

## 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, Aix-Marseille Université, 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 clinical, systems, and theoretical neuroscience in the analysis, visualization and modeling of neural and neuroimaging dynamics. TVB is composed of a flexible simulator for neural dynamics measured across scales from local populations to large-scale dynamics measured by electroencephalography (EEG), magnetoencephalography (MEG) and functional magnetic resonance imaging (fMRI), and core analytic and visualization functions, all accessible through a web browser user interface. A datatype system modeling neuroscientific data ties together these pieces with persistent data storage, based on a combination of SQL & HDF5. These datatypes combine with adapters allowing TVB to integrate other algorithms or computational systems. TVB provides infrastructure for multiple projects and multiple users, possibly participating under multiple roles. For example, a clinician might import patient data to identify several potential lesion points in the patient's connectome. A modeler, working on the same project, tests these points for viability through whole brain simulation, based on the patient's connectome, and subsequent analysis of dynamical features. TVB also drives research forward: the simulator itself represents the culmination of several simulation frameworks in the modeling literature on human resting state. The availability of the numerical methods, set of neural mass models and forward solutions allows for the construction of a wide range of brain-scale simulation scenarios. This paper briefly outlines the history and motivation for TVB, describing the framework and simulator, giving usage examples in the web UI and Python scripting. **Keywords:** large-scale brain network, simulation, web platform, Python, connectivity, connectome, neural mass, time delays, functional MRI, electroencephalography, magnetoencephalography

### 1 INTRODUCTION

Neuroscience has evolved through extensive interactions among disciplines to advance our appreciation of the relation between brain and behavior. The interdisciplinary nature of the field presents formidable challenges for effective collaboration.

These challenges call for two kinds of solutions. First, there is a need for comprehensive, modern computational libraries written in

a widely used and available programming languages; current examples include MNE-Python (Gramfort et al., 2013), a Python package for treating M/EEG data via time-frequency analyses and inverse solutions and the Brain Connectivity Toolbox (Rubinov and Sporns, 2010a) for analyzing the graph theoretic properties of structural and functional connectivity. Second, there is a need for the implementation of collaborative infrastructure for sharing not only data, but expertise; CARMEN (Austin et al., 2011) and G-Node (Herz et al., 2008) are two such examples of developing platforms for collaborative work and data sharing, in the domains of cellular and systems neurophysiology.

TheVirtualBrain (TVB) provides new tools to facilitate the collaboration between experimentalists and modelers by exposing both a comprehensive simulator for brain dynamics and an integrative framework for the management, analysis simulation of structural and functional data in an accessible, web-based interface. The choice of Python was made based on its wide use as the high-level language in scientific programming, the unparalleled open-source libraries and tools available, and strong software engineering culture. This choice was confirmed by the publication of the first issue of Python in Neuroscience and has made it possible for the entirety of TVB from the numerical algorithms to the web server to be written in Python.

#### 1.1 The framework

The TVB framework was developed to allow easy integration of any computational tools along with a system for describing typical types of data. Additionally, two main constraints for the framework were to provide a web interface to allow remote collaboration and the exchange of data (or simulation results) between users, especially for users not comfortable programming. The web interface and database backend are built on combination of the CherryPy and SQLAlchemy packages.<sup>1</sup> The overall structure of TVB is depicted in Figure 1.

#### 1.2 The simulator

A significant part of TVB is simulating large-scale brain networks. While several existing simulators could have been adapted, we have estimated that TVB style simulations are far enough outside the design of other simulators to make a new development necessary. We discuss these reasons in the following.

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

<sup>1</sup> The full list of dependencies can be found in the project's documentation.

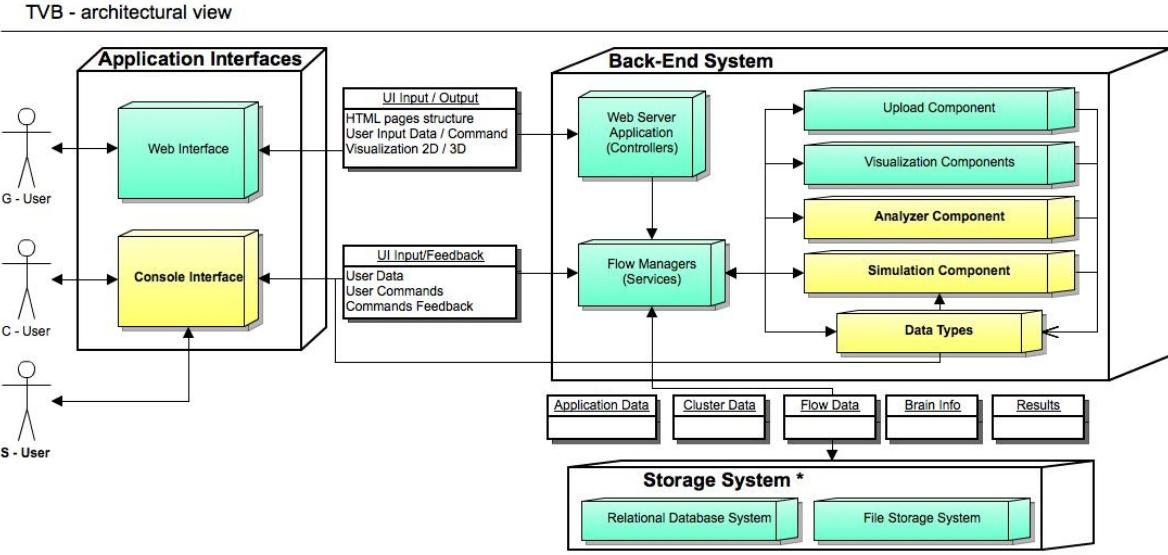


Figure 1: TVB architecture: Yellow blocks are part of the scientific library of TVB, while the green blocks are part of TVB Framework. TVB provides two independent interfaces, depending on the interaction type desired the end-user (web or console). TVB Storage layer is compulsory for the web interface, but it can be switched on/off for the console interface. The terms *G-User*, *C-User* and *S-User* refer to users of the graphical, console & database and scripting components. Here, console interface is conceptual and refers only to existing Python command lines, IPython included.

Existing neural network simulators typically focus either on abstract rate neurons, modelling neurocognitive processes, or full multicompartmental neurons treating complex spatial geometries, e.g. NEURON (Hines and Carnevale, 2001), modelling the interaction of channel distributions in dendrites. More recently, due to interest in the computational properties of spiking neurons and their relevance to experimental observations, simulators designed for spiking or oscillating neurons have become prominent, including Brian (Goodman and Brette, 2009), which we initially considered for our simulations. In TVB the network is defined with neural mass or field models (Deco et al., 2008; Coombes, 2010) rather than cellular models. The spatial extent of the modeled dynamics is macroscopic and scales reasonably to the entire cortex, and uses empirical measurements of corticocortical connectivity. Several technical issues are unique to this scale, such as efficient handling of dense  $N^2$  inter-regional delays and integration of neural field-like models and connectivity on triangular meshes in 3D. Finally, comparison with experimental data requires forward solutions that transform physiological signals to the commonly used imaging modalities such as EEG, MEG and fMRI need to be employed. For these reasons, TVB required a new simulator, built around the paradigm of whole-brain scale simulation.

### 1.3 Obtaining and contributing to TVB

The easiest way to get started with TVB is to download a distribution from <http://www.thevirtualbrain.org>, which is available for Windows, Mac OS X and Linux. This distribution includes all of pieces, from the simulator to the web interface, and has no requirements other than a modern web browser supporting WebGL, SVG and HTML5.

Alternatively, because TVB is licensed under the GPL, the sources may be readily obtained from the public Git repositories hosted on Github (Dabbish et al., 2012), at <https://github.com/the-virtual-brain>. In order to use the simulator, only the standard scientific Python packages are required (NumPy & SciPy). The framework and web interface depend on a few more packages.

We provide an API doc, built from the docs strings using Sphinx. User's, Contributor's and Developer's manuals are also provided with TVB distributions in PDF format. The source files (in .rst) are available through the Github repository <https://github.com/the-virtual-brain/docs>. In addition, IPython notebooks (Pérez and Granger, 2007) for interactive tutorials are provided. These are based on the demonstration scripts provided with the scientific library, and include a more detailed description of the scientific goal (if applicable), the components and stages of a simulation as well as a brief description in the case of reproducing previous work. Users interacting with *TheVirtualBrain* GUI may also benefit from these tutorials ([https://github.com/the-virtual-brain/scientific\\_library/wiki/Tutorials](https://github.com/the-virtual-brain/scientific_library/wiki/Tutorials)) when displayed as snippets with IPython nbviewer.

Additionally, the user interface provides an online help overlay, that pulls information from the User's manual.

## 2 ARCHITECTURE

TVB is logically and technically divided into a scientific library and a framework package, where the scientific library includes datatypes, basic analyses and the simulator, while the framework handles execution infrastructure, the web-based user interface and data storage. The scientific library can function as an independent

Python module, but the framework depends the scientific library for datatype definitions and algorithms.

## 2.1 Basic Concepts

The TVB framework is oriented around data and the operations that introduce, generate, transform and visualize the data. The relevant interface classes derive from the metaclasses in Python's abstract base class library, and provide a foundation for defining any type of data from a tuple to a set of EEG channel labels to a simulation, each of which is defined by a class implemented the datatype interface and possessing one or more datatypes as attributes. An operation is the execution of any algorithm that has been *adapted* to the framework in a class implementing the abstract adapter interface, and range from the simulator to data importers to visualizers. The goal of these twin, generic abstractions is to provide a solid basis on which to implement a storage back-end, workflow management and a number of features to support collaborative work.

Importantly, the framework supports both the use of the web-based graphical interface and the console interface for advanced user and developers. Where the console user does not wish to rely on the database persistence, this storage layer can be disabled. Specifically, TVB uses the notion of *profile* to identify in what context the application is currently running, and thus what components are expected to be loaded. For example, when the scientific library is used alone, a specific profile (*library profile*) class gets linked as current profile, which, in this case, disables data storage and the web interface. Other profiles available in TVB are: *command profile*, *deployment profile* (with web interface), and *test profiles*.

## 2.2 Datatypes and storage

**2.2.1 TVB Traits** Because an explicit goal of TVB is to provide a user interface to each of the entities and algorithms contained within, it is necessary at some point to provide metadata on how to built that interface. A traits system was developed, similar to that of IPython or EPD, allowing for attributes on a TVB class to be written out with its full metadata. An extensive set of building blocks implements from numeric types and arrays to lists, tuples, string, and dictionaries.

When methods of such a class with annotated attributes are invoked, they may use the traited attributes directly, accessing either a default value or one given during the instantiation of the object. Additionally, this allows the web-based user interface to introspect a class for all of its attributes and their descriptions, to provide help and choose the proper display form. The explicit typing also allows such classes to be nearly automatically mapped to storage tables, providing persistence, when the storage layer is enabled. Lastly, because such metadata is used to build the docstring of a class, the console user also may obtain extensive descriptions of class, attributes, methods and arguments in the usual way. Table 1 lists the various parts of a traited attribute and how they are used.

**2.2.2 Datatypes** 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 TVB, we consider a *DataType* to be a full, formal description of an entity involved in an algorithm that would be part of TVB.

In TVB, datatypes represent the common language, to be used between different application parts: like uploaders, analyzers, simulator and visualizers. Some of the algorithms are producing these datatypes, while others are reading them as input. In order to decouple the definition and several usages of such entities, datatypes are declared outside the algorithms and shared between them. For example an instance of datatype *TimeSeriesRegion* is created by the Simulator, and it can be accepted as input for several visualizers or analyzed by principal component analysis and cross coherence algorithms.

Technically, TVB datatypes are annotated Python classes, which contain one or more fields 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, mentioned in previous section.

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. See Code 1.

```
class ConnectivityData(MappedType):
    region_labels = arrays.StringArray(
        label="Region_labels",
        doc="""Labels for the regions""")
    weights = arrays.FloatArray(
        label="Connection_strengths",
        doc="""... strength of connections""")
    tract_lengths = arrays.FloatArray(
        label="Tract_lengths",
        doc="""... length of myelinated fibre tracts""")
    speed = arrays.FloatArray(
        label="Conduction_speed",
        default=numpy.array([3.0]),
        file_storage=core.FILE_STORAGE_NONE,
        doc="""... matrix of conduction speeds""")
    centres = arrays.PositionArray(
        label="Region_centres",
        doc="""... locations for the region centers""")
```

Code 1: The ConnectivityData listing

## 2.3 Adapters

The framework expects algorithms to be adapted by providing a class that inherits from the base adapter, *ABCAdapter*, implementing the adapter interface:

```
class AdapterExample(ABCAdapter):
    @abstractmethod
    def get_input_tree(self):
        pass
    def configure(self, **kwargs):
        pass
    @abstractmethod
    def launch(self):
        pass
```

Code 2: Excerpt of the ABCAdapter

where *get\_input\_tree* builds a dictionary of input arguments required for the algorithm and for presenting menus and fields in the user interface, *configure* allows the adapter

| Traited Attribute | Description   |
|-------------------|---|
| default           | Default value for current attribute. Will be set on any new instance if not specified otherwise in the constructor.   |
| console_default   | Define how a default value can be computed for current attribute, when console interface is enabled.  |
| range             | Specify the set of accepted values for current attribute. Mark that this attribute is usable for parameter space exploration.   |
| label             | Short text to be displayed in UI, in front of current attribute.  |
| doc               | Longer description for current attribute. To be displayed in UI as help-text.   |
| required          | Mark current attribute as required for when building a new instance of the parent class.  |
| locked            | When present and <i>True</i> , current attribute will be displayed as read-only in the web interface.   |
| options           | Used for attributes of type <i>Enumerate</i> , specifying the accepted options as a list of strings.  |
| filters_ui        | SQL filters on other attributes, to be applied in UI.   |
| select_multiple   | When <i>True</i> , current attribute will be displayed as a select with multiple options in UI (default is single-select)   |
| order             | Optional number identifying the index at which current attribute will be displayed in UI.   |
| use_storage       | When negative, the attribute is not displayed at all. Ascending order for indices is considered when displaying.  |
| file_storage      | When <i>False</i> , current attribute is not stored in database or file storage.<br>Valid values for this attribute are: <i>None</i> , <i>HDF5</i> , or <i>expandable_HDF5</i> ,<br>When <i>None</i> , current attribute is not stored in the file-storage at all. When <i>HDF5</i> , we use regular H5 file storage.<br>When <i>expandable_HDF5</i> value is set, a H5 stored in chunks is used. |

**Table 1.** TVB currently available Traited Attributes

to initialize itself and its algorithm based on arbitrary arguments and `launch` invokes the algorithm. Additional methods include `get_output`, `get_required_memory_size`, and `get_required_disk_size`.

Several categories of adapters have been defined in TVB:

- *creators* which are internal algorithms for producing datatype instances. Each creator has one or multiple pages in the web interface, in which the user configures input parameters and chooses from the available options for computing a particular datatype.
- *uploaders*: allow the upload into TVB framework of external data, such as *gifti* files of plain *csv* files.
- *simulator* is an adapter for the simulator, adjusting it to fit the workflow mechanisms inside the framework .
- *analyzers* which offer the interface to libraries containing algorithms for the analysis of the data (wavelets, FastICA, BCT, etc).
- *visualizers*, derived from the *ABCDisplayer* base class, prepare a datatype for display. Each visualizer (Python adapter class) requires a complementary set of Javascript and HTML files.
- *portlets* provide a chain of analyzers leading to a visualizer.
- *exporters* prepare a datatype for export & download.

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 a *SimulatorAdapter* which, despite being relatively complex, is built with *traits* all the way down.

It is reasonable to ask 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 TVB, the context in which an object is used is more varied, e.g. not simply initialized but loaded through SQLAlchemy's object relational mapping, and that the adapter is required to perform more tasks than just initialization

and invocation, e.g. provide expected shape of result, estimate occupied memory and do not start if insufficient resources are found on current machine, it was advantageous to create a distinct set of interfaces built on top of the abstract base class framework provided by Python's standard library.

*An adapter for FastICA* An example of an adapter applying the FastICA algorithm to each of the state variables in a simulation time series is given in Code 3.

```
class ICAAdapter(ABCAsynchronous):
    _ui_name = "Independent_Component_Analysis"
    _ui_description = "ICA_for_a_TimeSeries_input(DataType."
    _ui_subsection = "ica"

    def get_input_tree(self):
        algorithm = fastICA()
        algorithm.trait.bound = self.INTERFACE_ATTRIBUTES_ONLY
        tree = algorithm.interface[self.INTERFACE_ATTRIBUTES]
        filt = FilterChain(fields=
            [FilterChain.datatype + '.nr_dimensions'],
            operations=['=='], values=[4])
        for node in tree:
            if node['name'] == 'time_series':
                node['conditions'] = filt
        return tree

    def get_output(self):
        return [IndependentComponents]

    def configure(self, time_series, n_components=None):
        self.input_shape = time_series.read_data_shape()
        log_debug_array(LOG, time_series, "time_series")

        self.algorithm = fastICA()
        self.algorithm.n_components = \
            n_components or self.input_shape[2]

    def get_required_memory_size(self, **kwargs):
        used_shape = (self.input_shape[0], 1,
                      self.input_shape[2], self.input_shape[3])
        input_size = numpy.prod(used_shape) * 8.0
        output_size = self.algorithm.\
            result_size(used_shape)
        return input_size + output_size

    def launch(self, time_series, n_components=None):
        ica_result = IndependentComponents()
```

```

source=time_series,
n_components=int(self.algorithm.n_components),
storage_path=self.storage_path)

# 4-D simulation time series
node_slice = [slice(self.input_shape[0]),
              None, slice(self.input_shape[2]),
              slice(self.input_shape[3])]

ts = TimeSeries(use_storage=False):
for var in range(self.input_shape[1]):
    node_slice[1] = slice(var, var + 1)
    ts.data = time_series.read_data_slice(
        tuple(node_slice))
    self.algorithm.time_series = ts
    partial_ica = self.algorithm.evaluate()
    ica_result.write_data_slice(partial_ica)
    ica_result.close_file()
return ica_result

```

Code 3: ICA adapter for FastICA library

*Interfacing with MATLAB* One of the well-known libraries for characterizing anatomical and functional connectivity is the *Brain Connectivity Toolbox* (Rubinov and Sporns, 2010b). Because it is written in MATLAB, with developers who prefer MATLAB, we chose not to port routines of the library to Python but instead build a MATLAB adapter that runs arbitrary MATLAB code.

This generic Matlab adapter works by generating at runtime a script with MATLAB code, wrapping the script call in Python with a try-except clause, loading and saving the workspace before and after the call, 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 without problems 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 SIMULATOR

The simulator in TVB 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. In the following, we will highlight some of the interesting principles and capabilities of TVB's simulator and give rough characterization of the execution time and memory required in typical simulations.

#### 3.1 Node dynamics

In TVB, nodes are not considered to be abstract neurons nor necessarily small groups thereof, but rather large populations of neurons. Concretely, the main assumption of the neural mass modeling approach in TVB is that large pools of neurons on the millimeter scale are strongly approximated by population level equations describing the major statistical modes of neural dynamics (Freeman, 1975). Often, averaging techniques are employed, though techniques retaining several modes have been developed (Stefanescu and Jirsa, 2008, 2011). Such an approach is certainly not new; one of the early examples of this approach consist of the well known Wilson-Cowan equations (Wilson and Cowan, 1973). Nevertheless, there are important differences in the assumptions and goals from modeling of individual neurons, where the goal may be to reproduce correct spike timing or predict the effect of a specific neurotransmitter. A second difference lies in coupling: chemical coupling is often

assumed to be pulsatile, or discrete, between neurons, whereas for mesoscopic models it is considered continuous. Typically the goal of neural mass modeling is to study the dynamics that emerge from the interaction of two or more neural masses and the network conditions required for stability of a particular spatiotemporal pattern. In the following, we shall briefly discuss some of the models available in TVB.

As we have noted, many neural mass models have been developed. One of the more prominent examples in the systems neuroscience literature is the Jansen-Rit model of rhythms and evoked responses arising from coupled cortical column (Zetterberg et al., 1978; Jansen and Rit, 1995; David et al., 2004; Spiegler et al., 2010). Advantages of the Jansen-Rit model stem from the connection made between empirical studies of neural tissue and the model's parameters, making it easier in certain cases to make concrete predictions about the relation between a dynamical regime and its neurobiological mechanism. However, because the form of the model used often employs at least six dimensions, it is not always clear how it should be analyzed or visualized. Lastly, the model requires frequent computation of exponentials, requiring considerable computational time.

For these reasons, it is often desirable to have a simpler mathematical model, which can reproduce the same qualitative phenomena as other models, implemented with fewer and simpler equations. Such is the motivation for the generic two-dimensional oscillator model provided by TVB (Strogatz, 2001; Guckenheimer and Holmes, 1983). This model can produce oscillations which are damped, spike-like or sinusoidal in nature. While these kinds of dynamics alone are not interesting, they permit the study of network phenomena, such as synchronization of rhythms or propagation of evoked potentials, while requiring less time to simulate.

However, the modeler's goals may not be served by using either the Jansen-Rit or generic 2D oscillator, so several other mass models are provided by TVB: the previously mentioned Wilson-Cowan description of functional dynamics of neural tissue (Wilson and Cowan, 1972), the Kuramoto model describing synchronization (Kuramoto, 1975; Cabral et al., 2011), two and three dimensional mode-level models describing populations with excitability distributions (Stefanescu and Jirsa, 2011, 2008), a reduction of Wong and Wang's (2006) model as presented by Deco et al. (2013) and a lumped version of Liley's model (Liley et al., 1999; Steyn-Ross et al., 1999) are among the available models in TVB.

Finally, should any of these be insufficient, a new model can be implemented with minimal effort by subclassing a base `Model` class and providing a `dfun` method to compute the right hand sides of the differential equations. Please refer to the `tvb.simulator.models` module of the main scientific library or the `contrib` folder for examples. In the `contrib` folder, models from the work of Larter et al. (1999); Breakspear et al. (2003); Morris and Lecar (1981); Hindmarsh and Rose (1984); Brunel and Wang (2001) have been implemented.

#### 3.2 Network structure

The network of neural masses in TVB simulations directly follows from two geometrical constraints on cortical dynamics. The first is the large-scale white matter fibers that form a non-local and heterogeneous (translation variant) connectivity, either measured by anatomical tracing (CoCoMac, Kötter (2004)) or diffusion-weighted imaging (Hagmann et al., 2008; Honey et al., 2009; Bastiani et al., 2012). The second is that of horizontal projections along the surface,

which are often modeled with a translation invariant connectivity kernel, approximating a neural field (however, as with other parameters in the simulator, connectivity kernel parameters that vary across space can also be used).

**3.2.1 Large-scale connectivity** The large-scale region level connectivity at the scale of centimeters, resembles more a traditional neural network than a neural field, in that, neural space is discrete, each node corresponding to a neuroanatomical region of interest, such as V1, etc. It is at this level that inter-regional time delays play a large role, whereas the time delays due to lateral, local projections are subsumed under the dynamics of the node.

It is often seen in the literature that the inter-node coupling functions are part of the node model itself. In TVB, we have instead chosen to factor such models into the intrinsic neural mass dynamics, where each neural mass's equations specify how connectivity contributes to the node dynamics, and the coupling function, which specifies how the activity from each region is mapped through the connectivity matrix. Common coupling functions are provided such as the linear, difference and periodic functions often used in the literature.

**3.2.2 Local connectivity** The local connectivity of the cortex at the scale of millimeters provides a continuous 2D surface along horizontal projections connect cortical columns. Such a structure has previously been modeled by neural fields (Amari, 1977; Jirsa and Haken, 1997; Liley et al., 1999). In TVB, a cortical mesh, as obtained from structural MRI data and simplified, provides a spatial discretization on which neural masses are placed and connected with a local connectivity kernel, itself only a function of the geodesic distance between the two masses. This is considered to provide a reasonable approximation of a neural field, the appropriateness of which depends on the properties of the mesh and the imaging modalities that sample the activity simulated on the mesh (Spiegler and Jirsa, 2013). In fact, the implementation of the local connectivity kernel is such that it can be re-purposed as a discrete Laplace-Beltrami operator, allowing for the implementation of true neural field models that use a second-order spatial derivative as their explicit spatial term.

TVB currently provides several connectivity kernels, of which a Gaussian is one commonly used. Once a cortical surface mesh and connectivity kernel and its parameters are chosen, the geodesic distance (i.e. the distance along the cortical surface) is evaluated between all neural masses (Mitchell et al., 1987), and a cutoff is chosen past which the kernel falls to 0. This results in a sparse matrix that is used during integration to implement the approximate neural field.

### 3.3 Integration of stochastic delay differential equations

In order to obtain numerical approximations of the network model described above, TVB provides both deterministic and stochastic Euler and Heun integrators, following recent literature on numerical solutions to stochastic differential equations (Klöden and Platen, 1995; Mannella, 2002; Mannella and Palleschi, 1989).

While the literature on numerical treatment of delayed or stochastic systems exists, it is less well known how to treat the presence of both. For the moment, the methods implemented by TVB treat stochastic integration separately from delays. This separation coincides with a modeling assumption that in TVB the dynamical phenomena to be studied are largely determined by the interaction of

the network structure and neural mass dynamics, and that stochastic fluctuations do not fundamentally reorganize the solutions of the system (Ghosh et al., 2008; Deco et al., 2009, 2011, 2012).

Due to such a separation, the implementation of delays in the regional coupling is performed outside the integration step, by indexing a circular buffer containing the recent simulation history, and providing a matrix of delayed state data to the network of neural masses. While the number of pairwise connections rises with  $n_{region}^2$ , where  $n_{region}$  is the number of regions in the large-scale connectivity, a single buffer is used, with a shape  $(horizon, n_{cvar}, n_{region})$  where  $horizon = \max(delay) + 1$ , and  $n_{cvar}$  is the number of coupling variables. Such a scheme helps lower the memory requirements of integrating the delay equations.

### 3.4 Forward solutions

A primary goal of TVB is not only to model neural activity itself but just as importantly the imaging modalities common in human neurosciences, using so-called forward solutions, which allow for the projection of neural activity into sensor space. To account parsimoniously for other ways in which simulated data might be saved, such as simple temporal averaging, we refer to each of these simply as *Monitors*, which take as input neural activity and output a particular projection thereof. In most cases, this takes the discrete-time form of

$$\hat{y}[j, t] = \sum_{i=1, \tau=1}^{N_W, N_k} W[j, i] K[\tau] y[i, t - \tau]$$

where  $y[i, t]$  is the amplitude of the  $i^{th}$  neural mass at time  $t$ ,  $K[\tau]$  is a temporal kernel, and  $W[j, i]$  is a spatial kernel, usually projecting the state variable of interest of the  $i^{th}$  neural mass to the  $j^{th}$  sensor.

Where necessary for computational reasons, monitors employ more than one internal buffer. The fMRI monitor is one example: given a typical sampling frequency of simulation may be upward of 64 kHz, and the haemodynamic response function may last several seconds, using only a single buffer could require many gigabytes of memory for the fMRI monitor alone. Given that the time-scale of simulation and fMRI differ by several orders of magnitude, the subsequent averaging and downsampling is justified.

In the cases of the EEG and MEG monitors,  $K$  implements a simple temporal average, and  $W$  consists of a so-called lead-field matrix as typically derived from a combination of structural imaging data, which provides the locations and orientations of the neural sources, and the locations and orientations of the EEG electrodes and MEG gradiometers and magnetometers. As the development and implementation of such lead-fields is well developed elsewhere (Sarvas, 1987; Hamalainen and Sarvas, 1989; Jirsa et al., 2002; Nolte, 2003; Gramfort et al., 2010), TVB provides access to the well-known OpenMEG package, however, the user is free to provide their own.

### 3.5 Performance

A priority of the simulator in TVB is to attain a high level of performance while remaining in pure Python. In order to see where the simulator spends most of its time, we have profiled a set of eight characteristic simulations on function timing as measured by the `cProfile` module of the standard library.

Measurements were performed on an HP Z420 workstation, with a single Xeon E5-1650 six-core CPU running at 3.20 GHz, L1-3 cache sizes 384 KB, 1536 KB and 12 MB respectively, with main

| Sim.         | Time (s) | Module/Function                   |
|--------------|----------|-----------------------------------|
| R / G2D / 20 | 11.7274  | <numpy.core.multiarray.array>     |
|              | 6.1415   | numpy.lib.npyio.loadtxt           |
|              | 5.18777  | tvb.simulator.simulator, __call__ |
|              | 3.18454  | numpy.lib.npyio.pack_items        |
|              | 2.5699   | numexpr.necompiler.evaluate       |
| 2            | 11.8713  | <numpy.core.multiarray.array>     |
|              | 6.10448  | numpy.lib.npyio.loadtxt           |
|              | 5.54546  | tvb.simulator.simulator, __call__ |
|              | 3.16143  | numpy.lib.npyio.pack_items        |
|              | 2.50565  | numexpr.necompiler.evaluate       |
| JR / 20      | 14.2185  | <numpy.core.multiarray.array>     |
|              | 9.99497  | tvb.simulator.simulator, __call__ |
|              | 7.28498  | tvb.simulator.models, dfun        |
|              | 6.202    | numpy.lib.npyio.loadtxt           |
|              | 3.24052  | numpy.lib.npyio.pack_items        |
| 2            | 14.2118  | <numpy.core.multiarray.array>     |
|              | 10.5795  | tvb.simulator.simulator, __call__ |
|              | 7.40853  | tvb.simulator.models, dfun        |
|              | 6.12069  | numpy.lib.npyio.loadtxt           |
|              | 3.25811  | numpy.lib.npyio.pack_items        |
| S / G2D / 20 | 126.618  | <_csc.csc_matvec>                 |
|              | 57.5616  | <numpy.core.multiarray.array>     |
|              | 56.1767  | <gdist.local_gdist_matrix>        |
|              | 9.055    | <numpy.core._dotblas.dot>         |
|              | 7.56356  | numpy.lib.npyio.loadtxt           |
| 2            | 125.957  | <_csc.csc_matvec>                 |
|              | 57.7524  | <numpy.core.multiarray.array>     |
|              | 56.1681  | <gdist.local_gdist_matrix>        |
|              | 12.1041  | <numpy.core._dotblas.dot>         |
|              | 7.37826  | numpy.lib.npyio.loadtxt           |
| JR / 20      | 126.312  | <numpy.core.multiarray.array>     |
|              | 56.3731  | <gdist.local_gdist_matrix>        |
|              | 19.5234  | <numpy.core._dotblas.dot>         |
|              | 9.47585  | tvb.simulator.models, dfun        |
|              | 8.87173  | <mtrand.RandomState.normal>       |
| 2            | 126.098  | <numpy.core.multiarray.array>     |
|              | 56.7912  | <gdist.local_gdist_matrix>        |
|              | 29.109   | <numpy.core._dotblas.dot>         |
|              | 14.1866  | <mtrand.RandomState.normal>       |
|              | 9.57245  | tvb.simulator.models, dfun        |

**Table 2.** Profiling results for several simulation types, “R” for region level simulations, “S” for surface level. “G2D” signifies the generic two-dimensional oscillator whereas “JR” is the Jansen-Rit model. Finally, either 20 m/s or 2 m/s conduction velocity was used. Time is given as the total time spent in the method or function listed in the right column.

memory 4 x 4 GB DDR3 at 1600 Mhz, running Debian 7.0, with Linux kernel version 3.2.0-4-amd64. The 64-bit Anaconda Python distribution was used with additional Accelerate package which provides acceleration of common routines based on the Intel Math Kernel Library. A Git checkout of the trunk branch of TVB was used with SHA 6c644ab3b5.

Eight different simulations were performed corresponding to the combinations of either the generic 2D oscillator or Jansen-Rit model, region-only or use of cortical surface, and two conduction speeds,  $v_c = 2.0$  and  $v_c = 20.0$  (m/s). In each case, a temporal average monitors at 512 Hz is used, and the results are discarded. The region-only simulation was run for one second while the surface simulation was run for 100 ms.

As can be seen in Table 2, the time spent during simulation is dominated by numerical, built-in methods (marked by those functions surrounded by angle brackets) from the NumPy and SciPy libraries, with the exception of a few calls to the `loadtxt` function, which is necessary when importing TVB’s datatypes the first time. For surface simulations, it is also necessary to compute a local distance matrix `<gdist.local_gdist_matrix>`. Nevertheless, some of the time spent on array operations such as computing the right hand sides of the differential equations can be recovered by using the `numexpr` module which eliminates the use of temporary arrays.

## 4 USER INTERACTION

### 4.1 Graphical Interface Interaction

A graphical web interface was designed for user friendly interaction with TVB . The web interface is accessible locally or remotely through a web browser; it can be used by different types of users, including those without programming knowledge, and it offers good support while the user is learning TVB concepts and workflow expectations.

**4.1.1 Projects, Accounts, Operations & Data** TVB uses several kinds of entities to model user actions and artifacts.

An *Account* or *User* is needed for accessing TVB through the web interface. When TVB web interface is started for the first time, the user is requested to provide a username and password for the first account, which acts as an *administrator*. Thereafter, others users on the same TVB server can *register* for accounts, which are validated by the administrator account.

A *Project* in TVB is a logical grouping of data and operations: for example, one could choose to create a project for each experiment in TVB, while others might create projects for each subject of a group. Each project has a single user (or account) as owner, but a project may be shared with other users to allow for collaboration.

The execution of an adapter results in an operation, in the context of a project. For example, operations are created for each simulation, a spectral analysis or surface visualization. Operations have show their status in the user interface, changing from *started* to *canceled*, *finished successfully* or *finished with error*.

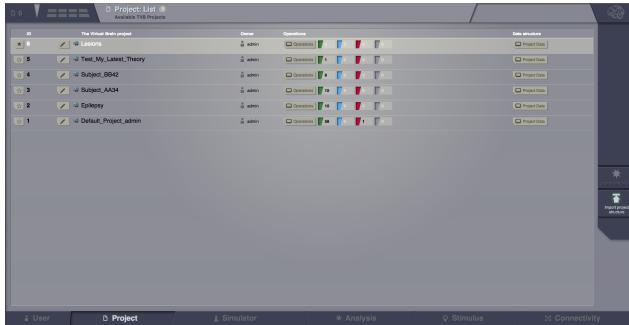
A *workflow* in TVB is, conceptually, sequence of operations from simulation, through analyses, to visualization(s). A particular workflow applies a default *tag* on the associated operations and datatypes. The user may also add custom tags to further organize data.

**4.1.2 Simulator Interface** The *Simulator* page is a configurable multicolumn interface that combines TVB simulation, analysis and visualization functionalities (Fig.3 A).

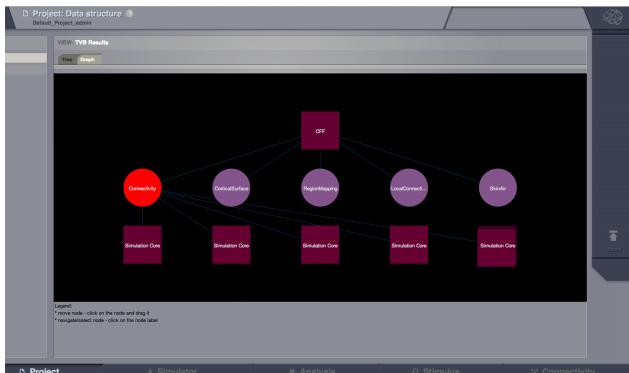
In the left column, a history of all simulations is kept, allowing quick access to previous results and their metadata. Here, the user may check metadata, add tags and rename or delete the simulation.

The middle column of the *Simulator* area is where users configure their large-scale brain network model. On the top of this column there is a blank field to name each particular configuration and a button to launch simulations. Via this column, users have access to all the configurable components that might be used in a simulation, namely:

- long range connectivity (i.e., the connectome or connectivity matrix);

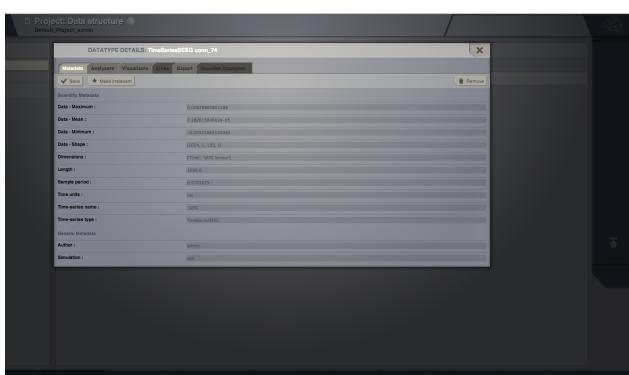


(A)



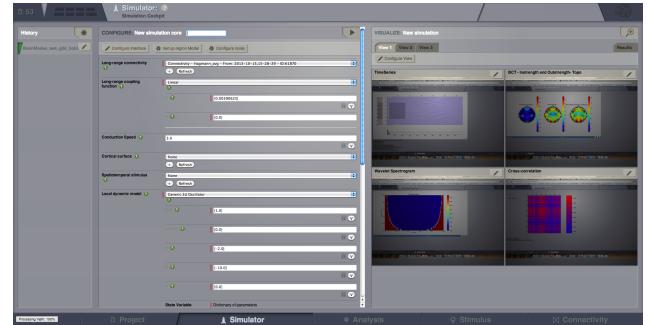
(B)

(C)

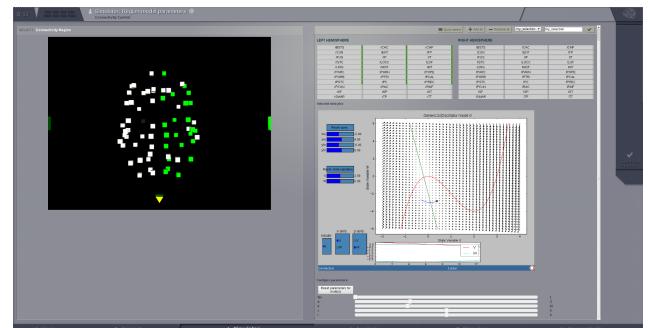


(D)

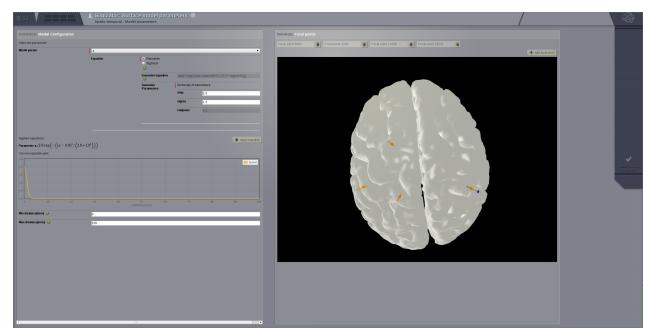
Figure 2: *TheVirtualBrain* data organization (A) View all projects (B) 2D graph display of operations with their input and output datatypes (C) View all operations in current project with their status, duration, results, etc (D) Datatype details and further available operations for it. This menu becomes available after clicking a datatype result from several places in TVB



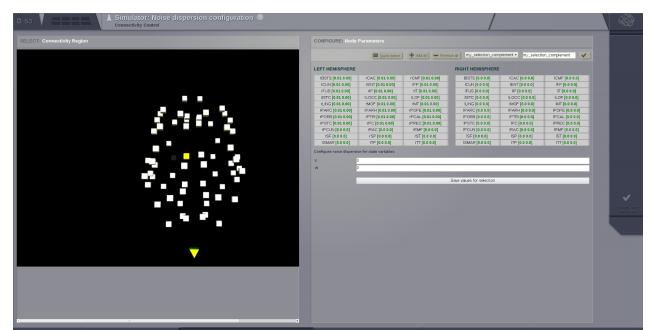
(A)



(B)



(C)



(D)

Figure 3: *TheVirtualBrain* Simulator Area: (A) View of the main interface. *Left*, the history column displays the information about the simulations and their status; *middle*, the simulation column is where simulations are configured, change of model, parameters, simulation length; *right*, the visualization column, provided configurable, multipanel visualizers to display the resulting time series or other visualizers that will add an analysis step before displaying the results. (B) Region-based simulation configuration. The parameters of the mass model can be set independently for each node. (C) Surface-based simulation configuration, allowing the user to define spatially inhomogeneous parameters. (D) Define parameters of the simulation's noise.

- long range coupling function (to scale the weights in the connectivity matrix);
- conduction speed;
- cortical surface;
- local connectivity kernel;
- local connectivity strength (can be configured to be different for each vertex);
- region mapping (correspondences between vertices of the surface and anatomical regions in the connectome);
- stimulus
- local dynamics model (i.e. neural mass model)
  - state variable range;
  - state variables to be recorded and stored;
  - initial conditions (seed and type of noise used to set random initial conditions);
- integration scheme;
  - if stochastic, the random seed can be set as well as the type of noise (white, coloured);
  - integration time step size;
  - monitors (you can set the monitors period to downsample the raw time-series)
  - simulation length

Additional information about the components (e.g., modules, datatypes and their attributes) is available by clicking on the question mark icon next to each element. This documentation is pulled from the annotations provided by the traits system and documentation strings.

Specific pages are accessible for setting up the neural mass model parameters in region-based and surface-based simulations, as well as for configuring the noise amplitude in stochastic integrations (only for region-based simulations) (Figs. 3 B, C and D). The “Set up region Model” area consist of an interactive phase-plane display. This tool shows the 2-dimensional planes of the general N-dimensional phase space of the local dynamic model. This tool is used to observe how the dynamics of the physical model change as a function of its parameters, acting as a guide to set those parameters appropriately.

On the right column, different displays can be configured to show the simulated time series and to add an analysis step with a subsequent visualization of the results, for example, computing the correlation coefficients and visualizing the resulting matrix.

For many of the parameters, such as a parameter of the local dynamics model, coupling strength, conduction speed, among others, it is possible to specify a range of evenly spaced values, and launch the corresponding sweep in parallel, using several CPU cores. When the simulation results are available, a common metric, such as average variance or degree of synchronization, is applied to the time series, and a one or two dimensional plot, determined by the number of parameters for which parameter ranges were provided, is shown, allowing the user to see how the simulation changes with the chosen parameters (Fig. 4 D).

**4.1.3 Analysis & Visualizers** TVB does not intend to provide fully featured, complex data analysis techniques, which have been well covered by other packages. Instead, we offer a minimal set of standard algorithms to quickly validate simulation results or compare with imported patient data; these include principal and independent component analyses, Fourier and wavelet spectral analyses, correlation and coherence analyses, and for connectivity, an interface to the Brain Connectivity Toolbox (Rubinov and Sporns, 2010b).

For each of the datatypes produced in TVB, one or more visualizers are available, each taking advantage of the best techniques available:

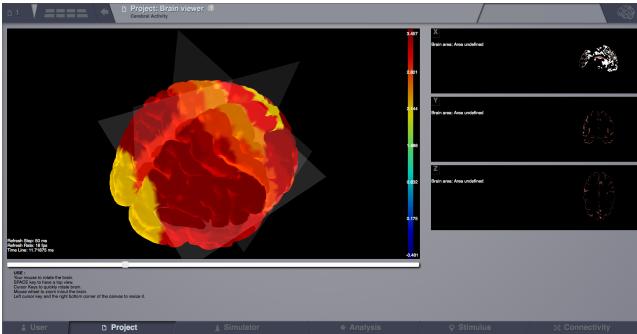
1. *WebGL viewers* are based on *HTML 5 Canvas* element and the *GL* context. These viewers offer a 3D display, user interaction with the scene (rotate, translate, zoom), and high performance for complex visualizations. WebGL is used primarily for visualizing cortical surfaces.
2. *SVG viewers* use a high-quality, highly interactive vectorial format. We use such viewers for manipulating and displaying time series, covariance or cross coherence datatype results. These visualizers were developed specifically for these data types in TVB, and provide a richer level of interaction.
3. *MPLH5 viewers* use *emphMatplotlib*’s *HTML 5* backend for viewing some of TVB’s datatypes, such as the Fourier and spectral analyses.
4. *Other* simpler viewers in TVB use the JIT <http://philogb.github.io/jit/> or FLOT <http://www.flotcharts.org/> Javascript libraries. These are mainly 2D plots for simple TVB generated data.

**4.1.4 Connectivity Tools** *Connectivity*, in the context of TVB, is another datatype, mapping structural information about a subject (a real single individual or an average template). For editing and viewing a connectivity, TVB has a dedicated interface, where the *G-User* can manipulate connectivity strength and lengths at the granularity of individual edges. Note that we do not store or use information about the exact trajectory of a connection, only the region centers and connection weights and lengths.

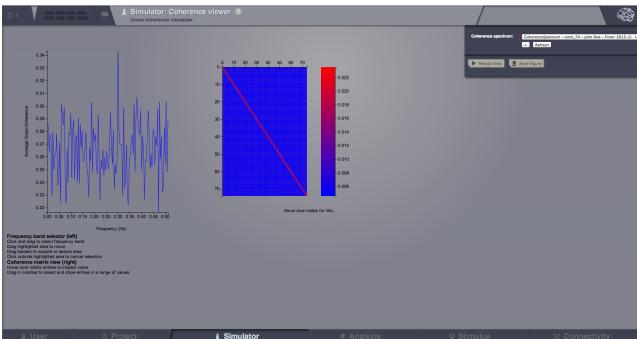
**4.1.5 Stimulus Editor** The *Stimulus* interface assists users in generating stimulation patterns that may be used later in a simulation, for example to model the effects of transcranial magnetic stimulation. The way in which stimuli are designed depends on whether the simulation will be at region-based or surface-based. For region-based simulations, a unique temporal profile can be specified for each node, with the amplitude of the stimulation modulated individually for each node. For surfaces, in addition to the temporal profile, it is possible to select several vertices on the surface to define the foci around which the spatial pattern is centered. A stimulation profile in TVB is a *pattern* datatype, either uniquely temporal or spatiotemporal depending the spatial support of the network.

## 4.2 Console and scripting

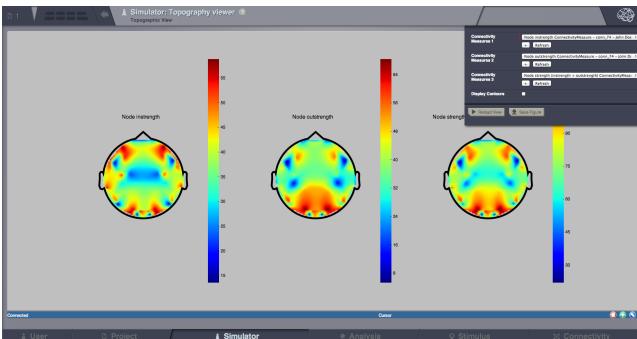
The components of TVB are of course accessible from a Python script or, more conveniently, from any of the IPython interfaces. For example, the tutorials for the simulator have been developed as IPython notebooks, due to its ability to mix text, mathematics, code and figures.



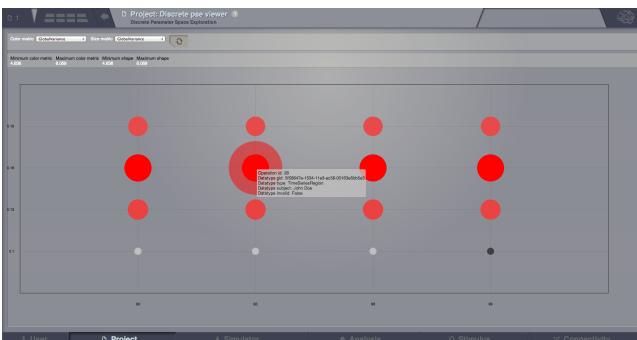
(A)



(B)



(C)

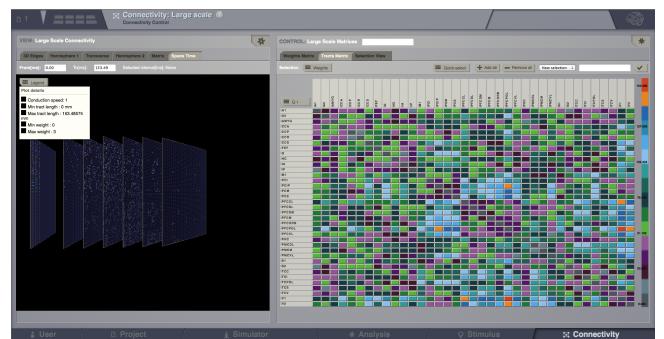


(D)

Figure 4: TheVirtualBrain visualizers: (A) WebGL: 3D display of region level simulated signal, mapped on a brain cortical surface (B) SVG: Cross Coherence (C) MPLH5: Topographic view with Connectivity in/out strength measures (D) FLOT: Parameter Space Exploration results grid



(A)



(B)

Figure 5: Connectivity tools: (A) Left: Displaying weighted connections between selection of nodes, with 3D manipulation. Right: Editing weight connections, one at a time, or many. (B) Left: Show effect of connectivity delays (in milliseconds) when conduction speed is 1 mm/ms. Right: Editing and displaying one quadrant from the matrix of connection tracts.

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

```
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 TVB'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. Each of these components can be replaced by a user's subclass of the appropriate base class, e.g. the value of `model` should subclass `models.Model`.

While TVB 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

```
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 TVB, 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).

For more details see the IPython notebook Tutorial: Anatomy of a Region Simulation [http://nbviewer.ipython.org/urls/raw.github.com/the-virtual-brain/scientific\\_library/trunk/tvb/simulator/doc/tutorials/Tutorial\\_Anatomy\\_Of\\_A\\_Region\\_Simulation/Tutorial\\_Anatomy\\_Of\\_A\\_Region\\_Simulation.ipynb](http://nbviewer.ipython.org/urls/raw.github.com/the-virtual-brain/scientific_library/trunk/tvb/simulator/doc/tutorials/Tutorial_Anatomy_Of_A_Region_Simulation/Tutorial_Anatomy_Of_A_Region_Simulation.ipynb)

## 5 CONCLUSIONS

Since the recent release of the 1.0 version of TVB, it has been officially considered *feature complete*, however, in several cases, the development of features has outstripped documentation and testing. Going forward, general priorities include advancing test coverage, improving documentation for users, and preparing PyPI and Debian packages. In the mean time, TVB's Google groups mailing list continues to provide an open forum for troubleshooting and sharing of user experiences.

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 TVB 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 TVB as well.

Lastly, as TVB was originally motivated to allow a user to move from acquired data to simulated data as easily as possible, we will continue to integrate several of the requisite steps that are not currently covered, such as analysis of DSI data to produce connectivity matrices, though in many cases, such as parcellation and labeling of cortical areas, these steps may continue to require intervention with other software. Nevertheless, TVB provides a promising platform for integrating neuroinformatics tools with an emphasis on analysis and modeling of human neuroimaging data.

## ACKNOWLEDGMENTS

Several authors have also participated in the development of TVB. They are listed in the AUTHORS file of the source code and deserve our warm acknowledgments. LP wishes to thank specifically Y. Mahnoune for his role in the design and development of the first prototype of the TVB architecture. The research reported herein was supported by the Brain Network Recovery Group through the James S. McDonnell Foundation and the FP7-ICT BrainScales. PSL and MMW received support by from the French Ministère de Recherche and the Fondation de Recherche Medicale.

## REFERENCES

- Amari, S. (1977). Dynamics of pattern formation in lateral-inhibition type neural fields. *Biol. Cybern.*, 22:77–87.
- Austin, J., Jackson, T., Fletcher, M., Jessop, M., Liang, B., Weeks, M., Smith, L., Ingram, C., and Watson, P. (2011). Carmen: Code analysis, repository and modeling for e-neuroscience. *Procedia Computer Science*, 4:768–777.
- Bastiani, M., Shah, N. J., Goebel, R., and Roebroeck, A. (2012). Human cortical connectome reconstruction from diffusion weighted mri: the effect of tractography algorithm. *Neuroimage*, 62(3):1732–1749.
- Breakspear, M., Terry, J. R., and Friston, K. J. (2003). Modulation of excitatory synaptic coupling facilitates synchronization and complex dynamics in a biophysical model of neuronal dynamics. *Network*, 14(4):703–732.
- Brunel, N. and Wang, X.-J. (2001). Effects of neuromodulation in a cortical network model of object working memory dominated by recurrent inhibition. *J. Comput. Neurosci.*, 11:63–85.
- Cabral, J., Hugues, E., Sporns, O., and Deco, G. (2011). Role of local network oscillations in resting-state functional connectivity. *Neuroimage*, 57(1):130–139.
- Coombes, S. (2010). Large-scale neural dynamics: simple and complex. *Neuroimage*, 52(3):731–739.
- Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM.
- David, O., Cosmelli, D., and Friston, K. J. (2004). Evaluation of different measures of functional connectivity using a neural mass model. *Neuroimage*, 21(2):659–673.
- Deco, G., Jirsa, V., and McIntosh, A. (2011). Emerging concepts for the dynamical organization of resting-state activity in the brain. *Nat. Rev. Neurosci.*, 12(1):43–56.

- Deco, G., Jirsa, V., McIntosh, A., Sporns, O., and Kötter, R. (2009). Key role of coupling, delay, and noise in resting brain fluctuations. *Proc. Natl. Acad. Sci. U.S.A.*, 106(25):10302–10307.
- Deco, G., Jirsa, V., Robinson, P. A., Breakspear, M., and Friston, K. (2008). The dynamic brain: from spiking neurons to neural masses and cortical fields. *PLoS Comput. Biol.*, 4(8):e1000092.
- Deco, G., Ponce-Alvarez, A., Mantini, D., Romani, G. L., Hagemann, P., and Corbetta, M. (2013). Resting-state functional connectivity emerges from structurally and dynamically shaped slow linear fluctuations. *J Neurosci*, 33(27):11239–11252.
- Deco, G., Senden, M., and Jirsa, V. (2012). How anatomy shapes dynamics: a semi-analytical study of the brain at rest by a simple spin model. *Front. Comput. Neurosci.*, 6:68.
- Freeman, W. J. (1975). *Mass Action in the Nervous System*. ACADEMIC PRESS New York San Francisco London.
- Ghosh, A., Rho, Y., McIntosh, A., Kötter, R., and Jirsa, V. (2008). Noise during rest enables the exploration of the brain's dynamic repertoire. *PLoS Comput. Biol.*, 4(10):e1000196–e1000196.
- Goodman, D. F. M. and Brette, R. (2009). The brian simulator. *Front. Neurosci.*, 3(2):192–197.
- Gramfort, A., Luessi, M., Larson, E., Engemann, D., Strohmeier, D., Brodbeck, C., Parkkonen, L., and Hämäläinen, M. (2013). Mne software for processing meg and eeg data. *Neuroimage*.
- Gramfort, A., Papadopoulou, T., Olivi, E., and Clerc, M. (2010). Openmeege: opensource software for quasistatic bioelectromagnetics. *Biomed. Eng. Online*, 9:45.
- Guckenheimer, J. and Holmes, P. (1983). Nonlinear oscillations, dynamical systems, and bifurcations of vector fields.
- Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C. J., Wedeen, V. J., and Sporns, O. (2008). Mapping the structural core of human cerebral cortex. *PLoS Biol.*, 6(7):e159.
- Hamalainen, M. and Sarvas, J. (1989). Realistic conductivity geometry model of the human head for interpretation of neuromagnetic data. *IEEE Trans. Biomed. Eng.*, 36(2):165–171.
- Herz, A. V., Meier, R., Nawrot, M. P., Schiegel, W., and Zito, T. (2008). G-node: an integrated tool-sharing platform to support cellular and systems neurophysiology in the age of global neuroinformatics. *Neural Networks*, 21(8):1070–1075.
- Hindmarsh, J. and Rose, R. (1984). A model of neuronal bursting using three coupled first order differential equations. *Proc. R. Soc. London, Ser. B*, 221(1222):87–122.
- Hines, M. L. and Carnevale, N. T. (2001). Neuron: a tool for neuroscientists. *Neuroscientist*, 7(2):123–135.
- Honey, C. J., Sporns, O., Cammoun, L., Gigandet, X., Thiran, J. P., Meuli, R., and Hagmann, P. (2009). Predicting human resting-state functional connectivity from structural connectivity. *Proc. Natl. Acad. Sci. U.S.A.*, 106(6):2035–2040.
- Jansen, B. and Rit, V. (1995). Electroencephalogram and visual evoked potential generation in a mathematical model of coupled cortical columns. *Biol. Cybern.*, 73(4):357–366.
- Jirsa, V. and Haken, H. (1997). A derivation of a macroscopic field theory of the brain from the quasi-microscopic neural dynamics. *Phys. D*, 99(4):503–526(24).
- Jirsa, V., Jantzen, K., Fuchs, A., and Kelso, J. (2002). Spatiotemporal forward solution of the eeg and meg using network modeling. *IEEE Trans. Med. Imag.*, 21(5):493–504.
- Klöden and Platen (1995). *Numerical solution of stochastic differential equations*. Springer.
- Kuramoto, Y. (1975). *Self-entrainment of a population of coupled nonlinear oscillators*, chapter 52, page 420. Number 39 in Lectures On Physics: International Symposium on Mathematical Problems in Theoretical Physics. Springer.
- Kötter, R. (2004). Online retrieval, processing, and visualization of primate connectivity data from the cocomac database. *Neuroinformatics*, 2(2):127–144.
- Larter, R., Speelman, B., and Worth, R. (1999). A coupled ordinary differential equation lattice model for the simulation of epileptic seizures. *Chaos*, 9(3):795–805.
- Liley, D. T., Alexander, D. M., Wright, J. J., and Aldous, M. D. (1999). Alpha rhythm emerges from large-scale networks of realistically coupled multicompartmental model cortical neurons. *Network*, 10(1):79–92.
- Mannella, R. (2002). Integration of stochastic differential equations on a computer. *Internat. J. Modern Phys. C*, 13(9):1177–1194.
- Mannella, R. and Palleschi, V. (1989). Fast and precise algorithm for computer simulation of stochastic differential equations. *Phys. Rev. A*, 40:3381–.
- Mitchell, J. S., Mount, D. M., and Papadimitriou, C. H. (1987). The discrete geodesic problem. *SIAM Journal on Computing*, 16(4):647–668.
- Morris, C. and Lecar, H. (1981). Voltage oscillations in the barnacle giant muscle fibre. *Biophys. J.*, 35(1):193–213.
- Nolte, G. (2003). The magnetic lead field theorem in the quasi-static approximation and its use for magnetoencephalography forward calculation in realistic volume conductors. *Phys Med Biol*, 48(22):3637–3652.
- Pérez, F. and Granger, B. E. (2007). IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29.
- Rubinov, M. and Sporns, O. (2010a). Complex network measures of brain connectivity: uses and interpretations. *Neuroimage*, 52(3):1059–1069.
- Rubinov, M. and Sporns, O. (2010b). Complex network measures of brain connectivity: uses and interpretations. *Neuroimage*, 52(3):1059–1069.
- Sarvas, J. (1987). Basic mathematical and electromagnetic concepts of the biomagnetic inverse problems. *Phys. Med. Biol.*, 32(1):11–22.
- Spiegler, A. and Jirsa, V. (2013). Systematic approximations of neural fields through networks of neural masses in the virtual brain. *Neuroimage*, 83C:704–725.
- Spiegler, A., Kiebel, S. J., Atay, F. M., and Knösche, T. R. (2010). Bifurcation analysis of neural mass models: Impact of extrinsic inputs and dendritic time constants. *Neuroimage*, 52(3):1041–1058.
- Stefanescu, R. and Jirsa, V. (2008). A low dimensional description of globally coupled heterogeneous neural networks of excitatory and inhibitory. *PLoS Comput. Biol.*, 4(11):26–36.
- Stefanescu, R. and Jirsa, V. (2011). Reduced representations of heterogeneous mixed neural networks with synaptic coupling. *Phys. Rev. E: Stat., Nonlinear, Soft Matter Phys.*, 83(2):–.
- Steyn-Ross, M. L., Steyn-Ross, D. A., Sleigh, J. W., and Liley, D. T. (1999). Theoretical electroencephalogram stationary spectrum for a white-noise-driven cortex: evidence for a general anesthetic-induced phase transition. *Phys Rev E Stat Phys Plasmas Fluids Relat Interdiscip Topics*, 60(6 Pt B):7299–7311.
- Strogatz, S. (2001). Nonlinear dynamics and chaos: with applications to physics, biology, chemistry and engineering.
- Wilson, H. and Cowan, J. (1972). Excitatory and inhibitory interactions in localized populations of model neurons. *Biophys. J.*, 12(1):1–24.
- Wilson, H. and Cowan, J. (1973). A mathematical theory of the functional dynamics of cortical and thalamic nervous tissue.

- Kybernetik*, 13(2):55–80.
- Wong, K.-F. and Wang, X.-J. (2006). A recurrent network mechanism of time integration in perceptual decisions. *J. Neurosci.*, 26(4):1314–1328.
- Zetterberg, L. H., Kristiansson, L., and Mossberg, K. (1978). Performance of a model for a local neuron population. *Biol Cybern*, 31(1):15–26.