pyoperant Documentation

Release 0.1.0

Justin Kiggins, Marvin Thielk

Apr 25, 2017

Contents

NOTICE!! This package has been renamed to “opyrant” and development has moved to https://github.com/opyrant/opyrant

CHAPTER 1

pyoperant

Pyoperant is a framework to easily construct and share new operant behavior paradigms.

With PyOperant, you can write a single behavior script that works across different species, different computers, different hardware, different rewards, different modalities.

Operant logic is easy

1. Present a stimulus
2. Get the subject's response
3. If the response matches the stimulus, then reward the subject

Writing operant protocols should be easy, but in practice...

Error checking, data storage, and machine-specific hardware interactions often obfuscate the simplicity of the task, limiting its flexibility and power. This limitation becomes increasingly apparent when deploying high-throughput behavioral experiment control systems, transferring subjects from a training panel to an electrophysiology panel, or simply trying to share behavioral protocols.

A better way

PyOperant deals with these challenges by providing a cross-platform object-oriented framework to easily construct, conveniently share, and rapidly iterate on new operant behavior paradigms.

1. Abstract physical component manipulation from low-level hardware manipulation
2. Define behavioral protocols as classes which can be extended through object inheritance

Further, experimenters are able to integrate their behavioral protocols with other Python packages for online data analysis or experimental control. We currently use pyoperant in the Gentner Lab to control 36 operant panels.

Documentation

PyOperant abstracts behavioral protocol logic from hardware interactions through a machine-specific configuration file. In the local.py configuration file, the experimenter defines the operant panels available for use. A Panel consists of a collection of Component objects and a set of standard methods to manipulate the Component. These Component objects are mirrors of their physical counterparts, such as a food hopper, response port, speaker, or house light.

Behavioral protocols can be modifed and extended through object inheritance. The modular architecture of PyOperant also allows experimenters to integrate their behavioral protocols with other Python packages for online data analysis or experimental control.

PyOperant's hardware support currently includes PortAudio & Comedi. Future support will include NiDAQmx and Cambridge Electronic Designs.

http://pyoperant.readthedocs.org/en/dev/index.html

Architecture

Behaviors

Behaviors are Python classes which run the operant experiment. They associate the subject with the hardware panel the subject is interacting with and save experimental data appropriately. They are instantiated with various experimental parameters, such as stimulus identities and associations, block designs, and reinforcement schedules.

There are a couple of built-in behaviors: TwoAltChoice, which runs two alternative choice tasks and Lights, which simply turns the house light on and off according to a schedule. These can be inherited to change specific methods without changing the rest of the behavioral protocol.

Panels

Panels are the highest level of hardware abstraction. They maintain panel components as attributes and have standard methods for resetting and testing the panel. Many Behaviors rely on specific panel components and methods to be present.

Panels are defined by the experimenter locally.

Components

Components are common hardware components, such as a Hopper, a ResponsePort, a HouseLight, or an RGBLight. Many components rely on multiple hardware IO channels. For example, a Hopper requires both a solenoid (to activate the Hopper) and an IR beam detector (to check if the Hopper is raised). Calling the 'feed' method on a Hopper checks to make sure that the hopper is down, raises the hopper, checks to make sure the hopper raised, waits the appropriate length of time, then lowers the hopper, finally checking one more time to make sure the hopper dropped. If there is an incongruity between the status of the solenoid and the IR beam, the Hopper component raises the appropriate error, which the Behavior script can deal with appropriately.

Hardware IO Classes

Hawdware IO classes standardize inputs and outputs that are available for Components and Panels to use.

Hardware interfaces

Hardware interfaces are wrappers around hardware drivers and APIs that allow hardware IO classes to work.

Developers

Justin Kiggins & Marvin Thielk

Gentner Lab - http://gentnerlab.ucsd.edu

CHAPTER 2

Contents

pyoperant package

Subpackages

pyoperant.interfaces package

Submodules

pyoperant.interfaces.base_ module

class pyoperant.interfaces.base_.**BaseInterface**(**args*, ***kwargs*)

Bases: object

docstring for BaseInterface

close()

open()

pyoperant.interfaces.comedi_ module

pyoperant.interfaces.console_ module

class pyoperant.interfaces.console_.**ConsoleInterface**(**args*, ***kwargs*)

Bases: *pyoperant.interfaces.base_.BaseInterface*

docstring for ComediInterface

pyoperant.interfaces.pyaudio_ module

pyoperant.interfaces.spike2_ module

class pyoperant.interfaces.spike2_.**Spike2Interface**
Bases: *pyoperant.interfaces.base_.BaseInterface*

docstring for Spike2Interface

close()

open()

Module contents

pyoperant.behavior package

Submodules

pyoperant.behavior.base module

pyoperant.behavior.lights module

pyoperant.behavior.shape module

pyoperant.behavior.three_ac_matching module

pyoperant.behavior.two_alt_choice module

Module contents

Submodules

pyoperant.components module

pyoperant.errors module

exception pyoperant.errors.**ComponentError**
Bases: exceptions.Exception

raised for errors with a component.

this should indicate a hardware error in the physical world, like a problem with a feeder.

this should be raised by components when doing any internal validation that they are working properly

exception pyoperant.errors.**EndBlock**
Bases: exceptions.Exception

exception for when a block should terminate

exception pyoperant.errors.**EndSession**
Bases: exceptions.Exception

exception for when a session should terminate

exception pyoperant.errors.**Error**
Bases: exceptions.Exception

base class for exceptions in this module

exception pyoperant.errors.**GoodNite**
Bases: exceptions.Exception

exception for when the lights should be off

exception pyoperant.errors.**InterfaceError**
Bases: exceptions.Exception

raised for errors with an interface.

this should indicate a software error, like difficulty connecting to an interface

pyoperant.hwio module

class pyoperant.hwio.**AudioOutput**(*interface=None*, *params={}*, **args*, ***kwargs*)
Bases: *pyoperant.hwio.BaseIO*

Class which holds information about audio outputs and abstracts the methods of writing to them

Keyword arguments: interface – Interface() instance. Must have the methods '_queue_wav',

'_play_wav', '_stop_wav'

params – dictionary of keyword:value pairs needed by the interface

Methods: queue(wav_filename) – queues read() – if the interface supports '_read_bool' for this output, returns

the current value of the output from the interface. Otherwise this returns the last passed by write(value)

toggle() – flips the value from the current value

play()

queue(*wav_filename*)

stop()

class pyoperant.hwio.**BaseIO**(*interface=None*, *params={}*, **args*, ***kwargs*)
Bases: object

any type of IO device. maintains info on interface for query IO device

class pyoperant.hwio.**BooleanInput**(*interface=None*, *params={}*, **args*, ***kwargs*)
Bases: *pyoperant.hwio.BaseIO*

Class which holds information about inputs and abstracts the methods of querying their values

Keyword arguments: interface – Interface() instance. Must have '_read_bool' method. params – dictionary of keyword:value pairs needed by the interface

Methods: read() – reads value of the input. Returns a boolean poll() – polls the input until value is True. Returns the time of the change

config()

poll(*timeout=None*)
runs a loop, querying for pecks. returns peck time or "GoodNite" exception

read()
read status

class pyoperant.hwio.**BooleanOutput**(*interface=None*, *params={}*, **args*, ***kwargs*)
Bases: *pyoperant.hwio.BaseIO*

Class which holds information about outputs and abstracts the methods of writing to them

Keyword arguments: interface – Interface() instance. Must have '_write_bool' method. params – dictionary of keyword:value pairs needed by the interface

Methods: write(value) – writes a value to the output. Returns the value read() – if the interface supports '_read_bool' for this output, returns

the current value of the output from the interface. Otherwise this returns the last passed by write(value)

toggle() – flips the value from the current value

config()

read()
read status

toggle()

write(*value=False*)
write status

pyoperant.local module

pyoperant.local_vogel module

pyoperant.local_zog module

pyoperant.panels module

class pyoperant.panels.**BasePanel**(**args*, ***kwargs*)
Bases: object

Returns a panel instance.

This class should be subclassed to define a local panel configuration.

To build a panel, do the following in the __init__() method of your local subclass:

1. **add instances of the necessary interfaces to the 'interfaces' dict** attribute: >>> self.interfaces['comedi'] = comedi.ComediInterface(device_name='/dev/comedi0')
2. **add inputs and outputs to the 'inputs' and 'outputs' list attributes:**

```
>>> for in_chan in range(4):
        self.inputs.append(hwio.BooleanInput(interface=self.interfaces[
↪'comedi'],
                                             params = {'subdevice': 2,
                                                       'channel': in_chan
                                                       },
                                             )
```

3. **add components constructed from your inputs and outputs:**

```
>>> self.hopper = components.Hopper(IR=self.inputs[3],solenoid=self.
↪outputs[4])
```

4. **assign panel methods needed for operant behavior, such as 'reward':**

```
>>> self.reward = self.hopper.reward
```

5. **finally, define a reset() method that will set the entire panel to a** neutral state:

```
>>> def reset(self):
>>>     for output in self.outputs:
>>>         output.set(False)
>>>     self.house_light.write(True)
>>>     return True
```

reset()

pyoperant.queues module

pyoperant.reinf module

class pyoperant.reinf.**BaseSchedule**

Bases: object

Maintains logic for deciding whether to consequate trials.

This base class provides the most basic reinforcent schedule: every response is consequated.

Methods: consequate(trial) – returns a boolean value based on whether the trial

should be consequated. Always returns True.

consequate(*trial*)

class pyoperant.reinf.**ContinuousReinforcement**

Bases: *pyoperant.reinf.BaseSchedule*

Maintains logic for deciding whether to consequate trials.

This base class provides the most basic reinforcent schedule: every response is consequated.

Methods: consequate(trial) – returns a boolean value based on whether the trial

should be consequated. Always returns True.

consequate(*trial*)

class pyoperant.reinf.**FixedRatioSchedule**(*ratio=1*)

Bases: *pyoperant.reinf.BaseSchedule*

Maintains logic for deciding whether to consequate trials.

This class implements a fixed ratio schedule, where a reward reinforcement is provided after every nth correct response, where 'n' is the 'ratio'.

Incorrect trials are always reinforced.

Methods: consequate(trial) – returns a boolean value based on whether the trial

should be consequated.

consequate(*trial*)

class pyoperant.reinf.**PercentReinforcement**(*prob=1*)
Bases: *pyoperant.reinf.BaseSchedule*

Maintains logic for deciding whether to consequate trials.

This class implements a probabalistic reinforcement, where a reward reinforcement is provided x percent of the time.

Incorrect trials are always reinforced.

Methods: consequate(trial) – returns a boolean value based on whether the trial

should be consequated.

consequate(*trial*)

class pyoperant.reinf.**VariableRatioSchedule**(*ratio=1*)
Bases: *pyoperant.reinf.FixedRatioSchedule*

Maintains logic for deciding whether to consequate trials.

This class implements a variable ratio schedule, where a reward reinforcement is provided after every a number of consecutive correct responses. On average, the number of consecutive responses necessary is the 'ratio'. After a reinforcement is provided, the number of consecutive correct trials needed for the next reinforcement is selected by sampling randomly from the interval [1,2*ratio-1]. e.g. a ratio of '3' will require consecutive correct trials of 1, 2, 3, 4, & 5, randomly.

Incorrect trials are always reinforced.

Methods: consequate(trial) – returns a boolean value based on whether the trial

should be consequated.

pyoperant.utils module

Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

V

W