Helmholtz: a customizable framework for neurophysiology data management

Andrew P. Davison[1], Thierry Brizzi[1], Domenico Guarino[1], Olivier Manette[1], Cyril Monier[1], Gérard Sadoc[1], Yves Frégnac[1]

[1]Unité de Neurosciences, Information et Complexité, Centre Nationale de la Recherche Scientifique (CNRS), Gif sur Yvette, France

Data problem

Databasing of experimental neurophysiology data together with the annotations/metadata needed to understand it promises major payoffs both for the scientists who generate the data and for the progress of neuroscience in general.

However, systematically putting the annotations and other metadata into a digital form is generally an arduous task at present, and the benefits difficult to realize, so that the cost/benefit ratio for the experimentalist is a poor one, with the corollary that the flow of data shared with the broader community is more a trickle than a flood.



To improve this situation requires tools that make it easier for neuroscientists to put their data into a database, to annotate it appropriately, to obtain some immediate benefit from the effort, and to share the data with others, under terms they control, if they so wish.

Here we present Helmholtz, an open-source framework for developing databases that are customized to the needs of individual neurophysiology labs, named after the 19th century physicist and physiologist *Hermann Ludwig Ferdinand von Helmholtz* for his work on perception and in the hope that we can contribute to the conservation of energy in neuroscience **:)**

The tension between immediate benefit in having a better tool for scientists to manage their own data and longer-term considerations of sharing with others has several implications:

1. the metadata stored should be **customizable**, since the needs of different labs can vary widely, but there should also be a common core to ensure interoperability if the data are published
2. we should support storing both **raw and processed/analyzed data**, so that the scientist can manage all phases of their workflow and so that the provenance of an individual result or graph can be easily tracked
3. the same tool should be usable both as both **local and public resource**
4. both data and metadata should have **fine-grained and easy-to-use** access controls

Web Solution

The framework is built on top of the Django web framework (http://www.djangoproject.com/). The advantages of using a web framework are:

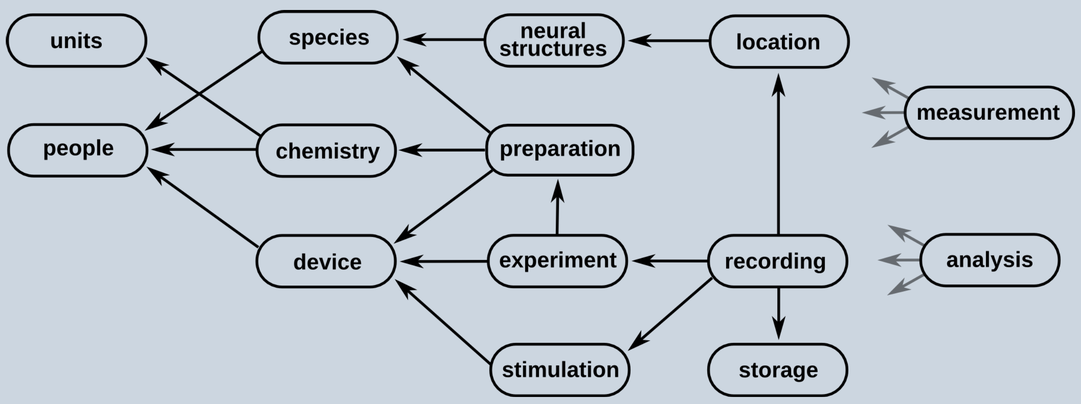
1. it makes it **easy to setup** either a **public or local database** (Django comes with a simple built-in web-server)
2. a **highly modular structure** makes the database **easy to customize and extend**
3. **abstraction of the underlying database layer**, so that (i) any supported relational database can be used (e.g. MySQL, PostgreSQL, Oracle or the built-in SQLite); (ii) knowledge of SQL is not required, making it **easy for non-database specialists to develop tools and extensions**
4. it is easy to develop **multiple interfaces**, e.g. a web interface, a web-services interface, interfaces to desktop acquisition or analysis software

Architecture

Helmholtz provides core components which handle elements that are common to all or many domains of neurophysiology. For example, information about data acquisition: metadata for **experimental setups** (equipment, etc.), **subjects** (species, weight, anaesthesia, surgery, etc.), **stimulation** and **recording protocols**, for electrophysiology (in vivo and in vitro), optical imaging and morphological reconstructions.

A generic form of referencing supports databasing of **measurements** and **analysis** methods and steps, which can be linked to any object in the database, supplying the first building block for additional operations.

Each component of the structure is represented as a Django application and has a database table associated with it.



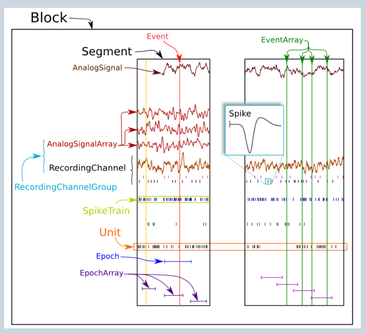
Extension components to support the specific needs of individual labs are straightforward to write, requiring minimal programming experience.

In order to explain the principles underlying Helmholtz architecture, let's take a closer look at the central data structures representing metadata associated with recordings.

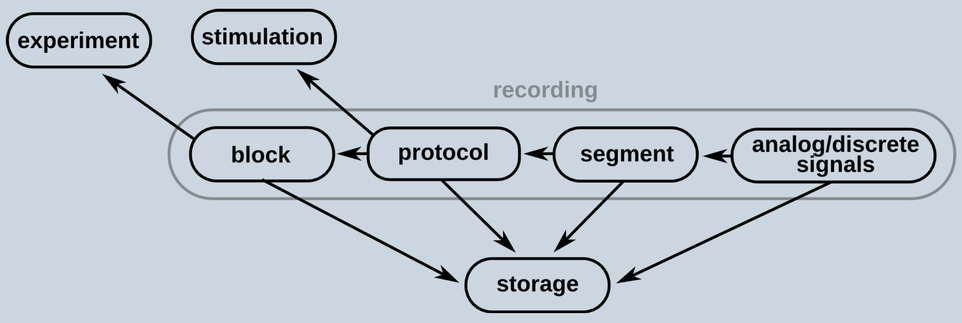
Following interoperability directions, Helmholtz structures for recording metadata mirror those in **Neo**, a python package for representing hierarchies of electrophysiology data and metadata, together with support for reading and writing a wide range of neurophysiology file formats, including Spike2, NeuroExplorer, AlphaOmega, Axon, Blackrock, Plexon, Tdt, Matlab and HDF5.

A direct mapping of **Neo**'s object at attribute level exists among **Block**, **Segment** (with reference to the protocol presentation), analog and discrete **Signal**s, the actual recordings.

These objects are integrated seamlessly into Helmholtz architecture, by including references to the experiment and stimulation metadata that generated the recordings, as well as the storage location of files containing the recordings.



Neo 0.3 architecture



Future plans

Following deployment of the web-services API for the BrainScaleS database, the next steps are:

- Implement a high-level Python client library, building on top of requests, enabling simulation and analysis scripts to directly and automatically record their parameters, inputs and outputs in the database
- to fully develop the analysis support in order to capture, for each step, not only the input and output (already implemented) but also the algorithm used, with possible additional parameters
- further development of the query interface

Interactions

To populate, inspect and change the metadata, Helmholtz users have several options ranging from python interactive shell to self-developed ones. Let's take a look at some of them...

Python shell

An experimental setup is defined by its location and the various devices that makes it up:

```
>>> from helmholtz.devices.models import Setup, SubSystem
>>> amps = SubSystem(label="amplifiers")
>>> amps.save()
>>> amps.device.add(my_amp)
>>> amps.save()
>>> vis = SubSystem(label="visualisation")
>>> vis.save()
>>> my_setup = Setup(place=my_lab, room="C253")
>>> my_setup.save()
>>> my_setup.subsystems.add(amps, vis)
>>> my_setup.save()
```

Once the general information about the experimental setup are stored in Helmholtz, a new Experiment object can be created:

```
>>> from helmholtz.devices.models import Setup
>>> from helmholtz.people.models import Researcher
>>> setup = Setup.objects.get(room="B254")
>>> researchers = Researcher.objects.filter(first_name__in=["Alice", "Bob"])
>>> expt = Experiment(label=datetime.now().strftime("%F"), start=datetime.now(), setup=setup)
>>> expt.save()
>>> expt.researchers.add(*researchers)
>>> expt.save()
```

Of course, several other elements could be included in the description, like animal characteristics, drugs applications, preparation properties ...

Web Service API

We decided to implement a web-services API using a RESTful architecture, so to see the database as a **collection of resources**, each of which is referenced with a globally unique identifier (URI).

A client that wishes to obtain information from the database sends an HTTP GET request to the URL of the resource of interest, and receives a representation of that resource. This representation could be in several different formats. The Helmholtz API supports **XML**, **JSON**, **YAML**, **odML** and **HTML**. The preferred format is included as part of the query or as part of the URL.
As an example, a query to:

```
https://www.dbunic.cnrs-gif.fr/visiondb/experiments/?format=xml
```

will return the following document:

```
−<response>
 −<objects type="list">
  −<object>
    <end>2011-05-24T00:12:38</end>
    <setup>/device/setup/2/</setup>
    <label>201308</label>
    <preparation>/preparation/preparation/2/</preparation>
    <start>2011-05-22T12:34:56</start>
    <id type="integer">2</id>
    <type>Simulation</type>
    <notes/>
   −<researchers type="list">
     <value>/people/researcher/4/</value>
    </researchers>
    <resource_uri>/experiment/experiment/2/</resource_uri>
  </object>
  −<object>
    <end>2008-06-03T19:01:34</end>
    <setup>/device/setup/7/</setup>
    <label>CAT_VISUAL_INVIVO_INTRA_2008_23</label>
    <preparation>/preparation/preparation/105/</preparation>
    <start>2008-06-03T19:01:34</start>
    <id type="integer">103</id>
    <type>CAT VISUAL INVIVO INTRA</type>
    <notes/>
    <researchers type="list"/>
    <resource_uri>/experiment/experiment/103/</resource_uri>
  </object>
 </objects>
 −<meta type="hash">
```

any web browser can read the API

Several features can already be noted from this example:

- representations are both human- and machine-readable, which facilitates debugging
- the same information can be returned in different formats, depending on the needs of the client
- references to other resources (as for *preparation*, *setup* and *researchers*) take the form of a URL (relative to the host)

If a client wishes to modify this resource (and has the permission to do so) it sends an HTTP PUT (replace the entire resource with a new one) or PATCH (modify part of the resource) request to the same URL, containing a JSON document in the same format, e.g.

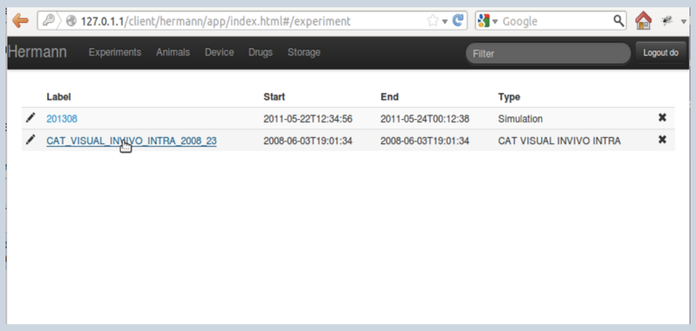
```
$ curl --dump-header - -H "Content-Type: application/json" \
  -X GET https://www.dbunic.cnrs-gif.fr/visiondb/experiment/experiment/
{
  "id": 103,
  "label": "CAT VISUAL INVIVO INTRA 2008 23"
}
```

To create a new resource in the database, the client sends an HTTP POST request, and return to the client the URL of the new resource. Finally, an HTTP DELETE request can be used to delete a resource.

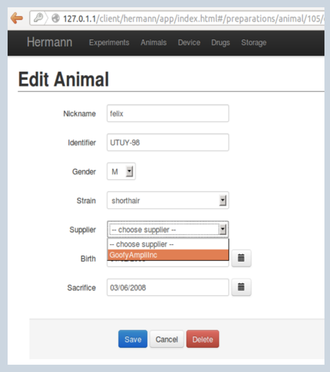
The amount of data stored can rapidly grow and be overwhelming, not only for humans but also for automatic treatment. Here we show how easy is to add some filtering criteria using the API. For example, here we ask for the subset of Drifting Grating stimuli that have contrast 10%:

```
>>> import requests
>>> adds = {'contrast': '10.00 \%'}
>>> heads = { 'Authorization': 'Basic ZG86ZG8=' }
>>> r = requests.get('https://www.dbunic.cnrs-gif.fr/visiondb/stimulation/driftinggratingstimulus/', params=adds, headers=heads)
>>> r.status_code
200
>>> r.text
{"objects": [
    {
      "contrast": "10.00 \%",
      "resource_uri": "/stimulation/driftinggratingstimulus/4/",
    },
    ... other objects
  ]
}
```

Using the Python **requests** library, it is easy to interact with the API, but please note that we do not expect users to interact with the database at such a low level. Rather, tool developers will wrap the API interaction within a graphical interface or within a programming-language-specific high-level library.
Using this API and AngularJS, a **ready-to-use electronic lab notebook** is under construction and it will be deployed together with the rest of Helmholtz code:



WebApp: Experiment list



WebApp: Edit Animal details

This will serve the purpose of fully detailed example application, enabling further customizations or other implementations.

Acknowledgments

This work was supported by European Union projects FP7-269921 (BrainScaleS) and FP6-015879 (FACETS).

Contact us at {name.surname}@unic.cnrs-gif.fr