h-BNM Documentation

Release 1.0.0.

Murat Demirtas

February 06, 2019

CONTENTS

1 Overview **3**
1.1 Requirements . . . 3
1.2 Structure . . . 3
1.3 Usage . . . 5

2 Contents **9**
2.1 Docstrings . . . 9

Python Module Index **21**

Index **23**

Heterogeneous Brain Network Model (h-BNM) is a biophysically-based large-scale computational modeling toolbox. The scripts were developed in Python.

CHAPTER ONE

OVERVIEW

1.1 Requirements

Requirements can be found in `requirements.txt` in the project's root directory.

Note: This package was developed in python 2.7.12.

1.1.1 Python packages

Required python packages and versions used for code development:

- numpy (1.11.1)
- pandas (0.18.1)
- scipy (0.18.1)
- sympy (0.7.6)
- h5py (2.5.0)
- nibabel (2.1.0)

1.1.2 Other software

Connectome Workbench, an open-source tool developed by the Human Connectome Project (HCP) consortium, is also required for GUI- and command line-based visualizations of neuroimaging files. Documentation for all Connectome Workbench commands can be found here.

1.2 Structure

The package is structured in the following manner:

```
h-BNM
|
|_ hbnm
|
|_ _ _ data
|
|_ _ _ docs
|
|_ _ _ model
|_ _ _ _ _ __init__.py
|_ _ _ _ _ dmf.py
|_ _ _ _ _ hemo.py
|_ _ _ _ _ sim.py
|_ _ _ _ _ utils.py
|
|_ _ _ scripts
|_ _ _ _ _ __init__.py
|_ _ _ _ _ optimization.py
|_ _ _ _ _ model_optimization.py
|_ _ _ _ _ model_sampling.py
|
|_ _ _ __init__.py
|_ _ _ bnm.py
|_ _ _ io.py
|_ _ _ pmc.py
```

1.2.1 hbnm

bnm.py `bnm.Bnm` is a wrapper for the main brain network model class `model.dmf.Dmf`. The function of this file to easily access and manipulate model parameters for model fitting.

pmc.py Contains functions and class `pmc.Pmc` for hierarchical Population Monte Carlo (hPMC) algorithm.

io.py Specification of paths to package directories and files. Load and save data in various formats including: `.hfd5`, `.mat`, `.xlsx`, `.csv`, `.txt`, `.nii`.

1.2.2 hbnm.model

model.dmf The class `model.dmf.Model` contains all the functionality of the main brain network model including implementation of heterogeneity map parametrization, analytical computation of feedback inhibition, linearization of the dynamical system, computational of covariance/correlation, numerical integration...etc.

model.hemo The class `model.hemo.Baloon` contains the functions to construct the Jacobian of the extended hemodynamic system to compute analytical estimate of the model BOLD FC.

model.sim The class `model.sim.Sim` contains the functions to setup and store the variables for numerical integration.

model.utils Contains miscellaneous auxiliary functions

1.2.3 hbnm.scripts

optimization.py An example script containing homogeneous and heterogeneous modeling objects for optimization.

model_optimization.py An example parallelized optimization script.

run_optimization.sh A bash script to execute model optimization scripts.

model_sampling.py An example script to resample and visualize the results

1.3 Usage

Note: The package must be saved to a directory included in the local `$PYTHONPATH` environment variable; for users running macOS, this variable can be configured in `~/.bash_profile`.

1.3.1 Create model optimization class

To use Population Monte Carlo optimization an appropriate class should be inherited from the base `pmc.Pmc`. Specifically, the methods *get_appendices*, *set_prior*, *run_particle*. *generate_data*, and *distance_fuction* are abstract methods and they define the model optimization approach. For example, a simple class for homogeneous model would have following structure:

```
from hbnm.pmc import Pmc
import numpy as np
from scipy import stats
from hbnm.model.utils import subdiag, linearize_map, normalize_sc, fisher_z
class Homogeneous(Pmc):
def get_appendices(self, run_id):
    return

def set_prior(self):
    self.prior = [stats.uniform(0.001, 5.0),
                  stats.uniform(0.001, 15.0),
                  stats.uniform(0.001, 10.0)]

def run_particle(self, theta):
    self.model.set('w_EI', theta[0])
    self.model.set('w_EE', theta[1])
    self.model.set('G', theta[2])

def generate_data(self):
    self.model.moments_method(BOLD=True)
    return subdiag(self.model.get('corr_bold'))

def distance_function(self, synthetic_data):
    fit = pearsonr(self.fc_objective, fisher_z(synthetic_data))[0]
    penalty = (self.fc_objective.mean() - synthetic_data.mean()) ** 2
    distance = 1.0 - (fit - penalty)
    return distance
```

Likewise, a heterogeneous model class stucture would have following structure:

```
from hbnm.pmc import Pmc
import numpy as np
from scipy import stats
from hbnm.model.utils import subdiag, linearize_map, normalize_sc, fisher_z
class Heterogeneous(Pmc):
def get_appendices(self, run_id):
    return

def set_prior(self):
    self.prior = [stats.uniform(0.001, 2.), stats.uniform(0.0, 2.5),
                  stats.uniform(0.001, 5.0), stats.uniform(0.0, 15.0),
                  stats.uniform(0.001, 5.0)]

def run_particle(self, theta):
    self.model.set('w_EI', (theta[0], theta[1]))
    self.model.set('w_EE', (theta[2], theta[3]))
    self.model.set('G', theta[4])

def generate_data(self):
    self.model.moments_method(BOLD=True)
    return subdiag(self.model.get('corr_bold'))

def distance_function(self, synthetic_data):
    fit = pearsonr(self.fc_objective, fisher_z(synthetic_data))[0]
    penalty = (self.fc_objective.mean() - synthetic_data.mean()) ** 2
    distance = 1.0 - (fit - penalty)
    return distance
```

Note that in these examples we provided an additional penalty term controlling the average magnitudes in the model FCs. Here this penalty term involves the squared difference between average empirical and model FCs. Adding a penalty term is highly recommended since Pearson similarity is demeaned. Therefore, when the Pearson similarity between model and empirical FCs is high, the magnitudes can be unrealistically high or low in the model FC. Alternatively, custom distance metrics can be implemented such as Euclidean distance.

These examples performs a simple Population Monte Carlo, where the distance measure is based on the Pearson correlation between average model and empirical FCs. Hierarchical Population Monte Carlo implementation would involve the subjectwise distance between model and emepirical FCs. For example, in this case the distance function can be written as:

```
from hbnm.model.utils import vcorrcoef
def distance_function(self, synthetic_data):
    fit = vcorrcoef(self.fc_objective.T, synthetic_data).mean()
    penalty = (self.fc_objective.mean() - synthetic_data.mean())**2
    distance = 1.0 - (np.mean(fit) - penalty)
    return distance
```

The example scripts were provided as `optimization.py` inside `scripts` folder. To perform optimization, the inherited class should be defined by providing an input and output directory to save the results. Then, the class object should be initialized by providing structural connectivity (SC), heterogeneity map (hmap), the empirical FC (fc_objective), number of particles (n_particles) and the initial rejection threshold (rejection_threshold).

```
from optimization import Heterogeneous
pmc_opt = Heterogeneous(input_dir, output_dir + '/')
pmc_opt.initialize(sc, fc=fc_obj, gradient=hmap, n_particles=n_samples,
                   rejection_threshold=rejection_threshold)
```

For homogeneous model the gradient parameter should be set to `None`. A naive implementation of the optimization may involve a simple loop such as:

```
n_iterations = 25
n_samplers = 5
for iteration in range(n_iterations):
    for sampler_id in range(n_samplers):
        pmc_opt.run(sampler_id)
    pmc_opt.wrap(n_samplers)
```

However, this implementation might not be computationally efficient for large number of particles and iterations. Therefore, the optimizer class can be parallelized. The example script `model_optimization.py` can be executed from the terminal by providing keyword arguments for:

- model type: heterogeneous or homogeneous
- fc objective: average or individual

- minimum number of samples / sampler
- number of samplers
- sampler id
- optimization task: sampler or wrapper
- append to output directory

Therefore, a parallelized executing can be as follows:

```
filename="model_optimization.py"
model_name="homogeneous"
objective="average"
n_particles=10
n_tasks=5
n_iterations=25
output_directory="homogeneous_average_fc"

for iter in {1..25}
do
for samplers in {0..4}
do
python $filename $model_name $objective $n_particles $n_tasks $samplers sampler $output_directory &
done
wait
python $filename $model_name $objective $n_particles $n_tasks 0 wrapper $output_directory
done
```

This script will execute 5 samplers in parallel generating 10 particles in each sampler. For each iteration, these samplers will be collected (5x10=50 particles in total) and saved as an hdf5 file inside the output directory.

The example optimizations were performed using SC and FC matrices involving 68 cortical regions as in Deco et al., 2017. T1w/T2w map was provided from Human Connectome Project (HCP), averaged across 220 subjects. The dense T1w/T2w was obtained from Balsa database, and it was provided for the multimodal parcellation study Glasser et al., 2016 (see <https://balsa.wustl.edu/study/show/RVVG>). The dense T1w/T2w images parcellated into 68 regions (Figure 1.1.).

The optimization was performed using Population Monte Carlo approach with 25 iterations and 50 samples.

1.3.2 Analyzing the results

Once the optimization in complete, the approximated posterior distributions can be loaded from the `.hdf5` file in the output directory. In this example, the last iteration was 25, therefore the file `iteration_25.hdf5` was loaded. In the iteration files, among others, the key `theta` provides the particles. For example, the heterogeneous model optimization can be loaded as:

```python
from hbnm.io import Data
import numpy as np
import os

current_path = os.getcwd()
parent_path = os.path.abspath(os.path.join(current_path, os.pardir))
input_dir = parent_path + '/data/'
output_dir = parent_path + '/outputs/'

"""
Load data
"""
```

```
fin = data.load('heterogeneous_average_fc/iteration_25.hdf5', from_output=True)
theta_heterogeneous = fin['theta'].value
fin.close()
```

Then, the for an arbitrary particle the model can be initialized and executed as:

```
from hbnm.bnm import Bnm

heterogeneous = Bnm(sc, gradient=hmap)
heterogeneous.set('w_EI', (theta_heterogeneous[0,0], theta_heterogeneous[1,0]))
heterogeneous.set('w_EE', (theta_heterogeneous[2,0], theta_heterogeneous[3,0]))
heterogeneous.set('G', theta_heterogeneous[4,0])
heterogeneous.moments_method()
model_FC = heterogeneous.get('corr_bold')
```

An example output (provided in `scripts/model_sampling.py`) can be seen in Figure 2.1.

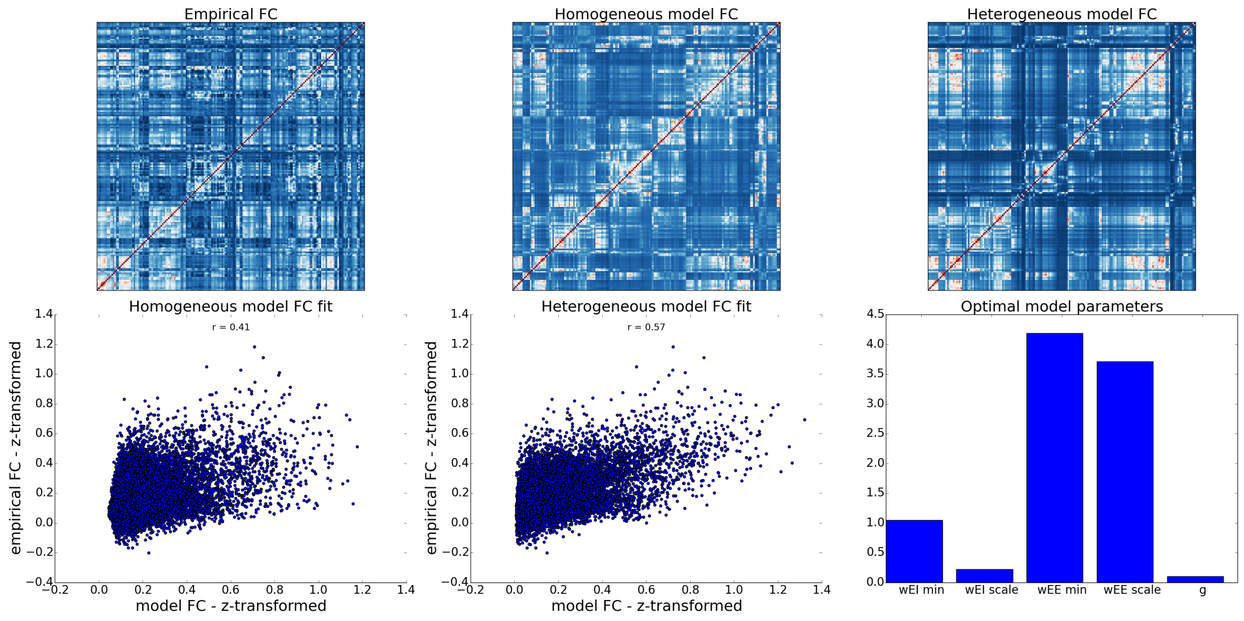

Fig. 1.1: Example homogeneous and heterogeneous model fit.

CHAPTER

TWO

CONTENTS

2.1 Docstrings

All class and function definitions are located in `hbnm/`. The core functionality is encapsulated in *`hbnm.bnm.Bnm`*.

2.1.1 Module hbnm.bnm

class `hbnm.bnm.`**`Bnm`**(*sc*, *gradient=None*, **args*, ***kwargs*)
Wrapper class for the large-scale computational model.

`check_stability`(*compute_FIC=True*)
Check stability of the system

Parameters **`compute_FIC`** (*bool, optional*) – If True, the feedback inhibition is recomputed.

Returns The truth value for the largest eigenvalue of the system is smaller than 0

Return type bool

`disconnect`()

Notes

This method creates another Model object named 'dmf_disconnected', the total input to each region
is set to long-range input level and global coupling parameter is set to 0.

`get`(*parameter*)
Get method for the model

Parameters **`parameter`** (*str*) – The parameter to get, such as 'w_EE', 'w_EI', 'corr_bold'...etc.

`moments_method`(*BOLD=True*, **args*, ***kwargs*)
Calls moments_method method in the Model class

Parameters **`BOLD`** (*bool, optional*) – If True, the linearized covariances are computed for hemodynamic system (BOLD)

`psd_bold`(*freqs*)
Calculates the power spectral density for the BOLD activity

Parameters **`freqs`** (*ndarray*) – The frequency bins for which the power spectral density is computed

Returns Power spectral density

Return type ndarray

`psd_syn` (*freqs*, *pop='E'*)
Calculates the power spectral density for the synaptic gating variables

Parameters

- **freqs** (*ndarray*) – The frequency bins for which the power spectral density is computed
- **pop** (*str, optional*) – If 'E', the power spectral density is computed for excitataory populations, if 'I', the power spectral density is computed for inhibitory populations.

Returns Power spectral density

Return type ndarray

`set` (*parameter*, *values*, *separate=False*)
Set method for the model

Parameters

- **parameter** (*str*) – The name of the parameter, such as 'w_EE' or 'w_EI'
- **values** (*float or tuple*) – The parameter values to set. The values should be tuple for heterogeneous model.
- **separate** (*bool, optional*) – If True, the value passes to left and right hemisphere separately. This flag is required only if the global coupling parameter is separate for each hemisphere.

2.1.2 Module hbnm.pmc

class `hbnm.pmc.Pmc` (*input_directory*, *output_directory*, *verbose=True*)
Class for particle monte carlo optimization

This class is derived from the Python package SimpleABC: A Python package for Approximate Bayesian Computation Version 0.2.0

Available in http://rcmorehead.github.io/SIMPLE-ABC/ Sunnaker et al. - [Approximate Bayesian Computation](http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3547661/)

`distance_function` (*synthetic_data*)

An abstract method to calculate distance. For example: model_fit = pearsonr(self.fc_objective, synthetic_data)[0] return 1.0 - model_fit

Returns Summary distance statistic

Return type float

`generate_data` ()
An abstract method to execute the linearization and generate model FC. For example: self.model.moments_method(BOLD=True) return self.model.get('corr_bold')

Returns The model FC measure

Return type ndarray

`get_appendices` (*run_id*)
An abstract method to provide additional parameters to save

initialize(*sc*, *fc=None*, *gradient=None*, *n_particles=10*, *rejection_threshold=None*, **args*, ***kwargs*)
Initialization for the optimization.

Parameters

- **sc** (*ndarray or list*) – Empirical structural connectivity matrix
- **fc** (*ndarray*) – Empirical functional connectivity
- **gradient** (*ndarray or list*) – Heterogeneity map to parametrize the model
- **n_particles** (*int*) – Maximum number of particles
- **rejection_threshold** (*float*) – Initial rejection threshold

Notes

This method requires a list for SC and heterogeneity map, if the model will be fitted for left and right hemispheres separately. The dimensions of the empirical functional connectivity should be (N_connextions x N_subjects), where N_connections is the number of connections, i.e. N x (N-1)/2, and N_subjects is the number of subjects.

run(*run_id*)
Run a single iteration for the particle Monte Carlo algorithm

Parameters **run_id** (*int*) – The labels for the run (required for paralelization)

Notes

This method executes a single iteration for the PMC. For paralelize the code, it samples multiple batches of particles independently and saves the output to a file called, 'samples_n.npy'.

run_particle(*theta*)

An abstract method to draw a particle. An example use would be: self.model.set('w_EI',theta[0]) self.model.set('w_EE',theta[1]) ...etc.

set_prior()
Provide prior distributions for the parameters. This method requires defining prior such as: e.g. self.prior = [stats.uniform(0.001, 2.), stats.uniform(0.0, 2.5)]...etc.

wrap(*n_outputs*)
Wrapper function to collect the samples.

Parameters **n_outputs** (*int*) – Total number of samplers that are run in parallel

Notes

This method collects all the results from previously saves files ('samples_n.npy'), and then dumps them into a single HDF file (iteration_n.hdf5).

2.1.3 Module hbnm.io

class hbnm.io.**Data**(*input_dir*, *output_dir*)
An auxiliary class to facilitate loading and saving data

load (*filename*, *numeric=False*, *type=None*, *from_path=False*, *from_output=False*, **args*, ***kwargs*)
Load method

Parameters

- **file_name** (*str*) – The name of the file to load
- **numeric** (*bool, optional*) – This keyword is required only when load csv or txt files
- **type** (*str, optional*) – The type of the file. If None the method will guess the file type based on its extension
- **from_output** (*bool, optional*) – If True the file will be loaded from the output directory, otherwise it will be loaded from the input directory

Returns File object (depends on the extension of the file)

Return type object

Notes

This method supports files with extensions '.hdf5', '.mat', '.npy/.npz', '.xlsx', '.txt'. It loads the file using the appropriate method based on the extensions. E.g. uses pandas for '.xlsx'

load_cifti (*filename*, *from_input=False*)
CIFTI loader

Parameters **filename** (*str*) – The name of the file to load

Returns The data from the CIFTI file

Return type ndarray

load_nifti (*filename*)
NIFTI loader

Parameters **filename** (*str*) – The name of the file to load

Returns The data from the NIFTI file

Return type ndarray

save (*filename*, *data=None*, *numeric=False*, **args*, ***kwargs*)
Save method

Parameters

- **filename** (*str*) – The name of the file to save
- **data** (*ndarray, optional*) – The data to be saved. It is not necessary if multiple data will be saved.
- **numeric** (*bool, optional*) – This keyword is required only when saving csv or txt files

Returns File object (depends on the extension of the file)

Return type object

Notes

This method supports files with extensions '.hdf5', '.mat', '.npy/.npz', '.xlsx', '.txt'. It loads the file using the appropriate method based on the extensions. E.g. uses pandas for '.xlsx'

save_cifti(*output*, *filename*, *template*)

NIFTI save

Parameters

- **output** (*ndarray*) – The data to save as CIFTI file
- **filename** (*str*) – The name of the file to save
- **template** (*str*) – The name of the template file to be used when generating the CIFTI

save_nifti(*output*, *filename*, *template*)

NIFTI save

Parameters

- **output** (*ndarray*) – The data to save as NIFTI file
- **filename** (*str*) – The name of the file to save
- **template** (*str*) – The name of the template file to be used when generating the NIFIT

2.1.4 Module hbnm.model

class hbnm.model.dmf.**Model**(*sc*, *g=0*, *norm_sc=True*, *hmap=None*, *wee=(0.15, 0.0)*, *wei=(0.15, 0.0)*, *syn_params=None*, *bold_params='obata'*, *verbose=True*)

Class for the large-scale computational model with optional heterogeneous parametrization of w^{EE} and w^{EI}, based on provided heterogeneity map.

G

Returns Global coupling strength.

Return type ndarray

I_ext

Returns External current

Return type ndarray

J_NMDA

Returns Effective NMDA conductance

Return type float

Q

Returns Input covariance matrix

Return type ndarray

SC

Returns Empirical structural connectivity.

Return type ndarray

corr

Returns Correlation matrix (model FC) for synaptic system

Return type ndarray

corr_bold

Returns Correlation matrix (model FC) for BOLD

Return type ndarray

`cov`

Parameters **`full`** (*bool, optional*) – If True, returns the full covariance matrix.

Returns Covariance matrix of linearized fluctuations about fixed point.

Return type ndarray

`cov_bold`

Returns Covariance matrix of linearized fluctuations for BOLD

Return type ndarray

`csd` (*freqs*, *pop='E'*)

Computes cross-spectral density of synaptic variables.

Parameters

- **`freqs`** (*ndarray or list*) – An array or list containing the frequency bins for which the CSD will be computed
- **`pop`** (*str, optional*) – If 'E', the CSD of excitatory populations will return (default). if 'I', the CSD of inhibitory populations will return. Any other string will be ignored and the full CSD will be returned

Returns **csd** – CSD of the synaptic system in shape NxNxf, where N is the number of regions and f is the number of frequency bins

Return type ndarray

`csd_bold` (*freqs*)

Computes cross-spectral density of hemodynamic variables.

Parameters **`freqs`** (*ndarray or list*) – An array or list containing the frequency bins for which the CSD will be computed

Returns CSD of the BOLD transformed hemodynamic system in shape NxNxf, where N is the number of regions and f is the number of frequency bins.

Return type ndarray

Notes

The computation is performed for the hemodynamic system, the results are returned for BOLD signals.

`evals`

Returns Eigenvalues of Jacobian matrix.

Return type ndarray

`evecs`

Returns Left Eigenvextors of Jacobian matrix.

Return type ndarray

`hmap`

Returns Heterogeneity map values

Return type ndarray

integrate(*t*, *dt=0.0001*, *n_save=10*, *stimulation=0.0*, *delays=False*, *distance=None*, *velocity=None*, *include_BOLD=True*, *from_fixed=True*, *sim_seed=None*, *save_mem=False*)

Computes cross-spectral density of hemodynamic variables.

Parameters

- **t** (*int*) – Total simulation time in seconds.
- **dt** (*float, optional*) – Integration time step in seconds. By default dt is 0.1 msec.
- **n_save** (*int, optional*) – Sampling rate (time points). By default n_save is 10, therefore in dt is 0.1 msec, all the variables will be sampled at 1 msec.
- **stimulation** (*ndarray or float, optional*) – An array or matrix containing external currents if required. The size of array should match to the number time points (i.e. int(t / dt + 1)) (0.0 by default)
- **delays** (*bool, optional*) – If True, delays are included during the integration (False by default)
- **distance** (*ndarray, optional*) – The distance matrix, If delays will be taken into account. The distance matrix should contain the euclidean or geodesic distance between regions in mm.
- **velocity** (*float, optional*) – The conduction velocity in m/sec, if conduction delays are not ignored.
- **include_BOLD** (*boolean, optional*) – If True, the simulation will also include hemodynamic model and BOLD signals (True by default)
- **from_fixed** (*boolean, optional*) – If True, the simulation will begin using steady state values of the parameters, otherwise the last available values will be used (i.e. from previous simulations...etc.)
- **sim_seed** (*int, optional*) – The seed for random number generator.

Returns

Return type None

Notes

This method simulates the system for the given simulation time and the parameter values are stored. After successfull simulation, The excitatory synaptic variables can be obtained by .sim.S_E or BOLD signals can be obtained by .sim.y

jacobian

Returns Jacobian of linearized fluctuations about fixed point.

Return type ndarray

moments_method(*bold=False*, *use_lyapunov=False*)

Computes the linearized covariance and the correlation matrices between model variables.

Parameters

- **bold** (*boolean, optional*) – if True, the covariance and correlation are computed for the extended hemodynamic system (BOLD), otherwise it computes only for the synaptic system of equations. (False by default)

- **`use_lyapunov`** (*boolean, optional*) – if True, the builtin function scipy.linalg.solve_lyapunov is used to solve Lyapunov equations. (False by default)(not recommended, only for debugging)

Notes

The covariance and correlation matrices can be called only after performing this method.

`nc`

Returns Number of cortical areas.

Return type ndarray

`set_jacobian`(*compute_fic=True*)

Set Jacobian matrix given the model parameters.

Parameters **`compute_fic`** (*boolean, optional*) – if True, local feedback inhibition parameters (w^{IE}) are adjusted to set the firing rates of excitatory populations to ~3Hz

Returns If True, the system is stable and linearized covariance can be successfully calculated, otherwise moments method fails because the solution is not stable for the given parameters.

Return type boolean

Notes

This method should be executed before calculating the linearized covariance or performing numerical integration, each time the model parameters are modified.

`sigma`

Returns Input noise to each area

Return type ndarray

`state`

Returns All state variables, 6 rows by len(nodes) columns. Rows are I_I, I_E, r_I, r_E, S_I, S_E.

Return type ndarray

`steady_state`

Returns All steady state variables, shape 6 x nc. Rows are, respectively, I_I, I_E, r_I, r_E, S_I, S_E.

Return type ndarray

`var`

Parameters **`full`** (*bool, optional*) – If True, returns the full variance.

Returns Variances of linearized fluctuations about fixed point.

Return type ndarray

`var_bold`

Returns Variances of linearized fluctuations for BOLD.

Return type ndarray

w_EE

Returns Local recurrenm excitatory strengths.

Return type ndarray

w_EI

Returns Local excitatory to inhibitory strengths.

Return type ndarray

w_IE

Returns Feedback inhibition weights.

Return type ndarray

class `hbnm.model.hemo.`**`Balloon`**(*nc*, *linearize=False*, *parameters='obata'*)

The class containing hemodynamic response function

`BOLD_tf`(*freqs*)

The analytic solution to the transfer function of the BOLD signal y as a function of the input synaptic signal z, at a given frequency f, for the Balloon-Windkessel hemodynamic model. For derivation details see Robinson et al., 2006, BOLD responses to stimuli.

Parameters **`freqs`** (*float*) –

Returns Transfer function

Return type ndarray

`linear_step`(*dt*, *z*)

Evolve linearized hemodynamic equations by time dt, and update state variables. System evolved according to linearized Balloon - Windkessel hemodynamic model.

Parameters

- **`dt`** (*float*) – Time step of the integration
- **`z`** (*ndarray*) – Synaptic activity in each brain region

Returns

Return type none

`linearize_BOLD`(*z*, *A_syn*, *Q*)

Calculates the Jacobian of the full system. Such that three numpy arrays are produced: (2n x 2n) S covariance matrix (4n x 4n) hemodynamic covariance matrix (n x n) BOLD covariance matrix

Parameters

- **`z`** (*ndarray*) – The steady state values synaptic gating parameters
- **`A_syn`** (*ndarray*) – The Jabobian matrix for the synaptic system of equations
- **`Q`** (*ndarray*) – The input noise matrix for the synaptic system of equations

Returns

Return type none

`nonlinear_step`(*dt*, *z*)

Evolve hemodynamic equations by time dt and update state variables. System evolved according to Balloon - Windkessel hemodynamic model.

Parameters

- **`dt`** (*float*) – Time step of the integration
- **`z`** (*ndarray*) – Synaptic activity in each brain region

Returns

Return type none

`reset_state`()

Reset hemodynamic state variables to steady state values.

Returns

Return type none

`state`

All hemodynamic state variables, 5 rows by len(nodes) columns. Rows are x, v, f, q, y.

Returns The collection of hemodynamic state variables.

Return type ndarray

class `hbnm.model.sim.`**`Sim`**

`BOLD_corr`(*t_cutoff=0*)

Simulated BOLD time series correlations, omitting the first t_cutoff seconds from the time series.

`BOLD_cov`(*t_cutoff=0*)

Simulated BOLD time series covariances, omitting the first t_cutoff seconds from the time series.

`Hemo_cov`(*t_cutoff=0*)

Simulated hemodynamic quantity covariances, omitting the first t_cutoff seconds from the time series.

`S_corr`(*t_cutoff=0*)

Full simulated S FC. The first quadrant is E-E.

`S_cov`(*t_cutoff=0*)

Full simulated S covariance. The first quadrant is E-E.

`load_sim`(*sim_file*)

Reconstructs simulation outputs from pickle file.

`time_series`(*var_type='S_E'*)

Returns the simulated time series for all nodes for a given variable type, with each row corresponding to a unique node, and each column representing a point in time. var_type can be any of the following:

> 'I_I': inhibitory current [nA] 'I_E': excitatory current [nA] 'r_I': inhibitory rate [Hz] 'r_E': excitatory rate [Hz] 'S_I': inhibitory synaptic gating fraction 'S_E': excitatory synaptic gating fraction 'x': vasodilatory signal 'f': normalized inflow rate 'v': normalized blood volume 'q': normalized deoxyhemoglobin content 'y': BOLD signal (% change)

2.1.5 Module hbnm.model.utils

`hbnm.model.utils.`**`clean_builtins`**(*my_dict*)

Cleans the dictionaries

`hbnm.model.utils.`**`cov_to_corr`**(*cov, full_matrix=True*)

Generate correlation matrix from covariance matrix. full_matrix: set True if cov is the full 2n x 2n covariance matrix and you wish to return the covariance matrix of the upper left quadrant (i.e. the E-E block)

`hbnm.model.utils.`**`fisher_z`**`(`*r*`)`
: Perform Fisher z-tranform to the correlation matrix

`hbnm.model.utils.`**`linearize_map`**`(`*x*`)`
: linearization function for T1w/T2w maps

`hbnm.model.utils.`**`load_model_params`**`()`
: Returns the model's synaptic parameters defined in synaptic.py as a python dictionary.

`hbnm.model.utils.`**`normalize_sc`**`(`*x*`)`
: Normalizes the Structural Connectivity and removes the diagonal terms

`hbnm.model.utils.`**`prefix_keys`**`(`*my_dict*, *prefix*, *sep='_'*`)`
: Adds prefix to each key

`hbnm.model.utils.`**`subdiag`**`(`*x*`)`
: Returns the upper diagonal of the matrix

`hbnm.model.utils.`**`vcorrcoef`**`(`*X*, *y*`)`
: Calculates the Pearson correlation coefficient between the columns of the matrix X, and the array y. Here, X is expected to be the upper diagonal terms of the empirical FC matrices of each subject, and y is expected to be the uppoer diaoonal terms of the model FC matrix.

PYTHON MODULE INDEX

h

hbnm.model.utils, 18

B

Balloon (class in hbnm.model.hemo), 17
Bnm (class in hbnm.bnm), 9
BOLD_corr() (hbnm.model.sim.Sim method), 18
BOLD_cov() (hbnm.model.sim.Sim method), 18
BOLD_tf() (hbnm.model.hemo.Balloon method), 17

C

check_stability() (hbnm.bnm.Bnm method), 9
clean_builtins() (in module hbnm.model.utils), 18
corr (hbnm.model.dmf.Model attribute), 13
corr_bold (hbnm.model.dmf.Model attribute), 13
cov (hbnm.model.dmf.Model attribute), 14
cov_bold (hbnm.model.dmf.Model attribute), 14
cov_to_corr() (in module hbnm.model.utils), 18
csd() (hbnm.model.dmf.Model method), 14
csd_bold() (hbnm.model.dmf.Model method), 14

D

Data (class in hbnm.io), 11
disconnect() (hbnm.bnm.Bnm method), 9
distance_function() (hbnm.pmc.Pmc method), 10

E

evals (hbnm.model.dmf.Model attribute), 14
evecs (hbnm.model.dmf.Model attribute), 14

F

fisher_z() (in module hbnm.model.utils), 18

G

G (hbnm.model.dmf.Model attribute), 13
generate_data() (hbnm.pmc.Pmc method), 10
get() (hbnm.bnm.Bnm method), 9
get_appendices() (hbnm.pmc.Pmc method), 10

H

hbnm.model.utils (module), 18
Hemo_cov() (hbnm.model.sim.Sim method), 18
hmap (hbnm.model.dmf.Model attribute), 14

I

I_ext (hbnm.model.dmf.Model attribute), 13
initialize() (hbnm.pmc.Pmc method), 10
integrate() (hbnm.model.dmf.Model method), 14

J

J_NMDA (hbnm.model.dmf.Model attribute), 13
jacobian (hbnm.model.dmf.Model attribute), 15

L

linear_step() (hbnm.model.hemo.Balloon method), 17
linearize_BOLD() (hbnm.model.hemo.Balloon method), 17
linearize_map() (in module hbnm.model.utils), 19
load() (hbnm.io.Data method), 11
load_cifti() (hbnm.io.Data method), 12
load_model_params() (in module hbnm.model.utils), 19
load_nifti() (hbnm.io.Data method), 12
load_sim() (hbnm.model.sim.Sim method), 18

M

Model (class in hbnm.model.dmf), 13
moments_method() (hbnm.bnm.Bnm method), 9
moments_method() (hbnm.model.dmf.Model method), 15

N

nc (hbnm.model.dmf.Model attribute), 16
nonlinear_step() (hbnm.model.hemo.Balloon method), 17
normalize_sc() (in module hbnm.model.utils), 19

P

Pmc (class in hbnm.pmc), 10
prefix_keys() (in module hbnm.model.utils), 19
psd_bold() (hbnm.bnm.Bnm method), 9
psd_syn() (hbnm.bnm.Bnm method), 10

Q

Q (hbnm.model.dmf.Model attribute), 13

R

reset_state() (hbnm.model.hemo.Balloon method), 18

run() (hbnm.pmc.Pmc method), 11
run_particle() (hbnm.pmc.Pmc method), 11

S

S_corr() (hbnm.model.sim.Sim method), 18
S_cov() (hbnm.model.sim.Sim method), 18
save() (hbnm.io.Data method), 12
save_cifti() (hbnm.io.Data method), 12
save_nifti() (hbnm.io.Data method), 13
SC (hbnm.model.dmf.Model attribute), 13
set() (hbnm.bnm.Bnm method), 10
set_jacobian() (hbnm.model.dmf.Model method), 16
set_prior() (hbnm.pmc.Pmc method), 11
sigma (hbnm.model.dmf.Model attribute), 16
Sim (class in hbnm.model.sim), 18
state (hbnm.model.dmf.Model attribute), 16
state (hbnm.model.hemo.Balloon attribute), 18
steady_state (hbnm.model.dmf.Model attribute), 16
subdiag() (in module hbnm.model.utils), 19

T

time_series() (hbnm.model.sim.Sim method), 18

V

var (hbnm.model.dmf.Model attribute), 16
var_bold (hbnm.model.dmf.Model attribute), 16
vcorrcoef() (in module hbnm.model.utils), 19

W

w_EE (hbnm.model.dmf.Model attribute), 16
w_EI (hbnm.model.dmf.Model attribute), 17
w_IE (hbnm.model.dmf.Model attribute), 17
wrap() (hbnm.pmc.Pmc method), 11