Python in Neuroscience

Pages 1–10

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, 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, and identify potential lesion points, and then share this data with a computational expert who would then enter to contribute simulation parameter sweeps and analyses, to test which lesion point is most probably given certain empirical imaging data, et cetera; this is one of many multi-user use cases supported by TVB. TVB also drives research forward on many levels: the simulator itself represents the systematization of several recent ad-hoc simulations in the modeling literature on human rest state. In these ways, TVB serves as an integrating platform for disparate expertises in the high level analysis and modeling of the human brain. This paper will begin with a brief outline of the history and motivation for TVB as a unified project *per se*. We proceed to describe the framework and simulator, giving usage examples in the web UI and in plain Python scripting. Finally, we compare TVB with the nearest neighbors in brain modeling, simulation performance, recent advances thereupon with native code compilation and GPUs, and the role of Python and its rich scientific ecosystem in TVB.

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

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

1 MOTIVATION

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

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

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

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

In effect, we wish for a theoretician and clinician to be able to collaborate; to enable such a possibility, we require a platform to enable such opportunity. *mw:* [expand]

1.1 Why another simulator?

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

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

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

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

1.2 Why Python?

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

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

2 ARCHITECTURE

TheVirtualBrain is logically and technically divided at deploy time into a scientific library and a framework 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. *TheVirtualBrain* Scientific Library can function independently, as a Python module, but *TheVirtualBrain* Framework needs the scientific library to wrap around it at runtime.

TheVirtualBrain source code is available for download on Github at `https://github.com/the-virtual-brain/`. Previous Git and Python knowledge is required for contributing. Although you could independently install Python and the rest of *TheVirtualBrain* 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 *TheVirtualBrain* dependencies already prepared and at your disposal, as part of *TVB_Distribution*.

2.1 Basic Concepts

TheVirtualBrain 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 *TheVirtualBrain* scientific library, but not restricted to it) that have been *adapted* to the framework. As a consequence, central concepts in *TheVirtualBrain* 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.

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

Because an explicit goal of *TheVirtualBrain* 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 *TheVirtualBrain* 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.

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 **??**.

2.1.2 DataTypes In *TheVirtualBrain* , DataTypes represent the common language, to be used between different other 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, *TheVirtualBrain* DataTypes are annotated structures (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

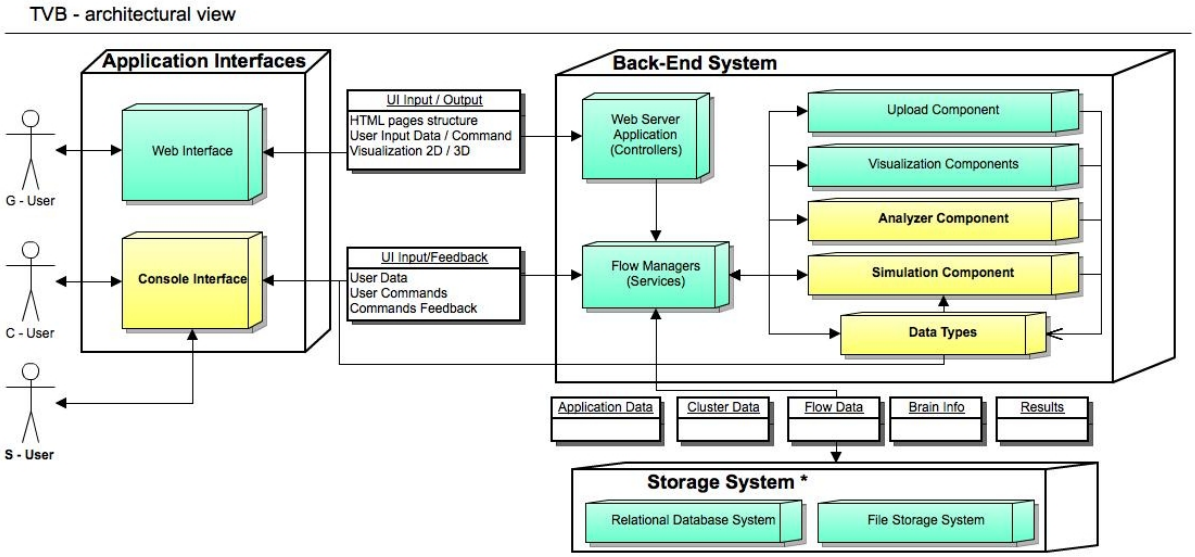


Figure 1: *TheVirtualBrain* architecture: Yellow blocks are part of the Scientific Library of TVB, while the green blocks are part of TVB Framework. TVB provides two independent interfaces, depending on the interaction type wanted by the end-user (web or console). TVB Storage layer is compulsory for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

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.

is achieved using *TheVirtualBrain* 's traiting system, mentioned in previous section.

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 *TheVirtualBrain* , we consider a *DataType* to be a full, formal description of an entity involved in an algorithm that would be part of *TheVirtualBrain* . For example, the `Connectivity` DataType, which may elsewhere be represented by a simple N by N NumPy array, is written as a class in which one of the attributes, `weights`, is a explicitly typed `FloatArray`, and the declaration of this type is complemented by explicit label, default values, and documentation strings.

```
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""")
```

2.1.3 Profile *TheVirtualBrain* 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 *TheVirtualBrain* 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 *TheVirtualBrain* are: *command profile*, *deployment profile* (with web interface), and *test profiles*.

2.2 *TheVirtualBrain* Scientific Library

2.2.1 General description

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

2.3 *TheVirtualBrain* Framework

2.3.1 General description *TheVirtualBrain* 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.

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

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

2.3.3 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, *TheVirtualBrain* 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):
```

Several categories of adapters have been defined in *TheVirtualBrain* :

- *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 *TheVirtualBrain* framework of external data, such as *gifti* files of plain *csv* files.
- *simulator* is an adapter wrapping over *TheVirtualBrain* simulator library, and adjusting it to fit the workflow mechanisms inside *TheVirtualBrain* framework .
- *analyzers* which offer the interface between libraries containing algorithms for the analysis of the data (wavelets, FastICA, BCT, etc.) and the *TheVirtualBrain* 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 *TheVirtualBrain* , the context in which an object is used is more varied, e.g. not simply initialized but loaded through SqlAlchemy's ORM, and that the adapter is required to perform more tasks than just initialization and invocation, e.g. provide expected shape of result, 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]]

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

3 USER INTERACTION

3.1 Graphical Interface Interaction

A graphical web interface was chosen as option of *TheVirtualBrain* 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 *TheVirtualBrain* concepts and workflow expectations. In our modeling diagrams, we called the actor accessing *TheVirtualBrain* through the web interface a *G-User*.

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

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

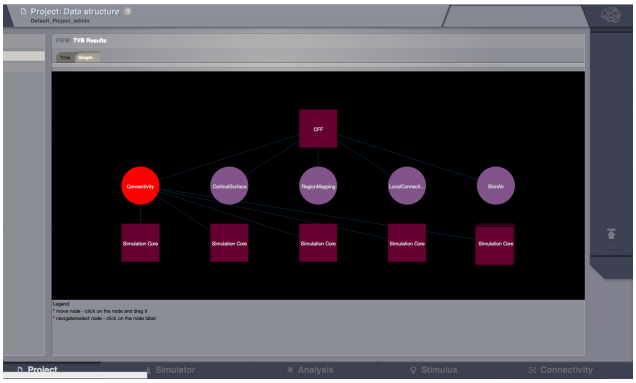
An *Account* or *User* is needed for accessing *TheVirtualBrain* through the web interface. When *TheVirtualBrain* web interface is fired for the first time, the G-User is requested to set the username and password for the first account. Later on, people can *register* for other accounts, but for using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

A *Project* in *TheVirtualBrain* 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 *TheVirtualBrain* , while others might 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.

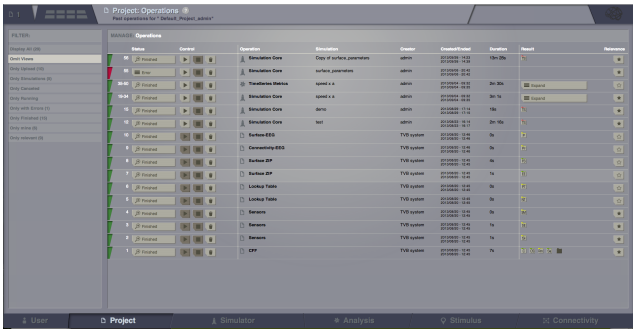
Any execution of an Adapter results into an *Operation* in the context of a project. Multiple operations will be executed under the same project. For example we will have operations created for each execution of a simulation, each run of a Fourier analyzer, or launch of a Brain Visualizer. An operation 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 *TheVirtualBrain* 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

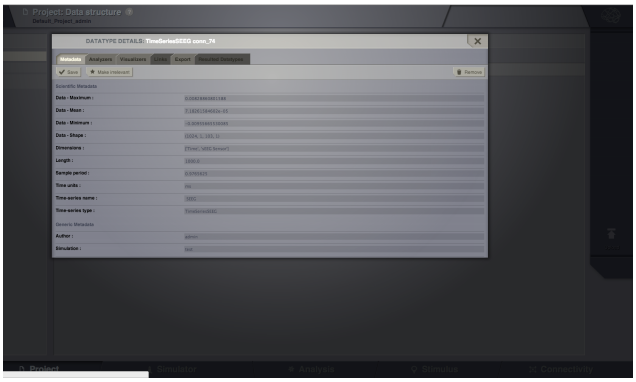
(A)



(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 *TheVirtualBrain*

both on DataTypes and Operations, for tracking entities inside a Project.

3.1.2 *Simulator Interface* [ld: [Fill description for this]]

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

We have created inside *TheVirtualBrain* adapters for *Fast ICA* from the python library *sklearn*, we've implemented a python version of *Fourier Spectral Analysis*, we have even wrapped the Matlab library *BCT* `https://sites.google.com/site/bctnet/`, and others as analyzers.

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

1. *WebGL viewers:* 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 *TheVirtualBrain* DataTypes (like Fourier or Wavelet) `https://code.google.com/p/mplh5canvas/`
4. *Other* simpler viewers in *TheVirtualBrain* 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 *TheVirtualBrain* generated data.

3.1.4 Connectivity Tool Connectivity in the context of *TheVirtualBrain* is a DataType, mapping structural information about a subject (real patient or theoretical model). For editing and viewing a Connectivity, *TheVirtualBrain* has a specific page, where the *G-User* can manipulate connectivity strength and lengths starting from the granularity of an edge.

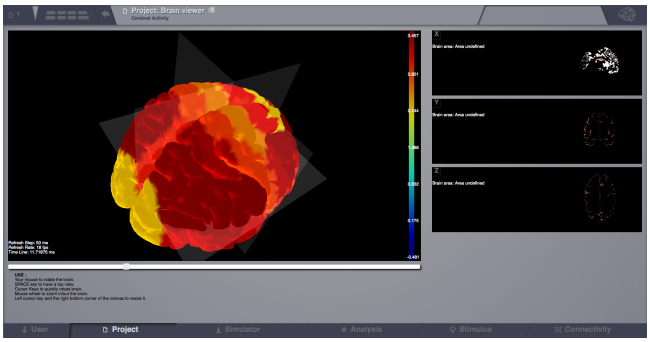
We do not store or use information about the exact anatomical path or a connection, only the region centers and connection weights and lengths.

3.2 Python Scripting

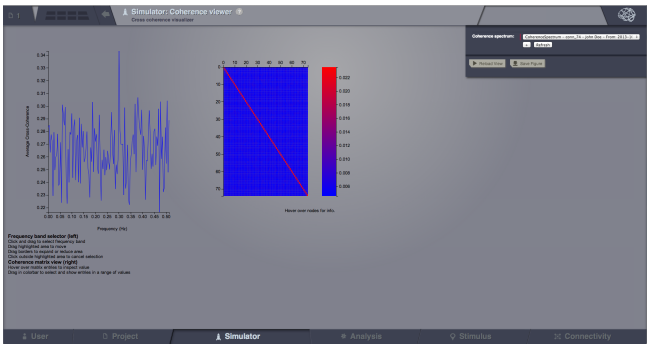
`hello_brain.py` To give a basic feel for scripting *TheVirtualBrain* simulations, we will walk through a simple example of a region-level simulation. We start with

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

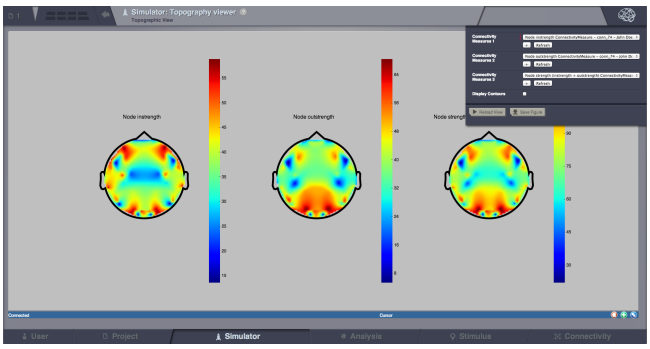
which is an all-in-one module making writing scripts shorter, in the style of `pylab`, as it imports everything from `pylab`, `numpy` and



(A)



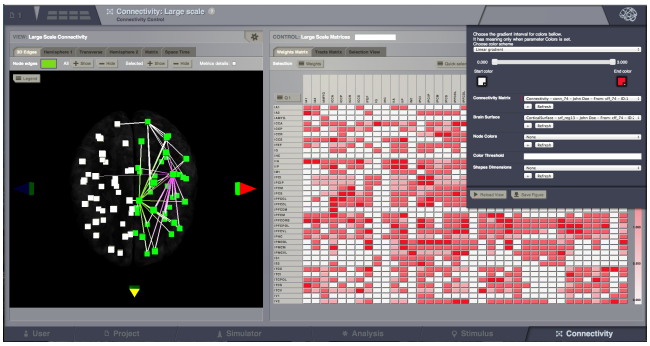
(B)



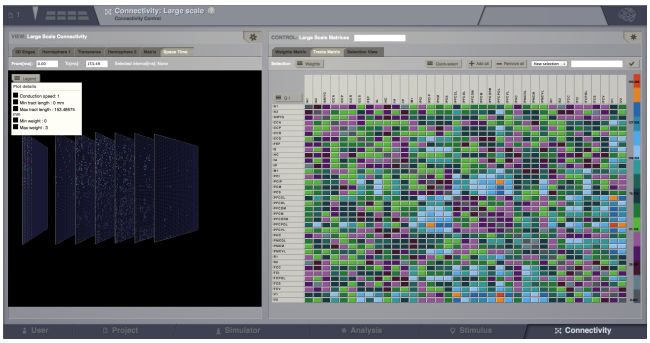
(C)

(D)

Figure 3: *TheVirtualBrain* 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



(A)



(B)

Figure 4: 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.

most of *TheVirtualBrain* 's simulator modules. Next, we build a simulator object:

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

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

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

```
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 been found in scripts distributed with the sources of *TheVirtualBrain* , or browsed online at `https://github.com/the-virtual-brain/scientific_library/tree/trunk/tvb/simulator/demos`. In the next section, we will go into detail about the different components of the simulator.

3.3 MATLAB Scripting

Due to the popularity of MATLAB in the neuroscience community, an interface from MATLAB to *TheVirtualBrain* 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`. In the following, we give a short demonstration and describe implementation and rationale.

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):

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

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

```
vb_reset(sv, 6)
```

Next, we can query the server for information on the classes available, and also get help for each of the classes

```
info = vb_dir(sv);
```

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

```
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;
```

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

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

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

Lastly, we submit the struct as a new simulation

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

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:

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

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.

4 SIMULATOR

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

4.1 Node dynamics

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

mw: [quick list of models & their relevance]

4.2 Network structure

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

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

mw: [Need to explain regional & surface simulations]

4.3 Integration of stochastic delay differential equations

Rather unlike other simulation paradigms, both noise and delays are common features of simulations in *TheVirtualBrain* . As such, the simulator is equipped with Heun and Euler methods for stochastic integration and the pairwise inter-regional delays are treated in an efficient fashion.

mw: [the general equation here]

mw: [Describe handling of delays?]

4.4 Forward solutions

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

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

4.5 Native code generation for C & GPU

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

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

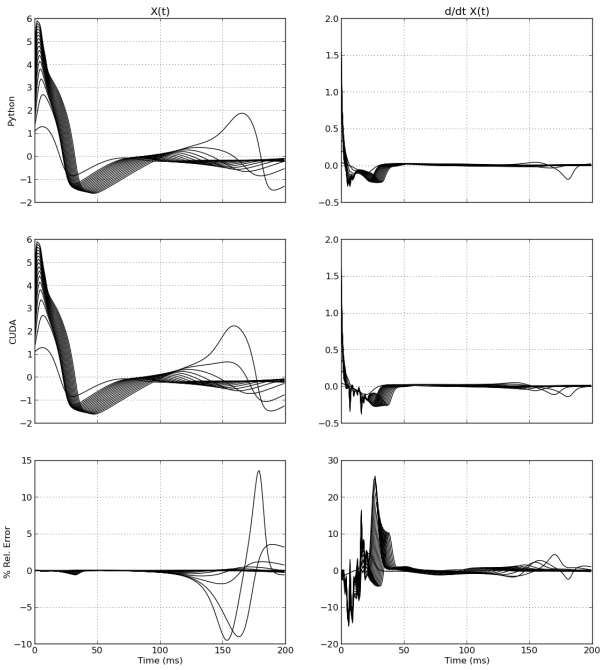


Figure 5: Right A typical parameter space exploration, 32 x 32 grid of coupling strength (y-axis) v. neural excitability (x-axis). This grid of simulations was run on both TVB's Python/NumPy implementation and the new GPU backend for 200 ms simulation time with otherwise default parameters. The former 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.

```
/* wrapper for model specific code computing RHSs of diff. eqs */
__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
```

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).

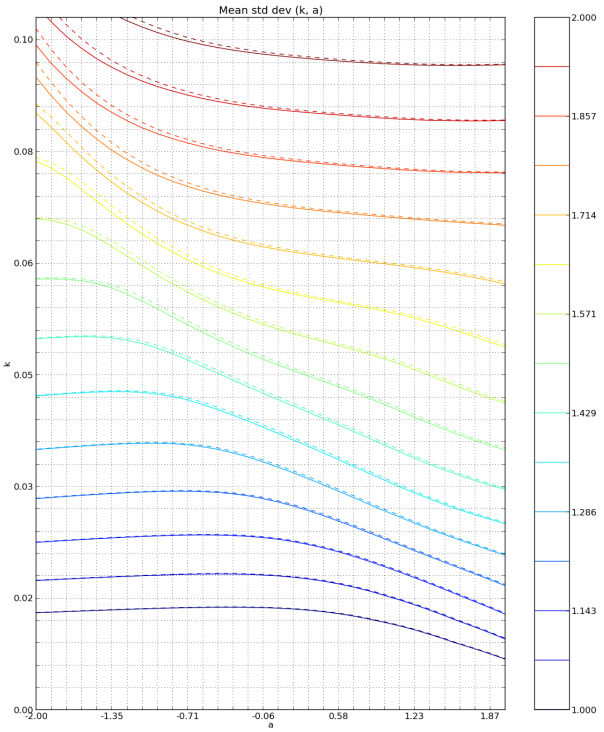


Figure 6

As can be seen in the listing, 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 7, 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 5, we show in more detail time series of the Python and GPU solutions.

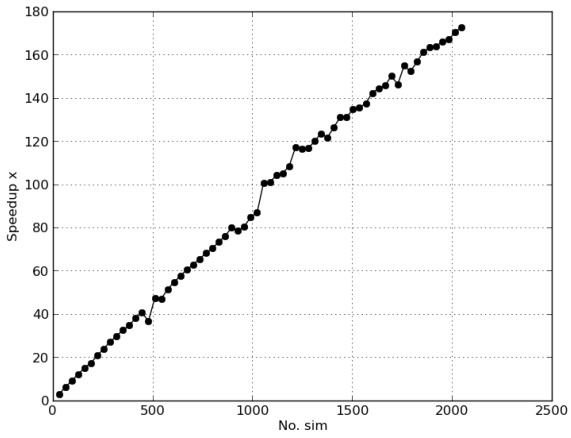


Figure 7

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 velocity, 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 **??**, 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.

4.6 Other simulators compared to *TheVirtualBrain*

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

- several spiking models in the literature use the notain of a reset condition and potential, for which Brian provides explicit functionality. TVB does not because typically the deterministic dynamics of neural mass models are continuous.

5 FUTURE WORK

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

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

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

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

REFERENCES