PyBpod Documentation

Release 1.7.8

['Ricardo Ribeiro']

Dec 09, 2019

GETTING STARTED

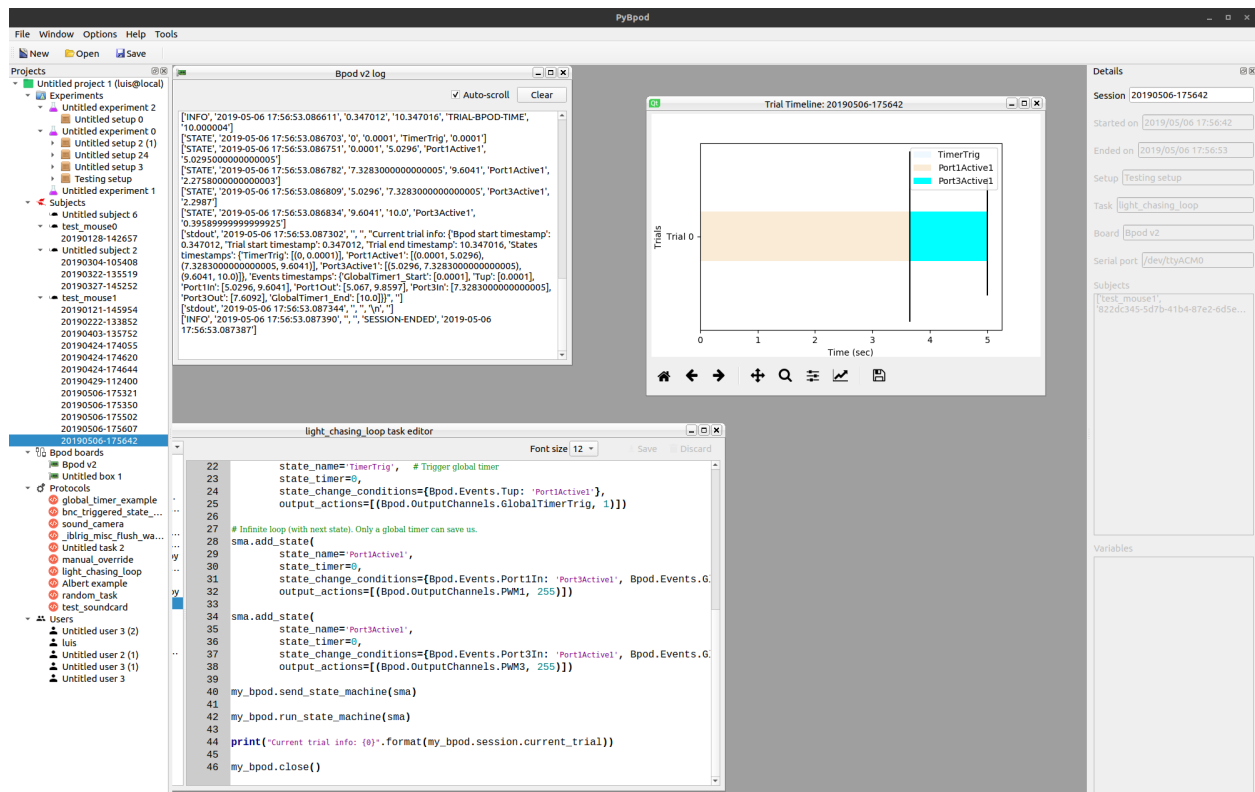**Note:** All examples and Bpod's state machine and communication logic were based on the original version made available by Josh Sanders (Sanworks).

WHAT IS PYBPOD?

PyBpod is a GUI application that enables interaction with the latest Bpod devices version.

This project is maintained by a team of SW developers at the Champalimaud Foundation. Please find more information on section *Project Info*.

1.1 What is Bpod?

Image retrieved from: https://sanworks.io/shop/products/images/1001_2.jpg

Bpod is a system from Sanworks for precise measurement of small animal behavior. It is a family of open source hardware devices which includes also software and firmware to control these devices. The software was originally developed in Matlab providing retro-compatibility with the BControl system.

See also:

Bpod device: https://sanworks.io/shop/viewproduct?productID=1011

Bpod on Github: https://github.com/sanworks/Bpod

Bpod Wiki: https://sites.google.com/site/bpoddocumentation/

BControl project: http://brodywiki.princeton.edu/bcontrol/index.php/Main_Page/

1.2 Why a Python port?

Python is one of the most popular programming languages today [1]. This is special true for the science research community because it is an open language, easy to learn, with a strong support community and with a lot of libraries available.

QUESTIONS?

If you have any questions or want to report a problem with this library please fill in an issue here.

CONTENTS

3.1 Installing and updating

Note:

Linux

- Make sure your user has permissions to access the serial ports.
- Execute the next command:

```
sudo usermod -a -G dialout [your username]
```

- Restart the computer.

3.1.1 User installation

1. Install Python 3.6.
2. Install PyBpod from PyPi:

```
pip install pybpod
```

3. Execute PyBpod:

```
start-pybpod
```

Note: On the first execution a user_settings.py file will be created on the User system folder.

3.1.2 Installation for developers

1. Download & install Anaconda or Miniconda.

Warning:

Windows

> - On windows if you install Anaconda/Miniconda for all the users, you should make sure you run the "Anaconda Prompt" as administrator.
> - To avoid issues, make sure you install Anaconda/Miniconda only for your user.

2. Download the environment configuration file for your Operating System and create a virtual environment with it by executing the following commands in the "Anaconda Prompt".

Windows 10: environment-windows-10.yml (right click->Save Link as):

```
conda env create -f utils/environment-windows-10.yml
```

Ubuntu 17.10 and up: environment-ubuntu-17.10.yml (right click->Save Link as):

```
conda env create -f utils/environment-ubuntu-17.10.yml
```

Mac OSx: environment-macOSx.yml (right click->Save Link as):

```
conda env create -f utils/environment-macOSx.yml
```

3. Activate the environment you just created.

```
activate pybpod-environment
```

4. Clone the PyBpod repository and initialize all the submodules.

```
git clone https://bitbucket.org/fchampalimaud/pybpod.git
git submodule update --init
```

5. Access the created repository folder.

```
cd pybpod
```

6. Run the "install.py" script to install all necessary dependencies.

```
python utils/install.py
```

7. Run the PyBpod application.

```
start-pybpod
```

3.1.3 Execute PyBpod

1. Open "Anaconda Prompt" and activate the "pybpod-environment".

```
activate pybpod-environment
```

2. Run the application, in your pybpod directory.

```
start-pybpod
```

3.1.4 Update PyBpod

1. Open the "Anaconda Prompt" and activate the "pybpod-environment".
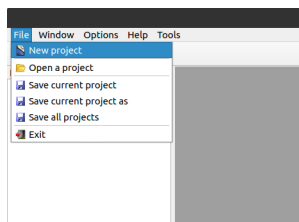
```
activate pybpod-environment
```

2. Execute the next commands in your pybpod directory.

```
git pull
git submodule update --recursive --remote
```

3.2 Basic usage

3.2.1 Projects

When you open PyBpod for the first time, you can create a new project or load a previous project from your filesystem.

With PyBpod you can easily organize your work. Projects allow you to aggregate several experiments in one place.

Note: You can open several projects at the same time to compare data.

Each project has a set of experiments, subjects, boards, protocols and users as it is possible to see from the example in the next figure.
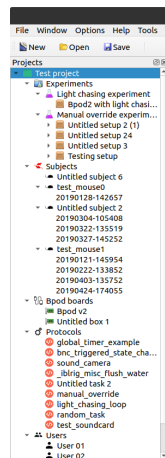
An experiment can have one or more Setups. Each Setup mostly defines an association between a Board and a Protocol. It is also possible to associate one or more subjects as well as variable definitions for that particular Setup.

When running a particular Setup, the Session data will appear under it in the project tree as well as under the subjects that were part of that Session.

3.2.2 Bpod boards

PyBpod GUI supports multiple Bpod boards to be run in parallel. Just add a new board, select serial port, run your experiment and open the console window.
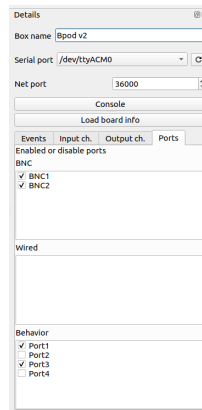
The console window allows you to see real time output from the Bpod.
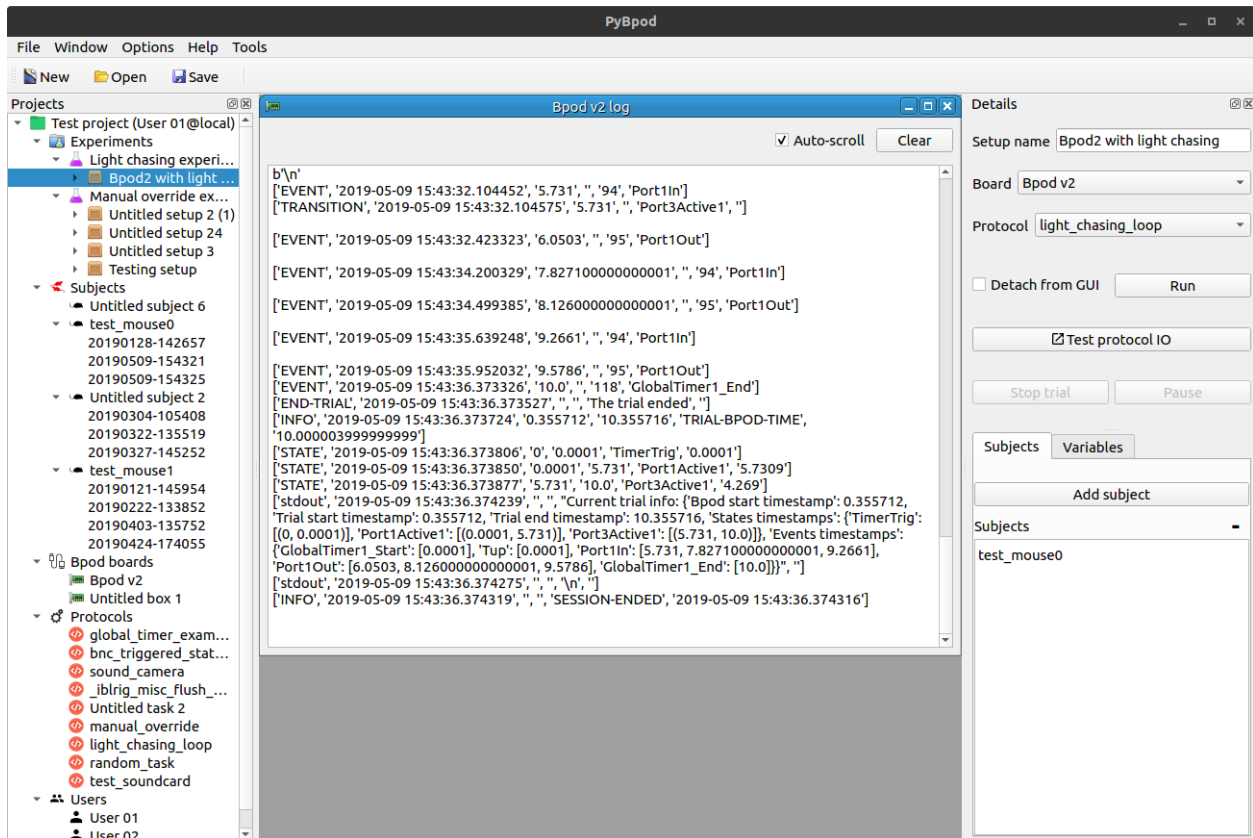
Note: On the Board details window you can activate or deactivate ports by checking or unchecking them in the ports list.

3.2.3 Subjects

The subjects allow you to manage the animals for your experiments. By keeping the list of animals updated you will have better control of your experiments. PyBpod allows you to know in which Sessions each subject participated and you can see details for each Session within PyBpod.
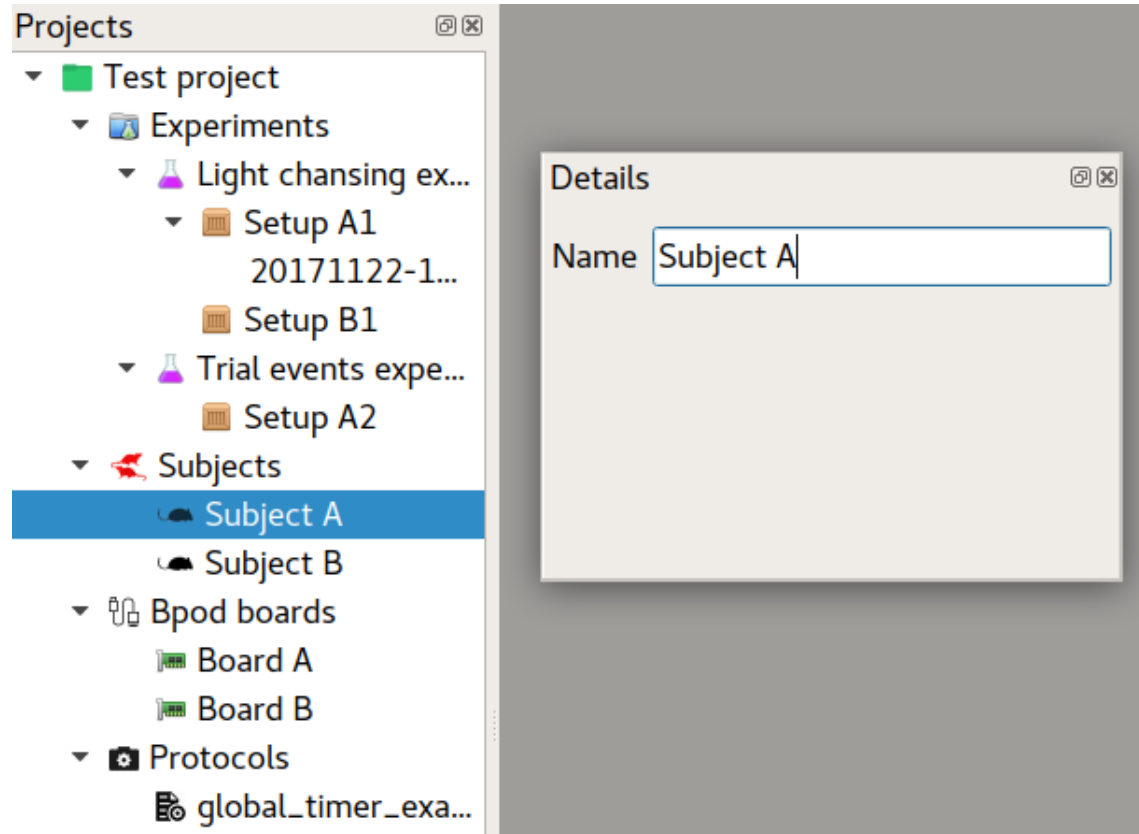
3.2.4 Protocols

Protocols allow you to define how the state matrix works. They are fully written in Python but follow a similar syntax from the Bpod Matlab equivalents.

PyBpod GUI ships with a code editor with syntax highlight and you don't have to hardcode the serial port or other settings. Let the GUI do the job for you and focus on your experiments!
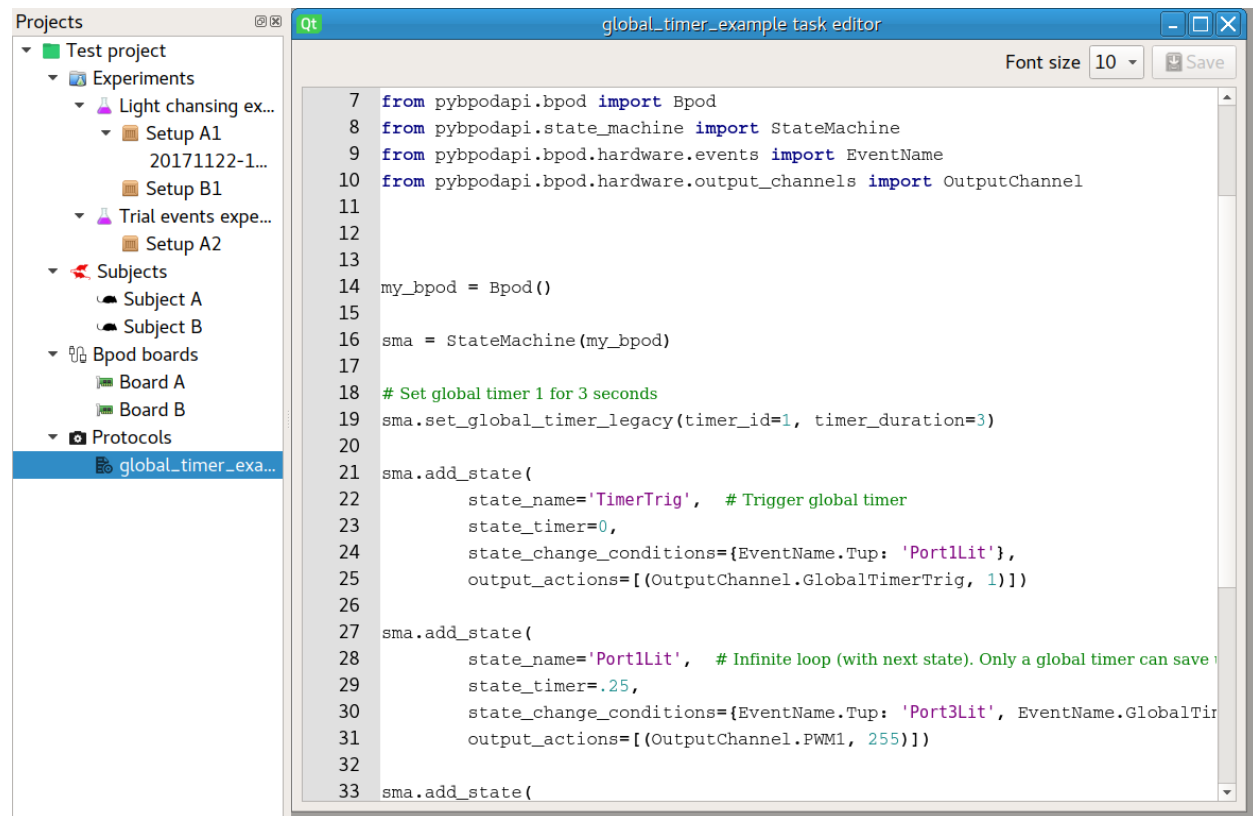
3.2.5 Experiments and experimental setups

The experiments node hold all your experiments important data. Each experiment, runs a protocol and has a list of experimental setups (you can also call it arenas), where one or more subjects can be placed to run execute the selected protocol. From each setup you should associate a corresponding Bpod board that will be responsibile for running your protocol.

The workflow goes like this:

1. Inside the project, add an experiment.

2. Assign a protocol to the experiment.

3. Inside the experiment, add several setups.

4. Assign a Bpod board to each of the setups.

5. Assign one or more subjects to each setup.

6. Run the experiment for one setup or run them all at the same time!

Projects

- ▼ 🟩 Test project
  - ▼ 🖼 Experiments
    - ▼ ⚗ Light chansing exp...
      - ▼ 🏛 Setup A1
        - 20171122-131...
      - 🏛 Setup B1
    - ▼ ⚗ Trial events experi...
      - 🏛 Setup A2
  - ▼ 🐀 Subjects
    - 🐭 Subject A
    - 🐭 Subject B
  - ▼ 🔌 Bpod boards
    - 🖴 Board A
    - 🖴 Board B
  - ▼ 📷 Protocols
    - 📄 global_timer_exam...
    - 📄 light_chasing

Details

Exp. name Light chansing experiment

Protocol light_chasing ▾

Run all
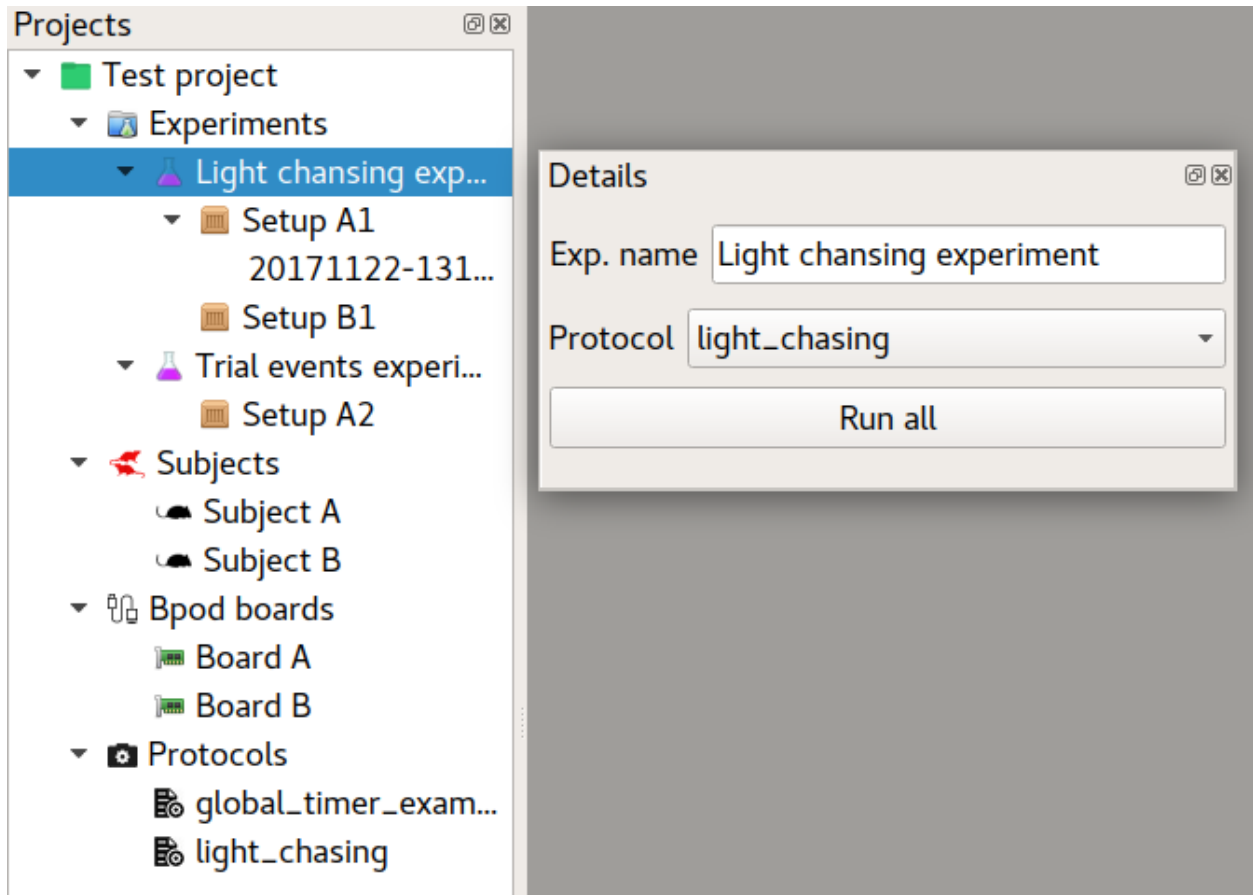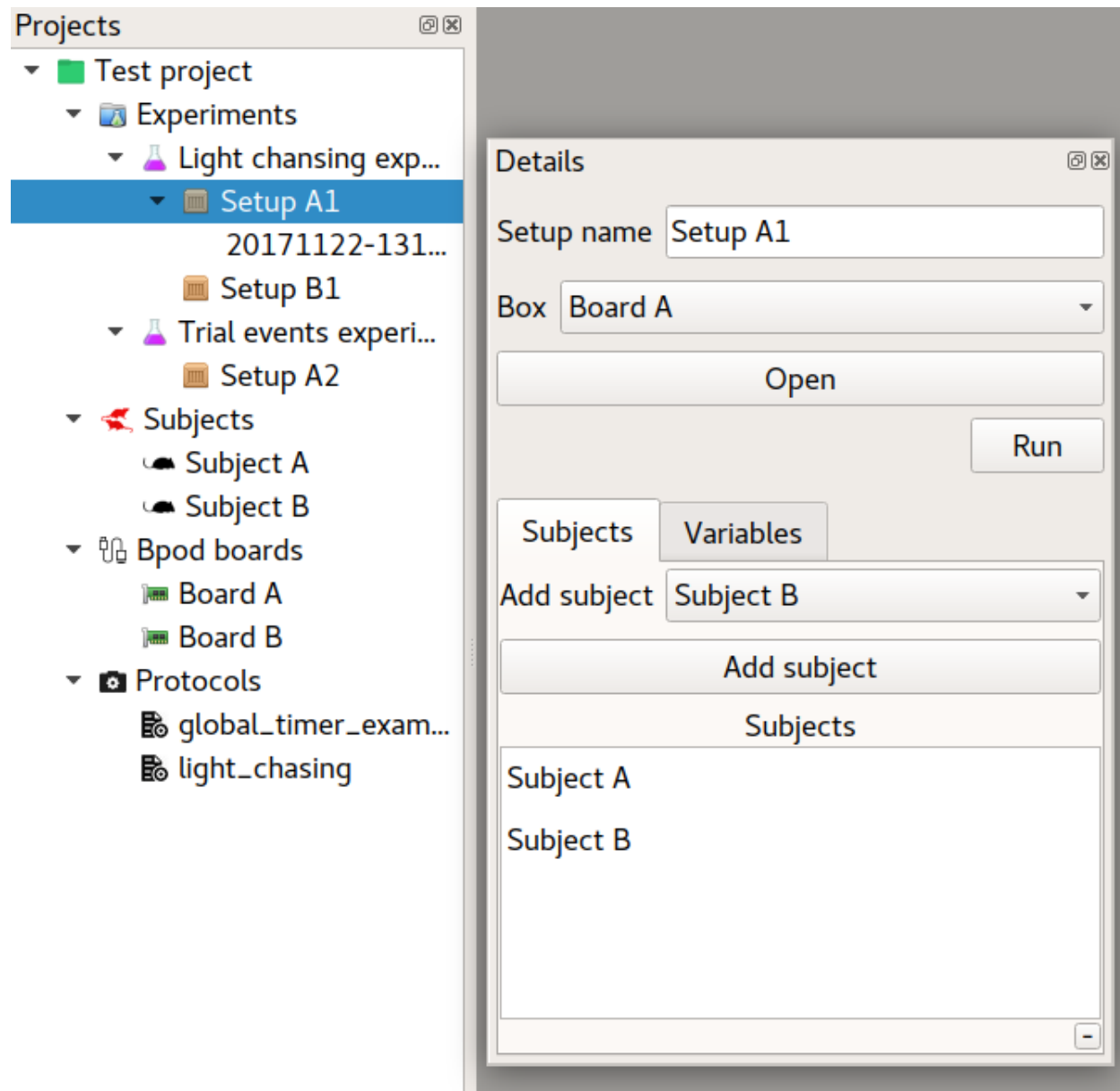
3.2.6 Sessions

Each time you run a Bpod protocol on a setup a new session is created. The GUI collects the output from the PyBpod API and processes these events on a list (which we call session history). Besides being on memory, this history is automatically saved on a text file, so you never lose Bpod data.

If you navigate to your project on the filesystem, and locate the desired setp, you should find several files:

- CSV and JSON are default outputs from the pybpod-api (for example, you can open CSV on excel and quickly produce some plots)

- Plain text file is the output from the GUI

```
.
├── boards
│   ├── Board A
│   │   └── board-settings.json
│   └── Board B
│       └── board-settings.json
├── experiments
│   ├── Light chansing experiment
│   │   ├── experiment-settings.json
│   │   └── setups
│   │       ├── Setup A1
│   │       │   ├── 20171122-143113.csv
│   │       │   └── setup-settings.json
│   │       └── Setup B1
│   │           └── setup-settings.json
│   └── Trial events experiment
│       ├── experiment-settings.json
│       └── setups
│           └── Setup A2
│               └── setup-settings.json
├── project-settings.json
├── subjects
│   ├── Subject A
│   │   └── subject-settings.json
│   └── Subject B
│       └── subject-settings.json
└── tasks
    ├── global_timer_example
    │   ├── global_timer_example.py
    │   └── __init__.py
    └── light_chasing
        ├── __init__.py
        └── light_chasing.py
```
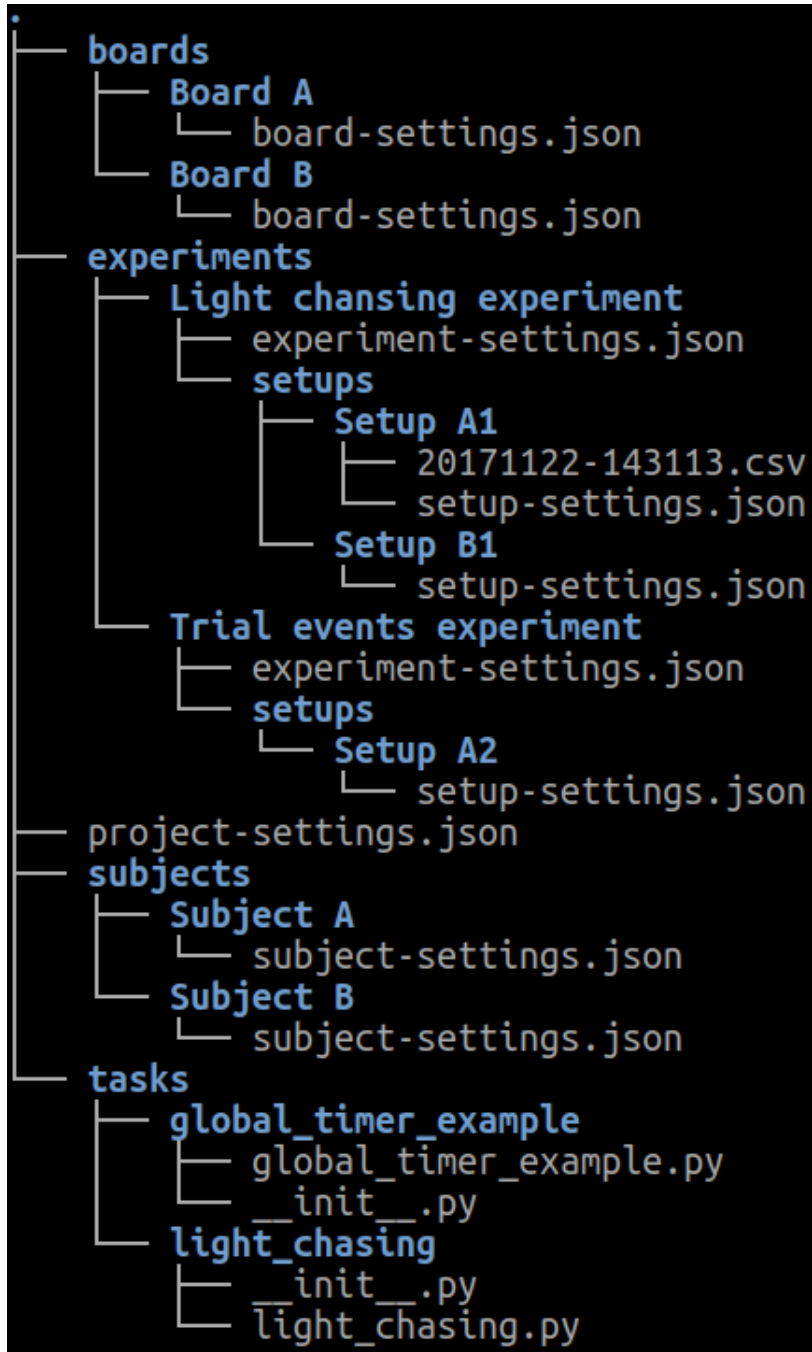
You can also develop plugins that enhance session data visualization and access them by right-clicking the desired
session.

3.2.7 GUI User settings

You can edit user settings directly from the GUI. User settings enable you to tweak the GUI the way you like it. Example of parameters you may change are:

- Loaded plugins

- Default project path

- Refresh time for console window

- And much more...
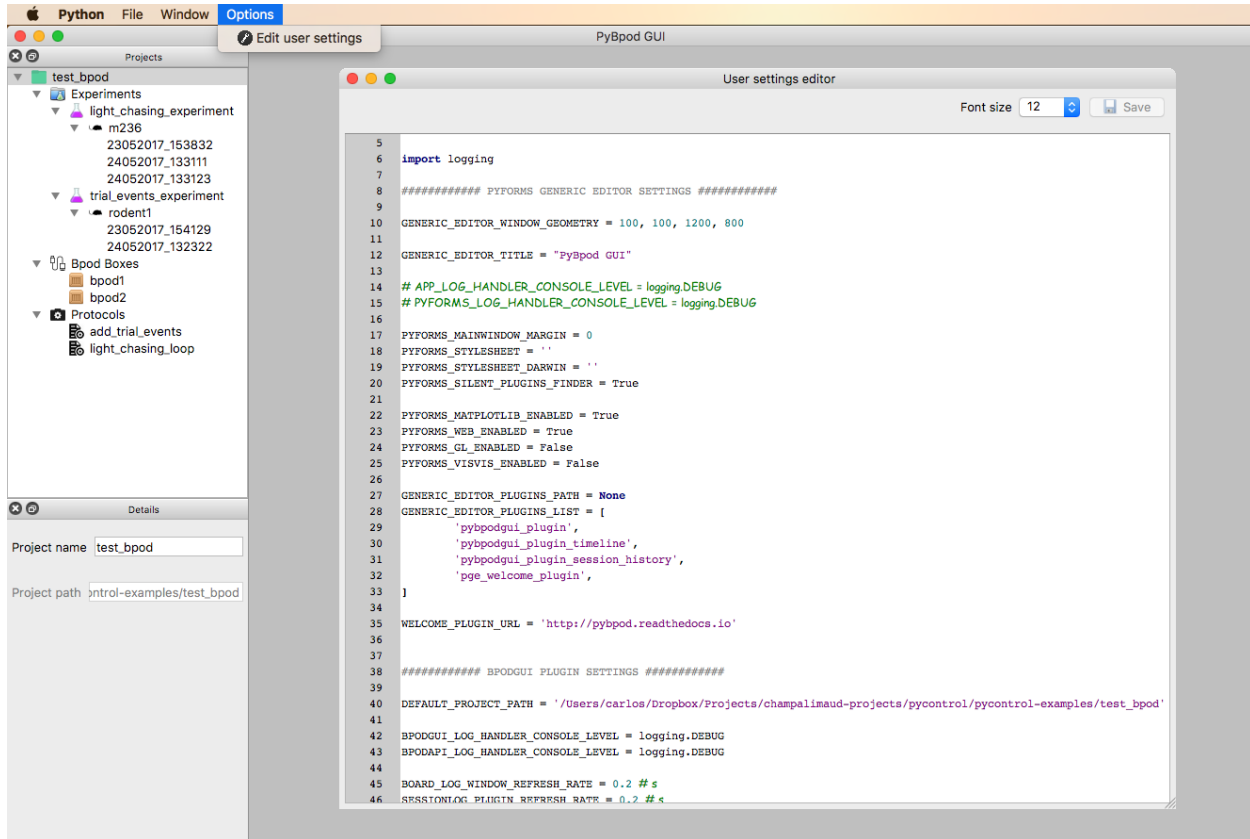
3.3 Writing a protocol for Bpod

3.3.1 What is a Bpod protocol?

To use Bpod, you must first program a behavioral protocol. The following guide is based on the original version for Bpod Matlab.

3.3.2 Protocol example explained

First, you will need to import Bpod modules.

```python
# Bpod main module and State machine module
from pybpodapi.protocol import Bpod, StateMachine
```

Then, initialize Bpod. The GUI will automatically set the serial port based on the serial port selected for the board and the workspace will be the subject folder.

You can run several trials for each Bpod execution. In this example, we will use 5 trials. Each trial can be of type1 (rewarded left) or type2 (rewarded right).

```python
nTrials = 5
trialTypes = [1, 2]  # 1 (rewarded left) or 2 (rewarded right)

for i in range(nTrials):  # Main loop
```

(continues on next page)

```
10          print('Trial: ', i+1)
11          thisTrialType = random.choice(trialTypes)  # Randomly choose trial type =
12          if thisTrialType == 1:
13              stimulus = 'PWM1'   # set stimulus channel for trial type 1
14              leftAction = 'Reward'
15              rightAction = 'Punish'
16              rewardValve = 1
17          elif thisTrialType == 2:
18              stimulus = 'PWM3'   # set stimulus channel for trial type 1
19              leftAction = 'Punish'
20              rightAction = 'Reward'
21              rewardValve = 3
```

Now, inside the loop, we will create and configure a state machine for each trial. A state machine has *state name*, *state timer*, *names of states to enter if certain events occur* and *output actions*. Please see State Machine API for detailed information about state machine design.

```
22          sma = StateMachine(my_bpod.hardware)
23
24          sma.add_state(
25              state_name='WaitForPort2Poke',
26              state_timer=1,
27              state_change_conditions={'Port2In': 'FlashStimulus'},
28              output_actions=[(OutputChannel.PWM2, 255)])
29          sma.add_state(
30              state_name='FlashStimulus',
31              state_timer=0.1,
32              state_change_conditions={'Tup': 'WaitForResponse'},
33              output_actions=[(stimulus, 255)])
34          sma.add_state(
35              state_name='WaitForResponse',
36              state_timer=1,
37              state_change_conditions={'Port1In': leftAction, 'Port3In': rightAction},
38              output_actions=[])
39          sma.add_state(
40              state_name='Reward',
41              state_timer=0.1,
42              state_change_conditions={'Tup': 'exit'},
43              output_actions=[('Valve', rewardValve)])  # Reward correct choice
44          sma.add_state(
45              state_name='Punish',
46              state_timer=3,
47              state_change_conditions={'Tup': 'exit'},
48              output_actions=[('LED', 1), ('LED', 2), ('LED', 3)])  # Signal incorrect␣
    ↪choice
```

After configuring the state machine, we send it to the Bpod device by calling the method *send_state_machine*. We are then ready to run the next trial, by calling the *run_state_machine* method. On run completion, we can print the data available for the current trial including events and states.

```
49          my_bpod.send_state_machine(sma)   # Send state machine description to Bpod␣
    ↪device
50
51          print("Waiting for poke. Reward: ", 'left' if thisTrialType == 1 else 'right')
52
53          my_bpod.run_state_machine(sma)   # Run state machine
```

```
54
55          print("Current trial info: ", my_bpod.session.current_trial())
```

Finally, after the loop finishes, we can stop Bpod execution.

```
56      my_bpod.close()   # Disconnect Bpod and perform post-run actions
```

See also:

PyBpod API

3.4 Plugins

PyBpod GUI can be enhanced with plugins. This way you can easily adapt the GUI for your needs.

You can use plugins for:

- extending or overwriting basic PyBpod functionalities
- creating new visualization tools for PyBpod sessions (e.g., plots, message filters)
- adding new windows, tools or any other GUI-related functionality

For detailed information on how to develop plugins please see *Developing PyBpod GUI plugins*.

3.4.1 How to install plugins

Installing plugins only takes 3 steps.

First, you will need to edit your user settings. On the top menu, go to **Options > Edit user settings**. Then, locate the following labels:

- GENERIC_EDITOR_PLUGINS_PATH -> this variable expects a string value which should correspond to a filesystem folder path where your plugins are located
- GENERIC_EDITOR_PLUGINS_LIST -> this variable expects a list of strings which are the names of the plugins to be loaded when the GUI starts up

> **Warning:** If you are using Windows OS, you must use double slash for paths. Example: GENERIC_EDITOR_PLUGINS_PATH = 'C:\\Users\\YOUR_NAME\\bpod_plugins'.

Second, download the plugin folder you want and place it on the "plugins" folder you have just indicated before.

Finally, restart the GUI. Depending on the kind of plugin, you will see a new option on the top menu or by right-clicking a node in the project tree.

Note: If you are developing plugins and you already installed them with PIP, you may leave the `GENERIC_EDITOR_PLUGINS_PATH = None` because they will be already on the Python path.
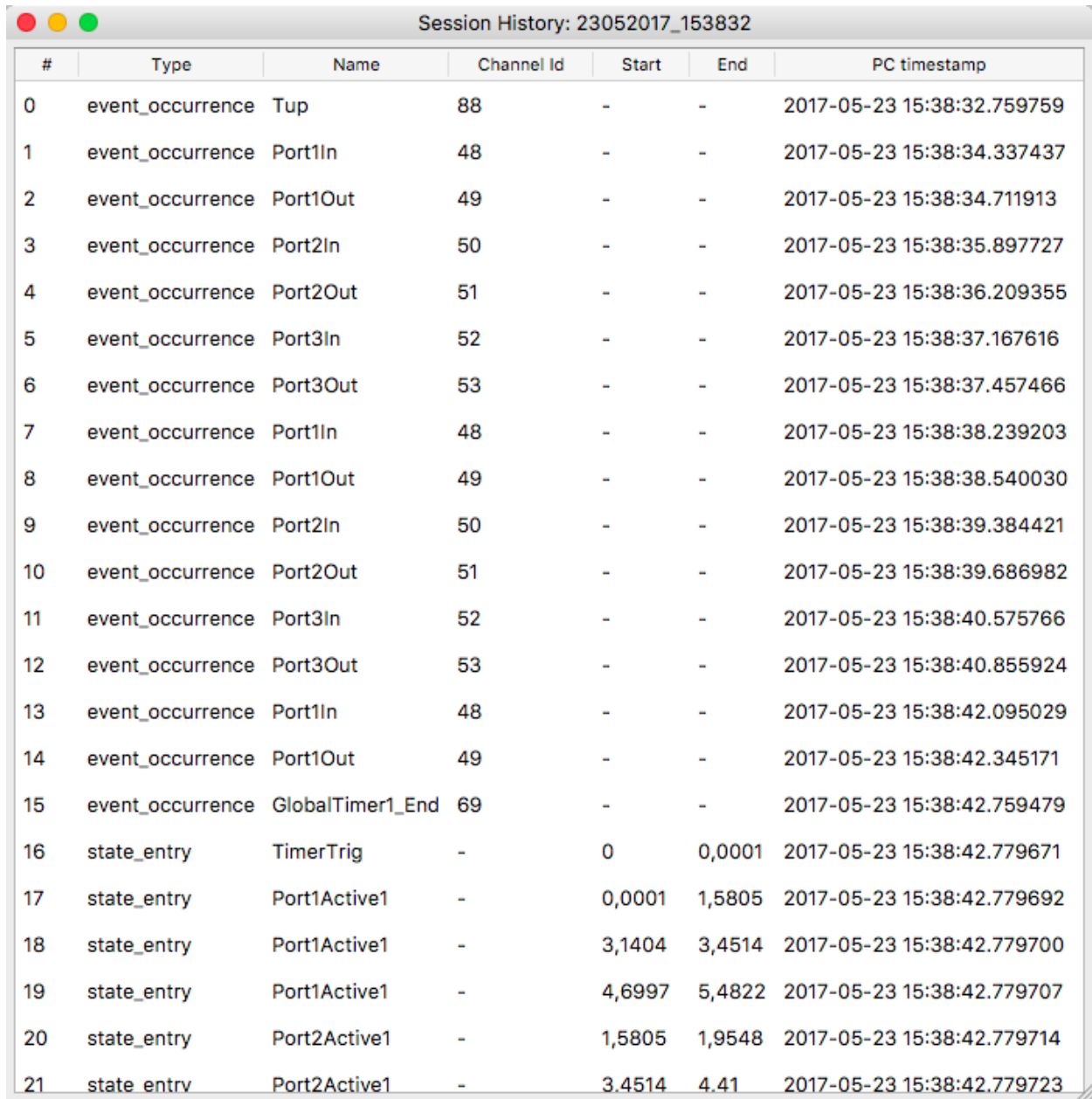
3.4.2 Examples of available plugins

Session history

This plugin allows you to display session data in a table view and you can order events by column.

https://bitbucket.org/fchampalimaud/pybpod-gui-plugin-session-history

#	Type	Name	Channel Id	Start	End	PC timestamp
0	event_occurrence	Tup	88	-	-	2017-05-23 15:38:32.759759
1	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:34.337437
2	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:34.711913
3	event_occurrence	Port2In	50	-	-	2017-05-23 15:38:35.897727
4	event_occurrence	Port2Out	51	-	-	2017-05-23 15:38:36.209355
5	event_occurrence	Port3In	52	-	-	2017-05-23 15:38:37.167616
6	event_occurrence	Port3Out	53	-	-	2017-05-23 15:38:37.457466
7	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:38.239203
8	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:38.540030
9	event_occurrence	Port2In	50	-	-	2017-05-23 15:38:39.384421
10	event_occurrence	Port2Out	51	-	-	2017-05-23 15:38:39.686982
11	event_occurrence	Port3In	52	-	-	2017-05-23 15:38:40.575766
12	event_occurrence	Port3Out	53	-	-	2017-05-23 15:38:40.855924
13	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:42.095029
14	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:42.345171
15	event_occurrence	GlobalTimer1_End	69	-	-	2017-05-23 15:38:42.759479
16	state_entry	TimerTrig	-	0	0,0001	2017-05-23 15:38:42.779671
17	state_entry	Port1Active1	-	0,0001	1,5805	2017-05-23 15:38:42.779692
18	state_entry	Port1Active1	-	3,1404	3,4514	2017-05-23 15:38:42.779700
19	state_entry	Port1Active1	-	4,6997	5,4822	2017-05-23 15:38:42.779707
20	state_entry	Port2Active1	-	1,5805	1,9548	2017-05-23 15:38:42.779714
21	state_entry	Port2Active1	-	3,4514	4,41	2017-05-23 15:38:42.779723

Session timeline

This plugin displays trial states in a bar plot.

https://bitbucket.org/fchampalimaud/pybpod-gui-plugin-timeline
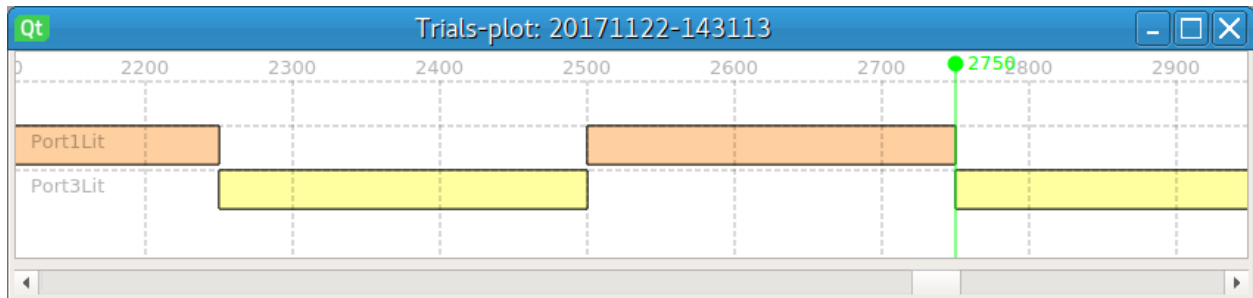
3.5 GUI explained

Note: Quick rationale

There are a lot of things going on under the hood when you run the **PyBpod GUI**. We will try to resume the basic concepts here but keep in mind that it is a lot of information so don't get frustrated if you don't get it at first. There were a lot of man-hours involved in this project and we strongly believe that dividing code in modules and libraries, though it seems more complicated at first, greatly improves code reusability and low-coupling.

3.5.1 Libraries that make up the GUI

Qt and PyQt

First, **PyBpod GUI** relies on PyQt, a set of Python v2 and v3 bindings for The Qt Company's Qt application framework which runs on all platforms supported by Qt including Windows, OS X, Linux, iOS and Android. Qt is more than a GUI toolkit. It includes abstractions of network sockets, threads, Unicode, regular expressions, SQL databases, SVG, OpenGL, XML, a fully functional web browser, a help system, a multimedia framework, as well as a rich collection of GUI widgets.

https://riverbankcomputing.com/software/pyqt/intro

PyForms

Because developing complex applications with PyQt can be hard to maintain, we rely on **Pyforms**, a Python 2.7.x and 3.x cross-enviroment framework for developing GUI applications, which promotes modular software design and code reusability with minimal effort.

https://github.com/UmSenhorQualquer/pyforms

PyformsGenericEditor

But we got even far, and since several GUI applications share the same concepts, we have developed **PyformsGenericEditor**, which allows for quickly bootstrapping a GUI application with file menus, project trees, form fields, etc. It also abstracts the concepts of creating, saving and deleting projects. This generic editor can then be adapted for specific use cases by providing plugins which define how projects are saved on filesystem, how to populate the project tree, which options show up in the menus and so on.

Thus **PyBpod GUI** is itself a plugin for the **PyformsGenericEditor**.
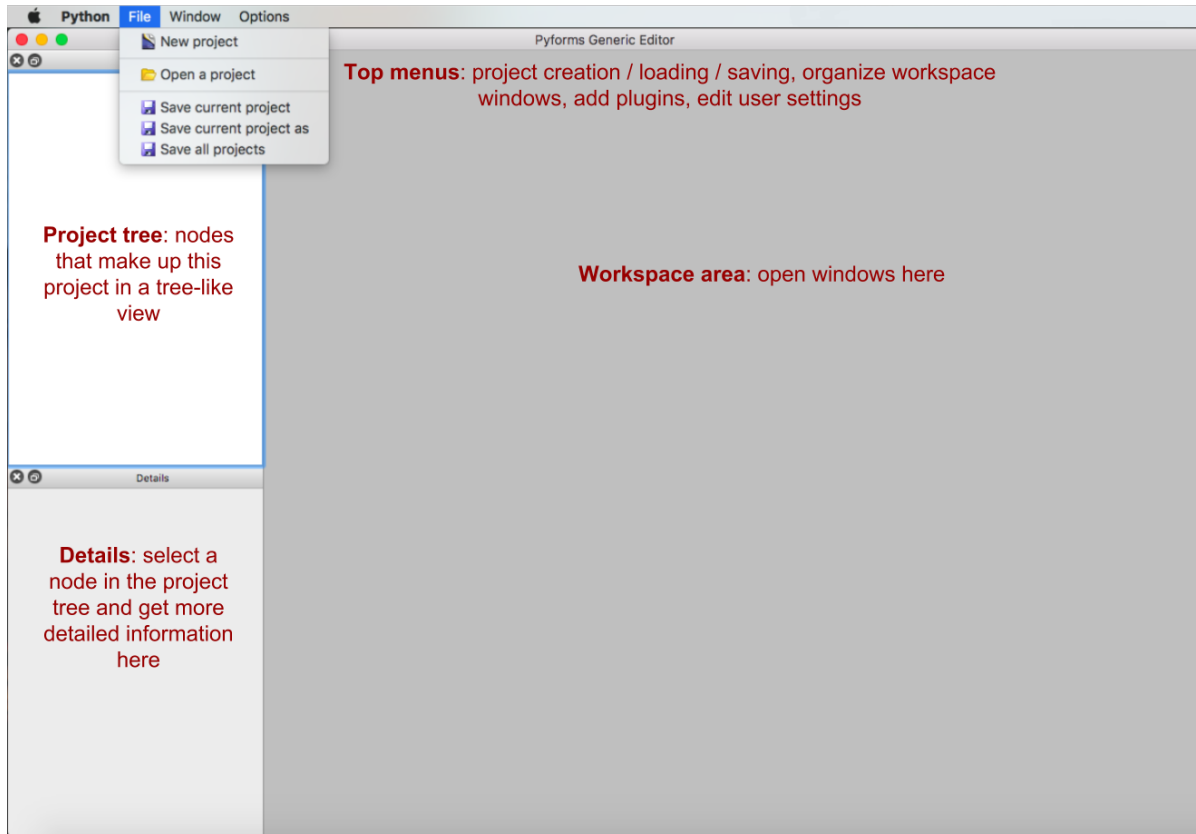
PyBpod API

Pybpod API is a Python library that enables communication with the latest Bpod device version. It is responsible for translating the protocols you write into a state matrix array and send it to the Bpod device.

http://pybpod-api.readthedocs.io/en/latest/

Confapp

Python library to provide settings files for modular applications.

https://github.com/UmSenhorQualquer/confapp

Pybranch

Library that offers multiprocessing communication.

https://bitbucket.org/fchampalimaud/pybranch

Logging-bootstrap

Library that provides simple methods for bootstrapping a logger with default settings.

https://bitbucket.org/fchampalimaud/logging-bootstrap

3.6 GUI Windows

Explaining how GUI windows work, how they relate to PyformsGenericEditor and Pyforms, how the model is organized (inheritance).

> **Warning:** To be implemented soon

3.7 Bpod interaction

On this page you will learn:

- How multiprocessing works

- How GUI handles protocols and communicates with Bpod

- How messages from bpod since "_publish_data" go to the multiprocessing queue until they are parsed on the factory

3.7.1 Multiprocessing

PyBpod GUI allows to control several Bpod boxes from a single application. To achieve parallel execution, we use Qt Threads to avoid interface from freezing and Python3 multiprocessing for getting the most out of computer CPU parallelization capabilities.

For each setup you run a protocol, a new Qt Thread and a process child will be created and get allocated to a single Bpod connection. This ensures the maximum performance. Moreover, if a Bpod box fails, the other Bpod boxes are not affected.

3.7.2 Starting protocol on Bpod

From the moment you press the button "run" on the GUI until output shows up in the console and it is saved on session history file, a lot of stuff is going on. First, let's see how the GUI handles the "run" button. SetupWindow class is responsible for painting the setup window, including input fields and buttons.

```python
class SetupWindow(Setup, BaseWidget):
    """
    Define here which fields from the setup model should appear on the details␣
↪section.

    The model fields shall be defined as UI components like text fields, buttons,␣
↪combo boxes, etc.

    You may also assign actions to these components.

    (...)

    """

    def __init__(self, experiment=None):
        BaseWidget.__init__(self, 'Experiment')
```

(continues on next page)

```python
        self._name = ControlText('Subject name')
        self._board = ControlCombo('Box')
        self._run_task_btn = ControlButton('Run')

        Setup.__init__(self, experiment)

        self.reload_boards()

        self._formset = [
            '_name',
            '_board',
            (' ', ' ', '_run_protocol_btn'),
            ' '
        ]

        (...)

        self._run_protocol_btn.value = self._run_protocol

    def _run_protocol(self):
        """
        Defines behavior of the button :attr:`SetupWindow._run_task_btn`.

        This methods is called every time the user presses the run button.
        """
        try:
            if self.status == SetupWindow.STATUS_RUNNING_PROTOCOL_HANDLER:
                self.stop_protocol()
            elif self.status == SetupWindow.STATUS_READY:
                self.run_protocol()
        except RunSetupError as err:
            QMessageBox.warning(self, "Warning", str(err))
        except Exception as err:
            QMessageBox.critical(self, "Unexpected Error", str(err))
```

The "run" button click event will fire a complex sequence of calls. Detail explanation of this process is out of scope of this tutorial. However, the following diagram resumes this process. What is important to retain is that the pybpod GUI library makes us of the Pybranch library to handle Qt Threads and python multiprocessing. Notice how inheritance is used for several classes to promote code reusability and separating concepts.

3.8 Developing plugins

3.8.1 What is a PyBpod GUI plugin?

PyBpod relies on a generic GUI framework, called **PyformsGenericEditor** which offers a basic user interface and can be extended to provide specific functionality through the installation of plugins.

You can use plugins for:

- extending or overwriting PyBpod core concepts (e.g., experiments, subjects, boxes);

- creating new visualization tools for PyBpod sessions (e.g., plots, message filters);
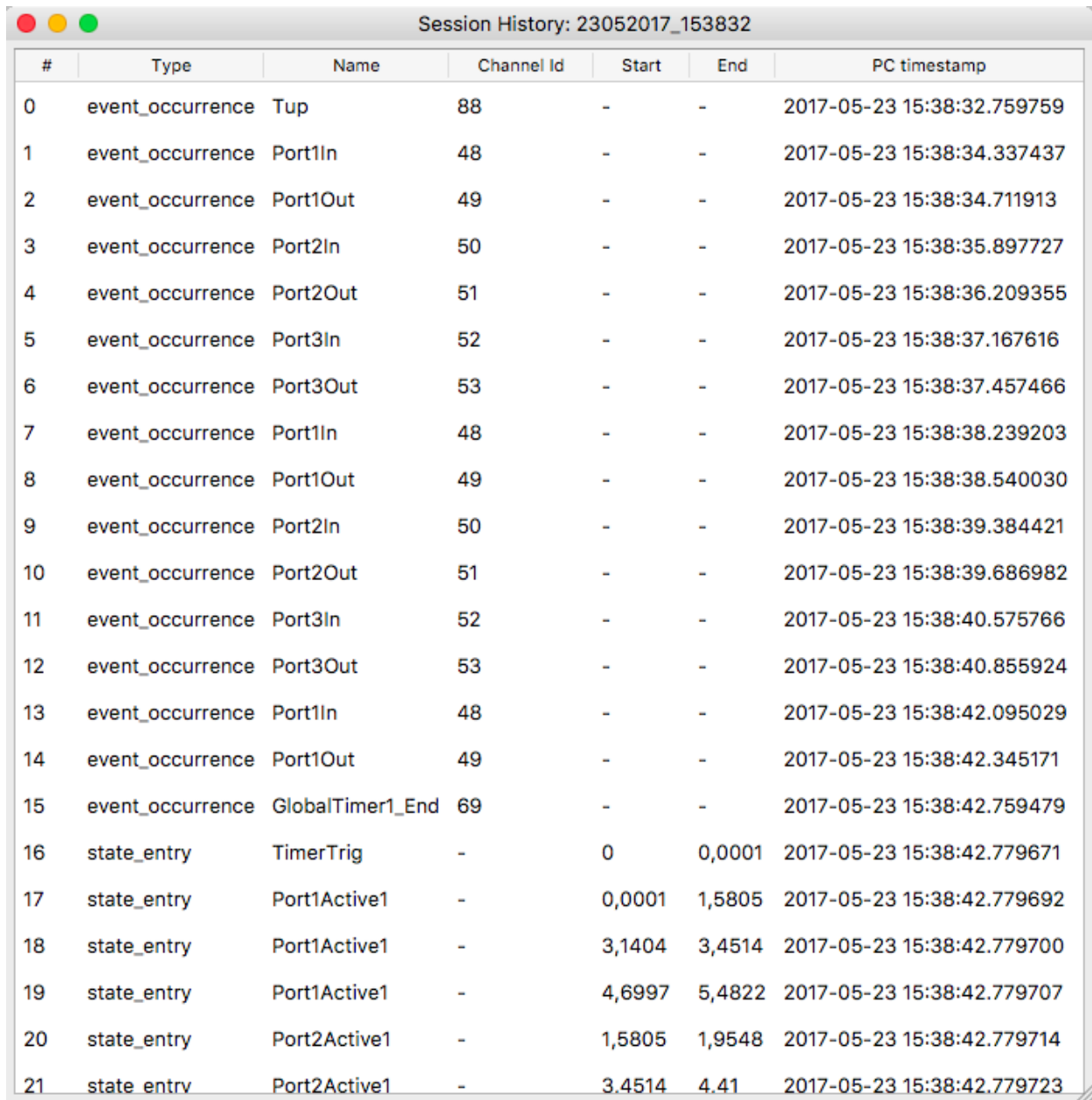
- adding new windows, tools or any other UI-related functionality;

Each plugin will be associated with a specific element of **PyformsGenericEditor** (e.g., project tree node, menu option, workspace area, etc).

3.8.2 Session history plugin, an example

This plugin allows you to display session data in a table view and you can order events by column.

https://bitbucket.org/fchampalimaud/pybpod-gui-plugin-session-history

#	Type	Name	Channel Id	Start	End	PC timestamp
0	event_occurrence	Tup	88	-	-	2017-05-23 15:38:32.759759
1	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:34.337437
2	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:34.711913
3	event_occurrence	Port2In	50	-	-	2017-05-23 15:38:35.897727
4	event_occurrence	Port2Out	51	-	-	2017-05-23 15:38:36.209355
5	event_occurrence	Port3In	52	-	-	2017-05-23 15:38:37.167616
6	event_occurrence	Port3Out	53	-	-	2017-05-23 15:38:37.457466
7	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:38.239203
8	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:38.540030
9	event_occurrence	Port2In	50	-	-	2017-05-23 15:38:39.384421
10	event_occurrence	Port2Out	51	-	-	2017-05-23 15:38:39.686982
11	event_occurrence	Port3In	52	-	-	2017-05-23 15:38:40.575766
12	event_occurrence	Port3Out	53	-	-	2017-05-23 15:38:40.855924
13	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:42.095029
14	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:42.345171
15	event_occurrence	GlobalTimer1_End	69	-	-	2017-05-23 15:38:42.759479
16	state_entry	TimerTrig	-	0	0,0001	2017-05-23 15:38:42.779671
17	state_entry	Port1Active1	-	0,0001	1,5805	2017-05-23 15:38:42.779692
18	state_entry	Port1Active1	-	3,1404	3,4514	2017-05-23 15:38:42.779700
19	state_entry	Port1Active1	-	4,6997	5,4822	2017-05-23 15:38:42.779707
20	state_entry	Port2Active1	-	1,5805	1,9548	2017-05-23 15:38:42.779714
21	state_entry	Port2Active1	-	3,4514	4,41	2017-05-23 15:38:42.779723

Quick review on sessions

Each time you run a Bpod protocol on a subject, a new session is created. The GUI collects output from the PyBpod API and processes these events on a list (which we call session history). Besides being on memory, this history is automatically saved on a text file, so you never loose Bpod data.

If you navigate to your project on the filesystem, and locate the desired subject, you should find several files:

- CSV and JSON are default outputs from the pybpod-api (for example, you can open CSV on excel and quickly produce some plots)

- Plain text file is the output from the GUI
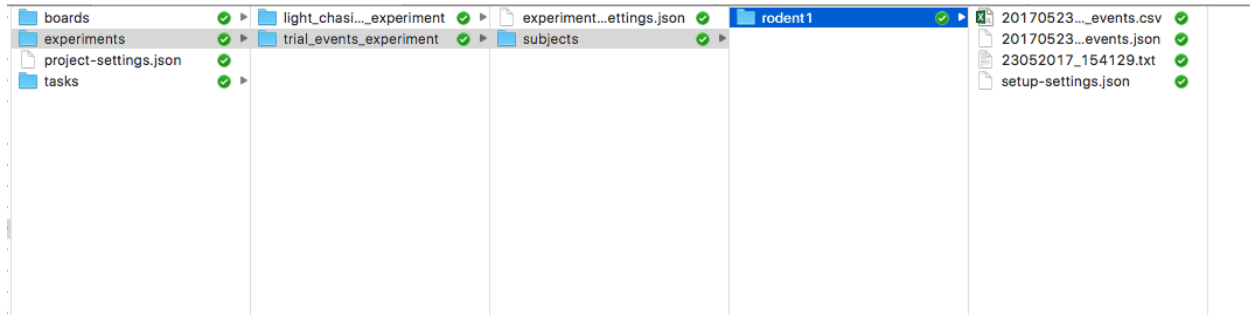
Let's take a look at a plain text file which was output from running a protocol on the GUI.

```
print_statement, 2017-05-23T15:41:29.638353, Trial:
print_statement, 2017-05-23T15:41:29.654188, Waiting for poke. Reward:
event_occurrence, 2017-05-23T15:41:33.672094, 50, Port2In, 2017-05-23 15:41:33.672094
event_occurrence, 2017-05-23T15:41:33.771925, 88, Tup, 2017-05-23 15:41:33.771925
state_entry, 2017-05-23T15:41:41.324848, 3, WaitForResponse, 4.1312, 4.3405
state_entry, 2017-05-23T15:41:41.324861, 4, Punish, 4.3405, 11.6663
state_entry, 2017-05-23T15:41:41.324908, 5, Reward, nan, nan
state_change, 2017-05-23T15:41:41.324930, 1, Port2In, 4.0312
state_change, 2017-05-23T15:41:41.324939, 2, Tup, 4.1312
state_change, 2017-05-23T15:41:41.324947, 2, Tup, 11.6663
print_statement, 2017-05-23T15:41:42.317543, Current trial info: {'Bpod start␣
→timestamp': 0.011, 'States timestamps': {'WaitForPort2Poke': [(0, 4.0312)],
→'FlashStimulus': [(4.0312, 4.1312)], 'WaitForResponse': [(4.1312, 4.3405)], 'Punish
→': [(4.3405, 11.6663)], 'Reward': [(nan, nan)]}, 'Events timestamps': {'Port2In':␣
→[4.0312], 'Tup': [4.1312, 11.6663], 'Port2Out': [4.3405], 'Port3In': [8.6663],
→'Port3Out': [8.8762]}}
print_statement, 2017-05-23T15:41:42.322411, Trial:
print_statement, 2017-05-23T15:41:42.325805, Waiting for poke. Reward:
event_occurrence, 2017-05-23T15:41:48.035732, 48, Port1In, 2017-05-23 15:41:48.035732
event_occurrence, 2017-05-23T15:41:48.136440, 88, Tup, 2017-05-23 15:41:48.136440
state_entry, 2017-05-23T15:41:48.160769, 3, WaitForResponse, 3.2538, 3.4102
state_entry, 2017-05-23T15:41:48.160775, 4, Reward, 3.4102, 5.8133
state_entry, 2017-05-23T15:41:48.160781, 5, Punish, nan, nan
state_change, 2017-05-23T15:41:48.160791, 1, Port2In, 3.1538
state_change, 2017-05-23T15:41:48.160804, 3, Port2Out, 3.4102
state_change, 2017-05-23T15:41:48.160808, 4, Port1In, 5.7133
print_statement, 2017-05-23T15:41:49.142529, Current trial info: {'Bpod start␣
→timestamp': 12.689, 'States timestamps': {'WaitForPort2Poke': [(0, 3.1538)],
→'FlashStimulus': [(3.1538, 3.2538)], 'WaitForResponse': [(3.2538, 3.4102)], 'Reward
→': [(3.4102, 5.8133)], 'Punish': [(nan, nan)]}, 'Events timestamps': {'Port2In': [3.
→1538], 'Tup': [3.2538, 5.8133], 'Port2Out': [3.4102], 'Port1In': [5.7133]}}
print_statement, 2017-05-23T15:41:49.147563, Trial:
```

(continues on next page)

```
print_statement, 2017-05-23T15:41:49.151724, Waiting for poke. Reward:
event_occurrence, 2017-05-23T15:41:52.731798, 50, Port2In, 2017-05-23 15:41:52.731798
event_occurrence, 2017-05-23T15:41:53.845332, 48, Port1In, 2017-05-23 15:41:53.845332
event_occurrence, 2017-05-23T15:41:53.946396, 88, Tup, 2017-05-23 15:41:53.946396
state_entry, 2017-05-23T15:41:53.974354, 1, WaitForPort2Poke, 0, 3.5869
state_entry, 2017-05-23T15:41:53.974475, 5, Punish, nan, nan
state_change, 2017-05-23T15:41:53.974495, 1, Port2In, 3.5869
state_change, 2017-05-23T15:41:53.974536, 3, Port2Out, 3.8881
state_change, 2017-05-23T15:41:53.974545, 4, Port1In, 4.7007
print_statement, 2017-05-23T15:41:54.955371, Current trial info: {'Bpod start␣
→timestamp': 19.513, 'States timestamps': {'WaitForPort2Poke': [(0, 3.5869)],
→'FlashStimulus': [(3.5869, 3.6869)], 'WaitForResponse': [(3.6869, 3.8881)], 'Reward
→': [(3.8881, 4.8007)], 'Punish': [(nan, nan)]}, 'Events timestamps': {'Port2In': [3.
→5869], 'Tup': [3.6869, 4.8007], 'Port2Out': [3.8881], 'Port1In': [4.7007]}}
```

What is going on here? Each line is a new message, where the first column identifies the type of an event on the session history: it can be a bpod state change, state entry, a user print, etc. These events represent messages that were sent from the Bpod and processed by the GUI.

Parsing board messages

Currently, PyBpod GUI supports the following events from Bpod board:

Session History Event Type	Occurs during trial run?	Description
Event occurrence	YES	Any Bpod event during trial run
State change	NO	Events detected by Bpod's inputs can be set to trigger transitions between specific states.
State entry	NO	State entered during the state matrix run
Print statement	YES	User defined print messages on protocol

All these classes represent board messages and inherit a generic class BoardMessage. For more information on how the GUI parses these messages, see Message factory.

Register plugin on the GUI

The first thing you need to do is to register your plugin. For that, edit your user settings. From the top menu, go to **Options > Edit user settings**. Edit the GENERIC_EDITOR_PLUGINS_PATH variable as this:

```
    GENERIC_EDITOR_PLUGINS_LIST = [
    'pybpodgui_plugin',
    'pybpodgui_plugin_session_history',
]
```

For the GUI to be able to detect the plugin source code you have 2 options:

1. Download the plugin folder you want and place it on the "plugins" folder you have just indicated before (useful when you run pybpod GUI as an executable)

2. Install the plugin with PIP (only applies if you are running the GUI from source code).

On this example, we will assume option #2 since we will be developing a plugin from the source code. In that case, you may leave the GENERIC_EDITOR_PLUGINS_PATH = None because the plugin will be already on the Python

path. **But don't forget! Every time you make changes to the plugin you have to install it with PIP again (unless your IDE does that for you).**

Finally, restart the GUI. The Session History plugin is a type of plugin that will be connected to a session and extend its behavior. Thus, after installing this plugin, you will see a new option by right-clicking a session node in the project tree. But how this works?

Connecting the plugin with a session node

Every node on the project tree node has a window assigned to it. In order to plugins show up on a project tree node, we need to extend the corresponding node window behavior. For example:

- an experiment node is connected to the `pybpodgui_api.models.experiment.experiment_treenode.ExperimentTreeNode` class
- a board node is connected to the `pybpodgui_api.models.board.board_treenode.BoardTreeNode` class
- a session node is connected to the `pybpodgui_api.models.session.session_treenode.SessionTreeNode` class

The **PyformsGenericEditor** enables that all these classes may be extended by looking for classes on plugins that have the same name and path.

On the Session History plugin, since we want to override session behavior we need to have the following structrure:
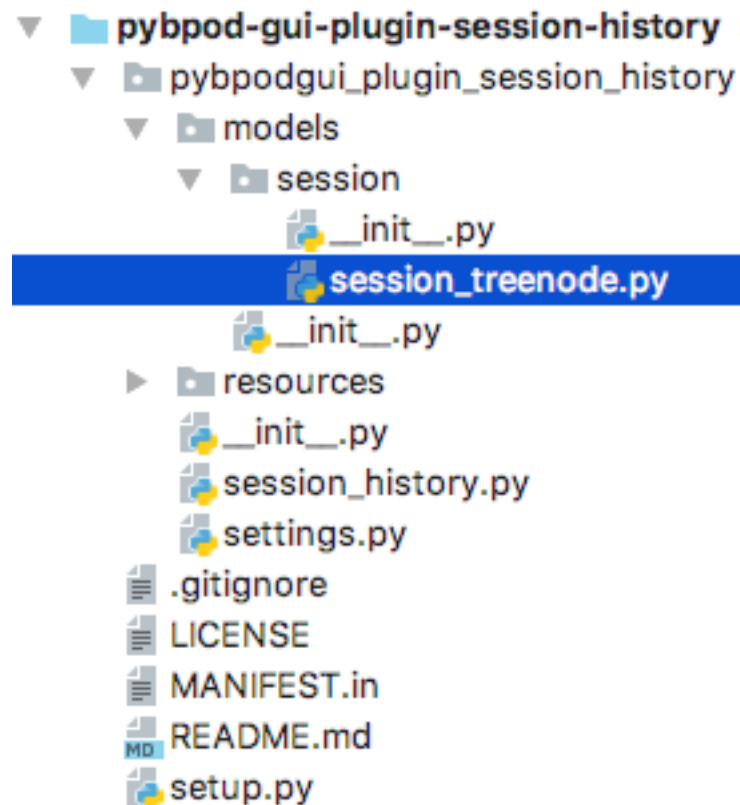
On the *models.session.__init__.py* module, you must define the class that will override the original SessionTreeNode class. If you inspect the *__init__.py* you will find this:

```
from pybpodgui_plugin_session_history.models.session.session_treenode import␣
→SessionTreeNode as Session
```

By using Python inheritance, **PyformsGenericEditor** discovers that *SessionTreeNode* will match the original class from the GUI.

On the *session_treenode.py* file on our plugin, one can now redefine the behavior of the desired methods. In this case, we are overriding the *create_treenode* method to add a new option when the user right-clicks the project tree node. We also override other methods to personalize details such as window title or double-clicking.

```python
(...)

from pybpodgui_plugin_session_history.session_history import SessionHistory

(...)

class SessionTreeNode(object):
    def create_treenode(self, tree):
        """
        Extends create_treenode behavior by calling the parent and adding a new option
        when user right-clicks the node.

        See also: pybpodgui_api.models.session.session_treenode.SessionTreeNode.
→create_treenode

        """
        node = super(SessionTreeNode, self).create_treenode(tree)

        tree.add_popup_menu_option('History', self.open_session_history_plugin,␣
→item=self.node,
                                   icon=QIcon(conf.SESSIONLOG_PLUGIN_ICON))

        return node

    def node_double_clicked_event(self):
        super(SessionTreeNode, self).node_double_clicked_event()
        self.open_session_history_plugin()

    def open_session_history_plugin(self):
        if not hasattr(self, 'session_history_plugin'):
            self.session_history_plugin = SessionHistory(self)
            self.session_history_plugin.show()
            self.session_history_plugin.subwindow.resize(*conf.SESSIONLOG_PLUGIN_
→WINDOW_SIZE)
        else:
            self.session_history_plugin.show()

    def remove(self):
        if hasattr(self, 'session_history_plugin'): self.mainwindow.mdi_area -= self.
→session_history_plugin
        super(SessionTreeNode, self).remove()

    @property
    def name(self):
        return super(SessionTreeNode, self.__class__).name.fget(self)

    @name.setter
```

(continues on next page)

```
    def name(self, value):
        super(SessionTreeNode, self.__class__).name.fset(self, value)
        if hasattr(self, 'session_history_plugin'): self.session_history_plugin.title
↪= value
```

This should be the final result:
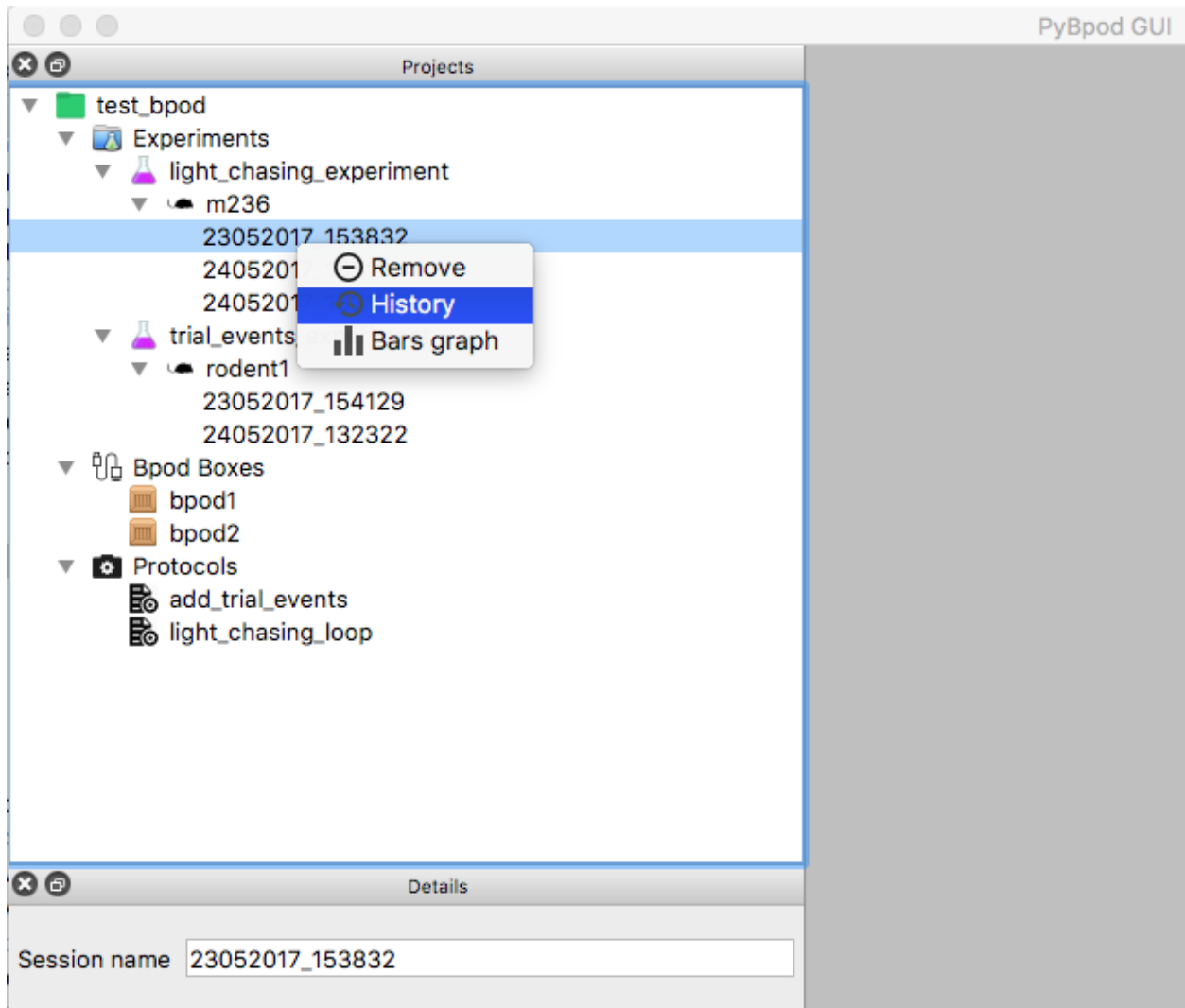
Handling session history from the plugin

On the previous section, we defined a new action for the session node. We have done that by linking the "History" menu option to the method *open_session_history_plugin*. Inside this method we invoke a class from the *session_history.py* module.

The *session_history.py* is responsible for creating a new window that shows up in the GUI workspace. This window must inherit from *BaseWidget* in order to make use of the necessary PyForms controls.

Since the GUI holds session history on memory, a list of board messages, session plugins can easily access to this list and process the events as needed. In our window, we will define a ControlList to list all the session history events. We will then define a timer that fires periodically to check for new messages and update the list.

```
(...)

from pyforms.basewidget import BaseWidget
from pyforms.controls import ControlProgress
from pyforms.controls import ControlList

(...)

from pybpodgui_plugin.com.messaging import ErrorMessage
from pybpodgui_plugin.com.messaging import PrintStatement
from pybpodgui_plugin.com.messaging import StateChange
from pybpodgui_plugin.com.messaging import StateEntry
from pybpodgui_plugin.com.messaging import EventOccurrence

(...)

class SessionHistory(BaseWidget):
    """ Plugin main window """

    def __init__(self, session):
        (...)

        self._log = ControlList()

        self._formset = [
            '_log',
        ]

        self._history_index = 0
        self._log.readonly = True
        self._log.horizontal_headers = ['#', 'Type', 'Name', 'Channel Id', 'Start',
→'End', 'PC timestamp']
        self._log.set_sorting_enabled(True)

        (...)

        self._timer = QTimer()
        self._timer.timeout.connect(self.read_message_queue)

    (...)

    def read_message_queue(self, update_gui=False):
        """ Update board queue and retrieve most recent messages """
        messages_history = self.session.messages_history
        recent_history = messages_history[self._history_index:]

        if update_gui:
            self._progress.show()
            self._progress.value = 0
        try:
            for message in recent_history:

                table_line = None
                if issubclass(type(message), StateChange):
                    table_line = (self._history_index, message.MESSAGE_TYPE_ALIAS,
→message.event_name,
                                  '-', message.board_timestamp, message.board_
→timestamp, str(message.pc_timestamp))
```

```python
            if issubclass(type(message), StateEntry):
                table_line = (self._history_index, message.MESSAGE_TYPE_ALIAS,
→message.state_name,
                              '-', message.start_timestamp, message.end_timestamp,
→ str(message.pc_timestamp))

            if issubclass(type(message), EventOccurrence):
                table_line = (self._history_index, message.MESSAGE_TYPE_ALIAS,
→message.event_name,
                              message.event_id, '-', '-', str(message.pc_
→timestamp))

            if table_line:
                self._log += table_line
                QEventLoop()

                if update_gui:
                    self._progress += 1
                    if self._progress.value >= 99: self._progress.value = 0

            self._history_index += 1

    except Exception as err:
        if hasattr(self, '_timer'):
            self._timer.stop()
        logger.error(str(err), exc_info=True)
        QMessageBox.critical(self, "Error",
                             "Unexpected error while loading session history.
→Pleas see log for more details.")

    if update_gui:
        self._progress.hide()

(...)
```

This should be the final result:

#	Type	Name	Channel Id	Start	End	PC timestamp
			Session History: 23052017_153832			
0	event_occurrence	Tup	88	-	-	2017-05-23 15:38:32.759759
1	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:34.337437
2	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:34.711913
3	event_occurrence	Port2In	50	-	-	2017-05-23 15:38:35.897727
4	event_occurrence	Port2Out	51	-	-	2017-05-23 15:38:36.209355
5	event_occurrence	Port3In	52	-	-	2017-05-23 15:38:37.167616
6	event_occurrence	Port3Out	53	-	-	2017-05-23 15:38:37.457466
7	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:38.239203
8	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:38.540030
9	event_occurrence	Port2In	50	-	-	2017-05-23 15:38:39.384421
10	event_occurrence	Port2Out	51	-	-	2017-05-23 15:38:39.686982
11	event_occurrence	Port3In	52	-	-	2017-05-23 15:38:40.575766
12	event_occurrence	Port3Out	53	-	-	2017-05-23 15:38:40.855924
13	event_occurrence	Port1In	48	-	-	2017-05-23 15:38:42.095029
14	event_occurrence	Port1Out	49	-	-	2017-05-23 15:38:42.345171
15	event_occurrence	GlobalTimer1_End	69	-	-	2017-05-23 15:38:42.759479
16	state_entry	TimerTrig	-	0	0,0001	2017-05-23 15:38:42.779671
17	state_entry	Port1Active1	-	0,0001	1,5805	2017-05-23 15:38:42.779692
18	state_entry	Port1Active1	-	3,1404	3,4514	2017-05-23 15:38:42.779700
19	state_entry	Port1Active1	-	4,6997	5,4822	2017-05-23 15:38:42.779707
20	state_entry	Port2Active1	-	1,5805	1,9548	2017-05-23 15:38:42.779714
21	state_entry	Port2Active1	-	3,4514	4,41	2017-05-23 15:38:42.779723

3.9 Contributing

This is an open source project and we welcome contributions from anyone interested.

You can contribute in the following ways:

- Work on the main bpod plugin and do a git merge request: https://bitbucket.org/fchampalimaud/pybpod-gui-plugin
- Fill in issues, suggestions, ask for new features, etc: https://bitbucket.org/fchampalimaud/pybpod-gui-plugin/issues
- *Developing plugins for Bpod*

3.10 Project Info

3.10.1 The SWP Team

Scientific Software Platform (Champalimaud Foundation)

The Scientific Software Platform (SWP) from the Champalimaud Foundation provides technical know-how in software engineering and high quality software support for the Neuroscience and Cancer research community at the Champalimaud Foundation.

We typical work on computer vision / tracking, behavioral experiments, image registration and database management.

3.10.2 Bpod project

PyBpod is a python port of the Bpod Matlab project.

All examples and Bpod's state machine and communication logic were based on the original version made available by Josh Sanders (Sanworks).

3.10.3 License

This is Open Source software. We use the GNU General Public License version 3.

3.10.4 Maintenance team

The current and past members of the **pybpod-api** team.

- @cajomferro Carlos Mão de Ferro
- @JBauto João Baúto
- @UmSenhorQualquer Ricardo Ribeiro
- @MicBoucinha Luís Teixeira

3.10.5 Questions?

If you have any questions or want to report a problem with this library please fill a issue here.

3.11 Changelog

3.11.1 v1.7.8 (2019/06/03)

- Fixed a problem with the setup requirements

3.11.2 v1.7.7 (2019/06/03)

- Fixed a problem with pybpod-gui-plugin-waveplayer.

3.11.3 v1.7.6 (2019/06/03)

- Requirements for PyBpod now point to specific package versions to ease upgrades
- **pybpod-api (updated to v1.6.4)**
    - Fixed problem with bad indexing when accessing modules in _bpodcom_module_write
- **pybpod-gui-plugin-waveplayer (v1.0)**
    - Corrected version number in the package and PyPi

3.11.4 v1.7.5 (2019/05/15)

- **pybpod-gui-plugin (updated to v1.6.2)**
    - Fixed png that was creating a warning on PyBpod initialization
    - Now it points correctly to the master branch
- **pyforms-gui (updated to v4.901.2)**
    - Version update so that PyPI considers a new version and the updates mentioned in v1.7.2 release are applied.
- **pybpod-alyx-module (updated to v1.1.1)**
    - Removed unnecessary requests package requirement
- **pybpod-gui-plugin-emulator (v0.1.3)**
    - Fixed override messages not being sent properly on Windows
    - Fix for pause not working
- **pybpod-rotaryencoder-module (v0.1.1)**
    - Fix for version override which would present always as version 0
- **pybpod-soundcard-module (v0.1.5)**
    - Added bumpversion support to this module

3.11.5 v1.7.4 (2019/05/08)

- The pybpodgui_plugin_session_history is now pointing to the master branch as it should (v1.4.1)

3.11.6 v1.7.3 (2019/05/08)

- Fixed problem with wrong pybpod-alyx-module version (now it is v1.1)

3.11.7 v1.7.2 (2019/05/03)

- Python base version changed to v3.6.6

- Conda environment files are now more coherent between Windows and Linux

- **pybpod-api (updated to v1.6.3)**

    - Data from interrupted trials are ignored

    - Added new trigger_input message to manually override inputs and trigger events

    - Fixed manual override of output channels

    - Fixed problem with GlobalTimers that were writing to the wrong indexes

    - Added new 'message' options to send serial messages to the modules connected to BPod's State Machine

- **pybpod-gui-api (updated to v1.2.2)**

    - Setups ran through a subject are now ran correctly

    - Added PYBPODGUI_API_AUTO_SAVE_PROJECT_ON_RUN option to user_settings

    - ScriptCmds are now executed as subprocesses

- **pybpod-gui-plugin (updated to v1.6.1)**

    - Fixed bug when subject were added to setups when canceling the confirmation dialog

    - Subject window now works properly and with the same options as within the setup (run, pause, detach from GUI option)

    - Fixed path problem in Pre and Post commands on Windows that prevented to run Pre and Post commands properly.

- **pyforms-gui (updated to v4.9.2)**

    - Code Editor now is presented properly on Windows

    - Normalized font labels size

- **pybpod-alyx-module (updated to v1.1)**

    - Import of Alyx subjects now allows to ignore all existing subject or replace all

    - Subjects that are dead, are now removed automatically from the list

- **New modules and plugins**

    - pybpod-soundcard-module (v0.1.4). More details on this module in: https://pybpod-soundcard-module.readthedocs.io/

    - pybpod-gui-plugin-emulator (v0.1). More details on this module in: https://pybpod-gui-plugin-emulator.readthedocs.io/en/v0.1.0/

3.12 Indices and tables

- genindex

- modindex

- search