioned above, the equivalent dipole model for each component and dataset must already have been pre-computed. If one component is modeled using two symmetrical dipoles, pop_preclust.m will use the average location of the two dipoles for clustering purposes (Note: this choice is not optimum).
- Scalp maps: The fourth checkbox allows you to include scalp map information in the component 'cluster location'. You may choose to use raw component map values, their laplacians, or their spatial gradients. (Note: We have obtained fair results for main components using laplacian scalp maps, though there are still better reasons to use equivalent dipole locations instead of scalp maps. You may also select whether or not to use only the absolute map values, their advantage being that they do not depend on (arbitrary) component map polarity. As explained in the ICA_decomposition.m, ICA component scalp maps themselves have no absolute scalp map polarity.
- ERSPs and/or ITCs: The last two checkboxes allow including event-related spectral perturbation information in the form of event-related spectral power changes (ERSPs), and event-related phase consistencies (ITCs) for each condition. To compute the ERSP and/or ITC measures, several time/frequency parameters are required. To choose these values, you may enter the relevant timefreq.m keywords and arguments in the text box. You may for instance enter *'alpha', 0.01* for significance masking. See the timefreq.m help message for information about time/frequency parameters to select.
- Final number of dimensions: An additional checkbox at the bottom allows further reduction of the number of dimensions in the component distance measure used for clustering. Clustering algorithms may not work well with measures having more than 10 to 20 dimensions. For example, if you selected all the options above and retained all their dimensions, the accumulated distance

measures would have a total of 53 dimensions. This number may be reduced (e.g., to a default 10) using the PCA decomposition invoked by this option. Note that, since this will distort the cluster location space (projecting it down to its smaller dimensional 'shadow'), it is preferable to use this option carefully. For instance, if you decide to use reduced-dimension scalp maps and dipole locations that together have 13 dimensions (13 = the requested 10 dimensions for the scalp maps plus 3 for the dipole locations), you might experiment with using fewer dimensions for the scalp maps (e.g., 7 instead of 10), in place of the final dimension reduction option (13 to 10).

Finally, the pop_preclust.m gui allows you to choose to save the updated *STUDY* to disk.

In the pop_preclust.m select all methods and leave all default parameters (including the dipole residual variance filter at the top of the window), then press *OK*. As explained below, for this tutorial STUDY, measure values are already stored on disk with each dataset, so they need not be recomputed, even if the requested clustering limits (time, frequency, etc.) for these measured are reduced.

Re-using component measures computed during pre-clustering: Computing the spectral, ERP, ERSP, and ITC measures for clustering may, in particular, be time consuming -- requiring up to a few days if there are many components, conditions, and datasets! The time required will naturally depend on the number and size of the datasets and on the speed of the processor. Future EEGLAB releases will implement parallel computing of these measures for cases in which multiple processors are available. Measures previously computed for a given dataset and stored by std_preclust.m will not be recomputed, even if you narrow the time and/or frequency ranges considered. Instead, the computed measure information will be loaded from the respective Matlab files in which it was saved by previous calls to pop_preclust.m.

Measure data files are saved in the same directory/folder as the dataset, and have the same dataset name -- but different filename extensions. For example, component ERSP information for the dataset *syn02-S253-clean.set* is stored in a file named *syn02-S253-clean.icaersp*. As mentioned above, for convenience it is recommended that each subject's data be stored in a different directory/folder. If all the possible clustering measures have been computed for this dataset, the following Matlab files should be in the */S02/* dataset directory:

- *syn02-S253-clean.icaerp* (ERPs)
- *syn02-S253-clean.icaspec* (power spectra)
- *syn02-S253-clean.icatopo* (scalp maps)
- *syn02-S253-clean.icaersp* (ERSPs)
- *syn02-S253-clean.icaitc* (ITCs)

The parameters used to compute each measure are also stored in the file, for example the frequency range of the component spectra. Measure files are standard Matlab files that may be read and processed using standard Matlab commands. The variable names they contain should be self-explanatory.

Note: For ERSP-based clustering, if a parameter setting you have selected is different than the value of the same parameter used to compute and store the same measure previously, a query window will pop up asking you to choose between recomputing the same values using the new parameters or keeping the existing measure values. Again, narrowing the ERSP latency and frequency ranges considered in clustering will not lead to recomputing the ERSP across all datasets.

Finding clusters with PCA (original) method

KEY STEP 4: Computing and visualizing clusters.

Calling the cluster function pop_clust.m, then selecting menu item Study > Cluster components will open the following window.

Currently, two clustering algorithms are available: 'kmeans' and 'neural network' clustering. As explained earlier, 'kmeans' requires the Matlab Statistics Toolbox, while 'neural network' clustering uses a function from the Matlab Neural Network Toolbox.

Both algorithms require entering a desired number of clusters (first edit box). An option for the *kmeans()* algorithm can relegate 'outlier' components to a separate cluster. Outlier components are defined as components further than a specified number of standard deviations (3, by default) from any of the cluster centroids. To turn on this option, click the upper checkbox on the left. Identified outlier components will be put into a designated *Outliers* cluster (Cluster 2). Click on the lower left checkbox to save the clustered studyset to disk. If you do not provide a new filename in the adjacent text box, the existing studyset will be overwritten.

In the pop_clust.m gui, enter *10* for the number of clusters and check the *Separate outliers ...* checkbox to detect and separate outliers. Then press *OK* to compute clusters (clustering is usually quick). The cluster editing interface detailed in the next section will automatically pop up. Alternatively, for the sample data, load the provided studyset *N400clustedit.study* in which pre-clustering information has already been stored.

Preparing to cluster (Pre-clustering) with Affinity Product Method

In Affinity Product clustering, IC measures, except equiv. dipoles, (ERP, ERSP...) are compared for each IC pair and their dissimilarity is multiplied together to form a combined pairwise dissimilarity matrix. This matrix is then normalized, weighted and added to the normalized and weighted IC equiv. dipole distance matrix. The final dissimilarity matrix is then clustered using affinity clustering method (Fig. below). As you can see this method does not perform any dimensionality reduction on EEG measures (i.e PCA) a it only calculates pairwise (dis)similarities.These similarity matrices (correlations) has to be calculated in the pre-clustering step.

-->-->

KEY STEP 3: Computing pairwise measures for AP clustering.

Invoke the pre-clustering graphic interface by using menu item Study > Affinity Product clustering -> Build pre-clustering array.

In the GUI you can select EEG measures for which pre-clustering matrices should be calculated. Use 'Re-Calculate All' option to remove all these matrices before calculating new ones. This might be useful if you have changed the subset of STUDY components to be clustered.

Finding clusters with Affinity Product Method

After the pre-clustering step (above) is finished, you can cluster STUDY components based on any combination of measures included in pre-clustering. The final clustering is performed on the combined pairwise distance matrix using Affinity Propagation algorithm.

KEY STEP 4: Computing clusters.

> Invoke the MP clustering graphic interface by using menu item Study > Affinity Product clustering > Cluster Components will open the following window.

Here you can specify the number of clusters and control the effect of equiv. dipole distances in the clustering by setting the 'Relative dipole weight' parameter. For example, by setting this value to 0.8, the final dissimilarity matrix will consist of 80% distance dissimilarity and 20% of other measures combined together.

Please note that the number of returned clusters may slightly (up to 5%) differ from the number requested in the GUI. Also, currently only clustering the parent cluster (containing all components) is supported.

An option for the Affinity Clustering algorithm can relegate 'outlier' components to a separate cluster. Outlier components are defined as components further than a specified number of standard deviations (3, by default) from any of the cluster centroids. To turn on this option, click the lower checkbox on the left. Identified outlier components will be put into a designated Outliers cluster (Cluster 2).

Viewing component clusters

Calling the cluster editing function pop_clustedit.m using menu item Study > Edit > plot clusters will open the following window. Note: The previous menu will also call automatically this window after clustering has finished.

Of the 305 components in the sample *N400STUDY* studyset, dipole model residual variances for 154 components were above 15%. These components were omitted from clustering. The remaining 151 components were clustered on the basis of their dipole locations, power spectra, ERPs, and ERSP measures into 10 component clusters.

Visualizing clusters: Selecting one of the clusters from the list shown in the upper left box displays a list of the cluster components in the text box on the upper right. Here, *SO2 IC33* means "independent component 33 for subject SO2," etc. The *All 10 cluster centroids* option in the (left) text box list will cause the function to display results for all but the *ParentCluster* and *Outlier* clusters. Selecting one of the plotting options below (left) will then show all 10 cluster centroids in a single figure. For example, pressing the *Plot scalp maps* option will produce the figure below:

In computing the mean cluster scalp maps (or scalp map centroids), the polarity of each of the cluster's component maps was first adjusted so as to correlate positively with the cluster mean (recall that component maps have no absolute polarity). Then the map variances were equated. Finally, the normalized means were computed.

To see individual component scalp maps for components in the cluster, select the cluster of interest in the left column (for example, *Cluster 8* as in the figure above Then press the *Plot scalp maps* option in the left column. The following figure will appear. (Note: Your *Cluster 8* scalp map may differ after you have recomputed the clusters for the sample STUDY).

To see the relationship between one of the cluster centroid maps and the maps of individual components in the cluster, select the cluster of interest (for instance Cluster 8), and press the *Plot scalp maps* option in the *right* pop_clustedit.m column.

Note: Channels missing from any of the datasets do not affect clustering or visualization of cluster scalp maps. Component scalp maps are interpolated by the toporeplot.m function, avoiding the need to restrict *STUDY* datasets to a common 'always clean' channel subset or to perform 'missing channel' interpolation on individual datasets.

You may also plot scalp maps for individual components in the cluster by selecting components in the right column and then pressing *Plot scalp maps* (not shown).

A good way to visualize all the average cluster measures at once is to first select a cluster of interest from the cluster list on the left (e.g., 'Cluster 8'), and then press the *Plot cluster properties* push button. The left figure below presents the Cluster-8 mean scalp map (same for both conditions), average ERP and spectrum (for these, the two conditions are plotted in different colors), average ERSP and ITC (grand means for both conditions; the individual conditions may be plotted using the *Plot cluster properties* push button). The 3-D plot on the bottom left presents the locations of the centroid dipole (red) and individual component equivalent dipoles (blue) for this cluster.

To quickly recognize the nature of component clusters by their activity features requires experience. Here Cluster 8 accounts for some right occipital alpha activity -- note the strong 10-Hz peak in the activity spectra. The cluster ERPs show a very slow (1-Hz) pattern peaking at the

appearance of *first* words of the word pairs (near time -1 s). The apparent latency shift in this slow wave activity between the two conditions may or may not be significant. A positive (though still quite low, 0.06) ITC follows the appearance of the first word in each word pair (see Experimental Design), indicating that quite weakly phase-consistent theta-band EEG activity follows first word onsets. Finally, blocking of spectral power from 7 Hz to at least 25 Hz appears just after onset of the *second* words of word pairs (at time 0) in the grand mean ERSP plot (blue region on top right)

To review all 'Cluster 8' component dipole locations, press the *Plot dipoles* button in the left column. This will open the plot viewer showing all the cluster component dipoles (in blue), plus the cluster mean dipole location (in red). You may scroll through the dipoles one by one, rotating the plot in 3-D or selecting among the three cardinal views (lower left buttons), etc. Information about them will be presented in the left center side bar (see the image below).

As for the scalp maps, the pop_clustedit.m gui will separately plot the cluster ERPs, spectra, ERSPs or ITCs. Let us review once more the different plotting options for the data spectrum. Pressing the *Plot spectra* button in the left column will open a figure showing the two mean condition spectra below.

Pressing the *Plot spectra* button in the right column with *All components* selected in the left column will open a figure displaying for each condition all cluster component spectra plus (in bold) their mean.

Finally, to plot the condition spectra for individual cluster components, select one component from the *Select component(s) to plot* list on the right and press *Plot spectra* in the right column. For example, selecting component 37 from subject S02 (*SO2 IC37*) will pop up the figure below. Here, the single component spectra are shown in light blue as well as the mean cluster spectrum (in black).

Editing clusters

The results of clustering (by either the 'k-means' or 'Neural network' methods) can also be updated manually in the preview cluster viewing and editing window (called from menu item Study > Edit/plot clusters). These editing options allow flexibility for adjusting the clustering. Components can be reassigned to different clusters, clusters can be merged, new clusters can be created, and 'outlier' components can be rejected from a cluster. Note that if you make changes via the pop_clustedit.m gui, then wish to cancel these changes, pressing the *Cancel* button will cause the *STUDY* changes to be forgotten.

Renaming a cluster: The *Rename selected cluster* option allows you to rename any cluster using a (mnemonic) name of your choice. Pressing this option button opens a pop-up window asking for the new name of the selected cluster. For instance, if you think a cluster contains components accounting for eye blinks you may rename it "Blinks".

Automatically rejecting outlier components: Another editing option is to reject 'outlier' components from a cluster *after* clustering. An 'outlier' can be defined as a component whose cluster location is more than a given number of standard deviations from the location of the cluster centroid. Note that standard deviation and cluster mean values are defined from the N-dimensional clustering space data created during the pre-clustering process.

For example, if the size of the pre-clustering cluster location matrix is 10 by 200 (for 200 components), then N = 10. The default threshold for outlier rejection is 3 standard deviations. To reject 'outlier' components from a cluster, first select the cluster of interest from the list on the left and then press the *Auto-reject outlier components* option button. A window will open, asking you to set the outlier threshold. 'Outlier' components selected via either the '*eject outlier components* option or the *Remove selected outlier component(s)* option (below) are moved from the current cluster '[Name]' to a cluster named 'Outlier [Name]'.

Removing selected outlier components manually: Another way to remove 'outlier' components from a cluster is to do so manually. This option allows you to de-select seeming 'outlier' components irrespective of their distance from the cluster mean. To manually reject components, first select the cluster of interest from the list on the left, then select the desired 'outlier' component(s) from the component list on the right, then press the *Remove selected outlier comps.* button. A confirmation window will pop up.

Creating a new cluster: To create a new empty cluster, press the *Create new cluster* option, this opens a pop-up window asking for a name for the new cluster. If no name is given, the default name is 'Cls #', where '#' is the next available cluster number. For changes to take place, press the *OK* button in the pop-up window. The new empty cluster will appear as one of the clusters in the list on the left of the editing/viewing cluster window.

Reassigning components to clusters: To move components between any two clusters, first select the origin cluster from the list on the left, then select the components of interest from the component list on the right, and press the *Reassign selected component(s)* option button. Select the target cluster from the list of available clusters.

Saving changes: As with other pop_ functions, you can save the updated *STUDY* set to disk, either overwriting the current version - by leaving the default file name in the text box - or by entering a new file name.

Hierarchic sub-clustering (PCA method only)

The clustering tools also allow you to perform hierarchic sub-clustering. For example, imagine clustering all the components from all the datasets in the current *STUDY* (i.e., the Parent Cluster) into two clusters. One cluster turn out to contain only artifactual non-EEG components (which you thus rename the 'Artifact' cluster) while the other contains non-artifactual EEG components (thus renamed 'Non-artifact').

NOTE: *This is only a quite schematic example for tutorial purposes: It may normally not be possible to separate all non-brain artifact components from cortical non-artifact components by clustering all components into only two clusters -- there are too many kinds of scalp maps and artifact activities associated with the various classes of artifacts!*

At this point, we might try further clustering only the 'Artifact' cluster components, e.g., attempting to separate eye movement processes from heart and muscle activity processes, or etc. A schematic of a possible (but probably again not realistic) further decomposition is shown schematically above.

In this case, the parent of the identified eye movement artifact cluster is the 'Artifact' cluster; the child cluster of 'eye' artifacts itself has no child clusters. On the other hand, clustering the 'Non-artifact' cluster produces three child clusters which, presumably after careful inspection, we decide to rename 'Alpha', 'Mu' and 'Other'.

To refine a clustering in this way, simply follow the same sequence of event as described above. Call the pre-clustering tools using menu item Study > Build preclustering array. You may now select a cluster to refine. In the example above, we notice that there seem to be components with different spectra in Cluster 8, so let us try to refine the clustering based on differences among the Cluster-8 component spectra.

Select the Study > Cluster components menu item. Leave all the defaults (e.g., 2 clusters) and press *OK*.

The visualization window will then pop up with two new clusters under Cluster 8.

Below the component spectra for Cluster 13 are plotted. Note that Cluster-13 components have relatively uniform spectra.

You may also plot the scalp map of Cluster 13. All the cluster components account for occipital activity and have similarly located equivalent dipole sources.

Figure 3
File Edit View Insert Tools Desktop Window Help
Cls 13 average scalp map, 3Ss
ic37/S02
ic14/S05
ic16/S05
ic14/S08
ic15/S08
ic21/S08
ic24/S08
ic31/S08

Chapter 06: Study Statistics and Visualization Options

Computing statistics is essential to observation of group, session, and/or condition measure differences. EEGLAB allows users to use either parametric or non-parametric statistics to compute and estimate the reliability of these differences across conditions and/or groups. The same methods for statistical comparison apply both to component clusters and to groups of data channels (see following). Here, we will briefly review, for each measure (ERP, power spectrum, ERPS, ITC), how to compute differences accross the two conditions in a STUDY. At the end, we will show examples of more complex analyses involving 3 groups of subjects and 2 conditions. We will also briefly describe the characteristics of the function that performs the statistical computations, and discuss how to retrieve the p-values for further processing or publication.

Contents

-->

- 1 Parametric and non-parametric statistics
- 2 Options for computing statistics on and plotting results for scalp channel ERPs
- 3 Computing statistics for studies with multiple groups and conditions

Parametric and non-parametric statistics

EEGLAB allows performing classical parametric tests (paired t-test, unpaired t-test, ANOVA) on ERPs, power spectra, ERSPs, and ITCs. *Below, we will use channel ERPs as an example, though in general we recommend that independent component ERPs and other measures be used instead. This is because no data features of interest are actually generated in the scalp, but rather in the brain itself, and in favorable circumstances independent component filtering allows isolation of the separate brain source activities, rather than their mixtures recorded at the scalp electrodes.*

For example, on 15 subjects' ERPs in two conditions, EEGLAB functions can perform a simple two-tailed paired t-test at each trial latency on the average ERPs from each subject. If there are different numbers of subjects in each condition, EEGLAB will use an unpaired t-test. If there are more than two STUDY conditions, EEGLAB will use ANOVA instead of a t-test. For spectra, the p-values are computed at every frequency; for ERSPs and ITCs, p-values are computed at every time/frequency point. See the sub-section on component cluster measures below to understand how to perform statistical testing on component measures.

EEGLAB functions can also compute non-parametric statistics. The null hypothesis is that there is no difference among the conditions. In this case, the average difference between the ERPs for two conditions should lie within the average difference between 'surrogate' grand mean condition ERPs, averages of ERPs from the two conditions whose condition assignments have been shuffled randomly. An example follows:

> Given 15 subjects and two conditions, let us use a1, a2, a3, ... a15 the scalp channel ERP values (in microvolts) at 300 ms for all 15 subjects in the first condition, and b1, b2, b4, ... b15 the ERP values for the second condition. The grand average ERP condition difference is
>
> d = mean((a1-b1) + (a2-b2) + ... + (a15-b15)).
>
> Now, if we repeatedly shuffle these values between the two condition (under the null hypothesis that there are no significant differences between them, and then average the shuffled values,
>
> d1 = mean((b1-a1) + (a2-b2) + ... + (b15-a15)).
> d2 = mean((a1-b1) + (b2-a2) + ... + (a15-b15)).
> d3 = mean((b1-a1) + (b2-a2) + ... + (a15-b15)).
> ...
>
> we then obtain a distribution of surrogate condition-mean ERP values dx constructed using the null hypothesis (see their smoothed histogram below). If we observe that the initial d value lies in the very tail of this surrogate value distribution, then the supposed null hypothesis (no difference between conditions) may be rejected as highly unlikely, and the observed condition difference may be said to be statistically valid or significant.

Note that the surrogate value distribution above can take any shape and does not need to be gaussian. In practice, we do not compute the mean condition ERP difference, but its t-value (the mean difference divided by the standard deviation of the difference and multiplied by the square root of the number of observations less one). The result is equivalent to using the mean difference. The advantage is that when we have more conditions, we can use the comparable ANOVA measure. Computing the probability density distribution of the t-test or ANOVA is only a "trick" to be able to obtain a difference measure across all subjects and conditions. It has nothing to do with relying on a parametric t-test or ANOVA model, which assume underlying gaussian value distributions. Note that only measure that are well-behaved (e.g. are not highly non-linear) should be used in this kind of non-parametric testing.

We suggest reading in a relevant statistics book for more details: An introduction to statistics written by one of us (AD) is available here. A good textbook on non-parametric statistics is the text book by Rand Wilcox, "Introduction to robust estimation and hypothesis testing", Elsevier, 2005.

Below we illustrate the use of these options on scalp channel ERPs, though they apply to all measures and are valid for both scalp channels and independent component (or other) source clusters.

Options for computing statistics on and plotting results for scalp channel ERPs

Call again menu item Study > Plot channel measures. In the middle of the two *Plot ERPs* buttons, click on the *Params* pushbutton. The following graphic interface pops up:

Check the *Plot conditions on the panel* checkbox to plot both conditions on the same figure. Click on the *Compute condition statistics* checkbox. Note that, since there are no groups in this study, the *Compute group statistics* is not available. Enter 0.01 for the threshold p value as shown above. Press *OK* then select channel "Fz" in the left columns and press the *Plot ERPs* pushbutton in the same column. The following plot appears. The black rectangles under the ERPs indicate regions of significance ($p < 0.01$).

To show the actual p-values, call back the ERP parameter interface and remove the entry in the *Threshold* edit box. You may also unclick *Plot conditions on the panel* to plot conditions on different panels.

Press *OK*. Now click again *Plot ERPs*; the following figure pops up.

All of these plots above reported parametric statistics. While parametric statistics might be adequate for explorating your data, it is better to use permutation-based statistics (see above) to plot final results. Call back the graphic interface and select *Permutation* as the type of statistics to use, as shown below.

Below, we will use non-parametric statistics for all data channels. Click on the *Sel. all* pushbutton in the channel selection interface, and then push the *Plot ERPs* button. The shaded areas behind the ERPs indicate the regions of significance.

A command line function, std_envtopo.m, can also visualize cluster contributions to the grand mean ERP in one or two conditions, and to their difference. For more details refer here.

Finally, for data channels (but not for component clusters) an additional option is available to plot ERP scalp maps at specific latencies. Using the graphic interface once again, enter "200" as the epoch latency (in ms) to plot.

Select all channels (or a subset of channels) and press the *Plot ERPs* button. The following figure appears.

There are options in the gui above we have not discussed -- they should be self-explantory. For instance, the 'Time range to plot' edit box allows plotting a shorter time range than the full epoch. The *Plot limits* edit box allows setting fixed lower and upper limits to the plotted potentials. Finally the *Display filter* edit box allows entering a frequency (for instance 20 Hz) below which to filter the ERP. This is only applied for the ERP display and does not affect computation of the statistics. This option is useful when plotting noisy ERPs for single subjects.

The graphic interfaces for both power spectral and ERSP/ITC measures are similar to that used for ERPs and need not be described in detail. You may refer to the relevant function help messages for more detail. Below, we describe plotting results for a more complex STUDY containing both subject groups and experimental conditions.

Computing statistics for studies with multiple groups and conditions

This functionality is under developments and although we believe it is working properly, we are planning to update the graphic interface to make it more understandable. The problem with the current interface is that it is showing the ANOVA interaction term along with marginal statistics for groups and conditions (in future release we are planning to have users toggle either marginal statistics or statistical main effect). Here we just show one plot obtained from a study on three groups of patient of 16 subjects each and for two experimental conditions (KAN, representing responses to the appearance of Kaniza triangles, and NONKAN, representing responses to the appearance of inverted Kaniza triangles; Courtesy of Rael Cahn). Selecting only condition statistics and plotting conditions on the same panel returns the figure below.

Figure 6 <2>
File Edit View Insert Tools Desktop Window Help
Group 1
Group 2
Group 3
KAN
NONKAN
8
6
4
2
0
-2
-4
-1000
0
1000

Chapter 07: EEGLAB Study Data Structures

The STUDY structure

This section gives details of EEGLAB structures necessary for writing custom Matlab scripts, functions, and plug-ins that operate on EEGLAB STUDY structures and studysets.

The *STUDY* structure contains information for each of its datasets, plus additional information to allow processing of all datasets sequentially. After clustering the independent components identified for clustering in each of the datasets, each of the identified components in each dataset is assigned to one component cluster (in addition to Cluster 1 that contains all components identified for clustering). The *STUDY* structure also contains the details of the component clusters.

Below is a prototypical *STUDY* structure. In this tutorial, the examples shown were collected from analysis of a small sample studyset comprising ten datasets, two conditions from each of five subjects. After loading a studyset (see previous sections, or as described below) using the function pop_loadstudy.m, typing *STUDY* on Matlab command line will produce results like this:

```
>> STUDY

STUDY =

        name: 'N400STUDY'
        filename: 'N400ClustEditHier.study'
        filepath: '/home/eeglab/data/N400/'
        datasetinfo: [1x10 struct]
        session: []
        subject: {1x5 cell}
        group: {'old' 'young'}
        condition: {'non-synonyms' 'synonyms'}
        setind: [2x5 double]
        cluster: [1x40 struct]
        notes: ' '
        task: 'Auditory task: Synonyms Vs. Non-synonyms, N400'
        history: [1x4154 char]
        etc: [1x1 struct]
```

-->-->

The field *STUDY.datasetinfo* is an array of structures whose length is the number of datasets in the *STUDY*. Each structure stores information about one of the datasets, including its subject, condition, session, and group labels. It also includes a pointer to the dataset file itself (*as explained below in more detail*).

STUDY.datasetinfo sub-fields *subject*, *session*, *group* and *condition* label the subject, subject group, session, and condition associated with each dataset in the study. This information must be provided by the user when the *STUDY* structure is created. Otherwise, default values are assumed.

The *STUDY.setind* field holds the indices of the *datasetinfo* cells, ordered in a 2-D matrix in the format [conditions by (subjects x sessions)].

The *STUDY.cluster* field is an array of cluster structures, initialized when the *STUDY* is created and updated after clustering is performed (*as explained below in more detail*).

The *STUDY.history* field is equivalent to the *history* field of the *EEG* structure. It stores all the commandline calls to the functions from the gui. For basic script writing using command history information, see the EEGLAB script writing tutorial.

The *STUDY.etc* field contains internal information that helps manage the use of the *STUDY* structure by the clustering functions. In particular, pre-clustering data are stored there before clustering is performed.

The STUDY.datasetinfo sub-structure

The *STUDY.datasetinfo* field is used for holding information on the datasets that are part of the study. Below is an example *datasetinfo* structure, one that holds information about the first dataset in the *STUDY* :

```
>> STUDY.datasetinfo(1)

ans =
```

```
            '/eegdata/data/N400/S01/'  filepath:
            'syn01_S253-clean.set'  filename:
            'S01'  subject:
            'young'  group:
            'syncomp'  condition:
            []  session:
            1  index:
            []  comps:
```

-->-->

This information was posted when the *STUDY* was created by the user.

The *datasetinfo.filepath* and *datasetinfo.filename* fields give the location of the dataset on disk.

The *datasetinfo.subject* field attaches a subject code to the dataset. Note: Multiple datasets from the same subject belonging to a *STUDY* must be distinguished as being in different experimental conditions and/or as representing different experimental sessions.

The *datasetinfo.group* field attaches a subject group label to the dataset.

The *datasetinfo.condition* and *datasetinfo.session* fields hold dataset condition and session labels. If the *condition* field is empty, all datasets are assumed to represent the same condition. If the *session* field is empty, all datasets in the same condition are assumed to have been recorded in different sessions.

The *datasetinfo.index* field holds the dataset index in the *ALLEEG* vector of currently-loaded dataset structures. (Note: Currently, *datasetinfo.index = 1* must correspond to *ALLEEG(1)* (typically, the first dataset loaded into EEGLAB), *datasetinfo.index = 2* to *ALLEEG(2)*, etc. This constraint will be removed in the future).

The *datasetinfo.comps* field holds indices of the components of the dataset that have been designated for clustering. When it is empty, all of its components are to be clustered.

The STUDY.cluster sub-structure

The *STUDY.cluster* sub-structure stores information about the clustering methods applied to the *STUDY* and the results of clustering. Components identified for clustering in each *STUDY* dataset are each assigned to one of the several resulting component clusters. Hopefully, different clusters may have spatially and/or functionally distinct origins and dynamics in the recorded data. For instance, one component cluster may account for eye blinks, another for eye movements, a third for central posterior alpha band activities, etc. Each of the clusters is stored in a separate *STUDY.cluster* field, namely, *STUDY.cluster(2)*, *STUDY.cluster(3)*, etc...

The first cluster, *STUDY.cluster(1)* , is composed of all components from all datasets that were identified for clustering. It was created when the STUDY was created and is not a result of clustering; it is the *ParentCluster*. This cluster does not contain those components whose equivalent dipole model exhibit a high percent variance from the component's scalp map. These components have been excluded from clustering. Typing *STUDY.cluster* at the Matlab command line returns

```
>> STUDY.cluster

ans =

     struct 1x23 array with fields:
     [string]   name
       [integer] parent
      [cell]    child
      [array]   comps
     [array]    sets
        [cell]  setinds
        [cell]  allinds
          [cell] algorithm
         [struct] preclust
        [cell]  erpdata
         [array] erptimes
         [cell] specdata
          [array] specfreqs
         [cell] erspdata
          [array] ersptimes
          [array] erspfreqs
        [cell]  itcdata
         [array] itctimes
         [array] itcfreqs
     [2-D array] topo
      [array]   topox
      [array]   topoy
        [cell]  topoall
        [array] topopol
```

```
        [struct] dipole
```

-->-->

All this information (including the clustering results) may be edited manually from the command line, or by using the interactive function pop_clustedit.m. Use of this function is explained above (see Editing clusters).

The *cluster.name* sub-field of each cluster is initialized according to the cluster number, e.g. its index in the cluster array (for example: 'cls 2', 'cls 3', etc.). These cluster names may be changed to any (e.g., more meaningful) names by the user via the command line or via the pop_clustedit.m interface.

The *cluster.comps* and *cluster.sets* fields describe which components belong to the current cluster: *cluster.comps* holds the component indices and *cluster.sets* the indices of their respective datasets. (Note that this did not change in EEGLAB v9 that added STUDY.design capabilities). Note also that several datasets may use the same component weights and scalp maps -- for instance two datasets containing data from different experimental conditions for the same subject and collected in the same session, therefore using the same ICA decomposition.

The *cluster.allinds* cell array contains the index of the component for each of the design defined in the *cluster.setinds* cell array (see also the STUDY.design structure below). The *cluster.comps/cluster.sets* and *cluster.allinds/cluster.setinds* values are by no mean redundant. The first couple of variable *cluster.comps/cluster.sets* is necessary to track components which are common across datasets and is being used to build the second couple of variable *cluster.allinds/cluster.setinds*. When selecting a new design *cluster.comps/cluster.sets* remains unchanged whereas *cluster.allinds/cluster.setinds* is updated.

The *cluster.preclust* sub-field is a sub-structure holding pre-clustering information for the component contained in the cluster. This sub-structure includes the pre-clustering method(s), their respective parameters, and the resulting pre-clustering PCA data matrices (for example, mean component ERPs, ERSPs, and/or ITCs in each condition). Additional information about the *preclust* sub-structure is given in the following section in which its use in further (hierarchic) sub-clustering is explained.

The *cluster.centroid* field holds the cluster measure centroids for each measure used to cluster the components (e.g., the mean or centroid of the cluster component ERSPs, ERPs, ITCs, power spectra, etc. for each *STUDY* condition), plus measures not employed in clustering but available for plotting in the interactive cluster visualization and editing function, pop_clustedit.m.

The *cluster.algorithm* sub-field holds the clustering algorithm chosen (for example *kmeans*) and the input parameters that were used in clustering the pre-clustering data.

The *cluster.parent* and *cluster.child* sub-fields are used in hierarchical clustering (see Hierarchic Clustering). The *cluster.child* sub-field contains indices of any clusters that were created by clustering on components from this cluster (possibly, together with additional cluster components). The *cluster.parent* field contains the index of the parent cluster.

The *cluster.erpdata* field contains the grand average ERP data for the component cluster. For instance for two conditions (as shown in the next example), the size of this cell will contain two elements of size [750x7] (for 750 time points and 7 components in the cluster). This cell array may be given as input to the statcond() function to compute statistics. Note that when both groups and conditions are present, the cell array may be of size { 2x3 } for 2 conditions and 3 groups, each element of the cell array containing the ERP for all the component of the cluster. The *cluster.erptimes* contains time point latencies in ms.

The *cluster.specdata* field contains spectrum data stored in a form similar to the one described above, and the *cluster.specfreqs* array contains the frequencies at which the spectrum was computed.

The *cluster.erspdata* field contains the ERSP for each component of the cluster. For instance, if this cell array contains an array of size [50x60x7], this means that there were 50 frequencies, 60 time points in the ERSP and that 7 components are present for each condition. *cluster.erspfreqs* and *cluster.ersptimes* contain the time latencies and frequency values for the ERSPs. The *cluster.itcdata* and other ITC arrays are structured in the same way.

The *cluster.topo* field contains the average topography of a component cluster. Its size is 67x67 and the coordinate of the pixels are given by *cluster.topox* and *cluster.topoy* (both of them of size [1x67]). This contains the interpolated activity on the scalp so different subjects having scanned electrode positions may be visualized on the same topographic plot. The *cluster.topoall* cell array contains one element for each component and condition. The *cluster.topopol* is an array of -1s and 1s indicating the polarity for each component. Component polarities are not fixed, in the sense that inverting both one component activity and its scalp map does not modify the back-projection of the component to the data channels). The true scalp polarity is taken into account when displaying component ERPs.

Finally, the *cluster.dipole* structure contains the centroid equivalent dipole location of the component cluster. This structure is the same as for a single component dipole (see DIPFIT2 tutorial).

Continuing with the hierarchical design introduced briefly above (in Hierarchic Clustering), suppose that Cluster 2 (*artifacts*) comprises 15 components from four of the datasets. The *cluster* structure will have the following values:

```
>> STUDY.cluster(2)

ans =
```

```
                    'artifacts'
                    {'ParentCluster 1'}
                    {'muscle 4' 'eye 5' 'heart 6'}
                    [6 10 15 23 1 5 20 4 8 11 17 25 3 4 12]
                    [1 1 1 2 2 2 3 3 3 3 4 4 4]
                    {'Kmeans' [2]}
                    [1x1 struct]
                    { [750x7 double]; [750x7 double] }
                    [750x1 double]
```

-->-->

This structure field information says that this cluster has no other *parent* cluster than the *ParentCluster* (as always, Cluster 1), but has three *child* clusters (Clusters 4, 5, and 6). It was created by the 'Kmeans' *algorithm* and the requested number of clusters was '2'.

Preclustering details are stored in the *STUDY.cluster(2).preclust* sub-structure (not shown above but detailed below). For instance, in this case, the *cluster.preclust* sub-structure may contain the PCA-reduced mean activity spectra (in each of the two conditions) for all 15 components in the cluster.

The *cluster.preclust* sub-structure contains several fields, for example:

```
>> STUDY.cluster(2).preclust

ans =

                preclustdata: [15x10 double]
                preclustparams: {1x9 cell}
            {1x4 cell} preclustcomps:
```

-->-->

The *preclustparams* field holds an array of cell arrays. Each cell array contains a string that indicates what component measures were used in the clustering (e.g., component spectra (*spec*), component ersps (*ersp*), etc...), as well as parameters relevant to the measure. In this example there is only one cell array, since only one measure (the power spectrum) was used in the clustering.

For example:

```
>> STUDY.cluster(1).preclust.preclustparams

ans =

'spec' 'npca' [10] 'norm' [1] 'weight' [1] 'freqrange' [3 25]
```

-->-->

The data measures used in the clustering were the component spectra in a given frequency range (*?freqrange' [3 25]*), the spectra were reduced to 10 principal dimensions (*'npca' [10]*), normalized (*'norm' [1]*), and each given a weight of 1 (*'weight' [1]*). When more than one method is used for clustering, then *preclustparams* will contain several cell arrays.

The *preclust.preclustdata* field contains the data given to the clustering algorithm (*Kmeans*). The data size width is the number of ICA components (15) by the number of retained principal components of the spectra (10) shown above. To prevent redundancy, only the measure values of the 15 components in the cluster were left in the data. The other components' measure data was retained in the other clusters.

The *preclust.preclustcomps* field is a cell array of size (nsubjects x nsessions) in which each cell holds the components clustered (i.e., all the components of the parent cluster).

The STUDY.changrp sub-structure

The *STUDY.changrp* sub-structure is the equivalent of the the *STUDY.cluster* structure for data channels. There is usually as many element in *STUDY.changrp* as there are data channels. Each element of *STUDY.changrp* contains one data channels and regroup information for this data channel accross all subjects. For instance, after precomputing channel measures, typing *STUDY.changrp(1)* may return

```
>> STUDY.cluster

ans =

     struct 1x14 array with fields:
                name
                channels
                chaninds
        [cell]  erpdata
```

```
        [array]erptimes
        [cell] specdata
         [array]specfreqs
        [cell] erspdata
         [array]erspfreqs
         [array]erspfreqs
       [cell]  itcdata
        [array]itctimes
        [array]itcfreqs
```

-->-->

The *changrp.name* field contains the name of the channel (i.e. 'FP1'). The *changrp.channels* field contains the channels in this group. This is because a group may contain several channels (for instance for computing measures like ERP accross a group of channels, or for instance for computing the RMS accross all data channels; note that these features are not yet completely supported in the GUI). The *changrp.allinds* cell array contains the index of the channel for each of the design defined in the *changrp.allinds* cell array (see also the STUDY.design structure below). If no channels are missing in any of the datasets, this array contains a constant for all datasets (for instance [1 1 1 1 1 1 1 1 1 ...]).

As for the component cluster structure, the *cluster.erpdata* field contains the grand average ERP data for the given channel. For instance for two conditions, the size of this cell will contain two elements of size [750x9] (for 750 time points and 9 subjects). As in cluster structure, this cell array may also be given as input to the pop_clustedit.m function to compute statistics.

When both groups and conditions are present, the cell array may expand in size similar to its component cluster counterpart: it will be of size { 2x3 } for 2 values for the first independent variable and 3 values for the second independent variable defined in the STUDY design, each element of the cell array containing the ERP for a given channel for all subjects. The *cluster.erptimes* contains time point latencies in ms. For the spectrum, ERSP and ITC array, you may refer to the cluster sub-substructure since the organization is identical (except that the last dimensions of elements in cell arrays *specdata*, *erspdata*, and *itcdata* contain subjects and not components).

The STUDY.design sub-structure

For the purpose of performing inference testing, any (m x n) design is possible (including choosing independent variables from among conditions, groups, sessions, particular stimulus-related trials, or other trial subsets). Below is a description of the STUDY design fields.

This is the current (v9.0) STUDY.design sub-structure:

```
  STUDY.design(1)

  ans =

             name: 'Design 1 - compare letter types'
         variable: [1x2 struct]
            cases: [1x1 struct]
          include: {}
            cells: [1x39 struct]
```

-->-->

Exploding the contents of each of these sub-structures, we obtain

```
             name: 'Design 1 - light and audio all subjects'

         variable: [1x2 struct]
              (1).label  : 'condition'
              (1).pairing: 'on'
              (1).value  : {'ignore'  'memorize'  'probe'}
              (2).label  : ''
              (2).pairing: 'off'
              (2).value  : {}

           cases: [1x1 struct]
              label: 'subject'
              value: {'S01'  'S02'  'S03'  'S04'  'S05'  'S06' }

         include: {}

           cells: [1x39 struct]
               dataset : 2
               trials  : {[1x267 double]}
               value   : {'ignore'  ''}
               case    : 'S01'
               filebase: '/Volumes/donnees/data/STUDYstern/S01/Ignore'
```

-->-->

- The "variable" field stands for "independent variable." Currently, up to two independent variables may be defined. STUDY.design(x).variable(1) contains the description of the first independent variable for STUDY design number x, and STUDY.design(x).variable(2) contains the description of the second independent variable (if any). Each independent variable has a 'label', a pairing status ('on', for paired data and 'off' for unpaired data), and associated values. For instance, in this specific example the independent variable 'condition' may take the values 'ignore', 'memorize' and 'probe'. As detailed in the graphic interface STUDY.design section, values may be combined by concatenating the value labels and separating them with a '-' character. For instance 'memorize - probe' is a new value for the variable 'condition' and it points to datasets containing either the 'memorize' or the 'probe' stimuli.

- The "cases" field contains the descriptions of the single 'cases' (a term adopted in statistics from clinical studies). Using the current interface, it is not possible to define "cases" other than subjects (although when plotting single subjects, selecting the option to use single trials for statistics automatically makes 'cases equivalent to 'trials'). In future versions, it will be possible to use an arbitrary variable for case.

- The "include" field is a list of independent variables and values to include in the design - for instance, to include "memorize" stimuli only (and ignore all subject datasets (or, for single-trial statistics, all trials) that do not have this independent variable value.

- The "cells" sub-structure contains the description of each cell of a given design. The number of cells is equal to number of subjects, times the number of values of first independent variable, times the number of values for the second independent variable if any (assuming each subject's data is represented for each combination of independent variable values). "dataset" contains the dataset index. "trials" contains trial indices for each dataset. "value" contains the value for the first and second independent variables. "cases" contains the case value (i.e. in V9, the subject name); "filebase" is the name of the file where the data for each cell is stored. When a given cell combines data stored in several datasets, the selected trials for each of the contributing datasets are concatenated.

Below is another example of a STUDY.design with two types of stimuli ('light' and 'audio') and two groups of subjects ("control" and "meditator").

```
STUDY.design(4).setinfo(1)

ans =

          dataset: [1 2]
           trials: { [1x43 double] [1x32 double] }
            value: { 'audio - light' 'control' }
            cases: 'c1'
         filebase: 'c1/design4_c1_audio-light_control'
```

-->-->

The first independent variable value is "audio - light"; this means that the "audio" and "light" datasets for each subject are to be merged. Therefore, before computing any given measures, selected trials (as per the "trials" field) for each dataset (1 and 2) will be concatenated.

Definition of STUDY design independent variables

Most independent variables are defined in the main STUDY interface when creating a STUDY. "condition", "group" and "session" are independent variables defined in the first STUDY editing GUI. In addition to these variables, EEGLAB will extract independent variables for each subset of data epochs based on epoch field information for the time-locking event in each dataset. The function *std_maketrialinfo* creates the "trialinfo" substructure in STUDY.datasetinfo. For instance, the first dataset in the STUDY may have the properties:

```
STUDY.datasetinfo(1).condition = 'a';
STUDY.datasetinfo(1).group = 'g1';
STUDY.datasetinfo(1).trialinfo(1).presentation = 'evoked';
STUDY.datasetinfo(1).trialinfo(2).presentation = 'evoked';
STUDY.datasetinfo(1).trialinfo(3).presentation = 'evoked';
STUDY.datasetinfo(1).trialinfo(4).presentation = 'spontaneous1';
STUDY.datasetinfo(1).trialinfo(5).presentation = 'spontaneous1';
STUDY.datasetinfo(1).trialinfo(6).presentation = 'spontaneous1';
STUDY.datasetinfo(1).trialinfo(7).presentation = 'spontaneous2';
STUDY.datasetinfo(1).trialinfo(8).presentation = 'spontaneous2';
STUDY.datasetinfo(1).trialinfo(9).presentation = 'spontaneous2';
```

-->-->

The "trialinfo" structure describes the property of each trial for dataset number 1. Trial number 1,2,3 are trials of presentation type "evoked", whereas trials 4,5,6 are trials of presentation type "spontaneous1" and trials 7,8,9 are trials of presentation type "spontaneous2".

If you want to add an independent variable, simply write a script that scans the EEG.event structure for each dataset and add a relevant field for the time locking event (event at time 0) of each data trial. For example, you may add the field "previous_event_type" that would contain the type of previous event. Once you create the STUDY, this field will automatically taken into account and appears in the STUDY design interface.

The function *std_makedesign* (or its GUI equivalent *pop_studydesign*) uses the information defined above to create STUDY designs. The content for 'variable1' entry of the *std_makedesign* function can be either a field of 'datasetinfo' or a field of 'datasetinfo.trialinfo'. Fields from 'trialinfo' are used in a similar way to fields of STUDY.datasetinfo structure. For instance if the STUDY.datasetinfo is defined as above.

To call std_makestudy(), simply say

```
std_makedesign(STUDY, ALLEEG, 1, 'variable1', 'condition');
```

-->-->

This will create

```
std_makedesign(STUDY, ALLEEG, 1, 'variable1', 'presentation');
```

-->-->

To select specific values for 'presentation'

```
std_makedesign(STUDY, ALLEEG, 1, 'variable1', 'presentation', ...
        'values1', { 'spontaneous1' 'spontaneous2' } );
```

-->-->

Note that this is equivalent to selecting specific 'presentation' values using the 'datselect' entry.

```
std_makedesign(STUDY, ALLEEG, 1, 'indvar1', 'presentation', 'datselect' ...
        , { 'presentation' { 'spontaneous1' 'spontaneous2' });
```

-->-->

In the example above, all the 'evoked' trials will be omitted from the STUDY design.

For channel and component plotting, the ".setind" and ".allind" substruture of "STUDY.changrp" (channel) and "STUDY.cluster" (ICA component clusters) keep track of the design cell index for each of the subject and independent variables. These structures are shown below

```
STUDY.changrp(x).setinds{ }
STUDY.changrp(x).allinds{ }
```

-->-->

and

```
STUDY.cluster(x).setinds{ }
STUDY.cluster(x).allinds{ }
```

-->-->

These cell arrays are of size (m, n), the first dimension referring to the first independent variable (with m values) and the second dimension referring to the second independent variable (with n values).

```
STUDY.changrp(x).setinds

ans =

    [1x5 double]
    [1x5 double]

STUDY.changrp(1).setinds{1}

ans =

     1     2     3     4     5

STUDY.changrp(1).setinds{1}

ans =

     6     7     8     9     10
```

-->-->

This indicates above that there are 5 cases (i.e., 5 subjects) for each of the two first independent variable values. There is no second independent variable (as the .setind cell array only contains 1 row). For the first value (for instance 'spontaneous1') of the independent

variable (for example 'presentation'), the indices of the cells for the 5 cases (subjects) are 1, 2, 3, 4, and 5. For the first value (for instance 'spontaneous2') of the independent variable ('presentation') the indices of the cells for the 5 cases (subjects) are 6, 7, 8, 9, and 10. It is then possible to index the cell directly. For instance, assuming we are using design number 1, the content of cell number 10 is given by

```
STUDY.design(1).cell(10).

     dataset: 5
      trials: {[1x196 double]}
       value: {'spontaneous1'  ''}
        case: 'S10'
    filebase: './s10/syn10-spontaneous1'
```

-->-->

This information allows EEGLAB to look up the files on disk. Within the data files, specific trials are selected using the .allind field (similar to the .setind field).

```
STUDY.changrp(1).allinds

ans =

    [1x5 double]
    [1x5 double]

STUDY.changrp(1).allinds{1}

ans =

     1     1     1     1     1

STUDY.changrp(1).allinds{2}

ans =

     1     1     1     1     1
```

-->-->

The array above indicates that for all cells, the datafile contain channel 1 (since we are looking at subfield STUDY.changrp(1) at position number 1 - all values are 1). This is true unless some datasets contain less electrodes than others. Also for components, the .allinds field indexes the component index of a given cluster and all the number within .allind are different.

Note that every time a new design is selected these fields are updated.

Based on the information above, to access datasets associated with a given cluster (cluster 1 below), one may simply specify.

```
[ STUDY.design(STUDY.currentdesign).setinfo(STUDY.cluster(1).setinds{1}).setindex ]
  [ STUDY.design(STUDY.currentdesign).setinfo(STUDY.cluster(1).setinds{2}).setindex ]
```

-->-->

The first line corresponds to the first value of the independent variable ('spontaneous1' in our example), the second to the second value of the independent variable ('spontaneous2'). This is how datasets are referenced in the STUDY design scheme. **Note**: In EEGLAB 8.0 and prior versions, the "setind" substructure indexed datasets directly. *There is therefore a lack of backward compatibility in v9 for this specific sub-structure*. However, we expect that all or nearly all users were only using these structures indirectly, e.g. when calling STUDY plotting functions, and these **are** backward compatible in v9.

```
STUDY.cluster(1).setinds{1}
STUDY.cluster(1).setinds{2}
```

-->-->

Understanding the .sets, .comps, .setinds, .allinds substructures for STUDY clusters

In this part, *clust* will indicate the current cluster of interest. The STUDY.cluster(clust).setinds and STUDY.cluster(clust).allinds work exactly the same as the STUDY.changrp(clust).setinds and STUDY.changrp (clust).allinds structure. STUDY.changrp(clust).setinds helps retrieve the dataset indices. STUDY. changrp (clust).allinds provide the channel indices and STUDY.cluster(clust).allinds provide the component indices.

Below we contrast the different structures used for Clustering

STUDY.cluster(clust).setinds and STUDY.cluster(clust).allinds are cell arrays that are linked with the current STUDY.design. They are adjusted each time a new STUDY.design is selected. STUDY.cluster(clust).setinds are indexes into the STUDY.design structure and

STUDY.cluster(clust).allinds are the corresponding component indices.

In contrast, STUDY.cluster(clust).sets and STUDY.cluster(clust).comps fields correspond to each other and **do NOT change with a new STUDY.design**. STUDY.cluster(clust).sets is a [datasets_with_same_ica x ncomps] matrix giving the index of the corresponding dataset in STUDY.datasetinfo and corresponds to the components listed in STUDY.cluster(clust).comps. STUDY.cluster(clust).sets and STUDY.cluster(clust).comps have the same number of columns. However, STUDY.cluster(clust).sets may have several raws if some datasets have the same ICA decomposition. If, for some reasons, the STUDY.cluster(clust).sets in not homogeneous ? some subjects have several datasets with the same decompositions and other subjects have a different number of datasets with the same decompositions, NaN are inserted for the missing datasets. However, the presence of these missing datasets may break some analysis (warning messages are displayed when relevant).

You will find that STUDY.cluster(clust).setinds{1}, for example, will have similar values as in STUDY.cluster(clust).sets. However, the order of the index values refer to different things. That is because STUDY.cluster(clust).setinds refers to datasets in a different structure: STUDY.design(currentdesign).cell where the actual datasets indices can be retrieved. If x = STUDY.cluster(clust).setinds{1}(1), then this refers to datasets indexed in STUDY.design(currentdesign).cell(x).datasets. There might be several datasets in a given cell since a STUDY design cell. For instance, if you have two groups of subjects and two conditions, you might want to simply to create a STUDY design to look at the difference between the two groups while merging the data for the two conditions. If you happen to have one dataset per conditions, EEGLAB will merge these two datasets before computing any measure (ERP, etc?).

In short, while STUDY.cluster(clust).sets refers to dataset indices, STUDY.cluster(clust).setinds are indices corresponding to the structure STUDY.design.cell. For example, if you have a cluster such as:

```
>> STUDY.cluster(clust)
ans =
         name: 'Cls 3'
         sets: [2x13 double]
        comps: [23 5 13 47 38 3 50 5 12 4 11 3 5]
       parent: {'Parentcluster 1'}
        child: []
     preclust: [1x1 struct]
      allinds: {2x1 cell}
      setinds: {2x1 cell}
    algorithm: {'dipole'  'spec'  'map'}
         topo: [67x67 double]
        topox: [67x1 double]
        topoy: [67x1 double]
      topoall: {1x13 cell}
      topopol: [1 1 -1 1 1 1 1 1 1 1 1 1 1]
```

-->-->

Here we have a STUDY design with 2 conditions (STUDY.cluster(clust).setinds and STUDY.cluster(clust).allinds have 2 rows). Assume you are interested in finding the STUDY.datasetinfo index for the first component in the STUDY.cluster(clust).allinds{1} field:

```
>> STUDY.cluster(clust).allinds{1}
ans =
4    5   23   12   13   47    3   38   50    5    3    5   11
```

-->-->

In this case, we are interested in component 4, but we do not yet know which subject or dataset this is referencing. To find the dataset index, we have to look first in STUDY.cluster(clust).setinds{1}:

```
>> STUDY.cluster(clust).setinds{1}
ans =
1    3    4    5    6    7    8   10   12   15   17   17   19
```

-->-->

Again, remember that these indices are NOT referring to STUDY.datasetinfo directly. Take the first index, in this case 1, and plug that into STUDY.design(1).cell:

```
>> STUDY.design(1).cell(1)
ans =
     dataset: 28
      trials: {[1x346 double]}
       value: {'GoTrial'  ''}
        case: 'MC02'
    filebase: [1x66 char]
```

-->-->

NOW, we can retrieve the actual STUDY.datasetinfo index,which in this case is dataset number 28.

Note that if you had more than one design, and your active design was not the first one, you might have to find the dataset index in STUDY.design(2).cell(1).

This apparently complex scheme allows to use the same clusters for all designs. STUDY.cluster(clust).sets and STUDY.cluster(clust).comps are static and do not depend on the current design. STUDY.cluster(clust).setinds and STUDY.cluster(clust).allinds are dynamically recomputed every time you change of design. We agree that it would have been better to use "cellinds" for the substructure name instead of "setinds". However, these were existing structures in EEGLAB and we wanted to keep the framework fully backward compatible.

Chapter 08: Command line STUDY functions

Building a *STUDY* from The graphic interface (as described in previous sections) calls eponymous Matlab functions that may also be called directly by users. Below we briefly describe these functions. See their Matlab help messages for more information. Functions whose names begin with *std_* take *STUDY* and/or *EEG* structures as arguments and perform signal processing and/or plotting directly on cluster activities.

Creating a STUDY

If a *STUDY* contains many datasets, you might prefer to write a small script to build the *STUDY* instead of using the pop_study.m gui. This is also helpful when you need to build many studysets or to repeatedly add files to an existing studyset. Below is a Matlab script calling the GUI-equivalent command line function std_editset.m from the "5subjects" folder:

```
[STUDY ALLEEG] = std_editset( STUDY, [], 'commands', { ...
{ 'index' 1 'load' 'S02/syn02-S253-clean.set' 'subject' 'S02' 'condition' 'synonyms' }, ...
{ 'index' 2 'load' 'S05/syn05-S253-clean.set' 'subject' 'S05' 'condition' 'synonyms' }, ...
{ 'index' 3 'load' 'S07/syn07-S253-clean.set' 'subject' 'S07' 'condition' 'synonyms' }, ...
{ 'index' 4 'load' 'S08/syn08-S253-clean.set' 'subject' 'S08' 'condition' 'synonyms' }, ...
{ 'index' 5 'load' 'S10/syn10-S253-clean.set' 'subject' 'S10' 'condition' 'synonyms' }, ...
{ 'index' 6 'load' 'S02/syn02-S254-clean.set' 'subject' 'S02' 'condition' 'non-synonyms' }, ...
        { 'index' 7 'load' 'S05/syn05-S254-clean.set' 'subject' 'S05' 'condition' 'non-synonyms' }, ...
        { 'index' 8 'load' 'S07/syn07-S254-clean.set' 'subject' 'S07' 'condition' 'non-synonyms' }, ...
        { 'index' 9 'load' 'S08/syn08-S254-clean.set' 'subject' 'S08' 'condition' 'non-synonyms' }, ...
        { 'index' 10 'load' 'S10/syn10-S254-clean.set' 'subject' 'S10' 'condition' 'non-synonyms' }, ...
{ 'dipselect' 0.15 } });
```

-->-->

Above, each line of the command loads a dataset. The last line preselects components whose equivalent dipole models have less than 15% residual variance from the component scalp map. See *>> help std_editset* for more information.

Once you have created a new studyset (or loaded it from disk), both the *STUDY* structure and its corresponding *ALLEEG* vector of resident *EEG* structures will be variables in the Matlab workspace. Typing *>> STUDY* on the Matlab command line will list field values:

```
>> STUDY =
        name: 'N400STUDY'
        filename: 'N400empty.study'
        filepath: ''
        datasetinfo: [1x10 struct]
        group: []
        session: []
        subject: {'S02' 'S05' 'S07' 'S08' 'S10'}
        condition: {'non-synonyms' 'synonyms'}
        setind: [2x5 double]
        cluster: [1x1 struct]
        notes: ''
        task: 'Auditory task: Synonyms Vs. Non-synonyms, N400'
        history:
        etc: ''
```

-->-->

Component clustering and pre-clustering

To select components of a specified cluster for sub-clustering from the command line, the call to pop_preclust.m should have the following format:

>> [ALLEEG, STUDY] = pop_preclust(ALLEEG, STUDY, cluster_id);

where 'cluster_id' is the number of the cluster you wish to sub-cluster (start with Cluster 1 if no other clusters are yet present). Components rejected because of high residual variance (see the help message of the std_editset.m function above) will not be considered for clustering.

Note: For the STUDY created above, we will first compute (or in this case load, since the measures have been precomputed) all available activity measures. Note that changing the pre-existing measure parameters might require EEGLAB to recompute or adapt some of these measures (spectral frequency range [3 25] Hz; ERSP /ITC frequency range [3 25] Hz, cycles [3 0.5], time window [-1600 1495] ms, and 'padratio' 4). To specify clustering on power spectra in the [3 30]-Hz frequency range, ERPs in the [100 600]-ms time window, dipole location information (weighted by 10), and ERSP information with the above default values, type:

```
>> [STUDY ALLEEG] = std_preclust(STUDY, ALLEEG, [], ...
{'spec' 'npca' 10 'norm' 1 'weight' 1 'freqrange' [3 25]} , ...
{'erp' 'npca' 10 'norm' 1 'weight' 1 'timewindow' [100 600]} , ...
{'dipoles' 'norm' 1 'weight' 10} , {'ersp' 'npca' 10 'freqrange' [3 25] 'cycles' [3 0.5] ...
```

```
 'alpha' NaN 'padratio' 4 'timewindow' [-1600 1495] 'norm' 1 'weight' 1});
```

-->-->

Alternatively, to enter these values in the graphic interface, type:

```
>> [STUDY ALLEEG] = pop_preclust(STUDY, ALLEEG);
```

The equivalent command line call to cluster the *STUDY* is:

```
>> [STUDY] = pop_clust(STUDY, ALLEEG, 'algorithm', 'kmeans', 'clus_num', 10);
```

or to pop up the graphic interface:

```
>> [STUDY] = pop_clust(STUDY, ALLEEG);
```

Visualizing component clusters

The main function for visualizing component clusters is pop_clustedit.m. To pop up this interface, simply type:

```
>> [STUDY] = pop_clustedit(STUDY, ALLEEG);
```

This function calls a variety of plotting functions for plotting scalp maps (std_topoplot.m), power spectra (std_specplot.m), equivalent dipoles (std_dipplot.m), ERPs (std_erpplot.m), ERSPs (std_erspplot.m), and/or ITCs (std_itcplot.m). All of these functions follow the same calling format (though std_dipplot.m is slightly different; refer to its help message). Using function std_topoplot.m as an example:

```
>> [STUDY] = std_topoplot(STUDY, ALLEEG, 'clusters', 3, 'mode', 'centroid');
```

will plot the average scalp map for Cluster 3.

```
>> [STUDY] = std_topoplot(STUDY, ALLEEG, 'clusters', 3, 'mode', 'comps');
```

will plot the average scalp map for Cluster 3 plus the scalp maps of components belonging to Cluster 3.

```
>> [STUDY] = std_topoplot(STUDY, ALLEEG, 'clusters', 4, 'comps', 3);
```

will plot component 3 of Cluster 4.

The function std_clustread.m may be used to read any information about the cluster (scalp map, power spectrum, ERSP, ITC, etc...) for further processing under Matlab. See the function help for details.

The EEGLAB developers plan to develop more functions allowing users to directly access clustering data. Some plotting functions, like the one described below, are currently available only from the command line.

Computing and plotting channel measures

You may use the function pop_precomp.m (which calls function std_precomp.m to precompute channel measures). For instance,

```
>> [STUDY ALLEEG] = pop_precomp(STUDY, ALLEEG);
```

Calls the preclustering interface. Entering:

```
>> [STUDY ALLEEG] = std_precomp(STUDY, ALLEEG, {}, 'interpolate', 'on', 'erp', 'on');
```

will interpolate all channel, and compute ERP for all channels (**{}** input) and all datasets of a given study. Plotting may then be performed using the same functions that are used for component clusters. For instance to plot the grand average ERP for channel 'FP1', you may try,

```
>> STUDY = std_erpplot(STUDY,ALLEEG,'channels',{ 'FP1'});
```

Try some other commands from the channel plotting graphic interface and look at what is returned in the history (via the eegh.m function) to plot ERP in different formats.

Plotting statistics and retrieving statistical results

All plotting function able to compute statistics will return the statistical array in their output. You must first enable statistics either from the graphic interface or using a command line call. For instance to compute condition statistics for ERP (bot channel and component clusters), type:

>> STUDY = pop_erpparams(STUDY, 'condstats', 'on');

Then, for a given channel, type

>> [STUDY erpdata erptimes pgroup pcond pinter] = std_erpplot(STUDY,ALLEEG,'channels',{ 'FP1'});

Or, for a given component cluster, type

>> [STUDY erpdata erptimes pgroup pcond pinter] = std_erpplot(STUDY,ALLEEG,'clusters', 3);

Now, typing

```
>> pcond

        pcond =

[750x1]
```

-->-->

The statistical array contains 750 p-values, one for each time point. Note that the type of statistics returned depends on the parameter you selected in the ERP parameter graphic interface (for instance, if you selected 'permutation' for statistics, the p-value based on surrogate data will be returned).

The 'pgroup' and 'pinter' arrays contain statistics across groups and the ANOVA interaction terms, respectively, if both groups and conditions are present. Note that for more control, you may also use directly call the statcond.m function, giving the 'erpdata' cell array as input (the 'erpdata' cell array is the same as the one stored in the *STUDY.cluster.erpdata* or the *STUDY.changrp.erpdata* structures for the cluster or channel of interest). See the help message of the statcond.m function for more help on this subject.

Other functions like std_specplot.m, std_erspplot.m , and std_itcplot.m behave in a similar way. See the function help messages for more details.

Modeling condition ERP differences using std_envtopo()

Important note: this functionality has been disabled in EEGLAB v9 as it was not compatible with the new design structure. This will be fixed in the future.

A *STUDY* and component cluster-based version of the ERP plotting function envtopo.m is std_envtopo.m. The std_envtopo.m function allows you to determine the contribution of specific clusters to the grand ERP of the *STUDY* - the grand mean ERP for all the datasets from each *STUDY* condition, or to determine which clusters contribute the most to the grand ERP. To find, in the clustered dataset above (before editing), the four clusters that contribute the most to the grand mean ERPs for each condition, use the following command:

```
>> std_envtopo(STUDY, ALLEEG, 'clusters', [],'subclus',[3 4 6 7 8 ], ...
 'env_erp', 'all', 'vert', -1100, 'baseline', [-200 0], ...
'diff', [1 2], 'limits', [-1300 1500 -5 5], 'only_precomp', 'on', ...
'clustnums' , -4,'limcontrib', [300 600]);
```

-->-->

The example above searches among all the clusters ('clusters', []) for the four component clusters that contribute most to the grand ERP ('clustnums', -4) in the time window [300 600] ms. In constructing the grand ERPs, five designated artifact component clusters are excluded ('subclus', [3 4 6 7 8]). The computation uses all the datasets in the STUDY ('env_erp', 'all') but only the components that were actually clustered ('only_precomp', 'on'). In this example, components with residual dipole model variance above 15% (0.15) were not clustered and thus were also not included in the grand mean ERPs. We also ask for the largest contributing clusters to the *difference wave* between the two conditions ('diff', [1 2]). For more information type: *>> help std_envtopo*

Below are the three resulting figures, one for each condition and a third for the ERP difference between the two conditions. The N400 can be easily distinguished in the grand difference ERP on the right.

In the three cases above, the four largest contributing clusters together account for 40%, 77%, and 67% of the respective grand mean ERP variances (pvaf). Three clusters (Cls 5, Cls 9, Cls 12) are among the four largest contributors to both conditions (left and middle images). These three, plus Cluster 10, account for roughly equal parts of the N400 difference (right image), implying that the N400 effect is not spatially focal or unitary, in line with results from invasive recordings.

Note that the percent variance accounted for (pvaf) values for the four clusters add to over 100%, though together they account for only 67% of difference-ERP variance. This is because the scalp projections of the four clusters are spatially correlated, not orthogonal. Therefore, their projections can cancel each other. Here, for example, the positive right frontocentral channel projections of components in Cluster 10 partially cancel the spatially overlapping negative projections of Clusters 9, 12, and 5 at the same scalp locations. In this sense, ICA can re-capture more of the source data than is captured directly in the recorded scalp channel signals.

Some other clustering procedure and a brief discussion and example of independent component clustering, from a chapter in press, is available here.

III.Advanced Topics

The EEGLAB Tutorial is split into four parts. The third part is the Advanced Topics. In the Advanced Topics, the user is introduced to more advanced elements of EEGLAB such as script writing and artifact rejection. These topics are recommended for the user to successfully master EEGLAB. These sections are not as step by step as the single subject or multiple subject processing tutorials and should be used as a reference and not a manual. As of 2/10/2012, there are three topics in the Advanced Topics.

Chapter 01: Rejecting Artifacts

Rejecting artifacts in continuous and epoched data

Strategy: The approach used in EEGLAB for artifact rejection is to use 'statistical' thresholding to 'suggest' epochs to reject from analysis. Current computers are fast enough to allow easy confirmation and adjustment of suggested rejections by visual inspection, which our eegplot.m tool makes convenient. We therefore favor *semi-automated rejection* coupled with visual inspection, as detailed below. In practice, we favor applying EEGLAB rejection methods to *independent components* of the data, using a seven-stage method outlined below.

Examples: In all this section, we illustrate EEGLAB methods for artifact rejection using the same sample EEG dataset we used in the single subject data analysis tutorial. These data are not perfectly suited for illustrating artifact rejection since they have relatively few artifacts! Nevertheless, by setting low thresholds for artifact detection it is possible to find and mark outlier trials. Though these may not necessarily be artifact related, we use them here to illustrate how the EEGLAB data rejection functions work. We encourage users to make their own determinations as to which data to reject and to analyze using the set of EEGLAB rejection tools.

Rejecting artifacts in continuous data

Rejecting data by visual inspection

Rejecting portions of continuous data can be performed visually using function pop_eegplot.m. Select Tools > Reject continuous data. This will open a warning message window, simply press *Continue*. Select data for rejection by dragging the mouse over a data region. After marking some portions of the data for rejection, press *REJECT* and a new data set will be created with the rejected data omitted. EEGLAB will adjust the *EEG.event* structure fields and will insert *boundary* events where data has been rejected, with a duration field holding the duration of the data portion that was rejected. The *boundary* events will appear in the new dataset, marked in red (see the second image below). Thus, rejection on continuous data must be performed 'before' separating it into data epochs. Note: To *deselect* a portion of the data, simply click on the selected region. This allows re-inspection of the data portions marked for rejection in two or more passes, e.g., after the user has developed a more consistent rejection strategy or threshold. See the section on Visualizing EEG data under the main tutorial for more information about how to use this interactive window.

Note: To select portions of data that extend out of the plotting window, simply drag the mouse over the new region and connect it to a previously marked region. For instance, in the following plotting window which already had the time interval 2.1 seconds to 3.4 seconds selected (as shown above), drag the mouse from 6.9 seconds back to 4.7.

Rejecting data channels based on channel statistics

Channel statistics may help determine whether to remove a channel or not. To compute and plot one channel statistical characteristics, use menu Plot > Data statistics > channel statistics. In the pop-up window below enter the channel number. The parameter "Trim percentage" (default is 5%) is used for computing the trimmed statistics (recomputing statistics after removing tails of the distribution). If P is this percentage, then data values below the P percentile and above the 1-P percentile are excluded for computing trimmed statistics.

Pressing *OK* will open the image below.

Some estimated variables of the statistics are printed as text in the lower panel to facilitate graphical analysis and interpretation. These variables are the signal mean, standard deviation, skewness, and kurtosis (technically called the 4 first cumulants of the distribution) as well as the median. The last text output displays the Kolmogorov-Smirnov test result (estimating whether the data distribution is Gaussian or not) at a significance level of p=0.05.

The upper left panel shows the data histogram (blue bars), a vertical red line indicating the data mean, a fitted normal distribution (light blue curve) and a normal distribution fitted to the trimmed data (yellow curve). The P and 1-P percentile points are marked with yellow ticks on the X axis. A horizontal notched-box plot with whiskers is drawn below the histogram. The box has lines at the lower quartile (25th-percentile), median, and upper quartile (75th-percentile) values. The whiskers are lines extending from each end of the box to show the extent of the rest of the data. The whisker extends to the most extreme data value within 1.5 times the width of the box. Outliers ('+') are data with values beyond the ends of the whiskers.

The upper right panel shows the empirical quantile-quantile plot (QQ-plot). Plotted are the quantiles of the data versus the quantiles of a Standard Normal (i.e., Gaussian) distribution. The QQ-plot visually helps to determine whether the data sample is drawn from a Normal distribution. If the data samples do come from a Normal distribution (same shape), even if the distribution is shifted and re-scaled from the standard normal distribution (different location and scale parameters), the plot will be linear.

Empirically, 'bad' channels have distributions of potential values that are further away from a Gaussian distribution than other scalp channels. For example, plotting the statistics of periocular (eye) Channel 1 below, we can see that it is further away from a normal distribution than Channel 10 above. (However, Channel 1 should not itself be considered a 'bad' channel since ICA can extract the eye movement artifacts from it, making the remained data from this channel usable for futher analysis of neural EEG sources that project to the face). The function plotting data statistics may provide an objective measure for removing a data channel. Select menu item Edit > Select data to remove one or more data channels. It is also possible to plot event statistics from the EEGLAB menu -- see the EEGLAB event tutorial for details.

Note that the function above may also be used to detect bad channels in non-continuous (epoched) data.

Rejecting artifacts in epoched data

In contrast to continuous EEG data, we have developed several functions to find and mark for rejection those data epochs that appear to contain artifacts using their statistical distributions. Since we want to work with epoched dataset, you should either load an earlier saved epoched dataset, or using the downloaded dataset (better loaded with channel locations info), use the following Matlab script code:

```
EEG = pop_eegfilt( EEG, 1, 0, [], [0]); % Highpass filter cutoff freq. 1Hz.
[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET, 'setname', 'Continuous EEG Data'); % Save as new dataset.
EEG = pop_reref( EEG, [], 'refstate',0); % Re-reference the data
EEG = pop_epoch( EEG, { 'square' }, [-1 2], 'newname', 'Continuous EEG Data epochs', 'epochinfo', 'yes'); % Epoch dataset using 'square' ev
EEG = pop_rmbase( EEG, [-1000 0]); % remove baseline
[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET, 'setname', 'Continuous EEG Data epochs', 'overwrite', 'on');
% save dataset

[ALLEEG EEG] = eeg_store(ALLEEG, EEG, CURRENTSET); % store dataset
eeglab redraw % redraw eeglab to show the new epoched dataset
```

-->-->

After having an epoched dataset, call the main window for epoched data rejection, select Tools > Reject data epochs > Reject data (all methods). The window below will pop up. We will describe the fields of this window in top-to-bottom order. First, change the bottom multiple choice button reading Show all trials marked for rejection by the measure selected above or checked below to the first option: Show only the new trials marked for rejection by the measure selected above. We will come back to the default option at the end.

Rejecting epochs by visual inspection

As with continuous data it is possible to use EEGLAB to reject epoched data simply by visual inspection. To do so, press the *Scroll data* button on the top of the interactive window. A scrolling window will pop up. Chage the scale to '79'. Epochs are shown delimited by blue dashed lines and can be selected/deselected for rejection simply by clinking on them. Rejecting parts of an epoch is not possible.

Rejecting extreme values

During 'threshold rejection', the user sets up bounding values the data should not exceed. If the data (at the selected electrodes) exceeds the given limits during a trial, the trial is marked for rejection. Enter the following values under *find abnormal values* (which calls function pop_eegthresh.m), and press *CALC/PLOT*. The parameters entered below tell EEGLAB that EEG values should not exceed +/-*75* µV in any of the 32 channels (*1:32*) at any time within the epoch (-*1000* to *2000* ms).

Marked trials are highlighted in the eegplot.m window (below) that then pops up. Now epochs marked for rejection may be un-marked manually simply by clicking on them. Note that the activity at some electrodes is shown in red, indicating the electrodes in which the outlier values were found. Press *Update Marks* to confirm the epoch markings.

At this point, a warning will pop up indicating that the marked epochs have not yet been rejected and are simply marked for rejection. When seeing this warning, press *OK* to return to main rejection window.

Also note that several thresholds may be entered along with separate applicable time windows. In the following case, in addition to the preceding rejection, additional epochs in which EEG potentials went above *25* µV or below *-25* µV during the *-500* to ms interval will be marked for rejection.

Rejecting abnormal trends

Artifactual currents may cause linear drift to occur at some electrodes. To detect such drifts, we designed a function that fits the data to a straight line and marks the trial for rejection if the slope exceeds a given threshold. The slope is expressed in microvolt over the whole epoch (50, for instance, would correspond to an epoch in which the straight-line fit value might be 0 µv at the beginning of the trial and 50 µ v at the end). The minimal fit between the EEG data and a line of minimal slope is determined using a standard R-square measure. To test this, in the main rejection window enter the following data under *find abnormal trends* (which calls function pop_rejtrend.m, and press *CALC/PLOT*.

The following window will pop up. Note that in our example the values we entered are clearly too low (otherwise we could not detect any linear drifts in in this clean EEG dataset). Electrodes triggering the rejection are again indicated in red. To deselect epochs for rejection, click on them. To close the scrolling window, press the *Update marks* button.

Note: Calling function pop_rejtrend.m either directly from the command line, or by selecting Tools > Reject data epochs > Reject flat line data, allows specifying additional parameters.

Rejecting improbable data

By determining the probability distribution of values across the data epochs, one can compute the probability of occurrence of each trial. Trials containing artifacts are (hopefully) improbable events and thus may be detected using a function that measures the probability of occurrence of trials. Here, thresholds are expressed in terms of standard deviations of the mean probability distribution. Note that the probability measure is applied both to single electrodes and the collection of all electrodes. In the main rejection window, enter the following parameters under *find improbable data* press *Calculate* (which calls function pop_jointprob.m), and then press *PLOT*.

The first time this function is run on a dataset, its computation may take some time. However the trial probability distributions will be stored for future calls. The following display shows the probability measure value for every trial and electrode (each number at the bottom shows an electrode, the blue traces the trial values). On the right, the panel containing the green lines shows the probability measure over all the electrodes. Rejection is carried out using both single- and all-channel thresholds. To disable one of these, simply raise its limit (e.g. to 15 std.). The probability limits may be changed in the window below (for example, to 3 standard deviations). The horizontal red lines shift as limit values are then modified. Note that if you change the threshold in the window below the edit boxes in the main window will be automatically updated (after pressing the *UPDATE* button).

It is also possible to look in detail at the trial probability measure for each electrode by clicking on its sub-axis plot in the above window. For instance, by clicking on electrode number two on the plot, the following window would pop up (from left to right, the first plot is the ERPimage of the square values of EEG data for each trial, the second plot indicates which trials were labeled for rejection (all electrodes combined) and the last plot is the original probability measure plot for this electrode with limits indicated in red).

Close this window and press the *UPDATE* button in the probability plot window to add to the list of trials marked for rejection.

Now press the '*Scroll data* button to obtain the standard scrolling data window. Inspect the trials marked for rejection, make any adjustments manually, then close the window by pressing *UPDATE MARKS*.

Rejecting abnormally distributed data

Artifactual data epochs sometimes have very 'peaky' activity value distributions. For instance, if a discontinuity occurs in the activity of data epoch for a given electrode, the activity will be either close to one value or the other. To detect these distributions, a statistical measure called kurtosis is useful. Kurtosis is also known as the fourth cumulant of the distribution (the skewness, variance and mean being the first three). A high positive kurtosis value indicates an abnormally 'peaky' distribution of activity in a data epoch, while a high negative kurtosis value indicates abnormally flat activity distribution. Once more, single- and all-channel thresholds are defined in terms of standard deviations from mean kurtosis value. For example, enter the following parameters under *find abnormal distribution* press *Calculate* (which calls function pop_rejkurt.m), then press *PLOT*.

In the main rejection window note that the *Currently marked trials* field have changed to 5. In the plot figure all trials above the threshold (red line) are marked for rejection, note that there are more than 5 such lines. The figure presents all channels, and the same trial can exceed the threshold several times in different channles, and therefore there are more lines than marked trials.

As with probability rejection, these limits may be updated in the pop-up window below, and pressing *Scroll data* in the main rejection window (above) brings up the standard data scrolling window display which may be used to manually adjust the list of epochs marked for rejection.

Rejecting abnormal spectra

According to our analysis (Delorme, A., Makeig, S., Jung, T.P., Sejnowski, T.J. (2001), "Automatic rejection of event-related potential trials and components using independent component analysis"), rejecting data epochs using spectral estimates may be the most effective method for selecting data epochs to reject for analysis. In this case, thresholds are expressed in terms of amplitude changes relative to baseline in dB. To specify that the spectrum should not deviate from baseline by *+/-50* dB in the *0-2* Hz frequency window, enter the following parameters under *find abnormal spectra* (which calls function pop_rejspec.m), then press *CALC/PLOT*.

Computing the data spectrum for every data epoch and channel may take a while. We use a frequency decomposition based on the slepian multitaper (Matlab pmtm() function) to obtain more accurate spectra than using standard Fourier transforms. After computing the trial spectra, the function removes the average power spectrum from each trial spectrum and tests whether the remaining spectral differences exceed the selected thresholds. Here, two windows pop up (as below), the top window being slaved to the bottom. The top window shows the data epochs and the bottom window the power spectra of the same epochs. When scrolling through the trial spectra, the user may use the top window to check what caused a data trial to be labeled as a spectral outlier. After further updating the list of marked trials, close both windows by pressing *UPDATE MARKS* in the bottom window.

As with standard rejection of extreme values, several frequency thresholding windows may be defined. In the window below, we specify that the trial spectra should again not deviate from the mean by *+/-50* dB in the *0-2* Hz frequency window (good for catching eye movements) and should not deviate by *+25* or *-100* dB in the *20-40* Hz frequency window (useful for detecting muscle activity).

Inspecting current versus previously proposed rejections

To compare the effects of two different rejection thresholds, the user can plot the currently and previously marked data epochs in different colors. To do this, change the option in the long rectangular tab under *Plotting options* (at the buttom of the figure). Select Show previous and new trials marked for rejection by this measure selected above. For instance, using *abnormal distribution*, enter the following parameters and press *Scroll data*.

The scrolling window appears and now shows that at trials 11 & 12 the blue window is actually divided into two, with dark blue on the top and light blue on the bottom. Dark blue indicates that the trial is currently marked as rejected. The light blue indicates that the trial was already rejected using the previous rejection threshold.

Note: Pressing *UPDATE MARKS* only updates the currently rejected trials and optional visual adjustments. Previously rejected trials are ignored and can not be removed manually in the plot window.

Inspecting results of all rejection measures

To visually inspect data epochs marked for rejection by different rejection measures, select Show all trials marked for rejection measures by the measure selected above or checked below in the long rectangular tab under *Plotting options*. Then press *CALC/PLOT* under *Find abnormal spectra*. Rejected windows are divided into several colored patches, each color corresponding to a specific rejection measure. Channels colored red are those marked for rejection by any method.

When visually modifying the set of trials labeled for rejection in the window above, all the *current rejection measure* are affected and are reflected in the main rejection window after you press the *UPDATE MARKS* button.

Notes and strategy

- The approach we take to artifact rejection is to use statistical thresholding to suggest epochs to reject from analysis. Current computers are fast enough to allow easy confirmation and adjustment of suggested rejections by visual inspection. We therefore

favor *semi-automated rejection* coupled with visual inspection.

- All the functions presented here can also be called individually through Plot > Reject data epochs or from the Matlab command line.
- After labeling trials for rejection, it is advisable to *save* the dataset before actually rejecting the marked trials (marks will be saved along with the dataset). This gives one the ability go back to the original dataset and recall which trials were rejected.
- To actually reject the marked trials either use the option *Reject marked trials* at the buttom of the main rejection window, or use the main eeglab window options Tools > Reject data epochs > Reject marked epochs.
- All these rejection measures are useful, but if one does not know how to use them they may be inefficient. Because of this, we have not provided standard rejection thresholds for all the measures. In the future we may include such information in this tutorial. The user should *begin* by visually rejecting epochs from some test data, then adjust parameters for one or more of the rejection measures, comparing the visually selected epochs with the results of the rejection measure. All the measures can capture both simulated and real data artifacts. In our experience, the most efficient measures seem to be frequency threshold and linear trend detection.
- We typically apply semi-automated rejection to the *independent component activations* instead of to the raw channel data, as described below.

Rejection based on independent data components

We usually apply the measures described above to the activations of the independent components of the data. As independent components tend to concentrate artifacts, we have found that bad epochs can be more easily detected using independent component activities. The functions described above work exactly the same when applied to data components as when they are applied to the raw channel data. Select Tools > Reject data using ICA > Reject data (all methods). We suggest that the analysis be done iteratively in the following seven steps:

1. Visually reject unsuitable (e.g. paroxysmal) portions of the continuous data.
2. Separate the data into suitable short data epochs.
3. Perform ICA on these epochs to derive their independent components.
4. Perform semi-automated and visual-inspection based rejection of data epochs on the derived components. (*)
5. Visually inspect and select data epochs for rejection.
6. Reject the selected data epochs.
7. Perform ICA a second time on the pruned collection of short data epochs -- This may improve the quality of the ICA decomposition, revealing more independent components accounting for neural, as opposed to mixed artifactual activity. If desired, the ICA unmixing and sphere matrices may then be applied to (longer) data epochs from the same continuous data. Longer data epochs are useful for time/frequency analysis, and may be desirable for tracking other slow dynamic features.
8. Inspect and reject the components. Note that components should NOT be rejected before the second ICA, but after.

(*) After closing the main ICA rejection window, select Tools > Reject data using ICA > Export marks to data rejection and then Tools > Reject data epochs > Reject by inspection to visualize data epochs marked for rejection using ICA component activities.

Chapter 02: Writing EEGLAB Scripts

This section is intended for users who have learned at least the basics of Matlab script writing and wish to use EEGLAB and its many functions to automate and/or customize data analyses. This section mainly uses the same sample EEG dataset as the single subject data analysis tutorial.

Why write EEGLAB Matlab scripts?

EEGLAB is a collection of Matlab functions many of which can be called from a main graphic interface. Writing EEGLAB Matlab scripts simply involves calling these functions from a script file or from the command line instead of calling them interactively from the EEGLAB gui. EEGLAB's history mechanism keeps track of all operations performed on datasets from the EEGLAB graphic interface and eases the transition from menu-based to script-based computing. It allows the user to perform exploratory signal processing on a sample dataset, then use the accumulated commands issued from the EEGLAB window in a script file, which can then be modified using any text editor.

Writing Matlab scripts to perform EEGLAB analyses allows the user to largely automate the processing of one or more datasets. Because advanced analyses may involve many parameter choices and require fairly lengthy computations, it is often more convenient to write a custom script, particularly to process multiple datasets in the same way or to process one dataset in several ways.

Note: Writing EEGLAB Matlab scripts requires some understanding of the EEGLAB data structure (EEG) and its substructures (principally *EEG.data*, *EEG.event*, *EEG.urevent*, *EEG.epoch*, *EEG.chanlocs* and *EEG.history*). We will introduce these data structures as needed for the tutorial examples and will discuss the five reserved variable names used by EEGLAB and their uses:

```
EEG        - the current EEG dataset
ALLEEG     - array of all loaded EEG datasets
CURRENTSET - the index of the current dataset
LASTCOM    - the last command issued from the EEGLAB menu
ALLCOM     - all the commands issued from the EEGLAB menu
```

You may refer at any time to Appendix A5. EEGLAB Data Structures for a more complete description of the EEG structure.

Using dataset history to write EEGLAB scripts

This section begins with a basic explanation of the EEG Matlab structure used by EEGLAB. It then explains how to take advantage of the history of modifications of the current dataset for writing scripts. Finally we describe how EEGLAB functions are organized and how to use these functions in Matlab scripts. Following sections describe how to take advantage of EEGLAB structures to process multiple datasets.

In EEGLAB, the data structure describing the current dataset can be accessed at all times from the Matlab command line by typing *>> EEG*. The variable *EEG* is a Matlab structure used by EEGLAB to store all the information about a dataset. This includes the dataset name and filename, the number of channels and their locations, the data sampling rate, the number of trials, information about events in each of the trials/epochs, the data itself, and much more. For a more complete description of the *EEG* fields along with examples on sample data, see Appendix A2. The contents of any field of the *EEG* structure may be accessed by typing *EEG.fieldname*. For instance, typing *>> EEG.nbchans* on the Matlab command line returns the number of channels in the current dataset.

Dataset history: the EEG.history field

EEGLAB commands issued through the EEGLAB menu that have affected the current EEG dataset are preserved in the *EEG.history* field. The contents of the history field includes those function calls that modified the current dataset, as well as calls to plotting functions. For instance, following the single subject tutorial to the end of section "Importing channel locations" and then typing *>> EEG.history* on the command line would return the following text:

```
pop_eegplot( EEG, 1, 1, 1);
EEG.setname='Continuous EEG Data';
EEG = eeg_eegrej( EEG, [295 512] );
EEG.chanlocs=pop_chanedit(EEG.chanlocs, 'load',{ '/matlab/eeglab/sample_data/eeglab_chan32.locs', 'filetype', 'autodetect'});
figure;
topoplot([],EEG.chanlocs, 'style', 'blank', 'electrodes', 'labelpoint');
```

-->-->

The first command *pop_eegplot(EEG, 1, 1, 1);* plotted the data. The second and the third commands removed portions of the data. These three commands were inherited from the parent dataset. The last command plotted the channel locations by name. As we will see, a basic method for writing EEGLAB scripts is simply to save or copy and paste these history commands into a Matlab script file.

Processing multiple datasets: One typical use of dataset history for writing scripts is to process several datasets. To process the first dataset,

you can use EEGLAB graphic interface. To process subsequent similar datasets, you may simply copy or save the history from the first dataset into a script file (a text file with the extension "*.m*", for example, *doitagain.m*), load a different dataset, and then run the script from the Matlab command line. Note that the script file *doitagain.m* must be in your current Matlab path, which normally includes the present working directory. Read the help messages for Matlab functions *path()* and *addpath()* to learn more about the Matlab path. For instance, to repeat the processing applied to your current dataset onto another dataset, use menu File > Save history > Dataset history to save the history of your current dataset as file *doitagain.m* as shown below.

Clicking ?Save? in the window above will cause the command history to be saved into the Matlab script file *doitagain.m* (you can choose any name for this file, as long as it ends in the standard Matlab script file extension, "*.m*").
Note: When the file was saved, an extra command, *>> eeglab redraw* was added at the end to insure that the main graphic interface would be updated after the dataset was processed.

Now, to process another dataset using the same commands you used for processing the current dataset, try closing the current Matlab session, restart Matlab then EEGLAB, reload a different dataset and run the script saved above by typing

```
>> doitagain
```

Most of the commands in the history field call EEGLAB *pop_* functions. These are functions that take as input the EEG structure. The next sub-section discusses how to use these functions in EEGLAB scripts.

EEGLAB Function Architecture

EEGLAB was designed for use by both novice and expert Matlab users. Depending on their level of Matlab expertise, users can either interact only with the EEGLAB graphic interface (GUI), else they can call EEGLAB functions directly from the Matlab command line or write their own Matlab scripts using EEGLAB functions and structures.

EEGLAB functions are grouped in three layers:

1. The main eeglab function and its menu handlers: EEGLAB users typically call these functions by selecting menu items from the main EEGLAB window menu.

1. Pop_functions: Matlab functions with their own graphic interfaces. Called with no (or few) arguments (as from the EEGLAB user interface), these functions pop up a query window to gather additional parameter choices from users. They then generally call one or more of the EEGLAB toolbox signal processing functions. The pop_functions can also be called from the Matlab command line or from Matlab scripts.

1. Signal processing functions: The experienced Matlab user can call the ICA toolbox functions directly from the Matlab command line or from their own analysis scripts. Some EEGLAB helper functions are also in this layer.

We will first see how pop_function work.

EEGLAB pop_ functions

Functions with the prefix *pop_* or *eeg_* are functions that take the EEG structure as their first input argument. Functions with the prefix *pop_* can be called either from the EEGLAB menu or from the Matlab command line, while functions with the prefix *eeg_* can only be called from the Matlab command line. When you select a menu entry in the EEGLAB main window, EEGLAB calls a *pop_* function, usually providing it

with one parameter only, the EEG structure containing the current dataset (when selecting a menu item, the pop_ function it calls is listed in the title bar of the pop-up window). Since the pop_ function is not given enough parameters to actually perform any processing, it pops up a window to ask the user to provide additional parameters. When you have entered the required parameters into the pop_ window, the data processing is performed. EEGLAB then adds the complete function call to the dataset history, including the parameters you have entered in the pop-up window. If you later copy this command from the dataset history and paste it onto the Matlab command line, the processing will be performed directly, without popping up an interactive query window. However, try removing all the input parameters to the function call except the first, naming the EEG structure and the pop_function will now pop up a query window before performing any processing.

For example, open a new Matlab session and try (you may have to type *>> eeglab* to add access paths to the functions below)

```
>> EEG = pop_loadset;
```

An interactive window will pop up to ask for the dataset name, just as it would do if the pop_loadset.m command were issued from the EEGLAB menu via File > Load dataset. If, on the other hand, the user provides two string arguments to the pop_loadset.m function, the first containing the filename and the second the file path, no interactive window appears and the dataset is loaded directly.
Try another example:

```
>> EEG = pop_eegfilt(EEG);
```

This will pop up an interactive window allowing you to filter the data according to the parameters you enter in the window. If you wish to filter the EEG dataset without involving this graphic interface, type:

```
>> EEG = pop_eegfilt( EEG, 1, 0);
```

This command will highpass filter the data above 1 Hz. To see which parameter this function takes as argument see pop_eegfilt.m help. Keep in mind that all the interactive EEGLAB pop_ functions work this way. You may copy commands from the EEG history fields and modify the function input as desired. Function help messages are available either from the EEGLAB graphic interface Help > EEGLAB functions > Interactive pop_ function, from the Internet, or from the command line (type *>> help pop_*functioname).

Note: Only *pop_[funcname]()* functions or *eeg_[funcname]()* functions process the EEG dataset structure; *eeg_funcname()* functions take the EEG data structure as an argument, but do not pop up interactive windows. Thus, they are typically not available from the EEGLAB menu, but only from the command line.

What do pop_ functions return?

When called from the EEGLAB interface, pop_ functions do not return variables. Instead, they may alter (when called for) the EEG data structure itself. However, when called from the command line, many of the visualization functions in the EEGLAB toolbox do return variables when useful (e.g., the results plotted). To determine which variables are returned by a function, it is important to understand how they work. To carry out their required data processing, most of the pop_ functions (each named *pop_[funcname]()*) call a similarly named processing function (*[funcname]*). You may directly call these functions to perform more advanced scripting (see low level scripting below). The important thing is that both the pop_ function and its corresponding processing function return the same variables (usually the pop_ function help messages refer the user to the processing function help message which describes the output variables). For example, the pop_erpimage.m function returns the same outputs as the erpimage.m function:

```
figure; [outdata, outvar, outtrials] = pop_erpimage(EEG,1,12); % ERP-image plot of channel 12

% or the equivalent non-pop function call
figure; [outdata, outvar, outtrials] = erpimage( EEG.data(12,:), zeros(1,EEG.trials), EEG.times, , 10, 1, 'nosort');
```

Important note: If *pop_[funcname]()* is a plotting function, then a new figure is created automatically only when the function is called in pop-up window mode. Otherwise, *pop_[funcname]()* plotting commands (as well as all non-pop plotting commands, except *eegplot()*) should be preceded by a Matlab *figure;* command, as in the example above (Note: the *figure;* is added before the command by the EEGLAB history mechanism). This feature allows you to create compound figures using Matlab *subplot()*or the more flexible EEGLAB version sbplot.m.

Script examples using dataset history

Making use of the *EEG.history* is the easiest way to start learning about EEGLAB scripting. For example, import a binary dataset (for instance **TEST.CNT**), then follow the single subject tutorial until the end of Section "Extracting data epochs" (in this section, do not enter any event type name for epoch extraction; the function will use all events). Then type *>> EEG.history* on the command line. You should obtain the following text:

```
EEG = pop_loadcnt('/home/arno/temp/TEST.CNT' , 'dataformat', 'int16');
EEG.setname='CNT file';
pop_eegplot( EEG, 1, 1, 1);
EEG.setname='Continuous EEG Data';
EEG = eeg_eegrej( EEG, [295 512] );
EEG.chanlocs=pop_chanedit(EEG.chanlocs, 'load',{ '/matlab/eeglab/sample_data/eeglab_chan32.locs', 'filetype',  autodetect'});
figure; topoplot([],EEG.chanlocs, 'style', 'blank', 'electrodes', 'labelpoint');
```

```
figure; pop_spectopo(EEG, 1, [0 238304.6875], 'EEG' , 'percent', 15, 'freq', [6 10 22], 'freqrange',[2   25],'electrodes','off');
EEG = pop_eegfilt( EEG, 1, 0, [], [0]);
EEG.setname='Continuous EEG Data';
EEG = pop_epoch( EEG, { 'square' }, [-1 2], 'newname', 'Continuous EEG Data epochs', 'epochinfo', 'yes');
EEG.setname='Continuous EEG Data epochs';
EEG = pop_rmbase( EEG, [-1000 0]);
```

-->-->

This is the history field of dataset 2 (the epoched dataset), if you switch to dataset 1 (the original continuous dataset), by selecting menu item Datasets > dataset 1, and then type >> *EEG.history* on the command line, you will retrieve the same list of commands as above except for the last three. This is because dataset 2 is derived from dataset 1, so it inherits all the history of modifications that were applied to dataset 1 up to the time dataset 2 was created from it.

The *EEG.history* command can be very useful when you have several datasets (for example, from several subjects) and wish to apply the same processing to all of them. The *EEG.history* field is a part of the dataset EEG structure, so you can use it in any EEGLAB session. For example, when you have new dataset you wish to process the same way as a previous dataset, just load the old dataset into EEGLAB and type >> *EEG.history* to see the list of commands to execute on the new dataset. More specifically,

1. Load all the datasets you wish to process into EEGLAB.
2. Perform the processing you wish from the Matlab menu on the first dataset.
3. Ask for the command history (type >> *EEG.history*) and copy the data processing commands.
4. Switch (via the EEGLAB menu) to the second dataset and paste the buffered commands onto the Matlab command line to execute them again on the new dataset.
5. Go on like this till the last dataset is processed.

At this point you may want to save all the modified datasets to the computer. You may also use menu File > Save history > Dataset history to save the current dataset history (*EEG.history* field) into a Matlab script file and recall this Matlab script from the command line as described in the previous sub-section.

Note: EEGLAB loading dataset commands (menus File > Load dataset) are not stored in the dataset history. The reason for this is that if you were to load a dataset repeatedly, you would not want the repeated load command to be in your dataset history.

More advanced scripting examples will be presented in the following sections.

Updating the EEGLAB window

Whenever you wish to switch back from interacting with the EEG dataset on the command line to working with the EEGLAB graphic interface, you should perform one of the two commands below:

1. If no EEGLAB window is running in the background, type:

   >> eeglab redraw;
2. If there is an open EEGLAB session, first type:

```
>> [ALLEEG EEG CURRENTSET] = eeg_store(ALLEEG, EEG);
```

to save the modified information, and then:

```
>> eeglab redraw;
```

to see the changes reflected in the EEGLAB window.

To learn more about scripting and EEGLAB data structures see the next section Scripting at the EEGLAB Structure Level

Using EEGLAB session history to perform basic EEGLAB script writing

There are two main EEGLAB Matlab data structures, *EEG* and *ALLEEG*. The *ALLEEG* array contains all the dataset structures that currently loaded in the EEGLAB session. The *EEG* structures contains all the information about the current dataset being processed.

There are two main differences between EEGLAB ?dataset history? and ?session history?. As the names imply, ?session history? saves all the function calls issued for all the datasets in the current EEGLAB session, while ?dataset history? saves only the function calls that modified the current dataset. Session history is available only during the current session of EEGLAB -- starting a new EEGLAB session will create a new session history -- whereas dataset history is saved in the *EEG.history* field of the EEG dataset structure when you save the dataset at the end of the session. It therefore will be retrieved when the dataset is re-loaded in future EEGLAB sessions (assuming, of course, that you save the dataset at the end of the current session!).

EEGLAB session history allows you to manipulate and process several datasets simultaneously. Thus, its use can be considered the next level of EEGLAB scripting.

The h command

To view the session history for the current EEGLAB session, use the *h* (history) command. Typing:

```
>> h
```

under Matlab prints the EEGLAB session history in the Matlab command line window. For instance, after performing the first step of the main tutorial (simply opening an existing dataset), typing *h* on the command line should return the following text:

```
>>[ALLEEG EEG CURRENTSET ALLCOM] = eeglab;
>>EEG = pop_loadset( 'eeglab_data.set', '/home/Matlab/eeglab/script/');
>>[ALLEEG EEG CURRENTSET] = eeg_store(ALLEEG, EEG);
```

The first command (eeglab.m) runs EEGLAB and initializes several EEGLAB variables listed in the function output. Except for modifying these variables and adding the path to EEGLAB functions (if necessary), the eeglab.m call will not modify anything else in the Matlab workspace (there is no global variable in EEGLAB). The second command (pop_loadset.m) loads the dataset into the *EEG* structure, and the last (eeg_store.m) stores the dataset in the *ALLEEG* structure. For more detailed information, you must study the Matlab help messages for these functions as explained below:

Either (1) via the EEGLAB menu selections:

For pop_loadset.m]: via Help > EEGLAB functions > Interactive pop_functions or via Help > EEGLAB menus

For eeg_store.m: via Help > EEGLAB advanced > Admin functions

Or (2) using Matlab command line help

```
>> help [functioname]
```

Now use menu item File > Save history > Session history to save the command history into an ascii-text Matlab script file. Save the file into the current directory, or into a directory in the Matlab command path (i.e., in the list returned by *>> path*). Selecting the ?Save session history? menu item above will pop up the window below:

Clicking ?Save? in the window above will cause the session command history to be saved into the Matlab script file *eeglabhist.m* (you can choose any name for this file, as long as it ends in the standard Matlab script file extension, ?.m?). Now try closing the current Matlab session, restarting Matlab, and running the script saved above by typing

>> eeglabhist

The main EEGLAB window is created and the same dataset is loaded.

Now open the script file *eeglabhist.m* in any text editor so you may modify function calls.
Note: as for the dataset history, when the file was saved, an extra command, *>> eeglab redraw* was added at the end to insure that the main graphic interface would be updated after the dataset was (re)loaded.

Script example using session history

Building and running short or long EEGLAB Matlab scripts saved by EEGLAB history can be that simple. Simply perform any EEGLAB processing desired via the EEGLAB menu, save the EEGLAB command history, and re-run the saved script file. Matlab will repeat all the steps you performed manually. For instance, following the first several steps of the main tutorial, the command *>> h* would return (with Matlab-style comments in black italic format added for clarity):

```
[ALLEEG EEG CURRENTSET ALLCOM] = eeglab; % start EEGLAB under Matlab
EEG = pop_loadset( 'ee114squares.set', '/home/payton/ee114/'); % read in the dataset
[ALLEEG EEG CURRENTSET] = eeg_store(ALLEEG, EEG); % copy it to ALLEEG

EEG = pop_editeventfield( EEG, 'indices', '1:155', 'typeinfo', 'Type of the event'); % edit the dataset event field
[ALLEEG EEG] = eeg_store(ALLEEG, EEG, CURRENTSET); % copy changes to ALLEEG

% update the dataset comments field
EEG.comments = pop_comments('', '', strvcat( 'In this experiment, stimuli can appear at 5 locations ',
'One of them is marked by a green box ', 'If a square appears in this box, the subject must respond, otherwise
he must ignore the stimulus.', ' ', 'These data contain responses to (non-target) circles appearing in the attended
box in the left visual field '));

[ALLEEG EEG] = eeg_store(ALLEEG, EEG, CURRENTSET);% copy changes to ALLEEG
pop_eegplot( EEG, 1, 0, 1); % pop up a scrolling window showing the component activations

EEG.chanlocs=pop_chanedit(EEG.chanlocs, { 'load', '/home/payton/ee114/chan32.locs'},{ 'convert',{ 'topo2sph', 'gui'}},{ 'convert',{ 'sph2ca
% read the channel location file and edit the channel location information ''

figure; pop_spectopo(EEG, 0, [-1000 237288.3983] , 'percent', 20, 'freq', [10], 'icacomps', [1:0],'electrodes','off');
% plot RMS power spectra of the ICA component activations;
% show a scalp map of total power at 10 Hz plus maps of the components contributing most power at the same frequency''
```

-->-->

Important note: As briefly mentioned previously, functions called from the main EEGLAB interactive window display the name of the underlying pop_function in the window title bar. For instance, selecting File > Load an existing dataset to read in an existing dataset uses EEGLAB function pop_loadset.m.

The next steps in learning to write EEGLAB Matlab scripts involve learning to change EEGLAB function parameters and adding loops to perform multiple analyses.

Scripting at the EEGLAB structure level

The type of scripting illustrated above might involve going back and forth between EEGLAB graphic interface and the Matlab command line. To maintain consistency between the two main EEGLAB structure (*EEG* and *ALLEEG*), you need to update the *ALLEEG* every time you modify the *EEG* structure (see exception below in (1)). To add or directly modify *EEG* structure values from a script or the Matlab command line, one must respect some simple rules:

1) If the EEGLAB option to ?Retain parent dataset?, selected via the File > Maximize Memory menu item (for details, see Appendix A3. for the maximizing memory menu), is set (default), then all current EEGLAB datasets are stored in the structure array *ALLEEG,*. If you modify a dataset, you should take care to copy the modified EEG dataset into *ALLEEG*.

Thus, after loading and then modifying an *EEG* structure to create a new dataset, one might simply type:

```
>>ALLEEG(2) = EEG;
>>CURRENTSET = 2;
```

This command 'might' work as expected (if the new dataset is internally consistent with the previous one). However, it is better to use the command eeg_store.m' which performs extensive dataset consistency checking before storing the modified dataset. Either use the following command to set the new dataset to be dataset number 2,

```
>> [ALLEEG EEG] = eeg_store(ALLEEG, EEG, 2);
```

or

```
>>[ALLEEG EEG CURRENTSET] = eeg_store(ALLEEG, EEG);
```

to create a new dataset at the next available free space in the *ALLEEG* variable. The dataset number will then be available in the variable *CURRENTSET*. Note that if a previous dataset is already assigned as dataset 2, then only the last command (above) will not overwrite it. To view the changes in the main EEGLAB window, use the command: *>> eeglab redraw;*
Another command that can be used to modify the *ALLEEG* structure is pop_newset.m. This command, which also performs extensive dataset consistency checks, has more useful advanced options. To modify the current dataset with its accumulated changes type:

```
[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET, 'overwrite', 'on');
```

If you wish to create a new dataset to hold the modified structure use:

```
>>[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET);
```

The returned argument *CURRENTSET* holds the set number of the new dataset stored in EEGLAB.
Note: the *EEG* contains only the current dataset, so you must use extra caution whenever updating this structure. e.g., Be sure it contains the dataset you actually want to process!

The functions above call the function eeg_checkset.m to check the internal consistency of the modified dataset.

```
>> EEG = eeg_checkset(EEG);
```

or

```
>> EEG = eeg_checkset(EEG, 'eventconsistency');
```

The second command above runs extra checks for event consistency (possibly taking some time to complete) and regenerates the *EEG.epoch* structures from the *EEG.event* information. This command is only used when the event structure is being altered. See the Event tutorial to learn how to work with EEG events.

The commands above are very useful if the option to maintain multiple datasets is on. If the option to maintain multiple datasets is off (via the File > Maximize Memory menu item), the *ALLEEG* variable is not used and *EEG* is the only variable that contains dataset information. When using this option you can only process one dataset at a time (the goal here is to use less memory and being able to process bigger datasets). Any changes made by the user to the *EEG* structure are thus applied instantaneously and are irreversible. For consistency, all the commands above will work, however the *ALLEEG* variable will be empty.

2) New fields added to the *EEG* structure by users will not be removed by EEGLAB functions. Any additional information about a dataset might be stored in the user-added field:

```
>>EEG.analysis_priority = 1;
```

3) The following are the reserved variable names used by EEGLAB:

```
EEG        - the current EEG dataset
ALLEEG     - array of all loaded EEG datasets   (Thus >> EEG = ALLEEG(CURRENTSET);)
CURRENTSET - the index of the current dataset
LASTCOM    - the last command issued from the EEGLAB menu
ALLCOM     - all the commands issued from the EEGLAB menu during this EEGLAB session
```

Note: that EEGLAB does not use global variables (the variables above are accessible from the command line but they are not used as global variables within EEGLAB). The above variables are ordinary variables in the global Matlab workspace. All EEGLAB functions except the main interactive window function eeglab.m (and a few other display functions) process one or more of these variables explicitly as input parameters and do not access or modify any global variable. This insures that they have a minimum chance of producing unwanted 'side effects' on the dataset.

Basic scripting examples

Below is a simple example of a Matlab script that includes some of the first basic manipulations that must be performed on a dataset. This example works with the tutorial dataset *eeglab_data.set* and the corresponding channel location file *eeglab_chan32.locs*, which are assumed to be located on your computer in the following directory: */home/Matlab/eeglab/script/*.

```
[ALLEEG EEG CURRENTSET ALLCOM] = eeglab;
% Load eeglab
EEG = pop_loadset( 'eeglab_data.set', '/home/Matlab/eeglab/script/');
% Load the dataset
EEG.chanlocs=pop_chanedit(EEG.chanlocs, 'load',{ '/home/Matlab/eeglab/script/eeglab_chan32.locs', 'filetype', 'autodetect'});
% Load the channel location file, enabling automatic detection of channel file format'
[ALLEEG EEG CURRENTSET ] = eeg_store(ALLEEG, EEG);% Store the dataset into EEGLAB
EEG = pop_eegfilt( EEG, 1, 0, [], [0]);''''' % High pass filter the data with cutoff frequency of 1 Hz.
% Below, create a new dataset with the name filtered Continuous EEG Data
[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET, 'setname', 'filtered Continuous EEG Data');% Now CURRENTSET= 2
EEG = pop_reref( EEG, [], 'refstate',0); % Re-refrence the new dataset
% This might be a good time to add a comment to the dataset.
EEG.comments = pop_comments(EEG.comments,'','Dataset was highpass filtered at 1 Hz and rereferenced.',1);
% You can see the comments stored with the dataset either by typing >> EEG.comments or selecting the menu option Edit->About this dataset.
EEG = pop_epoch( EEG, { 'square' }, [-1 2], 'newname', 'Continuous EEG Data epochs', 'epochinfo', 'yes');
% Extract epochs time locked to the event - 'square', from 1 second before to 2 seconds after those time-locking events.
% Now, either overwrite the parent dataset, if you don't need the continuous version any longer, or create a new dataset
%(by removing the 'overwrite', 'on' option in the function call below).
[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET, 'setname', 'Continuous EEG Data epochs', 'overwrite', 'on');
EEG = pop_rmbase( EEG, [-1000 0]); % Remove baseline
% Add a description of the epoch extraction to EEG.comments.
EEG.comments = pop_comments(EEG.comments,'','Extracted ''square'' epochs [-1 2] sec, and removed baseline.',1);
[ALLEEG EEG] = eeg_store(ALLEEG, EEG, CURRENTSET);  %Modify the dataset in the EEGLAB main window
eeglab redraw % Update the EEGLAB window to view changes
```

-->-->

Some other useful scripting examples (see the function help messages for more details):

1. Reduce sampling rate

```
% Reduce the sampling rate to 128 Hz (the above example was already sampled at 128 Hz'')
EEG = pop_resample( EEG, 128);
% Save it as a new dataset with the name Continuous EEG Data resampled
[ALLEEG EEG CURRENTSET] = pop_newset(ALLEEG, EEG, CURRENTSET, 'setname', 'Continuous EEG Data resampled');
% If you wish to return to the previous dataset (before downsampling), type
'EEG = eeg_retrieve(ALLEEG, 1); CURRENTSET = 1;
```

-->-->

2. Print a series of ERP scalp maps

```
% Plot ERP maps (via the second argument choice 1), every 100 ms from 0 ms to 500 ms [0:100:500]
% with the plot title - 'ERP image', in 2 rows and 3 columns. Below, the 0 means do not plot dipoles.
% Plot marks showing the locations of the electrodes on the scalp maps.
pop_topoplot(EEG,1, [0:100:500] , 'ERP image',[2:3] ,0, 'electrodes', 'on');
```

-->-->

In the next section, we will directly call some lower-level EEGLAB data processing functions. For instance, the command above can be executed by directly calling the signal processing function topoplot.m as shown below:

```
times = [0:100:500];
% Define variables:
pos = eeg_lat2point(times/1000, 1, EEG.srate, [EEG.xmin EEG.xmax]);
% Convert times to points (or >pos = round( (times/1000-EEG.xmin)/(EEG.xmax-EEG.xmin) * (EEG.pnts-1))+1;)
% See the event tutorial for more information on processing latencies
mean_data = mean(EEG.data(:,pos,:),3);
% Average over all trials in the desired time window (the third dimension of EEG.data allows to access different data trials).
% See appendix A1 for more information
maxlim = max(mean_data(:));
minlim = min(mean_data(:));
% Get the data range for scaling the map colors.
maplimits = [ -max(maxlim, -minlim) max(maxlim, -minlim)];
% Plot the scalp map series.
figure
for k = 1:6
sbplot(2,3,k);
% A more flexible version of subplot.
topoplot( mean_data(:,k), EEG.chanlocs, 'maplimits', maplimits, 'electrodes', 'on', 'style', 'both');
title([ num2str(times(k)) ' ms']);
end
cbar; % A more flexible version of colorbar.
```

-->-->

Low level scripting

As mentionned at the end of section IV.2.3, *pop_funcname()* function is a graphic-user interface (gui) function that operates on the *EEG* data structure using the stand-alone processing function *funcname()*. The stand-alone processing function, which has no knowledge of the dataset structure, can process any suitable data matrix, whether it is an EEGLAB data matrix or not.

For instance, pop_erpimage.m calls the data processing and plotting function erpimage.m. To review the input parameters to these functions, either use the EEGLAB help menu (from the EEGLAB window) or the Matlab function help (from the Matlab command line). For example:

>> help pop_erpimage
>> help erpimage

As mentioned earlier, the two following function calls are equivalent:

>> figure; [outdata, outvar, outtrials] = pop_erpimage(EEG,1,12);
>> figure; [outdata, outvar, outtrials] = erpimage(EEG.data(12,:), zeros(1,EEG.trials), EEG.times, *, 10, 1, 'nosort')*

Using EEGLAB data processing functions may require understanding some subtleties of how they work and how they are called. Users should read carefully the documentation provided with each function. Though for most functions, the function documentation is supposed to describe function output in all possible situation, occasionally users may need to look in the function script files themselves to see exactly

how data processing is performed. Since EEGLAB functions are open source, this is always possible.

Example script for processing multiple datasets

For example, when computing event-related spectral power (ERSP) transforms for sets of data epochs from two or more experimental conditions, the user may want to subtract the same (log) power baseline vector from all conditions. Both the pop_timef.m function and the timef.m function it calls return spectral baseline values that can be used in subsequent *timef()* computations. For instance, assuming that three sets of data epochs from three experimental conditions have been stored for 10 subjects in EEGLAB dataset files named *subj[1:10]data[1:3].set* in directory */home/user/eeglab*, and that the three datasets for each subject contain the same ICA weights, the following Matlab code would plot the ICA component-1 ERSPs for the three conditions using a common spectral baseline for each of the 10 subjects:

```
eeglab; % Start eeglab
Ns = 10; Nc = 3; % Ns - number of subjects; Nc - Number of conditions;'
for S = 1:Ns  % For each of the subjects
        mean_powbase = % Initialize the baseline spectra average over all conditions for each subject
for s =1:Nc  % Read in existing EEGLAB datasets for all three conditions
          ['subj' int2str(S) 'data' int2str(s) '.set'];  % Build dataset name
              EEG = pop_loadset('filepath', '/home/user/eeglab/'); % Load the dataset
[ALLEEG EEG] = eeg_store(ALLEEG, EEG, Nc*(S-1) + s);  % Store the dataset in ALLEEG
[ersp,itc,powbase{s}] =pop_timef( ALLEEG(s),0, 1, [-100 600], 0, 'plotitc', 'off', 'plotersp', 'off' );
% Run simple timef() for each dataset, No figure is created because of options 'plotitc', 'off', 'plotersp', 'off'
                [mean_powbase; =powbase{s}];  % Note: curly braces
end % condition
% Below, average the baseline spectra from all conditions
        mean_powbase = mean(mean_powbase, 1);

% Now recompute and plot the ERSP transforms using the same baseline
figure;  % Create a new figure (optional figure('visible', 'off'); would create an invisible figure)
for s = 1:Nc; % For each of the three conditions
      (1,3,s); %subplot Select a plotting region
         ( ALLEEG(s),0, 1, [-100 600], 0, 'powbase', mean_powbase, ...
title', ['Subject ' int2str(S)]);''''' % Compute ERSP using mean_powbase''
end % End condition plot
        plotname = ['subj' int2str(S) 'ersp' ];  % Build plot name
eval(['print -depsc ' plotname]); % Save plot as a color .eps (postcript) vector file
end % End subject
eeglab redraw  % Update the main EEGLAB window
```

-->-->

Repetitive processes, such as the computation performed above, may be time consuming to perform by hand if there are many epochs in each dataset and many datasets. Therefore it may be best performed by an EEGLAB Matlab script that is left to run until finished in a Matlab session. Writing scripts using EEGLAB functions makes keeping track of data parameters and events relatively easy, while maintaining access to the flexibility and power of the Matlab signal processing and graphics environment.

Notes:

- Normally, the user might want to accumulate and save the ERSPs and other output variables returned by *timef()* above to make possible further quantitative comparisons between subjects. The function described in the next paragraph tftopo.m allows the user to combine ERSP outputs from different subjects and apply binary statistics.
- In the current version of EEGLAB, the cross-coherence function crossf.m can calculate significance of differences between coherences in two conditions.
- In the future, timef.m will be extended to allow comparisons between multiple ERSP and ITC transforms directly.
- The same type of iterative normalization (illustrated above) may be applied for the "baseamp" parameter returned by pop_erpimage.m

Example script performing time-frequency decompositions on all electrodes

This more advanced example demonstrates some of the power of low-level scripting that goes beyond the scope of functions currently available through the graphical interface. You can run this script on any epoched dataset including the tutorial dataset.

```
% Compute a time-frequency decomposition for every electrode
for elec = 1:EEG.nbchan
[ersp,itc,powbase,times,freqs,erspboot,itcboot] = pop_timef(EEG, ?
1, elec, [EEG.xmin EEG.xmax]*1000, [3 0.5], 'maxfreq', 50, 'padratio', 16, ...
'plotphase', 'off', 'timesout', 60, 'alpha', .05, 'plotersp','off', 'plotitc','off');
if elec == 1  % create empty arrays if first electrode
          zeros([size(ersp) EEG.nbchan]);
         zeros([size(itc) EEG.nbchan]);
             zeros([size(powbase) EEG.nbchan]);
           zeros([size(times) EEG.nbchan]);
           zeros([size(freqs) EEG.nbchan]);
```

```
            zeros(size(erspboot) EEG.nbchan]);
            zeros(size(itcboot) EEG.nbchan]);
end;
        allersp(sec) = ersp;
        allitc(sec) = itc;
        allpowbase(sec) = powbase;
        alltimes(sec) = times;
        allfreqs(sec) = freqs;
        allerspboot(sec) = erspboot;
        allitcboot(sec) = itcboot;
end;
% Plot a tftopo() figure summarizing all the time/frequency transforms
figure;
tftopo(allersp,alltimes(:,:,1),allfreqs(:,:,1),'mode','ave','limits', ?
[nan nan nan 35 -1.5 1.5],'signifs', allerspboot, 'sigthresh', [6], 'timefreqs', ...
[400 8; 350 14; 500 24; 1050 11], 'chanlocs', EEG.chanlocs);
```

-->-->

Executing the following code on the tutorial dataset (after highpass filtering it above 1 Hz, extracted data epochs, and removing baseline), produces the following figure.

Creating a scalp map animation

A simple way to create scalp map animations is to use the (limited) EEGLAB function eegmovie.m from the command line. For instance, to make a movie of the latency range -100 ms to 600 ms, type:

```
pnts = eeg_lat2point([-100:10:600]/1000, 1, EEG.srate, [EEG.xmin EEG.xmax]);
% Above, convert latencies in ms to data point indices
figure; [Movie,Colormap] = eegmovie(mean(EEG.data(:,128:2:192),3), EEG.srate, EEG.chanlocs, 0, 0);
seemovie(Movie,-5,Colormap);
```

-->-->

A second solution here is to dump a series of images of your choice to disk, then to assemble them into a movie using another program. For instance, type

```
counter = 0;
for latency = -100:10:600 %-100 ms to 1000 ms with 10 time steps
figure; pop_topoplot(EEG,1,latency, 'My movie', [] ,'electrodes', 'off'); % plot'
print('-djpeg', sprintf('movieframe%3d.jpg', counter)); %save as jpg
close;  % close current figure
        counter = counter + 1;
end;
```

-->-->

Then, for example in Unix, use *% convert movieframe*.jpg mymovie.mpg* to assemble the images into a movie.

Refer to the event scripting tutorial for more script and command line examples (see next tutorial regarding accessing events from the command line).

(More example scripts would be useful here. Send us your own commented script).

Chapter 03: Event Processing

This tutorial complements information about EEGLAB event handling covered in other parts of the EEGLAB tutorial. It first describe how to import, modify, select, and visualize EEGLAB events, within the EEGLAB graphic interface. It then describes in more details the *EEG.event*, *EEG.urevent*, and related *EEG.epoch* structures, and discusses importing, editing, and using event information in EEGLAB scripts. We assume that readers are already familiar with the single subject tutorial as well as with the EEGLAB script tutorial.

Important Note: The EEGLAB event structure has evolved through EEGLAB Version 4.4, after which we hope it may remain stable (note that functions processing events are backward compatible with old event structures).

Event Processing in the EEGLAB GUI

Event fields

The EEG event structure contains records of the experimental events that occurred while the data was being recorded, as well as structure handling events which are automatically inserted by EEGLAB. The experimental events can be imported into EEGLAB using its GUI: refer to importing event information for continuous EEG data for details. To view the information for an event, use menu Edit > Event values (see the window below, which shows events of the tutorial dataset after the main tutorial section on extracting data epochs).

The *type* and *latency* fields are the most important EEGLAB event fields (see below). Some EEGLAB functions recognize and use these two fields for plotting events, sorting trials, etc. (Note: One can also process events lacking these fields, though this strongly limits the range of event processing possibilities). Other fields, including *epoch*, *duration*, *urevent*, are also recognized by EEGLAB (they are created automatically during extracting of epochs, rejecting data, or storing event data). User-defined fields can have any other name that is relevant to the data (for example, *position* in the example above).

A short description of recognized event fields is given below. Further information can be found in the event scripting section in the next sections.

- type - specifies the type of the event. For example, 'square' in the example above is a stimulus type, 'rt' is a subject button-press (i.e., reaction-time) event, etc... In continuous datasets, EEGLAB may add events of type 'boundary' to specify data boundaries (breaks in the continuous data). The next section on event scripting provides more detailed information about this special event type.
- latency - contains event latencies. The latency information is displayed in seconds for continuous data, or in milliseconds relative to the epoch's time-locking event for epoched data. As we will see in the event scripting section, the latency information is stored internally in data samples (points or EEGLAB 'pnts') relative to the beginning of the continuous data matrix (EEG.data).
- duration - duration of the event. This information is displayed in seconds for continuous data, and in milliseconds for epoched data. Internally, duration is (also) stored in data samples (pnts).
- urevent - contains indices of events in the original ('ur' in German) event structure. The first time events are imported, they are copied into a separate structure called the 'urevent' structure. This field is hidden in the graphic interface (above) since they should not be casually modified.
- epoch - indices of the data epochs (if any) the event falls within.

Note: all event fields may contain either numbers or strings (except for *latency*, *duration*, *urevent* and *epoch* which must contain numbers). For instance, the *type* field can contain a number (i.e. 1,2, etc.) or a string (i.e. 'square', 'rt', etc.). However, EEGLAB cannot process a

mixture of both formats (numeric and string) for one field. A function that checks the event structure (eeg_checkset.m with flag *eventconsistency*), called each time a dataset is modified, enforces each field's contents to remain consistent, automatically converting numeric values to strings if necessary. This constraint was imposed to speed up event manipulation by a factor of about 1000X.

Importing, adding, and modifying events

Events can be imported into EEGLAB by selecting menu item File > Import event info. To find out more about this option, refer to the "importing event information for a continuous EEG dataset" tutorial. The same method may also be used to add new events to a pre-existing event table.

To insert new events manually, select menu item Edit > Event values. Click on the *Insert event* button to add a new event before the current event. The *Append event* button adds an event *after* the current event. After a new event has been inserted or appended, its event-field information can immediately be changed in the corresponding boxes. For instance, to insert a event of type *new* 500 ms after the onset of the time-locking stimulus, enter *new* in the event *type* edit box, and 500 in the event *latency* edit box.
Note: If you then click on the *cancel* button, none of the new information will be saved.

After you press *OK*, the events may be resorted (events must always be in order of increasing latency), and some field contents may be modified to ensure consistency, as indicated at the end of the previous section.

In the graphic interface above, all experimental events can be manually modified by simply typing the new values in the edit window for each field. Events may also be deleted (*Delete event* button) or resorted (see the bottom part of the window). Resorting events is only for viewing purposes (as EEGLAB cannot process an event structure that is not sorted by increasing event latency).

Another way to insert new events is to select menu item Edit > Event fields as shown below. Through its graphic interface, you may add new fields and import Matlab arrays or text files to fill them. For instance, to add a *test* field containing only *0*s, enter *test* in the edit box of the *Edit field* column. Then enter *0* in the *file/array* column and press *OK*. Using this window, you may also delete event fields (last column) or rename them (window bottom). Finally you may change the description of event fields by clicking on buttons in the *Field description* column. (Note: It is efficient to enter event field description information in the first dataset you create for an experiment, and then to copy this information to subsequent datasets from the same experiment. Entering event field descriptions will allow you or others who may explore the same data later (perhaps when new analysis tools become available or of interest) to quickly grasp the meaning of the event information.

Selecting events

To select specific events, use menu item Edit > Select epochs/events. This can be exploited to compare different types of data epochs, as described in the "Selecting events and epochs for two conditions" section of the main tutorial. That section describes a basic event selection process. However, you can also specify more complex combinations of event field selections and ranges as the criterion for selecting trials. For instance, the window below would select epochs containing events of type *rt*, not following stimuli presented at 'position' 1, in which subject reaction time latency is between 0 ms and 400 ms, from among the 40 first epochs of the parent dataset.
Note: The option set above the *Cancel* button (below): *Remove epochs not referenced by any selected event*. If this checkbox were left unset and the checkbox *Keep only selected events and remove all other events*, the function would simply select the specified events but would not remove epochs not containing those events.
Note: To select events outside the given range, check the *Select events NOT selected above* box to the right of the field range entry. It is possible to rename the type of the selected events (while (optionally) keeping the old event type name as a new field) using the last two edit boxes.

Using events for data processing

Event information can be used in several EEGLAB processing functions as listed below.

1. The event *type* field can be used to extract data epochs, as described in section "I.5 Extracting data epochs". Select menu option Tools > extract epochs to open the window below.

   This operation will extract epochs of duration one second before to two seconds after all events of type "square".

2. Events can also be used for making ERP-image plots. Select Plot > Channel ERP image. This brings up the pop_erpimage.m window (below).

The *Epoch-sorting field* button allows you to choose any of the event fields as the sorting criteria. The next button, *Event type(s)*, enables you to choose only a subset of events. In this example, events of type "square" are sorted by their position field value. For more details see the "Plotting ERP images" section of the main tutorial.

3. Event statistics can be plotted with regard to different event fields. For example, selecting menu option Plot > Data statistic > Event statistic the following window pops up.

Entering 'rt' as the event type, and "[250 500]" (ms) as the event latency range, and pressing *OK* will show the following 'rt' event statistics.

See the (rejection tutorial on data channel statistics) for more information on this graphic interface.

4. Most EEGLAB functions take events into account. For instance, functions that remove data (pop_eegplot.m, pop_select.m) will also remove events that occur during the data removed (though not their corresponding urevents). Functions that process continuous data (pop_spectopo.m, pop_resample.m], pop_mergeset.m) will take into account 'boundary' events (data events added by EEGLAB to note portions of the data that have been removed or at 'hard' boundaries between concatenated datasets).

Event processing from the Matlab command line

This section is essential for users who want to write EEGLAB/Matlab scripts that select and/or process data time-locked to given types or combinations of events. It is also important for any EEGLAB user who wants to know how EEGLAB data and events are actually stored and manipulated.

Accessing events from the command line

Event types

Event fields of the current data structure can be displayed by typing *>> EEG.event* on the Matlab command line. For instance, using the unprocessed tutorial dataset,

```
>> EEG.event
```

returns,

```
ans =

      stim: 1x157 struct array with fields:
type
                position
```

```
                latency
                urevent
```

-->-->

To display the field values for the first event, type:

>> EEG.event(1)

This will return,

```
ans =
type: 'square'
        position:
        latency:
        urevent:
```

-->-->

Remember that custom event fields can be added to the event structure and will thereafter be imported with the event information whenever the dataset is loaded. Therefore, the names of some event fields may differ in different datasets. Note that event field information can be easily retrieved using commands such as *>> {EEG.event.fieldname}*. For example,

>> {EEG.event(1:5).type}

returns the contents of the *type* field for the first 5 events:

```
ans =
'square' 'square' 'rt' 'square' 'rt'
```

-->-->

Use the following commands to list the different event types in the unprocessed tutorial dataset:

```
>> unique({EEG.event.type});

ans=
'rt' 'square'
```

-->-->

The command above assumes that event types are recorded as strings. Use *>> unique(cell2mat({EEG.event.type}));* for event types stored as numbers.

You may then use the recovered event type names to extract epochs. Below is the command-line equivalent of the epoch extraction procedure presented above in section ?I.5 Extracting data epochs?.

>>EEG = pop_epoch(EEG, { 'square' }, [-1 2], 'epochinfo', 'yes');

If you decide to process data from the command-line directly, do not forget to first remove baseline activity,

>> EEG = pop_rmbase(EEG, [-1000 0]);

See the Script tutorial for more details on how to write Matlab scripts using EEGLAB.

Event latencies

We may use the same command as in the section above to display the contents of the event latency field. Event latencies are stored in units of data sample points relative to (0) the beginning of the continuous data matrix (EEG.data). For the tutorial dataset (before any processing), typing:

>>{EEG.event(1:5).latency}

returns the first 5 event latencies,

ans =

[129.0087] [218.0087] [267.5481] [603.0087] [659.9726]

To see these latencies in seconds (instead of sample points above), you need first to convert this cell array to an ordinary numeric array (using EEGLAB cell2mat.m function), then subtract 1 (because the first sample point corresponds to time 0) and divide by the sampling rate. Therefore,

>>(cell2mat({EEG.event(1:5).latency})-1)/EEG.srate

returns the same event latencies in seconds,

ans =

1.0001 1.6954 2.0824 4.7032 5.1482

For consistency, for epoched datasets the event latencies are also encoded in sample points with respect to the beginning of the data (as if the data were continuous). Thus, after extracting epoch from the tutorial dataset ("Data Epochs")

>>{EEG.event(1:5).latency}

returns the first 5 event latencies,

ans =

[129] [218.00] [267.5394] [424] [513]

Note that for an epoched dataset this information has no direct meaning. Instead, select menu item Edit > Event values (calling function pop_editeventvals.m) to display the latency of this event in seconds relative to the epoch time-locking event. From the command-line, you may use the eeg_function eeg_point2lat.m to convert the given latencies from data points relative to the beginning of the data to latencies in seconds relative to the epoch time-locking event. For example:

```
>> eeg_point2lat(cell2mat({EEG.event(1:5).latency}), ...
cell2mat({EEG.event(1:5).epoch}), EEG.srate, [EEG.xmin EEG.xmax])

ans =

0 0.6953 1.0823 -0.6953 0
```

-->-->

(The reverse conversion can be accomplished using function eeg_lat2point.m)

The most useful function for obtaining event information from the command-line is EEGLAB function eeg_getepochevent.m. This function may be used for both continuous and epoch data to retrieve any event field for any event type. For example, using the tutorial data (after epoch extraction), type in

>> [rt_lat all_rt_lat] = eeg_getepochevent(EEG, 'rt', [], 'latency');

to obtain the latency of events of type *rt*. The first output is an array containing one value per data epoch (the *first* event of the specified type in each data epoch). The second output is a cell array containing the field values for *all* the relevant events in each data epoch. Latency information is returned in milliseconds. (Note: The third input allows searching for events in a specific time window within each epoch. An empty value indicates that the whole epoch time range should be used).

Similarly to obtain the value of the event 'position' field for 'square' events in each epoch, type:

>> [rt_lat all_rt_lat] = eeg_getepochevent(EEG, 'square', [], 'position');

Continuous data behave as a single data epoch, so type:

>> [tmp all_sq_lat] = eeg_getepochevent(EEG, 'square');

to obtain the latencies of all 'square' events in the continuous data (via the second output).

Urevents

In EEGLAB v4.2 and above, a separate "ur" (German for "original") event structure, *EEG.urevent*, holds all the event information originally loaded into the dataset. If some events or data regions containing events are removed from the data, this should not affect the *EEG.urevent* structure. If some new events are inserted into the dataset, the urevent structure is automatically updated. This is useful to obtain information on the *context* of events in the original experiment. Even after extracting data epochs, the prior *context* of each event in a continuous or epoched dataset is still available. Currently the *EEG.urevent* structure can only be examined from the command line.

The *EEG.urevent* structure has the same format as the '*EEG.event* structure. The *urevent* field in the event structure (e.g.,

EEG.event(n).urevent) contains the index of the corresponding event in the urevent structure array -- thereby 'pointing' from the event to its corresponding urevent, e.g., its "original event" number in the continuous data event stream. For example, if a portion of data containing the second event is removed from a continuous dataset during artifact rejection, the second event will not remain in the *EEG.event* structure. It *will* remain, however, in the *EEG.urevent* structure. e.g. the second *EEG.event* might now point to the third *EEG.urevent*:

```
>> EEG.event(2).urevent

ans =
3
```

-->-->

Note that 'urevent' indices in the *EEG.event* structure do not have to increase linearly. For example after epochs were extracted from the tutorial dataset,

```
>> {EEG.event(1:5).urevent}

ans =

[1] [2] [3] [1] [2]
```

-->-->

This means that events 1 and 2 (in the first data epoch) and events 4 and 5 (in the second data epoch) are actually the same original events.

At present (v4.4), a few EEGLAB command-line functions use the urevent structure: eeg_time2prev.m, eeg_urlatency.m and eeg_context.m. We will use the important eeg_context.m function in a script example below. The next section provides more insight into the relation between the *EEG.event* and *EEG.urevent* structures.

Boundary events

Events of reserved type 'boundary' are created automatically by EEGLAB when portions of the data are rejected from continuous data or when continuous datasets are concatenated. These events indicate the latencies and (when possible) the durations of the discontinuities these operations introduce in the data. In the image below, a portion of data that included event #2 was rejected, and a type 'boundary' event was added to the event structure. Its index became 2, since events are sorted by their latencies. The urevent structure continues to hold the original event information, with no added 'boundary' event.

Boundary events are standard event structures with *event.type* = 'boundary'. They also have an *event.duration* field that holds the duration of the rejected data portion (in data samples). (Note: Since all events in a dataset must have the same set of fields, in datasets containing boundary events every event will have a 'duration' field -- set by default to 0 except for true boundary type events. Boundary events are used by several signal processing functions that process continuous data. For example, calculating the data spectrum in the pop_spectopo.m function operates only on continuous portions of the data (between boundary events). Also, data epoching functions will (rightly!) not extract epochs that contain a boundary event.

Epoched datasets do not have boundary events between data epochs. Instead of being stored in a 2-D array of size (channels, sample_points) like continuous data, epoched data is stored in a 3-D array of size (channels, sample_points, trials). Events in data epochs are stored as if the data were continuous, in effect treating the data as a 2-D array of size (channels, (sample_points*trials)). This format makes handling events from the command-line more convenient.

The purpose of the *EEG.urevent* structure is to retain the full record of experimental events from the original continuous data, as shown in the image below. Function eeg_context.m uses 'urevent' information to find events defined by their neighboring event 'context' in the experiment (and original data).

'Hard' boundaries between datasets

When continuous datasets are concatenated, a 'harder' type of boundary event must be inserted, this time into both the *EEG.event* and *EEG.urevent* structures. In particular, if the first urevent in the pair was the last event in one dataset, and the next urevent was the first event in the next concatenated dataset (which need not have been recorded at the same time), the latencies of the neighboring pair of urevents cannot be compared directly. Such so-called "hard" boundary events marking the joint between concatenated datasets have the usual type "boundary" but a special "duration" value, *NaN* (Matlab numeric value "not-a-number"). They are the only "boundary" events present in *EEG.urevent* and are the only type "boundary" events in *EEG.event* with a "duration" of "NaN" and an *EEG.event.urevent* pointer to an urevent. Hard "boundary" events are important for functions such as eeg_context.m that are concerned with temporal relationships among events in the original experiment (i.e., among urevents).

The 'epoch' structure

The *EEG.epoch* structure is empty in continuous datasets, but is automatically filled during epoch extraction. It is computed from the *EEG.event*structure by the function eeg_checkset.m (with flag 'eventconsistency') as a convenience for users who may want to use it in writing EEGLAB scripts. One of the few EEGLAB functions that use the *EEG.epoch* structure (all 'eeg_' command-line functions) is eeg_context.m. Each *EEG.epoch* entry lists the type and *epoch* latency (in msec) of every event that occurred during the epoch. The following example was run on the tutorial set after it was converted to data epochs.

```
>> EEG.epoch

ans =
     struct 1x80 array with fields:
                event
                eventlatency
                eventposition
                eventtype
                eventurevent
```

-->-->

Note that this dataset contains 80 epochs (or trials). Now type

```
>> EEG.epoch(1)

ans =

       [1x2 int8]
        event: [1 2]
        eventlatency: {[0] [1.695.3125] [1.0823e+03]}
        eventposition: {[21] [2] [2]}
        eventtype: {'square' 'square' 'rt'}
        eventurevent: {[1 2] [2] [3]}
```

-->-->

The first field *EEG.epoch(1).event* contains the indices of all events that occurred during this epoch. The fields *EEG.epoch(1).eventtype*, *EEG.epoch(1).eventposition* and *EEG.epoch(1).eventlatency* are cell arrays containing the event field values of each of the events in that epoch. Note that the latencies in *EEG.epoch(1).eventlatency* are in milliseconds with respect to the epoch time-locking event.

When extracting epochs, it is possible to remove all but a selected set of events from the data. For example, if there is only one event in an epoch, the epoch table may look more readable. Using the tutorial dataset after extracting data epochs, select item Edit > Select epoch/event in the menu, and then enter (in the pop-up window below) "rt" in the *Event type* field, then select *Keep only selected events and remove all other events* instead of *Remove epochs not referenced by any event*. Press *OK* to create a new data set. Note: This daughter (and its future child datasets, if any) contains no trace (except in *EEG.urevent*) of all the events that actually occurred during the experiment but were

erased by this process. Therefore, we strongly recommend against using this option!

then typing

```
>> EEG.epoch(1)
```

-->-->

returns

```
ans =

       1event:
        eventlatency: 1.0823e+03
        eventposition:
        eventtype:
        eventurevent:
```

-->-->

This means that epoch number "1" contains a single event of type "rt" at latency 1082.3 ms, it also indicates that this is the first event in the dataset (i.e. *event: 1*), but note that it was the third event in the original dataset, since its corresponding urevent (stored in *EEG.event.urevent*) is "3".

Writing scripts using events

The scripts below are relatively advanced EEGLAB/Matlab scripts. The Script tutorial contains basic details on how to write Matlab scripts using EEGLAB.

The following script illustrates how to manipulate a dataset event list using the (epoched) tutorial dataset, by adding a new class of (pseudo) events to the existing dataset. Here, we want to add new events whose latencies are 100 ms before existing 'square'-type events. (Note: Say we might want to do this to allow re-epoching of the data relative to the new events).

```
% Add new events to a loaded dataset
nevents = length(EEG.event);
for index = 1 : nevents
if ischar(EEG.event(index).type) && strcmpi(EEG.event(index).type, 'square')
% Add events relative to existing events
    event(end+1)=EEG.event(index); % Add event to end of event list
% Specifying the event latency to be 0.1 sec before the referent event (in real data points)
    event(end).latency = EEG.event(index).latency - 0.1*EEG.srate;
    event(end).type = 'cue'; '% Make the type of the new event 'cue
end;
end;

EEG = eeg_checkset(EEG, 'eventconsistency'); % Check all events for consistency
[ALLEEG EEG CURRENTSET] = eeg_store(ALLEEG, EEG, CURRENTSET); % Store dataset
eeglab redraw % Redraw the main eeglab window
```

-->-->

Event context (as more reliably retrieved from the 'EEG.urevent' table) can be valuable in data analysis. For example, one may want to study all events of a certain type (or set of types) that precede or follow events of some other types by delays within a given range. A classic ERP

example was the demonstration by Squires et al. that the P300 peak was larger when a rare stimulus followed a string of standard stimuli than when it followed other rare stimuli. The (v4.4) command-line function eeg_context.m looks for events of specified *target* type(s), then for preceding and/or succeeding events of specified *neighbor* type(s).

The script below is a simple demonstration script using eeg_context.m that finds target events in the sample dataset that are followed by appropriate button press responses. Some response ambiguity is resolved by making sure the response does not follow a succeeding target. (In that case, we assume the response is to the second target). It also finds the responded targets that follow a non-responded target since, we expect, the EEG dynamics time-locked to these targets may differ in some way from those not following a 'miss'.

```
% For each target 'square' event, find the succeeding 'rt' and 'square' events

[targsrt,nextrts,nxtypert,rtlats] = eeg_context(EEG,{'square'},{'rt'},1);  % find succeeding rt events
[targssq,nextsqs,nxtypesq,sqlats] = eeg_context(EEG,{'square'},{'square'},1);  % find succeeding square events

% find target 'square' events with appropriate button-press responses

selev = []; % selected events
rejev = [];  % rejected events
for e=1:length(targsrt) % For each target,
if rtlats(e) > 200 & rtlats(e) < 1500 ...  % if 'rt' latency in acceptable range
  (sqlats(e) > rtlats(e) | isnan(sqlats(e))) % and no intervening 'square' event,
        [selev targsrt(e),1)];  % mark target as responded to
else  % Note: the isnan() test above tests for NaN value
% of sqlats (when no next "square")
        [rejev targsrt(e),1)];
end
end
rts = rtlats(selev); % collect RT latencies of responded targets
whos rtlats sqlats nextrts nextsqs
selev = targs(selev); % selev now holds the selected EEG.event numbers
rejev = targs(rejev);
fprintf(['Of %d "square" events, %d had acceptable "rt" responses ',...
'not following another "square".\n'],length(targsrt),length(selev));

%
% find targets following non-responded targets
%
aftererr = rejev+1;  % find target events following an unresponded target
aftererr(find(ismember(aftererr,selev)==0)) = [];  % remove events themselves
% not responded
whos selev rejev aftererr
fprintf(['Of %d "square" events following non-responded targets, ',...
'%d were themselves responded.\n\n'], length(rejev), length(aftererr));
```

-->-->

The EEGLAB 'urevent' structure allows researchers to explore the point of view that *no two events* during an experiment are actually equivalent, since each event occurs in a different *context*. Although hard to dispute, insights into brain function to be gleaned from this point of view are sacrificed when dynamics time-locked to all events of a certain type are simply *averaged* together. In particular, the EEGLAB erpimage.m and pop_erpimage.m functions allow users to visualize differences in potential time course or spectral amplitudes in epochs time-locked to events sorted by some feature of their context. A simple example is sorting stimulus-locked epoch by subsequent response time ('rt'), though this is only a simple and straightforward example. Using the eeg_context.m function, one might also sort stimulus event-locked epochs by the amount of time since the *previous* button press, or by the fraction of 'position'=1 events in the last minute, etc. Some future EEGLAB functions (and many user scripts) will no doubt exploit such possibilities in many ways.

IV.Appendix

The EEGLAB Tutorial is split into four parts, the last of which is the Appendices. In the Appendices, the user is introduced to more advanced and technical elements of EEGLAB such as input data formats and Matlab data structures. These topics are presented to the user to fully describe the architecture, etc. of the EEGLAB system. These sections are a reference only; more information can be found by contacting the appropriate SCCN personnel. As of 2/10/2012, there are thirteen Topics in the Appendix.

A01: Importing Continuous and Epoched Data

Importing continuous and epoched data

Supported Data Formats

File Format	File Extension	File type	Events	Channel Labels	EEGLAB	Biosig	File IO	Support
ANT EEProbe	.avr	_	_	_	_	_	_	Comments
ANT EEProbe	.cnt	_	_	_	y	y	y	Comments
ASCII	.txt	_	_	_	y	y	_	Comments
BCI2000	.bci2000	continuous	_	_	p	_	_	Comments
BCI2000	.gdf	continuous	_	_	p	_	_	Comments
Biologic	.eeg	_	_	_	_	_	_	Comments
Biosemi	.bdf	continuous	Channel	_	y	y	y	Comments
Brain Vision Analyzer	.mat	continuous & segmented	Embedded	_	y	y	n	Comments
Brain Vision Analyzer	.vhdr	_	file	_	y	y	n	Comments
BrainStorm	.vsm	_	_	_	_	_	_	Comments
Cogniscan	_	_	_	_	p	_	_	Comments
Compumedics Profusion (.raw	_	_	_	_	see comments	_	_	Comments
CTF/BrainStorm	.ctf	_	_	_	y	y	y	Comments
EGI/Netstation	.RAW	continuous & segmented	Channel	_	y	y	y	Comments
ERPSS	.raw	_	_	_	y	n	n	Comments
ERPSS	.rdf	_	_	_	y	n	n	Comments
European Data Format (16-bit)	.edf	_	Channel	_	y	y	n	Comments
INSTEP	.asc	_	_	_	y	n	n	Comments
Matlab Array	.mat	_	Channel	_	y	y	n	Comments
Micromed	_	_	_	_	p	_	_	Comments
Neuroimaging4D	.m4d	_	_	_	y	n	n	Comments
Neuromag	.fif	_	_	_	see comments	n	see comments	Comments
Neuroscan	.avg	_	_	_	_	_	_	Comments
Neuroscan	.CNT	_	Embedded (see comments)	_	y	y	y	Comments
Neuroscan	.eeg	continuous	_	_	y	y	y	Comments
Profusion	.slp	_	_	_	_	_	_	Comments
Snapmaster	.SMA	_	Channel	_	y	y	n	Comments
Tucker-Davis Technology	.tdt	_	_	_	p	_	_	Comments

Please upload test files at ftp://sccn.ucsd.edu/incoming to help us fill the table.

File Format Compatibility is noted by the following:

Supported Formats	y
Supported but requires MEX file	y
Unverified	_
Currently unsupported formats or not working	n
Functionality via third-party EEGLAB Plug-Ins	p

See also Electrode Data Formats and how to work with Other Formats

Importing a Matlab array

We first construct a 2-D Matlab array 'eegdata' containing simulated EEG data in which rows are channels and columns are data points:

```
eegdata = rand(32, 256*100);
% build a matrix of random test data (32 channels, 100 seconds at 256 Hz)
```

-->-->

To import these data, select the menu item File > Import data > from ascii/float file or Matlab array. At the Data file/array click on option Matlab variable from the list and set the name to eegdata. Set the sampling frequency to 256 Hz, press *OK*. Other dataset parameters will be automatically adjusted.

Note on importing data from other file formats: To import continuous data from a Matlab *.mat* file instead from a Matlab array, scroll the list of choices in the box above that shows Matlab *.mat* file.

Note: When reading a Matlab *.mat* file, EEGLAB assumes it contains only one Matlab variable. For reading a (32-bit) binary float-format data file, two choices are available: *float le* (little-endian) and *float be* (big-endian) The correct choice here depends on operating system. In case the bit ordering is unknown, try each of them. Note that the toolbox command line function shortread.m can also be used to read data from a (16-bit) short-integer file. The resulting Matlab array may then be imported into EEGLAB as shown above.

Now is a good time to add a *Description* about the dataset. A window will pop up containing a text box for this purpose. Enter as much information about the dataset as possible. The information you enter will be saved with the dataset for future reference by you or by others who may load the data. You can add to or revise these comments later by selecting menu item Edit > Dataset info. It is also possible to save the newly created data into a new dataset, either retaining or overwriting (in Matlab process memory) the old dataset. To also save the new dataset (with all its accompanying information fields) to disk, enter a filename in the lower edit field. Press *OK* to accept.

Then use menu item Plot > Channel data (scroll) to visualize the imported data.

Importing Biosemi .BDF files

Biosemi has extended the 16-bit European standard "EDF" (European Data Format) to their 24-bit data format, BDF (Biosemi Data Format). Select menu item File > Import data > From Biosemi .BDF file (calling function pop_readbdf.m). A window will pop up to ask for a file name.

Press *OPEN* to import a file.

Then a second window pops up, press *Ok*.

The third window to pop up ask for a new dataset name.

Press *OK*, then select menu item Plot > Channel data (scroll) to inspect the data. A sample .BDF file is available -- TEST_256.BDF (use *save link as* in your browser). (More sample files and reading functions are also available from the Biosemi ftp site). To extract event records from .BDF data, select menu item File > Import event info > From data channel as explained elsewhere in this tutorial.

Importing European data format .EDF files

To import data collected in the joint European 16-bit data format (EDF), select menu item File > Import data > From European data format .EDF file (calling function pop_readedf.m). A window will pop up to ask for a file name. Then select menu item Plot > EEG data (scroll) to inspect the data. A sample .EDF file is available -- TEST.EDF (use *save link as* in your browser). To extract event records from .EDF data, select menu item File > Import event info > From data channel as explained elsewhere in this tutorial.

Importing Netstation/EGI files

To read EGI (Electrical Geodesics Incorporated) .RAW data files, select menu item File > Import data > From Netstation binary simple file. Note that native EGI files in the current 2009 version of the EGI Netstation software are not EEGLAB compatible and need to be converted to

Netstation binary simple. They can also be converted to EDF and imported into EEGLAB but events may be corrupted in EDF files so the "Netstation binary simple" format is to be preferred. Once the EEGLAB menu is selected, the following window will appear

The function pop_readegi.m] should be able to read EGI Version 2 and Version 3 data files. The presence of events in the EGI format is recorded in an additional EGI data channel. Information from this channel is automatically imported into the EEGLAB event table and this channel is then removed by EEGLAB from the EEGLAB data. (If, for any reason this fails to occur, extract events using menu item File > Import event info > From data channel as explained elsewhere in this tutorial.)

Note that current Native Netstation files cannot be imported in EEGLAB/Matlab. Only "Netstation binary simple" files can be imported in EEGLAB (current Netstation files can easily be converted to Netstation binary simple from the Netstation software itself). EDF files can also be exported from Netstation and imported in EEGLAB, but the events are not stored in the last channel by Netstation. Converting data files to "Netstation binary simple" is the simpliest way to get EGI files under EEGLAB/Matlab.

When a subject is made of multiple segments (xxx001.raw; xxx002.raw; xxx003.raw etc...), select File > Import data > From multiple seg. Netstation files. A window will pop up to ask for the file name. Then select menu item Plot > EEG data (scroll) to inspect the imported data.

Finally, it is also possible to import Matlab files exported by Netstation using menu item (File > Import data > From Netstation epoch Matlab files) - the function can only import data exported using the "individual fields" option which is the default. These files contain data epochs but are missing a lot of information. It is therefore recommended to use the file "Netstation binary simple" format.

Importing Neuroscan .CNT continuous files

Note: In our experience, importing Neuroscan files is not an easy matter and no universal read function may exist. It seems that the EEGLAB function works properly in most cases, even for recent (2002) data files. For more about how to read Neuroscan files under Matlab, see a helper page(obsolete). A stand-alone Matlab function for reading this format is also available on the function index page as loadcnt.m. You may import the EEG test file TEST.CNT and use it to test the following tutorial steps.

Start by selecting the menu item File > Import data > From .CNT data file, which calls the pop_loadcnt.m function. The following window will appear:

Select the file to input (after changing the filter to the correct directory if necessary), and press *OPEN*. The following window will pop up:

The first input field concerns the structure of the .CNT file. If the imported data don't look like continuous EEG, try changing this number. Most often it should be 1 or 40, but other values may work. Now press *OK*. A window asking for a new set name will pop up, press OK to save the new data set.

Next, use menu item Plot > Channel data (scroll) to inspect the input data and menu item Edit > Event values to inspect the input event field latencies. EEGLAB (from version 4.31) automatically reads channel labels. If you call the menu Edit > Channel locations, EEGLAB will automatically find the location of most channels (based on channel labels).

We will now illustrate how to import additional epoch event information contained in an accompanying Neuroscan .DAT file into the EEGLAB dataset. Before importing the .DAT file, you must first epoch the loaded data (i.e., you must separate it into data epochs). To epoch the data, select Tools > Extract epochs

Simply press *OK* to epoch the data based on the events contained in the .CNT file (here using a time window extending from -1 s before to 2 s after events). The following window will appear:

Use this window to enter description and save the new dataset. For computers with limited memory (RAM), try overwriting the parent dataset (here, the continuous dataset we have just imported) by checking the *Overwrite parent* box in this window. One may also save the dataset to disk. Press *OK* when done. The following baseline removal window will pop up:

Simply press *OK* and continue.

Importing Neuroscan .EEG data epoch files

Select File > Import data > From .EEG data file, calling function pop_loadeeg.m. A first window will pop up to ask for the file name and a second window (below) will query for data format (16 or 32 bits) and for data range. See the pop_loadeeg.m function for more details (a 16-bit sample file is available here for testing). Data epochs have now been extracted from the EEG data. See also the section below about importing DAT file containing epoch information.

Then select menu item Plot > EEG data (scroll) to inspect the imported data. In this case, epoch events have also been imported from the file and can be visualized using menu item Edit > Event values.

Importing Snapmaster .SMA files

Select menu item File > Import data > From Snapmaster .SMA file (calling function pop_snapread.m). A window will pop up to ask for a file name. The following window will pop up to ask for relative gain, leave it on 400 and press *Ok*.

A window will pop up to ask for a file name. Then select menu item Plot > Channel data (scroll) to inspect the data and item Edit > Event values to inspect event latencies and types. A sample .SMA data file is available -- TEST.SMA (use *save link as* in your browser).

Importing ERPSS .RAW or .RDF data files

To read ERPSS files (Event-Related Potential Software System, JS Hansen, Event-Related Potential Laboratory, University of California San Diego, La Jolla, CA, 1993), select menu item File > Import data > From ERPSS .RAW or .RDF file (calling function pop_read_erpss.m). A window will pop up to ask for a file name. Then select menu item Plot > Channel data (scroll) to inspect the data and item Edit > Event values to inspect event latencies and types. A sample .RDF data file is available -- TEST.RDF (use *Save link as* in your browser). A header file for the ERPSS format is also available here.>

Importing Brain Vision Analyser Matlab files

To read Brain Vision Analyser (BVA) Matlab files, first export Matlab files from the BVA software. Then use menu item File > Import data > From Brain Vis. Anal. Matlab file (calling function pop_loadbva.m). A window will pop up asking for a file name. After reading the data file, select menu item Plot > Channel data (scroll) to inspect the data and item Edit > Event values to inspect event latencies and types. Channel locations will also be imported and can be visualized using menu item Plot > Channel locations > By name. A sample BVA Matlab data file is available -- TESTBVA.MAT (use *Save link as* in your browser). Note that you need a macro (and Matlab installed) to be able to export Matlab files from Brain Vision Analyser.

Importing sets of data averages into EEGLAB

EEGLAB was made to process and visualize single-trial data. Event-related potential (ERP) averages can also be processed and visualized, but they should not be imported directly. Note that in our current practice, we perform averaging over trials after applying ICA and not before (see Makeig et al. Science, 2002).

However, an increasing number of ERP researchers find it of interest to apply ICA to related *sets* of ERP grand averages (see Makeig et al, J. Neuroscience, 1999 and Makeig et al., Royal Society, 1999). To import data grand-average epochs into EEGLAB, stack the different conditions in a single array as explained below.

First, the data averages for different conditions must be imported to Matlab. For example, one may export these averages in text format and then use the standard Matlab function

```
>> load -ascii filename.txt
```

Note that to import ASCII files to Matlab, all column names and row names must be removed. For Neuroscan user, we have also programmed the function loadavg.m which can read most binary .AVG Neuroscan files.

Then you will need to concatenate data averages. For example, from a three-condition experiment, we may derive three ERP averages with a sampling rate of 1000 Hz, covering from -100 to 600 ms with respect to stimulus onsets (Note that we always process each subject individually and then compare the results across subjects at the end of the analysis).

For instance typing *>> whos* under Matlab might return:

```
Name                    Size            Bytes               Class
avgcond1                31x600          14880               double array
avgcond2                31x600          14880               double array
avgcond3                31x600          14880               double array

Grand total is 55800 elements using 446400 bytes
```

-->-->

Note: If necessary, transpose the arrays (so rows=channels, colunmns=data samples, i.e. chan*samp) like this (not necessary for this example)

```
>> avgcond1 = avgcond1';
```

Then concatenate the arrays

```
>> allcond = [ avgcond1 avgcond2 avgcond3 ];
```

Finaly, you will need to import concatenated data averages into EEGLAB. First start EEGLAB:

```
>> eeglab
```

Select menu item File > Importing data > From ascii/float file or Matlab array

Enter the information as indicated above. The following window pops up and allows you to add comments and/or save the new dataset immediately. Press *OK* to create a new dataset.

Select Plot > Channel ERPs> in rect. array and set the last option, 'Plot single trials', to *YES* to visualize the three condition ERPs.

It is possible to process the three average-ERP epochs as if they were single-trial epochs (although in this case some EEGLAB functions may not be meaningful). See the Single subject data analysis tutorial for more information.

Importing data in other data formats

The Biosig toolbox (biosig.sf.net) contains links to functions to read other EEG data formats in Matlab. You may download the Biosig plugin for EEGLAB (see the EEGLAB plugin page).

For other non-suported data format, the home page of Alois Schlolg contains links to other Matlab reading functions. We are also willing to add other data importing functions to EEGLAB, so please send us a sample raw data file together with a Matlab function to read it and a README file describing the origin of the data and its format. We will attempt to include such functions in future releases of EEGLAB. Contributing authors will retain all rights to the functions they submit, and the authors' name(s) will be displayed in the function header. See our page on how to contribute to EEGLAB.

The EEGLAB discussion list archive also contains messages from users for importing specific data formats. You may search the list archive (and the rest of the EEGLAB web site) archive using Google from the bottom of the main EEGLAB web page.

A02: Importing Event Epoch Info

Importing event and epoch information

Supported Data Formats

File Format	File Extension	File type	Events	Channel Labels	EEGLAB	Biosig	File IO	Support
Presentation	.LOG	_	_	_	y	y	_	Comments
Neuroscan	.DAT	_	_	_	y	y	_	Comments
Eprime	On demand(1)	_	_	_	p	_	_	Comments
Custom text	_	_	_	_	y	_	_	Comments

(1) Email us at eeglab@sccn.ucsd.edu. We have programmed this function but have not had time to include it.

Please upload test files at ftp://sccn.ucsd.edu/incoming to help us fill the table.

File Format Compatibility is noted by the following:

Supported Formats	y
Supported but requires MEX file	y
Unverified	_
Currently unsupported formats or not working	n
Functionality via third-party EEGLAB Plug-Ins	p

EEGLAB counts records of the time and nature of experimental events to analyze the EEG data. This section details how to load in event information which coded in one of the data channels, stored in a Matlab array or separate ascii file. When event information is read in, (as of v4.2) EEGLAB copies the resulting EEG.event structure to a back-up (*ur*) copy, EEG.urevent and creates links from each event to the corresponding urevent. This allows the user to select events based on the previous (or future) event *context*, even *after* data containing some events has been rejected from the data (see the event tutorial for more information).

Importing events from a data channel

Often, information about experimental events are recorded onto one of the rows (channels) of the EEG data matrix. Once more we create simulated data to illustrate how to import events from a data channel. Assuming an EEG dataset with 33 rows (channels), out of which the first 32 are channels and the last (33) is an event channel with values 1 (stimulus onset), 2 (subject response), and 0 (other), Matlab code for generating such data follows (to test, copy and paste the code to the Matlab command line):

```
>> eegdata = rand(32, 256*100); % 32 channels of random activity (100 s sampled at 256 Hz).
>> eegdata(33,[10:256:256*100]) = 1; % simulating a stimulus onset every second
>> eegdata(33,[100:256:256*100]+round(rand*128)) = 2; % simulating reaction times about 500 ms after stimulus onsets
```

After copying the code above to Matlab and importing the array *eegdata* into EEGLAB as a test dataset (see Matlab arrays in this section), select menu item File > Import event info > from data channel to call function pop_chanevent.m .

Enter *33* as the event channel and set the edge-extract type to *up (leading)* (Note: place the mouse over the text *Transitions to extract* to see contextual help).

Press *OK*. Now, the event information will have been imported into the test EEGLAB dataset. At the same time, channel 33 will have been deleted from the test data. Select menu item Edit > Event values to inspect the imported event types and latencies.

Importing events from a Matlab array or text file

Using the random EEG dataset created above, we import event information stored in an ASCII text file, tutorial_eventtable.txt. This text file is composed of three columns, the first containing the latency of the event (in seconds), the second the type of the event, and the third a parameter describing the event (for example, the position of the stimulus). For example, the top lines of such a file might be:

Latency	Type	Position
1	target	1
2.3047	response	1
3	target	2
4.7707	response	2
5	target	1
6.5979	response	1
...		

Select menu item File > Import event info > Import Matlab array or ASCII file

Browse for the tutorial text file, set the number of header lines to 1 (for the first line of the file, which gives the column field names) and set the input fields (i.e., the names associated with the columns in the array) to *latency type position*. If these field names are quoted or separated by commas, these extra characters are ignored. (NOTE: It is NECESSARY to use the names *latency* and *type* for two of the fields. These two field names are used by EEGLAB to extract, sort and display events. These fields must be lowercase since Matlab is case sensitive.) In this interactive window the input *Event indices* and checkbox *Append events?* can be used to insert new events or replace a subset of events with new events (for instance for large EEG files which may have several event files).

Important note about aligning events

An essential input above is *Align event latencies to data events* which aligns the first event latency to the existing event latency and checks latency consistency. A value of *NaN* (Matlab for not-a-number) indicates that this option is ignored (as in the example above). However, for most EEG data, the EEG is recorded with basic events stored in an event channel (see Import events from a data channel above) for instance. Detailed event information is recorded separately in a text file: as a consequence the events recorded in the text file have to be aligned with the events recorded in the EEG.

To do so, set the input for *Align event latencies to data events* to 0 if the first event in the text file correspond to the first event recorded in the EEG (i.e., if the offset between the two is 0). Setting this value to 1 indicates that event 1 in the event text file corresponds to event number 2 in the EEG data. Here, negative values can also be used to indicate that events in text file start before those recorded in the EEG).

When aligning events, as shown in the following section, the function displays the latencies of the two event types, so the user can check that they are aligned based on his knowledge of the experiment (for instance, there may be more events in the text file than recorded in the EEG).

The last checkbox allow to automatically adjust the sampling rate of the new events so they best align with the closest old event. This may take into account small differences in sampling rates that could lead to big differences by the end of the experiment (e.g., a 0.01% clock difference during would lead to a 360-ms difference after one hour if not corrected).

Importing events from a Presentation file

EEGLAB can also import events saved using the Presentation software. Sample files are available for download here: TEST.SMA(a SnapMaster .SMA file) and TEST.LOG (a Presentation event log file).

To test the use of these sample files, first import the .SMA data file using menu item File > Import data > From .SMA data file. Then select File > Import event info > from Presentation LOG. file to import events from the Presentation log file as shown below

Then the following window pops-up

Scroll file fields to select which field (i.e., file column) contain the event type and which column contain the event latency. The default is fine with this specific file, so simply press *OK*. Matlab then returns:

```
Replacing field 'Event Type' by 'type' for EEGLAB compatibility
Replacing field 'Time' by 'latency' for EEGLAB compatibility
Renaming second 'Uncertainty' field
Reading file (lines): 6
Check alignment between pre-existing (old) and loaded event latencies:
Old event latencies (10 first): 10789 21315 31375 41902 51962 62489 ?
New event latencies (10 first): 10789 21315 31376 41902 51963 62489 ?
Best sampling rate ratio found is 0.9999895. Below latencies after adjustment
Old event latencies (10 first): 10789 21315 31376 41902 51963 62488 ?
New event latencies (10 first): 10789 21315 31375 41902 51962 62489 ?
Pop_importevent warning: 0/6 have no latency and were removed
eeg_checkset: value format of event field 'Duration' made uniform
eeg_checkset: value format of event field 'Uncertainty2' made uniform
eeg_checkset note: creating the original event table (EEG.urevent)
Done.
```

The function aligns the first event latency recorded in the Presentation file to the first event latency recorded in the EEG in the SnapMaster file. Check that the events recorded in the SnapMaster file have the same latencies as the ones recorded in the .LOG presentation file. The function then computes the best sampling rate ratio: this may account for small differences in sampling rate that could lead to big differences at the end of the experiment (e.g., 0.01% clock difference during half an hour would lead to a 360-ms difference after one hour if not corrected). Note that if the events are shifted (with respect to events from the binary EEG file), it is always possible to suppress events manually or to import the presentation file as a text file, as described in the previous section. Note that some Presentation files that contain comments at the end of the file may not be supported. If you are not able to import a Presentation file, try removing any comments from the end of the file. If it still does not work, try importing the Presentation file as a text file as described in the previous section.

Importing E-Prime information files

The E-prime format is highly configurable, so you may use the ASCII importer to import data files. Usually, we would use EEGLAB's Import data -> From ASCII/float file or Matlab array functionality to import the data file and configure the interface with the name of the E-Prime columns. It might be necessary in some cases to export the E-Prime to a tab-delimited file first (for example in a spreadsheet application) and edit some of the columns information that might not be read correctly under Matlab. Send us your E-Prime files (at eeglab at sccn.ucsd.edu) so we may tailor the ASCII import menu for E-Prime files.

Importing Neuroscan .DAT information files

To import the .DAT file linked to a previously loaded .CNT file, select menu item File > Import epoch info > From Neuroscan .DAT info file (calling function pop_loaddat.m). The sample .DAT file associated with the continuous .CNT file we used above is available for download -- TEST.DAT (both the .DAT file and the .CNT contains 100 epoch events).
Select the file to import in the resulting window. A second window will then appear:

In .DAT files, there must be a reaction time (in milliseconds) for each epoch. However, depending on experiment design there may be no reaction time in a given epoch. Then one has to use a code value for reaction time latencies in these epochs. For instance, you might decide that a value of *1000* (ms) would indicate that the subject did not respond. (If all the epochs of the experiment already have a reaction time, do not enter anything here.)

Importing epoch info Matlab array or text file into EEGLAB

Importing epoch information means that data epochs have already been extracted from the continuous EEG data, and that the Matlab array or text epoch information file has one entry per epoch. To illustrate how to import such a file or array, we will once more create some simulated EEG data.

```
>>eegdata = rand(32, 256, 10); % 32 channels, 256 time points per epoch, 10 epochs
```

Select menu item File > Import data > From ascii/float data file or Matlab array.

Press *OK* in this window (Note: If you select a data importing format different from *Matlab variable*, be sure to click on it to actually select the option.) Then, a second window will pop up.

Press *OK* in this window (see how to import at the beginning of this page for more information). Note that the Matlab array, being 3-D, is automatically imported as data epochs: the first dimension is interpreted as data channels, the second as data points and the third as data epochs or trials (e.g., our sample data matrix above contains 10 epochs). Let us imagine that our simulated EEG data came from a simple stimulus/response task, with subject responses being either 'correct' or 'wrong' and response latencies recorded in milliseconds. Then the epoch event file might look something like this:

Epoch	Response	Response_latency
1	Correct	502
2	Correct	477
3	Correct	553
4	Correct	612
5	Wrong	430
6	Correct	525

7	Correct	498
8	Correct	601
9	Correct	398
10	Correct	573

This file tutorial_epoch.txtcan be downloaded or copied from the array above in a text file. Then select menu item File > Import epoch info > from Matlab array or ascii file, bringing up the following window:

Above, browse for the *tutorial_epoch.txt* file, set the input fields to *epoch response rt* (where rt is an acronym for 'reaction time'). The only one of these fields that contains latency information is *rt*, so it is entered to the as input to the *Field name(s) containing latencies* query box. This file (see above) has 1 header line, as we need to specify in the *Number of file header lines to ignore* box. Finally the reaction times are recorded in milliseconds, which we indicate as *1E-3* (i.e., one-thousandth of a second). Note that the last entry, *Remove old epoch ...*, allows the user to import other information later if it is unset. Press *OK* when done. Now select Edit > Event fields.

It is a good idea to click each of the *Field description* buttons above to add detailed descriptions of the meaning of each of the fields and the coding of the field values (for example: 1 = correct, 2 = wrong, etc.). This information is stored along with the event field values when the dataset is saved (very useful for later analysis by you or others!).

Above, there are now five fields, not three as for the file data. Also note that the field *rt* is not present. All of this is normal because EEGLAB does not store epoch information as such, but converts it into fields in its events structure. Though this seems a bit complicated in the beginning, it helps avoid redundancy and inconsistencies between epoch and event information. It also means that new data epochs can be re-extracted from data epochs based on the stored events. Now select menu item Edit > Event values to inspect what happened to the reaction time information (use the arrow to move to the second event):

As shown above, when epoch information was imported, events with type named *rt* were created and assigned a latency. If we had had several columns containing latency information, the function would have created several types.
Programming note: For convenience, standard epoch information is available from the command line in the variable *EEG.epoch*. Also, event information available in *EEG.event* can be used for script or command line data processing. See the script writing tutorial for more information.

Exporting event information

Event information can also be exported to a .csv file (a plain text file containing comma-separated values; a tab is used as delimiter). This file can be opened with any text editor, OpenOffice Calc, or Microsoft Excel, for example. To export all events of the currently loaded EEG file, select File > Export > Events to text file. A dialog window pops up asking for the name and location of the .csv file. The first row of the file contains the names of the event fields. Note that there is additional column "number", which is not an event field.

If you want more control over export options, you should use the command line version of this tool. In the MATLAB command window, enter the following code to achieve the same result as before with the GUI:

```
events = eeg_eventtable(EEG, 'exportFile', 'test.csv');
```

In addition, a table with all events is displayed in the command window by default. The additional index column can be disabled, and the time unit of the latency and duration event fields can be set to seconds or samples (default). The event structure is stored in a cell array for convenient access. Type `help eeg_eventtable` to get more help on this command.

A03: Importing Channel Locations

Supported Electrode Data Formats

File Format	File Extension	Coordinate System	Import Mode	EEGLAB	Biosig	File IO	Support
BESA	.elp	Spherical	_	y	_	_	Comments
BESA/EGI	.sfp	Cartesian	native	y	_	_	Comments
EEGLAB	.loc, .locs, .eloc	Polar	_	y	_	_	Comments
EEGLAB (ASCII)	.ced	Cartesian	default (eeglab)	y	_	_	Comments
EEGLAB (Matlab)	.xyz	Cartesian	default (eeglab)	y	_	_	Comments
EETrack	.elc	Cartesian	_	y	_	_	Comments
EMSE	.elp	_	_	_	_	_	Comments
FreeSurfer	.tri	_	_	_	_	_	Comments
Matlab	.sph	Spherical	_	y	_	_	Comments
Neuroscan	.asc	Cartesian or Polar	_	y	_	_	Comments
Neruoscan	.dat	Cartesian or Polar	_	y	_	_	Comments
Neuroscan	.tri	_	_	_	_	_	Comments
Neuroscan	.3dd	_	_	_	_	_	Comments
Polhemus	.elp	Cartesian	default (eeglab)	y	_	_	Comments

Please upload test files at ftp://sccn.ucsd.edu/incoming or Channel Location Files to help us build the table.

File Format Compatibility is noted by the following:

Supported Formats	y
Supported but requires MEX file	y
Unverified	_
Currently unsupported formats or not working	n
Functionality via third-party EEGLAB Plug-Ins	p

For more information, please refer to readlocs.m or go to Channel Location Files

Importing channel location

To plot EEG scalp maps in either 2-D or 3-D format, or to estimate source locations for data components, an EEGLAB dataset must contain information about the locations of the recording electrodes.

To load or edit channel location information contained in a dataset (with or without channel location), select Edit > Channel locations. If you do not have a dataset loaded, you can test this section by typing on the Matlab command line:

```
>>pop_chanedit([]);
```

If you imported a binary data file in Neuroscan or Biosemi formats, channel labels will be present in the dataset (as of EEGLAB v4.31). When you call the channel editing window, a dialog box will appear asking you if you want to use standard channel locations corresponding to the imported channel labels (for example. Fz) from an channel locations file using an extended International 10-20 System. Otherwise, you must load a channel locations file manually.

To load a channel locations file, press the *Read Locations* button and select the sample channel locations file *eeglab_chan32.locs* (this file is located in the *sample_data* sub-directory of the EEGLAB distribution).

In the next pop-up window, simply press *OK*. If you do not specify the file format, the pop_chanedit.m function will attempt to use the filename extension to assess its format. Press button *Read help* in the main channel graphic interface window to view the supported formats.

Now the loaded channel labels and polar coordinates are displayed in the pop_chanedit.m window.

To plot scalp maps only inside the head cartoon, enter 0.5 at the *Plot radius* edit box. In this case, the two eye electrodes will not be displayed nor considered computing interpolated 2-D scalp maps. If you do not see enough of the recorded field, set this dialogue box to 1.0 to interpolate and show scalp maps including all channels, with parts of the head below the (0.5) head equator shown in a 'skirt' or 'halo' outside the cartoon head boundary. (More precise separate controls of which channel locations to interpolate and plot are available from the command line in the topographic map plotting function topoplot.m.

In the window below, you may scroll through the channel field values 1-by-1 using the < and > buttons, or in steps of 10 using << and >>.

The *Set channel type* button allows you to enter a *channel type* associated with the channel (for example, 'EEG', 'MEG', 'EMG', 'ECG', 'Events', etc.). Although channel types are not yet (v5.0b) widely used by other EEGLAB functions, they will be used in the near future to restrict plotting and computation to a desired subset of channel types, allowing easier analysis of multi-modal datasets. It is therefore well worth the effort to add channel types to your data. It is important to press *OK* in the channel editing window above to actually import the

channel locations!. Note that in the main EEGLAB window, the *channel location* flag now shows *yes*.</blockquote>

Following are examples of ascii channel locations data in EEGLAB-supported formats.

Supported channel locations file formats.

- Four channels from a polar coordinates file (with filename extension *.loc*, Do not include the (light blue) header line:

#	Deg.	Radius	Label
1	-18	,352	Fp1
2	18	.352	Fp2
3	-90	,181	C3
4	90	,181	C4

- The same locations, from a spherical coordinates file (estension, *.sph*):

#	Azimut	Horiz.	Label
1	-63.36	-72	Fp1
2	63.36	72	Fp2
3	32.58	0	C3
4	32.58	0	C4

- The same locations from a Cartesian coordinates file (extension, *.xyz*):

#	X	Y	Z	Label
1	-0.8355	-0.2192	-0.5039	Fp1
2	-0.8355	0.2192	0.5039	Fp2
3	0.3956	0	-0.9184	C3
4	0.3956	0	0.9184	C4

Note that in all the above examples, the first header line must not be present.

Other supported channel locations file formats

- Polhemus (*.elp*) files
- Neuroscan spherical coordinates (*.asc, .dat*)
- Besa spherical coordinates (*.elp, .sfp*)
- EETrak sofware files (*.elc*)
- EEGLAB channel locations files saved by the pop_chanedit.m function (*.loc, .xyz, .sph, .ced*)

Note that pop_chanedit.m and readlocs.m can also be called from the command line to convert between location formats. For more details, see the readlocs.m help message.

Exploratory Step: Viewing Channel Locations.

Reopen Edit > Channel locations if you closed it. To visualize the 2-D locations of the channels, press *Plot 2-D* above the *Read Locations* button. Else, at any time during an EEGLAB session you may refer to a plot showing the channel locations by selecting Plot > Channel location > By name. Either command pops up a window like that below. Note: In this plot, click on any channel label to see its channel number.

All channels of this montage are visible in the 2-D view above, since none of these channels are located below the head center and equator line. If the montage contained channel locations whose polar coordinate radius values were larger than the default value (e.g., *0.5*) you entered in the pop_chanedit.m window, those locations would not appear in the top-down 2-D view, and the interpolated scalp map would end at the cartoon head boundary. The 2-D plotting routine topoplot.m gives the user full flexibility in choosing whether to show or interpolate data for inferior head channels; topoplot.m is used by all EEGLAB functions that plot 2-D scalp maps.
To visualize the channel locations in 3-D, press *Plot 3-D (xyz)*. The window below will appear. The plotting box can be rotated in 3-D using the mouse:

You may change channel locations manually using the edit box provided for each channel's coordinates. However, after each change, you must update the other coordinate formats. For instance, if you update the polar coordinates for one channel, then press the *polar->sph. & xyz* button on the right of the pop_chanedit.m window (see above) to convert these values to other coordinate formats.

Retrieving standardized channel locations

This section does not use the tutorial dataset. It intent is to provide guidelines for automatically finding channel locations when channel names are known. For instance, when importing a Neuroscan, an EGI, or a Biosemi channel locations file, channel names are often stored in the file header. EEGLAB will automatically read these channel labels. When you then call the channel editing window, the function will look up channel locations in a database of 385 defined channel labels, the file "Standard-10-5-Cap385.sfp" in the "function/resources" sub-folder of the EEGLAB distribution. You may add additional standard channel locations to this file if you wish. Channel locations in this file have been optimized for dipole modeling by Robert Oostenveld.

For example, download the sample file, TEST.CNT, then call up the channel editing window. As an alternate example not requiring a data download, we will build a channel structure using channel labels only, then will call the channel editing window. In the Matlab command window, type:

```
>> chanlocs = struct('labels', { 'cz' 'c3' 'c4' 'pz' 'p3' 'p4' 'fz' 'f3' 'f4'});
>> pop_chanedit( chanlocs );
```

The following window will pop-up:

Press *Yes*. The function will automatically look up channel locations for these known channel labels. The following channel editing window will then pop up.

Press *Plot 2-D* to plot the channel locations. Close the channel editing window (using *Cancel* to discard the entered locations), then proceed to the next section.

Importing measured 3-D channel locations information

This section does not use the tutorial dataset. Its intent is to provide guidelines for importing channel locations measured in Cartesian coordinates using 3-D tracking devices (such as Polhemus). Use the EEGLAB menu Edit > Channel location or type the following command on the Matlab command line

```
>> pop_chanedit([]);
```

An empty channel editing window will appear:

Press the *Read locations* button and select the file *scanned72.dat* from the *sample_data* subfolder of the EEGLAB distribution. This is a channel locations file measured with the Polhemus system using Neuroscan software (kindly supplied by Zoltan Mari). Use autodetect ([ ]) for the file format. When the file has been imported, press the *Plot 2-D* button. The following plot will pop up.

As you can see, the measured 3-D channel coordinates may not be accurately distributed on the 2-D head model. This is because the measured values have not been shifted to the head center. To fix this problem, you must first find the head sphere center that best fits the imported 3-D electrode locations. To do so, press the *Opt. head center* (optimize head center). The following window will pop up:

Possibly, some of the channels should not be included in the head center optimization, if they are not on the head and/or do not have recorded locations. Enter electrodes indices to use (here, 1:3 33 35 64:72) in the edit window. You may also press the *Browse* button above to select channels that are not on the head. When you press *OK* in the browser window, the channel indices will be copied, as shown in the window above. Then press *Ok*. After the optimization has finished, press the *Plot 2-D* button once more.

In the view above, some channel locations are still incorrect. For instance, you may expect channel "Cz" to be at the vertex (plot center). To adjust this, press the *Rotate axis* button. The following window will pop up:

Simply press *OK* to align channel 'Cz' to the vertex (by default). Then press the *Plot 2-D* button once more to again plot the scalp map.

You may now close the channel editing window.

This section has illustrated operations you may want to perform to adapt measured 3-D channel locations for use in EEGLAB.

A04: Exporting Data

Supported Data Formats

Data Type	Export Type	File Extension	EEGLAB	BIOSIG	Support
EEG Data	ASCII Text	.txt	y	_	Comments
EEG Data	European Data Format (16-bit)	.edf	y	_	Comments
EEG Data	Biosemi	.bdf	y	_	Comments
EEG Data	BCI2000	.gdf	y	_	Comments
EEG Data	Brain Vision Analyzer	.dat, .vhdr, .vmrk	y	_	Comments
ICA Activity	ASCII Text	.txt	y	_	Comments
Weight Matrix	ASCII Text	.txt	y	_	Comments
Inverse Weight Matrix	ASCII Text	.txt	y	_	Comments
Binary	Neuroscan	.cnt	y	_	Comments

- Find More information regarding possible supported data formats in the BIOSIG documentation, here.

Please leave comments to help us fill and expand this table.

File Format Compatibility is noted by the following:

Supported Formats	y
Supported but requires MEX file	y
Unverified	_
Currently unsupported formats or not working	n
Functionality via third-party EEGLAB Plug-Ins	p

Exporting data and ICA matrices

Exporting data to an ASCII text file

EEGLAB datasets can be exported as ASCII files using menu item File > Exports> Data and ICA activity to text file. Enter a file name (*mydata.txt*, for instance). Check the second checkbox to export the average ERP instead of the data epochs. By default, the electrode labels are saved for each row (4th check box) and the time values are saved for each column (5th checkbox). Time units can be specified in the edit box closest to the time values checkbox. Finally, check the third checkbox to transpose the matrix before saving.

The file written to disk may look like this:

```
FPz             EOG1       F3          Fz         F4         EOG2       FC5        FC1           ...
-1000.0000      -1.1091    2.0509      0.1600     -0.1632    -0.4848    -1.3799    -0.0254       -0.4788 ...
```

```
-992.1880       0.6599     1.7894      0.3616     0.6354    0.8656    -2.9291    -0.0486       -0.4564 ...
-984.3761       1.8912     1.3653      -0.6887   -0.0437   0.2176     -0.9767    -0.6973       -1.5856 ...
-976.5641       0.5129     -0.5399     -1.4076   -1.2616   -0.8667    -3.5189    -1.5411       -1.9671 ...
-968.7521       -0.0322    -0.4172     -0.9411   -0.6027   -0.9955    -2.3535    -1.6068       -1.0640 ...
-960.9402       0.1491     0.0898      -0.0828   0.3378    0.0312     -2.4982    -0.9694       -0.0787 ...
-953.1282       -1.9682    -1.5161     -1.2022   -0.8192   -1.1344    -3.3198    -1.6298       -0.9119 ...
-945.3162       -3.7540    -2.1106     -2.6597   -2.4203   -2.2365    -3.5267    -1.9910       -2.7470 ...
-937.5043       -2.4367    -0.1690     -0.9962   -1.7021   -2.8847    -2.1883    -0.2790       -1.5198 ...
-929.6923       -2.8487    -0.3114     -1.6495   -2.6636   -4.0906    -1.7708    -1.2317       -2.3702 ...
-921.8803       -2.8535    0.1097      -1.5272   -2.0674   -3.8161    -3.1058    -0.8083       -1.5088 ...
-914.0684       -3.9531    -0.4527     -1.8168   -2.2164   -3.4805    -2.1490    -1.0269       -1.3791 ...
-906.2564       -3.9945    -0.1054     -1.8921   -2.8029   -3.5642    -3.4692    -1.1435       -2.2091 ...
-898.4444       -4.4166    -0.8894     -3.3775   -3.8089   -3.8068    -1.7808    2.5074        -3.5267 ...
-890.6325       -3.0948    0.5812      -2.5386   -1.7772   -1.8601    -2.8900    -2.0421       -2.0238 ...
-882.8205       -3.1907    0.7619      -3.6440   -2.1976   -1.4996    -0.6483    -3.4281       -2.7645 ...
-875.0085       -1.7072    2.5182      -3.2136   -2.4219   -1.3372    -1.5834    -2.9586       -2.8805 ...
-867.1966       -1.8022    1.7044      -2.6813   -3.2165   -2.7036    0.0279     -2.5038       -3.4211 ...
-859.3846       -3.1016    -0.1176     -3.6396   -4.3637   -3.9712    -3.5499    -3.4217       -4.5840 ...
-851.5726       -1.7027    0.7413      -3.3635   -3.8541   -3.5940    -1.3157    -2.9060       -3.8355 ...
-843.7607       -0.2382    0.5779      -1.9576   -2.6630   -1.8187    -1.1834    -1.4307       -2.4980 ...
-835.9487       0.7006      0.4125     -0.4827   -1.7712    -2.0397    0.2534    0.2594        -1.2367 ...
-828.1367       -0.2056    -0.3509     0.4829    -0.6850   -1.1222    0.0394     1.4929        0.7069 ...
-820.3248       0.3797     -0.3791     0.9267    0.2139    -0.6116    -0.7612    1.3307        1.5108 ...
-812.5128       -0.8168    -1.4683     -0.3252   -0.8263   -1.5868    -0.7416    -0.2708       -0.1987 ...
-804.7009       -0.7432    -0.3581     -0.9168   -0.8350   -1.7733    -0.4928    -0.7747       -0.6256 ...
...
```

The first column contains the time axis and the other the data for each electrode. This file might, for example, be imported into SPSS or BMDP.

Exporting ICA weights and inverse weight matrices

Use menu item File > Export> Weight matrix to text file to export the ICA unmixing matrix (weights*sphere). Simply enter a file name in the pop-up window and press *Save*.

The text file on disk then contains the weight matrix. It may be re-imported into another EEGLAB dataset using menu item Edit > Dataset info. As shown below, enter the filename in the *ICA weight array* edit box. Leave the sphere edit box empty, or empty it if it is not empty. See the ICA decomposition tutorial for more details on sphere and weight matrices.

Edit dataset info -- pop_editset()
Dataset name (optional):
About
d from "ee114 continuous"
(EEG.setname)
Data file or array
(EEG.data)
Channels in data:
32
(EEG.nbchan)
Time points per epoch:
385
(EEG.pnts)
Data sampling rate (Hz):
128.0087
(EEG.srate)
Epoch start time (sec):
-1
(EEG.xmin)
Channel position file or array:
Help
EEG.chanlocs
Browse
ICA weight array or text file (if any):
myweigthmatrix.txt
Browse
ICA sphere array or text file (if any):
Browse
Data reference
common
(EEG.ref)
Cancel
Help
Ok

A05: Data Structures

EEGLAB Data Structures

This section is intended for users who wish to use EEGLAB and its functions in Matlab scripts. We have tried to make EEG structures as simple and as transparent as possible so that advanced users can use them to efficiently process their data.

EEG and ALLEEG

EEGLAB variable *EEG* is a Matlab structure that contains all the information about the current EEGLAB dataset. For instance, following the single subject tutorial documentation until data epochs have been extracted and the ICA decomposition computed, and then typing *>>EEG* will produce the following command line output:

```
>> EEG

        EEG =
              'Epochs from "ee114 continuous"'
              'ee114epochs.set'
              '/home/arno/ee114/'
              384 pnts:
                 nbchan: 32
              80 trials:
              128 srate:
               1 xmin:                  -
              1.9922 xmax:
              [32x384x80 double]
              [32x32 icawinv:]
              [32x32 icasphere:]
              [32x32 icaweights:]
              [] icaact:
              [1x157 struct]
              [1x80 epoch:]
              [1x32 chanlocs:]
              [8x150 chaninfo:]
              'no ref:'
              [] rt:
                 eventdescription: {1x5 cell}
                 epochdescription:
              [] specdata:
              [] specicaact:
              [1x1 reject:]
              [1x1 stats:]
              [] splinefile:
              'common'
              [7x13 char]
              [1x154 event:]
              [1x384 double]
```

-->-->

Above, we have italized several important fields. See the help message of the eeg_checkset.m function (which checks the consistency of EEGLAB datasets) for the meaning of all the fields.

EEGLAB variable *ALLEEG* is a Matlab array that holds all the datasets in the current EEGLAB/Matlab workspace. In fact *ALLEEG* is a structure array of *EEG* datasets (described above). If, in the current EEGLAB session you have two datasets loaded, typing *>> ALLEEG* on the Matlab command line returns:

```
ALLEEG =
    struct 1x2 array with fields:
                setname
                filename
                filepath
                pnts
                nbchan
                trials
                srate
                xmin
                xmax
                data
                icawinv
                icasphere
                icaweights
                icaact
                event
```

```
                epoch
                chanlocs
                comments
                averef
                rt
                eventdescription
                epochdescription
                specdata
                specicaact
                reject
                stats
                splinefile
                ref
                history
                urevent
                times
```

-->-->

Typing *>> ALLEEG(1)* returns the structure of the first dataset in ALLEEG, and typing *>> ALLEEG(2)* returns the structure of the second dataset. See the Script Tutorial for more information on manipulating these structures.

Most fields of the *EEG* structure contain single values (as detailed in eeg_checkset.m). However some important fields of the *EEG* structure contain sub-structures. We will describe briefly three of those below: *EEG.chanlocs*, *EEG.event*, and *EEG.epoch*.

EEG.chanlocs

This EEG-structure field stores information about the EEG channel locations and channel names. For example, loading the tutorial dataset and typing *>> EEG.chanlocs* returns

```
>> EEG.chanlocs

ans =

     struct 1x32 array with fields:
                theta
                radius
                labels
                sph_theta
                sph_phi
                sph_radius
                X
                Y
                Z
```

-->-->

Here,*EEG.chanlocs* is a structure array of length 32 (one record for each of the 32 channels in this dataset). Typing *>>EEG.chanlocs* returns:

```
>> EEG.chanlocs(1)

ans =
             0  theta:
             0.4600  radius:
             'FPz'  labels:
             0  sph_theta:
             7.2000  sph_phi:
             1  sph_radius:
             0.9021  X:
             0  Y:
             0.1253  Z:
```

-->-->

These values store the channel location coordinates and label of the first channel ('FPz'). You may use the pop_chanedit.m funciton or menu item Edit > Channel locations to edit or recompute the channel location information. The value of any EEG structure field may also be modified manually from the Matlab command line. See also the EEGLAB channel location file database page and the main tutorial section regarding 'Using channel locations'.

EEG.event

The EEG structure field contains records of the experimental events that occurred while the data was being recorded, plus possible additional user-defined events. Loading the tutorial dataset and typing

```
>> EEG.eventreturns

ans =
      struct array with fields:
type
                        position
                        latency
                        urevent
```

-->-->

In general, fields *type*, *latency*, and *urevent* are always present in the event structure. *type* contains the event type. *latency* contains the event latency in data point unit. *urevent* contains the index of the event in the original (= "ur") urevent table (see below). Other fields like *position* are user defined and are specific to the experiment. The user may also define a field called *duration* (recognized by EEGLAB) for defining the duration of the event (if portions of the data have been deleted, the field *duration* is added automatically to store the duration of the break (i.e. boundary) event). If epochs have been extracted from the dataset, another field, *epoch*, is added to store the index of the data epoch(s) the event belongs to. To learn more about the EEGLAB event structure, see the event processing tutorial.

There is also a separate "ur" (German for "original") event structure, *EEG.urevent* (in EEGLAB v4.2 and above), which holds all the event information that was originally loaded into the dataset plus events that were manually added by the user. When continuous data is first loaded, the content of this structure is identical to contents of the *EEG.event* structure (minus the *urevent* pointer field of *EEG.event*). However, as users remove events from the dataset through artifact rejection or extract epochs from the data, some of the original (ur) events continue to exist only in the urevent structure.

The *urevent* field in the EEG.event structure above contains the index of the same event in the *EEG.urevent* structure array. For example: If a portion of the data containing the second urevent were removed from a dataset during artifact rejection, the second event would <u>not</u> remain in the *EEG.event* structure -- but would still remain in the *EEG.urevent* structure. Now, the second event left in the data might be the original third event, and so will be linked to the third *EEG.urevent*, i.e. checking

```
>> EEG.event(2).urevent

ans =
3
```

-->-->

<u>WARNING:</u> Datasets created under EEGLAB 4.1 and loaded into 4.2 had an *EEG.urevent* structure created automatically. If some data containing events had been rejected BEFORE this time, then the urevent structure information IS INCOMPLETE (i.e. to some degree wrong!). Most new datasets created under 4.2 had the urevent structure saved correctly when the event information was first added. Be cautious about using urevent information from legacy 4.1 datasets.

You may refer to the Event Processing Tutorial for more details on the event and urevent stuctures.

EEG.epoch

In an epoched dataset, this structure is similar to the *EEG.event* structure, except that there is only one record for each epoch. Note that EEGLAB functions never use the epoch structure. It is computed from the *EEG.event* structure by the eeg_checkset.m function (using flag 'eventconsistency') as a convenience for users who may want to use it to write advanced EEGLAB scripts. For the tutorial dataset, after extracting data epochs the epoch structure looks like this:

```
>> EEG.epoch

ans =

     struct array with fields:
                event
                eventlatency
                eventposition
                eventtype
                eventurevent
```

-->-->

Note that this dataset contains 80 epochs (or trials). Now type

```
>> EEG.epoch(1)

ans =

              [1 2 3]
```

```
    eventlatency: {[0] [695.2650] [1.0823e+03]}
  eventposition: {[2] [21] [5]}
  eventtype: {'square' 'rt' 'square'}
  eventurevent: {[1] [2] [3]}
```

-->-->

The first field *EEG.epoch.event* is an array containing the indices of all dataset events that occurred during this epoch. The fields *EEG.epoch.eventtype*, *EEG.epoch.eventposition* and *EEG.epoch.eventlatency* are cell arrays containing values for each of the events (*EEG.epoch.event*) that occurred during the epoch. Note that the latencies in *EEG.epoch.eventlatency* have been recomputed in units of milliseconds with respect to the epoch time-locking event. When there is only one event in an epoch, the epoch table is more readable.

You may refer to the Event Processing Tutorial for more details.

The next section will cover the supervening data structure, new as of EEGLAB v5.0b, the *STUDY* or *studyset*.

A06: Maximizing Memory

Options to Maximize Memory and Disk

Maximize memory menu

This section is intended for users who use large EEGLAB data-sets and need to optimize their use of main memory (RAM) and disk space. To modify the relevant EEGLAB options, select File > Maximize memory. If you cannot modify the file *eeg_options.m* in the toolbox distribution, the window below will pop up.

Simply press *Yes*. A copy of *eeg_options.m* will be stored in the local directory and modified when you change any of the EEGLAB options. (Note: For EEGLAB to use this file, the current directory (.) must appear BEFORE the EEGLAB toolbox directory in the Matlab path; see *path()*). If the original *eeg_option.m* file is writable, EEGLAB will modify it instead.

If the original distribution copy of the options file is modified, the new options will affect all EEGLAB sessions using that file. If, however, EEGLAB (or the user from the Matlab or shell commandline) copies the options file to the current working directory, then only EEGLAB sessions having this directory in their Matlab path (before the EEGLAB distribution directory) will follow the modified options. It is advised to have only one such option file to avoid confusion. Now the following window will pop up.

When the top option is set, EEGLAB can hold in memory more than one data-set at a time. New data-sets are created by most of the operations called under the Tools menu. With this option set, users can undo data-set changes immediately by going back to working with the parent (previous) data-set (unless they set the "overwrite data-set" option when they saved the new data-set). Processing multiple data-sets may be necessary when comparing data-sets from separate subjects or experimental conditions.

The second option allow to save the raw data in a separate file. This will be useful in future version of EEGLAB to read one channel at a time from disk. This also allow faster reading of data-set when they are part of a STUDY.

If the 3rd option is set, all the ICA activations (time courses of activity) will be pre-computed each time the ICA weights and sphere matrices are specified or changed. This may nearly double the main memory (RAM) required to work with the data-set. Otherwise, ICA activations will be computed by EEGLAB only as needed, e.g. each time EEGLAB calls a function that requires one or more activations. In this case, only the activations needed will be computed, thus using less main memory.

The bottom option is used to remember folder when reading data-sets.

The icadefs.m file

Using a text editor, you should edit file "icadefs.m" in the distribution before beginning to use EEGLAB. This file contains EEGLAB constants used by several functions. In this file, you may:

- Specify the filename directory path of the EEGLAB tutorial. Using a local copy of this EEGLAB tutorial (available online at the SCCN EEGLAB wiki requires a (recommended) Tutorial Download (obsolete).

  TUTORIAL_URL = 'http://sccn.ucsd.edu/wiki/EEGLAB'; % online version

- Reference the fast binary version of the *runica()* ICA function *ica* (see the Binica Tutorial). This requires another (recommended) download from the SCCN EEGLAB site.

  ICABINARY = 'ica_linux2.4'; % <=INSERT name of ica executable for binica.m

You can also change the colors of the EEGLAB interface using other options located in the *icadefs* file.

A07: Contributing to EEGLAB

EEGLAB operates in the very rich Matlab environment. The structure of EEGLAB functions makes it easy to combine them in new ways in original Matlab scripts which may use any of the wide variety of processing tools and methods available in Matlab. Thus, the most straightforward way to add to EEGLAB capabilities is to use EEGLAB functions and data structures in your own Matlab scripts, something many or most EEGLAB users do routinely.

Matlab functions provide a level of encapsulation and isolation that minimizes the possibility of interference with variable names and processing outside the functions themselves, e.g. in Matlab scripts that call them. EEGLAB is, in essence, a large set of Matlab functions. Please send us (eeglab@sccn.ucsd.edu) any EEGLAB-compatible functions you think would be of interest to other researchers. We will consider any functions sent us for inclusion in the EEGLAB distribution. Include a succinct explanation and references for the signal processing methods used. If possible, use EEGLAB help message style.

Open source policy

1. EEGLAB is distributed under the GPL GNU license, which states that the software cannot be modified for commercial purposes. Any contributed functions we add to EEGLAB will be made available for free non-commercial use under this license.
2. We will credit your authorship of the functions on the function help pages. The authors will retain all commercial rights to the functions.
3. Functions that we find to be accurate, stable, of some general interest and complementary to existing EEGLAB functions will first be included in a 'contributed' directory distributed as a part of EEGLAB. If a contributed function proves to be stable and of widespread interest, we may integrate it into the main EEGLAB menu.

Also consider the 'plug-in' option (described in Section below). EEGLAB _plug-ins_ allow authors to flexibly incorporate new functions in the EEGLAB menu of all users who have also downloaded their plug-in.

How to write EEGLAB functions

Adding new functionality to EEGLAB requires a pair of functions, a signal processing function (Ex: sample.m) and an accompanying pop_function (Ex: pop_sample.m). The pop_function pops up a text input window allowing the user to specify arguments to the signal processing function. The Matlab help messages for each function should state clearly what input arguments the functions require and what they output, using the help message format explained below. You should read the beginning of the script section to understand the different levels of functions in EEGLAB.

The signal processing function

This function should comply with the EEGLAB help-message syntax to allow the header to be converted into an .html help page by (functions makehtml() and help2html(). We propose this sample header. Below is the GNU license disclaimer to include in the function header.

> *This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.*
> *This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.*
> *You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA*

The associated pop_ function

The pop_function creates a graphic interface to the signal processing function. It should include the same disclaimer as the signal processing function and

1. It must take the EEGLAB data structure 'EEG' as a first input. The second parameter may specify whether the signal processing function will be applied to ICA component activations or to the raw EEG data channels.
2. Additional parameters should be optional. If they are left blank in the pop_function call, a window should pop-up to ask for their values.
3. The pop_function must return a 'string' containing the resulting call to the signal processing function (or in some cases to the pop_function). When this string is evaluated (using the Matlab function 'eval()'), the result must be the same as that of the pop_function itself, e.g. including all the parameters the user may enter into the pop window. This string will be pushed into the EEGLAB command history stack.
4. By convention, if the function draws a figure, calling the function without enough arguments should pop up a new figure. However, with enough arguments (macro call), the function should directly draw the output graphic in the current figure (thus allowing the user to build multi-part figures, e.g. using 'sbplot()' commands). Writing the pop_function is easy and can usually be done in a few minutes if you just modify the [BROKEN LINK pop_sample.m*|download/pop_sample.m] function source.

How to write an EEGLAB plug-in

EEGLAB also has a **plug-in** facility that automatically searches for functions loaded into a specified plug-in directory, causing them to appear in the EEGLAB menu. EEGLAB **plugin functions** can be offered for download and use by anyone, independent of the EEGLAB distribution, or by mutual agreement with the EEGLAB developers may be included in the EEGLAB distribution itself.

EEGLAB will automatically incorporate any appropriately named "plugin" functions (Ex: 'eegplugin_myfunc.m') that EEGLAB finds in the same directory as 'eeglab.m'. Creating an EEGLAB plugin function in this location will add a menu item with the menu label(s) specified in your plugin function to the bottom of an EEGLAB menu (the top menu label possibly linked to an unlimited number of sub-menus). These menu item(s) can call standard or custom data processing and "pop" functions (see examples below). When a user downloads an EEGLAB plugin (either from the main EEGLAB site or from any other site), he or she simply has to uncompress the plugin function files into the plugin function sub-directory or into the main EEGLAB directory (in which *eeglab.m* is located). The plugin function will then be detected by EEGLAB at startup by looking for files or directories with names beginning with *eegplugin_* in the main EEGLAB directory (i.e., in the same directory as 'eeglab.m'). You may also place this file in a sub-directory of the EEGLAB plugin function directory.

The plugin function

To create a new EEGLAB plug-in, simply create a Matlab function file whose name begins with **eegplugin_** and place it in the plugin function subdirectory or in your main EEGLAB directory. This function must take three arguments, as in the 'test' plugin function shown below:

>> eegplugin_test (fig, try_strings, catch_strings);

The three arguments above are provided to the plugin function by *eeglab()*. The first argument ('fig') is the handle of the main EEGLAB window. The second and third arguments are structures passed by EEGLAB that allow the plugin function to check parameters, detect errors, etc. (see below). If you do not want your plug-in to alter EEGLAB history and error message handling, you can ignore the latter two parameters (although the plugin function definition still must list all three arguments).

Adding a sub-menu

To create a new sub-menu under a top-level EEGLAB menu, simply add a command like this to your plugin function:

>> uimenu(fig, 'label', 'My function', 'callback', ... ['EEG = pop_myfunc(EEG, ...); [ALLEEG EEG CURRENTSET] ... = eeg_store(ALLEEG, EEG, CURRENTSET);']);

The statement *[ALLEEG EEG CURRENTSET] = eeg_store(ALLEEG, EEG, CURRENTSET);* above insures that your modified EEG dataset will be stored in the EEGLAB *ALLEEG* structure.

Plug-ins and EEGLAB history

If you want your plug-in to interact with the EEGLAB history mechanism, you should take advantage of the second ('try_strings') and third ('catch_strings') arguments to your plugin function. The second argument (see *eegplugin_test()* above) contains commands (organized into a Matlab structure) that check the input dataset and attempt to execute your command. The third argument ('catch_strings') contains commands to handle errors and add the contents of the LASTCOM (i.e., last command) variable to the EEGLAB history.

Plugin functions should declare one or more EEGLAB menu items. Each menu declaration should look like this:

```
uimenu( submenu, 'label', 'My function', 'callback', ...
          [ try_strings.anyfield '[EEG LASTCOM] ... = pop_myfunc(EEG, ...);' arg3.anyfield ]);
```

-->-->

Possible fields for 'try_strings' (above) are:

- *try_strings.no_check* : check for the presence of a non-empty EEG dataset only
- *try_strings.check_ica* : check that the dataset includes ICA weights
- *try_strings.check_cont* : check that the dataset is continuous
- *try_strings.check_epoch* : check that the dataset is epoched
- *try_strings.check_event* : check that the dataset contains events
- *try_strings.check_epoch_ica* : check that the dataset is epoched **and** includes ICA weights
- *try_strings.check_chanlocs* : check that the dataset contains a channel location file

- *try_strings.check_epoch_chanlocs* : check that the dataset is epoched **and** includes a channel locations (chanlocs) file
- *try_strings.check_epoch_ica_chanlocs* : check that the dataset is epoched **and** includes ICA weights **and** a channel locations file.

Possible fields for 'catch_strings' are:

- *catch_strings.add_to_hist* : add the LASTCOM variable content (if not empty) to the EEGLAB history
- *catch_strings.store_and_hist* : add the LASTCOM variable content (if not empty) to the EEGLAB history **and** store the EEG dataset in the ALLEEG variable.
- *catch_strings.new_and_hist* : add the LASTCOM variable content (if not empty) to the EEGLAB history **and** pop up a window for a new dataset.

Plugin function examples

A simplest type of plugin function might only call a plotting function. For instance, to write a simple plug-in to plot the ERP trial average at every channel in a different color (without performing any data checking):

```
% eegplugin_erp() - plot ERP plugin
function eegplugin_erp( fig, try_strings, catch_strings);

% create menu
plotmenu = findobj(fig, 'tag', 'plot');
uimenu( plotmenu, 'label', 'ERP plugin', ...'callback', 'figure; plot(EEG.times, mean(EEG.data,3));');
```

-->-->

Save the text above as a file, 'eegplugin_erp.m' into the plugin function sub-directory of EEGLAB (or the EEGLAB directory where _eeglab.m_ is located) and restart EEGLAB (click here to download this .m file). Then select the menu item Plot > ERP plugin to plot the ERP of an epoched dataset.

Another, more complete example: To create a plug-in named 'PCA' that would apply PCA to your data and store the PCA weights in place of the ICA weights, save the Matlab commands below as file 'eegplugin_pca.m' into the plugin sub-directory of EEGLAB (or the EEGLAB directory in which **eeglab.m** is located) and restart EEGLAB (click here to download this .m file).

```
% eegplugin_pca() - a pca plug-in
function eegplugin_pca( fig, try_strings, catch_strings);

% create menu
toolsmenu = findobj(fig, 'tag', 'tools');
submenu = uimenu( toolsmenu, 'label', 'PCA plugin');

% build command for menu callback\\ cmd = [ '[tmp1 EEG.icawinv] = runpca(EEG.data(:,:));' ];
cmd = [ cmd 'EEG.icaweights = pinv(EEG.icawinv);' ];
cmd = [ cmd 'EEG.icasphere = eye(EEG.nbchan);' ];
cmd = [ cmd 'clear tmp1;' ];

finalcmd = [ try_strings.no_check cmd ];
finalcmd = [ finalcmd 'LASTCOM = ''' cmd ''';' ];
finalcmd = [ finalcmd catch_strings.store_and_hist ];

% add new submenu
uimenu( submenu, 'label', 'Run PCA', 'callback', finalcmd);
```

-->-->

Note: As of EEGLAB v4.3 you may use plugin functions to append new plug-in related menu items under different EEGLAB menu headings. Above, we add a sub-menu to the Tools menu by specifying *'tag','tools'* in the *findobj()* call. If the specified tag were *import data*, EEGLAB would add the plug-in menu item to the File > Import data menu. Using the tag *import epoch* would add the plugin function menu item to the File > Import epoch info menu. The tag *import event* would add the plugin menu item to the File > Import event info menu. The tag *export* would add the plugin menu item to the File > Export data menu. Finally, the tag *plot* would add the plugin menu item to the Plot menu. (Note that the tag call should be in lower case).

After installing the plugin function above, a new EEGLAB menu item Tools > PCA plugin will be created. Use this menu item to run PCA on the current EEG dataset. The resulting PCA decomposition will be stored in place of the ICA decomposition. (Note: This is possible (if not typically advisable) since both PCA and ICA are linear decompositions).

See the EEGLAB DIPFIT plugin eegplugin_dipfit.m for an example of a more elaborate plug-in.

Note: In EEGLAB4.3 we slightly modified how EEGLAB handles plugin functions. As a result, later EEGLAB versions might not have been compatible with earlier plugin functions. Subsequent versions of EEGLAB have and will support backwards compatibility of the plug-in conventions.

EEGLAB menu integration

As of EEGLAB11, it is now possible for the user to control menu behavior within EEGLAB. EEGLAB enables and disables menu items automatically based on data in memory (as shown to the user by greying the menu item text). This may result in some plug-in related menu items being automatically disabled under circumstances in which they should not be. In EEGLAB11, we have implemented a new plug-in menu item activation scheme that makes it possible to control when a plug-in's menu item(s) will be active. To do so, developers must use several keywords in the "userdata" field of the menu item:

Keyword	Default	Description
epoch	on	set to **off** to disable the menu item when an epoched dataset is being processed
continuous	on	set to **off** to disable the menu item when a continuous dataset is being processed
startup	off	set to **on** to enable the menu item at start up when no data have been loaded
study	off	set to **on** to enable the menu item when a STUDY is being processed
chanloc	off	when set to **on**, the menu item will be disabled if channel locations are not present

Keywords should be separated by semicolons. An example of a plug-in related menu item that is enabled only at startup and when processing EEGLAB studies is shown below:

```
toolsmenu = findobj(fig, 'tag', 'tools');
uimenu( submenu, 'label', 'My menu', 'callback', mycommand, 'userdata', 'startup:on;study:on');
```

-->-->

Note that adding **epoch:on** or **continuous:on** would not change the menu item behavior since these are default behaviors. The following example would enable the menu item only when processing continuous or epoched datasets that have channel locations present:

```
toolsmenu = findobj(fig, 'tag', 'tools');
uimenu( submenu, 'label', 'My menu', 'callback', mycommand, 'userdata', 'chanloc:on');
```

-->-->

Omitting the **userdata** parameter results in the plugin menu item adopting the default behaviors defined in the table above.

Using EEGLAB GUI design functions

The function **supergui()** is the main function used for creating graphic interfaces in EEGLAB. We will be describing below some basic and more advanced uses for it:

Using the supergui function

Supergui was designed to alleviate the burden of manually creating graphic interface for each function. Instead a series of controls are defined along with a geometry and the figure is automatically created. An example is shown below:

```
supergui( 'geomhoriz', { 1 1 1 }, 'uilist', { ...
       { 'style', 'text', 'string', 'Hello!' }, { }, ...
       { 'style', 'pushbutton' , 'string', 'OK' } } );
```

-->-->

This creates the following graphic interface

-->-->

The command line call above contains two main arguments: 'geometry' and 'uilist'. 'geometry' defines the geometry in this case 3 lines { 1 1 1} (see more complex geometry below). The 'uilist' defines the content for each of these lines. The first line contains some text "{ 'style', 'text', 'string', 'Hello!' }". The second line is empty "{ }" and the third line contains the "OK" button "{ 'style', 'pushbutton' , 'string', 'OK' }".

If you press OK, nothing will happen however because no action has been associated with this button. To associate an action to a UI control, simply use the 'callback' command and enter any Matlab command in the associated string parameter.

```
supergui( 'geomhoriz', { 1 1 1 }, 'uilist', { ...
       { 'style', 'text', 'string', 'Hello!' }, { }, ...
       { 'style', 'pushbutton' , 'string', 'OK' 'callback' 'close(gcbf);' } } );
```

-->-->

Above, we added the 'callback' command "close(gcbf)" which closes the current figure when users press the "OK" button.

Almost any geometry (but not all as detailed in the advanced section) may be defined using this scheme. For instance

```
supergui( 'geomhoriz', { [1 1] 1 1 }, 'uilist', { ...
       { 'style', 'text', 'string', 'Enter some text' }, ...
       { 'style', 'edit', 'string', 'Hello!' }, { }, ...
       { 'style', 'pushbutton' , 'string', 'OK' 'callback' 'close(gcbf);' } } );
```

-->-->

-->-->

The first line now has geometry [1 1] which means that two UI will share this line and have the same horizontal extent. Changing [1 1] to [1 0.5] as shown below forces the second UI on the line to have only half the horizontal extent of the first one.

-->-->

In some case, it is convinient to have a UI spam more than one line of text. This is often the case for listbox for instance. Below we define a listbox with multiple choice and use the additional command 'geomvert' to specify the vertical geometry. It is set to [3 1 1] which means that the UI on the first line will span 3 lines of text, and the other lines will only span one line of text (we have to add carriage returns - character 10 - to the text of the first UI so this text is aligned with the listbox).

```
supergui( 'geomhoriz', { [1 1] 1 1 }, 'geomvert', [3 1 1], 'uilist', { ...
       { 'style', 'text', 'string', [ 'Make a choice' 10 10 10 ] }, ...
       { 'style', 'listbox', 'string', 'Choice 1|Choice 2|Choice 3' }, { }, ...
       { 'style', 'pushbutton' , 'string', 'OK' 'callback' 'close(gcbf);' } } );
```

-->-->

-->-->

Using the inputgui function

Although supergui can be useful, it only creates the GUI and all the control flow of events needs to be handled by the UIs; i.e., creating a button "OK", a button "Cancel" and handling the events associated to these buttons are left to the user. The **inputgui()** function (which calls the supergui function) helps supplement this by automatically creating these buttons and also by processing outputs. For instance:

```
res = inputgui( 'geometry', { [1 1] }, ...
       'geomvert', [3], 'uilist', { ...
       { 'style', 'text', 'string', [ 'Make a choice' 10 10 10 ] }, ...
       { 'style', 'listbox', 'string', 'Choice 1|Choice 2|Choice 3' } } );
```

-->-->

-->-->

The "res" output contains the results in this case a cell array containing a single value from 1 to 3 indicating which choice the user made (for instance { [1] }). The number of elements in this cell array depends on the number of UIs returning output (in general text and button do not return output but checkbox, listbox, and edit box do). Another way to use this function is to set tag for each UI and use the 4th output

```
[res userdata err structout] = inputgui( 'geometry', { [1 1] }, ...
        'geomvert', [3], 'uilist', { ...
        { 'style', 'text', 'string', [ 'Make a choice' 10 10 10 ] }, ...
        { 'style', 'listbox', 'string', 'Choice 1|Choice 2|Choice 3' 'tag' 'list' } } );
```

-->-->

Now the output structure "structout" contains a field named "list" that will contain the output of the listbox (structout.list = 1 for instance). This is convenient for creating GUIs, as developers do not need to remember the order of the output. You may change the order of lines and you

will not have to change any other part of your script. Note that this is not the case when using the "res" output described at the beginning of this section.

A complex UI shown below uses this technique. It is inspired by the pop_newtimef() function.

```
g = [1 0.3 0.6 0.34];
geometry = { g g g g g g g g [1.025 1.27] [1] [1.2 1 1.2]};
uilist = { ...
  { 'Style', 'text', 'string', 'Channel number', 'fontweight', 'bold'  } ...
  { 'Style', 'edit', 'string', '1' 'tag' 'chan'} {} {} ...
  ...
  { 'Style', 'text', 'string', 'Time limits [min max] (msec)', 'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', ' ' 'tag' 'tlimits' } ...
  { 'Style', 'popupmenu', 'string', 'Use 50 time points|Use 100 time points' ...
    'tag' 'ntimesout' 'value' 4} { } ...
  { 'Style', 'text', 'string', 'Frequency limits (Hz)', 'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', ' ' 'tag' 'freqs'  } ...
  { 'Style', 'popupmenu', 'string', 'Use limits, padding 1|Use limits, padding 2' ...
    'tag' 'nfreqs' }  ...
  { 'Style', 'checkbox', 'string' 'Log spaced' 'value' 0 'tag' 'freqscale' } ...
  ...
  { 'Style', 'text', 'string', 'Baseline limits [min max] (msec) (0->pre-stim.)', ...
    'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', '0' 'tag' 'baseline' } ...
  { 'Style', 'popupmenu',  'string', 'Use divisive baseline|Use standard deviation' ...
    'tag' 'basenorm' } ...
  { 'Style', 'checkbox', 'string' 'No baseline' 'tag' 'nobase' } ...
  { 'Style', 'text', 'string', 'Wavelet cycles [min max/fact] or sequence', ...
    'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', '3 0.5' 'tag' 'cycle' } ...
  { 'Style', 'popupmenu', 'string', 'Use limits|Use actual seq.' 'tag' 'ncycles' } ...
  { 'Style', 'checkbox', 'string' 'Use FFT' 'value' 0 'tag' 'fft' } ...
  { 'Style', 'text', 'string', 'ERSP color limits [max] (min=-max)', ...
    'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', ' ' 'tag' 'erspmax'} ...
  { 'Style', 'checkbox', 'string' 'see log power (set)' 'tag' 'scale' 'value' 1} ...
  {} ...
  { 'Style', 'text', 'string', 'ITC color limits [max]', 'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', ' ' 'tag' 'itcmax'} ...
  { 'Style', 'checkbox', 'string' 'plot ITC phase (set)' 'tag' 'plotphase' } {} ...
  { 'Style', 'text', 'string', 'Bootstrap significance level (Ex: 0.01 -> 1%)', ...
    'fontweight', 'bold' } ...
  { 'Style', 'edit', 'string', ' ' 'tag' 'alpha'} ...
  { 'Style', 'checkbox', 'string' 'FDR correct (set)' 'tag' 'fdr' } {} ...
  { 'Style', 'text', 'string', 'Optional newtimef() arguments (see Help)', ...
    'fontweight', 'bold'} ...
  { 'Style', 'edit', 'string', ' ' 'tag' 'options' } ...
  {} ...
  { 'Style', 'checkbox', 'value', 1, 'string', ...
    'Plot Event Related Spectral Power', 'tooltipstring', ...
    'Plot log spectral perturbation image in the upper panel' 'tag' 'plotersp' } ...
  { 'Style', 'checkbox', 'value', 1, 'string', ...
    'Plot Inter Trial Coherence', 'tooltipstring', ...
    'Plot the inter-trial coherence image in the lower panel' 'tag' 'plotitc' } ...
  { 'Style', 'checkbox', 'value', 0, 'string', ...
    'Plot curve at each frequency' 'tag' 'plotcurve' } ...
  };

  [ tmp1 tmp2 strhalt structout ] = inputgui( geometry, uilist, ...
       'pophelp(''pop_newtimef'');', 'Plot channel time frequency -- pop_newtimef()');
```

-->-->

Copying and pasting the script above should generate the GUI below. Note how simply the geometry was defined. The supergui function adapts automatically the width and height of the window to the containing elements. Also note the "Help" button which executes the command "'pophelp(*pop_newtimef*);'" defined in the inputgui command line call. Upon pressing '**OK**', the fields of the the "structout" structure contain all the user entry in the GUI.

-->-->

Advanced GUI design

One limitation of the supergui calling format is that it is not possible to define random geometries. Any given line needs to span the same height. There is an advanced calling format that allows to circumvent that problem.

```
supergui('geom', { {2 2 [0 0] [1 2] } { 2 2 [1 0] [2 1] } { 2 2 [1 1] [2 2] } }, ...
         'uilist', { { 'style' 'listbox' 'string' 'cell 1|cell 2' } ...
                     { 'style' 'checkbox' 'string' 'cell 3' } ...
                     { 'style' 'text' 'string' 'cell 4' } });
```

-->-->

-->-->

This type of interface is not possible using standard calls. The 'geom' input is detailed in the help section of the supergui function.

A08: DIPFIT

DIPFIT plug-in: Equivalent dipole source localization of independent components

A major obstacle to using EEG data to visualize macroscopic brain dynamics is the under-determined nature of the inverse problem: Given an EEG scalp distribution of activity observed at given scalp electrodes, any number of brain source distributions can be found that would produce it. This is because there are any number of possible brain source area pairs or etc. that, jointly, add (or subtract) nothing to the scalp data. Therefore, solving this 'EEG inverse' problem uniquely requires making additional assumptions about the nature of the source distributions. A computationally tractable approach is to find some number of equivalent current dipoles (like vanishingly small batteries) whose summed projections to the scalp most nearly resemble the observed scalp distribution.

Unfortunately, the problem of finding the locations of more than one simultaneously active equivalent dipoles does not have a unique solution, and "best fit" solutions will differ depending on the observed scalp distribution(s) (or scalp 'map(s)') that are given to the source inversion algorithm. For this reason, approaches to solving the EEG inverse problem have tended to focus on fitting scalp maps in which a single dipolar scalp map is expected to be active, for instance very early peaks in ERP (or magnetic ERF) response averages indexing the first arrival of sensory information in cortex. Others attempt to fit multiple dipoles to longer portions of averaged ERP (or ERF) waveforms based on "prior belief" (i.e., guesses) as to where the sources *should* be located. This approach has often been criticized for not being based wholly on the observed data and thus subject to bias or ignorance.

Using ICA to un-mix the unaveraged EEG is a radically different approach. ICA identifies temporally independent signal sources in multi-channel EEG data as well as their pattern of projection to the scalp surface. These 'component maps' have been shown to be significantly more dipolar (or "dipole-like") than either the raw EEG or any average ERP at nearly any time point -- even though neither the locations of the electrodes nor the biophysics of volume propagation are taken into account by ICA (Delorme et al., ms. in preparation). Many independent EEG components have scalp maps that nearly perfectly match the projection of a single equivalent brain dipole. This finding is consistent with their presumed generation via partial synchrony of local field potential (LFP) processes within a connected domain or patch of cortex.

Fortunately, the problem of finding the location of a single equivalent dipole generating a given dipolar scalp map is well posed. Note, however, that the location of the equivalent dipole for synchronous LFP activity in a 'cortical patch' will in general not be in the center of the active cortical patch. In particular, if the patch is radially oriented (e.g. on a gyrus, thus aimed toward the supervening scalp surface), the equivalent dipole location tends to be deeper than the cortical source patch.

Component localization in EEGLAB: EEGLAB now includes two plug-ins for localizing equivalent dipole locations of independent component scalp maps: (1) the DIPFIT plug-in calling Matlab routines of Robert Oostenveld (F.C. Donders Centre, Netherlands); and (2) a BESAFIT plug-in linking to older versions of the BESA program (Megis Software GMBH, Germany) running outside of Matlab (see A7). Use of the DIPFIT plug-in is strongly recommended.

DIPFIT is an EEGLAB plug-in based on functions written and contributed by Robert Oostenveld, and ported to EEGLAB by Robert in collaboration with Arnaud Delorme. DIPFIT2, released near the beginning of 2006 with EEGLAB v4.6, can perform source localization by fitting an equivalent current dipole model using a non-linear optimization technique (scherg1990) using a 4-shell spherical model (kavanagh1978) or by using a standardized boundary element head model (oostendorp1989). DIPFIT2 makes use of the FieldTrip toolbox, which is installed by default with EEGLAB.

Advances in DIPFIT2 (first distributed with EEGLAB v4.6-) include:

- Spherical Head Model: The spherical head model has been better co-registered to the MNI brain. This results in a shift in dipole visualization of about 1 cm downward compare to the same dipoles plotted in DIPFIT1. Spherical dipoles coordinates are also now converted to MNI and to Talairach coordinates, making comparisons possible with other imaging results.
- Boundary Element Model: Localization within a three-shell boundary element model (BEM) of the MNI standard brain is now possible.
- Channel Co-registration: A new function has been adding for performing manual or automated alignment of measured dataset channel locations to stored spherical or BEM head model template channel montages.

Dipole fitting with DIPFIT2

The scalp maps of many ICA components are compatible with their generation in separate domains of partial synchrony in cortex. Because local connections in cortex have a much higher density than longer range connections, it is reasonable to assume that synchronous coupling of neuronal activity, as isolated by ICA, usually occurs within a single brain area. Some ICA component scalp maps do highly resemble the projection of a single equivalent dipole (or in some cases a bilaterally symmetric pair of dipoles). Residual scalp map variance of the best-fitting single- or two-dipole component models is often surprisingly low (see below), given the relative inaccuracy of the (spherical) head model used to compute it. Such ICA components may thus represent projection of activity from one (or two symmetric) patch(es) of cortex.

To fit dipole models to ICA components in an EEGLAB dataset, you first need to perform ICA decomposition and then select the components to be fitted. To use DIPFIT to fit independent component maps for an EEGLAB dataset, you must first build or load the dataset, import a channel location file (see the EEGLAB tutorial on Importing a Channel Location File) and compute an ICA decomposition of your data (see Performing Independent Component Analysis of EEG data).

To follow the dipole fitting example used in this tutorial, download a 69-channel sample dataset for DIPFIT and BESA dipole localization here: eeglab_dipole.set (>1 MB). This sample dataset contains a channel location file and pre-computed ICA weights. Note: to save space, this dataset has been reduced to only a single trial, after performing ICA on about 800 similar trials; You should not try to apply ICA to this dataset, but should instead use the pre-processed ICA weight matrix to test dipole localization. Load the dataset into EEGLAB using menu item File > Load Existing Dataset.

Next, you must select which component to fit. Independent component (IC) 5 of the sample dataset decomposition is a typical lateral posterior alpha rhythm process. To plot component scalp map, use menu item Plot > Component maps > In 2-D, enter *5* for the component number and option *'electrodes', 'pointlabels'* to obtain the plot below. Note that the channel location file for this subject has been scanned using the Polhemus system, so the electrode locations are not exactly symmetrical.

There are three steps required to create equivalent dipole models for independent components:

- Setting model and preferences: This involves choosing the model (spherical or boundary element) and excluding some channels from the fitting procedure (e.g., eye channels).
- Grid scanning: This involves scanning possible positions in a coarse 3-D grid to determine an acceptable starting point for fitting equivalent dipoles to each component.
- Non-linear interactive fitting: This involves running an optimization algorithm to find the best position for each equivalent dipole.

Below we describe these three steps in detail. Note that the grid scanning and non-linear optimization may also be performed automatically for a group of selected components. This is described later in this chapter.

Setting up DIPFIT model and preferences

Before running DIPFIT, we must select some input parameters. Select the EEGLAB menu item Tools > Locate dipoles using DIPFIT > Head model and settings to modify DIPFIT settings. This will pop up the window below:

The top edit box, *Model (click to select)*, specifies the type of head model -- spherical or boundary element (BEM). By default, the spherical head model uses four spherical surfaces (skin, skull, CSF, cortex) to model the brain. The BEM model is composed of three 3-D surfaces (skin, skull, cortex) extracted from the MNI (Montreal Neurological Institute) caconical template brain also used in SPM (see here for details, the brain used is referred to as "collin27"). In many case, the BEM model is more realistic than the four concentric spheres model, and will return more accurate results. However the BEM model is a numerical model and sometimes leads to incorrect solutions due to numerical instabilities, whereas the spherical model is an analytical model. Furthermore, the spherical model returns results that are comparable with the BESA tool (below). For this example, select *Boundary element model*.

Clicking on the model name updates the fields below. The entry *Model file* contains the head model parameters (surface information, conductances, etc...). These are Matlab files and may be edited. See the FieldTrip documentation for more information on the head model files. The entry *Output coordinates* contains the format of the output dipole coordinates. Note that all dipoles are always plotted in MNI/Talairach coordinates. These parameters should only be updated for custom brain models.

The entry *MRI* contains the name of the MRI image to plot dipoles within. You may enter a custom or individual subject MR image file, assuming this file has first been normalized to the MNI brain. See a tutorial page on How to normalize an MR brain image to the MNI brain template using SPM2. Note that the SPM2 software directory must be in your Matlab path to use the resulting file in DIPFIT.

The entry *Template channel .loc file* contains the name of the template channel location file associated with the head model. This information is critical, as your dataset channel location file may be different from the template. If so, a co-registration transform is required to align your channel locations using the template locations associated with the model. Before we move on to co-registration, note the last edit box *Omit channels for dipole fit*. By pressing *List*, a list of channels appears that allows users to exclude eye channels (or other, possibly non-head channels) from the dipole fitting procedure.

Channel co-registration is the most important part and critical part of the DIPFIT setting procedure. If your channel locations are not aligned with the surface of the selected head model, the coordinates of the dipoles returned by DIPFIT will be meaningless.

Solution 1 (required for fitting the sample data): If all your electrode locations are within the International 10-20 System, the easiest way to align your channels to the head model is to simply use the standard channel coordinates associated with the head model. Here, no co-registration is required. To do this, press the *No coreg* checkbox and close the DIPFIT settings window (by pressing *OK*). Then go to the channel editing window (select menu item Edit > Channel location). The resulting channel editor window is shown below:

Press the *Look up locs* to look up your channel locations (by matching the channel labels) in the template channel location file.

If you had wanted to use the template channel locations for the spherical model, you would have selected the first option in the pop-up menu *Use BESA file for four-shell DIPFIT spherical model*. If you want to use the template channel locations for the BEM model, scroll down the pop-up menu and click on the button labeled *Use MNI coordinate file for the BEM DIPFIT model*, then press *OK*. At this point you may perform coarse grid scanning and non-linear fine fitting as explained in the next section.

<u>Solution 2:</u> If you are using channel locations and/or labels *not* in the International 10-20 System -- for example, scanned electrode positions, or some commercial high-density electrode cap file -- you will need to align or co-register your electrode locations with the selected head model. DIPPLOT does not actually allow you to align your electrode locations to the head model itself, but rather allows you to align your electrode locations to matching template electrode locations associated with the head model.

To try this, click on *Manual coreg.* in the DIPFIT settings window. The following instruction window will first appear:

If you have selected the spherical head model and pressed *OK*, the following co-registration window will appear. Here, the electrode locations are plotted on the sphere that represents the head. Note the schematic nose on the lower right; this will rotate with the head sphere. Each small red or green sphere indicates an electrode location, with fiducial locations (conventionally, the nasion and ear canal centers) drawn as bigger and darker spheres (more visible in the second image below).

If you have selected the BEM model, the following window will appear:

Use the *Warp* button to align and scale your electrode locations file so that it becomes best aligned with the template electrode file associated with the head model.

If you have no channels with labels that are common to the labels in the template montage, a channel correspondence window will pop up:

The channel labels from your dataset electrode structure are shown in the right column, while the left column shows channel labels from the template channel file associated with the head model. Arrows in both columns indicate electrodes with the same labels in the other column. If your channels labels do not correspond to the International 10-20 System labels used in the template montage, press the *Pair channels* button and choose the nearest channel to each of your dataset channels in the template montage. See here and Oostenveld 2001 for more information about 10-20 channel locations.

When you press *OK*, the function will perform the optimal linear 3-D warp (translation, rotation, and scaling) to align your channel montage to the template montage associated with the head model. The warping procedure uses the Fieldtrip toolbox which is installed by default with EEGLAB. The result will be shown in the channel montage window (see below). You may press the *Labels on* button to toggle display of the channel labels for your channel structure (green) or the template channel file associated with the head model (red). You may also restrict the display to subsets of channels using the *Electrodes* buttons.

Fine tuning: To finely tune the alignment manually, repeatedly edit the values in the edit boxes. Here:

- *Yaw* means rotation in the horizontal plane around the z axis.
- *Pitch* and *Roll* are rotations around the x and y axes.

The resulting co-registration window should look something like this:

Note about fiducials: Your channel structure may contain standard fiducial locations (nasion and pre-auricular points). If you import a channel file with fiducial locations into the channel editor, in EEGLAB v4.6- give them the standard 'fiducial' channel type *FID* and they will be stored in the channel information structure, *EEG.chaninfo*. This will also be done automatically if your fiducials have the standard names, *Nz* (nasion), *LPA* (left pre-auricular point), and *RPA* (right pre-auricular point). Note that fiducial locations are stored outside the standard channel location structure, *EEG.chanlocs*, for compatibility with other EEGLAB plotting functions.

Thereafter, fiducial locations will appear in the channel co-registration window (above) and may be used (in place of location-matched scalp channels) to align your electrode montage to the template locations associated with the head model. Use the *Align fiducials* button to do this. Press *OK* to update the DIPFIT settings window. This will display the resulting talairach transformation matrix, a vector comprised of nine fields named *[shiftx shifty shiftz pitch roll yaw scalex scaley scalez]*. Then press *OK* in the DIPFIT settings window and proceed to localization.

Initial fitting - Scanning on a coarse-grained grid

Before you perform interactive dipole fitting, first allow DIPFIT to scan for the best-fitting dipole locations on a coarse 3-D grid covering the whole brain. The solutions of this grid search are not very accurate yet, but they are acceptable as starting locations for the non-linear optimalization. Starting from these best grid locations will speed up finding the final best-fitting solution. (The next section, A4.4, explains the fine tuning using the non-linear optimization). The tutorial instructions below are valid for both the spherical head model and the boundary element model. To scan dipoles on a coarse grid, select menu item Tools > Locate dipoles using DIPFIT > Coarse fit (grid scan). If you use the sample dataset, the window below will pop up:

The first edit box *Component(s)* allows you to select a subset of components to fit. Note: selecting only a few components does not increase the speed of the computation, since the forward model still has to be computed at each location on the grid. The *Grid in X-direction*,*Grid in Y-direction*, and *Grid in Z-direction* edit boxes allow you to specify the size and spacing of the coarse grid. By default, DIPFIT uses 10 steps equally spaced between -radius and +radius of the sphere that best matches the electrode positions. Since equivalent dipoles are not likely to be in the lower hemisphere of the model head, by default DIPFIT only scans through positive Z values. The last edit box, *Rejection threshold RV(%)*, allows you to set a threshold on the maximum residual variance that is accepted. Using this threshold, components that do not resemble a dipolar field distribution will not be assigned a dipole location.

Press *OK* and a bar will pop up to show the progress of the coarse grid scanning. Note: during dipole localization, the electrode positions are projected to the skin surface of the spherical or BEM head model, though the electrode positions as saved in the dataset itself are *not* altered. DIPFIT starts the grid scan by first excluding all 3-D grid locations that are outside the head. It then computes the forward model (dipole-to-electrode projections) at each remaining grid location and compares it with all component topographies. The progress of this process is shown in the text window. When the process is completed, select menu item Tools > Locate dipoles using DIPFIT > Plot component dipoles to call the window below:

Simply press *OK*, to produce the figure below:

Here, all the dipoles plotted had a residual variance (vis a vis their component maps) below 40% (as we specified in the coarse grid search interactive window). Note that some components end up having the same x, y and z coordinates because of the coarseness of the grid. You may view individual components by pressing the *Plot one* button. The component index and its residual variance are then indicated in the top-left corner (see the 'dipplot()' visualization tutorial below for further details).

Interactive fine-grained fitting

To scan dipoles interactively, call menu item Tools > Locate dipoles using DIPFIT > Fine fit (iterative). The following windows pop up. Enter a component index (here, 3) in the *Component to fit* edit box.

Prior to fitting the component, press the *Plot map* button to show the component scalp map. The following window pops up.

This component, showing a clear left-right symmetric activity, cannot be accurately modeled using a single dipole. To fit this component using two dipoles constrained to be located symmetrically across the (corpus callosum) midline, set both dipole 1 and 2 to 'active' and 'fit' (by checking the relevant checkboxes in the pop window). Then press the *Fit selected dipole position(s)* button. If fitting fails, enter different starting positions (e.g., [-10 -68 17] for first dipole and [10 -68 17] for the second dipole and refit). When fitting is done, note the low residual variance for the two-dipole model on the top right corner of the interactive interface (0.53 %). Then, press the *Plot dipole(s)* button. The following plot pops up (see the'dipplot()' visualization tutorial below).

Note that the polarity of components is not fixed (but their orientation is): the product of the component scalp map value and the component activitation value (i.e., the back-projected contribution of the component to the scalp data) is in the original data units (e.g., microvolts), but the polarities of the scalp map and of the activation values are undetermined. For example, you may multiply both the scalp map and the component activity by -1 while preserving the component back-projection to the scalp (since the two negative factors cancel out). As a result, you may choose to flip the visualized dipole orientations (use the pop-up window for fitting, then press the *Flip in|out* button and then the *Plot dipole(s)* button to re-plot the dipoles with indicated orientations reversed.

Important note: when you are done, press the *OK* button in the interactive interface for dipole fitting. Do not press *Cancel* or close the window -- if you do, all the dipole locations that you have computed using this interface will be lost! Also, this DIPFIT menu is the only one that does not generate EEGLAB history. The first menu item, Tools > Locate dipoles using DIPFIT > Autofit component, uses the same method for dipole fitting but also generates an EEGLAB history command that can be re-used in batch scripts.

Automated dipole fitting

Automated dipole fitting performs the grid search and the non-linear fitting on several component without human intervention. You still need to select the model in the DIPFIT settings interactive window though. To find a best-fitting equivalent dipole for the component above, select the EEGLAB menu item Tools > Locate dipoles using DIPFIT > Autofit (coarse fit, fine fit, plot) to automatically fit selected ICA components. Set the *Component indices* to *5*, enter *100* in the *rejection threshold* edit box so the iterative solution is computed regardless of the residual variance for the coarse fit, and check the *Plot resulting dipoles* checkbox to plot component dipoles at the end of fitting. Then, press *OK*.

The function starts by scanning a 3-D grid to determine acceptable starting positions for each component. Then it uses the non-linear optimization algorithm to find the exact dipole position for each component. At the end of the procedure, the following window pops up.

The residual variance *RV: 2.75%* is shown below the component number *Comp: 5* on the right top corner indicates that the component being plotted is component 5 and that the residual variance is 2.75%. The dipplot.m function above allows you to rotate the head model in 3-D with the mouse, plot MRI slices closest to the equivalent dipole, etc... (See the 'dipplot()' visualization tutorial below for more details.)

Visualizing dipole models

Use menu item Tools > Locate dipoles using DIPFIT > Plot component dipoles and select component number 3 and 5 to produce the plot shown below (assuming you performed the fit for both components 3 and 5 as described above). Select all options except the *Summary mode* and enter *'view', [51 18]* in the *Additional dipplot() options* text box to set the initial 3-D view; the function will print the component indices.

A plot pops-up. After 3D rotation it may look like the following plot.

Press the *Plot one* button. You may scroll through the components by pressing the *Next/Prev* buttons. Note that the closest MRI slices to the dipole being currently plotted are shown. Note also that if you do not select the option *Plot closest MRI slices* in the graphic interface, and then press the *Tight view* button in the dipplot window, you will be able to see the closest MRI slices at the location of the plotted dipole as shown below. Try moving the 3-D perspective of the tight view plot with the mouse, and/or selecting the three cardinal viewing angles (sagittal, coronal, top) using the lower left control buttons.

Note that it is not yet possible to toggle the dipole "stems" or "projection lines" on and off from within the graphic interface. You need to use the EEGLAB menu again, unselecting the relevant checkboxes. Also, it is not yet possible to use the subject's own anatomical MRI as background (it will be possible in the future upon completion of a EEG/MRI co-registration function as described below).

Finally, again call menu item Tools > Locate dipoles using DIPFIT > Plot component dipoles. Select *Summary mode* and press *OK*.

For comparison, using the spherical model, the location of the dipoles would have been as shown in the following picture. Notice the similarity of the talairach coordinates for the two models. Notice also that the BEM model does not overestimate dipole depth compare to the spherical model (which is usually the case when the BEM model mesh is too coarse). The MRI used to plot the spherical result is the average MNI brain (over about 150 subjects) whereas the MRI used to plot the BEM is the average MNI brain of a single subject (which was used to generate the model leadfield matrix).

The entry *Background image* contains the name of the MRI in which to plot dipoles. You may enter a custom MRI file assuming this file has been normalized to the MNI brain (see How to normalize an MR brain image to the MNI brain template using SPM2. Note that the SPM2 package directory in your network must be in your Matlab path to be able to use the resulting file in EEGLAB. This is true both for the spherical head model (co-registered to the MNI brain) and the BEM model (based on the MNI brain).

Note: Co-registration of the MNI average MR image with landmark electrode positions for the spherical model. To plot dipole locations in the average MR image (as in the dipplot.m plots above), we co-registered the MNI average brain MRI with landmark electrode positions. For the

average MRI image, we used a publicly available average brain image (average of 152 T1-weighted stereotaxic volumes made available by the ICBM project) from the MNI database (Montreal Neurological Institute (MNI), Quebec). Co-registration of the MNI brain and the standard EEG landmarks was accomplished automatically using fiducials and the Cz (vertex) electrode position, then slightly adjusted manually. The co-registration differs slightly from that used in DIPFIT1; dipoles are localized about 10 mm deeper than in DIPFIT1. Other coordinates (x,y) are nearly identical. We believe the co-registration in DIPFIT2 to be more accurate than in DIPFIT1.

You may also see the co-registered model head sphere plotted over the MNI brain in the dipplot.m window by pressing the *Mesh on* button in the BEM model below (here in 'tight view').

Or, for the spherical model below (in summary mode). Note that the sphere model is stretched in the visualization to the ellipsoid that best matches the shape of the headsurface of the average MRI.

Plotting dipole locations on scalp maps

Select menu item Plot > Component maps > In 2-D, enter *1:20* in the component index edit box. Note: The sample dataset structure contains pre-computed component dipoles. If you are processing another dataset, first select menu item Tools > Locate dipoles using DIPFIT > Autofit components to fit ICA components from 1 to 20 automatically as described in section A1.1. above. Check the *Plot dipole* checkbox (see below) and press *OK*. Note: You may also enter additional dipole fitting options in the last edit box; press the *Help* button for details.

The following plot will pop up.

Using DIPFIT to fit independent MEG components

Dipfit (v2.1) now supports source localization for MEG data acquired using CTF (Vancouver, CA) systems. Note that a custom head model needs to be used for MEG source localization since it is not possible to do accurate source localization on a template head model from MEG data. The main reason for this limitation is that, first, MEG sensor positions do not follow any positioning standard. Each system may have its own arrangement. Furthermore, unlike in EEG the sensors are not placed on an elastic cap that automatically fits to the head of the subject. The MEG sensor coils are located within a fixed helmet in which the majority of subjects' heads fit loosely. The position of a given sensor relative to a cortical structure could be quite different from subject to subject, and even across different recordings from the same subject. For this reason, the sensor's location is always reported relative to the center and the orientation of the subject's head during the recording, which is never exactly the same across recordings. Consequently, a head model needs to be created for each recording using the CTF MRIViewer and localSphere programs (see the CTF documentation). Then, in EEGLAB, call menu item Tools > Locate dipoles using DIPFIT > Head model and settings, and select the tab *CTF* in the upper pop-window. Set up the DIPFIT model and preferences as follows:

- Select CTF as model.
- The model file is the file *default.hdm* created by CTF localSphere, located in the *xxx.ds* folder.
- The MRI head image should be the *xxx.mri* file used by MRIViewer to extract the cortical surface.
- The channel file should be the *xxx.res4* file within the *xxx.ds* folder. This file also contains gradiometer-specific information, including sensor orientation, that is not in the EEG structure.

As for EEG data, you first need to scan a relavely coarse grid, to find an appropriate starting position for each dipole, by calling menu item Tools > Locate dipoles using DIPFIT > Coarse fit (grid scan). Note that CTF head models are in cm. Consequently, the grid used for gridsearch needs to be specified in cm (this is done automatically by pop_dipfit_gridsearch). Then as in EEG dipole fitting, use menu item Tools > Locate dipoles using DIPFIT > Fine fit (iterative) to optimize dipole positions nonlinearly.

Finally, you may plot dipoles on the subject MRI using menu item Tools > Locate dipoles using DIPFIT > Plot component dipoles as shown below. The corresponding scalp map is also shown on the right. Because of co-registration issues, it is not possible to plot the dipole positions on the scalp map as in EEG). It is strongly advisable to normalize dipole lengths.

You can also run dipfit2.1 for MEG data without a subject MR image if you have extracted the subject head surface from another source, like a head model acquired using a Polhemus 3-D location system. However, it would then not be possible to visualise the dipole within EEGLAB. Any comparison between or drawing of dipoles for different subjects requires a thourough understanding of the MEG coordinate system, since that is being used to express the dipole position. The default is in individual subjects' head coordinates, and given that head sizes differ between subject, the position of dipoles cannot be directly compared over subjects. It is therefore advisable to coregister and spatially normalize your subjects' MRIs to a common template (such as the MNI template) and to apply the same coregistration and normalization to the dipole positions before comparing them. These operations need to be executed outside EEGLAB. A test set is available here. Simply load the test file and follow the instruction above. Thanks to Nicolas Robitaille for his efforts to make this new feature work, in documenting it, and providing test data.

Using DIPFIT to fit EEG or ERP scalp maps

Though the implementation of the DIPFIT plug-in has not been expressly designed to fit dipoles to raw ERP or EEG scalp maps, EEGLAB provides a command line function allowing DIPFIT to do so. Fitting can only be performed at selected time points, not throughout a time window. First, you must specify the DIPFIT settings on the selected dataset (see section A4.2. Setting up DIPFIT model and preferences). Then, to fit a time point at 100 ms in an average ERP waveform (for example) from the main tutorial data set (eeglab_data_epochs_ica.set), use the following Matlab commands.

```
% Find the 100-ms latency data frame
latency = 0.100; <br /> pt100 = round((latency-EEG.xmin)*EEG.srate);

% Find the best-fitting dipole for the ERP scalp map at this timepoint
erp = mean(EEG.data(:,:,:), 3);
EEG = pop_dipfit_settings(EEG); <font color="black">''% Follow GUI instructions to perform co-registration
[ dipole model TMPEEG] = dipfit_erpeeg(erp(:,pt100), EEG.chanlocs, 'settings', EEG.dipfit, 'threshold', 100);

% Show dipole information (spherical or MNI coordinate based on the model selected) dipole

% plot the dipole in 3-D
pop_dipplot(TMPEEG, 1, 'normlen', 'on');

% Plot the dipole plus the scalp map
figure; pop_topoplot(TMPEEG,0,1, [ 'ERP 100ms, fit with a single dipole (RV ' num2str(dipole.rv*100,2) '%)'], 0, 1);
```

-->-->

Click here to download the script above.

DIPFIT structure and functions

Like other EEGLAB functions, DIPFIT functions are standalone and may also be called from the command line. Note that whenever you call the DIPFIT menu from EEGLAB, a text command is stored in the EEGLAB history as for any other EEGLAB command. Type *h* on the Matlab command line to view the command history.

DIPFIT creates a *EEG.dipfit* sub-structure within the main *EEG* dataset structure. This structure has several fields:

EEG.dipfit.chansel

	Array of selected channel indices to fit (for instance ECG or EYE channels might be excluded).
EEG.dipfit.current	Index of the component currently being fitted.
EEG.dipfit.hdmfile	Model file name. Contains informatin about the geometry and the conductivity of the head. This is a standard Matlab file and may be edited as such; for the default spherical model the conductivities and radii are the same as in BESA.
EEG.dipfit.mrifile	Contains the MRI file used for plotting dipole locations.
EEG.dipfit.chanfile	Contains the template channel location file associated with the current head model. This is used for electrode co-registration.
EEG.dipfit.coordformat	Contains the coordinate format in the model structure. This is "spherical" for the spherical model or "MNI" for the BEM model.
EEG.dipfit.coord_transform	Contains the talairach transformation matrix to align the user dataset channel location structure (*EEG.chanlocs*) with the selected head model. This is a length-9 vector *[shiftx shifty shiftz pitch roll yaw scalex scaley scalez]*. Type *>> help traditional* in Matlab for more information.
EEG.dipfit.model	A structure array containing dipole information.
EEG.dipfit.model.posxyz	Contains the 3-D position of the dipole(s). If a two-dipole model is used for a given component, each row of this array contains the location of one dipole.
EEG.dipfit.model.momxyz	Contains 3-D moment of the dipole(s). If a two-dipole model is used for a given component, each row of this array contains the moment of one dipole.
EEG.dipfit.model.rv	Gives the relative residual variance between the dipole potential distribution and th ecomponent potential distribution, ranging from 0 to 1.
EEG.dipfit.model.active	Remembers which dipole was active when the component model was last fitted.
EEG.dipfit.model.select	Remembers which dipole was selected when the component model was last fitted.

The main DIPFIT functions are:

pop_dipfit_settings.m	Specify DIPFIT parameters, i.e. head model details.
pop_dipfit_gridsearch.m	Perform initial coarse DIPFIT grid scan to find an acceptable starting positions for further fine fitting.
pop_dipfit_nonlinear.m	Perform fine model fitting via an interactive interface using non-linear optimization. Note: This function cannot be called from the command line. For automated processing, use the function below.
pop_multifit.m	Command the DIPFIT modeling process from a single function. This function allows the user to set parameters, initialize locations on a coarse grid, then perform non-linear optimization as in the separate functions above. This function can be used to automate batch fitting of components from one or more subjects.

Auxiliary DIPFIT functions used by the functions above:

dipfitdefs.m	defines a set of default values used in DIPFIT.
dipfit_gridsearch.m	same as pop_dipfit_batch.m but does not use a pop-up window and assumes that the input data is an *EEG* dataset structure.
dipfit_nonlinear.m	same as pop_dipfit_manual.m but does not use a pop-up window and assumes that the input data is an *EEG* dataset structure.

Note: There is no auxiliary "dipfit_settings" function, since selecting DIPFIT settings mostly involves copying default values to *EEG.dipfit* fields.

DIPFIT validation study using the spherical head model

We (AD) checked the results of fitting equivalent dipoles with DIPFIT (spherical model) against results of BESA(3.0) for independent components of data recorded in a working memory task in our laboratory. There were 72 channels, of which 69 were used for dipole fitting. In particular, we excluded tow EOG channels and the reference channel. We performed ICA on data from three subjects and fit all 69 resulting components with single dipoles using BESA (see the BESAFIT plug-in Appendix) and DIPFIT. We then compared only those dipoles whose residual variance was less than 15% (see plot below).

Distances between equivalent dipole locations returned by DIPFIT1 and BESA(3.0) were less than 3 mm (left panel) Note: The outliers were infrequent failures to converge by either algorithm. Dipole angle differences were below 2 degrees (middle panel). Residual variances (mismatch between the component scalp map and the model pole projection) were near equal (right panel). A few dipoles were localized with residual variance below 15% by DIPFIT but not by BESA (4 of 213 components); a few others with residual variance below 15% by BESA but not by DIPFIT (8 of 213 components). Optimization nonlinearities may account for these small differences between the two algorithms, as the solution models used in the two programs are theoretically identical.

The main advantage of using DIPFIT over BESA for localization of equivalent dipoles for independent component scalp maps is that it is integrated into Matlab and EEGLAB, and can be used in batch processing scripts (see Matlab code and structures & functions above). BESA has additional capabilities not relevant to DIPFIT. Succeeding versions of BESA did not allow a batch processing option.

Literature references

M. Scherg, "Fundamentals of dipole source potential analysis". In: "Auditory evoked magnetic fields and electric potentials" (eds. F. Grandori, M. Hoke and G.L. Romani). Advances in Audiology, Vol. 6. Karger, Basel, pp 40-69, 1990.

R. Kavanagh, T. M. Darcey, D. Lehmann, and D. H. Fender. "Evaluation of methods for three-dimensional localization of electric sources in the human brain." *IEEE Trans Biomed Eng,* 25:421-429, 1978.

T.F. Oostendorp and A. van Oosterom, "Source parameter estimation in inhomogeneous volume conductors of arbitrary shape", IEEE Trans Biomed Eng. 36:382-91, 1989.

R. Oostenveld and P. Praamstra. The five percent electrode system for high-resolution EEG and ERP measurements. Clin Neurophysiol, 112:713-719, 2001.

A09: Using custom MRI from individual subjects

To visualize dipoles (DIPFIT plugin) or distributed source activity (LORETA plugin) within subject's individual MRI, it is first necessary to normalize them to the MNI brain template. The procedure is relatively simple and is explained below.

Normalizing an MR head image to the MNI brain template

Normalizing an anatomical MR brain image to the standard MNI brain is quite simple and does not require specialized knowledge about MRI or fMRI data. First, download SPM2, uncompress, add path in Matlab. Start spm by typing *>> spm* on the Matlab command line.

Next, click on the *fMRI time series* button. The following menu will appear (along with other screens).

Click on the display button and select your anatomical MR image file by clicking on it. Then press, *Done* as shown below.

A display screen as shown below will appear. Your MR image must be oriented as the one shown below. If it is otherwise oriented, use the edit boxes: *Pitch*, *Roll* and *Yaw* to rotate it in 3-D until it matches the template below. Then press the *Reorient image* button on the bottom.

The next step is to normalize your MRI to the standard MNI brain image. To do this, press the *Normalize* button in the main menu below. A blue tab will appear below the main menu. Select *Determine Parameters & Write Normalized*.

The following file selection window will appear. Select *T1.mnc*, the most common MRI format. (See the SPM manual for other formats). Then press *Done*.

Then reselect MRI data file (the same one you selected originally). Press *Done*.

Another file selection window will appear asking for the name of the file to write. Reselect once more the same MRI data file as the output file. SPM2 will not overwrite it but will use its name. Press *Done*. A final window will appear asking for data for a second subject. Simply press *Done* again. SPM2 will take a few minutes to normalize your MRI to the MNI brain. (It first segments your image and then compares the segmentation to the segmented MNI brain). The output file will have the same name as your original file, but with the prefix "w". You may use this file in the DIPFIT interface as the background image. An example is shown below:

Note that the resolution of this MR image is relatively low and appears noisy because of the renormalization process used in SPM2. If you use other software to perform the normalization, EEGLAB should be able to read the output image using the Fieldtrip function *read_fcdc_mri()* to import various MRI data formats.

Thanks to J-R Duann for helpful advice - Arnaud Delorme, March 2005.

A10: MI-clust

The MI-clust Plug-in: Clustering independent components using mutual information

The infomax ICA algorithm tries to minimize mutual information between components and is greatly successful at this task. Since the activities of cortical and other EEG sources are not perfectly independent, after performing ICA a low level of dependence still exists between the returned maximally independent components(ICs). This residual mutual information can be used to infer relations between different ICs. Figure below shows the scalp maps 30 ICs returned for a sample dataset.

We can use the mi_pairs.m function to calculate the mutual information between the activities (or activations) of these first 30 ICs:

```
>> mutual_info = mi_pairs(EEG.icaact(1:30,:));
```

To visualize the resulting matrix, it is better to first remove the diagonal values (indicating the mutual information between each component and itself), which are quite higher than off-diagonal inter-IC pair values

```
>> mutual_info = mutual_info ? diag(diag(mutual_info));
```

Now we can use a simple Matlab call to visualize this matrix:

```
>> figure; imagesc(mutual_info);
```

giving

As we can see, there is some residual dependency between some of the component activities. For example, ICs 1-4 have a higher dependency on each other than on other components. The scalp maps of these ICs (below) show that they are all related to artifacts generated by subject eye movements, specifically eye blinks.

The *MI_clust* plug-in takes advantage of such residual mutual information to find subsets of maximally independent components whose activities are in some way related to each other. This makes it easier to differentiate various sources of the recorded EEG signal separated by ICA. To use this plug-in, we first need to perform ICA on our dataset. Then select Plot > Cluster dataset ICs

A pop-up dialog window will appear:

Here we need to input three values:

- Components to cluster: These are the indices of the ICs to cluster.
- Number of clusters: Number of different groups into which ICs are to be clustered. For example, we might want to keep at least one cluster per every type of EEG signal source (eye movements, noise subspace, cortical activities,...).
- Clustering method: Currently one can choose either "linkage" (which uses the Matlab *linkage()* function, using the 'ward' method) or "kmeans" (based on the Matlab *kmeans()* function) for clustering. "Linkage" seems to produce better results. An additional dendrogram of IC distance relationships is plotted if the "linkage" option is selected.

After receiving these inputs, the plug-in calculates mutual information between ICs using the mi_pairs.m function (this may take some time...). The function produces 2-D and 3-D IC "locations" whose distances from each other best represent the degree of dependence between each IC pair (closer ICs exhibiting more dependence). This uses the Matlab *mdscale()* function. These locations are clustered by the specified clustering function ("linkage" or "kmeans"). Several figures are plotted :

1. Component Cluster Plots:
   Each IC cluster is plotted in a separate figure. Figure below shows a cluster of vertical eye movement components:

   Another cluster, plotted below, consists of likely brain sources:

The brightness of the background in these plots indicates the confidence level (returned by the Matlab *silhouette()* function) that each IC belongs to this cluster. The figure below shows an example in which three ICs on the top-right corner have a dark background, meaning they are not a strong part of this cluster. (Notice that their scalp maps are quite different as well). In such cases, increasing the number of clusters is recommended. The plotting function interpolates the *silhouette()* values to show a smooth background transition between components.

Cluster 2
File Edit View Insert Tools Desktop Window Help
Cluster 2
29
20
18

2. The Silhouette Plot:
   Silhouette() measures the degree of confidence that a certain IC belongs to the indicated cluster versus other clusters. It is measured by a value between -1 and +1. The plug-in plots *silhouette()* values for all the ICs that belong to the given cluster. From the figure below we can see that the high confidence (positive silhouette value) ICs in cluster 5 (eye component cluster) and low confidence (negative silhouette) for ICs in cluster 2 (above figure). This suggests increasing the number of clusters to provide additional groups for ICs with low silhouette.

3. The 3-D Plot:

This plot shows a 3-D representation in which closer ICs have higher mutual information or activity dependence. Components are colored according to their respective cluster.

4. The Dendrogram Plot:
   If the "linkage" clustering method is selected, an IC dendrogram is plotted:

In this plot, IC are hierarchically joined together (in a binary manner) to form groups with high interdependency. For example, in the above figure, ICs 3 and 4 joint together to form a group that joins the group of ICs 1 and 2 at a higher level to form an \'eye component\' group. The height of the tree at each level is proportional to the distance (independence) between joined group at that level.

A11: BESA (outdated)

Equivalent dipole source localization of independent components using BESA

Component localization by equivalent dipoles: A simple example

Equivalent dipole source localization by BESA (MEGIS Software GmbH, Munich) can be applied to independent component maps under EEGLAB. The following window shows five independent component scalp maps from five different subjects. Each of these components was labelled by the experimenters as representing a left-occipital alpha rhythm generator.

BESA dipole localization applied to these component maps, accounted for each map with a single equivalent dipole in a three-shell spherical model with low residual variance (R.V.) Note that unlike in many BESA analyses, we rarely need to ignore signals at some scalp electrodes since ICA typically isolates artifactual signals into separate independent components, and independent components likely account mainly for activity with a connected synchronous cortical domain whose scalp projection should match that of a single equivalent dipole. (Note: This does not mean, however, that some independent components may be generated by tightly linked bilateral activity in domains joined through e.g. corpus callosum. In this case, the source may often be well modeled by two symmetrically placed equivalent dipoles).

How to localize components using BESA

BESA is a commercial software (www.besa.de) developed by Michael Scherg and colleagues for localizing brain source equivalent dipoles using a three-shell spherical model.

We have developed Matlab functions to automatically export/import and display BESA dipole information. To localize components, you must run BESA 2.x, 3.x, 99 or BESA 2000 Since BESA 99 and 2000 do not support macro files, if you use these versions you must also download BESA 3.0 (the previous version running macros) from the BESA web site (see below) .

BESA 3.0 installation for BESA 2000 and BESA 99 users

1 - Download BESA30 DOS-Window Upgrade from 2.x from the BESA web site (*even if you don't have BESA 2.x*)

2 - Install the hardlock driver for BESA 2000 (on the BESA 2000 CD. Note: you must usually go through the whole installation of BESA 2000 to install the hardlock driver).

3 - Download the BESA30 for BESA2000-Hardlock file and *place it in the BESA 3.0 directory* (if you only have a BESA 99 license, you should download BESA30 for BESA99-EEG-Hardlock or BESA30 for BESA99-MEG+EEG-Hardlock).

4 - Click on the file downloaded in (3) to start BESA3.0

5 - The BESA3.0 manual is not provided on the Internet. We use the BESA 2.2 Manual and find that most of the commands also work in version 3.0.

Install EEGLAB plugin

Then download the BESA plugin for EEGLAB (go to the plugin page, download the BESA plugin, and uncompress it in the EEGLAB plugin folder). EEGLAB BESA functions can also be used with BESA 2.x. However, BESA 2.x cannot import more than 66 channels, and has some restrictions about path lengths under Windows, we strongly recommend the use of BESA 3.0.

Exporting component information

You must first restart EEGLAB with the BESA-function menu included by typing:

```
>> eeglab besa
```

Note that this call will not erase your current dataset(s), but will simply add a new item to the 'Tools' menu: Tools > Localize dipoles using BESA.

After ICA components have been computed, select menu item Tools > Localize dipoles using BESA > Export components to BESA. This calls the function besaexport(). The following window will appear:

In the first edit box, enter the indices of the components to be localized using BESA. The other edit boxes are self explanatory. In case you choose to export the electrode locations from EEGLAB (by leaving blank the 4th edit box), it is very important to check that the polar coordinates have not been shrunk by any other program (e.g., the Fz radius value should be 0.25). If you imported a file with 3-D coordinates into EEGLAB, that should not be an issue. Note that under Windows the last edit box will not appear. After pressing *OK*, the following text will be displayed on the command line:

```
Accessing directory...
Exporting electrode locations ?
Exporting component maps...
Exporting parameter file...
Writing macro file...
NOW RUN BESA, PRESS "A" (for automatic processing) and enter filename "E:\besatmp\macro"
```

-->-->

Now, run BESA on your Windows machine. In BESA, type *A* and then enter the filename given above (*E:\besatmp\macro*). Dipole localization will be performed automatically. Since the number of characters in each BESA macro file is limited, the EEGLAB Matlab function generates many macro files that are called by parent macro files. In this way, up to 900 components can be localized at a time. Also note that minimizing the BESA window will greatly speed up the computation. Occasionally BESA will not optimize the dipole location (e.g., when residual variance remains 98% or more). If this happens, simply rerun the macro corresponding to the badly localized component (for instance **macro10.aut** for component 10).

Importing BESA component locations

Select menu itemTools > Localize dipoles using BESA> Import dipoles from BESA(calling the function besaimport()). The following interactive window will pop up.

In the upper edit box, you must provide the location of the directory in which the dipole information has been saved by BESA. In the lower edit box, you can either (1) import dipoles and erase all previous information (*no* option), (2) append the newly computed information (*yes* option), or (3) only import dipoles for components that do not have yet associated dipoles (*componly* option). Press *OK* and the function will return information about the components that have been imported:

```
Reading component 1 (1 dipoles) residual variance 2.09 %
Reading component 2 (1 dipoles) residual variance 15.67 %
Reading component 3 (1 dipoles) residual variance 1.94 %
Reading component 4 (1 dipoles) residual variance 1.23 %
Reading component 5 (1 dipoles) residual variance 3.49 %
Reading component 6 (1 dipoles) residual variance 12.88 %
```

-->-->

Note: We usually only trust dipole locations with low residual variance (e.g., below 5%, though this figure may vary with number of channels, montage, etc.).

Visualizing dipole locations

There are four menu items that visualize equivalent dipole locations. They all call the same function dipplot() with different options. Here, we illustrate two of these four menu items: First, Tools > Localize dipoles using BESA > Plot dipoles on BESA head.

Here, the dipoles are plotted in 3-D (Note: Use the buttons on the bottom-left to select a specific view). Note that the background images were captured from BESA and the scaling was adjusted manually using visual fitting of a sphere mesh and of the BESA images (press the *Mesh on* button (lower left) to display the mesh sphere). Note that the barlength of the dipole varies with the absolute component amplitude.

We also fit the head sphere model to an average-brain MRI average images (available for download from the ICBM project). MRI images were first shrunk along the X or Y axis to match the 3-D sphere mesh. We then changed the aspect ratio of the axis so that the MRI images'

original aspect ratio was preserved (and the sphere mesh became an ellipsoid). A dipole location summary, plotted on the top of the MRI image, can be shown using menu item Tools > Localize dipoles using BESA > Plot dipoles on BESA head.

Finally, by calling the plotting function from the command line

```
>> dipplot(EEG.sources);
```

many more options can be specified and one or more equivalent dipoles can be plotted individually for each component.

Miscellaneous

Using other dipole localization software: You may use the besaexport() function to export the electrode localization file and the component activations, then try to localize dipoles using some other localization software. If you succeed in doing this, or if you write dedicated functions for this, please contact us.

Localizing dipoles from ERP averages or continuous EEG: Because of the theoretical and practical advantages of using ICA for source localization, we have only implemented equivalent dipole localization for independent components. However, localizing ERP or EEG time series would simply involve exporting the data to BESA. For instance, to export an ERP raw data file that BESA can read, type in:

```
>> ERP = mean(EEG.data,3);
>> save -ascii ERP.raw ERP
```

You may also use the electrode location file generated by besaexport().

A12: Quick Tutorial on Rejection

Quick tutorial on rejecting data artifacts with EEGLAB

Independent Component Analysis is a powerful tool for eliminating several important types of non-brain artifacts from EEG data. EEGLAB allows the user to reject many such artifacts in an efficient and user-friendly manner. This short tutorial is designed to guide impatient users who want to try using EEGLAB to remove artifacts from their data. The many other capabilities of EEGLAB are explained in detail in the main EEGLAB tutorial. To perform artifact rejection:

1) Start Matlab and EEGLAB, then import your data

Type >> eeglab to start EEGLAB under Matlab. Select menu item File > Import data to import your data file in any of a variety of file formats including EGI and Neuroscan binary. See the Import data appendix for more details.

Scroll and check data using menu item Plot > Channel data (scroll).

2) Import a channel location file

Importing a channel location file is critical for visualizing the independent components of your data.

Select menu item Edit > Channel locations and press the button *Read locations* in the bottom right corner of the channel edit window. The program recognizes channel location files in most known formats (spherical BESA, polar Neuroscan, 3-D cartesian EGI, 3-D cartesian Polhemus, ...). Press *OK* after selecting the file and then press *OK* to have EEGLAB recognize the file format automatically from the file extension (Note: files with extension ".elp" are considered Polhemus files by default and not BESA files). Press *OK* in the channel edit window to import the channel locations into EEGLAB.

To check that your channel locations have been imported correctly, use menu item Plot > Channel locations > By name

3) Reject artifact-laden data

The quality of the data is critical for obtaining a good ICA decomposition. ICA can separate out certain types of artifacts -- only those associated with fixed scalp-amp projections. These include eye movements and eye blinks, temporal muscle activity and line noise. ICA may not be used to efficiently reject other types of artifacts -- those associated with a series of one-of-a-kind scalp maps. For example, if the subject were to scratch their EEG cap for several seconds, the result would be a long series of slightly different scalp maps associated with the channel and wire movements, etc. Therefore, such types of "non-stereotyped" or "paroxysmal" noise need to be removed by the user before performing ICA decomposition.

To reject ?noisy channels ?of either continuous or epoched data, select menu item Edit > select data.

To reject noisy portions of ?continuous data?, select menu item Tools > Reject continuous data, then mark noisy portions of continuous data for reject by dragging the mouse horizontally with the left button held down. Press *Reject* when done. A new window pops up to ask for a name for the new dataset.

To reject noisy ?data epochs?, select menu item Tools > Reject data epochs > Reject by inspection. Check the second checkbox to reject data at once (instead of simply marking epochs for rejection) and press *OK*. Then, in the scrolling window, click on data epochs you wish to reject. If you change your mind about a data epoch marked for rejection, click it again to un-select it. Press *Reject* when done. A new window pops up to ask for a name for the new dataset.

EEGLAB also has facilities to automatically suggest data channels, portions and/or epochs to reject. See menu item Tools > Reject data epochs > Reject data (all methods). See the Data rejection tutorial for more details.

4) Run ICA, select and reject artifactual components

Use menu Tools > Run ICA to run the ICA algorithm. To accept the default options, press *OK*.

Use menu Tools > Reject data using ICA > reject component by maps to select artifactual components. See the Data analysis (running ICA) tutorial for more details.

Select menu item Tools > Remove components to actually remove the selected component from the data.

See the Data analysis (running ICA) tutorial for more details and some hints on how to select artifactual components.

5) Further processing of and/or exporting the cleaned data

Your data has now hopefully been pruned of its major artifactual components. You may now proceed with further EEGLAB processing of the remaining non-artifactual independent components (see Data analysis (working with ICA components)).

You may also export your data by selecting menu item File > Export > Data and ICA activity to text file (EEGLAB v4.1). (Note: We believe Neuroscan versions 4.1 and higher can import data files in text format.)

A13: Compiled EEGLAB

As of version 7.1.2.10b, EEGLAB now exist as a compiled binary. Although the compiling function (compile_eeglab.m) should allow to compile EEGLAB on any platform, we only make available the Windows OS (32-bit) binary since this is the most popular platform. We have also successfully compiled EEGLAB under Linux 32 and 64 bits.

Installation

- Download the ZIP file and uncompress it
- Run the SETUP.BAT program and follow instructions.
- If EEGLAB does not start at the end of the instalation process, click on EEGLAB.EXE
- Create shortcuts if necessary on the Desktop and in the "Start" menu.

Similarity between the compiled and the Matlab version

- Both version graphical interface are identical
- There is nothing in the graphic interface that you can do under Matlab, that is not possible to do in the compiled version. This includes using all the plugins and all the external modules attached to EEGLAB, saving scripts and running Matlab scripts.

What is not possible to do using the compiled version

- It is not possible to add new plugins or download and install third party plugins
- When using scripts, is not possible to use external custom Matlab functions or to define new functions
- When using scripts, the range of possible Matlab command usable is limited to the Matlab core commands, all EEGLAB commands and a few commands from the signal processing, the statistics and the optimization toolbox (commands which are included in the compiled code).
- It is not possible to modify EEGLAB source code.

Frequently asked questions

- Can I run some of my own Matlab scripts using the EEGLAB compiled version: yes as long as they use standard Matlab functions or EEGLAB functions.
- Can I compile EEGLAB myself: yes, simply run the *compile_eeglab* script.
- Is the compiled code faster than the non compiled one: no, it is the same speed as it is still interpreted by the Matlab runtime engine.
- What version of Matlab was used to compile the code: under Windows OS, we used Matlab version 2009a.
- Is it legal: yes, it is perfectly legal. Although it is illegal to run cracked version of Matlab, it is perfectly legal to distribute the Matlab Runtime Engine and compiled Matlab code.
- Trouble shooting: do not hesitate to contact us if you encounter a problem running this version of EEGLAB.