**HEDTools CTAGGER: Community Tagging Tools**

**User Guide**

June 3, 2016

Table of Contents

[1. Getting Started with CTAGGER Tools 3](#_Toc452736818)

[1.1 Overview 3](#_Toc452736819)

[1.2 Requirements 3](#_Toc452736820)

[1.3 Installation 3](#_Toc452736821)

[1.3.1 Running with data files that only are .mat files 3](#_Toc452736822)

[1.3.2 Running with .set data file types 3](#_Toc452736823)

[1.3.3 Running as a plugin to EEGLAB 3](#_Toc452736824)

[1.4 Community tagging database 3](#_Toc452736825)

[2. Using the CTAGGER GUI 3](#_Toc452736826)

[2.1. Tagging a single EEGLAB EEG structure 4](#_Toc452736827)

[3. Tagging a directory of datasets 9](#_Toc452736828)

[4. Tagging an EEGLAB study 11](#_Toc452736829)

[5. Extracting data epochs with HED tags 13](#_Toc452736830)

[6. Data Formats 15](#_Toc452736831)

[6.1 XML tag hierarchy (HED) 16](#_Toc452736832)

[6.2 Tags are path strings 16](#_Toc452736833)

[6.3 Field and tag map representations as a MATLAB structure 17](#_Toc452736834)

[6.4 Representing tag maps as a JSON string 18](#_Toc452736835)

[6.5 Representing tag maps as tab-delimited text 19](#_Toc452736836)

[6.6 How tags are stored in a dataset 20](#_Toc452736837)

[6.7 The *fieldMap* object 21](#_Toc452736838)

[6.8 The tagMap object 22](#_Toc452736839)

[6.9 The *tagList* object 23](#_Toc452736840)

[7. Saving tags in the data set (the writetags function) 25](#_Toc452736841)

[8. Running the regression tests and examples 26](#_Toc452736842)

[9. Status and availability 26](#_Toc452736843)

[10. Acknowledgments 26](#_Toc452736844)

[11. References 26](#_Toc452736845)

# 1. Getting Started with CTAGGER Tools

## 1.1 Overview

*CTAGGER (*the tagging part of HEDTools*)* is a MATLAB/Java Toolbox designed to help users annotate events or other data elements using a predefined, but extensible, hierarchically structured annotation language. Although *CTAGGER* can be used in very general contexts, this document describes the MATLAB interface used specifically for tagging events in EEG data. The input to the system consists of two parts: a list of items to be annotated or tagged and an annotation hierarchy. In the case of EEG, users annotate the events that occur during an EEG experiment using the HED 2 hierarchical event description language as the vocabulary. *CTAGGER* allows users to annotate their events quickly and accurately by selecting tags from the HED hierarchy.

## 1.2 Requirements

CTAGGER has been tested with MATLAB 2012a (up to MATLAB 2015a) on systems running either 64-bit Windows or Ubuntu. No other toolboxes are required.

## 1.3 Installation

You can run *CTAGGER* as a standalone toolbox or as a plugin for EEGLAB.

### 1.3.1 Running with data files that only are .mat files

If your data files are *.mat* files, you can simply unzip the *HEDTools* anywhere you choose and set the current directory to the *matlab* directory. Execute the *setup* script to set the paths each time you run MATLAB. Alternatively, you can add the code contained in *setup* to your *startup* script. If you are not using EEGLAB, you can comment out the last section of the *setup* script.

### 1.3.2 Running with .set data file types

If you wish to use EEGLAB, you should follow the directions above without commenting out the last section of the setup script.

### 1.3.3 Running as a plugin to EEGLAB

To use *CTAGGER* as a plugin, unzip *HEDTools1.0.0.zip* under the *EEGLABPlugin* so that the *HEDTools1.0.0* directory is directly under the EEGLAB *plugins* directory. When you bring up EEGLAB, the paths will automatically be set up. *CTAGGER* has three options for tagging from the EEGLAB menus: *Tag current EEG* (from the *Edit* menu), *Tag EEG study* (from the *Tag files* submenu under the *File* menu), or *Tag directory* (from the *Tag files* submenu under the *File* menu).

## 1.4 Community tagging database

In order for *CTAGGER* to behave as a community tagging system, it must be run with the back-end database (*PostrgreSQL)* that stores a common tag hierarchy. **However, the database is currently under development and will not be available for this distribution of *CTAGGER*.**

# 2. Using the CTAGGER GUI

You can launch *CTAGGER* as a stand-alone program from the *ctagger.jar* file located in the *java/tagging* directory or from *MATLAB* through *EEGLAB* using a series of top-level tagging functions (*pop\_tagdir*, *pop\_tageeg*, *pop\_tagstudy*). This guide focuses on using *CTAGGER* through MATLAB. A companion guide discusses the *CTAGGER* GUI in more detail as well as its standalone operation. *CTAGGER* assumes the availability of a hierarchically structured vocabulary for selecting tags.

## 2.1. Tagging a single EEGLAB EEG structure

The *CTAGGER* *tageeg* function allows you to tag a single EEGLAB EEG structure either from a script, the command line with GUI input, or from EEGLAB. When run from EEGLAB, you can tag the current EEG structure through the EEGLAB *Edit* menu. The EEG input data only needs to be a structure with an *.event* subfield and does not have to conform fully to EEGLAB requirements.

The following example illustrates the simplest use of *tageeg* for interactive tagging of an *EEG* structure. The input *EEG* structure can be any structure, but *CTAGGER* extracts field and tag information only from the *EEG.etc.tags*, the *EEG.event*, and the *EEG.urevent* fields. If the data does not have any of these fields, *CTAGGER* does not extract any information.

**Example 2.1:** Tag the data in the *EEG* structure.

[EEG, fMap] = tageeg(EEG)

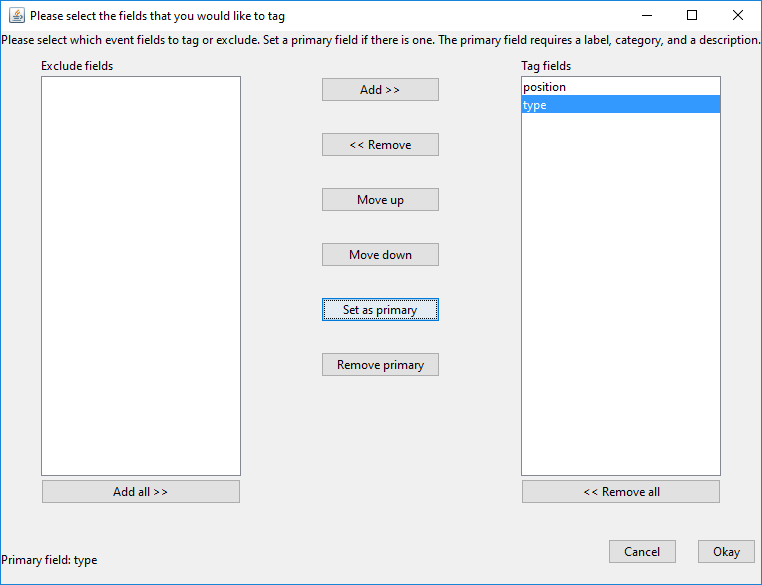
The *tageeg* function extracts any tag information in the *EEG* structure and uses this information to list any existing tags. The user can then tag or update event tags through the GUI that appears. When the user presses the okay button on the GUI, *CTAGGER* returns the *EEG* structure with new tags added in the *EEG.event.usertags* field of the event structure.

*CTAGGER* also adds information in the *EEG.etc.tags* field, which contains a string containing the XML vocabulary used to tag the data as well a *fieldMap* object associating event field names with tags. The latter is used to enable easy editing and modification of the tagging in subsequent sessions. The *fMap* return argument is the *fieldMap* object that contains the tag map information created during this call. Users can pass this map as an optional argument to subsequent calls to *tageeg* to apply tagging created from one dataset to another dataset. Most researchers use the same event codes or labels for all of the datasets in a collection, and this enables them to reuse the tagging performed on one dataset for many others.

*CTAGGER* allows the tagging of different fields in the *EEG.event* structure independently. Normally, you would just tag the unique values that appear in *EEG.event.type* field. However, if you added additional fields to *EEG.event* to better distinguish subcategories, the events are defined by the unique combinations of labels in multiple fields. You can select the fields to tag from a GUI. The GUI contains two lists which consists of fields to exclude and fields to include for tagging. This method requires that your labeling scheme be orthogonal --- that is, it assumes each field can be tagged separately and then the tags from the two fields combined to tag an event.

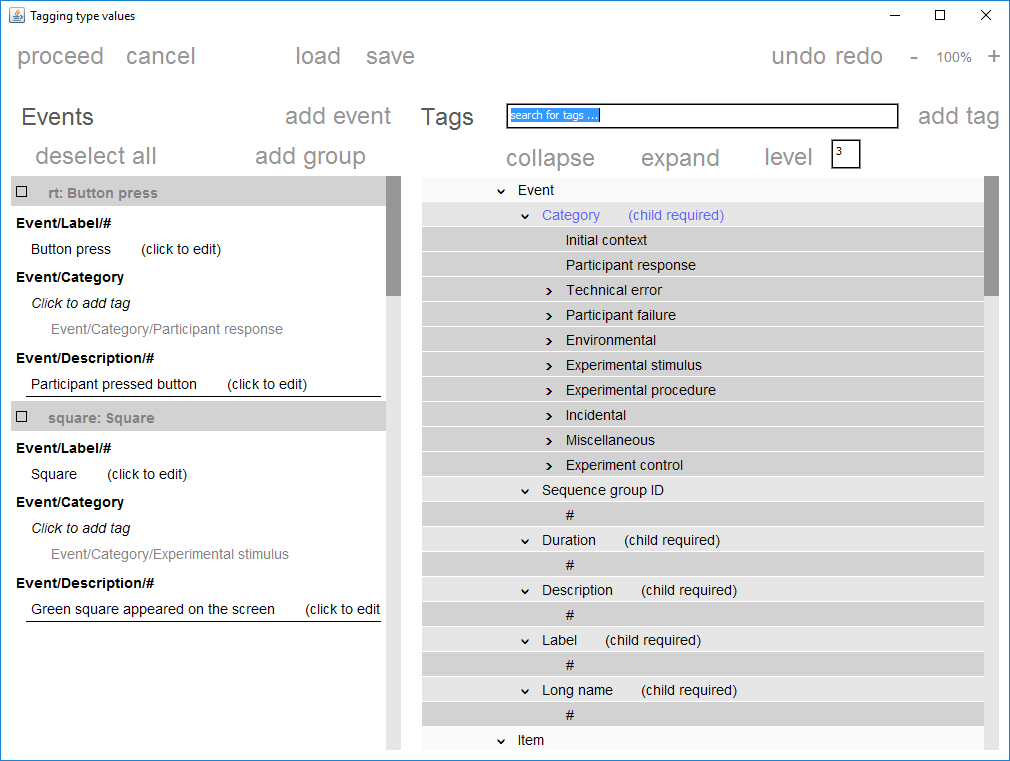
The example dialog in Fig. 2.1 shows the fields that can be selected for tagging. If you press the *Add* button, the *.type* field group will be included in the tag map. If you press the *Remove* button, the *.type* field will not be included. If the *Set as primary* button is pressed then the each value in the *.type* field is required to have a label, category, and description tag. If not pressed then each value is not required to have a label, category, and description tag.

**Figure 2.1:** Example of the field selection menu for selecting which fields to tag.

****

Once you have selected the fields to tag, the *tageeg* function displays a tagging GUI for each selected field as shown in Fig. 2.2 for the *.type* field of Fig. 2.1. After tagging is complete, *tageeg* writes the tag information to the *EEG.etc.tags* and the *EEG.event.usertags* fields of the returned *EEG* structure. The *fMap* return argument is a *fieldMap* object that contains the *tagMap* information created during this call.

**Figure 2.2:** Example of the tagging GUI for the *.type* field or group.

****

In Fig. 2.2, only the type value *square* has been tagged so far. The GUI allows the users to associate tags with each event field value. The key idea is that rather than having to choose tags at random, you select from a menu of potential tags organized in a hierarchical format from general to more specific. You can annotate your event types by choosing tags from this hierarchical menu. You can also modify the hierarchy by adding tags in the hierarchy. Depending on how you call the GUI (checking the *XML can be edited* checkbox), you may also delete or edit existing tags from the tag vocabulary.

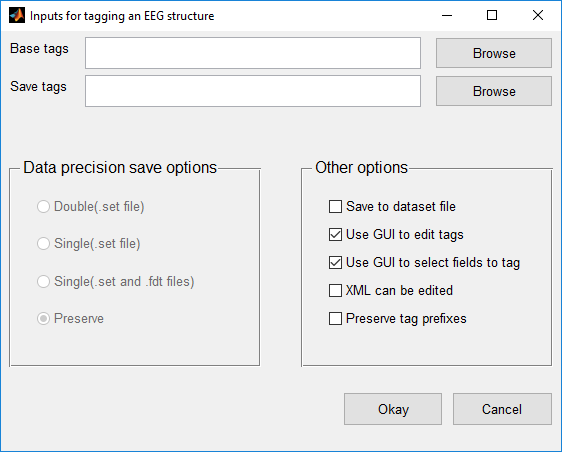
The *tageeg* function has several optional additional arguments specified in name-value pair format. Although these arguments are documented in the function, you can access them through a GUI by calling *pop\_tageeg*.

**Example 2.2:** Tag an *EEG* structure with additional arguments using a menu.

[EEG, com] = pop\_tageeg(EEG);

The *com* return parameter contains the command string that you would have typed to execute *tageeg* without using the menu. You can be save the *com* string for future use if you would like to tag another dataset using the same options without calling the GUI.

**Figure 2.3:** The input settings menu for *tageeg.*



The top section of the menu allows you to browse and select file names for a previously saved *fieldMap* and a location to save the output *fieldMap*. The left box below designates how you would like to write the tagging results back to your *EEG* dataset structure. The right box below allows you to select other options, such as whether you want to use the selection dialog (*Use GUI to select fields to tag)* and tagging GUI for each selected field (*Use GUI to edit tags)*, can the tagging vocabulary be modified (*XML can be edited)*, and will field value tags that share the same prefix all be listed or just the most specific tag starting with that prefix (*Preserve tag prefixes)*.

**Example 2.3:** Tag another dataset without user intervention using a *fieldMap*.

[EEG1, fMap1] = tageeg(EEG, 'BaseMap', fMap, ...

'SelectFields', false, 'UseGui', false);

The *SelectFields* argument controls whether or not *CTAGGER* presents a GUI to select which fields to tag. The *UseGui* argument controls whether or not to use the tagging GUI for each selected field. In Example 2.3, all user intervention is off. The idea here is that you tag one dataset and then call *tageeg* without user intervention to tag related datasets with no additional work.

**MATLAB Syntax**

[EEG, fMap, excluded] = tageeg(EEG)

[EEG, fMap, excluded] = tageeg(EEG, varargin)

**Table 2.1:** A summary of arguments for tageeg.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| EEG | Required | A structure containing data. |
| 'BaseMap' | Name-Value | A *fieldMap* object or the name of a file that contains a *fieldMap* object to be used to initialize tag information. |
| 'EditXml' | Name-Value | If false (default), the HED XML cannot be modified using the tagger GUI. If true, then the HED XML can be modified using the tagger GUI. |
| 'ExcludeFields' | Name-Value | A cell array of field names in the*.etc.tags*, *.event*, and*.urevent* substructures to ignore during the tagging process. By default, *CTAGGER* ignores the following subfields of the .*event* structure: *.latency*, *.epoch*, *.urevent*, *.hedtags*, and *.usertags*. The user can over-ride these tags using this name-value parameter. |
| 'Fields' | Name-Value | A cell array of field names of the fields to include in the tagging. If this parameter is non-empty, *CTAGGER* presents only these fields for tagging, provided they have not been excluded. |
| 'Precision' | Name-Value | The precision that the .data field should be converted to. The options are 'Preserve', 'Double' and 'Single'. 'Preserve' retains the .data field precision, 'Double' converts the .data field to double precision, and 'Single' converts the .data field to single precision. |
| 'PreservePrefix' | Name-Value | If *false* (default), *CTAGGER* combines tags for field values that share prefixes and retains only the most specific (e.g., /a/b/c and /a/b become just /a/b/c). If true, then *CTAGGER* retains all unique tags. |
| 'PrimaryField' | Name-Value | The name of the primary field. Only one field can be the primary field. A primary field requires a label, category, and a description. |
| 'SaveDataset' | Name-Value | If false (default), only save the tags to the EEG structure and not to the underlying dataset file. |
| 'SaveMapFile' | Name-Value | The full path name of the file for saving the final, consolidated *fieldMap* object that results from the tagging process. |
| 'SaveMode' | Name-Value | The options are 'OneFile' and 'TwoFiles'. 'OneFile' saves the EEG structure in a .set file. 'TwoFiles' saves the EEG structure without the data in a .set file and the transposed data in a binary float .fdt file. If the 'Precision' input argument is 'Preserve' then the 'SaveMode' is ignored and the way that the file is already saved will be retained. |
| 'SelectFields' | Name-Value | If *true* (default), the user is presented with a GUI that allow them to select which fields to tag. |
| 'UseGui' | Name-Value | If *true* (default), the tagging GUI is displayed after initialization. |

# 3. Tagging a directory of datasets

The *CTAGGER* *tagdir* function and its supporting functions (*tagdir\_input* and *pop\_tagdir*) allow you to tag an entire directory from a script with no user intervention, from the command line with GUI input, or from EEGLAB. When run from EEGLAB, you can tag a directory through the EEGLAB *Tag data* submenu under the *File* menu without reading any datasets into EEGLAB.

The following example illustrates the simplest use of *tagdir* for interactive tagging of a directory.

**Example 3.1:** Tag the data in the *inDir* directory.

[fMap, fPaths, excluded] = tagdir(inDir);

The *tagdir* function extracts tag information from all of the datasets stored in the *inDir* directory tree and uses this information to list any existing tags. The *fMap* return argument is a *fieldMap* object that contains the tag map information created during this call. The *fPaths* return argument is a cell array containing the full path names of the datasets tagged during this call. The *excluded* argument is a cell array of field names of excluded fields.

*CTAGGER* only considers *.set* datasets. You will then be presented with a series of dialogs, similar to the one of Fig. 2.1, allowing you to decide which fields should be tagged or excluded. Once you have picked the fields to tag, the *tagdir* function displays the tagging GUI (see Fig. 2.2) for each selected field. After tagging, *tagdir* writes the tag information back to the datasets.

You can also call the *tagdir* function with additional arguments like the *tageeg* function. Call the *pop\_tagdir* function to use a menu to set these arguments.

**Example 3.2:** Tag a directory with additional arguments using a menu.

[fMap, fPaths, com] = pop\_tagdir();

**Example 3.3:** Tag a directory without user intervention using the tag map information of Example 3.2.

[fMap1, fPaths, excluded] = tagdir(inDir, 'BaseMap', fMap, ...

'SelectFields', false, 'UseGui', false);

The *SelectOption* controls whether *CTAGGER* presents dialogs to select which fields to tag. The *UseGui* controls whether or not you see a tagging GUI for each selected field. In Example 4.3, all user intervention is off. The idea here is that you tag an entire directory without user intervention once you have created your tag mapping.

**MATLAB Syntax**

[fMap, fPaths, excluded] = tagdir(inDir)

[fMap, fPaths, excluded] = tagdir(indDir, varargin)

**Table 3.1:** A summary of arguments for *tagdir*.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| inDir | Required | The full path name of the directory to be tagged. |
| 'BaseMap' | Name-Value | A *fieldMap* object or the name of a file that contains a *fieldMap* object to be used to initialize the tag information. |
| 'DoSubDirs' | Name-Value | If *true* (default), the *CTAGGER* searches the entire *inDir* directory tree for datasets to tag. If *false*, it only searches the *inDir* directory itself. |
| 'EditXml' | Name-Value | If false (default), the HED XML cannot be modified using the tagger GUI. If true, then the HED XML can be modified using the tagger GUI. |
| 'ExcludeFields' | Name-Value | A cell array of field names in the*.etc.tags*,*.event* and *.urevent* structures to ignore during the tagging process. By default, *CTAGGER* ignores the following subfields of the *.event* structure: *.latency*, *.epoch*, *.urevent*, *.hedtags*, and *.usertags*. The user can override these tags using this name-value parameter. |
| 'Fields' | Name-Value | A cell array of field names of the fields to include in the tagging. If this parameter is non-empty, *CTAGGER* only tags these fields, provided they have not been explicitly excluded. |
| 'Precision' | Name-Value | The precision that the .data field should be converted to. The options are 'Preserve', 'Double' and 'Single'. 'Preserve' retains the .data field precision, 'Double' converts the .data field to double precision, and 'Single' converts the .data field to single precision. |
| 'PreservePrefix' | Name-Value | If *false* (default), *CTAGGER* combines tags of the same field value that share prefixes and retains only the most specific (e.g., /a/b/c and /a/b become just /a/b/c). If true, then *CTAGGER* retains all unique tags. |
| 'PrimaryField' | Name-Value | The name of the primary field. Only one field can be the primary field. A primary field requires a label, category, and a description. |
| 'SaveDatasets' | Name-Value | If true (default), save the tags to the underlying dataset files in the directory. |
| 'SaveMapFile' | Name-Value | The full path name of the file for saving the final, consolidated *fieldMap* object that results from the tagging process. |
| 'SaveMode' | Name-Value | The options are 'OneFile' and 'TwoFiles'. 'OneFile'saves the EEG structure in a .set file. 'TwoFiles'saves the EEG structure without the data in a .set file and the transposed data in a binary float .fdt file. If the 'Precision' input argument is 'Preserve' then the 'SaveMode' is ignored and the way that the file is already saved will be retained. |
| 'SelectFields' | Name-Value | If true (default), the user is presented with a GUI that allow users to select which fields to tag. |
| 'UseGui' | Name-Value | If *true* (default), *CTAGGER* displays the tagging GUI after initialization. |

# 4. Tagging an EEGLAB study

The *CTAGGER* *tagstudy* function and its supporting functions (*tagstudy\_input* and *pop\_tagstudy*) allow you to tag an EEGLAB study from a script with no user intervention, from the command line with GUI input, or from EEGLAB. When run from EEGLAB, you can tag a study through the EEGLAB *File* menu.

The following example illustrates the simplest use of *tagstudy* for interactive tagging of an EEGLAB study.

**Example 4.1:** Tag the data represented by the EEGLAB study specified by the *studyFile* file.

[fMap, fPaths, excluded] = tagstudy(studyFile);

The *tagstudy* function extracts any tag information from the study file or associated datasets and uses this information to list any existing tags. The *fMap* return argument is a *fieldMap* object that contains the tag map information created during this call. The *fPaths* return argument is a cell array containing the full path names of the datasets associated with the study. The *excluded* argument is a cell array of field names of excluded fields.

When you call *tagstudy*, *CTAGGER* presents a series of dialogs, similar to the one of Fig. 2.1. These dialogs allow you to decide which fields to tag or exclude. Once you have picked the fields to tag, the *tagstudy* function displays the tagging GUI (see Fig. 2.2) for each selected field. After you have completed your tagging, *tagstudy* writes the tag information back to the datasets and the study file, depending on the rewrite options that you have selected.

You can also call *tagstudy* function with additional arguments. Call the pop\_*tagstudy* function to use a menu to set these arguments.

**Example 4.2:** Tag a directory with additional arguments using a menu.

[fMap, com] = pop\_tagstudy();

**Example 4.3:** Tag a study without user intervention using tag map information of Example 4.2.

[fMap1, fPaths, excluded] = tagstudy(studyFile, 'BaseMap', fMap, ...

'SelectFields', false, 'UseGui', false);

The *SelectOption* controls whether *CTAGGER* presents dialogs to select which fields to tag. The *UseGui* controls whether or not you see a tagging GUI for each selected field. In Example 4.3, all user intervention is off. The idea here is that you tag a study without user intervention once you have created your tag mapping.

**MATLAB Syntax**

[fMap, fPaths, excluded] = tagstudy(studyFile)

[fMap, fPaths, excluded] = tagstudy(studyFile, varargin)

**Table 4.1:** A summary of arguments for *tagstudy.*

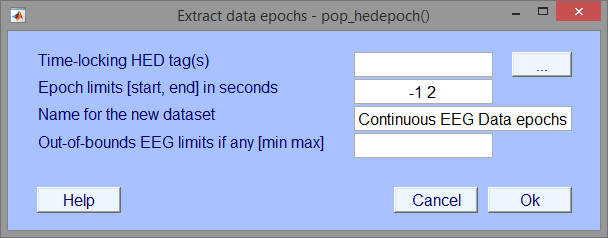
|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| studyFile | Required | The full path name of the study file for study to be tagged. |
| 'BaseMap' | Name-Value | A *fieldMap* object or the name of a file that contains a *fieldMap* object to be used for initial tag information. |
| 'EditXml' | Name-Value | If false (default), the HED XML cannot be modified using the tagger GUI. If true, then the HED XML can be modified using the tagger GUI. |
| 'ExcludeFields' | Name-Value | A cell array of field names in the *.etc.tags*, *.event* and *.urevent* substructures to ignore during the tagging process. By default, *CTAGGER* ignores the following subfields of the *.event* and .*ureven*t structure: *.latency*, *.epoch*, *.urevent*, *.hedtags*, and *.usertags*. You can over-ride these tags using this name-value parameter. |
| 'Fields' | Name-Value | A cell array of field names of the fields to include in the tagging. If this parameter is non-empty, *CTAGGER* only tags these fields, provided they have not been excluded. |
| 'Precision' | Name-Value | The precision that the .data field should be converted to. The options are 'Preserve', 'Double' and 'Single'. 'Preserve' retains the .data field precision, 'Double' converts the .data field to double precision, and 'Single' converts the .data field to single precision. |
| 'PreservePrefix' | Name-Value | If *false* (default), *CTAGGER* combines tags of the same event type that share prefixes and retains only the most specific (e.g., /a/b/c and /a/b become just /a/b/c). If *true*, then *CTAGGER* retains all unique tags. |
| 'PrimaryField' | Name-Value | The name of the primary field. Only one field can be the primary field. A primary field requires a label, category, and a description. |
| 'SaveDatasets' | Name-Value | If true (default), save the tags to the underlying dataset files in the directory. |
| 'SaveMapFile' | Name-Value | The full path name of the file for saving the final, consolidated *fieldMap* object that results from the tagging process. |
| 'SaveMode' | Name-Value | The options are 'OneFile' and 'TwoFiles'. 'OneFile' saves the EEG structure in a .set file. 'TwoFiles' the EEG structure without the data in a .set file and the transposed data in a binary float .fdt file. If the 'Precision' input argument is 'Preserve' then the 'SaveMode' is ignored and the way that the file is already saved will be retained. |
| 'SelectFields' | Name-Value | If true (default), the user is presented with a GUI that allow users to select which fields to tag. |
| 'UseGui' | Name-Value | If *true* (default), CTAGGER displays the tagging GUI after initialization. |

# 5. Extracting data epochs with HED tags

The EEGLAB *pop\_epoch* function extracts data epochs time locked to specified event types. This function works well for datasets that have predefined event types (codes) corresponding to the different stimuluses and participant responses within the experiment. However, some datasets use latency in place of codes to identify the events. This is the case because there are so many different scenarios that can occur within a complex experiment that mirrors the real world. These kind of datasets are generally tagged to describe what transpires throughout each event.

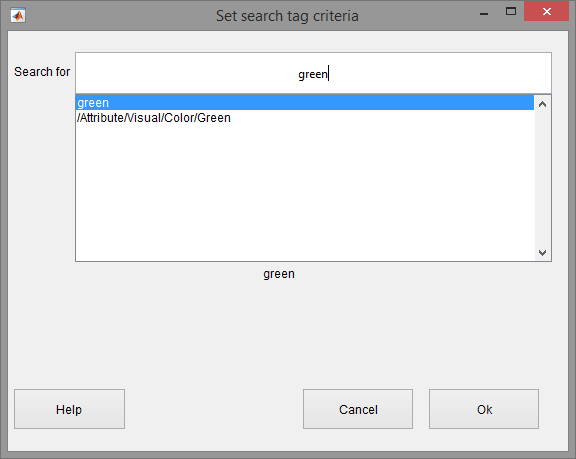
The CTAGGER function *pop\_hedepoch* shown in Fig 5.1 below provides a simple way for extracting data epochs from annotated datasets. The idea here is that instead of extracting epochs based on event type, epochs are extracted based on the HED tags. The HED tags are assumed to be stored as a comma separated string in the *.usertags* field under the *.event* field of the EEG dataset. If the datasets that you are working with are not tagged, then please refer to the sections above in this manual on how to tag the datasets.

**Figure 5.1:** Example of the pop\_hedepoch GUI.



The CTAGGER *pop\_hedepoch* menu in Fig 5.1 is essentially identical to the EEGLAB pop\_epoch menu with the exception of the first input field (*Time-locking HED tag(s)*). Instead of passing in or selecting from a group of unique event types the user passes in a comma separated list of HED tags or a Boolean search string explained in the next section. Clicking the adjacent button (with the label …) will open a new menu used for inputting HED tags shown below in Fig 5.2.

**Figure 5.2:** Example of the hedepoch search bar.



The advanced tag search uses Boolean operators (AND, OR, NOT) to widen or narrow the search. Two tags separated by a comma use the AND operator by default which will only return events that contain both of the tags. The OR operator looks for events that include either one or both tags being specified. The NOT operator looks for events that contain the first tag but not the second tag. To nest or organize the search statements use square brackets. Nesting will change the order in which the search statements are evaluated. For example, "/attribute/visual/color/green AND [/item/2d shape/rectangle/square OR item/2d shape/ellipse/circle]" will look for events that have a green square or a green circle.

When using the search bar and typing in something there will be a listbox below the search bar containing possible matches. Pressing the "up" and "down" arrows on the keyboard while the cursor is in the search bar will move to the next or previous tag in the listbox. Pressing "Enter" will select the current tag in the listbox and it will be added to the search bar. When done click the "Ok" button and it will take you back to the main epoching menu.

The following example illustrates the simplest use of *hedepoch* (called by *pop\_hedepoch*) for extracting data epochs based on HED tags.

**Example 5.1:** Extract data epochs from events with a green square.

EEG = hedepoch(EEG, '/attribute/visual/color/green, /item/2d shape/rectangle/square')

**Example 5.2:** Extract data epochs based on HED tags with menu in Fig 5.1.

[EEG, com] = pop\_hedepoch(EEG)

**MATLAB Syntax**

EEG = hedepoch(EEG, tags)

EEG = hedepoch(EEG, tags, varargin)

**Table 5.1:** A summary of arguments for hedepoch.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| EEG | Required | Input dataset. Data may already be epoched; in this case, extract (shorter) subepochs time locked to epoch events. The dataset is assumed to be tagged and has a .usertags field in the .event structure. |
| tags | Required | A search string consisting of tags to extract data epochs. The tag search uses boolean operators (AND, OR, NOT) to widen or narrow the search. Two tags separated by a comma use the AND operator by default which will only return events that contain both of the tags. The OR operator looks for events that include either one or both tags being specified. The NOT operator looks for events that contain the first tag but not the second tag. To nest or organize the search statements use square brackets. Nesting will change the order in which the search statements are evaluated. For example, "/attribute/visual/color/green AND [/item/2d shape/rectangle/square OR /item/2d shape/ellipse/circle]". |
| 'newname' | Name-Value | New dataset name. The default is "[old\_dataset] epochs". |
| 'timelim' | Name-Value | Epoch latency limits [start end] in seconds relative to the time-locking event. The default is [-1 2]. |
| 'valuelim' | Name-Value | [min max] data limits. If one positive value is given, the opposite value is used for lower bound. For example, use [-50 50] to remove artifactual epoch. The default is [-Inf Inf]. |
| 'verbose' | Name-Value | ['on'|'off']. The default is 'on'. |

# 6. Data Formats

Community tagging is structured and hence requires two items: a tag hierarchy and a map of tags to field or group values. The tag hierarchy is in XML format, and *CTAGGER* provides a schema for validation. The map of tags to field values is in one of four possible formats: a MATLAB structure array, a JSON string, a tab-delimited spreadsheet, or a *tagMap* object. Most of the *CTAGGER* functions use the *tagMap* object representation, which provides methods to convert to and from the other representations.

## 6.1 XML tag hierarchy (HED)

The *CTAGGER* tools assume that rather than inventing tags at random, you will have a menu of suggested tags presented in hierarchical form as shown on the right in Fig. 2.2. Internally, this hierarchy is represented as an XML string.

**Example 6.1:** A snippet from XML representation of the tagging menu displayed in Fig. 2.2.

<?xml version="1.0" encoding="utf-8"?>

<HED version="2.3.1">

<node>

<name>Event</name>

<node position="1" predicateType="passThrough" requireChild="true" required="true">

<name>Category</name>

<description>This is meant to designate the reason

this event was recorded</description>

...

The XML hierarchy shown in Example 6.1 is from *HED2.026.xml* maintained by specifically to support tagging of events in EEG experiments [1]. The Hierarchical Event Descriptor (HED) tags and supporting tools [2][3][4] [5] provide an infrastructure for data mining across data collections, once the datasets have been annotated.

*CTAGGER* works with any XML file that conforms to the *HEDSchema2.01.xsd*, the default XML schema specification. The *HEDSchema2.026.xsd* schema is quite general, and you can substitute any XML hierarchy that conforms to the schema or to build your own hierarchy from the ground up. You must take care in modifying the schema itself, as *CTAGGER* assume certain standard fields. The default XML hierarchy and validating schema are specified by the public constants *DefaultXml* and *DefaultSchema* in the *fieldMap* class defined in *helpers*.

## 6.2 Tags are path strings

Tags are simply path strings from the HED hierarchy. Each path string or tag uses forward slashes (“/”) to separate the components in the path. Commas (“,”) separate multiple tags for the same event. Users may group event tags with one level of parentheses to make the annotation clearer. Example 5.2 shows an example of the annotation for a stimulus event that consists of displaying a red circle in the center of the screen. The parentheses make it clear that the circle is red and located at the center of the screen. If the event designated the display of multiple objects of different colors, the parentheses would make the annotation more clear. The tagging also supports tag groups with embedded tilde (“~”) characters to designate a sentence-like structure. **You can use only one level of parentheses that contains at most two tildes separating the subject from the predicate and the direct object.**

**Example 6.2:** Tag path string representation.

/Event/Category/Experimental stimulus,

(/Item/2D shape/Ellipse/Circle, /Attribute/Visual/Color/Red, /Location/Screen/Center)

Normally, a tag that is more specific (i.e., the added tag has an existing tag as a prefix in string form or corresponds to an ancestor in the tag hierarchy) replaces a less specific tag. However, most *CTAGGER* functions take an optional *PreservePrefix* argument, which is *false* by default. If you set this argument to *true*, *CTAGGER* keeps both tags.

**Example 6.3:** When *PreservePrefix* argument is *true*, *CTAGGER* keeps all versions of the tags.

/Event/Category/Experimental stimulus

/Event/Category/Experimental stimulus/Instruction/Attend

## 6.3 Field and tag map representations as a MATLAB structure

A field map (implemented by the MATLAB *fieldMap* class) associates field names with tag maps (implemented by the MATLAB *tagMap* class). A tag map associates tags with a group of values identified by a name (the “field”). The discussion of this section assumes type/subtype encoding (as illustrated in the next example) to simplify the discussion. However, field maps and tag maps do not rely on a specific representation.

**Example 6.4:** An experiment has two types of events, a stimulus and a user button press response, that are encoded as *STIM*, *RT*, respectively. The stimulus consists of a circle presented in one of three positions: to the left, center, or right of the screen. The positions are encoded by the researcher with numeric codes 1, 2, and 3 respectively. If the dataset is in EEGLAB format, an event such as a circle presented on the left side of the screen at 162 ms after the experiment begins might be stored as a structure:

EEG.event(1) =

type: 'STIM'

stimpos: 1

latency: 162.048

urevent: 1

Only the *.type* and *.stimpos* fields of the *.event* substructure are relevant for tagging. The *.urevent* is an EEGLAB-specific field that relates this event to the original event encodings, while *.latency* specifies the time of this event in frames.

The *CTAGGER* creates a *fieldMap* object to hold the tag map information for each of the two fields or groups: *type* and *stimpos*. The tag map for *type* contains the associations between each of its two values (*STIM* and *RT*) and the corresponding tags.

**Example 6.5:** The structure representation of the field map corresponding to Example 6.4 is:

fMap =

xml: '<?xml version="1.0" ...'

map: [1x2 struct]

Each of the *.map* structures corresponds to a tag map structure as shown in the two next examples.

**Example 6.6:** The structure representation of the tag map *stimpos* corresponding to the field map of Example 6.5:

fMap.map(1) =

field: 'stimpos'

values: [1x3 struct]

fMap.map(1).values(1) =

code: '1'

tags: {'/Item/2D shape/Eclipse/Circle', '/Event/Description/Display of circle on left side of screen'}

fMap.map(1).values(2) =

code: '2'

tags: '/Event/Description/Display of circle in the center of the screen'

fMap.map(1).values(3) =

code: '3'

tags: '/Event/Description/Display of circle on right side of screen'

The *tag map* for the *type* field has the form:

fMap.map(2) =

field: 'type'

values: [1x2 struct]

In summary, a *field map* is a collection of *tag maps*, each identified by a group or field name. Field maps can be represented by a MATLAB structure that has two fields (*.xml*, and *.map*) at the top level. The *.xml* is a string representation of the tag hierarchy used for this tagging. Internally, *CTAGGER* represents a field map by a *fieldMap* object.

A tag map is an association of tags with a group of values identified by a name (field). *CTAGGER* represents tag maps by a MATLAB structure that has two fields (*.field*, and *.values*) at the top level. The *.values* field contains a structure with two fields (*.code* and *.tags*).

## 6.4 Representing tag maps as a JSON string

JSON (JavaScript Object Notation) is a compact, self-annotating data format that allows objects to be marshaled as strings for passing across network connections. Each JSON library converts a JSON string into a different native format. JSON is light-weight representation used for passing tag information between MATLAB and Java. The jsonlab MATLAB library [2] can translate between JSON strings and MATLAB structures. Our JSON representation of field maps and tag maps is the jsonlab translation of the structures described in the previous section.

The tags associated with each event are represented by a 2D array of strings. An array with a single tag is considered an event-level tag, while an array with multiple tags defines a tag group. In Example 6.7*,* the second event listed has two tag groups, and the rest are event-level tags. When using this JSON format to represent tagged events, the tagging GUI expects the corresponding tag hierarchy in XML as described above.

**Example 6.7:** JSON representation of events with tags and tag groups:

[

{

"code" : "1111",

"tags" : [

["/Event/Label/some event 2"],

["/Event/Description/some event 2's description"],

["/Event/Long name/Vehicle | Perturbation | Left | Offset"]

]

},

{

"code" : "1123",

"tags" : [

["/Event/Label/some event 1"],

["/Event/Long name/Environmental | MissionBoundary | DataCollection | Onset"],

["/Event/Description/some event 1's description"],

["/Sensory presentation/Taste"],

[

"/Item/Object/Person/Pedestrian",

"/Item/Object/Person/Mother-child",

"/Item/Object/Food"

],

["/Item/3D shape/Sphere"],

[

"/Item/Object/Animal",

"/Item/Object/Building"

]

]

}

]

We use two jsonlab functions: *loadjson* and *savejson*. The *loadjson* function converts the JSON string to the MATLAB structure described in Section 6.3. The *savejson* function converts the MATLAB structure to JSON.

jStruct = savejson('', jString);

jString = loadjson(jStruct);

## 6.5 Representing tag maps as tab-delimited text

In a typical experiment, researchers often organize their events in a spreadsheet and an XML representation is not convenient or understandable. *CTAGGER* supports loading of events and their annotations in a tab-delimited text format into *a tag map* mentioned in section 6.8 for this purpose. Each row or line of the file represents an event, with the event code or event latency in particular columns and tags optionally in other columns.

Within the tag columns, tags appear as text strings separated by commas. Define a tag group by placing parentheses around a comma-separated list of tags. Example 6.8 shows an example with three events (with lines wrapped to fit). The event codes are 1111, 1121, and 1112, respectively. Event code 1111 indicates the start of a perturbation of a car to the left in a driving experiment.

**Example 6.8:** Tab-delimited text representation of events

Event HED Tags

1111 /Event/Category/ExperimentalStimulus, /Event/Label/LeftPerturbOnset, (/Item/Object/Vehicle/Car, /Attribute/Object control/Perturb, /Attribute/Direction/Left)

1121 /Event/Category/ExperimentalStimulus, /Event/Label/RightPerturbOnset, (/Item/Object/Vehicle/Car, /Attribute/Object control/Perturb, /Attribute/Direction/Right)

1112 /Event/Category/ExperimentalStimulus, /Event/Label/LeftPerturbOffset, (Item/Object/Vehicle/Car, /Attribute/Object control/Perturb, Attribute/Direction/Left)

As mentioned above tab-delimited text format can be converted to a tag map. To do this you must call the *tagtsv* function.

tsvTagMap = tagtsv(filename, fieldname, eventColumn, tagColumns)

The *filename* is the name (including the path) of the tab-delimited file containing the events and their annotations. The *fieldname* argument is the unique field that is associated with the event column values. For Example 6.8 the *fieldname* would be *‘type’* which describes the different types of events in the file. The *eventColumn* and *tagColumns* are the column(s) that contain the events and tag respectively. The *eventColumn* is a single integer value while *tagColumns* can be a single integer value or a vector of integer values. When no tags are present in the file, pass in an empty vector ([]) for the *tagColumns* argument. The output argument is *tsvTagMap* which is a tag mapencapsulating all of the events.

## 6.6 How tags are stored in a dataset

Tags can be stored in any dataset that is a MATLAB structure. *CTAGGER* assumes that the dataset itself is a structure and can store a representation of a field map in the *etc.tags* field of the dataset. One approach is to write the entire structure to the dataset.

**Example 6.10:** Storing the field map structure of Example 6.6 in the dataset *s* as a structure.

s.etc.tags = fMap;

It is also possible to store multiple maps by making *s.etc.tags* a structure array. For datasets that have events represented as a structure with fields, you can store the tags applicable to a particular event.

**Example 6.11:** The tag information stored in the individual event of Example 6.4.

EEG.event(1) =

type: 'STIM'

stimpos: 1

latency: 162.048

urevent: 1

hedtags: ... direct mapped tags as a string

usertags:'/Item/2D shape/Eclipse/Circle'

The tags associated with a *type* value *STIM* as well as of a *stimpos* value *1* are consolidated in *EEG.event(1).usertags* to allow data-mining. *CTAGGER* extracts these tags from a field map that is also maintained to allow revision and remapping. Tags from automated annotation at acquisition are stored in *.hedtags* and are not able to be remapped. The true tags for a particular event consist of the union of the tags in the .*hedtags* and .*usertags* fields.

## 6.7 The *fieldMap* object

The *fieldMap* class manages a collection of named groups and the mappings of their values to tags.

**Example 6.12:** Storing a collection of mappings in a *fieldMap* object.

f = fieldMap();

for k = 1:length(fMap.map)

f.addValues(fMap.map(k).field, fMap.map(k).values, 'Merge');

end

The first statement creates an empty *fieldMap* object using the default XML. The loop adds the individual group mappings to the object. You can create multiple *fieldMap* objects and save them separately from the data. This allows you to maintain multiple tag mappings for different purposes.

**MATLAB Syntax**

fTags = fieldMap()

fTags = fieldMap(varargin)

**Table 6.1:** A summary of arguments for *fieldMap* constructor.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| 'Description' | Name-Value | Description of this object. |
| 'PreservePrefix' | Name-Value | If *false* (default), *CTAGGER* combines tags of the same field value that share prefixes and retains only the most specific (e.g., /a/b/c and /a/b become just /a/b/c). If *true*, then *CTAGGER* retails all unique tags. |
| 'XML' | Name-Value | A string containing the HED tag hierarchy used to create this object. |

**Table 6.2:** A summary of the public methods of the *fieldMap* class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| addValues | Include values in this object based on update type. |
| clone | Create a copy of this object. |
| getDescription | Return the description of this object. |
| getFields | Return the fields of this object. |
| getJson | Return the JSON string version of this object. |
| getJsonValues | Return a JSON array of the JSON of the tag maps for this object. |
| getMap | Return the *tagMap* object associated with a specified field name. |
| getMaps | Return the tag maps for this object as a cell array of *tagMap* objects. |
| getPreservePrefix | Return the *PreservePrefix* flag. |
| getStruct | Return this object as a structure. |
| getTags | Return the tag string associated with value event of field. |
| getValue | Return the value structure corresponding to specified field and key. |
| getValues | Return the values for field as a cell array of structures. |
| getXml | Return a string containing the xml. |
| merge | Combine another *fieldMap* with this object based on update type. |
| mergeXml | Merge an XML string with this object’s *HedXML* if valid. |
| removeMap | Remove the tag maps associated with specified field name. |
| setDescription | Set the description of this object. |

**Table 6.3:** A summary of the public static methods of the *fieldMap* class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| loadFieldMap | Load a field map from a *.mat* file that contains a *fieldMap* object. |
| saveFieldMap | Save a field map to a *.mat* file. |
| validateXml | Validate an XML string given an XML schema (can throw exception). |

## 6.8 The tagMap object

Internally, the *fieldMap* class uses the *tagMap* class to provide a common format for holding the tagging information for one group of values. This class has static methods for translating to and from the other formats and for merging tag maps.

**Example 6.13:** Representation of *fMap.map(1)* of Example 6.10 as a *tagMap* object.

t = tagMap('Field', 'stimpos');

for k = 1:length(fMap.map(1).values)

t.addValues(fMap.map(1).values(k), 'Merge', false);

end

The first statement creates a *tagMap* object representing the tag-value mapping for the group of values called *stimpos*. The second statement adds the actual mapping of tags to values.

**MATLAB Syntax**

tMap = tagMap()

tMap = tagMap(varargin)

**Table 6.4:** A summary of arguments for *tagMap* constructor.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| 'Field' | Name-Value | String identifying the group this map is associated with. |

**Table 6.5:** A summary of the public methods of the *tagMap* class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| addValue | Add the value (a structure) to this object based on update type. |
| clone | Create a copy of this object. |
| getField | Return the field name corresponding to this object. |
| getJson | Return a JSON string version of this object. |
| getJsonValues | Return a JSON string array with JSON for values of this object. |
| getCodes | Return the codes of keys associated with this object. |
| getStruct | Return this object as a structure. |
| getValue | Return the value structure corresponding to specified label or key. |
| getValues | Return the values as a cell array of structures. |
| getValueStruct | Return the values as an array of structures. |
| merge | Combine the *tagMap* object info with this one. |

**Table 6.6:** A summary of the public static methods of the *tagMap* class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| json2Values | Return a structure corresponding to a specified JSON string. |
| values2Json | Return a JSON string representation of a value structure array. |

# **6.9 The *tagList* obje**ct

Similar to how the *fieldMap* class uses the *tagMap* class, the *tagMap* class uses the *tagList* class. The *tagList* class represents each individual value and the associated tags in the *tagMap* group. This class also has static methods for translating to and from the other formats and for merging tag lists.

**Example 6.14:** Create a *tagList* representing a green square that belongs to *tagMap* group type.

tMap = tagMap('Field', 'type');

tList = tagList('square');

tList.add({'/Attribute/Visual/Color/Green','/Item/2D shape/Rectangle/Square'});

tMap. addValue(tList);

**MATLAB Syntax**

tList = tagList(code)

**Table 6.7:** A summary of arguments for *tagList* constructor.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| code | Required | A unique event code value associated with tags. |

**Table 6.8:** A summary of the public methods of the *tagList* class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| add | Add valid tag or tag group to this tagList. |
| addList | Add a list of tags or tag group to this tagList. |
| addString | Add a string of valid tags or tag groups to this tagList. |
| clone | Clone this *tagList* object by making a copy of the tag maps. |
| getCode | Returns the code associated with this tagList. |
| getCount | Returns the number of tags and tag groups in this tagList. |
| getJsonValues | Returns a JSON string version of this *tagList* object. |
| getKeys | Returns the tag map keys for this tagList. |
| getStruct | Returns this *tagList* as a structure array. |
| getTags | Returns a cell array with all of the tags and tag groups in this tagList. |
| intersect | Keep only the tag map keys that are in this *tagList* and in tagList newList. |
| isMember | Returns true if value is a valid tag or tag group in this *tagList*. |
| remove | Remove the tag or tag group in the tag map of this tagList corresponding to value. |
| removePrefixes | Remove the tags in this tagList that are prefixes in existing groups. |
| setCode | Returns the code associated with this tagList. |
| union | Adds the tags given in *tagList newList* to those of this tagList. |

**Table 6.9:** A summary of the public static methods of the *tagList* class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| deStringify | Create a cell array representing a comma-separated string of tags. |
| getCanonical | Returns a sorted version of a valid tag or tag group. |
| removeGroupDuplicates | Removes duplicates from a tag group based on prefix. |
| separateDuplicates | Returns a list of tags without duplicates from cellstr tlist. |
| splitTildesInGroup | Splits the tildes in the cellstr tag group. |
| stringify | Create a string from a cell array of strings or cellstrs. |
| stringifyElement | Create string from cellstr or from string. |
| tagList2Json | Convert a *tagList* to a JSON string. |
| validate | Validate the input as a valid tag or tag group. |
| validateTag | Validate a tag string. |
| validateTagGroup | Validate a cellstr containing a tag group. |

# 7. Saving tags in the data set (the writetags function)

All of *CTAGGER*’s higher-level functions call the *writetags* function to write the tag information to the dataset. *CTAGGER* writes the tags in two different ways: as a summary field map in the *.etc.tags* subfield of the data and as individual event information. In the latter situation, *CTAGGER* assumes that the events to be tagged are in stored in the *.event* structure array and writes a consolidated list of tags based on the actual values of different fields for the *i*th event to the *.event(i).usertags* subfield.

**Example 7.1:** Write the tags encapsulated by the *fieldMap* object *fMap* into the data structure x.

x = writetags(x, fMap);

By default, *CTAGGER* writes both the summary and individual event information, overwriting existing tagging information. If x doesn’t have an *.event* structure, no individual event information is written. The *fMap* object can come from anywhere. Thus, you can have multiple tagging schemes and merge them before writing, or use one at a time. An advantage of keeping the mappings as summaries, separate from the events is that you can edit your tags and rewrite for different uses.

**Example 7.2:** Only write tag information for individual events into the data structure x.

x = writetags(x, fMap, 'RewriteOption', 'Individual');

**MATLAB Syntax**

eData = writetags(eData, fMap)

eData = writetags(eData, fMap, varargin)

**Table 7.1:** Summary of arguments the *writetags* function.

|  |  |  |
| --- | --- | --- |
| **Name** | **Type** | **Description** |
| eData | Required | A dataset structure that tag information is to be written to. |
| fMap | Required | A *fieldMap* object with the tag information. |
| 'ExcludeFields' | Name-Value | A cell array of field names in the *.event* and *.urevent* substructures to ignore during rewrite. |
| 'PreservePrefix' | Name-Value | If *false* (default), *CTAGGER* combines tags of the same event type that share prefixes and only retains the most specific (e.g., /a/b/c and /a/b become just /a/b/c). If *true*, *CTAGGER* retains all unique tags. |

# 8. Running the regression tests and examples

*CTAGGER* uses the XUNIT unit-testing framework for its regression tests located in the *tests* directory. For tests that require user input, the instructions appear in caps in the command window. The regression tests use external data not located in the HEDTools repository. Download the test data archive ([*HEDToolsTestArchive.zip*](http://visual.cs.utsa.edu/software/hedtools/hedtools-1.0.0/hedtools-unit-test-data/view)) and unzip it. You will also need to edit the *tests/setup\_tests.m* file and adjust the *values.testroot* to contain the path of your unzipped archive. *CTAGGER* comes with examples contained in the *tagging\_example.m* script. The examples use external data not located in the HEDTools repository. Download the example data archive ([*HEDToolsExampleArchive.zip*](http://visual.cs.utsa.edu/software/hedtools/hedtools-1.0.0/hedtools-examples-data/view)) and unzip it. You will also need to edit the *exampleDir* that points to the example data archive.

# 9. Status and availability

The base *CTAGGER* is currently available and undergoing user testing. Also, the community tagging database is currently being developed and tested, but will not be available for this release.

# 10. Acknowledgments

The authors acknowledge helpful conversations with Christian Kothe, Nima Bigdely Shamlo, Alejandro Ojeda, Arno Delorme, and Scott Makeig, all of University of California San Diego as well as Scott Kerick, Jeanne Vettel of the Army Research Laboratories, Tony Johnson, Michael Dunkel, and Michael Nonte of DCS Corporation, and Rob Geary and Andrew Moseley-Gholl of the University of Michigan. This research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-10-2-0022. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

# 11. References

[1] N. Bidely-Shamlo, K. Kreutz-Delgado, M. Miyakoshi, M. Westerfield, T. Bel-Bahar, C. Kothe, J. Hsi, S. Makeig, and K. Robbins, “Hierarchical Event Descriptor (HED) Tags for Analysis of Event-Related EEG Studies,” presented at the IEEE GlobalSIP, Austin, TX (submitted), 2013.

[2] “ESS - SCCN.” [Online]. Available: http://sccn.ucsd.edu/wiki/ESS. [Accessed: 19-May-2013].

[3] “Simulation and Neuroscience Application Platform (SNAP).” [Online]. Available: https://github.com/sccn/SNAP. [Accessed: 08-Jun-2013].

[4] “XDF (Extensible Data Format).” [Online]. Available: https://code.google.com/p/xdf/. [Accessed: 08-Jun-2013].

[5] “HED - SCCN.” [Online]. Available: http://sccn.ucsd.edu/wiki/HED. [Accessed: 11-Jun-2013].