pyhrf Documentation

Release

Lotfi

October 20, 2016

Contents

CHAPTER 1

Overview

PyHRF is a set of tools for within-subject fMRI data analysis, focused on the characterization of the hemodynamics.

Within the chain of fMRI data processing, these tools provide alternatives to the classical within-subject GLM fitting procedure. The inputs are preprocessed images (except spatial smoothing) and the outputs are the contrast maps and the HRF estimates.

The package is mainly written in Python and provides the implementation of the two following methods:

- **The joint-detection estimation (JDE)** approach, which divides the brain into functionnaly homogeneous regions and provides one HRF estimate per region as well as response levels specific to each voxel and each experimental condition. This method embeds a temporal regularization on the estimated HRFs and an adaptive spatial regularization on the response levels.
- **The Regularized Finite Impulse Response (RFIR)** approach, which provides HRF estimates for each voxel and experimental conditions. This method embeds a temporal regularization on the HRF shapes, but proceeds independently across voxels (no spatial model).

See *Introduction* for a more detailed overview.

To cite PyHRF and get a comprehensive description, please refer to this paper:

> T. Vincent, S. Badillo, L. Risser, L. Chaari, C. Bakhous, F. Forbes and P. Ciuciu "Flexible multivariate hemodynamics fMRI data analyses and simulations with PyHRF" Font. Neurosci., vol. 8, no. 67, 10 April 2014.

CHAPTER 2

Site content:

2.1 Introduction

2.1.1 Scientific issues

PyHRF aims to provide advanced tools for single-subject analysis of functional Magnetic Resonance Imaging (fMRI) data acquired during an experimental paradigm (i.e. not resting-state). The core idea of PyHRF is to estimate the dynamics of brain activation by recovering the so-called Hemodynamic Response Function (HRF) in a spatially varying manner. To this end, PyHRF implements two different approaches:

1. a voxel-wise and condition-wise HRF estimation [1];
2. a parcel-wise spatially adaptive joint detection-estimation (JDE) algorithm [2,3].

The second approach is more powerful since it jointly addresses (i) the localization of evoked brain activity in response to external stimulation and (ii) the estimation of parcelwise hemodynamic filters. To this end, a parcellation (either functional or anatomical) has to be provided as input parameter.

This tool hence provides interesting perspectives for understanding the HRF differences between brain regions and also between individuals and populations (infants, children, adults, patients ...). Within the classical workflow of fMRI data analysis (preprocessings, estimation, statistical inference), PyHRF takes place at the estimation stage. However, we have also implemented posterior probability maps (PPMs) to allow the user performing statistical inference at the subject-level.

For the sake of computational efficiency, the variational expectation-maximization (VEM) [3] is used as default algorithm for computing the solution. The results that are generated in the context are the following:

– the 3D effect size maps associated with each experimental condition; – the 3D contrast maps specified by the user; – the time-series describing the HRFs for the set of all brain regions (4D volume). The analysis can also be performed on the cortical surface from projected BOLD signals and

then produces functional textures to be displayed on the input cortical mesh.

[1] P. Ciuciu, J.-B. Poline, G. Marrelec, J. Idier, Ch. Pallier, and H. Benali, "Unsupervised robust non-parametric estimation of the hemodynamic response function for any fMRI experiment", IEEE Trans. Medical Imaging; 22(10):1235-1251, Oct 2003.

[2] T. Vincent, L. Risser and P. Ciuciu, "Spatially adaptive mixture modeling for analysis of within-subject fMRI time series", IEEE Trans. Medical Imaging; 29(4):1059-1074, Apr 2010.

[3] L. Chari, T. Vincent, F. Forbes, M. Dojat and P. Ciuciu, "Fast joint detection-estimation of evoked brain activity in event-related fMRI using a variational approach", IEEE Trans. Medical Imaging; 32(5):821-837, May 2013.

2.1.2 Package overview

pyhrf is mainly written in Python, with some C-extension that handle computationally intensive parts of the algorithms. The package relies on classical scientific libraries: numpy, scipy, matplotlib as well as Nibabel to handle input/outputs and NiPy which provides tools for functional data analysis.

pyhrf can be used in a stand-alone fashion and provides a set of simple commands in a modular fashion. The setup process is handled through XML files which can be adapted by the user from a set of templates. This format was chosen for its hierarchical organisation which suits the nested nature of the algorithm parametrisations. A dedicated XML editor is provided with a PyQt graphical interface for a quicker edition and also a better review of the treatment parameters. When such an XML setup file is generated ab initio, it defines a default analysis which involves a small real data set shipped with the package. This allows for a quick testing of the algorithm and is also used for demonstration purpose.

An articifial fMRI data generator is provided where the user can test the behaviour of the algorithms with different activation configurations, HRF shapes, nuisance types (noise, low frequency drifts) and paradigms (slow/fast event-related or bloc designs).

Concerning the analysis process, which can be computationally intensive, pyhrf handles parallel computing through the python software soma-workflow for the exploitation of cluster units as well as multiple core computers.

Finally, results can be browsed by a dedicated viewer based on PyQt and matplotib which handles n-dimensionnal images and provide suitable features for the exploration of whole brain hymodynamics results.

2.2 Installation

Instructions are given for Linux and Python 2.7. The package is mainly developed and tested under Ubuntu 14.04, we then recommend this distribution. We officially support Ubuntu 12.04, Ubuntu 14.04 Debian stable (jessie) and Fedora 21 and 22. See *below* for instructions for each distribution.

For **windows users**, a linux distribution can be installed easily inside a virtual machine such as Virtual Box. This wiki page explains how to install ubuntu from scratch in Virtual Box or you can get some free VirtualBox images.

2.2.1 Requirements / Dependencies

Dependencies are:

- python 2.5, 2.6 or 2.7 **with development headers**
- numpy >= 1.0
- scipy >= 0.7
- matplotlib
- nibabel
- sympy (required by nipy)
- nipy

- a C compiler

Optional dependencies:

- PyQt4 (viewer and XML editor)
- joblib (local distributed computing)
- paramiko (local network distributed computing)
- soma-workflow (remote distributed computing)
- scikit-learn (clustering)
- sphinx (to generate documentation)
- pygraphviz (for optimized graph operations and outputs)
- munkres (parcellation operation)

Linux-based

Please refer to the page corresponding to your distribution.

Ubuntu 12.04

Ubuntu 14.04

Debian stable (jessie)

Fedora 21

Fedora 22

If you have another distribution or are an advanced user, consider `installing pyhrf in a virtual environment`

2.3 Manual

2.3.1 Variational (VEM) analysis

To run a JDE-VEM analysis you need:

- paradigm csv file corresponding to your experimental design
- the preprocessed (spatially UNsmoothed) BOLD data associated with a single run [4D-Nifti file]
- the TR (in s) used at acquisition time
- a parcellation mask (if you normalize your data into MNI space, you can use the one provided with pyhrf)

If you want to use provided parcellation mask, check the path of the file by running:

```
$ pyhrf_list_datafiles
```

and check for the `stanford_willard_parcellation_3x3x3mm.nii.gz` file.

Then you can run:

```
$ pyhrf_jde_vem_analysis 2.5 stanford_willard_parcellation_3x3x3mm.nii.gz paradigm.csv bold_
```

If you want to tune the parameters, check command line options (see commands). This is adviced to set the `dt` parameter (the temporal resolution of the HRF) and the output folder.

- The multirun extension is currently under development and testing

2.3.2 File format specification

Paradigm CSV format

To encode the experimental paradigm, the CSV file format comprises 5 columns:

1. Session number (integer)
2. Experimental Condition, as string with double quotes
3. Onset (float in s.)
4. Duration (float s.). 0 for event-related paradigm
5. Amplitude (float, default is 1.).

Example (extracted from pyhrf/python/datafiles/paradigm_loc.csv):

Table 2.1: Localizer paradigm

Session	Experimental Condition	Onset	Duration	Amplitude
0	calculaudio	35.400000	0.000000	1.
0	clicDaudio	143.400000	0.000000	1.
0	clicDaudio	162.000000	0.000000	1.
0	clicDaudio	230.000000	0.000000	1.
0	clicDvideo	18.000000	0.000000	1.
0	clicDvideo	69.000000	0.000000	1.
0	clicDvideo	227.700000	0.000000	1.

NB: Be carefull not to end float onset and duration with a point `.` because it could results in python not being able to split the paradigm correctly (end with a zero if necessary)

Contrasts JSON format

To encode contrasts definition, use the following json format:

```
{
    "contrast_name": "linear combination of conditions",
    "Audio-Video": "clicDaudio-clicDvideo"
}
```

The names in the contrast definition MUST correspond to the experimental conditions (case sensitive) defined in the above CSV file.

2.3.3 Multicore computing

Pyhrf can make computation on several cores using the joblib python package. Joblib is an optional dependency of pyhrf so if you plan to use multiprocessing computation you need to install it (refer to the installation page)

Configuration

You can configure the number of used cores by setting the `nb_procs` option in the pyhrf configuration file. If you want to auto-detect the number of processors and cores that the system can use, set this option to `0`.

Use

When calling the `pyhrf_jde_vem_analysis` script, the multiprocessing mode is active by default. If you want to disable it use `--no-parallel` command line option.

2.3.4 PyHRF commands

fMRI data analysis

pyhrf_jde_vem_analysis

Usage:

```
pyhrf_jde_vem_analysis [options] TR parcels_file onsets_file [bold_data_run1 [bold_data_run2
```

Description: Run an fMRI analysis using the JDE-VEM framework All arguments (even positional ones) are optional and default values are defined in the beginning of the script

Options:

```
positional arguments:
  tr                    Repetition time of the fMRI data
  parcels_file          Nifti integer file which define the parcels
  onsets_file           CSV onsets file
  bold_data_file        4D-Nifti file of BOLD data

optional arguments:
  -h, --help            show this help message and exit
  -c DEF_CONTRASTS_FILE, --def_contrasts_file DEF_CONTRASTS_FILE
                        JSON file defining the contrasts
  -d DT, --dt DT        time resolution of the HRF (if not defined here or in
```

```
                        the script, it is automatically computed)
  -l HRF_DURATION, --hrf-duration HRF_DURATION
                        time lenght of the HRF (in seconds, default: 25.0)
  -o OUTPUT_DIR, --output OUTPUT_DIR
                        output directory (created if needed, default:
                        /home/tperret/code/pyhrf)
  --nb-iter-min NB_ITER_MIN
                        minimum number of iterations of the VEM algorithm
                        (default: 5)
  --nb-iter-max NB_ITER_MAX
                        maximum number of iterations of the VEM algorithm
                        (default: 100)
  --beta BETA           (default: 1.0)
  --hrf-hyperprior HRF_HYPERPRIOR
                        (default: 1000)
  --sigma-h SIGMA_H     (default: 0.1)
  --estimate-hrf        (default: True)
  --no-estimate-hrf     explicitly disable HRFs estimation
  --zero-constraint     (default: True)
  --no-zero-constraint  explicitly disable zero constraint (default enabled)
  --drifts-type DRIFTS_TYPE
                        set the drifts type, can be 'poly' or 'cos' (default:
                        poly)
  -v {DEBUG,10,INFO,20,WARNING,30,ERROR,40,CRITICAL,50}, --log-level {DEBUG,10,INFO,20,WARNI
                        (default: WARNING)
  -p, --parallel        (default: True)
  --no-parallel         explicitly disable parallel computation
  --save-processing     (default: True)
  --no-save-processing
```

Parcellation tools

pyhrf_parcellate_glm

pyhrf_parcellate_spatial

pyhrf_parcellation_extract

Usage:

```
pyhrf_parcellation_extract [options] PARCELLATION_FILE PARCEL_ID1 [PARCEL_ID2 ...]
```

Description: Extract a sub parcellation comprising only `PARCEL_ID1`, `PARCEL_ID2`, ... from the input parcellation `PARCELLATION_FILE`.

Options:

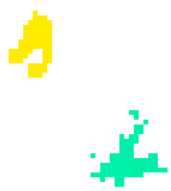
```
  -h, --help            show this help message and exit
  -v VERBOSELEVEL, --verbose=VERBOSELEVEL
                        { 0 :    'no verbose' 1 :    'minimal verbose' 2 :
                        'main steps' 3 :    'main function calls' 4 :    'main
                        variable values' 5 :    'detailled variable values' 6
                        :    'debug (everything)' }
  -o FILE, --output=FILE
                        Output parcellation file. Default is
```

```
                              <input_file>_<PARCEL_IDS>.(nii|gii)
-c, --contiguous              Make output parcel ids be contiguous
```

Example The input parcellation `parcellation.nii` looks like:

```
pyhrf_parcellation_extract parcellation.nii 36 136 -o parcellation_sub.nii
```

The output parcellation `parcellation_sub.nii` will look like:



Misc tools

pyhrf_list_datafiles

Usage:

```
pyhrf_list_datafiles [options]
```

Description:

This command lists all data files included in the package.

Options

```
-h, --help        show this help message and exit
-b, --base-name   Display only basenames
```

Examples:

```
pyhrf_list_datafiles

 /home/user/software/pyhrf/python/pyhrf/datafiles/SPM_v12.mat.gz
 /home/user/software/pyhrf/python/pyhrf/datafiles/SPM_v5.mat.gz
```

```
/home/user/software/pyhrf/python/pyhrf/datafiles/SPM_v8.mat.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/cortex_occipital_hrf_territories_3mm.nii
/home/user/software/pyhrf/python/pyhrf/datafiles/cortex_occipital_hrf_territories_convex_hu
/home/user/software/pyhrf/python/pyhrf/datafiles/cortex_occipital_right_GWmask_3mm.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/cortex_occipital_white_surf.gii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/dummySmallBOLD.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/dummySmallMask.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_V4.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_a.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_av.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_av_comma.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_av_d.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_c_only.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_cp_only.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/paradigm_loc_cpcd.csv
/home/user/software/pyhrf/python/pyhrf/datafiles/real_data_surf_tiny_bold.gii
/home/user/software/pyhrf/python/pyhrf/datafiles/real_data_surf_tiny_mesh.gii
/home/user/software/pyhrf/python/pyhrf/datafiles/real_data_surf_tiny_parcellation.gii
/home/user/software/pyhrf/python/pyhrf/datafiles/real_data_vol_4_regions_BOLD.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/real_data_vol_4_regions_anatomy.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/real_data_vol_4_regions_mask.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/simu.pck
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_hrf_3_territories.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_hrf_3_territories_8x8.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_hrf_4_territories.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_activated.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_ghost.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_house_sun.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_icassp13.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_invader.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_pacman.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_small_spots_1.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_small_spots_2.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_stretched_1.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_template.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_tiny_1.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_tiny_2.png
/home/user/software/pyhrf/python/pyhrf/datafiles/simu_labels_tiny_3.png
/home/user/software/pyhrf/python/pyhrf/datafiles/stanford_willard_parcellation_3x3x3mm.nii.
/home/user/software/pyhrf/python/pyhrf/datafiles/subj0_anatomy.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/subj0_bold_session0.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/subj0_parcellation.nii.gz
/home/user/software/pyhrf/python/pyhrf/datafiles/subj0_single_roi.nii.gz
```

```
pyhrf_list_datafiles -b

    SPM_v12.mat.gz
    SPM_v5.mat.gz
    SPM_v8.mat.gz
    cortex_occipital_hrf_territories_3mm.nii
    cortex_occipital_hrf_territories_convex_hull.tgz
    cortex_occipital_right_GWmask_3mm.nii.gz
    cortex_occipital_white_surf.gii.gz
    dummySmallBOLD.nii.gz
    dummySmallMask.nii.gz
    paradigm_V4.csv
    paradigm_loc.csv
```

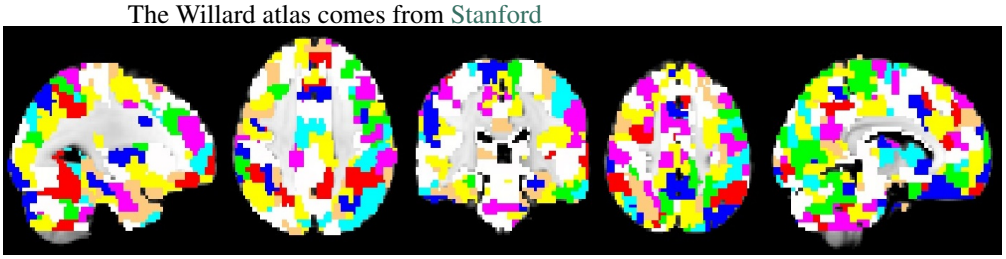
```
        paradigm_loc_a.csv
        paradigm_loc_av.csv
        paradigm_loc_av_comma.csv
        paradigm_loc_av_d.csv
        paradigm_loc_c_only.csv
        paradigm_loc_cp_only.csv
        paradigm_loc_cpcd.csv
        real_data_surf_tiny_bold.gii
        real_data_surf_tiny_mesh.gii
        real_data_surf_tiny_parcellation.gii
        real_data_vol_4_regions_BOLD.nii.gz
        real_data_vol_4_regions_anatomy.nii.gz
        real_data_vol_4_regions_mask.nii.gz
        simu.pck
        simu_hrf_3_territories.png
        simu_hrf_3_territories_8x8.png
        simu_hrf_4_territories.png
        simu_labels_activated.png
        simu_labels_ghost.png
        simu_labels_house_sun.png
        simu_labels_icassp13.png
        simu_labels_invader.png
        simu_labels_pacman.png
        simu_labels_small_spots_1.png
        simu_labels_small_spots_2.png
        simu_labels_stretched_1.png
        simu_labels_template.png
        simu_labels_tiny_1.png
        simu_labels_tiny_2.png
        simu_labels_tiny_3.png
        stanford_willard_parcellation_3x3x3mm.nii.gz
        subj0_anatomy.nii.gz
        subj0_bold_session0.nii.gz
        subj0_parcellation.nii.gz
        subj0_single_roi.nii.gz
```

2.3.5 Parcellation mask

The JDE framework estimates HRF parcels-wide. This means that you need a parcellation mask to compute the estimation-detection. PyHRF provides some tools to generate parcellation masks and provides also a parcellation mask (see *next section*)

Willard atlas

The Willard atlas comes from Stanford



To use it check where it is installed by issuing:

```
$ pyhrf_list_datafiles
```

and check for `stanford_willard_parcellation_3x3x3mm.nii.gz` file.

This parcellation mask has been created from the files distributed by Stanford (above website) with a voxel resolution of 3x3x3mm and a volume shape of 53x63x52 voxels.

If you use this parcellation mask, please cite the following paper:

The citation for the 499 fROI atlas, nicknamed the "Willard" atlas after the two creators, **Will** iam Shirer and Bern **ard** Ng, is the following publication:

Richiardi J, et al.: Correlated gene expression supports synchronous activity in brain networks. Science (2015).

2.3.6 Configuration

Package options are stored in `$HOME/.pyhrf/config.cfg`, which is created after the installation. It handles global package options and the setup of parallel processing. Here is the default content of this file (section order may change):

```
[global]
write_texture_minf = False          ; compatibility with Anatomist for texture file
tmp_prefix = pyhrftmp               ; prefix used for temporary folders in tmp_path
verbosity = 0                       ; default of verbosity, can be changed with option -v
tmp_path = /tmp/                    ; where to write file
use_mode = enduser                  ; "enduser": stable features only, "devel": +indev featu
spm_path = None                     ; path to the SPM matlab toolbox (indev feature)


[parallel-cluster]                  ; Distributed computation on a cluster.
                                    ; Soma-workflow is required.
                                    ; Authentification by ssh keys must be
                                    ; configured in both ways (remote <-> local)
                                    ; -> eg copy content of ~/.ssh/id_rsa.pub (local machine
                                    ;    at the end of ~/.ssh/authorized_keys (remote machin
                                    ;    Also do the converse:
                                    ;    copy content of ~/.ssh/id_rsa.pub (remote machine)
                                    ;    at the end of ~/.ssh/authorized_keys (local machine

server_id = None                    ; ID of the soma-workflow-engine server
server = None                       ; hostname or IP adress of the server
user = None                         ; user name to log in the server
remote_path = None                  ; path on the server where data will be stored

[parallel-local]                    ; distributed computation on the local cpu
niceness = 10                       ; niceness of remote jobs
nb_procs = 1                        ; number of distruted jobs, better not over
                                    ; the total number of CPU
                                    ; 'cat /proc/cpuinfo | grep processor | wc -l' on linux
                                    ; 'sysctl hw.ncpu' on MAC OS X
                                    ; Set it to 0 if you want to auto-detect the number of
                                    ; availables cpus and cores (taking into account the ker
                                    ; restrictions such as cgroups, ulimit, ...)

[parallel-LAN]                      ; Distributed computation on a LAN
                                    ; Authentification by ssh keys must be
                                    ; configured
```

```
remote_host = None                  ; hostname or IP address of a host on the LAN
niceness = 10                       ; niceness for distributed jobs
hosts = $HOME/.pyhrf/hosts_LAN      ; plain text file containing coma-separated list of host
user = None                         ; user name used to log in on any machine
                                    ; on the LAN
remote_path = None                  ; path readable from the machine where
                                    ; pyhrf is launched (to directly retrieve
                                    ; results)
```

2.3.7 Old manual pages

Volume processing

The main usage of pyhrf is made up of 4 steps:

1. *Build a parcellation*
2. *Configure a pyhrf treatment*
3. *Run a pyhrf treatment*
4. *Visualize the results of a pyhrf treatment*

The following procedure provides details on how to apply the Joint Detection Estimation (JDE) algorithm to fMRI data.

Parcellation

The default processing already refers to a parcellation data file which corresponds to bilateral auditory cortices. In this respect, the first way to create a parcellation is to defined a n-ary mask of ROIs (one integer per ROI, 0 is the background) which can be generated by the marsbar SPM toolbox.

PyHRF also provides alternatives to make a functional parcellation (Ward algorithm, balance partitioning). In the following, the parcellation file is referred to as the mask file, and the treatment will be iteratively performed over each and every parcel in an independent manner.

Treatment configuration

The minimal information needed by a Pyhrf treatment for single session (or run) data is:

- the BOLD data file in NIFTI format, concatenated over all scan volumes, ie 3D+time.
- a parcellation NIFTI file which is a 3D volume of labels defining the ROIs.
- a set of stimulus names or experimental conditions associated with onset sequences.

These inputs are gathered in an XML configuration file. A template of such file can be generated by running the follwing command line:

```
pyhrf_jde_buildcfg -e
```

A file detectestim.xml is then created. By default, it contains the definition of a treatment over test data (provided with the Pyhrf distribution for ease of demonstration). Such data have been acquired a long time ago during a fast event-related design known as the Localizer protocol (Pinel et al, BMC Neurosci 2007). It serves as example to build a customized treatment.

To edit this file, you can use the XML editor:

```
pyhrf_xmledit detecestim.xml
```

The **fmri_data** section comprises: all fMRI data (**sessions_data** ie, possibly multiple sessions), the time of repetition (**tr**) and the mask file (**mask_file**). Within **sessions_data**, each numbered child tag stands for a single session. By default only one session is present, indexed by **i0**. If you have to append other sessions, index them by *i1*, *i2*, and so on. Down one level, the definition of one session is made of:

- **onsets**, which stores the stimulus arrival times of experimental conditions. Every label of experimental condition must be unique. The sequence associated with each condition is the set of arrival times in seconds, separated by one space delimiter.
- **durations**, which must be consistent with the above mentioned **onsets** defition and define the durations of stimuli in seconds. If empty, then peaked stimuli or events (duration=0) are considered.
- **bold_file**, which indicates the location of the 3D+time BOLD data file. Supported formats: NIFTI, ANALYSE.

Following **fmri_data**, there are two other important parameters which are common to all sessions:

- **tr** is the time of repetition expressed in seconds.
- **mask_file**, which indicates the location of the mask file: a volume of labels (integers) where 0 is the background. This file can be either the result from any parcellation technique or simply a binary mask resulting for instance from thresholding an binarizing a statistical map.

The following **analyser** section groups all parameters for the actual analysis. The contrasts have to be defined in **analyser/sampler/responseLevels/contrasts**. Note that the condition labels must be consistent with those entered in the data definition part. See contextual comments within the xml for more information on other parameters. The estimation of the hemodynamics can be setup by changing parameter "do_sampling". If set to "True" then the response function is estimated, otherwise if it is set to "False" then it is fixed to the canonical HRF (the same as in SPM).

Treatment execution

To run the pyhrf treatment defined in detectestim.xml, type:

```
pyhrf_jde_estim -v1
```

Here -v1 stands for the level of verbosity (from 0: quiet, up to 6: everything - for debugging purpose)

Result output files The main outputs (NIFTI format) are written in the current directory by default (or the directory specified in detectestim.xml, see **sampler/output_dir**):

- **Contrast maps**: contrast maps are generated with the following file names: jde_vem_nrls_contrasts_<contrast_name>.nii. Their corresponding standard deviations are stored in files like jde_vem_nrls_contrasts_std_<contrast_name>.nii. Last, normalized contrasts are stored in files jde_vem_nrls_ncontrasts_<contrast_name>.nii

- **Response levels maps**: for each experimental condition, estimated effect sizes (or neural response levels) are stored in files of the form: jde_vem_nrl_pm_condition_<condition_name>.nii
- **Hemodynamic responses**: to each voxel, the product a*h is computed and stored in jde_vem_ah.nii. Note that one HRF shape is common to all conditions, but its magnitude varies from one voxel to another for a given condition and from one to condition to another in the same position.

Result visualization

The pyhrf viewer is called with NIFTI volumes as inputs. To view all the results within the JDE output directory, run:

```
pyhrf_view *nii
```

Parallel computing with PyHRF

WARNING: this is an old documentation about unmaintained feature. Check multiprocessing for local multiple core use.

LAN

Computation distribution uses *ssh* to launch commands on distant machines on a Local Aera Network (LAN) and retrieve results from them on the local machine.

Setup The setup information to configure the parallel LAN feature of pyhrf is in the section [parallel-LAN] of the file ~/.pyhrf/config.cfg:

```
[parallel-LAN]
niceness = 10
remote_path = /path/to/store/results
hosts = localhost,localhost,localhost,localhost
user = my_user
remote_host = my_remote_host
enable_unit_test = 1
```

where:

- "niceness" is the nice level with which command will be launched on distant machines. The higher, the less priority the jobs will have (useful to not disturb other people that are using the distant machines as well).
- "remote_path" must be a path on the network that is accessible by all machines. It is used to store temporary results.
- "hosts" is a coma-separated list of machine hostnames on which the job will be launched. If one wants to launch multiple jobs on a given machine, the hostname can be duplicated.
- "user" is a ssh login valid on all distant machines.
- "remote_host" is a specific distant machine hostname that will be used to retrieve results
- "enable_unit_test" is a flag to enable unit testing of the LAN feature

> **Warning:** The ssh connection on distant machines must be *non-interactive*. To do so, the ssh key of the local user must be appended to the file `authorized_keys` of the distant user.

Test To test the configuration prior to launching an actual analysis:

1. set the flag "enable_unit_test" to 1 in `~/.pyhrf/config.cfg`
2. run the following command:

```
pyhrf_maketests pyhrf.test.test_treatment.TreatmentTest.test_default_treatment_parallel_
```

Analysis To launch a LAN-distributed analysis, simply add the option "-x LAN" to any `pyhrf_..._estim` command. For example, the default JDE analysis can be run with:

```
pyhrf_jde_buildcfg          # build the XML parameter file
pyhrf_jde_estim -v1 -x LAN  # launch the analysis using LAN-distributed computation
```

Cluster

Configuration

Setup To work on a cluster, pyhrf relies on soma-workflow which has to be installed on both sides (client/cluster). See the soma-workflow installation page for detailled instructions.

Launch a soma-workflow database server on the cluster:

```
python -m soma.workflow.start_database_server SW_CLUSTER_ID &
```

Configure soma-workflow on client side, file ~/.soma-workflow.cfg:

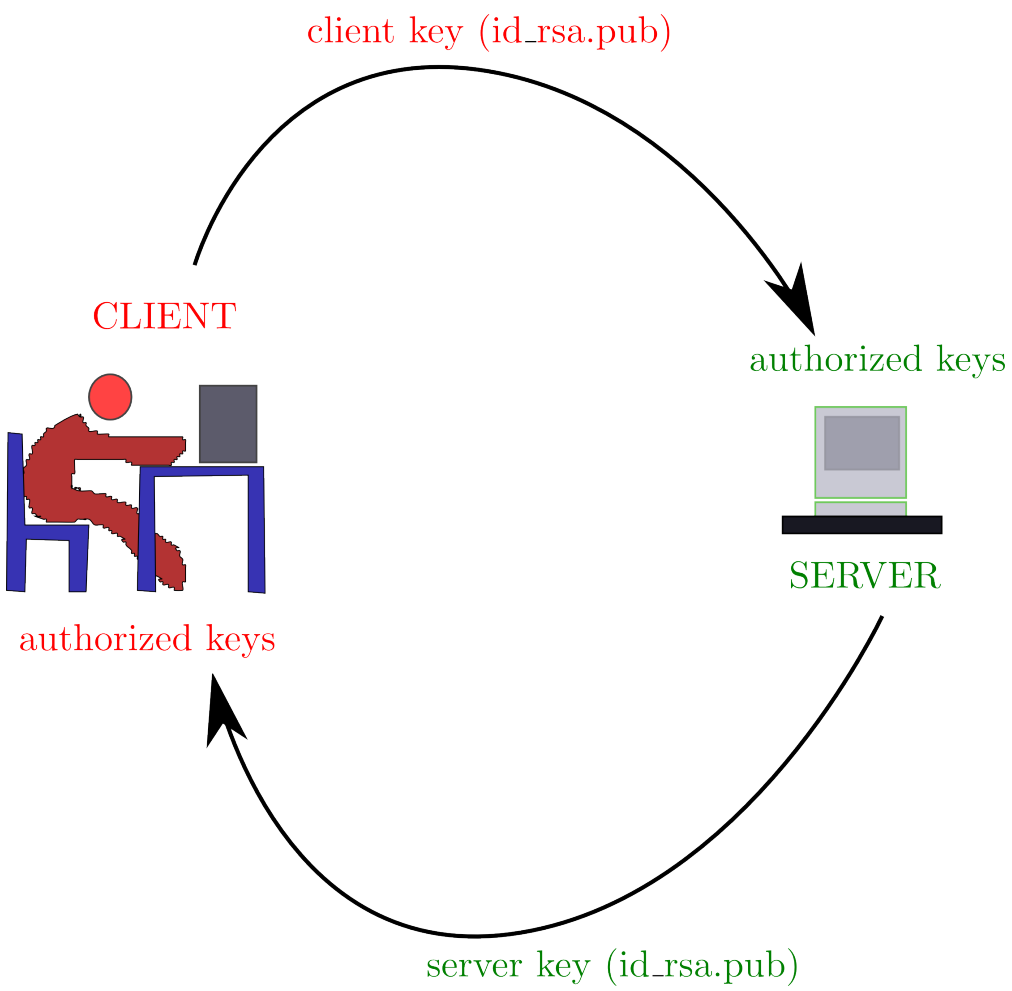
```
[SW_CLUSTER_ID]
cluster_address = CLUSTER_HOSTNAME
submitting_machines = CLUSTER_HOSTNAME
```

To ensure the communication from the server to the client, OpenSSH server must be installed and running:

```
sudo aptitude install sshd
sudo aptitude install openssh-server
```

Pyro must be installed to ensure the same configuration for the client and the server.

Connexion with the server To enable the connexion from the client to the server, the public key of the local user must be saved in the authorized keys of the server. To enable the connexion from the server to the client, the public key of the server must be saved in the authorized keys of the local user.



Configure the connexion with the server, file: ~/.pyhrf/config.cfg

Default content of the parallel-cluster section:

```
[parallel-cluster]
server_id = None
server = None
user = None
remote_path = None
```

With the previously defined configuration, one should get:

```
[parallel-cluster]
server_id = SW_CLUSTER_ID
server = CLUSTER_HOSTNAME
user = CLUSTER_USER
remote_path = CLUSTER_PYHRF_PATH
```

Launch a pyhrf analysis on the cluster In a directory containing the xml file (detectestim.xml), launch the analysis with:

```
pyhrf_jde_estim -v1 -x cluster
```

Monitor execution (user interface to visualize running jobs, progression...):

```
soma_workflow_gui -u CLUSTER_USER -a
```

Analysis of fMRI data projected on the cortical surface

This analysis comprises the following steps:

1. *Extract the cortical mesh*
2. *Project data onto the cortical surface*
3. Run a pyhrf treatment on the projected data

Data inputs are:

- Anatomical MRI (3D)
- fMRI data (3D+time)
- paradigm (condition names, event timings)

Extraction of the cortical surface.

The objective is to extract the grey/white matter interface (deep cortical or "white" surface) as a mesh from the anatomical MRI. It is worth noting that we would not want the gyral surface (superficial cortical or "pial" surface) nor the average cortical surface, because of the principle of the subsequent data projection. Indeed, the latter takes into account the orientation of the cortical fold and take the white surface as input.

Several tools are available to extract the cortical surface:

- Freesurfer
- Brainsuite
- T1 MRI toolbox of Brainvisa

The T1 MRI toolbox of Brainvisa (Morphologist treatment) is recommended here but any procedure may work as long as it provides a file in gifti format. The library used to read meshes in Pyhrf is Nibabel.

Here is an illustration of the process used to obtain the real data set shipped with PyHRF:

The default setup of the morphologist treatment produces meshes in a Brainvisa-specific format (''.mesh'') and not gifti. You may change the file format to gifti in the sub-treatment "Grey white Interface" or use the following command to convert the resulting mesh files:

```
AimsFileConvert -i subject_Xwhite.mesh -o subject_Xwhite.gii
```

Data Management

Data import/export with files (no DB)

In fMRI data analyses, files are often organised hierarchically in folders that has to be compatible with the analysis tools. Quite often, data files originate from data importer specific to some MRI scanner or to an online platform such as Shanoir. These tools provide archives where data files are stored in a flat structure.

Here is a tool to export data files from such flat structure into a hierarchicy of folder. It can also be used to import data files from a specific hierarchy to another.

To do so, the function `pyhrf.tools.io.rx_copy()` copies files with patterned names into a directory structure

- a source is defined by a regular expressions with named groups
- a target is defined by python format strings whose named arguments match group names in the source regexp.

Example: Folder ./raw_data contains the following files:

```
AC0832_anat.nii
AC0832_asl.nii
AC0832_bold.nii
PK0612_asl.nii
PK0612_bold.nii
```

I want to export these files into the following directory structure: ./export/<subject>/<modality>/data.nii where <subject> and <modality> have to be replaced by chunks extracted from the input files

To do so, define a regexp to catch useful chunks (or tags) in input files and also format strings that will be used to create target file names:

```
# regexp to capture values of subject and modality:
src = '(?P<subject>[A-Z]{2}[0-9]{4})_(?P<modality>[a-zA-Z]+).nii'
# definition of targets:
src_folder = './raw_data/'
dest_folder = ('./export', '{subject}', '{modality}')
dest_basename = 'data.nii'
# do the thing:
rx_copy(src, src_folder, dest_basename, dest_folder):
```

Should result in the following copied files:

```
./export/AC0832/bold/data.nii
./export/AC0832/anat/data.nii
./export/AC0832/asl/data.nii
./export/PK0612/bold/data.nii
./export/PK0612/asl/data.nii
```

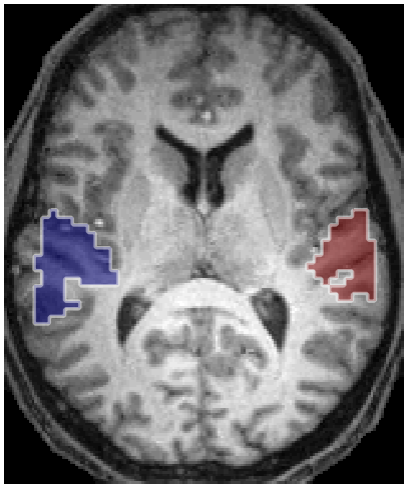
Demo of VEM-based JDE and JPDE analyses on default data

Default data set

The experimental paradigm consists of a fast event-related design with sixty auditory and visual stimulus events. The acquisition comprises one single session of 125 scans, TE=30ms,

TR=2400ms, slice thickness=3mm, FOV=192mm2, transversal orientation, 2x2mm2 in-plane resolution.

Two bilateral temporal regions of interest were previously manually selected. Each one comprises around 600 voxels. The subsequent analyses are applied independently on each region.



JDE VEM analysis

Build the XML configuration file

```
pyhrf_jde_buildcfg --vem
```

By default, the analysis is set up to be perfomed on a data set stored in the PyHRF package. The default analysis parameters are suitable for this treatment so that the user do not need to edit them. Still, one can review all the setup with the XML editor:

```
pyhrf_view detectestim.xml
```

Run the analysis

```
pyhrf_jde_estim -v1
```

View the results

```
pyhrf_view jde_vem_roi_mask.nii jde_vem_nrls.nii jde_vem_hrf.nii
```

Note: to view a region-specific HRF, the corresponding region has to be clicked when wiewing *jde_vem_roi_mask.nii*.

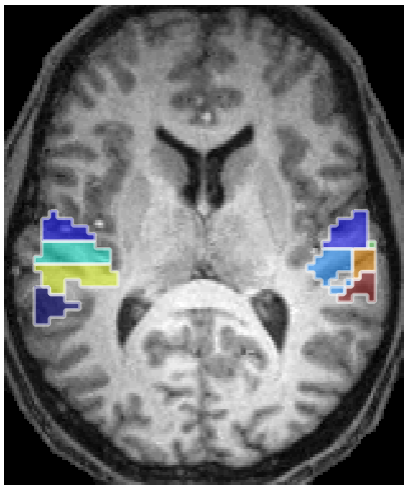
JPDE VEM analysis

The following JPDE procedure will divide the initial 2-regions parcellation into 10 parcels.

Build the XML configuration file

```
pyhrf_jde_buildcfg --jpde
```

This configuration file comprises the same data definition as for the previous JDE analysis. Note that the analyser part has the flag *jpde* set to *True*. The initial parcellation is also shipped with the default data and looks like:



Run the estimation

```
pyhrf_jde_estim -v1
```

View the results – estimated parcellation, response levels, HRFs

```
pyhrf_view jde_vem_hrf.nii jde_vem_Pmask.nii jde_vem_nrls.nii
```

The final estimated parcellation: figs/demo_jde_jpde/jpde_final_parcellation.png

Gibbs Sampling implementation (sandbox)

pyhrf provides tools to implement Gibbs Sampling for a prototyping purpose. The following features are available:

- control of initialization (section *Initialization*)
- trajectory monitoring (section gibbs_sampler_trajectory)
- online mean and variance computation
- checking of error w.r.t. ground-truth (in the context of inference)
- short profiling of sample steps

The implementation requires the subclassing of two classes:

- The top-level Gibbs Sampler: GibbsSampler class
- Variables: GSVariable class. One for each model variable.

In the following, a *Starting example* is presented where the Gibbs Sampler is used to fit the trivial model $y = I_n x + noise$ where y is a noisy data vector and x an unknown scalar.

This example is then complexified to illustrate the available features.

Computational efficiency

"For a prototyping purpose" means that code is not fully optimal w.r.t. to computation speed and memory. In the case of direct sampling, the time spent during the overall sampling loop is critical, hence the proposed tools are suitable only for testing. Once testing is done, one should consider implementing the validated sampler in a compiled language. In the case of inference, the time spent during the overall sampling loop may be negligible compared with the time spent during each sampling step. In this case, the critical sampling steps can be implemented in C-code and the pyhrf Gibbs implementation can be relatively fast.

Starting example

Consider the following model, where y is the noisy observed data of size N and we want to recover the unknown scalar value x:

$$\begin{aligned} y &= xI_N + b \\ b &\sim \mathcal{N}(0, \sigma^2 I_N), \sigma^2 = 0.04 \\ x &\sim \mathcal{N}(0, v_x), v_x = 1000. \end{aligned}$$

The variance of x is set to a high value to get a flat prior. The variance of the noise is also assumed to be known.

Conditional posterior for x:

$$(x|\sigma^2, v_x, y) \sim \mathcal{N}\left(\frac{\sum_N y_n}{N(1/\sigma^2 + 1/v_x)}, \frac{\sigma^2 v_x}{N(v_x + \sigma^2)}\right)$$

The sampling of "x" is handled by creating a specific subclass of pyhrf.sandbox.stats.GSVariable. The basic required elements are:

- the declaration of the name of the variable
- the setting of its initialization (here set to 0.)
- the implementation of the method *GSVariable.sample* that generates a new sample conditionally to the current other variable samples and data.

```
# -*- coding: utf-8 -*-
import numpy as np
from pyhrf.sandbox.stats import GibbsSampler, GSVariable

class GSVar_X(GSVariable):

    def __init__(self):
        """
        Create a variable object handling the sampling of variable 'x'.
        The sampling init value is set to 0.
        """
        GSVariable.__init__(self, 'x', initialization=0.)

    def sample(self):
```

```
        """
        Sample variable X according to:
        N( \sum_N y_n / (N * (1/v_x + \sigma^2)),
           v_x \sigma^2 / (N (v_x + \sigma^2) )
        """

        # retrieve other quantities:
        y = self.get_variable_value('y')
        s2 = self.get_variable_value('noise_var')
        x_prior_var = self.get_variable_value('x_prior_var')

        # Do the sampling:
        post_var = 1 / (y.size * (1/s2 + 1/x_prior_var))
        post_mean = y.sum() * post_var / s2
        return np.random.randn() * post_var + post_mean


class MyGS(GibbsSampler):

    def __init__(self):
        GibbsSampler.__init__(self, [GSVar_X()], nb_its_max=100)

# Generate some data
x_true = 1.
y = x_true + np.random.randn(500) * .2

# Instanciate & run the Gibbs Sampler
gs = MyGS()
gs.set_variables({'y':y, 'noise_var': .04, 'x_prior_var':1000.})
gs.run()

# Grab outputs:
outputs = gs.get_outputs()
x_pm = outputs['x_obs_mean']
print 'Posterior Mean estimate of x:', x_pm
```

Initialization

Several initialization scenarios are available and are specified by the argument *initialization* of GSVariable.__init__:

- Using a custom value: either a scalar or a numpy array.
- Random initialization. The method *GSVariable.get_random_init* must be implemented.
- Initialization to the true value (in the context of inference). The true value must be set by the method *GSVariable.set_true_value*
- Custom initialization. The method *GSVariable.get_custom_init* must be implemented.

```
# -*- coding: utf-8 -*-
import numpy as np
from pyhrf.sandbox.stats import GSVariable, GibbsSampler

class GSVar_X(GSVariable):

    def __init__(self):
        """
```

```
        Create a variable object handling the sampling of variable 'x'.
        """
        GSVariable.__init__(self, 'x', initialization=0.)

    def sample(self):
        """
        Sample variable X according to:
        N( \sum_N y_n / (N * (1/v_x + \sigma^2)),
           v_x \sigma^2 / (N (v_x + \sigma^2) )
        """

        # retrieve other quantities:
        y = self.get_variable_value('y')
        s2 = self.get_variable_value('noise_var')
        x_prior_var = self.get_variable_value('x_prior_var')

        # Do the sampling:
        post_var = 1 / (y.size * (1/s2 + 1/x_prior_var))
        post_mean = y.sum() * post_var / s2
        return np.random.randn() * post_var + post_mean

    def get_custom_init(self):
        """ Function called at Gibbs Sampling initialization
        if initialization scenario = 'custom'.
        Here, the mean of input data is a 'good guess'.
        """
        y = self.get_variable('y')
        return y.mean()

    def get_random_init(self):
        """ Function called at Gibbs Sampling initialization
        if initialization scenario = 'random'.
        Here, return a sample of the prior x ~ N(0,v_x)
        """
        v_x = self.get_variable_value('x_prior_var')
        return np.random.randn() * v_x**.5


class MyGS(GibbsSampler):

    def __init__(self):
        GibbsSampler.__init__(self, [GSVar_X()], nb_its_max=100)

# Generate some data
x_true = 1.
y = x_true + np.random.randn(500) * .2

# default init of variable x (zero):
gs = MyGS()
gs.set_variables({'y':y, 'noise_var': .04, 'x_prior_var':1000.})
gs.set_true_value('x', x_true)
gs.run()
x_pm = gs.get_outputs()['x_obs_mean']

print 'x PM with init to 0.:', x_pm

# custom init of variable x:
gs.reset()
```

```
gs.set_initialization('x', 'custom')
gs.run()
x_pm = gs.get_outputs()['x_obs_mean']

print 'x PM with custom init:', x_pm

# init of variable x to its true value:
gs.reset()
gs.set_initialization('x', 'truth')
gs.run()
x_pm = gs.get_outputs()['x_obs_mean']

print 'x PM with init to truth:', x_pm
```

Sampling history

The history of a given numpy.ndarray quantity can be "automatically" tracked provided that it is modified *inplace* during sampling (the quantity is tracked via its reference).

Two different type of quantities are considered:

- *sampled quantities*: their life cycle is the same as the gibbs samples. For example, it can be an intermediate result within a given sampling step. The tracking starts at the begining of the gibbs sampling.
- *observable quantities*: they are computed *after the burnin period* and thus do not take part in sampling. These computation are derived from the gibbs samples, hence the appelation "observable". For example, a posterior mean estimate is an observable. It is computed by default (attribute obs_mean of a GSVariable object). The tracking starts after the burnin period of the gibbs sampling.

Correpsonding to these two types, the initialization of a quantity tracking is done via:

- track_sampled_quantity(quantity_name, quantity_numpy_ndarray, pace) The tracking of "quantity_numpy.ndarray" is associated to the name "quantity_name". Recording starts at the beginning of the gibbs sampling and is incremented every "pace" iterations.
- track_obs_quantity(quantity_name, quantity_numpy_ndarray, pace) The tracking of "quantity_numpy.ndarray" is associated to the name "quantity_name" and recording is performed every "pace" iterations. Recording starts after the burnin period of the gibbs sampling and is incremented every "pace" iterations.

```
# -*- coding: utf-8 -*-
import numpy as np
from pyhrf.sandbox.stats import GibbsSampler, GSVariable

class GSVar_X(GSVariable):

    def __init__(self):
        """
        Create a variable object handling the sampling of variable 'x'.
        The sampling init value is set to 0.
        """
        GSVariable.__init__(self, 'x', initialization=0.)

    def init_sampling(self):
        # The tracked quantities have to inited here so that
        # their references remain the same during sampling
```

```
        self.post_mean = np.zeros_like(self.current_value)
        self.post_var = np.zeros_like(self.current_value)

        self.track_sampled_quantity('smpl_post_mean', self.post_mean)
        self.track_sampled_quantity('smpl_post_var', self.post_var)

    def sample(self):
        """
        Sample variable X according to:
        N( \sum_N y_n / (N * (1/v_x + \sigma^2)),
           v_x \sigma^2 / (N (v_x + \sigma^2) )
        """

        # retrieve other quantities:
        y = self.get_variable_value('y')
        s2 = self.get_variable_value('noise_var')
        x_prior_var = self.get_variable_value('x_prior_var')

        # Do the sampling:
        self.post_var[:] = 1 / (y.size * (1/s2 + 1/x_prior_var))
        self.post_mean[:] = y.sum() * self.post_var / s2

        return np.random.randn() * self.post_var + self.post_mean


class MyGS(GibbsSampler):

    def __init__(self, sample_hist_pace, obs_hist_pace):
        GibbsSampler.__init__(self, [GSVar_X()], nb_its_max=10,
                              sample_hist_pace=sample_hist_pace,
                              obs_hist_pace=obs_hist_pace)

# Generate some data
x_true = 1.
y = x_true + np.random.randn(500) * .2

# Instanciate & run the Gibbs Sampler
gs = MyGS(sample_hist_pace=1, obs_hist_pace=1)
gs.set_variables({'y':y, 'noise_var': .04, 'x_prior_var':1000.})
gs.run()

# Grab tracked quantity:
outputs = gs.get_outputs()
print 'Tracking of post mean:'
print outputs['smpl_post_mean']

print 'Tracking of post var:'
print outputs['smpl_post_var']
```

Error w.r.t. ground truth

When Gibbs Sampling is used in the context of inference and a ground truth (eg simulated value) is available for a given variable, one wants to compare the final estimate to this ground truth.

```python
# -*- coding: utf-8 -*-
import numpy as np
import pyhrf
from pyhrf.sandbox.stats import GibbsSampler, GSVariable

class GSVar_X(GSVariable):

    def __init__(self):
        """
        Create a variable object handling the sampling of variable 'x'.
        The sampling init value is set to 0.
        """
        GSVariable.__init__(self, 'x', initialization=0.)

    def init_sampling(self):
        # The tracked quantities have to inited here so that
        # their references remain the same during sampling
        self.post_mean = np.zeros_like(self.current_value)
        self.post_var = np.zeros_like(self.current_value)

        self.track_sampled_quantity('smpl_post_mean', self.post_mean)
        self.track_sampled_quantity('smpl_post_var', self.post_var)

    def sample(self):
        """
        Sample variable X according to:
        N( \sum_N y_n / (N * (1/v_x + \sigma^2)),
           v_x \sigma^2 / (N (v_x + \sigma^2) )
        """

        # retrieve other quantities:
        y = self.get_variable_value('y')
        s2 = self.get_variable_value('noise_var')
        x_prior_var = self.get_variable_value('x_prior_var')

        # Do the sampling:
        self.post_var[:] = 1 / (y.size * (1/s2 + 1/x_prior_var))
        self.post_mean[:] = y.sum() * self.post_var / s2

        return np.random.randn() * self.post_var + self.post_mean


class MyGS(GibbsSampler):

    def __init__(self, sample_hist_pace, obs_hist_pace):
        GibbsSampler.__init__(self, [GSVar_X()], nb_its_max=10,
                              sample_hist_pace=sample_hist_pace,
                              obs_hist_pace=obs_hist_pace)

# Generate some data
x_true = 1.
y = x_true + np.random.randn(500) * .2

# Instanciate & run the Gibbs Sampler
gs = MyGS(sample_hist_pace=1, obs_hist_pace=1)
gs.set_variables({'y':y, 'noise_var': .04, 'x_prior_var':1000.})
gs.set_true_value('x', x_true)
gs.run()
```

```

pyhrf.verbose.set_verbosity(3)
# check with default atol=0.1, rtol=0.1
gs.check_against_truth()

# check with atol=0.01, rtol=0.01
gs.check_against_truth(default_rtol=0.01, default_atol=0.01)
```

2.4 pyhrf package

2.4.1 Subpackages

pyhrf.boldsynth package

Subpackages

pyhrf.boldsynth.pottsfield package

Submodules

pyhrf.boldsynth.pottsfield.swendsenwang module

Submodules

pyhrf.boldsynth.field module

pyhrf.boldsynth.hrf module

pyhrf.boldsynth.scenarios module

pyhrf.boldsynth.spatialconfig module

pyhrf.jde package

Subpackages

pyhrf.jde.nrl package

Submodules

pyhrf.jde.nrl.ar module

pyhrf.jde.nrl.base module

pyhrf.jde.nrl.bigaussian module

pyhrf.jde.nrl.bigaussian_drift module

pyhrf.jde.nrl.gammagaussian module

pyhrf.jde.nrl.habituation module

pyhrf.jde.nrl.trigaussian module

Submodules

pyhrf.jde.asl module

pyhrf.jde.asl_2steps module

pyhrf.jde.asl_physio module

pyhrf.jde.asl_physio_1step module

pyhrf.jde.asl_physio_1step_params module

pyhrf.jde.asl_physio_det_fwdm module

pyhrf.jde.asl_physio_hierarchical module

pyhrf.jde.asl_physio_joint module

pyhrf.jde.beta module

pyhrf.jde.drift module

pyhrf.jde.hrf module

pyhrf.jde.jde_multi_sess module

pyhrf.jde.jde_multi_sujets module

pyhrf.jde.jde_multi_sujets_alpha module

pyhrf.jde.models module

pyhrf.jde.noise module

pyhrf.jde.samplerbase module

pyhrf.jde.wsampler module

pyhrf.sandbox package

Submodules

pyhrf.sandbox.data_parser module

pyhrf.sandbox.func_BMA_consensus_clustering module

pyhrf.sandbox.make_parcellation module

pyhrf.sandbox.parcellation module

pyhrf.sandbox.physio module

pyhrf.sandbox.physio_params module

pyhrf.sandbox.stats module

pyhrf.stats package

Submodules

pyhrf.stats.misc module

pyhrf.stats.random module

pyhrf.tools package

Submodules

pyhrf.tools.aexpression module

pyhrf.tools.backports module

pyhrf.tools.cpus module

pyhrf.tools.message module

pyhrf.tools.misc module

pyhrf.ui package

Submodules

pyhrf.ui.analyser_ui module

pyhrf.ui.glm_analyser module

pyhrf.ui.glm_ui module

pyhrf.ui.jde module

pyhrf.ui.rfir_ui module

pyhrf.ui.treatment module

pyhrf.ui.vb_jde_analyser module

pyhrf.ui.vb_jde_analyser_asl_fast module

pyhrf.ui.vb_jde_analyser_bold_fast module

pyhrf.vbjde package

Submodules

pyhrf.vbjde.vem_asl_models_fast module

pyhrf.vbjde.vem_asl_models_fast_ms module

pyhrf.vbjde.vem_bold module

pyhrf.vbjde.vem_bold_constrained module

pyhrf.vbjde.vem_bold_models_fast module

pyhrf.vbjde.vem_bold_models_fast_ms module

pyhrf.vbjde.vem_tools module

pyhrf.xmliobak package

Submodules

pyhrf.xmliobak.xmlbase module

pyhrf.xmliobak.xmlmatlab module

pyhrf.xmliobak.xmlnumpy module

2.4.2 Submodules

pyhrf.configuration module

pyhrf.core module

pyhrf.glm module

pyhrf.graph module

pyhrf.grid module

pyhrf.ndarray module

pyhrf.paradigm module

pyhrf.parallel module

pyhrf.parcellation module

pyhrf.plot module

pyhrf.rfir module

pyhrf.surface module

pyhrf.usemode module

pyhrf.version module

pyhrf.xmlio module

2.5 Changelog

2.5.1 Current development

2.5.2 Release 0.4.3

2016/07/08

Fixes

- Remove non-existing tests in devel mode
- Correct typo in documentation
- Correct list display in documentation
- Fix bug for numpy >= 1.11.1 version
- Fix bug in contrasts computation
- clean tmp folders after some unitary tests
- Fix VEM script example

2.5.3 Release 0.4.2

2016/07/07

Fixes

- **Fix VEM algorithm**
  - fix convergence criteria computation
  - fix underflow and overflow in labels expectation (set labels to previous value if necessary)
- Continue to clean setup.py
- fix some DeprecationWarning that will become Exceptions in the future
- fix detection of parcellation files
- fix for scikit-learn version >= 0.17
- fix bugs with matplotlib versions >1.4
- fix bug with Pillow latest version (see #146)
- fix bug with numpy when installing in virtual environment (see commit a971656)
- fix the zero constraint on HRF borders

Enhancements

- optimize some functions in vem_tools
- rewrite and optimize all VEM steps
- remove old calls to verbose module and replaced them by logging standard library module
- update website documentation

New

- **Updating documentation**
  - updating theme
  - fixing some reST and display errors
- autodetect cpus number (mainly to use on cluster and not yet documented)
- add covariance regularization matrix
- load contrasts from SPM.mat
- save contrasts in the same order that the xml configuration file
- compute and save PPMs
- Add multisession support for VEM BOLD and ASL
- add cosine drifts to VEM
- add commandline for VEM
- add Stanford Willard Parcellation

2.5.4 Release 0.4.1.post1

Fixes:

- missing function (#135)
- nipy version required for installation (#134)

2.5.5 Release 0.4.1

Fixes:

- logging level not set by command line (#113)
- error with VEM algorithm (#115)

Enhancements:

- clean and update setup.py (#84)
- update travis configuration file (#123)

2.5.6 Release 0.4

2015/03/19

API Changes:

- Deprecate verbose module and implements logging module instead

Fixes:

- clean up setup.py

Enhancements:

No changelog for previous version

CHAPTER 3

Licence and authors

PyHRF is currently under the CeCILL licence version 2. Originally developed by the former LNAO (Neurospin, CEA), pyHRF is now entering (since Sep 2014) in a new era under the joint collaboration of the the Parietal team (INRIA Saclay) and the MISTIS team (INRIA Rhones-Alpes).

People who have significantly contributed to the development are (by chronological order): Thomas Vincent[1,3], Philippe Ciuciu[1,2], Lotfi Chaari[3], Solveig Badillo[1,2], Christine Bakhous[3], Aina Frau-Pascual[2,3] and Thomas Perret[3]

1. CEA/DRF/I2BM/NeuroSpin, Gif-Sur-Yvette, France 1. INRIA/CEA Parietal, Gif-Sur-Yvette, France 2. INRIA, MISTIS, Grenoble, France

3.1 Contacts

philippe.ciuciu@cea.fr, florence.forbes@inria.fr, thomas.perret@inria.fr, thomas.tv.vincent@gmail.com