

## Integrating neuroinformatics tools in TheVirtualBrain

M. Marmaduke Woodman<sup>1,\*</sup>, Laurent Pezard<sup>1,\*</sup>, Lia Domide<sup>3</sup>, Stuart Knock<sup>1</sup>, Paula Sanz Leon<sup>1</sup>, Jochen Mersmann<sup>2</sup>, Anthony R. McIntosh<sup>4</sup> and Viktor Jirsa<sup>1\*</sup>

<sup>1</sup> Institut de Neurosciences des Systèmes, 27, Bd. Jean Moulin, 13005, Marseille, France.

<sup>3</sup> Codemart, 13, Petofi Sandor, 400610, Cluj-Napoca, Romania.

<sup>2</sup> CodeBox GmbH, Hugo Eckener Str. 7, 70184 Stuttgart, Germany.

<sup>4</sup> Rotman Research Institute at Baycrest, Toronto, M6A 2E1, Ontario, Canada

### ABSTRACT

TheVirtualBrain (TVB) is a neuroinformatics Python package representing the convergence of lines of work in clinical, systems, theoretical neuroscience in the integration, analysis, visualization and modeling of neural dynamics of the human brain as well as the imaging modalities through which these dynamics are measured. Specifically, TVB is composed of a flexible simulator for both neural dynamics and modalities such as MEG and fMRI, common analysis techniques such as wavelet decomposition and multiscale sample entropy, interactive visualizers for replaying cortical timeseries on the 3D surface or editing large-scale connectivity matrices, and an (optional) user interface accessible through modern web browsers. Tying together these pieces with persistent data storage, based on a combination of SQL & HDF5, is a rich, open-ended system of datatypes modeling (systems level) neuroscientific data and the relations among them. This data modeling system in parallel with the so-called adapter pattern architecture permit the integration of TVB with any other computational system, including MATLAB for which support is already available. Notably, TVB provides infrastructure for multiple projects and multiple users, possibly participating under multiple roles: a clinician may import diffusion spectrum imaging data, launch a tractography algorithm, and identify potential lesion points, and then share this data with a computational expert who would then enter to contribute simulation parameter sweeps and analyses, to test which lesion point is most probably given certain empirical imaging data, et cetera; this is one of many multi-user use cases supported by TVB. TVB also drives research forward on many levels: the simulator itself represents the systematization of several recent ad-hoc simulations in the modeling literature on human rest state. In these ways, TVB serves as an integrating platform for disparate expertises in the high level analysis and modeling of the human brain. This paper will begin with a brief outline of the history and motivation for TVB as a unified project *per se*. We proceed to describe the framework and simulator, giving usage examples in the web UI and in plain Python scripting. Finally, we compare TVB with the nearest neighbors in brain modeling, simulation performance, recent advances thereupon with native code compilation and GPUs, and the role of Python and its rich scientific ecosystem in TVB.

**Keywords:** connectivity, connectome, neural mass, neural field, time delays, full-brain network model, Python, virtual brain, large-scale simulation, web platform, GPUs

### 1 MOTIVATION

While whole-brain level simulators have been developed and published for several years now, making the final step of connecting these simulations to empirical results has remained a challenge due to several factors:

1. Source code is typically not distributed, effectively closing the behavior, black box, etc.
2. The forward solutions required to obtain simulated M/EEG & fMRI data are non trivial, requiring interaction with several pieces of software
3. Published simulation methods for stochastic, delayed systems on surfaces are almost non existent (XPPAUT is a notable exception). Efficient handling of  $N^2$  delays requires custom routines.
4. Managing all of the different computational pieces is typically challenging for those who work with empirical data.

To address these concerns, a flexible architecture was developed to allow easy integration of any computational tools along with a system for describing typically types of data. A web based UI was developed for users not comfortable with programming, as well as MATLAB toolbox for interacting with the Python based framework, given that many neuroscientists are already comfortable with the MATLAB workflow. Lastly, a high performance, highly documented simulator along with various forward solutions have been implemented and released under a GPL licence to ensure universal access to high quality simulations, developed on the well-known Github, making it extremely easy for anyone to contribute.

Large scale simulation implies flexible integration. We shall see how this is enable by the architecture.. <sup>mw:</sup> [\[expand\]](#)

In effect, we wish for a theoretician and clinician to be able to collaborate; to enable such a possibility, we require a platform to enable such opportunity. <sup>mw:</sup> [\[expand\]](#)

#### 1.1 Why another simulator?

A significant part of *TheVirtualBrain* is simulating brain-scale neural networks. While several existing simulators could have been adapted, we have estimated that *TheVirtualBrain* style simulations are far enough outside the design of other simulators to make starting from scratch a better idea.

Many neural network simulators have been developed and published, focusing first on abstract rate neurons (in the style of PDP),

\*to whom correspondence should be addressed:  
marmaduke.woodman@univ-amu.fr, viktor.jirsa@univ-amu.fr

modelling neurocognitive processes, on one hand, and on the other, full multicompartmental neuron simulators treating complex spatial geometries, e.g. NEURON. More recently, due to interest in the computational properties of spiking neurons and their relevance to experimental observations, simulators targeting specifically spiking neurons have been prominent, e.g. Brian.

However, another level description of neural dynamics has been treated in the literature of neural mass models and neural fields. Here, the spatial extent of the modeled dynamics is far larger and hence permits networks thereof to scale reasonably to the entire cortex, under the assumptions of the models, when combined with empirical measurements of cortico-cortical connectivity. Therefore, the physical scale modeled by the *TheVirtualBrain* simulators differs from that for which other simulators were designed. Several technical issues stem from this scale, e.g. efficient handling of dense  $N^2$  delays and neural field-like connectivity, which will be discussed in more detail below.

## 1.2 Why Python?

In fact, the core simulator began in MATLAB, however, as the needs expanded, the architecture, detailed in the next section, quickly outgrew the matrix-struct-function triumvirate that is conventional in MATLAB programming. While modern MATLAB permits advanced object-oriented programming, it has the disadvantages of being relatively unused, and largely unsupported by MATLAB's own IDE, the MATLAB Compiler, and the free alternative Octave, lastly it provides no support for metaclasses or data descriptors, which were extensively used in *TheVirtualBrain*'s architecture.

The architecture of *TheVirtualBrain* had been prototyped in Python, and in turn, both the language and the scientific ecosystem were more than rich enough to support continued development entirely within Python, of both the architecture and the simulator, in addition to it being a general purpose language.

## 2 ARCHITECTURE

*TheVirtualBrain* is logically and technically divided at deploy time into a scientific library and a framework, where the scientific library includes datatypes, common analyses and the simulator as its central piece, while the framework handles execution infrastructure, the web-based user interface and database connectivity. While TVB Scientific Library can function independently, as a Python module, TVB Framework needs the scientific library to wrap around it at runtime.

*TheVirtualBrain* source code is available for download on Github at <https://github.com/the-virtual-brain/>. Previous Git and Python knowledge is required for contributing. Although you could independently install Python and the rest of TVB dependencies on your machine, and then use the Github code as a local clone, we recommend you to download TVB Distribution from <http://www.thevirtualbrain.org/register/>, fork our repositories on Github and further use contributor-setup script from TVB-Distribution to link the two. In this recommended use-case, you will have all TVB Dependencies already installed and under control.

### 2.1 TVB Scientific Library

#### 2.1.1 General description

#### 2.1.2 Datatypes

*Description and examples* <sup>lp</sup>: [From FIN paper:] TVB-datatypes are annotated data structures which contain one or more data attributes and associated descriptive information, as well as methods for operating on the data they contain. The definition of a datatype is achieved using TVB's traiting system, which was inspired by the traiting system developed by Enthought?

<sup>lp</sup>: [The TVB-trait system might be worth described here?]

<sup>lp</sup>: [Table 2 "TVB datatypes" of FIN article might be interesting here?]

In scientific Python code, it is conventional to provide arguments of an algorithm as a "bare" array or collection thereof, and sanity checks of arguments proceed on the basis of array geometry, for example. In *TheVirtualBrain*, we consider a *datatype* to be a full, formal description of an entity involved in an algorithm that would be part of *TheVirtualBrain*. For example, the Connectivity datatype, which may elsewhere be represented by a simple  $N$  by  $N$  NumPy array, is written as a class in which one of the attributes, *weights* is a explicitly typed `FloatArray`, and the declaration of this type is complemented by explicit label, default values, and documentation strings.

<sup>mw</sup>: [Add a listing showing this].

Because an explicit goal of *TheVirtualBrain* was to provide a user interface to each of the datatypes and algorithms contained within, it is necessary at some point to provide metadata. A `traits` system was developed, similar to that of IPython or EPD, was developed, allowing for attributes of a datatype class to be written out with full metadata. An extensive set of existing building blocks are already provided from numeric types and arrays to lists, tuples, string, and dictionaries.

<sup>mw</sup>: [Show description of a data type]

When methods of such a class are invoked, they may use the traitled attributes directly, accessing either a default value or one given during the instantiation of the object. Additionally, this allows the web-based UI to introspect a class for all of its attributes and their descriptions, to provide help in the interface. The explicit typing also allows such classes to be nearly automatically mapped to SQLAlchemy & HDF5 tables, the combination of which provides persistence for datatypes when using the web UI. Lastly, because such metadata is used to build the docstring of a class, the IPython user also may obtain extensive descriptions of attributes and function arguments in the usual way.

<sup>lp</sup>: [The Connectivity datatype might be an example]

### 2.2 TVB Framework

**2.2.1 General description** The framework has been developed with generality and modularity in mind. It provides a database backend, workflow management and a number of features to support collaborative work. The central idea is data-oriented in a sense that data are fed and stored into the system and can be transformed into another type of data (including visualization) through the operation provided by an external library (including TVB scientific library, but not restricted to it) that have been *adapted* to the framework. As a consequence, the two central concepts are *datatypes* i.e. types of data that can be handled in the framework and *adapters* i.e. processes that allow to interface/adapt external libraries to the datatypes handled by within the framework.

The framework also provides user interfaces: web-based graphical interface and console interface for advanced user and developers.

Due to the generality of the framework, it relies on the Python *abstract classes* mechanism. <sup>lp</sup>: [From Python glossary:] Abstract base

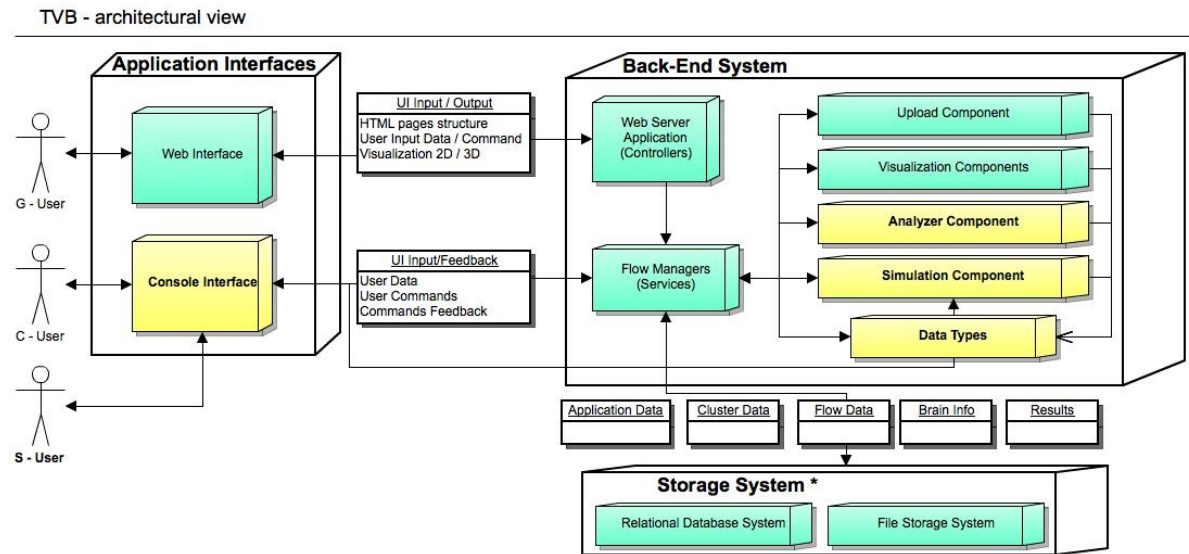


Figure 1: *TheVirtualBrain* architecture: Yellow blocks are part of the Scientific Library of TVB, while the green blocks are part of TVB Framework. TVB provides two independent interfaces, depending on the interaction type wanted by the end-user (web or console). TVB Storage layer is compulsory for the web interface, but it can be switched on/off for the console interface.

classes complement duck-typing by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong.

**2.2.2 Data Storage** Data storage is managed using *SQLAlchemy* <http://www.sqlalchemy.org/> which provides a common interface for the two proposed storage system: SQLite and PostgreSQL <sup>lp</sup>: [or flat file system?].

### 2.2.3 Adapters

**General description** While datatypes provide a way of description what algorithms work with, sufficing for the typical user looking to write scripts against the available libraries, the web-based UI requires algorithms to adhere to a generic interface, which is elsewhere referred to as the Adapter pattern. Typically, this implies that a class is written that is able to describe the collection to datatypes required and a single method to invoke the algorithm.

Adapters are derived from the abstract class named *ABCAdapter* which defines the common interface for the adapters with methods...

Several categories of adapters have been defined:

- *analyzers* which allow the interface between libraries containing algorithm for the analysis of the data (wavelets, FastICA, etc.) and the TVB framework and datatypes.
- *creators*
- *exporters*
- *portlets*
- *simulator*
- *uploaders*: allow the upload of external data into TVB framework such as gifti files of plain csv files.

- *visualizers* <sup>lp</sup>: [derived from the *ABCDisplayer* abstract class] allow to display the correct datatype.

<sup>mw</sup>: [Show the adapter for FastICA]

Note that the adapters and datatypes are intended to provide full power and flexibility of the framework; when the simulator is invoked from the web-based UI, it is done so through an *SimulatorAdapter* which, despite being relatively complex, is datatypes all the way down.

It is reasonable to ask to what such a scheme offers over the more conventional approach of Python, where presumably it would have been sufficient that each adapter consist of a class with an `__init__` and `__call__` method, in the case of a function type. We note that because in the case of *TheVirtualBrain*, the context in which an object is used is more varied, e.g. not simply initialized but loaded through SQLAlchemy's ORM, and that the adapter is required to perform more tasks than just initialization and invocation, e.g. provide expected shape of result, it was advantageous to create a distinct set of interfaces built on the abstract base class framework provided by Python's standard library.

#### Adapting *sklearn*'s *FastICA*

**Interfacing with MATLAB** One of the well-known libraries for characterizing anatomical and functional connectivity is the Brain Connectivity Toolbox. Because it is written in MATLAB, with maintainers who prefer MATLAB, we chose not to port routines of the library to Python but instead use a MATLAB adapter which runs arbitrary MATLAB code. This generic adapter works by generating a wrapper script for the MATLAB code, which wraps the code in a try-except clause, and loads and saves the workspace before and after execution, generating a workspace `.mat` file, invoking the MATLAB or Octave executable, and loading the resulting workspace file. Despite invocation of MATLAB being a relatively

slow operation, this works fine in a single user situation, and where Octave is available, it is quite fast. In the case that many operations are necessary, they can be batched into the same run.

### 3 USER INTERACTION

#### 3.1 Graphical Interface Interaction

A graphical web interface was chosen as option of TVB face for quick interaction. The web interface is easy to access (local or remotely), it can be used by different types of users, including the ones without programming knowledge, and it offers great support while learning about TVB concepts and workflow expectations. In our modeling diagrams, we called the actor accessing *TheVirtualBrain* through the web interface a *G-User*.

The http is served using *Cherrypy* <http://www.cherrypy.org/> which is a minimalist, object-oriented web framework, in combination with Genshi templating system, to support the separation as guided by MVC (Model View Controller) pattern.

**3.1.1 Projects, Accounts, Operations & Data** *TheVirtualBrain* uses entities like: Account, Project, Operation, DataType and Workflow for modeling G-User actions and artifacts.

A TVB Account or User is needed for accessing TVB through the web interface. When TVB web interface is fired for the first time, the G-User is requested to set the username and password for the first account. Later on, people can register for other accounts, but for using these accounts they will need first to get validated by the first account (which acts under "admin" role).

A TVB Project is a logical grouping entity that can be used in several ways by the end-user; for example one could choose to create a project for each experiment in TVB, while others might create projects for each subject they simulate. Each Project has a single User (or Account) as owner, but can be shared with multiple other users.

Any execution of an Adapter results into an Operation in the context of a project. Multiple Operations will be executed under the same project. For example we will have operations created for each execution of a simulation, each run of a Fourier analyzer, or launch of a Brain Visualizer. An operation changes status over time, from started into: canceled, finished successfully or finished with error. One operation could have multiple input and output parameters. Parameters can be scalars or DataTypes.

A Workflow in TVB is a set of Operations with they artifacts, and wraps around a simulation as leading operation. A workflow can be seen as a default Tag placed by the system on Operations and DataTypes which are logically connected, as resulting one after the other. Custom tags can also be added by the end-user both on DataTypes and operations, for tracking data inside a TVB Project.

#### 3.1.2 Simulator Interface

**3.1.3 Analysis & Visualizers** *TheVirtualBrain* does not aim to compete at the analysis level with other tools in Neuroscience, highly specialized and with great history in data analysis, like FSL or SPM. What we offer is a minimalist set of algorithms to post-process your simulated results (or even process imported patient measured scans) inside TVB, mainly for quick validations.

We have created inside *TheVirtualBrain* adapters for Fast ICA from the python library *sklearn*, we've implemented a python version of Fourier Spectral Analysis, we have even wrapped the Matlab

library BCT <https://sites.google.com/site/bctnet/>, and others as analyzers.

For each of the DataTypes produced in TVB, one or multiple visualizers are available. *TheVirtualBrain* has couple of visualizer types, each developed with the technology providing better support on the specific requirements for the visualization in course:

1. WebGL viewers: based on HTML 5 Canvas element and the gl context. These viewers offer 3D nice display, vectorial zoom support, user interaction with the scene (rotate, translate), quick response (even when thousands of vertices and edges are to be manipulated) and good resolution for the images exported.
2. SVG viewers: offer great selection, zoom and scaling effects and extraordinary quality for the exported artifacts, while having a relatively low number of elements to display on the page. We use such viewers for manipulating and displaying TimeSeries, Covariance or Cross Coherence DataType results.
3. MPLH5 viewers: Matplotlib has an HTML 5 backend that we use for viewing some TVB DataTypes (like Fourier or Wavelet) <https://code.google.com/p/mplh5canvas/>
4. Other simpler viewers are using JIT <http://philogb.github.io/jit/> or FLOT <http://www.flotcharts.org/> JS libraries. These are mainly 2D graph displays for some TVB generated data.

**3.1.4 Connectivity Tool** Connectivity in the context of *TheVirtualBrain* is a DataType, mapping structural information about a subject (real patient or theoretical model). For editing and viewing a Connectivity, *TheVirtualBrain* has a specific page, where the G-User could manipulate connectivity strength and lengths starting from the granularity of an edge. We do not store or use information about the exact anatomical path or a connection, only the region centers and connection weights and lengths.

#### 3.2 Scripting Interaction

*hello\_brain.py* To give a basic feel for scripting *TheVirtualBrain* simulations, we will walk through a simple example of a region-level simulation. We start with

```
from tvb.simulator.lab import *
```

which is an all-in-one module making writing scripts shorter, in the style of pylab, as it imports everything from pylab, numpy and most of *TheVirtualBrain*'s simulator modules. Next, we build a simulator object:

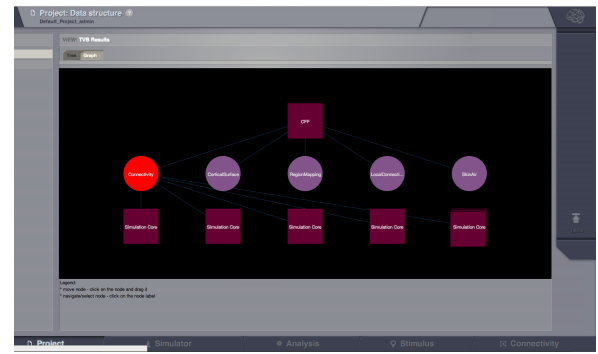
```
sim = simulator.Simulator(
    model = models.Generic2dOscillator(),
    connectivity = connectivity.Connectivity(),
    coupling = coupling.Linear(a=1e-2),
    integrator = integrators.HeunDeterministic(),
    monitors = (
        monitors.TemporalAverage(),
    )
)
```

where we've employed a two dimensional oscillator with default parameters, the default connectivity, a linear coupling function with a slope of  $1e-2$ , and deterministic Heun integrator and a monitor that temporally averages the network dynamics before providing output.

While *TheVirtualBrain* strives to keep modules independent of one another, it is typical for mathematical dependencies to arise between, for example, the mass model and the integration time step, so after configuring a simulator object, it is necessary to invoke

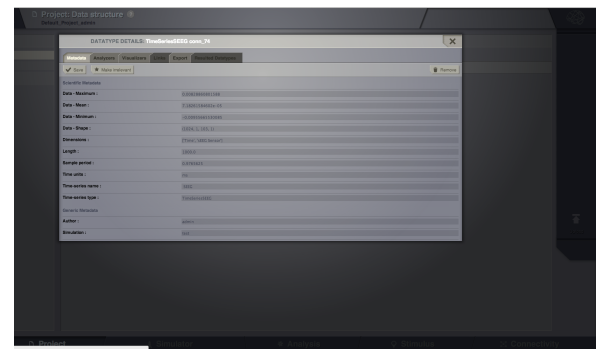


(A)



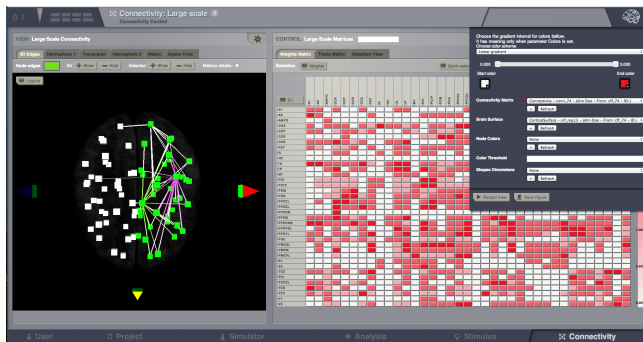
(B)

(C)

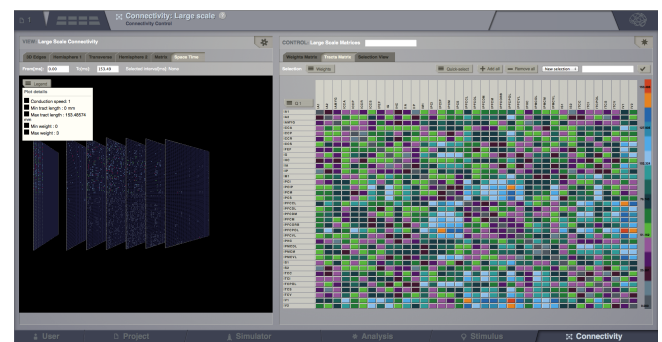


(D)

Figure 2: *TheVirtualBrain* Data Organization: (A) View All Projects. (B) 2D graph display of operations with their input and output DataTypes. (C) View Operations in current project, with their status, duration, results, etc. (D) DataType details and further available operations for it. This menu becomes available after clicking a DataType result from several places in *TheVirtualBrain*



(A)



(B)

Figure 3: Connectivity Tools: (A) Left side: Displaying weighted connections between selection of nodes, with 3D manipulation. Right side: Editing weight connections, singular or bulk. (B) Left side: Show effect of connectivity delays when conduction speed is 1. Right side: Editing and displaying one quadrant from the matrix of connection tracts.

```
sim.configure()
```

which results in walking the tree of objects, checking and configuring the constraints among parameters recursively.

The next step is to run through the simulation, collecting output from the simulator. In this case, it is as simple as

```
ys = array([y for ((t, y),)
```

```
in sim(simulation_length=3e2)])
```

where the simulator has been called, returning a generator which performs the integration and returns, for each monitor, the current time and activity. In a case where EEG and fMRI monitors, for example, were used, we might write

```
eeg, mri = [], []
```



```
for (t_eeg, y_eeg), (t_mri, y_mri) in sim(3e2):
    if y_eeg is not None:
        eeg.append(y_eeg)
    ...
```

Because fMRI and EEG monitors have very different timescales, whenever one monitor return data and the others do not, the others contain None, hence the check. Building more complex logic in this loop would permit, for example, online feedback and modification of connectivity.

After the simulation loop has finished, you may wish to see the result, following the previous listing,

```
plot(ys[:, 0, :, 0], 'k', alpha=0.1)
```

Here we note that `ys` is four dimensional. The simulator has the convention of treating mass model state as a three dimensional array of state variables by nodes by statistical modes. Because `ys` is an array collected over time, the first dimension is time, and the plot here is of each node's first state variable, over time.

Many more demonstrations of the various features of the simulator can be found in scripts distributed with the sources of *TheVirtualBrain*, or browsed online at [https://github.com/the-virtual-brain/scientific\\_library/tree/trunk/tvb/simulator/demos](https://github.com/the-virtual-brain/scientific_library/tree/trunk/tvb/simulator/demos). In the next section, we will go into detail about the different components of the simulator.

## 4 SIMULATOR

The *TheVirtualBrain* simulator resembles popular neural network simulators in many fundamental ways, both mathematically and in terms of informatics structures, however we have found it necessary to introduce auxiliary concepts particularly useful in the modeling of large scale brain networks.

### 4.1 Node dynamics

In our models, nodes are not abstract neurons nor necessarily small groups thereof, but rather large populations of neurons, considered by, for example, the work of Wilson and Cowan.

<sup>mw</sup>: [\[quick list of models & their relevance\]](#)

### 4.2 Network structure

The nodes are embedded in two scale network structure. The first of which is derived from empirical measurements of the myelinated corticocortical connectivity, which we shall refer to as the inhomogeneous, large scale structure. An implication of this structure is the inherent delays in communication due to finite conduction velocity.

The second form of network structure is prescribed in the case of a cortical surface by the combination of said surface and a connectivity kernel. Together, these generate a homogeneous, local connectivity. For the current work, we consider this coupling to be instantaneous.

<sup>mw</sup>: [\[Need to explain regional & surface simulations\]](#)

### 4.3 Integration of stochastic delay differential equations

Rather unlike other simulation paradigms, both noise and delays are common features of simulations in *TheVirtualBrain*. As such, the simulator is equipped with Heun and Euler methods for stochastic

integration and the pairwise inter-regional delays are treated in an efficient fashion.

<sup>mw</sup>: [\[the general equation here\]](#)

<sup>mw</sup>: [\[Describe handling of delays?\]](#)

## 4.4 Forward solutions

One of the primary goals of the *TheVirtualBrain* simulator is to allow for the simulation of empirical whole-brain data, such as EEG or fMRI. *TheVirtualBrain* implements several forward solutions as so-called monitors, including MEG, sEEG, EEG and fMRI.

Crucially, given the amount of data *TheVirtualBrain* may produce, especially for simulations on a cortical surface, each of the monitors is "on-line" in the sense that it runs in constant space.

## 4.5 Native code generation for C & GPU

Several of the core components (integrators, mass models, coupling functions) have targeted towards a C source code backend, which has allowed for the compilation of simulations to native code loaded either as a shared library accessed via the `ctypes` modules or as CUDA kernels accessed via the `pycuda` module. While such an approach may provide speed ups, they depend on the presence of a C compiler and, in the case of GPU, the CUDA toolkit and a compatible graphics card.

<sup>mw</sup>: [\[insert figures from CNS here\]](#)

## 4.6 Other simulators compared to *TheVirtualBrain*

Brian should be a particular focus in this section, as it may be one of the closest.

## 5 FUTURE WORK

Since the recent release of the 1.0 version of *TheVirtualBrain*, it has been officially considered *feature* complete, however, in several cases, the development of features has outstripped other essential parts of software projects. Going forward, general priorities include advancing test coverage and improving documentation for users. In the mean time, *TheVirtualBrain*'s Google groups mailing list continues to fill any gaps.

In the simulator itself, continued optimization of C and GPU code generation will take place to increase the rate at which parameter sweeps can be performed. Additionally, an interface *from* MATLAB to *TheVirtualBrain* is being developed to allow use of the simulator through a simple set of MATLAB functions. As this infrastructure is based on an HTTP and JSON API, it will likely enable other applications to work with *TheVirtualBrain* as well.

Lastly, as *TheVirtualBrain* was originally motivated to allow a user to move from acquired data to simulated data as easily as possible, we will continue to integrate the requisite steps

1. Diffusion tensor imaging & tractography pipeline
2. Connectome project
3. Structural imaging processing via FreeSurfer (pySurfer, NiPy, etc.)
4. NeuroML project