CPP BIDS

Release v2.2.0

the CPP BIDS dev team

May 02, 2024

CONTENT

CHAPTER

ONE

SETTING UP YOUR EXPERIMENT

1.1 Configuration

src.**checkCFG**(*cfg*)

Check the fields of the configuration structure `cfg`. If a required field is missing the default value will be assigned to that field. If this field already exists then the existing value will not be replaced.

USAGE:

```
cfg = checkCFG([cfg])
```

Parameters
cfg (`structure`) – The configuration variable to check.

Returns

cfg
(structure)

This function reuses a lot of code and comment from the BIDS starter kit:

https://github.com/bids-standard/bids-starter-kit/tree/master/matlabCode

Fields descriptions:
The following section describes the main fields set by `checkCFG()` with their associated default value.

- `cfg.testingDevice = 'pc'` sets the way the experiment is run and the different options match the imaging modality:
  - `pc` or `beh` is for behavioral test
  - `mri` is for fMRI
  - `eeg` is for EEG…
- `cfg.verbose = 0` sets how talkative the code will be. Possible values range from `0` to `2`.
  - `0`: "I don't want to hear anything from CPP_BIDS."
  - `1`: "I want to get my warnings."
  - `2`: "Tell me everything!"

  For implementation see `utils/talkToMe` and `utils/throwWarning`.
- `cfg.useGUI = false` sets whether a graphic interface should be used for the `userInputs()` to query about group name, as well as for session, subject and run number.

- `cfg.dir.output` sets where the data will be saved.

Filename options:
sets options that will help in creating the filenames.

- `cfg.fileName.task = ''` sets the name to be given to the task
- `cfg.fileName.zeroPadding = 3` sets the amount of 0 padding the subject, session and run number.
- `cfg.fileName.dateFormat = 'yyyymmddHHMM'` sets the format of the date and time stamp that will be appended to all files.

The following fields can be used to specify certain of the labels that are used to specify certain of the acquisition conditions of certain experimental runs in a BIDS data set. These are mostly for MRI and, if set, will be ignored for most other modalities. See `tests/test_createFilename()` for details on how to use these.

- `cfg.suffix.ce = []`
- `cfg.suffix.dir = []`
- `cfg.suffix.rec = []`
- `cfg.suffix.echo = []`
- `cfg.suffix.acq = []`
- `cfg.suffix.recording = []`

Group and session options:
All the fields of `cfg.subject` can be set using the `userInputs()` function but can also be set "manually" directly into the `cfg` structure.

- `cfg.subject.subjectGrp = ''` is set to empty in case no group was provided.
- `cfg.subject.sessionNb = 1` always sets to 1 in case no session was provided.
- `cfg.subject.askGrpSess = [true true]` means that `userInputs()` will always ask for group and session by default.

Eyetracker options:
Those options are mostly work in progress at the moment but should allow to track the some of the metadata regarding eyetracking data acquisition.

- `cfg.eyeTracker.do = false`
- `cfg.eyeTracker.SamplingFrequency = []`
- `cfg.eyeTracker.PupilPositionType = ''`
- `cfg.eyeTracker.RawSamples = []`
- `cfg.eyeTracker.Manufacturer = ''`
- `cfg.eyeTracker.ManufacturersModelName = ''`
- `cfg.eyeTracker.SoftwareVersions = ''`
- `cfg.eyeTracker.CalibrationType = 'HV5'`
- `cfg.eyeTracker.CalibrationPosition = ''`
- `cfg.eyeTracker.CalibrationDistance = ''`
- `cfg.eyeTracker.MaximalCalibrationError = []`

- `cfg.eyeTracker.AverageCalibrationError = []`
- `cfg.eyeTracker.RawDataFilters = {}`

`cfg.bids`:

`checkCFG()` will also initialize `cfg.bids` that contains any information related to a BIDS data set and that will end up in in one of the JSON "sidecar" files containing the metadata of your experiment.

If the content of some fields of `cfg` has been set before running `checkCFG()`, that content might be copied into the relevant field in `cfg.bids`. For example, if you have set the field `cfg.mri.repetitionTime`, then when you run `checkCFG()`, its content will also be copied into `cfg.bids.mri.RepetitionTime`.

`cfg.bids` is further sub-divided into several fields for the different "imaging modalities".

- `cfg.bids.datasetDescription` will be there for all type of experiments
- `cfg.bids.beh` is for purely behavioral experiment with no associated imaging
- `cfg.bids.mri` is for fMRI experiments
- `cfg.bids.eeg` is for EEG experiments
- `cfg.bids.meg` is for MEG experiments
- `cfg.bids.ieeg` is for iEEG experiments

The content of each of those subfields matches the different "keys" one can find in the JSON file for each modality. The content of those different keys is detailed in the code of `checkCFG()`, but a more extensive and updated descriptions will be found in the BIDS specifications themselves.

https://bids-specification.readthedocs.io/en/stable/

For the content of the `datasetDescription.json` files:

```
cfg.bids.datasetDescription.Name = '';
cfg.bids.datasetDescription.BIDSVersion =  '';
cfg.bids.datasetDescription.License = '';
cfg.bids.datasetDescription.Authors = {''};
cfg.bids.datasetDescription.Acknowledgements = '';
cfg.bids.datasetDescription.HowToAcknowledge = '';
cfg.bids.datasetDescription.Funding = {''};
cfg.bids.datasetDescription.ReferencesAndLinks = {''};
cfg.bids.datasetDescription.DatasetDOI = '';
```

For the content of the JSON files for behavioral data:

```
cfg.bids.beh.TaskName = [];
cfg.bids.beh.Instructions = [];
```

For the content of the JSON files for fMRI data:

```
cfg.bids.mri.TaskName = '';
cfg.bids.mri.Instructions = '';
cfg.bids.mri.RepetitionTime = [];
cfg.bids.mri.SliceTiming = '';
cfg.bids.mri.TaskDescription = '';
```

For the content of the JSON files for EEG:

```
cfg.bids.eeg.TaskName = '';
cfg.bids.eeg.Instructions = '';
cfg.bids.eeg.EEGReference = '';
cfg.bids.eeg.SamplingFrequency = [];
cfg.bids.eeg.PowerLineFrequency = 50;
cfg.bids.eeg.SoftwareFilters = 'n/a';
```

For the content of the JSON files for iEEG:

```
cfg.bids.ieeg.TaskName = '';
cfg.bids.ieeg.Instructions = '';
cfg.bids.ieeg.iEEGReference = '';
cfg.bids.ieeg.SamplingFrequency = [];
cfg.bids.ieeg.PowerLineFrequency = 50;
cfg.bids.ieeg.SoftwareFilters = 'n/a';
```

For the content of the JSON files for MEG:

```
cfg.bids.meg.TaskName = '';
cfg.bids.meg.Instructions = '';
cfg.bids.meg.SamplingFrequency = [];
cfg.bids.meg.PowerLineFrequency = [];
cfg.bids.meg.DewarPosition = [];
cfg.bids.meg.SoftwareFilters = [];
cfg.bids.meg.DigitizedLandmarks = [];
cfg.bids.meg.DigitizedHeadPoints = [];
```

src.utils.**transferInfoToBids**(*fieldsToSet*, *cfg*)

Transfers any info that might have been provided by the user in `cfg` to the relevant field of `fieldsToSet` for its reuse later for BIDS filenames or JSON.

USAGE:

```
fieldsToSet = transferInfoToBids(fieldsToSet, cfg)
```

Parameters

- **fieldsToSet** (`structure`) – List of the fields to set. See `checkCFG()`.
- **cfg** (`structure`) – The configuration variable where the user has predefined some fields. See `checkCFG()`.

Returns

fieldsToSet
Updated list of the fields to set.

This can be used for example to make sure that the repetition time set manually in `cfg.mri.repetitionTime` or in `cfg.task.name` will be passed to the correct field in right fields of `cfg.bids`.

1.2 Group, subject, session and run

You can use the `userInputs()` function to easily set the group name as well as the subject, session and run number. You can ask the function to not bother you with group and session

CHAPTER

TWO

FUNCTION DESCRIPTION

List of functions in the `src` folder.

`src.convertSourceToRaw(`*varargin*`)`

Function attempts to convert a source dataset created with CPP_BIDS into a valid BIDS data set.

USAGE:

```
convertSourceToRaw(cfg, 'filter', filter)
```

Parameters

- **`cfg`** (`structure`) – cfg structure is needed only for providing the path in `cfg.dir.output`.
- **`filter`** (`structure`) – bids.query filter to only convert a subset of files.

Output

- **creates**
  a dummy README and CHANGE file
- **copies**
  `source` directory to `raw` directory
- **removes**
  the date suffix `_date-*` from the files where it is present
- **zips**
  the `_stim.tsv` files.

`src.createDataDictionary(`*cfg*, *logFile*`)`

It creates the data dictionary to be associated with a _events.tsv file. It will create empty fields that you can then fill in manually in the JSON file.

USAGE:

```
createDataDictionary(cfg, logFile)
```

Parameters

- **`cfg`** (`structure`) – Configuration. See `checkCFG()`.
- **`logFile`** (`structure`) – Contains the data you want to save.

src.**createDatasetDescription**(*cfg*)

It creates dataset_description.json and writes in every entry contained in cfg.bids.datasetDescription. The file should go in the root of a BIDS dataset.

USAGE:

```
createDatasetDescription(cfg)
```

Parameters
cfg (structure) – Configuration. See checkCFG().

Output

- **dataset_description.json**
  (jsonfile)

src.**createFilename**(*cfg*)

It creates the BIDS compliant directories and fileNames for the behavioral output for this subject / session / run using the information from cfg.

The folder tree will always include a session folder.

Will also create the right fileName for the eyetracking data file. For the moment the date of acquisition is appended to the fileName.

USAGE:

```
[cfg] = createFilename(cfg)
```

Parameters
cfg (structure) – Configuration. See checkCFG().

Returns

cfg
(structure) Configuration update with the name of info about the participants.

The behavior of this function depends on:

- cfg.testingDevice:
  - set to pc (dummy try) or beh can work for behavioral experiment.
  - set on mri for fMRI experiment.
  - set on eeg or ieeg can work for electro encephalography or intracranial eeg
  - set on meg can work for magneto encephalography
- cfg.eyeTracker.do set to true, can work for simple eyetracking data.

See test_createFilename in the tests folder for more details on how to use it.

src.**createJson**(*varargin*)

Creates the side car JSON file for a run.

For JSON sidecars for bold files, this will only contain the minimum BIDS requirement and will likely be less complete than the info you could from a proper BIDS conversion.

USAGE:

```
createJson(cfg [, modality] [, extraInfo])
createJson(cfg [, extraInfo])
```

Parameters

- **cfg** (structure) – Configuration. See checkCFG().
- **modality** (string) – can be any of the following 'beh', 'func', 'eeg', 'ieeg', 'meg') to specify which JSON to save. If it is not provided it will read from cfg.fileName.modality.
- **extraInfo** (structure) – contains information in the JSON file. Beware that the BIDS validator is pretty strict on what information can go in a JSON so this can be useful to store additional information in your source dataset but it might have to be cleaned up to create a valid BIDS dataset.

Output

- ***.json**
  (jsonfile) The file name corresponds to the run + suffix depending on the arguments passed in.

src.**saveCfg**(*varargin*)

Saves config as JSON.

USAGE:

```
saveCfg(cfg [, filename])
```

Parameters

- **cfg** (structure) – Required. Configuration. See checkCFG().
- **filename** (path) – Optional. Fullpath filename for the output file.

Output
filename

If a filename is provided, this will be used as an output file (and will create any required directory).

If no filename is provided, it will try to create one based on the content of cfg.fileName and cfg.dir. This would for example create a file:

```
./output/source/sub-01/func/sub-01_task-testTask_run-001_date-202203181752_cfg.json
```

If this fails it will save the file in the pwd under 'date-yyyymmddHHMM_cfg.json'.

saveCfg can be used to save in a human readable format the extra parameters that you used to run your experiment. This will most likely make the json file non-bids compliant but it can prove useful, to keep this information in your source dataset for when you write your methods sections 2 years later after running the experiment. This ensures that those are the exact parameters you used and you won't have to read them from the setParameters.m file and wonder if those might have been modified when running the experiment and you did not commit and tagged that change with git.

And for the love of the flying spaghetti monster do not save all your parameters in a *.mat* file: think of the case when you won't have Matlab or Octave installed on a computer (plus not everyone uses those).

src.**readAndFilterLogfile**(*columnName*, *filterBy*, *saveOutputTsv*, *varargin*)

It will display in the command window the content of the `output.tsv` filtered by one element of a target column.

USAGE:

```
outputFiltered = readAndFilterLogfile(columnName, filterBy, saveOutputTsv, tsvFile)

outputFiltered = readAndFilterLogfile(columnName, filterBy, saveOutputTsv, cfg)
```

Parameters

- **columnName** (`char`) – the header of the column where the content of interest is stored (for example for `trigger` will be `trial_type`)
- **filterBy** (`char`) – The content of the column you want to filter out. Relies on the `filter` transformer of bids.matlab Supports:
  - >, <, >=, <=, == for numeric values
  - == for string operation (case sensitive)

  For example, `trial_type==trigger` or *onset > 1*.
- **saveOutputTsv** (`boolean`) – flag to save the filtered output in a tsv file
- **tsvFile** (`string`) – TSV file to filter
- **cfg** (`structure`) – Configuration. See `checkCFG()`. If `cfg` is given as input the name of the TSV file to read will be inferred from there.

Returns

outputFiltered
dataset with only the specified content, to see it in the command window use `display(outputFiltered)`.

See also: bids.transformers.filter

src.**saveEventsFile**(*action*, *cfg*, *logFile*)

Function to save output files for events that will be BIDS compliant.

USAGE:

```
[logFile] = saveEventsFile(action, cfg, logFile)
```

Parameters

- **action** (`string`) – Defines the operation to do. The different possibilities are `'init'`, `'init_stim'`, `'open'`, `'save'` or `'close'`. For more information on each case see below.
- **cfg** (`structure`) – Configuration variable. See `checkCFG()`.
- **logFile** (`structure`) – (n x 1) The `logFile` variable that contains the n events you want to save must be a nx1 structure.

See `tests/test_saveEventsFile()` for more details on how to use it.

Example:

```
logFile(1,1).onset = 2;
logFile(1,1).trial_type = 'motion_up';
logFile(1,1).duration = 1;
```

Actions:

Example:

```
logFile = saveEventsFile('init', cfg, logFile)
logFile = saveEventsFile('init_stim', cfg, logFile)
logFile = saveEventsFile('open', cfg, logFile)
logFile = saveEventsFile('save', cfg, logFile)
```

- `'init'` and `'init_stim'` are used for events and stimuli tsv files respectively. This initializes the extra columns to be save.
- `'open'` will create the file ID and return it in `logFile.fileID` using the information in the `cfg` structure. This file ID is then reused when calling that function to save data into this file. This creates the header with the obligatory `'onset'`, `'duration'` required by BIDS and other columns can be specified in varargin.

  Example:

  ```
  logFile = saveEventsFile('open', cfg, logFile);
  ```

- `'save'` will save the data contained in logfile by using the file ID `logFile.fileID`. If saving a stimulus file then the only the fields of `logFile.extraColumns` will be saved. For regular _events.tsv files, then `logFile` must then contain:
  - logFile.onset
  - logFile.duration

  Otherwise it will be skipped.

  Example:

  ```
  logFile = saveEventsFile('save', cfg, logFile);
  ```

- `'close'` closes the file with file ID `logFile.fileID`. If `cfg.verbose` is superior to 1 then this will tell you where the file is located.

  Example:

  ```
  logFile = saveEventsFile('close', cfg, logFile)
  ```

`src.`**`userInputs`**`(`*`cfg`*`)`

Get subject, run and session number and make sure they are positive integer values. Can do a graphic user interface if `cfg.useGUI` is set to `true`

USAGE:

```
cfg = userInputs(cfg)
```

Parameters

`cfg` (`structure`) – Configuration. See `checkCFG()`.

Returns

cfg
(structure) Configuration update with the name of info about the participants.

Behavior of this functions depends on `cfg.subject.askGrpSess` a 1 X 2 array of booleans (default is `[true true]`):

- the first value set to `false` will skip asking for the participants group
- the second value set to `false` will skip asking for the session

CHAPTER

THREE

UTILITY FUNCTIONS

List of functions in the `utils` folder.

---

src.utils.**checkInput**(*data*)

Check the data to write and convert to the proper format if needed.

Default will be 'n/a' for chars and NaN for numeric data.

USAGE:

```
data = checkInput(data)
```

src.utils.**getFullFilename**(*fileName*, *cfg*)

Returns the full path of a file (for a given subject and modality in a run).

USAGE:

```
fullFilename = getFullFilename(fileName, cfg)
```

Parameters

- **fileName** (string)
- **cfg** (structure) – Configuration. See `checkCFG()`.

src.utils.**initializeExtraColumns**(*logFile*)

Initialize the fields for the extra columns

USAGE:

```
logFile = initializeExtraColumns(logFile)
```

Parameters

logFile (structure) – It contains what to save in the experiment outputs.

Returns

logFile
(structure) `logfile` updated with extra columns.

Example:

```
logfile.extraColumns{'Speed', 'Response key'}
logFile = initializeExtraColumns(logFile)
```

src.utils.**isPositiveInteger**(*input2check*)

It checks whether the input is a positive integer and report it as `true` or `false`

USAGE:

```
trueOrFalse = isPositiveInteger(input2check)
```

Parameters
input2check (`vector`) – (1xn) The input to check (either a number or `NaN`)

Returns

trueOrFalse
(boolean)

src.utils.**nanPadding**(*cfg*, *data*, *expectedLength*)

For numeric data that don't have the expected length, it will be padded with NaNs. If the vector is too long it will be truncated

USAGE:

```
data = nanPadding(cfg, data, expectedLength)
```

src.utils.**printCreditsCppBids**(*cfg*)

It will print the credits of this repo. Depending on the level of verbosity set in `cfg.verbose` (default is 2 if not set), it will print the graphic and general information.

USAGE:

```
printCreditsCppBids(cfg)
```

Parameters
cfg (`structure`) – Configuration. See `checkCFG()`.

src.utils.**returnHeaderName**(*columnName*, *nbCol*, *iCol*)

It returns one by one the column name to be used as a header in a recently opened event file

USAGE:

```
headerName = returnHeaderName(columnName, nbCol, iCol)
```

Parameters

- **columnName** (`string`) – The column name to print
- **nbCol** (`integer`) – It is the number of columns associated to one entry of the extra column list
- **iCol** (`integer`) – Index of the columns associated to one entry of the extra column list

Returns

- **headerName**
  (string) return the extra column name to be used as header

src.utils.**returnNamesExtraColumns**(*logFile*)

It returns the extra columns name(s), in `cfg.extraColumns`, as header to add to the `event` file

USAGE:

```
[namesExtraColumns] = returnNamesExtraColumns(logFile)
```

Parameters

logFile (`structure`) – It contains all the information to be saved in the event/stim file

Returns

- **namesExtraColumns**
  (cell) (nx1)

src.utils.**returnNbColumns**(*logFile*, *nameExtraColumn*)

It returns the number of columns associated to one entry of the extra column list.

USAGE:

```
[nbCol] = returnNbColumns(logFile, nameExtraColumn)
```

Parameters

- **logFile** (`structure`) – It contains every information related to the experiment output(s)
- **nameExtraColumn** (`string`) – An entry of `logFile.extraColumns`

Returns

- **nbCol**
  (integer) The number of columns associated to one entry of the extra column list.

src.utils.**setDefaultFields**(*structure*, *fieldsToSet*)

Recursively loop through the fields of a structure and sets a value if they don't exist.

USAGE:

```
structure = setDefaults(structure, fieldsToSet)
```

Parameters

- **structure** (`structure`)
- **fieldsToSet** (`structure`)

Returns

- **structure**
  (structure)

CHAPTER

FOUR

USER INTERFACE

List of functions in the `ui` folder: those are mostly to handle the "graphic interface" that can be used to deal with `userInputs()`.

`src.ui.`**`askUserCli`**`(`*items*`)`

It shows the questions to ask in the command window and checks, when it is necessary, if the given input by the user is an integer.

USAGE:

```
items = askUserCli(items)
```

Parameters

`items` (`structure`) – It contains the questions list to ask and if the response given to one question must be checked to be a positive integer.

EXAMPLE:

```
items = returnDefaultQuestionnaire();
items = askUserCli(items);
```

See also: createQuestionnaire

`src.ui.`**`askUserGui`**`(`*items*`)`

It shows the questions to ask in in a GUI interface and checks, when it is necessary, if the given input by the user is a positive integer. If not, it keeps showing the GUI interface.

USAGE:

```
[responses] = askUserGui(questions, responses)
```

Parameters

- **`questions`** (`structure`) – It contains the questions list to ask and if the response given to one question must be checked to be an integer number.
- **`responses`** (`cell`) – It contains the responses set by default.

Returns

- **responses**
  (cell) Response updated with the user inputs.

src.ui.**createQuestionnaire**(*cfg*)

It creates a list of default questions to ask the users regarding:

- the subj number
- the run number
- the group ID (if required by user)
- and session nb (if required by user)

USAGE:

```
[items, cfg] = createQuestionnaire(cfg)
```

Parameters
cfg (structure) – Configuration. See checkCFG().

Returns

items
(structure) It contains the questions list to ask and if the response given to one question must be checked to be a positive integer.

EXAMPLE

items(1).question = 'Enter subject number (1-999): '; items(1).response = ''; items(1).mustBePosInt = true; items(1).show = true;

See also: returnDefaultQuestionnaire

src.ui.**askForGroupAndOrSession**(*cfg*)

It checks cfg if group, session, run are required in cfg.subject.ask by the user. If not specified, it will add these by default.

USAGE:

```
[cfg] = askForGroupAndOrSession(cfg)
```

Parameters
cfg (structure) – Configuration. See checkCFG().

Returns

cfg
(structure) Configuration update with the instructions if to ask for group and session.

src.ui.**getIsQuestionToAsk**(*questions*, *responses*)

While using the GUI interface to input the experiment information, it flags any question that will be presented in the GUI. If a response is not valid (e.g. is not an integer) it will keep flagging it as a 'question to ask' and represent the GUI.

USAGE:

```
isQuestionToAsk = getIsQuestionToAsk(questions, responses)
```

Parameters

- **questions** (structure) – It contains the questions list to ask and if the response given to one question must be checked to be an integer.
- **responses** (cell) – It contains the responses set by default or as input by the user

Returns

- **argout1**
  (type) (dimension)
- **argout2**
  (type) (dimension)

A set of function for matlab and octave to create BIDS-compatible structure and filenames for the output of behavioral, EEG, fMRI, eyetracking experiments.

CHAPTER

FIVE

OUTPUT FORMAT

5.1 Modality agnostic aspect

The files created by this toolbox will always follow the following pattern:

```
dataDir/sub-<[Group]SubNb>/ses-sesNb/sub-<[Group]SubNb>_ses-<sesNb>_task-<taskName>*_
↪modality_date-*.fileExtension
```

Subjects, session and run number labels will be numbers with zero padding up (default is set to 3, meaning that subject 1 will become `sub-001`).

The `Group` name is optional.

A session folder will ALWAYS be created even if not requested (default will be `ses-001`).

Task labels will be printed in `camelCase` in the filenames.

Time stamps are added directly in the filename by adding a suffix `_date-*` (default format is `YYYYMMDDHHMM`) which makes the file name non-BIDS compliant. This was added to prevent overwriting files in case a certain run needs to be done a second time because of a crash. Some of us are paranoid about keeping even cancelled runs during my experiments. This suffix should be removed to make the data set BIDS compliant. See `convertSourceToRaw()` for more details.

For example:

```
/user/bob/dataset002/sub-090/ses-003/sub-090_ses-003_task-auditoryTask_run-023_events_
↪date-202007291536.tsv
```

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

MATLAB MODULE INDEX

S

INDEX