Python in Neuroscience

Pages 1–13

Integrating neuroinformatics tools in TheVirtualBrain

M. Marmaduke Woodman [1,*], Laurent Pezard [1,*], Lia Domide [3], Stuart Knock [1], Paula Sanz Leon [1], Jochen Mersmann [2], Anthony R. McIntosh [4] and Viktor Jirsa [1*]

[1] Institut de Neurosciences des Systèmes, Aix-Marseille Université, 27, Bd. Jean Moulin, 13005, Marseille, France.
[3] Codemart, 13, Petofi Sandor, 400610, Cluj-Napoca, Romania.
[2] CodeBox GmbH, Hugo Eckener Str. 7, 70184 Stuttgart, Germany.
[4] 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 *sk:* [Is this actually true? I'm not really up to date on this aspect of things, but, the last time I paid attention this feature wasn't fully integrated/functional – If not, then it seems like a bad idea to publish that something is possible when it isn't...], 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: large-scale simulation, web platform, Python, vitrual brain, connectivity, connectome, neural mass, neural field, time delays, full-brain network model, GPUs

lp: [Emphasize the process of integrating new tools in TVB? This is not completely clear for me now.]

1 MOTIVATIONS

Neurosciences and more generally brain and behavioral sciences imply a large amount of interactions between scientific disciplines to fulfill their endeavour in the comprehension of brain and cognition relationship[c2]. Nevertheless, one major drawback of this interdisciplinary enterprise is the necessary distribution of competences most frequently between individuals but also,in a non negligeable number of cases, between intitutes. Moreover, the increase of technicalities for data analysis and brain simulation usually prevent from an optimal diffusion of these advances in the more experimentally oriented part of the community.

These problems appeal at least two orientations for their solution. Firstly, the development of up to date computing and simulating librairies using commonly used languages and secondly, the development of tools for sharing competences and data. Due to the high pace of new developments, these solutions should also remain open to incoming new tools. Several projects fall in the first category (from SPM to connectivity toolbox, fieldtrip in the MatLab "galaxy" and from "nitime" to neo... to name only a few). In the second category CARMEN (http://www.carmen.org.uk/) G-Node (https://portal.g-node.org/data/) *lp:* [Other projects?] are (web) plateform for collaborative work and data sharing. The diffusion of MatLab and its integrated environment also usually take place of the plateform / development environment.

TheVirtualBrain (TVB) provides its own solutions to these two problems. In an initial step, these two problems were addressed in two independent projects: one whole brain-simulator library developped in MatLab and a web plateform for collaborative interactions

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

[c2] Several important scientific projects such as The Human Brain Project are clear illustration of this.

in the context of multi-purpose data analysis which was developped in Python. In each case, the choice of the language was dictated both by the scientific context and the technical constraints. The whole brain simulation library was developped in MatLab due to the lp: [unfortunate] important widespread of the langage in the neuroscientific community. On the other hand, the plateform aimed a more generic purpose, envisioned several interfaces for user interaction (web and console) and should be able to adapt with existing libraries and future ones. In that case, the programming language should adapt to tasks from web development to numerical methods implementation and should be able to act as a "glue" language. These contraints lp: [naturally] oriented the development of the framework in Python[c5]. Moreover, Python is a language that provides the main blocs needed for the project: database interactions, web programming, object-oriented programming and abstraction. *TheVirtualBrain* benefits from this initial development which remain in the overall organization of the framework: the *architecture* and the *scientific library*.

1.1 Why another project? (TVB compared to others)

The reasons for developing a new project are different for each component of TVB.

1.1.1 The architecture In effect, we wish for a theoretician and a more experimentally-oriented neuroscientist to be able to collaborate; to enable such a possibility, we require a platform to enable such opportunity. 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,

The two mains constraints for the architecture were then to provide a web interface to allow remote collaboration and a data exchange system to allow the exchange of data (or simulation results) between scientists. Added to this that the user interface should have a particular design, one obtains the well-known "model-view-controller" design lp: [detail this]. In a first step, we tried to develop the architecture using an existing integrated / high-level web framework, but we rapidly found ourselves struggling with the implementation of the framework that did not fit our needs. Thanks to the high availability of Python Packages, we then choose to use more specific blocks for each purpose and build the necessary interaction between them ourselves. We then happen to choose CherryPy for the web and SQLAlchemy for the database exchanges[c1]

Moreover, the requirement for data visualisation impose to develop specific modules thanks to WebGL availability in modern browsers.

The architecture of TVB had been prototyped in Python, and in turn, both the language and the scientific ecosystem were more than rich enough to support continued developed entirely within Python, of both the architecture and the simulator, in addition to it being a general purpose language. Lastly, Python's emphasis on readability and idiomatic style facilitates integration of code contributions from programmers with disparate backgrounds.

lp: [For Lia: Was Python a "good" choice? and why? Which other language would have done the job?]

1.1.2 The simulator In the case of the simulator, the situation was a little different we happened to have a set of MatLab routines that did the job but were not ready for generic deployment. The choice happen to be between: develop our own simulator or use existing ones. As in the case of the architecture, we started to use an existing simulator, namely Brian lp: [cite BRIAN] since it has a very generic way of specifying models by differential equation expressions. Although we had some success in the implementation of several models in Brian we found ourselves in a similar situation to have to lp: [contourner] several process to reach our goals.

In fact, the core simulator began in MatLab, however, as the needs expanded, the framework 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 TVB's architecture.

The major constraints came from the scale of the simulation in TVB that is clearly different from the more usual "cellular" simulations. One needs specific population / field models and moreover should produce simulated experimental data at macroscopic levels such as EEG, MEEG and fMRI which are not usually simulated in the current neuronal simulators.

Although the scale of simulation is clearly different and imply specific orientations for TVB simulator, the overall structure of neural model remains and we happen to borrow concepts such as *monitors* (which are objects that record the course of a simulation) from Brian but implement them in the context of TVB simulations.

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. lp: [Is this really true for NEURON, GENESIS and others? I doubt...]
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 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.

A significant part of TVB is simulating brain-scale neural 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 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),

[c5] The first issue of Python in Neurosciences also confirmed the choice of the Python language.

[c1] Other dependencies of TVB are listed in `TVB_INSTALL_REQUIREMENTS` which currently lists `"apscheduler", "beautifulsoup", "cherrypy", "cfflib", "formencode", "lxml", "minixsv", "mod_pywebsocket", "networkx", "nibabel", "numpy", "numexpr", "psutil", "scikit-learn", "scipy", "simplejson", "sqlalchemy", "sqlalchemy-migrate"`.

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 TVB 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.

Brian should be a particular focus in this section, as it may be one of the closest. lp: [Dana also (but is it alive?) (http://dana.loria.fr/index.html)]

Large scale simulation implies flexible integration. We shall see how this is enable by the architecture.. mw: [expand]

1.2 Practical informations / contributors information

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.

TVB 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 simple 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 inside *TVB_Distribution* folder, to link the two. In this recommended use-case, you will have all TVB dependencies already prepared and at your disposal, as part of *TVB_Distribution*.lp: [Is there any plan for a .deb package with full dependencies taken into account in this context? Or Pypi?]

lp: [Generic description and goal of the paper]

The overall structure of TVB is depitcted on Figure 1 where components of the architecture and of the scientific library are shown with their relationships.

The goal of this article is firstly to decribre TVB framework from the development point of view and demonstrates how it interacts with other tools and how it can be extended (on the basis of extension already integrated in TVB).

2 ARCHITECTURE

TVB is logically and technically divided at deploy time into a scientific library and a framework package, where the scientific library includes datatypes, basic analyses and the simulator as its central piece, while the framework handles execution infrastructure, the web-based user interface and data storage. TVB Scientific Library can function independently, as a Python module, but TVB Framework needs the scientific library to wrap around it at runtime. lp: [How is it possible to add extensions to the current version of TVB?]

2.1 Basic Concepts

TVB has been developed with generality and modularity in mind. The central idea is data-oriented in a sense that data is fed and stored into the system and can be transformed into another type of data (including visualization) through operations provided by an external library (including TVB scientific library, but not restricted to it) that have been *adapted* to the framework. As a consequence, central concepts in TVB are *datatypes* i.e. types of data that can be handled in the framework and *adapters* i.e. classes that allow to interface/adapt external libraries to the datatypes handled by within the framework.

TVB framework provides a storage back-end, workflow management and a number of features to support collaborative work. The framework supports two user interfaces: web-based graphical interface or the console interface for advanced user and developers.

Due to the generality of the framework, it relies on the Python *abstract classes* mechanism. lp: [From Python glossary:] Abstract base classes complement duck-typing by providing a way to define interfaces when other techniques like hasattr() would be clumsy or subtly wrong.

2.2 Data: types and storage

lp: [datatypes were first defined in the "architecture" why did it moved to the scientific library?] lp: [Is there an "abstract datatype" defined? What is the process to add new datatype?]

2.2.1 TVB Traits System is inspired by the traiting module developed by Enthought **?**. Out traiting system offers a way to annotate fields and classes from inside TVB, with specific trait-attributes, which will be further used in different layers of the application. Trait-attributes are synonym with meta-data for TVB classes and their fields.

Because an explicit goal of TVB was 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, was developed, allowing for fields on a TVB class to be written out with full metadata on it. An extensive set of building blocks are already implements from numeric types and arrays to lists, tuples, string, and dictionaries.

When methods of such a class with annotated fields 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 fields 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, thus providing smooth persistence, when the storage layer is enabled. Lastly, because such metadata is used to build the docstring of a class, the IPython user also may obtain extensive descriptions of class, fields, methods and arguments in the usual way.

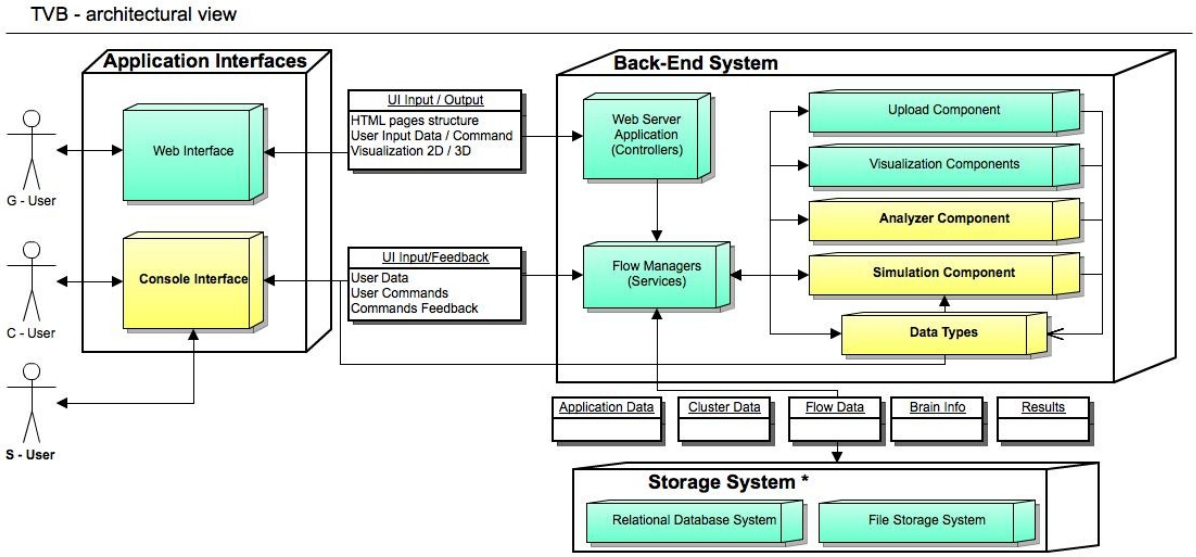


Figure 1: TVB architecture: Yellow blocks are part of the Scientific Library of TVB, while the green blocks are part of TVB Framework. TVB provides two independent interfaces, depending on the interaction type wanted by the end-user (web or console). TVB Storage layer is compulsory for the web interface, but it can be switched on/off for the console interface. [lp:] [It is said in the text that "console interface" is part of the "architecture" and not the "scientific library", this is the contrary on the figure] [lp:] [What is a "S-User"? I missed the definition?]

So, we have trait-attributes for describing how a certain class with its fields will be stored in the database and the file-storage system, for describing display manners in the web-interface, for documenting fields or classes, and even meta-data on what valid values are allowed on a certain field. The complete list of currently supported traited-attributes is described in Table **??**.

2.2.2 DataTypes In scientific Python code, it is conventional to provide arguments of an algorithm as a "bare" array or collection there of, 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 PCA and Cross Coherence algorithms.

In a more technical definition, 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

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

2.2.3 Data Storage We are storing data both in the file system and in a relational database. The file system is for storing big chunks of data, and the relational database is mainly for quick indexing, and storing entities which are used at display time in the web interface.

The relational database management is done from the code by using *SQLAlchemy* `http://www.sqlalchemy.org/`, which offers a transparent manner to connect underneath to either: SQLite or PostgreSQL, as the user chooses at install time. When choosing

Table 1. TVB currently available Traited Attributes

Traited Attribute	Description
default	Default value for current field. 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 field, when console interface is enabled.
range	Specify the set of accepted values for current field. Mark that this field is usable for parameter space exploration.
label	Short text to be displayed in UI, in front of current field.
doc	Longer description for current field. To be displayed in UI as help-text.
required	Mark current field as required for when building a new instance of the parent class.
locked	When present and *True*, current field will be displayed as read-only in the web interface.
options	Attribute present for fields of type *Enumerate*, specifying the accepted options as a list of strings.
filters_ui	Filters towards other fields, to be applied in UI.
select_multiple	When *True*, current field will be displayed as a select with multiple options in UI (default is single-select)
order	Optional number identifying the index at which current field will be displayed in UI. When negative, the field is not displayed at all. Ascending order for indices is considered when displaying.
use_storage	When *False*, current field is not stored in database or file storage.
file_storage	Valid values for this attribute are: *None* , *HDF5*, or *expandable_HDF5*, When *None*, current field is not stored in the file-storage at all. When *HDF5*, we use regular H5 file storage. When *expandable_HDF5* value is set, a H5 stored in chuncks is used.

PostgreSQL, the end-user needs to separately install and configure the database, and provide in TVB interface only the URL for connection.

ld: [Write more details here. E.g. about GIDs]

2.3 Adapters

While DataTypes provide a way of description what data do algorithms work with, sufficing for the typical user looking to write scripts against the available libraries, TVB framework 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 compulsory methods to be implemented like:

```
class ABCAdapter(object):

  @abstractmethod
  def get_input_tree(self):

  @abstractmethod
  def get_output(self):

  @abstractmethod
  def get_required_memory_size(self, **kwargs):

  @abstractmethod
  def get_required_disk_size(self, **kwargs):

  def get_execution_time_approximation(self, **kwargs):

  def configure(self, **kwargs):

  @abstractmethod
  def launch(self):
```

Code 2: The ABCAdapter listing

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 wrapping over TVB simulator library, and adjusting it to fit the workflow mechanisms inside TVB framework .
- *analyzers* which offer the interface between libraries containing algorithms for the analysis of the data (wavelets, FastICA, BCT, etc.) and the TVB framework and datatypes.
- *visualizers* are derived from the *ABCDisplayer* abstract class, and are preparing a DataType instance for display. Each Visualizer (Python adapter class) requires a complementary set of JS and HTML files for managing the actual display of data.
- *portlets* are wrapper classes for a chain of analyzers and a visualizer at the end of the chain.
- *exporters* are utility classes for preparing data in TVB (DataType or group of DataTypes) before export (web 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 an `SimulatorAdapter` which, despite being relatively complex, is build 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 ORM, 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.

Adapting sklearn's FastICA [mw: [Show the adapter for FastICA]]

```
class ICAAdapter(ABCAsynchronous):
    """ TVB adapter for calling the ICA algorithm. """

    _ui_name = "Independent Component Analysis"
    _ui_description = "ICA for a TimeSeries input DataType."
    _ui_subsection = "ica"


    def get_input_tree(self):
        """
        Return a list of lists describing the interface to the analyzer. This
        is used by the GUI to generate the menus and fields necessary for defining a simulation.
        """
        algorithm = fastICA()
        algorithm.trait.bound = self.INTERFACE_ATTRIBUTES_ONLY
        tree = algorithm.interface[self.INTERFACE_ATTRIBUTES]
        for node in tree:
            if node['name'] == 'time_series':
                node['conditions'] = FilterChain(fields=[FilterChain.datatype + '.ndim'],
                                                 operations=["=="], values=[4])
        return tree


    def get_output(self):
        return [IndependentComponents]

    def configure(self, time_series, n_components=None):
        """
        Store the input shape to be later used to estimate memory usage. Also
        create the algorithm instance.
        """
        self.input_shape = time_series.read_data_shape()
        log_debug_array(LOG, time_series, "time_series")

        ##-------------------- Fill Algorithm for Analysis -----------------##
        algorithm = fastICA()
        if n_components is not None:
            algorithm.n_components = n_components
        else:
            ## It will only work for Simulator results.
            algorithm.n_components = self.input_shape[2]
        self.algorithm = algorithm

    def get_required_memory_size(self, **kwargs):
        """
        Return the required memory to run this algorithm.
        """
        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 get_required_disk_size(self, **kwargs):
        """
        Returns the required disk size to be able to run the adapter (in kB).
        """
        used_shape = (self.input_shape[0], 1, self.input_shape[2], self.input_shape[3])
        return self.algorithm.result_size(used_shape) * TVBSettings.MAGIC_NUMBER / 8 / 2 ** 10

    def launch(self, time_series, n_components=None):
        """
        Launch algorithm and build results.
        """
        ##--------- Prepare a IndependentComponents object for result. ------##
        ica_result = IndependentComponents(source=time_series,
                                           n_components=int(self.algorithm.n_components),
                                           storage_path=self.storage_path)

        ##------------- NOTE: Assumes 4D, Simulator timeSeries. ---------------##
        node_slice = [slice(self.input_shape[0]), None, slice(self.input_shape[2]), slice(self.input_shape[3])]

        ##---------- Iterate over slices and compose final result ------------##
        small_ts = TimeSeries(use_storage=False)
        for var in range(self.input_shape[1]):
            node_slice[1] = slice(var, var + 1)
            small_ts.data = time_series.read_data_slice(tuple(node_slice))
            self.algorithm.time_series = small_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* **?**. Because it is written in MATLAB, with maintainers who prefer MATLAB, we chose not to port routines of the library to Python but instead build a MATLAB adapter which 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 the adaptation 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.

2.4 Other concepts

2.4.1 Profile TVB uses the notion of *profile* to identify in what context the application is currently running, and thus what components are expected to be plugged and how.

For example, when TVB scientific library is used alone, a specific profile (*library profile*) class gets linked as current profile, which, in this case, disables database storage and the web interface. Other profiles available in TVB are: *command profile*, *deployment profile* (with web interface), and *test profiles*.

3 SIMULATOR

The TVB 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 ancillary concepts particularly useful in the modeling of large scale brain networks.

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. Often, averaging techniques are employed, though techniques retaining several modes have been developed **?**. Such an approach is certainly not new: one of the early examples of this approach consist of the well known Wilson-Cowan equations **?**. Nevertheless, there are important differences in the 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 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 that the Jansen-Rit model of rhythms and evoked responses arising from coupled cortical column **?**. *mw:* [Continue description]. 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 to analyze or visualize. 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 may be 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. *mw:* [Continue, expand:] Model produces oscillations, damped, spike-like or sinusoidal activations, while requiring less time to solve.

However, the modeler's goals may not lead to either the Jansen-Rit or generic 2D oscillator, and several other mass models are provided by TVB: the previously mentioned Wilson-Cowan description of functional dynamics of neural tissue **?**, the Kuramoto model of synchronization **?**, two and three dimensional mode-level models describing populations with excitability distributions **??** are among the available models in TVB. Again, 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 for examples.

Because not all of the state variables of a model have the same meaning, for each of the models, we note two sets of variables, coupling variables, which must be propagated by the connectivity to provide input, and variables of interest whose values are recorded by monitors and saved for further analysis.

3.2 Network structure

The network of neural masses in TVB simulations directly follows from a pair of 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**??**) or diffusion-weighted imaging. The second is that of horizontal projections along the surface, which are usually modeled through a translationally invariant connectivity kernel though as with most parameters in TVB, spatially inhomogeneity is supported as well due to the use of generic NumPy operations which broadcast dimensions automatically.

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 delays play a large role.

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 **??**. In TVB, a cortical mesh, as obtained from structural MRI and simplified, provides a spatial discretization on which neural masses are placed and connected with a connectivity kernel, itself only a function of the geodesic distance between the two masses, and this is considered to provide an adequate approximation of a neural field, depending on the properties of the mesh and the imaging modalities that sample the activity simulated on the mesh **?**. *sk:* [The implementation of the local connectivity kernel is such that is 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, among others, the Laplacian and Gaussian kernels. 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 **?**, 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 **???**.

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 conincides 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 **????**.

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^2_{region}, 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 integrated 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, requiring 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 entirely 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 of the patient and the locations and orientations of the neural sources and the locations and orientions of the EEG electrodes and MEG gradiometers and magnetometers. As the development and implementation of such lead-fields is well developed elsewhere **???**, TVB provides access to the well-known OpenMEEG package, however, the user is free to provide his or her own.

3.5 Performance

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 **?**. *lp:* [WHy not Cython?] 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, and in the future, prepackaged versions of TVB will include precompiled objects for most kinds of simulations.

The approach used in compiling a simulation to native code takes advantage of the fact that CUDA is quite similar to C, and thus a generic template abstracts much of the boilerplate between the two. For each part of the simulator, a generic function is customized with a class specific kernel; for example, in the case of a neural mass model, we have in the Python class

```
        class Generic2dOscillator(Model):
            tau = FloatArray(...)
            # etc.

            device_info = model_device_info(
                pars=[tau, a, b, c, d, I],
                kernel="""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣float␣tau␣␣=␣P(0)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣,␣a␣␣␣␣=␣P(1)␣;␣//␣etc

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣state␣variables
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣,␣v␣␣␣␣=␣X(0)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣,␣w␣␣␣␣=␣X(1)

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣aux␣variables
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣,␣c_0␣␣=␣I(0)␣␣␣;

␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣derivatives
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣DX(0)␣=␣d␣*
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(tau␣*␣(w␣-␣v*v*v␣+␣3.0*v*v␣+␣I␣+␣c_0));
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣DX(1)␣=␣d␣*
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣((a␣+␣b*v␣+␣c*v*v␣-␣w)␣/␣tau);
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"""
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣)
```

Code 4: The Generic2dOscillator listing

where the device_info attribute is used to specify how the class's mathematical description fits into the general model function:

```
/* wrapper for model specific code computing RHSs of diff-e
__device__
void model_dfun(
  float * _dx, float *_x, float *mmpr, float *input)
{
#define X(i) _x[n_thr*i]
#define DX(i) _dx[n_thr*i]
#define P(i) mmpr[n_thr*i]
#define I(i) input[i]

    // begin model code
    \$model_dfun
    // end model specific code

#undef X
#undef DX
#undef P
#undef I
```

Code 5: The Listing

where the C preprocessor defines allow the model specific kernel to easily reference the correct parts of the multidimensional per-thread arrays (in the case of the GPU).

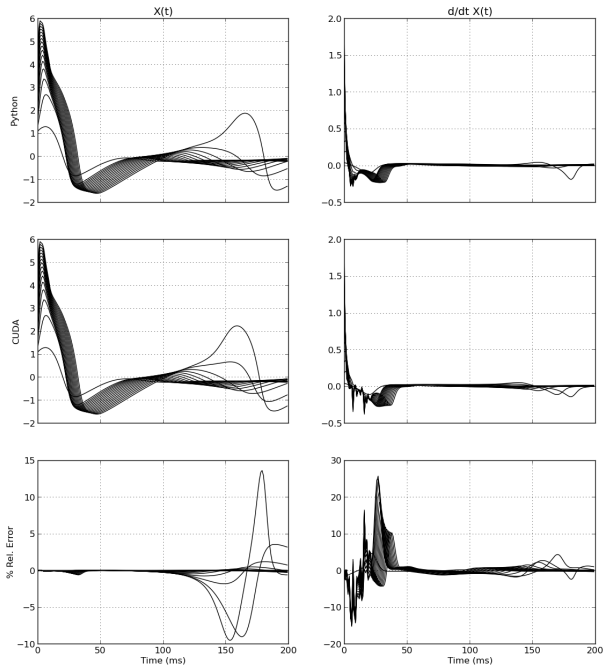


Figure 2: Right A typical parameter space exploration, 32 x 32 grid of coupling strength (y-axis) v. neural excitability (x-axis). This grid of simulations was run on both TVB's Python/NumPy implementation and the new GPU backend for 200 ms simulation time with otherwise default parameters. The former took 2 hours and the latter 1 min. Left Quantitative comparison of solutions and instantaneous derivatives is shown for an even sampling of the parameter space across k where a = -2, because this slice showed the most error on the GPU.

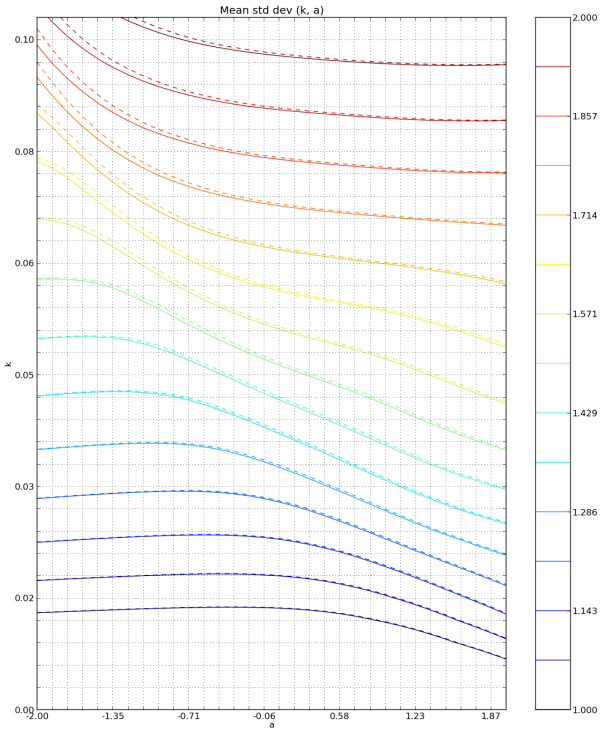


Figure 3

sk: [One figure or two, re captions]

sk: [Specify hardware (GPU/CPU) and whether Numpy is mkl-linked, to provide a more solid foundation for timing comparisons...]

sk: [It would be interesting to see a longer run (say a few seconds) to show if/how-quickly the error grows...]

As can be seen in the listing *sk:* [can listings be numbered and labelled...]*lp:* [Done see example and edit], the calculations in native code are performed with 32-bit floating point numbers, and it is reasonable to ask if this is numerically accurate. In Fig 3, we present a parameter space exploration performed with both the pure Python NumPy simulator and the GPU simulator, showing the isocontours of average standard deviation in the parameter space. Some deviation can be identified visually in parts of the parameter space, and in in Fig 2, we show in more detail time series of the Python and GPU solutions.

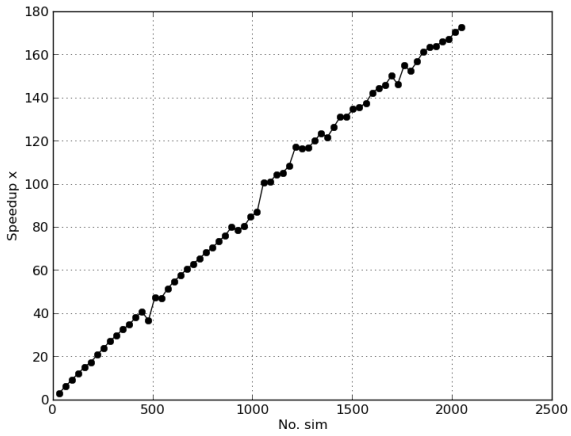


Figure 4

This approach allows significant acceleration of parameter sweeps in the case of the GPU by taking advantage of the fact that in many cases, only numerical values vary between different threads and not memory access patterns. Where one of the dimensions of a parameter sweep implies changing memory access patterns, for example conduction speed, it is advantageous to reorder the parameters, so that such memory varying parameters only change between grids of GPU threads and not within.

In Fig 4, we plot the speedup brought by the GPU over the Python NumPy simulator as a function of the number of simulations performed simulataneously on the GPU.

4 USER INTERACTION

4.1 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 TVB through the web interface a *G-User*.

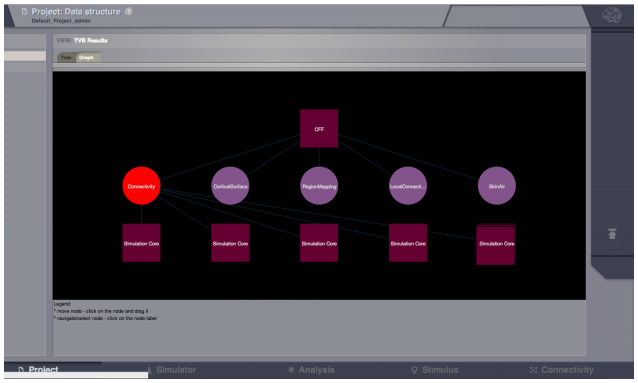
The http is served using *Cherrypy* `http://www.cherrypy.org/` which is a minimalist, object-oriented web framework, in combination with *Genshi* templating system, to support the separation of layers as guided by *MVC (Model View Controller)* pattern.

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

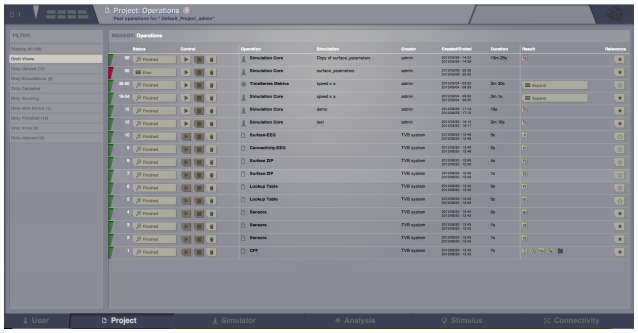
An *Account* or *User* is needed for accessing TVB through the web interface. When TVB web interface is fired for the first time, the G-User is requested to set the username and password for the first account. Later on, people can *register* for other accounts, but for using these new accounts they will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

A *Project* in TVB is a logical grouping entity, which 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

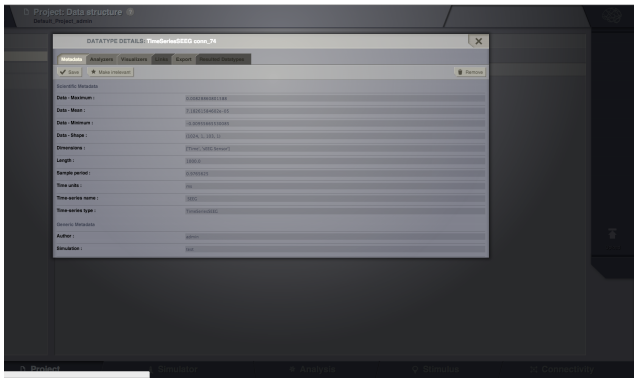
(A)



(B)



(C)



(D)

Figure 5: TVB 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

create projects for each subject they simulate. Each project has a single User (or Account) as owner, but a project 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 can have multiple input and output parameters and 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 component. 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 entities inside a Project.

4.1.2 *Simulator Interface*

ld: [Fill description for this]

4.1.3 *Analysis & Visualizers*

TVB 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 or algorithms to post-process your simulated results (or even process imported patient measured scans) inside TVB, mainly for quick validations.

We have created inside TVB 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.

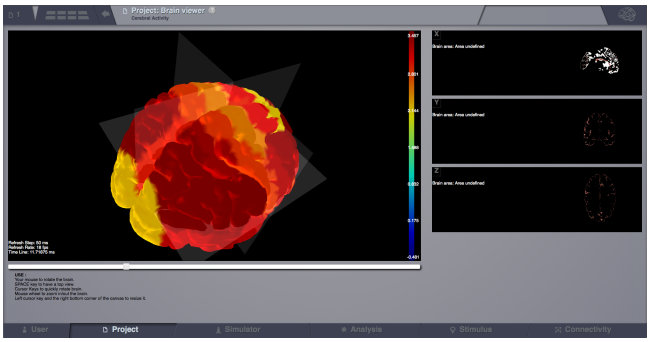
For each of the DataTypes produced in TVB, one or multiple visualizers are available. TVB has couple of visualizer types, each developed with the technology providing better support on the specific requirements for the visualization in course:

1. *WebGL viewers:* are 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 of TVB DataTypes (like Fourier or Wavelet) `https://code.google.com/p/mplh5canvas/`
4. *Other* simpler viewers in TVB are using JIT `http://philogb.github.io/jit/` or FLOT `http://www.flotcharts.org/` - JS libraries. These are mainly 2D graph displayers for some simple TVB generated data.

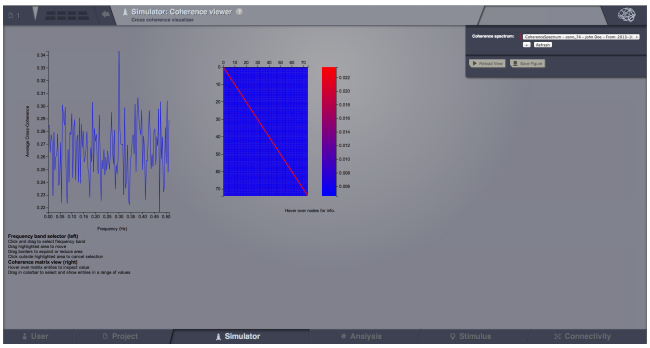
lp: [why not SVG here? Or why SVG elsewhere?]

4.1.4 *Connectivity Tool*

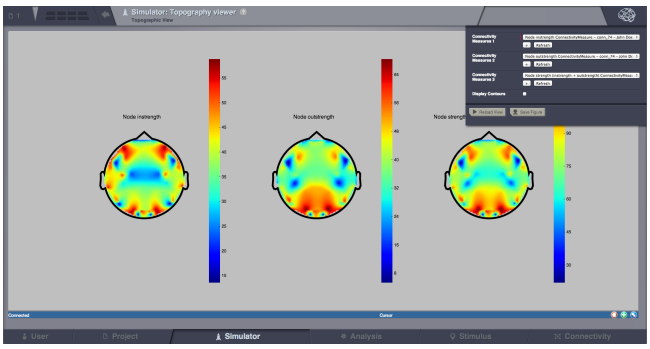
Connectivity in the context of TVB is a DataType, mapping structural information about a subject (real patient or theoretical model). For editing and viewing a Connectivity, TVB has a specific page, where the *G-User* can manipulate



(A)



(B)



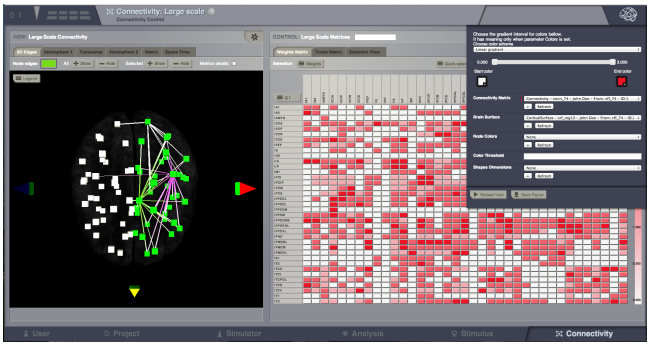
(C)

(D)

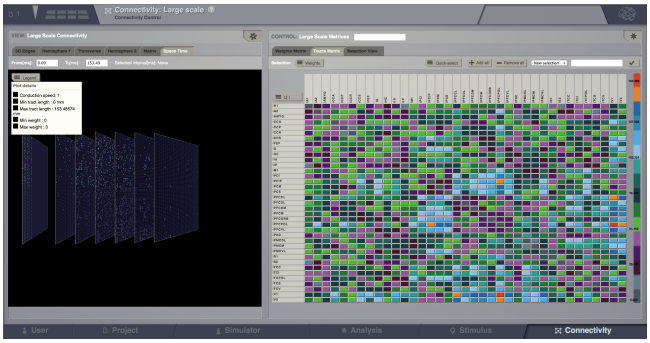
Figure 6: TVB visualizers: (A) WebGL: 3D display of region level simulated signal, mapped on a brain cortical surface (B) SVG: Cross Coherence (C) MPLH5: Topograhic view with Connectivity in/out strength measures (D) FLOT: Parameter Space Exploration results grid

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.



(A)



(B)

Figure 7: 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 conductions speed is 1. Right side: Editing and displaying one quadrant from the matrix of connection tracts.

4.2 Console and scripting interface

lp: [How does it work?]

4.2.1 Python Scripting `hello_brain.py` To give a basic feel for scripting TVB simulations, we will walk through a simple example of a region-level simulation shown in Code 6

```
from tvb.simulator.lab import *

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

sim.configure()

ys = array([y for ((t, y),)
               in  sim(simulation_length=3e2)])

# Other example
# eeg, mri = [], []
# for (t_eeg, y_eeg), (t_mri, y_mri) in sim(3e2):
#     if y_eeg is not None:
#       eeg.append(y_eeg)
    ...
plot(ys[:, 0, :, 0], 'k', alpha=0.1)
```

Code 6: Python Listing

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:
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 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

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 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 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, 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 been 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`.

4.2.2 MATLAB Scripting Due to the popularity of MATLAB in the neuroscience community, an interface from MATLAB to TVB has been introduced that allows a MATLAB script to design a TVB simulation, run it on TVB and retrieve the results. The MATLAB toolbox is provided separately from TVB, at `https://github.com/the-virtual-brain/matlab-tvb`.

The implementation of this interface is a combination of an additional CherryPy controller providing an HTTP/JSON API, running on the same server as the Web UI, and a set of MATLAB functions that send HTTP GET requests to the server. An implementation based on MEX functions invoking the Python library directly was considered, for reasons of performance, however, it was judged that such an implementation may be difficult to stabilize and maintain, given that it would require binary compatibility between MATLAB,

Python and the C compiler. Two additional advantages of an HTTP API are that most computational environments have the ability to connect and make HTTP requests, allowing other programs like Perl or Mathematica take advantage of TVB and the approach naturally extends to work over the network, should TVB be running on another machine.

In the following, we give a short demonstration and describe implementation and rationale.

```
sv = vb_url('http://127.0.0.1:8080/user/')

vb_reset(sv, 6)

info = vb_dir(sv);
sim = [];

sim.tf = 1e3 % simulation length milliseconds
sim.model.class = vb.models.Generic2dOscillator;
sim.model.a = -2.1;

sim.connectivity.class = 'Connectivity';
sim.connectivity.speed = 4.0;

sim.coupling.class = 'Linear';
sim.coupling.a = 0.002;

sim.integrator.class = 'HeunDeterministic';
sim.integrator.dt = 1e-2;

sim.monitors{1}.class = 'TemporalAverage';

sim.monitors{2}.class = 'Raw';
sim.monitors{2}.period = 1.0; % ms

# \note[sk]{Raw monitor has no period, or rather the period can't be set as it is
# fixed as the integration time step...}

[id, data] = vb_new(sv, sim);

plot(data.mon_0_TemporalAverage.ts,...
     squeeze(data.mon_0_TemporalAverage.ys)')
```

Code 7: Matlab lsiting

Because the MATLAB functions need to know the address of the server, so we take any of the URLs used by the Web UI (here, the one provided when launching TVB):

To run simulations without blocking MATLAB, a multiprocessing Pool is used. We reset the pool and change the number of processes to 6

Next, we can query the server for information on the classes available, and also get help for each of the classes
where info is a cell array of structs, one per module (models, monitors, etc.) and each struct has a field per class (models.JansenRit, models.Kuramoto, etc.). Each of these fields contains the details on the class, including all the parameters that can be set.

To build a simulation, we start with an empty struct and fill in the details for each part

Monitors are specified similarly but as a cell array there may be several of them:

sk: [Raw monitor has no period, or rather the period can't be set as it is fixed as the integration time step...]

Lastly, we submit the struct as a new simulation
Lastly, results are returned in a struct, here named data where each field contians the output of a monitor and can be plotted and analyzed as a regular MATLAB dataset:

5 FUTURE WORK

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 other essential parts of software projects. Going forward, general priorities include advancing test coverage and improving documentation for users. In the mean time, TVB'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 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 the requisite steps

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

ACKNOWLEDGMENTS

LP whishes to thank specifically Y. Manhoun for his implication in the conception and development of the first prototype of the architecture. Several authors have also participated in the development of TVB. They are cited in the AUTHORS file in TVB distribution and deserve also our warm acknowlegments.