psiexperiment

Psiexperiment Development Team

Sep 22, 2023

GETTING STARTED

Psiexperiment is a plugin-based framework for creating feature-rich auditory experiments with minimal effort. Psiexperiment can be run on any platform which supports Python and Qt or Pyside. The framework is robust enough to support a diverse range of closed and open-loop experiments. So far, the following experiments have been implemented:

- Auditory brainstem responses.
- Distortion product otoacoustic emissions (input-output functions, contralateral suppression, suppression using optogenetic stimuli).
- Envelope following responses.
- Operant behavior (go-nogo task) that can embed discrete sound tokens (i.e., targets and/or distractors) in a continuous background masker.
- Noise exposures of several hours in duration.

Note: If you're interested in the interleaved stimulus design for auditory brainstem responses (ABRs) described in Buran et al. (2019)[1], we have detailed information on how to use psiexperiment with the interleaved ABR program.

---

[1] Buran BN, Elkins S, Kempton JB, Porsov EV, Brigande JV, David SV. Optimizing Auditory Brainstem Response Acquisition Using Interleaved Frequencies. J Assoc Res Otolaryngol. 2020 Jun;21(3):225-242. doi: 10.1007/s10162-020-00754-3. Epub 2020 Jul 9. PMID: 32648066; PMCID: PMC7392976.

CHAPTER

ONE

WHAT YOU GET

1.1 A powerful, flexible user interface

We use a interface that offers dockable panes that can be "torn off", moved to other areas of the screen, minimized, maximized, or resized. This enables you to optimize the layout for your own workflow.

1.1.1 Simple hardware configuration

- You describe the devies and data acquisition channels available for your experiment in a *configuration file*. Psiexperiment will automatically configure the hardware based on the requirements of the experiment you're running.
- Since all stimulus generation and data acquisition is built on top of a hardware abstraction layer, your experiment can easily be shared with other labs who may have a different acquisition system. Provided psiexperiment has the appropriate interface for that system and the system supports the appropriate capabilities needed by the experiment (i.e, sampling rate, number of input and output channels, etc.), psiexperiment will be able to run the experiment on that system.
- Built-in caching and buffers to optimize stimulus generation and data acquisition.

1.2 Simple generation of auditory stimuli

- Psiexperiment has a set of robust auditory calibration utilities using tones, chirps and Golay sequences that can be used to calibrate both closed-field and free-field systems.
- An auditory stimulus generation system that incorporates calibration information and can generate complex stimuli that are either brief (i.e., clicks and tone pips for ABRs) or near-infinite (i.e., long noise exposures, continuous background masking during behavior, etc.) in duration.
- A powerful stimulus queue that allows you to queue stimuli for playout. Any type of stimulus can be added to the queue, allowing you to build complex trial structures (e.g., trains of tone pips with varying frequency and level; tone clouds; temporal orthogonal ripple noise combinations followed by a tone).
- Support for continuous generation of stimuli that are naturally infinite. For example, noise can be infinite in duration; however, the typical approach is to generate a short segment (e.g., 30 seconds) and repeat that segment. Instead, we support the generation of infinite stimuli without having to repeat a segment.
- Support for merging multiple stimuli into a single stream that can be played through a speaker. For example, this allows generation of continuous maskers during a behavioral experiment. The target stimuli are then embedded in this masker at the appropriate times.

1.3 Plugin-based system

- You can define actions based on what happens during an experiment. If the animal licks the spout, what should the program do?
- A native GUI with dockable components that reacts to user input (built on Atom and Enaml).
- Simple, intuitive experiment configuration that allows you to focus on experiment design (i.e., what you want the experiment to do) rather than implementation (how to write the code to make it work).
- *Easy configuration of experiment settings via the GUI*.
  - New settings can be added via a few lines of code
  - One or more settings can be marked for control as part of a sequence of values to be tested. For example, in many tests of peripehral auditory function multiple frequencies and levels will be tested.
  - In the GUI, values for settings can be expressed as equations. For example, if you have settings specifying the levels of two tones, `f1_level` and `f2_level`, you can fix the level of the second tone relative to the first by specifying its value as `f2_level = f1_level + 10`. Alternatively, you could randomly draw the level from a set of values on each trial as `f2_level = np.random.choice([30, 35, 40, 45])`.
- Large number of plugins for controlling the sequence of experiments, generating various stimuli, plotting results and saving data. Plugins can modify any part of psiexperiment (e.g., each plugin can contribute one or more dockable components to the GUI, contribute one or more new stimuli types, etc.).
- Easy to write new plugins using a declarative programming langauge with Python flavour.

CHAPTER

TWO

GETTING STARTED

2.1 Installing

2.1.1 Supported Hardware

Psiexperiment currently supports the following hardware:

- National Instruments (via the NIDAQmx drivers and pyDAQmx)
- TDT RZ6 (via the ActiveX drivers and tdtpy).
- Biosemi (via a custom port of the pyactivetwo library)

2.1.2 Using conda

If you use Anaconda or Miniconda, you can install psiexperiment from the psiexperiment channel. `ENV` is the name of the conda environment you wish to create to host psiexperiment. This is important because it allows you to have multiple versions of different Python packages (e.g., Numpy, Matplotlib, PyQtGraph, etc. without affecting psiexperiment).

```
conda create -n ENV -c psiexperiment psiexperiment
```

For example, if you want to call the environment *psi*, you would type:

```
conda create -n psi -c psiexperiment psiexperiment
```

If you plan to work on the source code, refer to the *instructions for developers*.

2.1.3 Dependencies

The code is written with Python >= 3.7 in mind. The actual list of requirements are lengthy, but you may only need a subset depending on the plugins you use. The core requirements are:

- enaml
- numpy
- palettable
- scipy
- pyqtgraph
- pandas

Plugin-specific requirements:

- **ZarrStore** or **BinaryStore** - zarr
- **BcolzStore** - bcolz
- **NIDAQEngine** - pydaqmx
- **TDTEngine** - tdtpy
- **BiosemiEngine** - pyactivetwo (customized port)

2.1.4 Configuring psiexperiment

First, create the required configuration file that contains information indicating where various files (logging, data, calibration, settings, temporary data, preferences, layout, IO and experiments) are saved. The configuration file defaults to `~/psi/config.py`. The `PSI_CONFIG` environment variable can be used to override the default path.

Open an Anaconda prompt. Be sure to activate the enviornment to which you installed psiexperiment, e.g., assuming that the enviornment was named `psi`:

```
conda activate psi
```

Standard hadware configurations

Let's create the configuration. Psiexperiment ships with several standard hardware configurations that are commonly available in research laboratories:

- **Medusa4ZTDT** - TDT RZ6 with a Medusa4Z connected.
- **RA4PATDT** - TDT RZ6 with a RA4PA connected.
- **PXIe-1032** - NI PXIe with a PXI-4461 card.
- **Biosemi32** - Biosemi with 32 channels (requires ActiView to be configured properly).
- **Biosemi64** - Biosemi with 64 channels (requires ActiView to be configured properly).

If one of these standard IO configurations matches your equipment, then the rest of the set-up process is relatively simple. In the command below, `PATH` is where you want the calibration, data and other configuration files should be stored. `IO` is the hardware configuration for your system (the bolded text from the list above).

```
psi-config create --base-directory PATH --io IO
```

For example, if you want to save data to `c:/data` and you have the TDT with the Medusa4Z:

```
psi-config create --base-directory c:/data --io Medusa4ZTDT
```

To view where the configuration file was saved:

```
psi-config show
```

Now, open that file in your preferred Python editor (Idle is fine as it's installed by default with Python) and update the variables to point to where you want the various files stored. By default, you can have all files created by psiexperiment saved under a single `BASE_DIRECTORY`. Alternatively, you may want to be more specific (e.g., log files go here, data goes there, etc.). Feel free to customize as needed.

- **LOG_ROOT**: Location where log files are stored. These files are used for debugging.

- **DATA_ROOT**: Location where data files are stored. These files are generated when running an experiment and contain all data acquired by the experiment.
- **CAL_ROOT**: Location where calibration data files are stored. These files are generated when running a calibration and are often required when running an experiment.
- **PREFERENCES_ROOT**: Location where experiment-specific preferences are stored.
- **LAYOUT_ROOT**: Location where experiment-specific layouts are stored.
- **IO_ROOT**: Location where system configuration is stored.
- **STANDARD_IO**: List of hardware configurations (see above). Usually there will be only one, but you may sometimes want to allow the user to select from several options (e.g., if you have both a RA4PA and Medusa4Z).

Once you have customized the configuration file, the folders can be created automatically if they don't already exist:

```
psi-config create-folders
```

Nonstandard hardware configurations

First, create a skeleton file for your hardware configuration. `SKELETON` is the name of the template you want to base the configuration on:

```
psi-config create-io SKELETON
```

Inside this file, you will describe the configuration of your system using Enaml syntax. This is known as the *input-output manifest*

2.2 Input-output manifest

2.2.1 Examples

Noise exposure

Basic configuration of a system with one output (connected to a speaker) and two inputs (from microphones) driven by a National Instruments DAQ card. This configuration is about as simple as it gets.

```
from enaml.workbench.api import PluginManifest, Extension

from psi.controller.engines.nidaq import (NIDAQEngine,
                                          NIDAQHardwareAIChannel,
                                          NIDAQHardwareAOChannel,
                                          NIDAQSoftwareDOChannel)


enamldef IOManifest(PluginManifest): manifest:
    '''
    Example of a simple configuration for noise exposure
    '''
    Extension:
        id = 'backend'
        point = 'psi.controller.io'
```

(continues on next page)

(continued from previous page)

```
    NIDAQEngine:
        # Each engine must have a unique name.
        name = 'NI'

        master_clock = True

        hw_ai_monitor_period = 0.1
        hw_ao_monitor_period = 1

        NIDAQHardwareAOChannel:
            # Label as shown in the GUI
            label = 'Noise exposure speaker'

            # Label as used in the code
            name = 'speaker'

            # Sampling rate the channel runs at. The engine may impose some
            # constraints on this sampling rate. For example, all analog
            # output channels configured on a particular NIDAQmx engine
            # must run at the same sampling rate.
            fs = 25e3

            # The data type required by the channel.
            dtype = 'float64'

            # This is a NIDAQmx-specific feature and is the channel
            # identifier used by the NIDAQmx library. Channels that
            # interface with other types of hardware will have their own
            # method for identifying channels.
            channel = 'Dev1/ao1'

            # Also a NIDAQmx-specific feature. This allows the NIDAQmx
            # library to optimize the channel configuration based on the
            # expected output range.
            expected_range = (-10, 10)

        NIDAQHardwareAIChannel:
            label = 'Experiment microphone'
            name = 'experiment_microphone'
            channel = 'Dev1/ai2'
            start_trigger = 'ao/StartTrigger'
            fs = 100e3
            expected_range = (-10, 10)
            dtype = 'float64'
            terminal_mode = 'differential'
            gain = 20

        NIDAQHardwareAIChannel:
            label = 'Calibration microphone'
            name = 'calibration_microphone'
            channel = 'Dev1/ai1'
```

(continues on next page)

(continued from previous page)

```
                start_trigger = 'ao/StartTrigger'
                fs = 100e3
                expected_range = (-10, 10)
                dtype = 'float64'
                terminal_mode = 'differential'
                gain = 0
```

Appetitive go-nogo behavior

Example configuration of a system designed for appetitive go-nogo behavior where the subject must nose-poke to start a trial and retrieve their reward from a food hopper. Both the nose-poke and food hopper have an infrared beam (photoemitter to photosensor) that generate an analog signal indicating the intensity of light falling on the photosensor. If the path between the photoemitter and photosensor is blocked (e.g., by the subject's nose), then the analog readout will reflect the change in light intensity. The analog readout of the photosensors are connected to the `nose_poke` and `reward_contact` channels.

For a go-nogo behavioral task, we need to convert this analog readout to a binary signal indicating whether the subject broke the infrared beam or not. In the following example we create a new processing chain, `AnalogToDigitalFilter` that performs this conversion and apply it to both the nose-poke and food hopper inputs.

```
from enaml.workbench.api import Extension, PluginManifest

from psi.controller.engines.nidaq import (NIDAQEngine,
                                          NIDAQHardwareAIChannel,
                                          NIDAQHardwareAOChannel,
                                          NIDAQSoftwareDOChannel)

from psi.controller.api import (CalibratedInput, Downsample, Edges, IIRFilter,
                                Threshold, Trigger, Toggle)


enamldef IRChannel(NIDAQHardwareAIChannel): irc:
    unit = 'V'
    start_trigger = 'ao/StartTrigger'
    fs = 100e3
    expected_range = (-10, 10)
    dtype = 'float64'
    terminal_mode = 'differential'

    IIRFilter: iir:
        name << irc.name + '_filtered'
        f_lowpass = 25
        ftype = 'butter'
        btype = 'lowpass'

        Downsample: ds:
            name << irc.name + '_analog'
            q = 1000
            Threshold: th:
                threshold = 2.5
                Edges: e:
```

(continues on next page)

(continued from previous page)

```
                    name << irc.name + '_digital'
                    debounce = 2



enamldef IOManifest(PluginManifest): manifest:
    '''
    This defines the hardware connections that are specific to the LBHB Bobcat
    computer for the appetitive experiment.
    '''

    Extension:
        id = 'backend'
        point = 'psi.controller.io'

        NIDAQEngine: engine:
            name = 'NI'
            master_clock = True

            # Since we're using an AnalogThreshold input to detect nose pokes
            # and reward contact, we want a fairly short AI monitor period to
            # ensure that we detect these events quickly.
            hw_ai_monitor_period = 0.025
            hw_ao_monitor_period = 1

            NIDAQHardwareAOChannel:
                label = 'Speaker'
                name = 'speaker'
                channel = 'Dev1/ao0'
                fs = 100e3
                expected_range = (-10, 10)
                dtype = 'float64'
                terminal_mode = 'RSE'
                calibration_user_editable = True

            NIDAQSoftwareDOChannel:
                name = 'food_dispense'
                channel = 'Dev1/port0/line0'

                Trigger:
                    # This is a required output for the food dispenser. The
                    # plugin will look for this output by name. If not present,
                    # the food dispenser plugin will not work!
                    label = 'Food dispense'
                    name = 'food_dispense_trigger'
                    duration = 0.1

            NIDAQSoftwareDOChannel:
                name = 'room_light'
                channel = 'Dev1/port0/line1'
```

(continues on next page)

(continued from previous page)

```
            Toggle:
                # This is a required output for the room light. The plugin
                # will look for this output by name. If not present, the
                # room light plugin will not work!
                name = 'room_light_toggle'
                label = 'Room light'

        IRChannel:
            label = 'Nose poke IR'
            name = 'nose_poke'
            channel = 'Dev1/ai0'

        IRChannel:
            label = 'Reward IR'
            name = 'reward_contact'
            channel = 'Dev1/ai1'
```

In the example code above, you'll note that we defined a `Toggle` output named `room_light_toggle`. If you look at the experiment manifest for appetitive experiments, you'll see that we've defined two actions that control this output:

```
ExperimentAction:
    event = 'to_start'
    command = 'room_light_toggle.off'

ExperimentAction:
    event = 'to_end'
    command = 'room_light_toggle.on'
```

The `to_start` and `to_end` events are generated by the appetitive controller when a timeout begins and ends. The rules above result in turning off the room light when a timeout beings and turning it back on when the timeout ends. By creating a rules-based action system, it simplifies the process of ensuring that a sequence of actions occur

In theory, we could configure the room light such that it was controlled by an Arduino if we had an Arduino backend implemented:

```
ArduinoEngine:
    DigitalOutput:
        name = 'room_light'
        channel = 'Pin1'
```

The engine is responsible for:

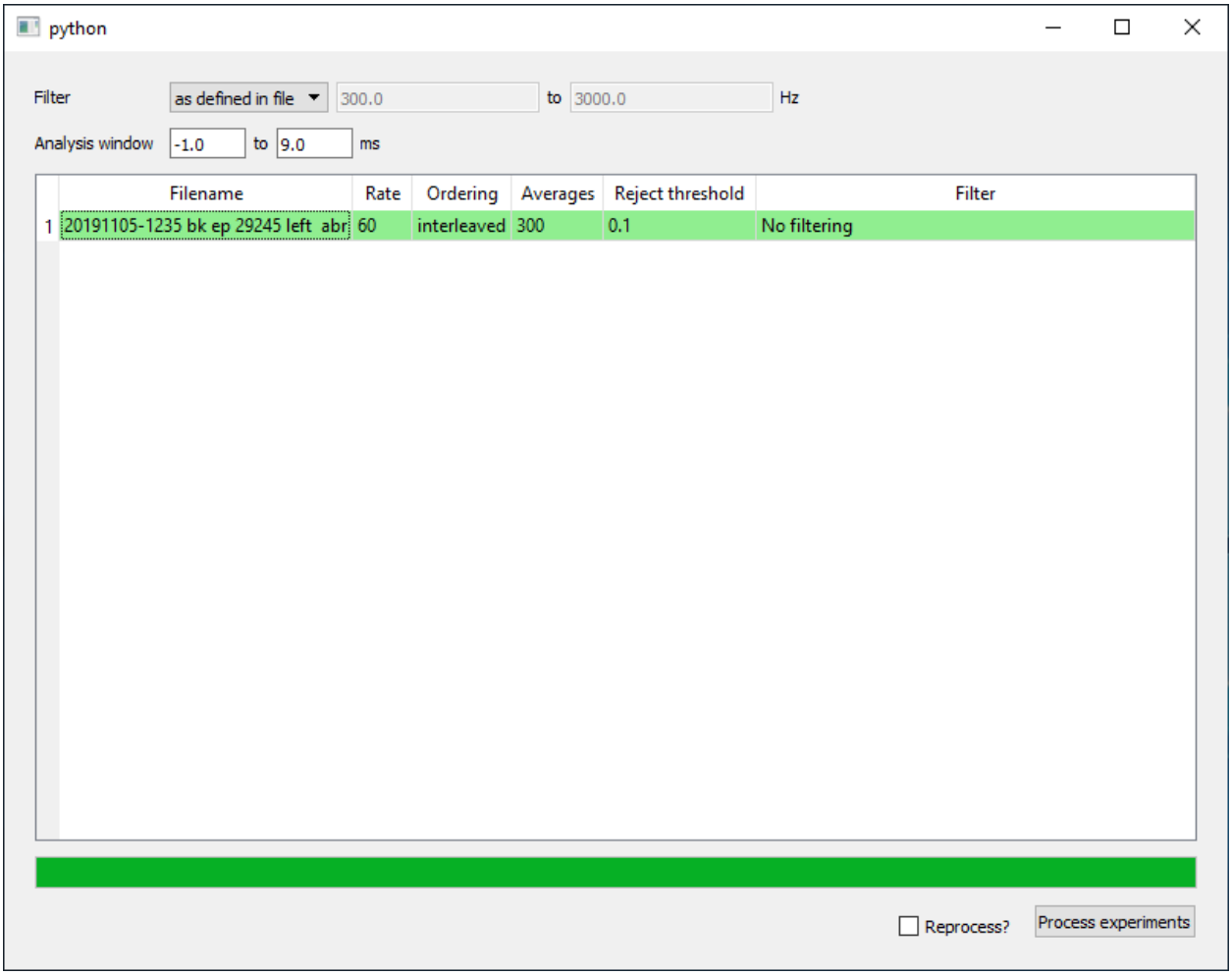
- Configuring the input and output channels.
- Responding to requests (e.g., uploading waveforms to analog output channels and toggling the state of digital output channels).
- Continuously pollign inptu channels and passing this through the input processing pipeline. All data acquisition is continuous (if you want epoch-based acquisition, you would add a special input, `ExtractEpochs`, to your input hierarchy).

You can have multiple engines in a single experiment.

2.3 Software

2.3.1 ABR post-processing script

Since psiexperiment saves single-trial, continuous data for ABRs, the data must be post-processed after acquisition for use with the ABR waveform analysis software. To run the program, select 'ABR post-acquisition processing' from the start menu (see XXX for command-line usage).



To load files, drag-and-drop the folder containing the ABR data onto the GUI. Unprocessed files are shown with a white background. If the file has already been processed, it will be shown with a green background. Once you have loaded the files you wish to process, you can specify the filter and epoch settings. By default, the filter settings specified during acquisition will be loaded from the file. If you wish to reprocess a file, you need to check the "reprocess files".

Be patient. It can take a while to proces each file. Once the files are done processing, they will be highlighted in green (if successful) or pink (if there was an error).

2.4 Installing

Instructions on installing psiexperiment and configuring your system.

2.5 Input-output manifest

Instructions on creating your input-output manifest which describes all equipment used for the experiment.

2.6 Software

Instructions for end-users.

CHAPTER

THREE

DEVELOPMENT

3.1 Development Workflow

3.1.1 Helpful resources

Psiexperiment leverages Enaml both for building the user-interface as well as implementing a plugin-based system. Extending psiexperiment requires familiarity with the Enaml Workbench plugin framework.

3.1.2 Setting up your environment

The following instructions assume that you use a conda-based distribution (e.g., Anaconda or Miniconda). This enables you to manage multiple versions of psiexperiment (e.g., a "production" version and a "development" version).

First, create an environment containing the dependencies required by psiexperiment. Substitute your preferred environment name (e.g., *psi-dev*) for *ENV*.

```
conda create -n ENV -c psiexperiment psiexperiment --only-deps
```

Now, install the psiexperiment source code. The *pip install* command explicitly installs dependencies for building documentation (e.g., *sphinx*) and testing (e.g., *pytest*):

```
mkdir %USERPROFILE%\projects\psi-dev\src
cd %USERPROFILE\projects\psi-dev\src
git clone https://github.com/bburan/psiexperiment
pip install -e ./psiexperiment[docs,test]
```

Create environment-specific environment variables and folders for your source code as needed. You can customize to your preferred workflow, but I find this particular workflow works well:

```
mkdir %USERPROFILE%\projects\psi-dev
conda env config vars set -n psi-dev PSI_CONFIG=%USERPROFILE%/projects/psi-dev/conf
```

Follow the instructions in the command prompt to reactivate your environment and load the environment variable you just set, i.e.,:

```
conda activate psi-dev
```

Make sure it worked properly:

```
psi-config show
```

Now create your config as described in the *install instructions*.

3.1.3 Building documentation

Documentation is built using Sphinx.

```
cd %USERPROFILE%\projects\psi-dev\src\psiexperiment\docs
make html
```

To update the API documentation:

```
cd %USERPROFILE%\projects\psi-dev\src\psiexperiment\docs
sphinx-apidoc ../psi -o source/api
```

It also automatically gets built on ReadTheDocs whenever new commits are pushed to the main branch on Github.

3.2 What makes an experiment?

An experiment is known as a *paradigm*. A paradigm is a collection of plug-in modules (some required, some optional) that interact with each other to run the experiment. The core paradigms that ship with psiexperiment can be found in `psi.paradigms.descriptions`. Psiexperiment ships with a collection of paradigms for operant behavior (go-nogo), auditory testing (ABR, DPOAE, MEMR, EFR), noise exposure, and basic calibration of acoustic systems.

3.2.1 What is a paradigm?

A paradigm is described using the `ParadigmDescription` class. Here's an example description:

```
from psi.experiment.api import ParadigmDescription
PATH = 'psi.paradigms.behavior.'
CORE_PATH = 'psi.paradigms.core.'

ParadigmDescription(
    'auto_gonogo', 'Auto GO-NOGO', 'animal', [
        {'manifest': PATH + 'behavior_auto_gonogo.BehaviorManifest'},
        {'manifest': PATH + 'behavior_mixins.BaseGoNogoMixin'},
        {'manifest': PATH + 'behavior_mixins.WaterBolusDispenser'},
        {'manifest': CORE_PATH + 'video_mixins.PSIVideo'},
        {'manifest': CORE_PATH + 'signal_mixins.SignalFFTViewManifest',
         'attrs': {'fft_time_span': 1, 'fft_freq_lb': 5, 'fft_freq_ub': 24000, 'y_label
↪': 'Level (dB)', 'source': 'microphone'}
         },
    ],
)
```

The first argument, `'auto_gonogo'` is a unique identifier for the paradigm. To launch this experiment, you would type `psi auto_gonogo` at the command line. The fourth argument is a list of manifests that can be loaded as part of the experiment. Some of them, such as the `BehaviorManifest`, are required. Others can be optionally loaded. The manifests are referenced by the module name plus class name. To override defaults on the class, a dictionary can be passed via `attrs`.

3.2.2 What is a manifest?

Briefly, a manifest defines how the associated plugin interacts with other plugins in the psiexperiment ecosystem. The manifest should subclass either `PSIManifest` or `ExperimentManifest`. For most custom-written plugins, you will use `ExperimentManifest`. Here's the definition for the `SignalFFTViewManifest`:

```
from enaml.workbench.api import Extension

from psi.core.enaml.api import ExperimentManifest
from psi.data.plots import (ChannelPlot, FFTChannelPlot, FFTContainer,
                            TimeContainer, ViewBox)

enamldef SignalFFTViewManifest(ExperimentManifest): manifest:

    id = 'signal_fft_view'
    name = 'signal_fft_view'
    title = 'Signal view (PSD)'

    # By re-defining attributes of the "children" in the graph hierarchy as
    # attributes of the top-level manifest, we can easily modify the values
    # of these attributes in the PluginDescription by passing in an attrs
    # dictionary.
    alias fft_time_span: fft_plot.time_span
    alias fft_freq_lb: fft_container.freq_lb
    alias fft_freq_ub: fft_container.freq_ub
    alias source_name: fft_plot.source_name
    alias y_label: fft_vb.y_label
    alias apply_calibration: fft_plot.apply_calibration
    alias waveform_averages: fft_plot.waveform_averages

    Extension:
        id = manifest.id  + '.plots'
        point = 'psi.data.plots'

        FFTContainer: fft_container:
            name << manifest.name + '_container'
            label << manifest.title
            freq_lb = 5
            freq_ub = 50000

            ViewBox: fft_vb:
                name << manifest.name + '_vb'
                y_min = -10
                y_max = 100
                y_mode = 'mouse'
                save_limits = True

                FFTChannelPlot: fft_plot:
                    name << manifest.name + '_plot'
                    source_name = 'microphone'
                    pen_color = 'k'
                    time_span = 0.25
```

This manifest extends the `psi.data.plots` extension point by adding a new FFT plot that will plot the running FFT of

the signal source defined by `source_name`. Another good example is the reward dispenser. This example demonstrates how you can define some basic functionality and then subclass the manifest to customize it:

```
enamldef BaseRewardDispenser(ExperimentManifest): manifest:

    attr duration
    attr output_name

    Extension:
        id = manifest.id + '.actions'
        point = 'psi.controller.actions'
        ExperimentAction:
            event = 'deliver_reward'
            command = f'{manifest.output_name}.fire'
            kwargs = {'duration': manifest.duration}

    Extension:
        id = manifest.id + '.status_item'
        point = 'psi.experiment.status'

        StatusItem:
            label = 'Total dispensed'
            Label:
                text << str(workbench \
                            .get_plugin('psi.controller') \
                            .get_output(manifest.output_name) \
                            .total_fired)

    Extension:
        id = manifest.id + '.toolbar'
        point = 'psi.experiment.toolbar'
        rank = 2000
        Action:
            text = 'Dispense reward'
            triggered ::
                controller = workbench.get_plugin('psi.controller')
                controller.invoke_actions('deliver_reward')
            enabled <<  workbench.get_plugin('psi.controller').experiment_state \
                not in ('initialized', 'stopped')


enamldef WaterBolusDispenser(BaseRewardDispenser): manifest:

    id = 'water_bolus_dispenser'
    name = 'Water bolus dispenser'
    required = True

    duration = C.lookup('water_dispense_duration')
    output_name = 'water_dispense'

    Extension:
        id = manifest.id + '.parameters'
        point = 'psi.context.items'
```

(continues on next page)

(continued from previous page)

```
        Parameter:
            name = 'water_dispense_duration'
            label = 'Water dispense trigger duration (s)'
            compact_label = 'D'
            default = 1
            scope = 'arbitrary'
            group_name = 'trial'
```

The first extension is to the `psi.controller.actions` point where we define a command that is called each time the `deliver_reward` event occurs. The `deliver_reward` event is generated by the core behavior controller whenever it determines that the subject has met the criteria for recieving a reward. By defining this as an event, we can link any number of actions (defined by `ExperimentAction`). Here, the action is to find the digital output defined by `output_name` and call the `fire` command with the specified trigger duration. In the LBHB system, this digital output is linked to a solenoid that opens when the trigger goes high, thereby allowing water to flow through into a lick spout that the subject has access to. This is a good example of where you can easily customize what happens during the `deliver_reward` stage to switch to an alternate reward system by loading a different plugin.

The second extension is to the `psi.experiment.status` point which is a user-facing panel that shows information about the experiment. This must subclass `StatusItem` and contribute a specific widget (in this case, a `Label`, but other good widgets include a `ProgressBar`). The extension to `psi.experiment.toolbar` adds a button that invokes all actions connected to the `deliver_reward` event. Finally, the `psi.context.items` extension manages all parameters (subclasses of `ContextItems`). Parameters are variables (e.g., intertrial duration, reward trigger duration, number fo trials, etc.) that the user may want to control.

3.3 Designing a new experiment

3.3.1 Getting started

For a broad overview of the components involved in defining an experiment, see *paradigms_overview*.

3.3.2 Core plugins

Psiexperiment ships with five core plugins that are always loaded when an experiment is started:

- **Context** -
- **Data** - Manages saving, analysis and plotting of data.
- **Controller** - Manages experiment.
- **Token** - Manages generation of both epoch (i.e., finite in duration) and continuous (i.e., infinite in duration) waveforms.
- **Calibration** - Manages calibrations of inputs and outputs. Right now only acoustic inputs and outputs (e.g., microphones and speakers) are supported. Offers chirp, golay and tone-based calibration algorithms.

3.3.3 Getting started

Psiexperiment is a plugin-based system where the plugins determine the experiment workflow. At a minimum, an experiment must provide the following:

- A list of parameters and/or results
- The inputs and outputs that will be used
- The stimuli (i.e., tokens) that will be generated
- Actions to take when certain events occur.

Your experiment configuration file will define *EXPERIMENT*, which is the name of the experiment and will typically contain the following extensions:

```
enamldef ControllerManifest(BaseManifest): manifest:

    Extension:
        id = EXPERIMENT + '.sinks'
        point = 'psi.data.sinks'
        ...

    Extension:
        id = EXPERIMENT + '.tokens'
        point = 'psi.token.tokens'
        ...

    Extension:
        id = EXPERIMENT + '.io'
        point = 'psi.controller.io'
        ...

    Extension:
        id = EXPERIMENT + '.selectors'
        point = 'psi.context.selectors'
        ...

    Extension:
        id = EXPERIMENT + '.context'
        point = 'psi.context.items'
        ...

    Extension:
        id = EXPERIMENT + '.actions'
        point = 'psi.controller.actions'
        ...
```

Let's take a closer look at each of the extensions.

Actions

At a minimum, you will typically define the following two actions (customized for your needs):

```
Extension:
    id = EXPERIMENT + '.actions'
    point = 'psi.controller.actions'

    ExperimentAction:
        event = 'experiment_initialize'
        command = 'psi.context.initialize'
        kwargs = {'selector': 'default', 'cycles': 1}

    ExperimentAction:
        event = 'engines_configured'
        command = 'dpoae.start'
        kwargs = {'delay': 0.5}
```

When you press the *start* button on the toolbar, this fires a sequence of three events, *experiment_initialize*, *experiment_prepare* and *experiment_start*. While *experiment_initialize* and *experiment_prepare* are very similar, certain actions may require that the context has been initialized. To simplify this, we have added *experiment_initialize*. The *psi.context.initialize* command should *always* be bound to this event (if it's bound to *experiment_prepare*, you may get errors if commands bound to *experiment_prepare* need to get the current value of a variable).

Under the hood, the controller will configure the engines during the *experiment_prepare* phase. If you want to configure one of the outputs (in this case, *dpoae*) during this phase, be sure to bind it to the *engine_configured* event to ensure it gets executed after the engine is configured (the engine must be configured before it can properly receive waveform samples from the outputs).

Sequence of events during an experiment

- *plugins_started* - All plugins have finished loading. Now, you can perform actions that may require access to another plugin; however, do not assume that the plugins have finished initializing. A number of logging actions are tied to this step.
- *experiment_initialize* - All plugins should have been initialized. This is where you will typically initialize the context (and nothing else).
- *context_initialized* - This only follows *experiment_initialize* if *psi.context.initialize* has properly been bound to *experiment_initialize*. The *psi.context.finalize_io* method is called during this event. During this step, all "orphan" inputs and outputs (i.e., ones where the target or source is specified by name rather than as part of the hierarchy) are connected.
- *experiment_prepare* - The majority of actions required prior to starting an experiment should be tied to this event since the context will now be available for queries.
- *engines_configured* - TODO
- *experiment_start* - Starts the data acquisition engines.
- *experiment_end* - Stops the data acquisition engines.

The power of actions

Actions allow you to insert your own code or invoke commands at any point in the experiment process. A few examples:

- The *abr_base.enaml* file calls a custom function when the *experiment_prepare* event is called. This function reviews the settings specified by the user to determine the sequence of the tone pips (e.g., conventional vs. interleaved, alternating polarity, etc.) and sets up the queue accordingly. While it's theoretically possible to set this using plugins offered by psiexperiment (e.g., alternating polarity could be specified as a "roving" context item), this custom function makes the user interface much simpler and more fool-proof.
- The *pistonphone_calibration.enaml* file calls a custom function, *calculate_sens* once the experiment is complete to calculate the sensitivity of the microphone. Note that the callback for the custom function is defined inside the extension to the *psi.controller.io* point.

Input/Output

Example of an input-output plugin:

```
Extension:
    id = EXPERIMENT + '.io'
    point = 'psi.controller.io'

    Blocked: hw_ai:
        duration = 0.1
        name = 'hw_ai'
        source_name = C.input_channel
        source ::
            # Once the channel is linked
            channel.start_trigger = ''
            channel.samples = round(C.sample_duration * channel.fs)
            channel.input_gain = C.input_gain
```

`C` is a controller manifest-level variable that allows for lookup of values defined via the context.

Creating your own custom plugins

When defining your own subclasses of `PSIManifest`, we recommend the following naming convetions to minimize name collisions:

```
Extension:
    id = manifest.id + '.commands'
    point = 'enaml.workbench.core.commands'

    Command:
        id = contribution.name + '.do_action'
        ...
```

All subclasses of `PSIManifest` have access to the attached `contribution` (an instance of `PSIContribution`) as an attribute.

3.3.4 Common gotchas

- Outputs and inputs are configured *only if they are deemed active*. If the output of a particular processing chain (e.g., microphone to IIR filter to extract epochs) is not saved to a data store or plotted, then it's assumed it is not used. The controller will then omit this particular processing chain from the configuration to alleviate system load. This allows us to design intensive processing chains but allow the user to disable them easily by not plotting the result. However, this can be a bit tricky when defining your own custom sinks For example, there's no target for `AnalyzeDPOAE` in `dpoae_base.enaml` (TODO finish).
- When adding new attributes to subclasses of `Declarative`, be sure to use `d_` as appropriate otherwise you will get a `TypeError` when attempting to assign to the attribute in an Enaml file.
- Use `set_default` when setting default values for classes derived from `Atom` where the original attribute was defined in a superclass (hint, `Declarative` is a subclass of `Atom`):

```
class Channel(Declarative):
    name = Str('input_A')

class ChannelB(Channel)
    name = set_default('input_B')
```

- Even if you define a `ContinuousOutput`, you still need to configure it to start using an `ExperimentAction`. Assuming your continuous output is named "masker", then it's as simple as adding the following action:

```
ExperimentAction:
    event = 'engines_configured'
    command = 'masker.start'
```

- You must always call `psi.context.initialize`. This is not automatically done for you for a variety of reasons. Usually it's sufficient to insert the following action:

```
ExperimentAction:
    event = 'experiment_initialize'
    command = 'psi.context.initialize'
    kwargs = {'selector': None}
```

3.4 Waveforms

We've already discussed this briefly. However, what we have not discussed is *how* we tell the program to present a particular waveform. The engines are *always* polling every output (continuous, epoch, queued, etc.) for samples. If an output is not active (e.g., we are in the intertrial period and therefore there is no target to present), the epoch output will simply return a string of zeros. The engine actually buffers about 10 to 30 seconds of data in the output queues (to protect the experiment from crashing when you stall the computer by checking your email).

We can activate an output called *target* to begin at 10.5 seconds (re. acquisition start):

```
core = workbench.get_plugin('enaml.workbench.core')
core.invoke_command('target.prepare')
core.invoke_command('target.start', {'ts': 10.5})
```

Commands are strings that map to curried functions.

3.5 Context

Every experiment has a set of variables that define the behavior. These variables range from the stimulus frequency and level to the intertrial interval duration. Sometimes these variables need to be expressed as functions of other variables, or the value of the variable needs to vary in a random fashion.

A *context item* provides information about a value that is managed by the context plugin. When defining a context item in one of your plugin manifests, you will provide basic information about the item (e.g., a GUI label, compact GUI label and numpy dtype). This information will be used by plugins that interact with the context plugin (for example, the name and dtype of the context item will be used by the TextStore plugin to set up the table that stores acquired trial data).

There are currently three specific types of context items. A *result* is a value provided by a plugin. It cannot be defined by the user. Common use cases may include the values computed after a trial is acquired (e.g., one can compute the *reaction_time* and provide it as a result).

A *parameter* is a value that can be configured by the user before and during an experiment. While the value of the parameter can be modified by the user during the experiment, it cannot be roved. There are some parameters that do not make sense as roving parameters. For example, if we define a *go_probability* parameter that determines the probability that the next trial is a GO trial instead of a NOGO, it does not make sense to rove this value from trial-to-trial. It, however, may make sense to change this during the couse of an experiemnt (e.g., during training).

A *roving parameter* is like a parameter, except that it can be roved from trial to trial. When selected for roving, the next value of the parameter is provided by a selector. A *selector* maintains a sequence of expressions for one or more roving parameters. In some experiments, you'll only have a single selector. In other experiments, you may want multiple selectors (e.g., one for go trials, one for remind trials and one for nogo trials). Right now, the only difference between different types of selectors will be the GUI that's presented to the user for configuring the sequence of values. Internally, all of them maintain a list of values that should be presented on successive trials.

You can define a list of parameters required by the experiment as well as a set of selectors that allow you to specify a sequence of values to test. Once you have defined these parameters, you can iterate through them:

```
context = workbench.get_plugin('psi.context')
context.next_setting(selector='go')
values = context.get_values()

context.next_setting(selector='nogo')
values = context.get_values()
```

This strategy is used in the appetitive go-nogo experiment.

3.6 psi package

3.6.1 Subpackages

psi.application package

Submodules

psi.application.api module

psi.application.base_launcher module

Submodules

psi.data.api module

psi.data.plots module

psi.data.plugin module

Module contents

psi.experiment package

Submodules

psi.experiment.api module

psi.experiment.paradigm_description module

psi.experiment.plugin module

psi.experiment.preferences module

psi.experiment.status_item module

psi.experiment.util module

psi.experiment.workbench module

psi.experiment.workspace module

Module contents

psi.templates package

Subpackages

psi.templates.io package

Module contents

Module contents

psi.token package

Submodules

psi.token.api module

psi.token.block module

psi.token.plugin module

Module contents

3.6.2 Submodules

3.6.3 psi.util module

3.6.4 Module contents

3.7 psi

3.8 Development Workflow

Suggestions for setting up psiexperiment for development.

3.9 What makes an experiment?

How are experiments defined?

3.10 Context

Define parameters and sequences.

3.11 Waveforms

Create stimuli.

3.12 Designing a new experiment

Instructions on how to create your own experiment.

3.13 psi

Overview of API.

CHAPTER

FOUR

REFERENCES

CHAPTER

FIVE

CONTRIBUTORS AND ACKNOWLEDGEMENTS

- Brad Buran (New York University, Oregon Health & Science University)
- Decibel Therapeutics, Inc.

Work on psiexperiment was supported by grants R01-DC009237 and R21-DC016969 from the National Institute on Deafness and Other Communication Disorders and an Emerging Research Grant from the Hearing Health Foundation. Ann Hickox provided extensive testing and feedback on the TDT backend for psiexperiment.