CPP_PTB

Release v1.2.1dev

CPP_PTB developers

Apr 09, 2024

CONTENTS:

This is the Crossmodal Perception and Plasticity lab (CPP) PsychToolBox (PTB) toolbox.

Those functions are mostly wrappers around some PTB functions to facilitate their use and their reuse (#DontRepeatYourself)

CHAPTER

ONE

INSTALLATION

1.1 Requirements

Make sure that the following toolboxes are installed and added to the matlab / octave path.

For instructions see the following links:

Requirements	Used version
PsychToolBox	>=3.0.14
Matlab	>=2015b
or Octave	4.?

Tested:

- matlab 2015b or octave 4.2.2 and PTB 3.0.14.

1.2 How to install

The easiest way to use this repository is to create a new repository by using the template PTB experiment repository: this creates a new repository on your github account with all the basic folders, files and submodules already set up. You only have to then clone the repository and you are good to go.

1.2.1 Download with git

```
cd fullpath_to_directory_where_to_install

# use git to download the code
git clone https://github.com/cpp-lln-lab/CPP_PTB.git

# move into the folder you have just created
cd CPP_PTB
```

Then get the latest commit to stay up to date:

```
# from the directory where you downloaded the code
git pull origin master
```

To work with a specific version, create a branch at a specific version tag number

```
# creating and checking out a branch that will be called version1 at the version tag v1.0.0
git checkout -b version1 v1.0.0
```

1.2.2 Add as a submodule

Add it as a submodule in the repo you are working on.

```
cd fullpath_to_directory_where_to_install

# use git to download the code
git submodule add https://github.com/cpp-lln-lab/CPP_PTB.git
```

To get the latest commit you then need to update the submodule with the information on its remote repository and then merge those locally.

```
git submodule update --remote --merge
```

Remember that updates to submodules need to be committed as well.

Example for submodule usage

So say you want to clone a repo that has some nested submodules, then you would type this to get the content of all the submodules at once (here with my experiment repo):

```
git clone --recurse-submodules https://github.com/user_name/yourExperiment.git
```

This would be the way to do it "by hand"

```
# clone the repo
git clone https://github.com/user_name/yourExperiment.git

# go into the directory
cd yourExperiment

# initialize and get the content of the first level of submodules  (CPP_PTB and CPP_BIDS)
git submodule init
git submodule update

# get the nested submodules JSONio and BIDS-matlab for CPP_BIDS
git submodule foreach --recursive 'git submodule init'
git submodule foreach --recursive 'git submodule update'
```

1.2.3 Direct download

Download the code. Unzip. And add to the matlab path.

Pick a specific version from here.

Or take the latest commit - NOT RECOMMENDED.

1.2.4 Add CPP_PTB globally to the matlab path

This is NOT RECOMMENDED as this might create conflicts if you use different versions of CPP_PTB as sub-modules.

Also note that this might not work at all if you have not set a command line alias to start Matlab from a terminal window by just typing *matlab*. :wink:

```
# from within the CPP_PTB folder
matlab -nojvm -nosplash -r "addpath(genpath(fullfile(pwd, 'src'))); savepath(); path();␣
↪exit();"
```

CHAPTER

TWO

SET UP

2.1 The CFG structure

The `cfg` structure is where most of the information about your experiment will be defined.

Below we try to outline what it contains.

Some of those fields you can set yourself while some others will be created and filled after running `setDefaultsPTB.m` and `initPTB.m`.

- `setDefaultsPTB.m` sets some default values for things about your experiment that that do not "depend" on your system or that PTB cannot "know". For example the width of the screen in cm or the dimensions of the fixation cross you want to use…
- `initPTB.m` will fill in the fields that ARE system dependent like the screen refresh rate, the reference of the window that PTB opened and where to flip stimulus to. When it runs, `initPTB.m` will call `setDefaultsPTB.m` to make sure that all the required fields are non-empty.

If no value is provided below, it means that there is no set default (or that the *initPTB* takes care of it).

2.1.1 Fields set `setDefaultsPTB`

```
cfg.testingDevice = 'pc';
```

Other options include:

- `'mri'`
- `'eeg'`
- `'meg'`

cfg.keyboard

```
cfg.keyboard.keyboard = [];         % device index for the main keyboard
                                      % (that of the experimenter)
cfg.keyboard.responseBox = [];      % device index used by the participants
cfg.keyboard.responseKey = {};      % list the keys that PTB should "listen" to when
                                      % using KbQueue to collect responses ;
                                      % if empty PTB will listen to all key presses
cfg.keyboard.escapeKey = 'ESCAPE';  % key to press to escape
```

cfg.debug

```
cfg.debug.do = true;         % if true this will make less PTB tolerant with
                               % bad synchronisation
cfg.debug.transpWin = true; % makes the stimulus windows semi-transparent:
                               % useful when designing your experiment
cfg.debug.smallWin = true;  % open a small window and not a full screen window ;
                               % can be useful for debugging
```

cfg.text

```
cfg.text.font = 'Courier New';
cfg.text.size = 18;
cfg.text.style = 1; % bold
```

cfg.color

```
cfg.color.background = [0 0 0]; % [r g b] each in 0-255
```

cfg.screen

```
cfg.screen.monitorWidth = 42;       % in cm
cfg.screen.monitorDistance = 134;   % in cm
cfg.screen.resolution = {[], [], []};
```

cfg.fixation

```
cfg.fixation.type = 'cross';    % can also be 'dot' or 'bestFixation'
cfg.fixation.xDisplacement = 0; % horizontal offset from window center
cfg.fixation.yDisplacement = 0; % vertical offset from window center
cfg.fixation.color = [255 255 255];
cfg.fixation.width = 1;         % in degrees of visual angle
cfg.fixation.lineWidthPix = 5;  % width of the lines in pixel
```

cfg.aperture

Mostly relevant for retinotopy scripts but can be reused for other types of experiments where an aperture is needed.

```
cfg.aperture.type = 'none';
```

Other options include:

- 'bar'
- 'wedge'
- 'ring'
- 'circle'

cfg.audio

Check the `scripts/ptbSoundDeviceTest.m` to help you figure out what devices are connected to the computer and which one you can use.

```
cfg.audio.do = false;            % set to true if you are going to play some sounds
cfg.audio.requestedLatency = 3;
cfg.audio.fs 44100;              % sampling frequency
cfg.audio.channels = 2;          % number of auditory channels
cfg.audio.initVolume = 1;
cfg.audio.repeat = 1;
cfg.audio.startCue = 0;
cfg.audio.waitForDevice = 1;
```

2.1.2 Fields set by `initPTB`

cfg.screen

```
cfg.screen.idx          % screen index
cfg.screen.win          % window index
cfg.screen.winRect      % rectangle definition of the window
cfg.screen.winWidth
cfg.screen.winHeight
cfg.screen.center       % [x y] ; pixel coordinate of the window center
cfg.screen.FOV          % width of the field of view in degrees of visual angle
cfg.screen.ppd          % pixel per degree
cfg.screen.ifi          % inter frame interval
cfg.screen.monRefresh   % monitor refresh rate ; 1 / ifi
```

cfg.audio

```
cfg.audio.requestOffsetTime = 1;
cfg.audio.reqsSampleOffset
cfg.audio.pushSize
cfg.audio.playbackMode = 1;
cfg.audio.devIdx = [];
cfg.audio.pahandle
```

operating system information

```
cfg.software.os
cfg.software.name = 'Psychtoolbox';
cfg.software.RRID = 'SCR_002881';
cfg.software.version % psychtoolbox version
cfg.software.runsOn % matlab or octave and version number
```

2.2 Setting up keyboards

To select a specific keyboard to be used by the experimenter or the participant, you need to know the value assigned by PTB to each keyboard device.

To know this copy-paste this on the command window:

```
[keyboardNumbers, keyboardNames] = GetKeyboardIndices;

disp(keyboardNumbers);
disp(keyboardNames);
```

You can then assign a specific device number to the main keyboard or the response box in the `cfg` structure:

- `cfg.keyboard.responseBox` would be the device number of the device used by the participant to give his/her response: like the button box in the scanner or a separate keyboard for a behavioral experiment
- `cfg.keyboard.keyboard` would be the device number of the keyboard on which the experimenter will type or press the keys necessary to start or abort the experiment.

`cfg.keyboard.responseBox` and `cfg.keyboard.keyboard` can be different or the same.

Using empty vectors (like `[]`) or a negative value for those means that you will let PTB find and use the default device.

CHAPTER

THREE

FUNCTION DESCRIPTION

List of functions in the `src` folder.

`src.`**`initPTB`**`(`*`cfg`*`)`

This will initialize PsychToolBox:

- screen
  - the windon opened takes the whole screen unless `cfg.debug.smallWin` is set to `true`
  - can skip synch test if you ask for it (nicely)
  - window transparency enabled by `cfg.debug.transpWin` set to `true`
  - gets the flip interval
  - computes the pixel per degree of visual angle: the computation for ppd assumes the windows takes the whole screen width
- set font details
- keyboard
- hides cursor
- sound

USAGE:

```
cfg = initPTB(cfg)
```

See the set up page of the documentation for more details on the content of cfg

`src.`**`drawFieldOfVIew`**`(`*`cfg`*`, `*`centerOnScreen`*`)`

Draws a red rectangle on the screen to materialize the field of view of the participant. This can be used during debugging to help design the stimuli if you know the FOV of the participant will be obstructed by something

USAGE:

```
fov = drawFieldOfVIew(cfg, centerOnScreen)
```

Parameters

- **`cfg`** (`structure`) –
- **`centerOnScreen`** (`boolean`) –

Returns

- **fov**
  (array) PTB rectangle

src.**eyeTracker**(*input*, *cfg*, *varargin*)

Wrapper function that deals with all the necessary actions to implement Eye Tracker recording with eyelink.

USAGE:

function [el, cfg] = eyeTracker(input, cfg, [message])

Parameters

- **input** (`string`) – Defines what we want the function to do
- **cfg** (`struct`) – structure that stores any info regarding the experiment
- **message** (`string`) – optional argument to pass in when you want to tag the output in a specific moment of the experiment (for example `Experiment-start`)

Returns

- **el**
  (struct) stores info related to the Eye Tracker
- **cfg**
  (struct)

- `Calibration` to initialize EyeLink and run calibration
  - `default calibration` (default) will run a calibration with 6 points
  - `custom calibration` (`cfg.eyeTracker.defaultCalibration = 'false'`) will run a calibration with 6 points but the experimenter can choose their position on the screen
- `StartRecording`: to start eye movements recording
- `Message`: will add a tag (e.g. `Block_n1`) in the ET output file, the tag is a string and it is input from *varargin*
- `StopRecordings`: to stop eye movements recording
- `Shutdown`: to save the `.edf` file with BIDS compliant name, from cpp-lln-lab/CPP_BIDS, in the output folder and shut the connection between the stimulation computer and the EyeLink computer

src.**getExperimentEnd**(*cfg*)

Wrapper function that will show a fixation cross and display in the console the whole experiment's duration in minutes and seconds

src.**getExperimentStart**(*cfg*)

Wrapper function that will show a fixation cross and collect a start timestamp in `cfg.experimentStart`

USAGE:

```
cfg = getExperimentStart(cfg)
```

src.**isOctave**()

Return: true if the environment is Octave. mostly used to testing when PTB is not in the path

Must stay in the 'src' folder for continuous integration with github action to work. Not sure why.

src.**waitForTrigger**(*varargin*)

Counts a certain number of triggers coming from the scanner before returning.

USAGE:

[lastTriggerTimeStamp] = waitForTrigger([cfg,] ...
[deviceNumber,] ... [quietMode,] ... [nbTriggersToWait])

Parameters

- **cfg** (struct) –
- **deviceNumber** (integer) – device number of the keyboard or trigger box in MRI
- **quietMode** (boolean) – a boolean to make sure nothing is printed on the screen or the prompt
- **nbTriggersToWait** (integer) – number of triggers to wait

Returns

- **lastTriggerTimeStamp**
  (optional) it can be used as experimentStart timestamp (cfg.experimentStart)

If you are not using the quietMode, it flips and waits for half a TR before starting to check for the next trigger (unless this was the last trigger to wait for and in this case it returns immediately).

Will print the count down in the command line and on the PTB window if one is opened.

If the fMRI sequence RT is provided (cgf.MRI.repetitionTime) then it will wait for half a RT before starting to check for next trigger, otherwise it will wait 500 ms.

When no deviceNumber is set then it will check the default device: this is probably only useful in debug as you will want to make sure you get the triggers coming from the scanner in a real case scenario.

src.**waitFor**(*cfg*, *timeToWait*)

Will either wait for a certain amount of time or a number of triggers.

USAGE:

```
waitFor(cfg, timeToWait)
```

3.1 Defaults

List of functions in the src/defaults folder.

src.defaults.**checkCppPtbCfg**(*cfg*)

Set some defaults values if none have been set before.

USAGE:

```
cfg = checkDefaultsPTB(cfg)
```

Parameters
cfg (structure) –

Returns

- **cfg**
  (structure)

src.defaults.**cppPtbDefaults**(*type*)

USAGE:

```
value = cppPtbDefaults(type)
```

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

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

USAGE:

```
structure = setDefaultFields(structure, fieldsToSet)
```

Parameters

- **structure** (structure) –
- **fieldsToSet** (structure) –

Returns

- **structure**
  (structure)

3.2 Aperture

List of functions in the src/aperture folder.

(to add saveAperture)

src.aperture.**apertureTexture**(*action*, *cfg*, *thisEvent*)

USAGE:

```
[cfg, thisEvent] = apertureTexture(action, cfg, thisEvent)
```

src.aperture.**eccenLogSpeed**(*cfg*, *time*)

Vary CurrScale so that expansion speed is log over eccentricity cf. Tootell 1997; Swisher 2007; Warnking 2002 etc

src.aperture.**getApertureName**(*cfg*, *apertures*, *iApert*)

src.aperture.**saveApertures**(*saveAps*, *cfg*, *apertures*)

src.aperture.**smoothOval**(*win*, *color*, *rect*, *fringe*)

Draws a filled oval (using the PTB parameters) with a transparent fringe.

USAGE:

```
SmoothOval(WindowPtr, Color, Rect, Fringe)
```

src.aperture.**smoothRect**(*win*, *color*, *rect*, *fringe*)

Draws a filled rect (using the PTB parameters) with a transparent fringe.

USAGE:

```
SmoothRect(WindowPtr, Color, Rect, Fringe)
```

3.3 Dot

List of functions in the src/dot folder.

src.dot.**computeCartCoord**(*positions*, *dotMatrixWidth*)

USAGE:

```
cartesianCoordinates = computeCartCoord(positions, dotMatrixWidth)
```

src.dot.**computeRadialMotionDirection**(*positions*, *dotMatrixWidth*, *dots*)

src.dot.**decomposeMotion**(*angleMotion*)

Decompose angle of start motion into horizontal and vertical vector.

USAGE:

```
[horVector, vertVector] = decomposeMotion(angleMotion)
```

Parameters

angleMotion (scalar) – in degrees

Returns

- **horVector**
  horizontal component of motion
- **vertVector**
  vertical component of motion

src.dot.**dotMotionSimulation**(*cfg*, *thisEvent*, *nbEvents*, *doPlot*)

To simulate where the dots are more dense on the screen relativeDensityContrast : hard to get it below 0.10.

USAGE:

```
relativeDensityContrast = dotMotionSimulation(cfg, thisEvent, nbEvents, doPlot)
```

src.dot.**dotTexture**(*action*, *cfg*, *thisEvent*)

src.dot.**generateNewDotPositions**(*dotMatrixWidth*, *nbDots*)

src.dot.**initDots**(*cfg*, *thisEvent*)

Initialize dots for RDK

USAGE:

```
dots = initDots(cfg, thisEvent)
```

Parameters

- **cfg** (structure) –
- **thisEvent** (structure) –

Returns

- **dots**
  (structure)

- `cfg.dot.lifeTime`: dot life time in seconds
- `cfg.dot.number`: number of dots
- `cfg.dot.coherence`: proportion of coherent dots.
- `thisEvent.direction`: direction (an angle in degrees)
- `thisEvent.speed`: speed expressed in pixels per frame
- `dots.direction`
- `dots.isSignal`: signal dots (1) and those are noise dots (0)
- `dots.directionAllDots`
- `dots.lifeTime`: in frames
- `dots.speeds`: `[ndots, 2]` ; horizontal and vertical speed ; in pixels per frame
- `dots.speedPixPerFrame`

`src.dot.`**`reseedDots`**(*dots*, *cfg*)

`src.dot.`**`seedDots`**(*varargin*)

`src.dot.`**`setDotDirection`**(*positions*, *cfg*, *dots*, *isSignal*)

Creates some new direction for the dots.

USAGE:

```
directionAllDots = setDotDirection(positions, cfg, dots, isSignal)
```

Parameters

- **`positions`** –
- **`cfg`** –
- **`dots`** –
- **`isSignal`** –

Returns

- **directionAllDots**

Coherent dots have a true value in the vector `isSignal` and get assigned a value equals to the one in `dots.direction`.

All the other dots get a random value between 0 and 360.

All directions are in end expressed between 0 and 360.

`src.dot.`**`updateDots`**(*dots*, *cfg*)

3.4 Errors

List of functions in the `src/errors` folder.

`src.errors.`**errorAbort**`()`

`src.errors.`**errorAbortGetReponse**`()`

`src.errors.`**errorDistanceToScreen**`(`*cfg*`)`

`src.errors.`**errorRestrictedKeysGetReponse**`()`

3.5 Fixation

List of functions in the `src/fixation` folder.

`src.fixation.`**drawFixation**`(`*cfg*`)`

Define the parameters of the fixation cross in *cfg*.

USAGE:

```
drawFixation(cfg)
```

There are 3 types of fixations:

- `cross`
- `dot`
- `bestFixation`

See initFixation for more info.

`src.fixation.`**initFixation**`(`*cfg*`)`

Prepare the details for fixation "cross".

USAGE:

```
cfg = initFixation(cfg)
```

the fixation has a width defined by `cfg.fixation.width` : in degrees of visual

The horizontal and vertical offset (in degrees of visual) with respect to the center of the screen is defined by:

- cfg.fixation.xDisplacement
- cfg.fixation.yDisplacement

For cfg.fixation.type == 'bestFixation'

Code adapted from: "What is the best fixation target?" DOI 10.1016/j.visres.2012.10.012

Contains a fixation cross and a dot

3.6 Keyboard

List of functions in the src/keyboard folder.

src.keyboard.**getResponse**(*action*, *deviceNumber*, *cfg*, *getOnlyPress*)

Wrapper function to use KbQueue which is definitely what you should use to collect responses. You can easily collect responses while running some other code at the same time.

The queue will be listening to key presses on a keyboard device: cfg.keyboard.responseBox or cfg.keyboard.keyboard are 2 main examples.

When no deviceNumber is set then it will listen to the default device.

You can use it in a way so that it only takes responses from certain keys and ignore others (like the triggers from an MRI scanner).

Check the CPP_getResponseDemo for a quick script on how to use it.

USAGE:

```
responseEvents = getResponse(action, deviceNumber, cfg, getOnlyPress)
```

Parameters

- **action** – Defines what we want the function to do
- **deviceNumber** (integer) – device number of the keyboard or trigger box in MRI
- **cfg** –
- **getOnlyPress** – if set to true the function will only return the key presses and will not return when the keys were released (default=true). See the section on *Returns* below for more info

Returns

- **responseEvents**
  returns all the keypresses and return them as a structure with field names that make it easier to save the output of in a BIDS format

  - responseEvents.onset this is an absolute value and you should subtract the "experiment start time" to get a value relative to when the experiment was started.
  - responseEvents.trial_type = response
  - responseEvents.duration = 0
  - responseEvents.keyName the name of the key pressed
  - responseEvents(iEvent,1).pressed if
    - pressed == 1 –> the key was pressed
    - pressed == 0 –> the key was released

—

action options:

- init to initialise the queue. Initialize the buffer for key presses on a given device (you can also specify the keys of interest that should be listened to).
- start to start listening to the key presses (carefully insert into your script - where do you want to start buffering the responses).

- `check` checks all the key presses events since 'start', or since last 'check' or 'flush' (whichever was the most recent)
  - can check for demand to abort if the `escapeKey` is listed in the Keys of interest
  - can only check for demands to abort when `getResponse('check')` is called so there will be a delay between the key press and the experiment stopping
  - abort errors send specific signals that allow the catch to get them and allows us to "close" nicely
- `flush` empties the queue of events in case you want to restart from a clean queue
- `stop` stops listening to key presses

`src.keyboard.`**`checkAbort`**(*cfg*, *deviceNumber*)

Will quit your experiment if you press the key you have defined in `cfg.keyboard.escapeKey`. When no deviceNumber is set then it will check the default device. When an abort key is detected this will throw a specific error that can then be caught.

USAGE:

```
checkAbort(cfg, deviceNumber)
```

EXAMPLE:

```
try

% Your awesome experiment

catch ME % when something goes wrong

    switch ME.identifier

      case 'checkAbort:abortRequested'

      % stuff to do when an abort is requested (save data...)

      otherwise

       % stuff to do otherwise
       rethrow(ME) % display the error

    end
end
```

`src.keyboard.`**`checkAbortGetResponse`**(*responseEvents*, *cfg*)

`src.keyboard.`**`collectAndSaveResponses`**(*cfg*, *logFile*, *experimentStart*)

`src.keyboard.`**`pressSpaceForMe`**()

Use that to stop your script and only restart when the space bar is pressed. This can be useful if as an experimenter you want to have one final check on some set up before giving the green light.

USAGE:

```
pressSpaceForMe()
```

src.keyboard.**testKeyboards**(*cfg*)

Checks that the keyboards asked for properly connected.

If no key is pressed on the correct keyboard after the `timeOut` time, this exits with an error.

USAGE:

```
testKeyboards(cfg)
```

3.7 Randomization

List of functions in the `src/randomization` folder.

src.randomization.**repeatShuffleConditions**(*baseConditionVector*, *nbRepeats*)

Given `baseConditionVector`, a vector of conditions (coded as numbers), this will create a longer vector made of `nbRepeats` of this base vector and make sure that a given condition is not repeated one after the other.

USAGE:

```
shuffledRepeats = repeatShuffleConditions(baseConditionVector, nbRepeats)
```

Parameters

- **baseConditionVector** (`vector`) –
- **nbRepeats** (`integer`) –

Returns

- **shuffledRepeats**
  (vector) (dimension)

src.randomization.**setTargetPositionInSequence**(*seqLength*, *nbTarget*, *forbiddenPos*)

For a sequence of length `seqLength` where we want to insert `nbTarget` targets, this will return `nbTarget` random position in that sequence and make sure that, they are not consecutive positions.

USAGE:

```
chosenPositions = setTargetPositionInSequence(seqLength, nbTarget, forbiddenPos)
```

Parameters

- **seqLength** (`integer`) –
- **nbTarget** (`integer`) –
- **forbiddenPos** (`vector of integers`) –

Returns

- **chosenPositions**

src.randomization.**shuffle**(*unshuffled*)

Is just there to replace the Shuffle function from PTB in case it is not in the path. Can be useful for testing or for continuous integration.

USAGE:

```
shuffled = shuffle(unshuffled)
```

3.8 Screen

List of functions in the src/screen folder.

src.screen.**farewellScreen**(*cfg*)

src.screen.**standByScreen**(*cfg*)

It shows a basic one-page instruction stored in *cfg.task.instruction* and wait for *space* stroke.

USAGE:

```
standByScreen(cfg)
```

3.9 Utilities

List of functions in the src/utils folder.

(to add makeGif)

src.utils.**checkPtbVersion**()

Checks that the right dependencies are installed.

USAGE:

```
checkPtbVersion()
```

src.utils.**cleanUp**()

A wrapper function to close all windows, ports, show mouse cursor, close keyboard queues and give access back to the keyboards.

USAGE:

```
cleanUp()
```

src.utils.**computeFOV**(*cfg*)

Computes the number of degrees of visual angle in the whole field of view.

USAGE:

```
FOV = computeFOV(cfg)
```

Parameters

cfg (structure) –

Returns

- **FOV**
  (scalar)

`delta = 2 arctan ( d / 2D )`

- delta is the angular diameter
- d is the actual diameter of the object
- D is the distance to the object

The result obtained is in radians.

`src.utils.`**`degToPix`**(*fieldName*, *structure*, *cfg*)

For a given field value in degrees of visual angle in the structure, this computes its value in pixel using the pixel per degree value of the cfg structure and returns a structure with an additional field with Pix suffix holding that new value.

USAGE:

```
structure = degToPix(fieldName, structure, cfg)
```

Parameters

- **`fieldName`** (string) –
- **`structure`** (structure) –
- **`cfg`** (structure) –

Returns

- **structure**
  (structure)

EXAMPLE:

```
fixation.width = 2;
cfg.screen.ppd = 10;

fixation = degToPix('width', fixation, cfg);
```

`src.utils.`**`pixToDeg`**(*fieldName*, *structure*, *cfg*)

For a given field value in pixel in the structure, this computes its value in degrees of viual angle using the pixel per degree value of the cfg structure and returns a structure with an additional field holding that new value and with a fieldname with any 'Pix' suffix removed and replaced with the 'DegVA' suffix .

USAGE:

```
structure = pixToDeg(fieldName, structure, cfg)
```

Parameters

- **`fieldName`** (string) –
- **`structure`** (structure) –
- **`cfg`** (structure) –

Returns

- **structure**
  (structure)

EXAMPLE:

```
fixation.widthPix = 20;
cfg.screen.ppd = 10;

fixation = degToPix('widthPix', fixation, cfg);
```

src.utils.**printCreditsCppPtb**()

src.utils.**printScreen**(*win*, *filename*, *frame*)

src.utils.**setUpRand**()

Resets the seed of the random number generator. Will "adapt" depending on the Matlab / Otave version.

USAGE:

```
setUpRand()
```

For an alternative from PTB see `ClockRandSeed()`

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

MATLAB MODULE INDEX

S

INDEX

T

U

W