pyoperant Documentation

Release 0.1.0

Justin Kiggins, Marvin Thielk

Jan 31, 2019

Contents

CHAPTER 1

pyoperant

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

1.1 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

1.2 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.

1.3 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.

1.4 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

1.5 Architecture

1.5.1 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.

1.5.2 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.

1.5.3 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.

1.5.4 Hardware IO Classes

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

1.5.5 Hardware interfaces

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

1.6 Developers

Justin Kiggins & Marvin Thielk

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

CHAPTER 2

Contents

2.1 pyoperant package

2.1.1 Subpackages

2.1.1.1 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

class pyoperant.interfaces.comedi_.**ComediInterface**(*device_name*, **args*, ***kwargs*)
Bases: *pyoperant.interfaces.base_.BaseInterface*

docstring for ComediInterface

close()

open()

pyoperant.interfaces.console_ module

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

Bases: *pyoperant.interfaces.base_.BaseInterface*

docstring for ComediInterface

pyoperant.interfaces.pyaudio_ module

class pyoperant.interfaces.pyaudio_.**PyAudioInterface**(*device_name='default'*, **args*, ***kwargs*)

Bases: *pyoperant.interfaces.base_.BaseInterface*

Class which holds information about an audio device

assign a simple callback function that will execute on each frame presentation by writing interface.callback

interface.callback() should return either True (to continue playback) or False (to terminate playback)

Before assigning any callback function, please read the following: https://www.assembla.com/spaces/portaudio/wiki/Tips_Callbacks

close()

open()

validate()

pyoperant.interfaces.spike2_ module

class pyoperant.interfaces.spike2_.**Spike2Interface**

Bases: *pyoperant.interfaces.base_.BaseInterface*

docstring for Spike2Interface

close()

open()

Module contents

2.1.1.2 pyoperant.behavior package

Submodules

pyoperant.behavior.base module

class pyoperant.behavior.base.**BaseExp**(*name=''*, *description=''*, *debug=False*, *filetime_fmt='%Y%m%d%H%M%S'*, *light_schedule='sun'*, *idle_poll_interval=60.0*, *experiment_path=''*, *stim_path=''*, *subject=''*, *panel=None*, *log_handlers=[]*, **args*, ***kwargs*)

Bases: object

Base class for an experiment.

Keyword arguments: name – name of this experiment desc – long description of this experiment debug – (bool) flag for debugging (default=False) light_schedule – the light schedule for the experiment. either 'sun' or

a tuple of (starttime,endtime) tuples in (hhmm,hhmm) form defining time intervals for the lights to be on

experiment_path – path to the experiment stim_path – path to stimuli (default = <experiment_path>/stims) subject – identifier of the subject panel – instance of local Panel() object

Methods: run() – runs the experiment

check_light_schedule()
returns true if the lights should be on

check_session_schedule()
returns True if the subject should be running sessions

deliver_free_food(*value*, *next_state*)
reward function with no frills

food_checker(*next_state*)

free_food_main()
reset expal parameters for the next day

free_food_post()

free_food_pre()

init_summary()
initializes an empty summary dictionary

log_config()

log_error_callback(*err*)

panel_reset()

run()

save()

session_main()

session_post()

session_pre()

sleep_main()
reset expal parameters for the next day

sleep_post()

sleep_pre()

write_summary()
takes in a summary dictionary and options and writes to the bird's summaryDAT

pyoperant.behavior.lights module

This submodule controls the light schedule in the animal's environment. It is documented in more detail in the main pyoperant module $pyoperant.components

class pyoperant.behavior.lights.**Lights**(**args*, ***kwargs*)
Bases: *pyoperant.behavior.base.BaseExp*

docstring for Lights

panel_reset()

pyoperant.behavior.shape module

class pyoperant.behavior.shape.**Shaper**(*panel*, *log*, *parameters*, *error_callback=None*)
Bases: object

Run a shaping routine in the operant chamber that will teach an to peck the center key to hear a stimulus, then peck one of the side keys for reward. training sequence: Block 1: Hopper comes up on VI (stays up for 5 s) for the first day

that the animal is in the apparatus. Center key flashes for 5 sec, prior to the hopper access. If the center key is pressed while flashing, then the hopper comes up and then the session jumps to block 2 immediately.

Block 2: The center key flashes until pecked. When pecked the hopper comes up for 4 sec. Run 100 trials.

Block 3: The center key flashes until pecked, then either the right or left (p = .5) key flashes until pecked, then the hopper comes up for 3 sec. Run 100 trials.

Block 4: Wait for peck to non-flashing center key, then right or left key flashes until pecked, then food for 2.5 sec. Run 100 trials.

block_name(*block_num*)

deliver_free_food(*value*, *next_state*)
reward function with no frills

food_checker(*next_state*)

free_food_main()
reset expal parameters for the next day

free_food_post()

free_food_pre()

reward(*value*, *next_state*)

run_shape(*start_state='block1'*)

sleep_main()
reset expal parameters for the next day

sleep_post()

sleep_pre()

class pyoperant.behavior.shape.**Shaper2AC**(*panel*, *log*, *parameters*, *error_callback=None*)
Bases: *pyoperant.behavior.shape.Shaper*

Run a shaping routine in the operant chamber that will teach an to peck the center key to hear a stimulus, then peck one of the side keys for reward. training sequence: Block 1: Hopper comes up on VI (stays up for 5 s) for the first day

that the animal is in the apparatus. Center key flashes for 5 sec, prior to the hopper access. If the center key is pressed while flashing, then the hopper comes up and then the session jumps to block 2 immediately.

Block 2: The center key flashes until pecked. When pecked the hopper comes up for 4 sec. Run 100 trials.

Block 3: The center key flashes until pecked, then either the right or left (p = .5) key flashes until pecked, then the hopper comes up for 3 sec. Run 100 trials.

Block 4: Wait for peck to non-flashing center key, then right or left key flashes until pecked, then food for 2.5 sec. Run 100 trials.

class pyoperant.behavior.shape.**Shaper3AC**(*panel*, *log*, *parameters*, *error_callback=None*)
Bases: *pyoperant.behavior.shape.Shaper*

run a shaping routine for 3AC the operant chamber termial proc: peck center key for stimulus presentation then peck one of three keys L-C-R, or give no response. Training sequence invoked as: Block 1: Hopper comes up on VI (stays up for 5 s) for the first day

that the animal is in the apparatus. Center key flashes for 5 sec, prior to the hopper access. If the center key is pressed while flashing, then the hopper comes up and then the session jumps to block 2 immediately.

Block 2: The center key flashes until pecked. When pecked the hopper comes up for 4 sec. Run 100 trials.

Block 3: The center key flashes until pecked, then either the right, left, or center key flashes (p=0.333) until pecked, then the hopper comes up for 3 sec. Run 150 trials.

Block 4: Wait for peck to non-flashing center key, then right, center,or left key flashes until pecked, then food for 2.5 sec. Run 150 trials.

class pyoperant.behavior.shape.**Shaper3ACMatching**(*panel*, *log*, *parameters*, *get_stimuli*, *error_callback=None*)
Bases: *pyoperant.behavior.shape.Shaper3AC*

class pyoperant.behavior.shape.**ShaperFemalePref**(*panel*, *log*, *parameters*, *error_callback=None*)
Bases: *pyoperant.behavior.shape.Shaper*

run a shaping routine for female pecking preferencein the operant chamber termial proc: peck one of the side keys for stimulus presentation followed by reward. Training sequence invoked as: Block 1: Hopper comes up on VI (stays up for 5 s) for the first day

that the animal is in the apparatus. Left and right keylights flash for 5 sec, prior to the hopper access. If either L or R key is pressed while flashing, then the hopper comes up and the session jumps to block 2 immediately.

Block 2: randomly choose either L or R key to flash until pecked. When pecked the hopper comes up for 4 sec.

Block 3: Wait for peck to non-flashing L or R key (chosen at random). When pecked, give food for 2.5 sec.

class pyoperant.behavior.shape.**ShaperGoNogo**(*panel*, *log*, *parameters*, *error_callback=None*)
Bases: *pyoperant.behavior.shape.Shaper*

accomodate go/nogo terminal procedure along with one or two hopper 2choice procedures Go/Nogo shaping works like this: Block 1: Hopper comes up on VI (stays up for 5 s) for the first day

that the animal is in the apparatus. Center key flashes for 5 sec, prior to the hopper access. If the center key is pressed while flashing, then the hopper comes up and then the session jumps to block 2 immediately.

Block 2: The center key flashes until pecked. When pecked the hopper comes up for 4 sec. Run 100 trials.

Block 3: Wait for a peck to non-flashing center key, when you get it, the hopper comes up for 2.5 sec. Run 100 trials.

NOTE: when you run the go/nog procedure in a 2 hopper apparatus, it uses only the right hand key and hopper. If you do this often, you may want to add the facility for use of the left hand key and hopper.

pyoperant.behavior.three_ac_matching module

class pyoperant.behavior.three_ac_matching.**ThreeACMatchingExp**(**args*, ***kwargs*)
Bases: *pyoperant.behavior.two_alt_choice.TwoAltChoiceExp*

docstring for ThreeACMatchingExp

analyze_trial()

correction_reward_main()

correction_reward_post()

correction_reward_pre()

get_stimuli(*trial_class*)
take trial class and return a tuple containing the stimulus event to play and a list of additional events

pyoperant.behavior.two_alt_choice module

class pyoperant.behavior.two_alt_choice.**TwoAltChoiceExp**(**args*, ***kwargs*)
Bases: *pyoperant.behavior.base.BaseExp*

A two alternative choice experiment

req_panel_attr
list

list of the panel attributes that are required for this behavior

fields_to_save
list

list of the fields of the Trial object that will be saved

trials
list

all of the trials that have run

shaper
Shaper

the protocol for shaping

parameters
dict

all additional parameters for the experiment

data_csv
string

path to csv file to save data

reinf_sched
object

does logic on reinforcement

analyze_trial()

check_session_schedule()
Check the session schedule

Returns True if sessions should be running

Return type bool

consequence_main()

consequence_post()

consequence_pre()

get_stimuli(***conditions*)
Get the trial's stimuli from the conditions

Returns stim, epochs

Return type *Event*, list

make_data_csv()
Create the csv file to save trial data

This creates a new csv file at experiment.data_csv and writes a header row with the fields in experiment.fields_to_save

new_trial(*conditions=None*)
Creates a new trial and appends it to the trial list

If *self.do_correction* is *True*, then the conditions are ignored and a new trial is created which copies the conditions of the last trial.

Parameters **conditions** (*dict*) – The conditions dict must have a 'class' key, which specifys the trial class. The entire dict is passed to *exp.get_stimuli()* as keyword arguments and saved to the trial annotations.

punish_main()

punish_post()

punish_pre()

response_main()

response_post()

response_pre()

reward_main()

reward_post()

reward_pre()

run_trial()

save_trial(*trial*)
write trial results to CSV

secondary_reinforcement(*value=1.0*)

session_main()
Runs the sessions

Inside of *session_main*, we loop through sessions and through the trials within them. This relies heavily on the 'block_design' parameter, which controls trial conditions and the selection of queues to generate trial conditions.

session_post()
Closes out the sessions

session_pre()
Runs before the session starts

For each stimulus class, if there is a component associated with it, that component is mapped onto *experiment.class_assoc[class]*. For example, if the *left* port is registered with the 'L' class, you can access the response port through *experiment.class_assoc['L']*.

stimulus_main()

stimulus_post()

stimulus_pre()

trial_post()
things to do at the end of a trial

trial_pre()
this is where we initialize a trial

update_adaptive_queue(*presented=True*)

Module contents

2.1.2 Modules

2.1.2.1 pyoperant.components module

class pyoperant.components.**BaseComponent**(*name=None*, **args*, ***kwargs*)
Bases: object

Base class for physcal component

class pyoperant.components.**Hopper**(*IR*, *solenoid*, *max_lag=0.3*, *inverted=False*, **args*, ***kwargs*)
Bases: *pyoperant.components.BaseComponent*

Class which holds information about a hopper

Parameters

- **solenoid** (*hwio.BooleanOutput*) – output channel to activate the solenoid & raise the hopper
- **IR** (hwio.BooleanInput) – input channel for the IR beam to check if the hopper is up

- **max_lag** (*float, optional*) – time in seconds to wait before checking to make sure the hopper is up (default=0.3)

solenoid
hwio.BooleanOutput

output channel to activate the solenoid & raise the hopper

IR
hwio.BooleanInput

input channel for the IR beam to check if the hopper is up

max_lag
float

time in seconds to wait before checking to make sure the hopper is up

check()
reads the status of solenoid & IR beam, then throws an error if they don't match

Returns True if the hopper is up.

Return type bool

Raises

- *HopperActiveError* – The Hopper is up and it shouldn't be. (The IR beam is tripped, but the solenoid is not active.)
- *HopperInactiveError* – The Hopper is down and it shouldn't be. (The IR beam is not tripped, but the solenoid is active.)

down()
Lowers the hopper.

Returns True if the hopper drops.

Return type bool

Raises *HopperWontDropError* – The Hopper did not drop.

feed(*dur=2.0*, *error_check=True*)
Performs a feed

dur [float, optional] duration of feed in seconds

Returns Timestamp of the feed and the feed duration

Return type (datetime, float)

Raises

- *HopperAlreadyUpError* – The Hopper was already up at the beginning of the feed.
- *HopperWontComeUpError* – The Hopper did not raise for the feed.
- *HopperWontDropError* – The Hopper did not drop fater the feed.

reward(*value=2.0*)
wrapper for *feed*, passes *value* into *dur*

up()
Raises the hopper up.

Returns True if the hopper comes up.

Return type bool

Raises *HopperWontComeUpError* – The Hopper did not raise.

exception pyoperant.components.**HopperActiveError**
Bases: *pyoperant.errors.ComponentError*

raised when the hopper is up when it shouldn't be

exception pyoperant.components.**HopperAlreadyUpError**
Bases: *pyoperant.components.HopperActiveError*

raised when the hopper is already up before it goes up

exception pyoperant.components.**HopperInactiveError**
Bases: *pyoperant.errors.ComponentError*

raised when the hopper is down when it shouldn't be

exception pyoperant.components.**HopperWontComeUpError**
Bases: *pyoperant.components.HopperInactiveError*

raised when the hopper won't come up

exception pyoperant.components.**HopperWontDropError**
Bases: *pyoperant.components.HopperActiveError*

raised when the hopper won't drop

class pyoperant.components.**HouseLight**(*light*, **args*, ***kwargs*)
Bases: *pyoperant.components.BaseComponent*

Class which holds information about the house light

light [hwio.BooleanOutput] output channel to turn the light on and off

Methods: on() – off() – timeout(dur) – turns off the house light for 'dur' seconds (default=10.0) punish() – calls timeout() for 'value' as 'dur'

off()
Turns the house light off.

Returns True if successful.

Return type bool

on()
Turns the house light on.

Returns True if successful.

Return type bool

punish(*value=10.0*)
Calls *timeout(dur)* with *value* as *dur*

timeout(*dur=10.0*)
Turn off the light for *dur* seconds

dur [float, optional] The amount of time (in seconds) to turn off the light.

Returns Timestamp of the timeout and the timeout duration

Return type (datetime, float)

class pyoperant.components.**LEDStripHouseLight**(*lights, color=[100.0, 100.0, 100.0, 100.0], *args, **kwargs*)

Bases: *pyoperant.components.BaseComponent*

Class which holds information about the RGBW LED Strip PWM house light

light [hwio.PWMOutputs] [R, G, B, W] output channels to turn the light on and off

Methods: on() – off() – set_color() – set the color change_color – sets color and turns on light timeout(dur) – turns off the house light for 'dur' seconds (default=10.0) punish() – calls timeout() for 'value' as 'dur'

change_color(*color*)

off()

Turns the house light off.

Returns True if successful.

Return type bool

on()

Turns the house light on.

Returns True if successful.

Return type bool

punish(*value=10.0*)

Calls *timeout(dur)* with *value* as *dur*

set_color(*color*)

timeout(*dur=10.0*)

Turn off the light for *dur* seconds

dur [float, optional] The amount of time (in seconds) to turn off the light.

Returns Timestamp of the timeout and the timeout duration

Return type (datetime, float)

class pyoperant.components.**PeckPort**(*IR*, *LED*, *inverted=False*, **args*, ***kwargs*)

Bases: *pyoperant.components.BaseComponent*

Class which holds information about peck ports

Parameters

- **LED** (*hwio.BooleanOutput*) – output channel to activate the LED in the peck port
- **IR** (*hwio.BooleanInput*) – input channel for the IR beam to check for a peck

LED

hwio.BooleanOutput

output channel to activate the LED in the peck port

IR

hwio.BooleanInput

input channel for the IR beam to check for a peck

flash(*dur=1.0*, *isi=0.1*)

Flashes the LED on and off with *isi* seconds high and low for *dur* seconds, then revert LED to prior state.

Parameters

- **dur** (*float, optional*) – Duration of the light flash in seconds.
- **isi** (*float,optional*) – Time interval between toggles. (0.5 * period)

Returns Timestamp of the flash and the flash duration

Return type (datetime, float)

off()
Turns the LED off

Returns True if successful

Return type bool

on(*val=100.0*)
Turns the LED on

Returns True if successful

Return type bool

poll(*timeout=None*)
Polls the peck port until there is a peck

Returns Timestamp of the IR beam being broken.

Return type datetime

status()
reads the status of the IR beam

Returns True if beam is broken

Return type bool

class pyoperant.components.**RGBLight**(*red*, *green*, *blue*, **args*, ***kwargs*)
Bases: *pyoperant.components.BaseComponent*

Class which holds information about an RGB cue light

red [hwio.BooleanOutput] output channel for the red LED

green [hwio.BooleanOutput] output channel for the green LED

blue [hwio.BooleanOutput] output channel for the blue LED

blue()
Turns the cue light to blue

Returns *True* if successful.

Return type bool

green()
Turns the cue light to green

Returns *True* if successful.

Return type bool

off()
Turns the cue light off

Returns *True* if successful.

Return type bool

red()
Turns the cue light to red

Returns *True* if successful.

Return type bool

2.1.2.2 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

2.1.2.3 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

callback(*func*)

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

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

Class which abstracts the writing to PWM outputs

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)

config()

read()
read status

toggle()
flip value

write(*val=0.0*)
write status

2.1.2.4 pyoperant.local module

2.1.2.5 pyoperant.local_vogel module

class pyoperant.local_vogel.**Vogel1**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel1 panel

class pyoperant.local_vogel.**Vogel2**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel2 panel

class pyoperant.local_vogel.**Vogel3**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel3 panel

class pyoperant.local_vogel.**Vogel4**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel4 panel

class pyoperant.local_vogel.**Vogel6**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel6 panel

class pyoperant.local_vogel.**Vogel7**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel7 panel

class pyoperant.local_vogel.**Vogel8**
Bases: *pyoperant.local_vogel.VogelPanel*

Vogel8 panel

class pyoperant.local_vogel.**VogelPanel**(*id=None*, **args*, ***kwargs*)
Bases: *pyoperant.panels.BasePanel*

class for vogel boxes

reset()

test()

2.1.2.6 pyoperant.local_zog module

class pyoperant.local_zog.**Zog1**
Bases: *pyoperant.local_zog.ZogPanel*

Zog1 panel

class pyoperant.local_zog.**Zog10**
Bases: *pyoperant.local_zog.ZogCuePanel*

Zog10 panel

class pyoperant.local_zog.**Zog11**
Bases: *pyoperant.local_zog.ZogCuePanel*

Zog11 panel

class pyoperant.local_zog.**Zog12**
Bases: *pyoperant.local_zog.ZogCuePanel*

Zog12 panel

class pyoperant.local_zog.**Zog13**
Bases: *pyoperant.local_zog.ZogPanel*

Zog13 panel

class pyoperant.local_zog.**Zog14**
Bases: *pyoperant.local_zog.ZogPanel*

Zog14 panel

class pyoperant.local_zog.**Zog15**
Bases: *pyoperant.local_zog.ZogPanel*

Zog15 panel

class pyoperant.local_zog.**Zog16**
Bases: *pyoperant.local_zog.ZogPanel*

Zog16 panel

class pyoperant.local_zog.**Zog2**
Bases: *pyoperant.local_zog.ZogPanel*

Zog2 panel

class pyoperant.local_zog.**Zog3**
Bases: *pyoperant.local_zog.ZogPanel*

Zog3 panel

class pyoperant.local_zog.**Zog4**
Bases: *pyoperant.local_zog.ZogPanel*

Zog4 panel

class pyoperant.local_zog.**Zog5**
Bases: *pyoperant.local_zog.ZogCuePanel*

Zog5 panel

class pyoperant.local_zog.**Zog6**
Bases: *pyoperant.local_zog.ZogPanel*

Zog6 panel

class pyoperant.local_zog.**Zog7**
Bases: *pyoperant.local_zog.ZogCuePanel*

Zog7 panel

class pyoperant.local_zog.**Zog8**
Bases: *pyoperant.local_zog.ZogPanel*

Zog8 panel

class pyoperant.local_zog.**Zog9**
Bases: *pyoperant.local_zog.ZogCuePanel*

Zog9 panel

class pyoperant.local_zog.**ZogAudioInterface**(**args*, ***kwargs*)
Bases: *pyoperant.interfaces.pyaudio_.PyAudioInterface*

docstring for ZogAudioInterface

validate()

class pyoperant.local_zog.**ZogCuePanel**(*id=None*)
Bases: *pyoperant.local_zog.ZogPanel*

ZogCuePanel panel

class pyoperant.local_zog.**ZogPanel**(*id=None*, **args*, ***kwargs*)
Bases: *pyoperant.panels.BasePanel*

class for zog boxes

reset()

test()

2.1.2.7 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()

2.1.2.8 pyoperant.queues module

class pyoperant.queues.**AdaptiveBase**(***kwargs*)

Bases: object

docstring for AdaptiveBase This is an abstract object for implementing adaptive procedures, such as a staircase. Importantly, any objects inheriting this need to define the *update()* and *next()* methods.

next()

no_response()

on_load()

update(*correct*, *no_resp*)

update_error_msg()

class pyoperant.queues.**DoubleStaircase**(*stims*, *rate_constant=0.05*, ***kwargs*)

Bases: *pyoperant.queues.AdaptiveBase*

Generates conditions from a list of stims that monotonically vary from most easily left to most easily right i.e. left is low and right is high

The goal of this queue is to estimate the 50% point of a psychometric curve.

This will probe left and right trials, if the response is correct, it will move the indices closer to each other until they are adjacent.

stims: an array of stimuli names ordered from most easily left to most easily right rate_constant: the step size is the rate_constant*(high_idx-low_idx)

next()

no_response()

update(*correct*, *no_resp*)

update_error_msg()

class pyoperant.queues.**DoubleStaircaseReinforced**(*stims*, *rate_constant=0.05*, *probe_rate=0.1*, *sample_log=False*, ***kwargs*)

Bases: *pyoperant.queues.AdaptiveBase*

Generates conditions as with DoubleStaircase, but 1-probe_rate proportion of the trials easier/known trials to reduce frustration.

Easier trials are sampled from a log shaped distribution so that more trials are sampled from the edges than near the indices

stims: an array of stimuli names ordered from most easily left to most easily right rate_constant: the step size is the rate_constant*(high_idx-low_idx) probe_rate: proportion of trials that are between [0, low_idx] or [high_idx, length(stims)]

next()

no_response()

on_load()

update(*correct*, *no_resp*)

update_error_msg()

class pyoperant.queues.**KaernbachStaircase**(*start_val=100*, *stepsize_up=3*, *stepsize_dn=1*, *min_val=0*, *max_val=100*, *crit=100*, *crit_method='trials'*)

Bases: *pyoperant.queues.AdaptiveBase*

generates values for a staircase procedure from Kaernbach 1991 This procedure returns values for each trial and assumes that larger values are easier. Thus, after a correct trial, the next value returned will be smaller and after incorrect trials, the next value returned will be larger. The magnitudes of these changes are stepsize_dn and stepsize_up, respectively. :param start_val: the starting value of the procedure (default: 100) :type start_val: float/int

Kwargs: stepsize_up (int): number of steps to take after incorrect trial (default: 3) stepsize_dn (int): number of steps to take after correct trial (default: 1) min_val (float): minimum parameter value to allow (default: 0) max_val (float): maximum parameter value to allow (default: 100) crit (int): minimum number of trials (default: 0) crit_method (int): maximum number of trials (default: 100)

Returns float

next()

update(*correct*, *no_resp*)

class pyoperant.queues.**MixedAdaptiveQueue**(*sub_queues*, *probabilities=None*, ***kwargs*)

Bases: *pyoperant.queues.PersistentBase*, *pyoperant.queues.AdaptiveBase*

Generates conditions from multiple adaptive sub queues.

Use the generator MixedAdaptiveQueue.load(filename, sub_queues) to load a previously saved MixedAdaptive-Queue or generate a new one if the pkl file doesn't exist.

sub_queues: a list of adaptive queues probabilities: a list of weights with which to sample from sub_queues

should be same length as sub_queues NotImplemented

filename: filename of pickle to save itself

next()

on_load()

update(*correct*, *no_resp*)

class pyoperant.queues.**PersistentBase**(*filename=None*, ***kwargs*)
Bases: object

A mixin that allows for the creation of an obj through a load command that first checks for a pickled file to load an object before generating a new one.

classmethod load(*filename*, **args*, ***kwargs*)

on_load()

save()

pyoperant.queues.**block_queue**(*conditions*, *reps=1*, *shuffle=False*)
generate trial conditions from a block

Parameters **conditions** (*list*) – The conditions to sample from.

Kwargs: reps (int): number of times each item in conditions will be presented (default: 1) shuffle (bool): Shuffles the queue (default: False)

Returns whatever the elements of 'conditions' are

pyoperant.queues.**random_queue**(*conditions*, *tr_max=100*, *weights=None*)
generator which randomly samples conditions

Parameters

- **conditions** (*list*) – The conditions to sample from.
- **weights** (*list of ints*) – Weights of each condition

Kwargs: tr_max (int): Maximum number of trial conditions to generate. (default: 100)

Returns whatever the elements of 'conditions' are

2.1.2.9 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.

2.1.2.10 pyoperant.utils module

class pyoperant.utils.**AuditoryStimulus**(**args*, ***kwargs*)
Bases: *pyoperant.utils.Stimulus*

docstring for AuditoryStimulus

class pyoperant.utils.**Command**(*command*)
Bases: object

Enables to run subprocess commands in a different thread with TIMEOUT option.

via https://gist.github.com/kirpit/1306188

Based on jcollado's solution: http://stackoverflow.com/questions/1191374/subprocess-with-timeout/4825933#4825933

command = None

error = ''

output = ''

process = None

run(*timeout=None*, ***kwargs*)
Run a command then return: (status, output, error).

status = None

class pyoperant.utils.**Event**(*time=None*, *duration=None*, *label=''*, *name=None*, *description=None*, *file_origin=None*, **args*, ***kwargs*)
Bases: object

docstring for Event

annotate(***kwargs*)

class pyoperant.utils.**NumpyAwareJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)
Bases: json.encoder.JSONEncoder

this json encoder converts numpy arrays to lists so that json can write them.

example usage:

```
>>> import numpy as np
>>> dict_to_save = {'array': np.zeros((5,))}
>>> json.dumps(dict_to_save,
               cls=NumpyAwareJSONEncoder
               )
'{"array": [0.0, 0.0, 0.0, 0.0, 0.0]}'
```

default(*obj*)
Implement this method in a subclass such that it returns a serializable object for o, or calls the base implementation (to raise a TypeError).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class pyoperant.utils.**Stimulus**(**args*, ***kwargs*)
Bases: *pyoperant.utils.Event*

docstring for Stimulus

class pyoperant.utils.**Trial**(*index=None*, *type_='normal'*, *class_=None*, **args*, ***kwargs*)
Bases: *pyoperant.utils.Event*

docstring for Trial

pyoperant.utils.**auditory_stim_from_wav**(*wav*)

pyoperant.utils.**check_cmdline_params**(*parameters*, *cmd_line*)

pyoperant.utils.**check_time**(*schedule*, *fmt='%H:%M'*)
determine whether trials should be done given the current time and the light schedule

returns Boolean if current time meets schedule

schedule='sun' will change lights according to local sunrise and sunset

schedule=[('07:00','17:00')] will have lights on between 7am and 5pm schedule=[('06:00','12:00'),('18:00','24:00')] will have lights on between

pyoperant.utils.**concat_wav**(*input_file_list*, *output_filename='concat.wav'*)
concat a set of wav files into a single wav file and return the output filename

takes in a tuple list of files and duration of pause after the file

input_file_list = [('a.wav', 0.1), ('b.wav', 0.09), ('c.wav', 0.0),]

returns a list of AuditoryStimulus objects

TODO: add checks for sampling rate, number of channels, etc.

pyoperant.utils.**get_num_open_fds**()
return the number of open file descriptors for current process

pyoperant.utils.**is_day**(*latitude='32.82'*, *longitude='-117.14'*)
Is it daytime?

(lat,long) – latitude and longitude of location to check (default is San Diego) Returns True if it is daytime

pyoperant.utils.**parse_commandline**(*arg_str=['-b', 'latex', '-D', 'language=en', '-d', '_build/doctrees', '.', '_build/latex']*)
parse command line arguments note: optparse is depreciated w/ v2.7 in favor of argparse

pyoperant.utils.**rand_from_log_shape_dist**(*alpha=10*)
randomly samples from a distribution between 0 and 1 with pdf shaped like the log function low probability of getting close to zero, increasing probability going towards 1 alpha determines how sharp the curve is, higher alpha, sharper curve.

pyoperant.utils.**run_state_machine**(*start_in='pre'*, *error_state=None*, *error_callback=None*, ***state_functions*)
runs a state machine defined by the keyword arguments

```
>>> def run_start():
>>>     print "in 'run_start'"
>>>     return 'next'
>>> def run_next():
>>>     print "in 'run_next'"
>>>     return None
>>> run_state_machine(start_in='start',
>>>                   start=run_start,
>>>                   next=run_next)
in 'run_start'
in 'run_next'
None
```

pyoperant.utils.**time_in_range**(*start*, *end*, *x*)
Return true if x is in the range [start, end]

pyoperant.utils.**wait**(*secs=1.0*, *final_countdown=0.0*, *waitfunc=None*)
Smartly wait for a given time period.

secs – total time to wait in seconds final_countdown – time at end of secs to wait and constantly poll the clock waitfunc – optional function to run in a loop during hogCPUperiod

If secs=1.0 and final_countdown=0.2 then for 0.8s python's time.sleep function will be used, which is not especially precise, but allows the cpu to perform housekeeping. In the final hogCPUsecs the more precise method of constantly polling the clock is used for greater precision.

2.1.3 Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z