Autopilot Documentation

Release 0.3.0

Jonny Saunders

Aug 18, 2021

USER GUIDE:

Autopilot is a Python framework to perform behavioral experiments with one or many Raspberry Pis.

Its distributed structure allows arbitrary numbers and combinations of hardware components to be used in an experiment, allowing users to perform complex, hardware-intensive experiments at scale.

Autopilot integrates every part of your experiment, including hardware operation, task logic, stimulus delivery, data management, and visualization of task progress – making experiments in behavioral neuroscience replicable from a single file.

Instead of rigid programming requirements, Autopilot attempts to be a flexible framework with many different modalities of use in order to adapt to the way you do and think about your science rather than the other way around. Use only the parts of the framework that are useful to you, build on top of it with its plugin system as you would normally, while also maintaining the provenance and system integration that more rigid systems offer.

For developers of other tools, Autopilot provides a skeleton with minimal assumptions to integrate their work with its broader collection of tools, for example our integration of DeepLabCut-live as the `DLC` transform ([KLS+20]).

Our long-range vision is to build a tool that lowers barriers to tool use and contribution, from code to contextual technical knowledge, so our broad and scattered work can be cumulatively combined without needing a centralized consortium or adoption of a singular standard.

For a detailed overview of Autopilot's motivation, design, and structure, see our whitepaper.

What's New v0.4.0 - Become Multifarious (21-08-03)

- The Autopilot Wiki is live!!!! The wiki will be the means of gathering and sharing knowledge about using Autopilot, but it will also serve as an additional tool for building interfaces and decentralizing control over its development. Head to the *changelog* or the *Plugins & The Wiki* page to learn more
- Autopilot `plugins` are now live!!! Anything in your plugin directory is a plugin, extend most types of autopilot classes to implement your own custom hardware and tasks and anything else without modifying autopilot itself, then submit it to the wiki to make it immediately available to everyone who uses the system! Link it to all the rest of your work, the parts it uses, let's make a knowledge graph!!!
- Tests and Continuous Integration are finally here!!! if there has been anything I have learned over the past few projects is that tests are god. Ours are hosted on travis and we are currently on the board with a stunning 27% coverage at coveralls.io
- Lots of new hardware and transform classes! Take a look! `cameras.PiCamera`, `timeseries.Kalman`, `geometry.IMU_Orientation`, `timeseries.Filter_IIR`, `timeseries.Integrate`, `geometry.Rotate`, `geometry.Spheroid`
- Major improvements like stereo sound (Thanks Chris Rodgers !), multihop messages, direct messaging, programmatic setup... see more in the changelog <changelog_v040
- Continued work on deconvoluting and remodularating all the code structure!
- Removed limits on python version, now testing on 3.7, 3.8, and 3.9

This documentation is very young and is very much a work in progress! Please submit an issue with any incompletenesses, confusion, or errors!

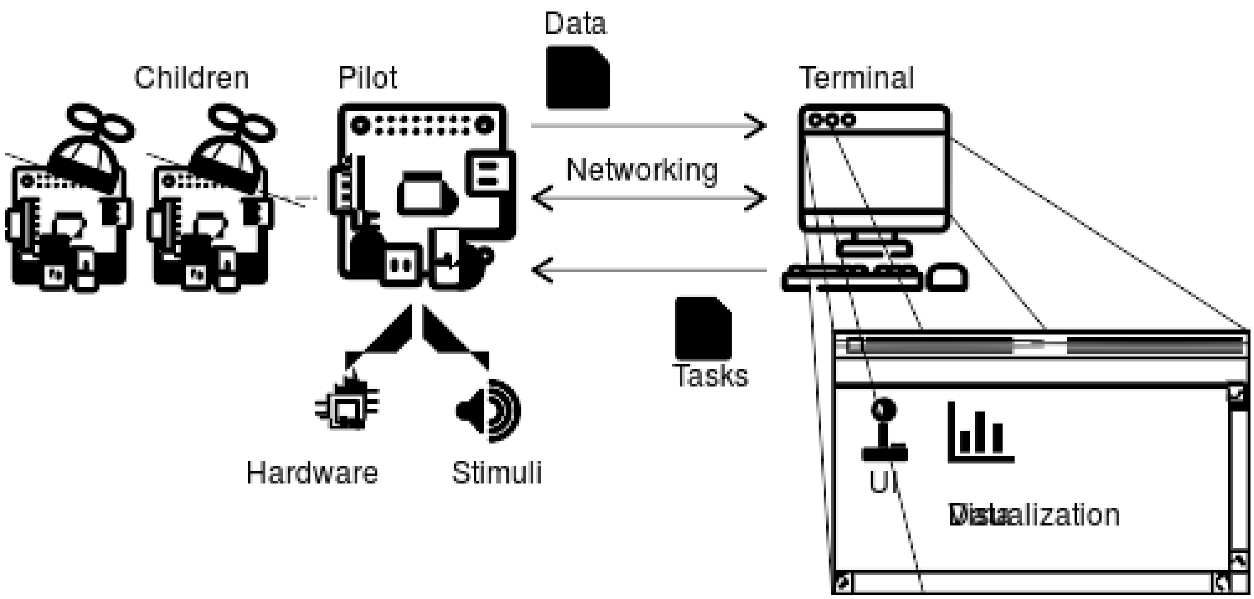
Todo: This page is still under construction! For a more detailed description, see the whitepaper, particularly "Program Structure"

https://www.biorxiv.org/content/10.1101/807693v1

CHAPTER

ONE

PROGRAM STRUCTURE



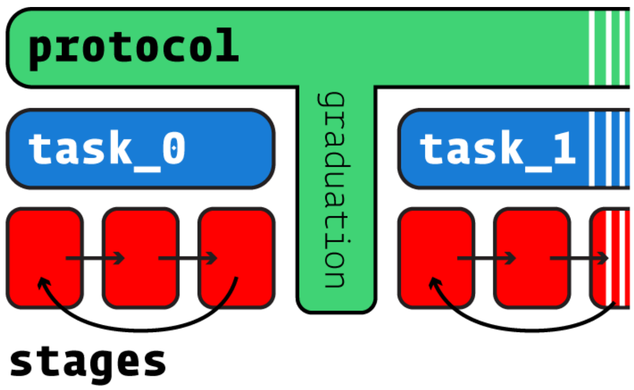
Autopilot performs experiments by distributing them over a network of desktop computers and Raspberry Pis. Each Computer or Pi runs an Autopilot **agent**, like the user-facing *Terminal* or a Raspberry Pi *Pilot* .

The *Terminal* agent provides a *gui* to operate the system, manage *Subject* s and experimental protocols, and *plots* for visualizing data from ongoing experiments.

Each *Terminal* manages a swarm of *Pilot* s that actually perform the experiments. Each *Pilot* coordinates *hardware* and *stim* uli in a *Task*. *Pilot* s can, in turn, coordinate their own swarm of networked `Children` that can manage additional hardware components – allowing *Task* s to use effectively arbitrary numbers and combinations of hardware.

CHAPTER TWO

TASKS



Behavioral experiments in Autopilot consist of *Task* s. Tasks define the parameters, coordinate the hardware, and perform the logic of an experiment.

Tasks may consist of one or multiple **stages**, completion of which constitutes a **trial**. Stages are analogous to states in a finite-state machine, but don't share their limitations: Tasks can use arbitrary transitions between stages and have computation or hardware operation persist between stages.

Multiple Tasks can be combined to make **protocols**, in which subjects move between different tasks according to *graduation* criteria like accuracy or number of trials. Protocols can thus be used to automate shaping routines that introduce a subject to the experimental apparatus and task structure.

For more details on tasks, see guide_task

CHAPTER THREE

MODULE TOUR

Todo: A more comprehensive overview is forthcoming, but the documentation for the most important modules can be found in the API documentation. A short tour for now...

- `Terminal` - user facing agent class used to control and configure program operation. See setup_terminal and `setup.setup_terminal`
- `gui` - GUI classes built with PySide2/Qt5 used by the terminal
- `plots` - Classes to plot data from ongoing tasks
- `pilot` - Experimental agent that runs tasks on Raspberry Pis
- `networking` - Networking modules used for communication between agents, tasks, and hardware objects
- `subject` - Data and metadata storage
- `hardware` - Hardware objects that can be used in tasks
- `tasks` - Customizable and extendable Task templates
- `stim` - Stimulus generation & presentation, of which sound is currently the most heavily developed

CHAPTER

FOUR

QUICKSTART

Autopilot is an integrated system for coordinating all parts of an experiment, but it is also designed to be permissive about how it is used and to make transitioning from existing lab tooling gentler – so its modules can be used independently.

To get a sample of autopilot, you can check out some of its modules without doing a fully configured *Installation* . As you get more comfortable using Autopilot, adopting more of its modules and usage patterns makes integrating each of the separate modules simpler and more powerful, but we'll get there in time.

4.1 Minimal Installation

Say you have a Raspberry Pi with Raspbian installed . Install autopilot and its basic system dependencies & configuration like this:

```
pip3 install auto-pi-lot
python3 -m autopilot.setup.run_script env_pilot pigpiod
```

4.2 Blink an LED

Say you connect an LED to one of the `gpio` pins - let's say (board numbered) pin 7. Love 7. Great pin.

Control the LED by using the `gpio.Digital_Out` class:

```
from autopilot.hardware.gpio import Digital_Out
led = Digital_Out(pin=7)

# turn it on!
led.set(1)

# turn if off!
led.set(0)
```

Or, blink "hello" in morse code using `series()` !

```
letters = [
    ['dot', 'dot', 'dot', 'dot'],  # h
    ['dot'],                       # e
    ['dot', 'dash', 'dot', 'dot'], # l
    ['dot', 'dash', 'dot', 'dot'], # l
```

(continues on next page)

(continued from previous page)

```
    ['dash', 'dash', 'dash']       # o
]
# make a series of 1's and 0's, which will last for the time_unit
times = {'dot': [1, 0], 'dash': [1, 1, 1, 0], 'space':[0]*3}
binary_letters = []
for letter in letters:
    binary_letters.extend([value for char in letter for value in times[char]])
    binary_letters.extend(times['space'])

time_unit = 100 #ms
led.series(id='hello', values=binary_letters, durations=time_unit)
```

4.3 Capture Video

Say you have a Raspberry Pi Camera Module , capture some video! First make sure the camera is enabled:

```
python3 -m autopilot.setup.run_script picamera
```

and then capture a video with *cameras.PiCamera* and write it to `test_video.mp4`:

```
from autopilot.hardware.cameras import PiCamera
cam = PiCamera()
cam.write('test_video.mp4')
cam.capture(timed=10)
```

Note: Since every hardware object in autopilot is by default nonblocking (eg. work happens in multiple threads, you can make other calls while the camera is capturing, etc.), this will work in an interactive python session but would require that you `sleep` or call `cam.stoppping.join()` or some other means of keeping the process open.

While the camera is capturing, you can access its current frame in its `frame` attribute, or to make sure you get every frame, by calling *queue()* .

4.4 Communicate Between Computers

Synchronization and coordination of code across multiple computers is a very general problem, and an increasingly common one for neuroscientists as we try to combine many hardware components to do complex experiments.

Say our first raspi has an IP address `192.168.0.101` and we get another raspi whose IP is `192.168.0.102` . We can send messages between the two using two `networking.Net_Node` s. `networking.Net_Node` s send messages with a `key` and `value` , such that the `key` is used to determine which of its `listens` methods/functions it should call to handle `value` .

For this example, how about we make pilot 1 ping pilot 2 and have it respond with the current time?

On pilot 2, we make a node that listens for messages on port 5000. The `upstream` and `port` arguments here don't matter since this node doesn't initiate any connection, just received them (we'll use a global variable here and hardcode the return id since we're in scripting mode, but there are better ways to do this in autopilot proper):

```
from autopilot.networking import Net_Node
from datetime import datetime
global node_2

def thetime(value):
    global node_2
    node_2.send(
        to='pilot_1', key='THETIME',
        value=datetime.now().isoformat()
    )

node_2 = Net_Node(
    id='pilot_2', router_port=5000, upstream='', port=9999,
    listens={'WHATIS':thetime}
)
```

On pilot 1, we can then make a node that connects to pilot 2 and prints the time when it receives a response:

```
from autopilot.networking import Net_Node

node_1 = Net_Node(
    id='pilot_1', upstream='pilot_2',
    port=5000, upstream_ip = '192.168.0.102',
    listens = {'THETIME':print}
)

node_1.send(to='pilot_1', key='WHATIS')
```

4.5 Realtime DeepLabCut

Autopilot integrates DeepLabCut-Live [KLS+20] ! You can use your own pretrained models (stored in your autopilot user directory under */dlc*) or models from the Model Zoo .

Now let's say we have a desktop linux machine with DeepLabCut and dlc-live installed. DeepLabCut-Live is implemented in Autopilot with the `transform.image.DLC` object, part of the `transform` module.

First, assuming we have some image `img` (as a numpy array), we can process the image to get an array of x,y positions for each of the tracked points:

```
from autopilot import transform as t
import numpy as np

dlc = t.image.DLC(model_zoo='full_human')
points = dlc.process(img)
```

Autopilot's transform module lets us compose multiple data transformations together with + to make deploying chains of computation to other computers. How about we process an image and determine whether the left hand in the image is raised above the head?:

```
# select the two body parts, which will return a 2x2 array
dlc += t.selection.DLCSlice(select=('wrist1', 'forehead'))

```

(continues on next page)

(continued from previous page)

```
# slice out the 1st column (y) with a tuple of slice objects
dlc += t.selection.Slice(select=(
    slice(start=0,stop=2),
    slice(start=1,stop=2)
))

# compare the first (wrist) y position to the second (forehead)
dlc += t.logical.Compare(np.greater)

# use it!
dlc.process(img)
```

4.6 Put it Together - Close a Loop!

We've tried a few things, why not put them together?

Let's use our two raspberry pis and our desktop GPU-bearing computer to record a video of someone and turn an LED on when their hand is over their head. We could do this two (or one) computer as well, but let's be extravagant.

Let's say **pilot 1, pilot 2, and the gpu computer** have ip addresses of `192.168.0.101`, `192.168.0.102`, and `192.168.0.103`, respectively.

4.6.1 Pilot 1 - Image Capture

On **pilot 1**, we configure our *PiCamera* to stream to the gpu computer. While we're at it, we might as well also save a local copy of the video to watch later. The camera won't stop capturing, streaming, or writing until we call *capture()*:

```
from autopilot.hardware.cameras import PiCamera
cam = PiCamera()
cam.stream(to='gpu', ip='192.168.0.103', port=5000)
cam.write('cool_video.mp4')
```

4.6.2 GPU Computer

On the **gpu computer**, we need to receive frames, process them with the above defined transformation chain, and send the results on to **pilot 2**, which will control the LED. We could do this with the objects that we've already seen (make the transform object, make some callback function that sends a frame through it and give it to a `Net_Node` as a `listen` method), but we'll make use of the *Transformer* "child" object – which is a peculiar type of `Task` designed to perform some auxiliary function in an experiment.

Rather than giving it an already-instantiated transform object, we instead give it a schematic representation of the transform to be constructed – When used with the rest of autopilot, this is to both enable it to be dispatched flexibly to different computers, but also to preserve a clear chain of data provenance by keeping logs of every parameter used to perform an experiment.

The *Transformer* class uses *make_transform()* to reconstitute it, receives messages containing data to process, and then forwards them on to some other node. We use its `trigger` mode, which only sends the value on to the final recipient with the key `'TRIGGER'` when it changes.:

```
from autopilot.tasks.children import Transformer
import numpy as np

transform_description = [
    {
        "transform": "DLC",
        "kwargs": {'model_zoo':'full_human'}
    },
    {
        "transform": "DLCSlice",
        "kwargs": {"select": ("wrist1", "forehead")}
    }
    {
        "transform": "Slice",
        "kwargs": {"select":(
            slice(start=0,stop=2),
            slice(start=1,stop=2)
        )}
    },
    {
        "transform": "Compare",
        "args": [np.greater],
    },
]

transformer = Transformer(
    transform = transform_description
    operation = "trigger",
    node_id = "gpu",
    return_id = 'pilot_2',
    return_ip = '192.168.0.102',
    return_port = 5001,
    return_key = 'TRIGGER',
    router_port = 5000
)
```

4.6.3 Pilot 2 - LED

And finally on **pilot 2** we just write a listen callback to handle the incoming trigger:

```
from autopilot.hardware.gpio import Digital_Out
from autopilot.networking.Net_Node

global led
led = Digital_Out(pin=7)

def led_trigger(value:bool):
    global led
    led.set(value)

node = Net_Node(
```

(continues on next page)

(continued from previous page)

```
    id='pilot_2', router_port=5001, upstream='', port=9999,
    listens = {'TRIGGER':led_trigger}
)
```

There you have it! Just start capturing on **pilot 1**:

```
cam.capture()
```

4.7 What Next?

The rest of Autopilot expands on this basic use by providing tools to do the rest of your experiment, and to make replicable science easy.

- write standardized experimental protocols that consist of multiple `Task` s linked by flexible *graduation* criteria
- extend the library to use your custom hardware, and make your work available to anyone with our *plugins* system integrated with the autopilot wiki
- Use our GUI that makes managing many experimental rigs simple from a single computer.

and so on. . .

CHAPTER FIVE

INSTALLATION

Autopilot must be installed on the devices running the Terminal and the Pilot agents. The Pilot runs on a Raspberry Pi (remember: Pi for "Pilot") and the Terminal runs on a regular desktop computer. So Autopilot must be installed on both. This document will show you how to do that.

5.1 Supported Systems

OS	• Pilot: raspiOS >=Buster (lite recommended) • Terminal: Ubuntu >=16.04
Python Version	3.7
Raspberry Pi	>=3b

Autopilot is **linux/mac** only, and supports **Python 3.7**. Some parts might accidentally work in Windows but we make no guarantees.

We have tried to take care to make certain platform-specific dependencies not break the entire package, so if you have some difficulty installing autopilot on a non-raspberry-pi linux machine please submit an issue!

5.2 Pre-installation

5.2.1 On the Pilot device

For Pilots, we recommend starting with a fresh Raspbian Lite image (see the raspi installation documentation). Note that the Lite image doesn't include a desktop environment or GUI, just a command-line interface, but that's all we need for the Pilot. It's easiest to connect a monitor and keyboard directly to the Pi while configuring it. Once it's configured, you won't need to leave the monitor and keyboard attached, and/or you can choose to connect to it with ssh.

After the Pi has been started up for the first time, run *sudo raspi-config* to do things like connect to a wifi network, set the time zone, and so on. It's very important to change the password for the *pi* user account to a new one of your choice so that you don't get hacked, especially if you're opening up ssh access.

It's also best to update the Pi's operating system at this time:

```
sudo apt update
sudo apt upgrade -y
```

Now install the system packages that are required by Autopilot. You can do this by running this command, or it's also available as a setup script in the guided installation of Autopilot. (`python -m autopilot.setup.run_script env_pilot`)

```
sudo apt install -y \
    python3-dev \
    python3-pip \
    git \
    libatlas-base-dev \
    libsamplerate0-dev \
    libsndfile1-dev \
    libreadline-dev \
    libasound-dev \
    i2c-tools \
    libportmidi-dev \
    liblo-dev \
    libhdf5-dev \
    libzmq-dev \
    libffi-dev
```

5.2.2 On the Terminal device

The following system packages are required by `PySide2` (which no longer packages `xcb`):

```
sudo apt-get update && \
sudo apt-get install -y \
  libxcb-icccm4 \
  libxcb-image0 \
  libxcb-keysyms1 \
  libxcb-randr0 \
  libxcb-render-util0 \
  libxcb-xinerama0 \
  libxcb-xfixes0
```

5.2.3 Creating a Virtual Environment

We recommend using autopilot within a virtual environment – we primarily develop with *virtualenv* but *conda* also appears to work.

First, install *virtualenv* (see the virtualenv docs):

```
pip3 install virtualenv
```

Then, create a venv. By convention, these virtual environments are stored in the directory *~/.venv*, but they can be located anywhere.

With `virtualenv`:

```
mkdir ~/.venv
python3 -m virtualenv ~/.venv/autopilot
```

With conda:

```
conda create --name autopilot python=3.7
```

The virtual environment must be "activated" now and any time you work with autopilot (`setup_autopilot` will detect which venv it is run from and source it in the launch script).

With `virtualenv`:

```
source ~/.venv/autopilot/bin/activate
```

With conda:

```
conda activate autopilot
```

Either way, you should see that the command prompt begins with the string "(autopilot)". If you want to exit the virtual environment at any time, just type *deactivate*.

5.3 Installing Autopilot

Now we're ready to install Autopilot on both the Pilot and Terminal devices. Follow the same instructions on both the Pi and the computer.

5.3.1 Method 1: Installation with pip

If you're just taking a look at Autopilot, the easiest way to get started is to install with pip!

```
pip3 install auto-pi-lot
```

5.3.2 Method 2: Installation from source

If you want to start writing your own experiments and tinkering with Autopilot, we strongly recommend forking the repository and developing directly in the library so your gorgeous insights can be integrated later.

Clone the repository and install an "editable" version with *-e*, this makes it so python uses the source code in your cloned repository, rather than from the system/venv libraries.:

```
git clone https://github.com/wehr-lab/autopilot.git
cd autopilot
pip3 install -e .
```

Note: Depending on your permissions, eg. if you are not installing to a virtual environment, you may get a permissions error and need to install with the `--user` flag

Note: Development work is done on the `dev` branch, which may have additional features/bugfixes but is much less stable! To use it just `git checkout dev` from your repository directory.

5.4 Configuration

After installation, set Autopilot up! Autopilot comes with a "guided installation" process where you can select the actions you want and they will be run for you. The setup routine will:

- install needed system packages
- prepare your operating system and environment
- set system preferences
- create a user directory (default `~/autopilot`) to store prefs, logs, data, etc.
- create a launch script

To start the guided process, run the following line.

```
python3 -m autopilot.setup
```

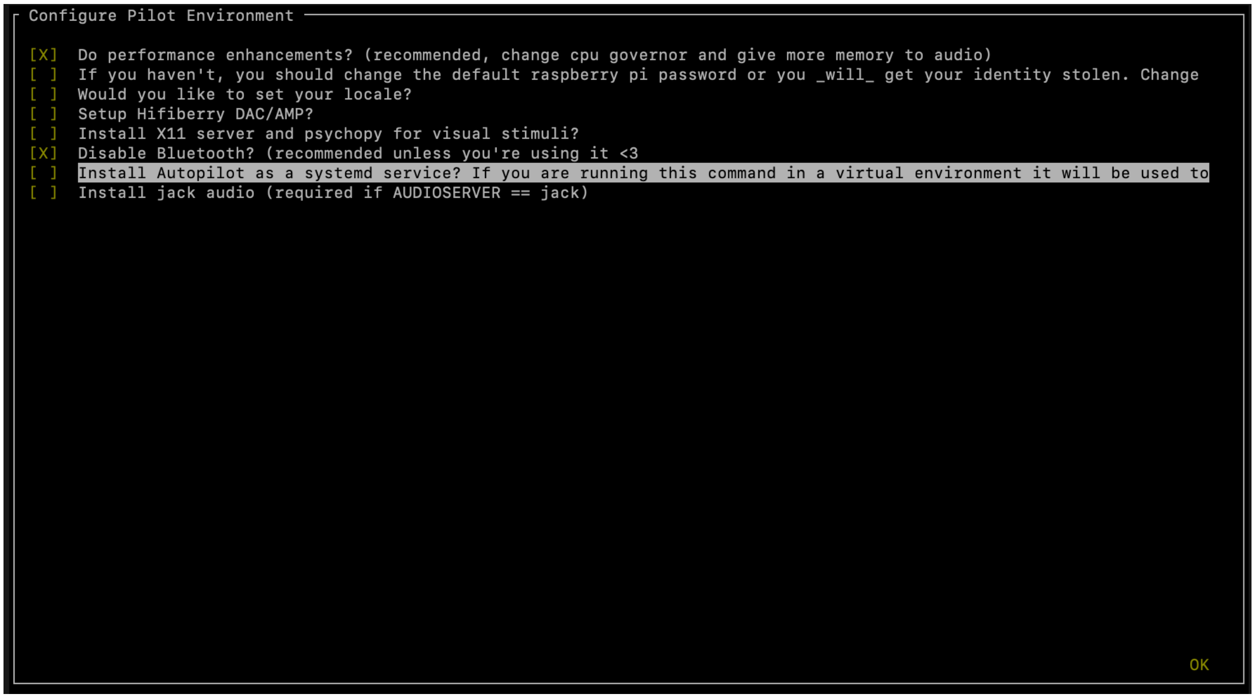
5.4.1 Select agent

Each runtime of Autopilot is called an "Agent", each of which performs different roles within a system, and thus have different requirements. If you're running the setup script on the Pi, select "Pilot". If you're running the setup script on a desktop computer, select "Terminal". If you're configuring multiple Pis, then select "Child" on the child Pis. Then hit "OK".

You can navigate this interface with the arrow keys, tab key, and enter key.

5.4.2 Select scripts

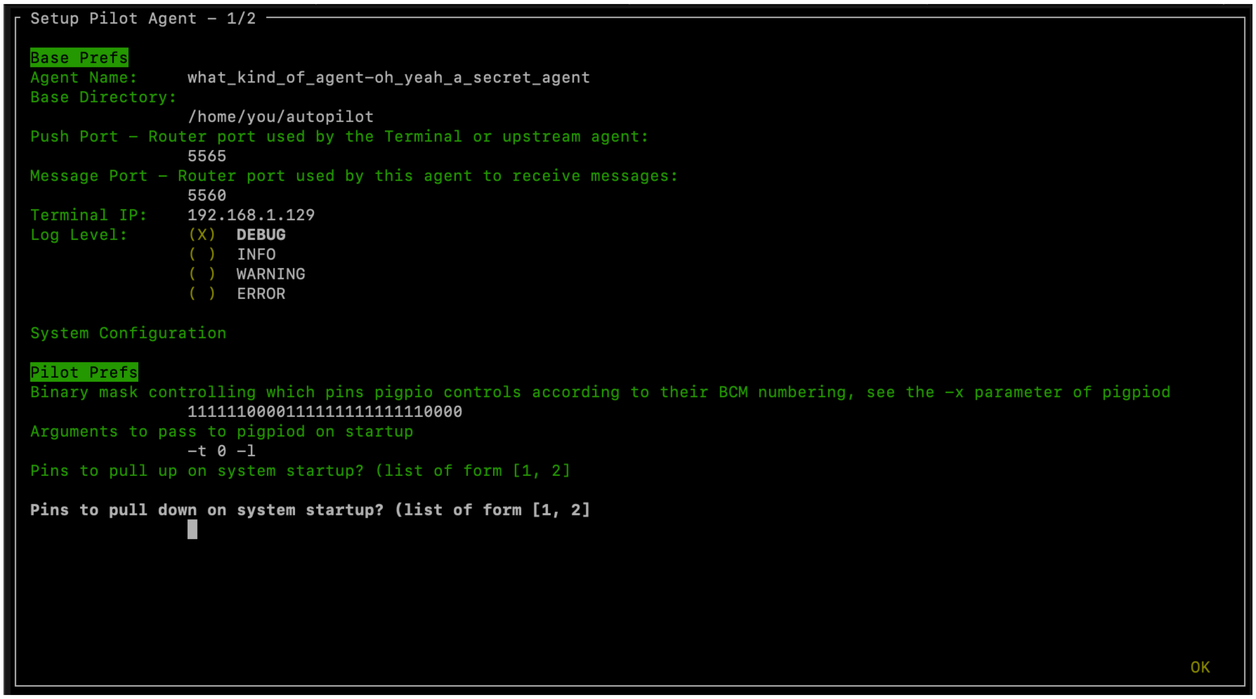
Now you will see a menu of potential scripts that can be run. Select the scripts you want to run, and then hit "OK". Note that even the simplest task ("free water") requires pigpio, so you may want to include that one. You can see the commands that will be run in each of these scripts with *setup.run_script* in the *setup.scripts.SCRIPTS* dictionary.



Note: Autopilot uses a slightly modified version of pigpio (https://github.com/sneakers-the-rat/pigpio) that allows it to get absolute timestamps (rather than system ticks) from gpio callbacks, increases the max number of scripts, etc. so if you have a different version of pigpio installed you will need to remove it and replace it with this one (you can do so with `python -m autopilot.setup.run_script pigpiod`

5.4.3 Configure Agent

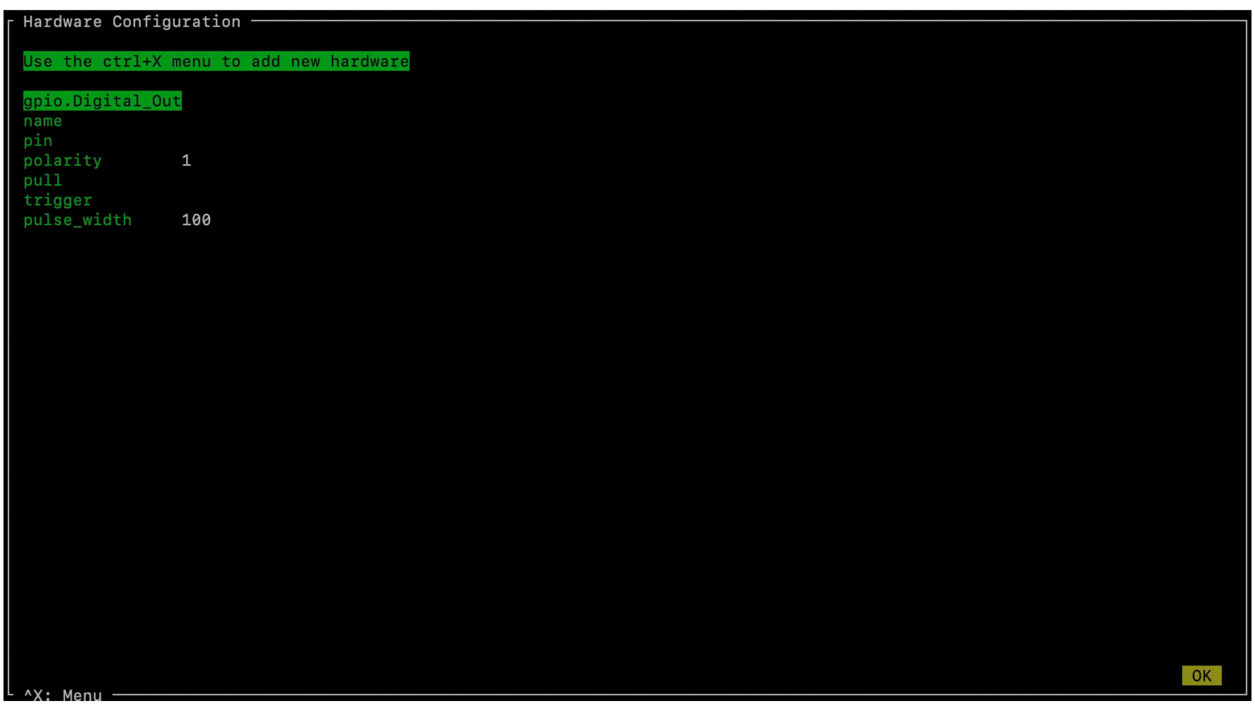
Each agent has a set of systemwide preferences stored in <AUTOPILOT_DIR>/prefs.json and accessible from *autopilot.prefs*.



5.4.4 Configure Hardware

If configuring a Pilot, you'll be asked to configure your hardware.

Press `ctrl+x` to add Hardware, and fill in the relevant parameters (most are optional and can be left blank). Consult the relevant page on the docs to see which arguments are relevant and how to use them.



After completing this step, the file *prefs.json* will be created if necessary and populated with the information you just provided. If it already exists, it will modified with the new information while preserving the previous preferences.

You can also manually edit the prefs.json file if you prefer. A template version for the Pilot is available that defines the ports, LEDs, and solenoids that are necessary for the "free water" task, which may be a useful way to get started.

5.5 Networking

Note: Networking is a point of major future development, particularly how agents discover one another and how ports are assigned. Getting networking to work is still a bit cumbersome, but you can track progress or contribute to improving networking at issue #48

5.5.1 IP Addresses

Pilots connect to a terminal whose IP address is specified as `TERMINALIP` in `prefs.json`

The Pilot and Terminal devices must be on the same network and capable of reaching one another. You must first figure out the IP address of each device with this command:

```
ipconfig
```

Let's say your Terminal is at 192.168.1.42 and your Pilot is at 192.168.1.200. Replace these values with whatever you actually find using *ipconfig*.

Then, you can test that each device can see the other with ping. On the Terminal, run:

```
ping 192.168.1.200
```

And on the Pilot, run:

```
ping 192.168.1.42
```

If that doesn't work, there is something preventing the computers from communicating from one another, typically this is the case if the computers are on university/etc. internet that makes it difficult for devices to connect to one another. We recommend networking agents together using a local router or switch (though some have reported being able to use their smartphone's hotspot in a pinch).

5.5.2 Ports

Agents use two prefs to configure their ports

- `MSGPORT` is the port that the agent receives messages on
- `PUSHPORT` is the port of the 'upstream' agent that it connects to.

So, if connecting a Pilot to a Terminal, the `PUSHPORT` of the Pilot should match the `MSGPORT` of the Terminal.

Ports need to be "open," but the central operation of a firewall is to "close" them. To open a port if, for example, you are using `ufw` on ubuntu (replacing with whatever port you're trying to open to whatever ip address):

```
sudo ufw allow from 192.168.1.200 to any port 5560
```

5.6 Testing the Installation

A launch script should have been created by *setup_autopilot* at <AUTOPILOT_DIR>/launch_autopilot.sh – this is the primary entrypoint to autopilot, as it allows certain system-level commands to precede launch (eg. activating virtual environments, enlarging shared memory, killing conflicting processes, launching an x server, etc.).

To launch autopilot:

```
~/autopilot/launch_autopilot.sh
```

Note: Selecting the script `alias` in *setup_autopilot* allows you to call the launch script by just typing `autopilot`

The actual launch call to autopilot resembles:

```
python3 -m autopilot.core.<AGENT_NAME> -f ~/autopilot/prefs.json
```

CHAPTER

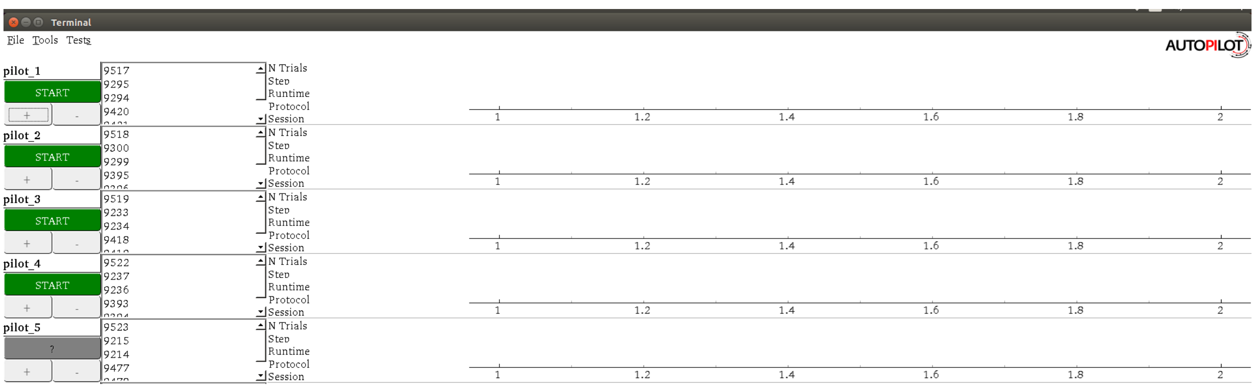
SIX

TRAINING A SUBJECT

After you have set up a Terminal and a Pilot, launch the Terminal.

6.1 Connecting the Pilot

If the TERMINAL_IP and port information is correctly set in the prefs.json file of the Pilot, it should automatically attempt to connect to the Terminal when it starts. It will send a handshake message that lets the Terminal know of its existence, its IP address, and its state. Once the Terminal receives its initial message, it will refresh, adding an entry to its pilot_db.json file and displaying a control panel for the pilot.



If the Pilot is not automatically detected, a pilot can be manually added with its name and IP using the "New Pilot" command in the file menu.

6.2 Creating a Protocol

A Protocol is one or a collection of tasks which the subject can 'graduate' through based on configurable graduation criteria. Protocols are stored as .json files in the protocols directory within prefs.BASEDIR.

6.2.1 Using the Protocol Wizard

> **Warning:** The Protocol Wizard does not currently support any Reward type except `time`, and the stimulus specification widget is limited to specifying 'L'(eft) and 'R'(ight) sounds. This is related to the unification of the parameter structure in Autopilot 0.3 (see *To-Do*). Protocols can be edited after creation in the Protocol Wizard using the format examples in the manual protocol creation section below.

The Protocol Wizard allows you to build protocols using all the tasks in `autopilot.tasks` (specifically that are registered in the `TASK_LIST`). It extracts the *PARAMS* dictionary from each task class, adds a few general parameters, and allows the user to fill them.

For this example, we will create a protocol for a freely-moving two-alternative forced choice task[1] . This task has three 'nosepokes,' which consist of an IR break beam sensor, a solenoid, and an LED. The subject is supposed to poke in the center port to present a stimulus and begin a trial, and then report the identity of that stimulus category by poking in the nosepokes on either side. If the subject is correct, they are rewarded with water.

It is relatively challenging for an animal subject to learn this task without having a few beginning shaping steps that introduce it to the nature of the arena and the structure of the task. In this example we will program a three-step shaping regimen:

- **Step 1 - Free Water:** The subject will be rewarded for merely poking the IR sensor in order to let them know that in this universe water comes out of these particular holes in the wall
- **Step 2 - Request Rewards:** The task will operate as normal (stimuli are presented, etc.), but the subject will be rewarded for the initial center-poke as well as for a correct answer. This teaches them the temporal structure of the task – center first, then side ports.
- **Step 3 - Frequency Discrimination:** The final step of the protocol, the mouse is taught to respond left to a low-frequency tone and right to a high-frequency tone.

1. To start, select `New Protocol` from the 'file' menu.
2. Add a step from the list of tasks in the leftmost column by selecting it and pressing the '+' button. Here we are adding the Free Water step.
3. Specify the parameters for the task in the rightmost window – we give 20ms of water every time the subject poke, etc.

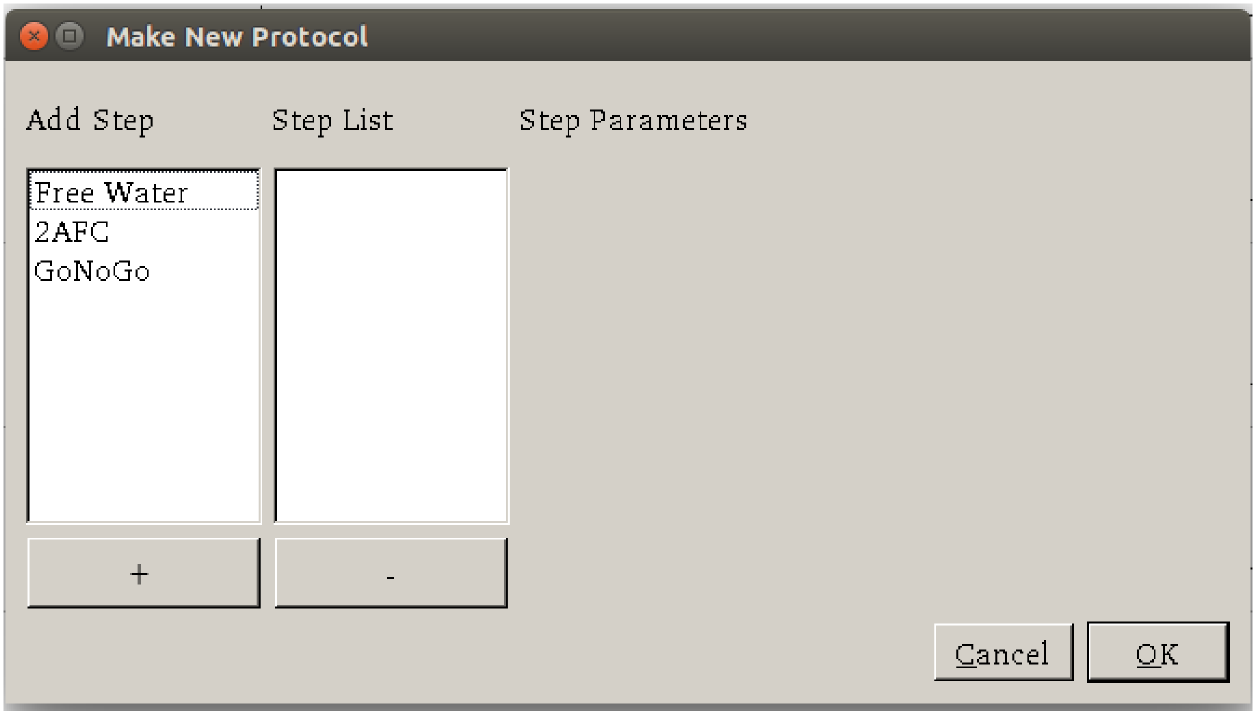
4. Add the second "Request Rewards" step, the remaining options that are configured are: * list * of * options

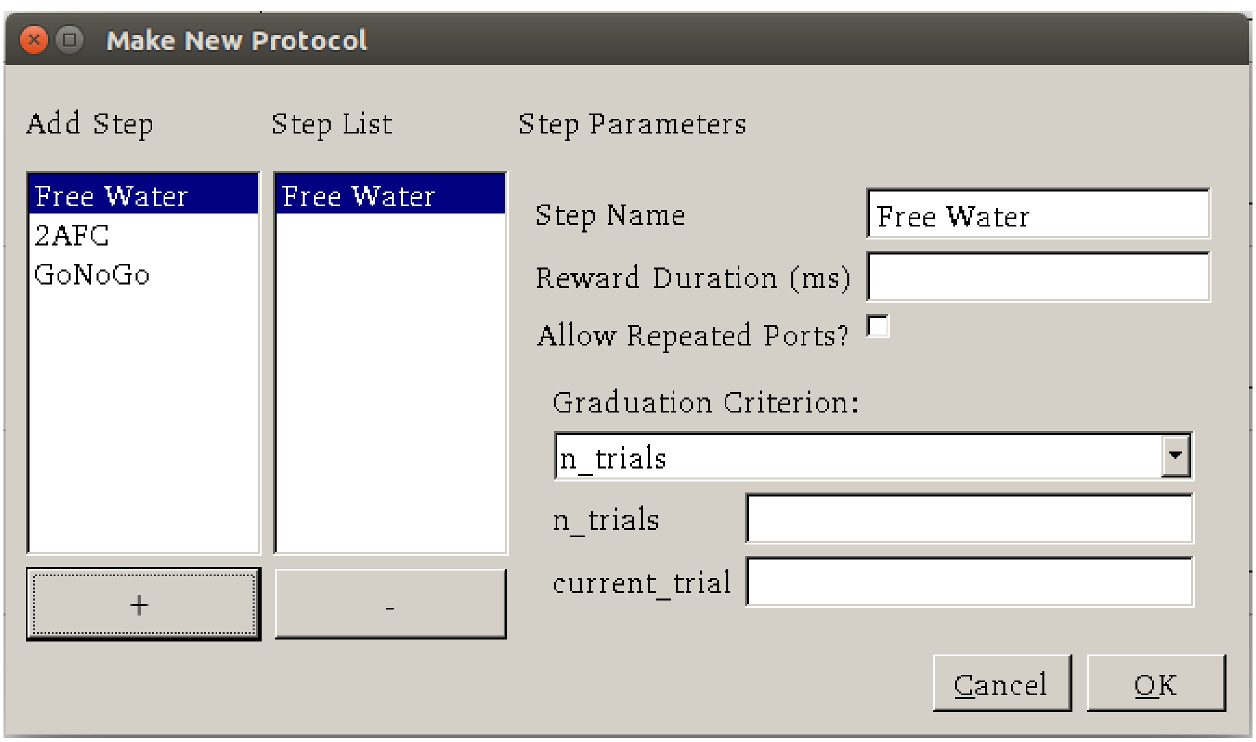
5. Press ok, save and name the protocol file.
6. That leaves us with a protocol file:

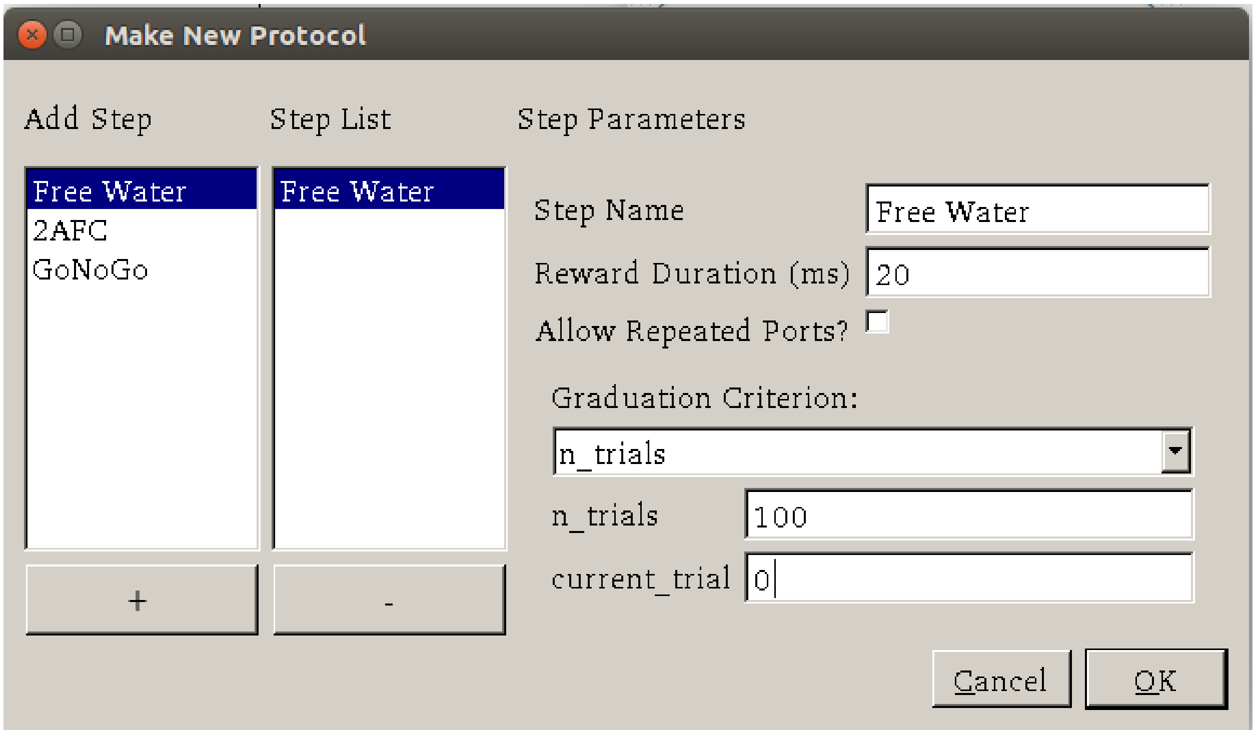
```
[
    {
        "allow_repeat": false,
        "graduation": {
            "type": "n_trials",
            "value": {
                "current_trial": "0",
                "n_trials": "100",
                "type": "n_trials"
            }
        },
```

(continues on next page)

[1] Yes we are aware that the "two-alternative forced choice" task described here is actually maybe called a "yes-no task" because there is only one stimulus presented at a time. The literature appears stuck with this term, however.


Make New Protocol
Add Step
Step List
Step Parameters
Free Water
2AFC
GoNoGo
+
-
Cancel
OK

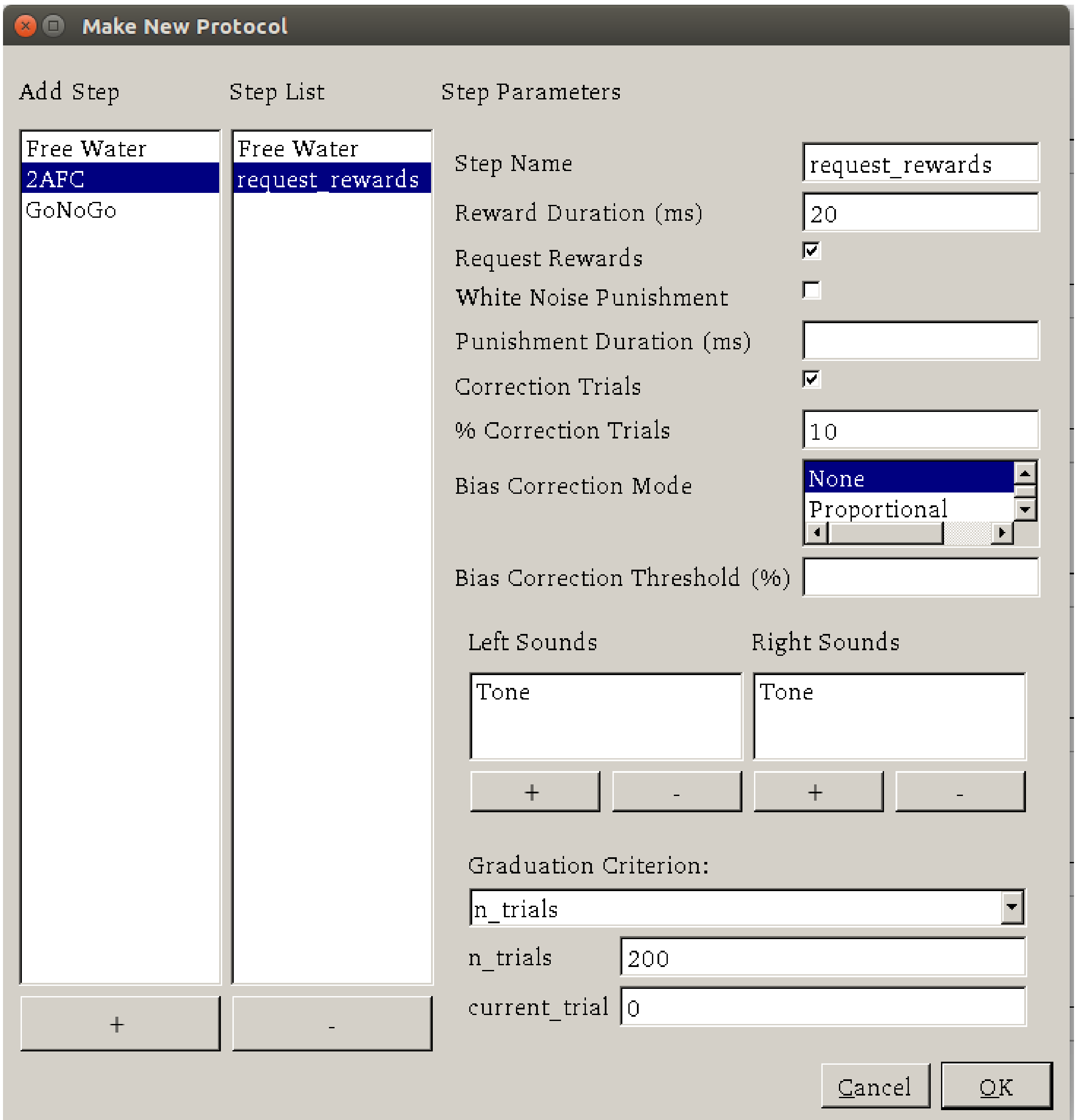

Make New Protocol
Add Step
Step List
Step Parameters
Free Water
2AFC
GoNoGo
Free Water
Step Name
Free Water
Reward Duration (ms)
Allow Repeated Ports?
Graduation Criterion:
n_trials
n_trials
current_trial
+
-
Cancel
OK



(continued from previous page)

```
        "reward": "20",
        "step_name": "Free Water",
        "task_type": "Free Water"
    },
    {
        "bias_mode": 0,
        "correction": true,
        "correction_pct": "10",
        "graduation": {
            "type": "n_trials",
            "value": {
                "current_trial": "0",
                "n_trials": "200",
                "type": "n_trials"
            }
        },
        "punish_stim": false,
        "req_reward": true,
        "reward": "20",
        "step_name": "request_rewards",
        "stim": {
            "sounds": {
                "L": [
                    {
                        "amplitude": "0.01",
                        "duration": "100",
                        "frequency": "4000",
```

(continues on next page)


Make New Protocol
Add Step
Step List
Step Parameters
Free Water
2AFC
GoNoGo
Free Water
request_rewards
Step Name
request_rewards
Reward Duration (ms)
20
Request Rewards
White Noise Punishment
Punishment Duration (ms)
Correction Trials
% Correction Trials
10
Bias Correction Mode
None
Proportional
Bias Correction Threshold (%)
Left Sounds
Right Sounds
Tone
Tone
+
-
+
-
Graduation Criterion:
n_trials
n_trials
200
current_trial
0
+
-
Cancel
OK

(continued from previous page)

```
                    "type": "Tone"
                }
            ],
            "R": [
                {
                    "amplitude": "0.01",
                    "duration": "100",
                    "frequency": "10000",
                    "type": "Tone"
                }
            ]
        },
        "tag": "Sounds",
        "type": "sounds"
    },
    "task_type": "2AFC"
},
{
    "bias_mode": 0,
    "correction": true,
    "correction_pct": "10",
    "graduation": {
        "type": "accuracy",
        "value": {
            "threshold": "80",
            "type": "accuracy",
            "window": "1000"
        }
    },
    "punish_stim": false,
    "req_reward": false,
    "reward": "20",
    "step_name": "2AFC",
    "stim": {
        "sounds": {
            "L": [
                {
                    "amplitude": "0.01",
                    "duration": "25",
                    "frequency": "100",
                    "type": "Tone"
                }
            ],
            "R": [
                {
                    "amplitude": "0.01",
                    "duration": "100",
                    "frequency": "100",
                    "type": "Tone"
                }
            ]
        },
```

(continues on next page)

(continued from previous page)

```
            "tag": "Sounds",
            "type": "sounds"
        },
        "task_type": "2AFC"
    }
]
```

6.2.2 Manual Protocol Creation

Protocols can be created manually by…

1. Extracting the task specific parameters, eg:

```
params = autopilot.tasks.Nafc.PARAMS
# for example...
params['param_1'] = value_1
```

2. Adding general task parameters `stim`, `reward`, `graduation`, `step_name`, and `task_type`. These are just examples, the `stim` and `reward` fields can be any parameters consumed by a `Reward_Manager` or `Stimulus_Manager`. The `graduation` field can be any parameters consumed by a *Graduation* object. The `step_name` and `task_type` need to be strings, the `task_type` corresponding to a key in the `TASK_LIST`.:

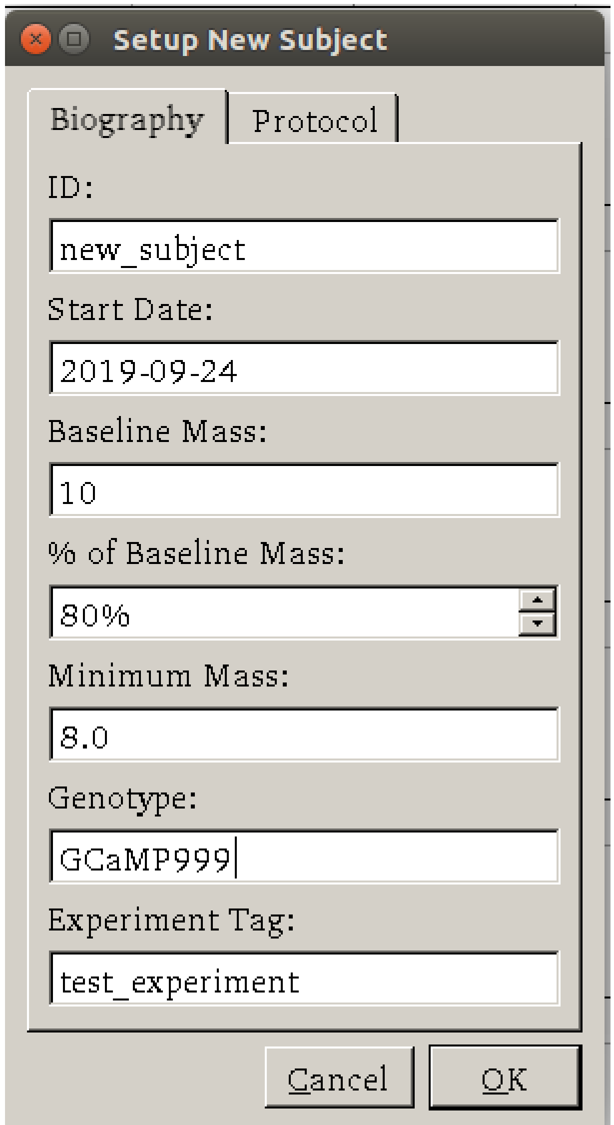
```
params.update({
    'stim': {
        'type': 'sounds',
        'sounds': {
            'L':[...],
            'R':[...]
        }
    },
    'reward': {
        'type': 'volume',
        'value': 2.5
    },
    'graduation': {
        'type': 'accuracy',
        'value': {
            'threshold': 0.8,
            'window': 1000
        }
    },
    'step_name': 'cool_new_step',
    'task_type': 'NAFC'
})
```

An example for our speech task can be found in `autopilot.tasks.protocol_scripts`.

6.3 Creating a Subject

A *Subject* stores the data, protocol, and history of a subject. Each subject is implicitly assigned to a Pilot by virtue of the structure of the `pilot_db.json` file, but they can be switched by editing that file.

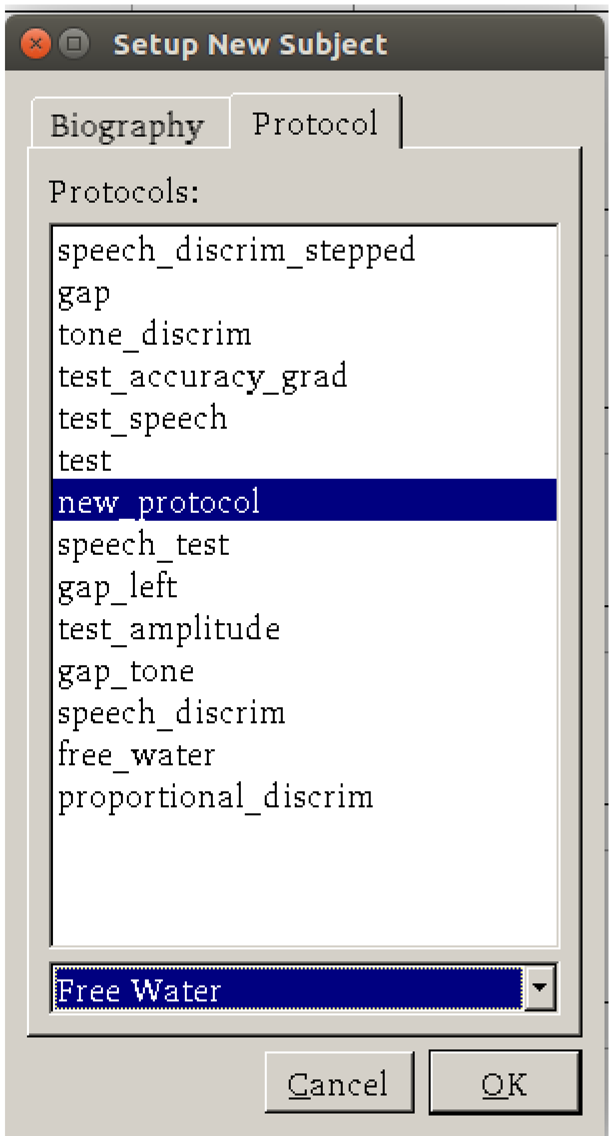
1. Create a subject by clicking the + button in the control panel of a particular Pilot
2. Fill out the basic biographical information

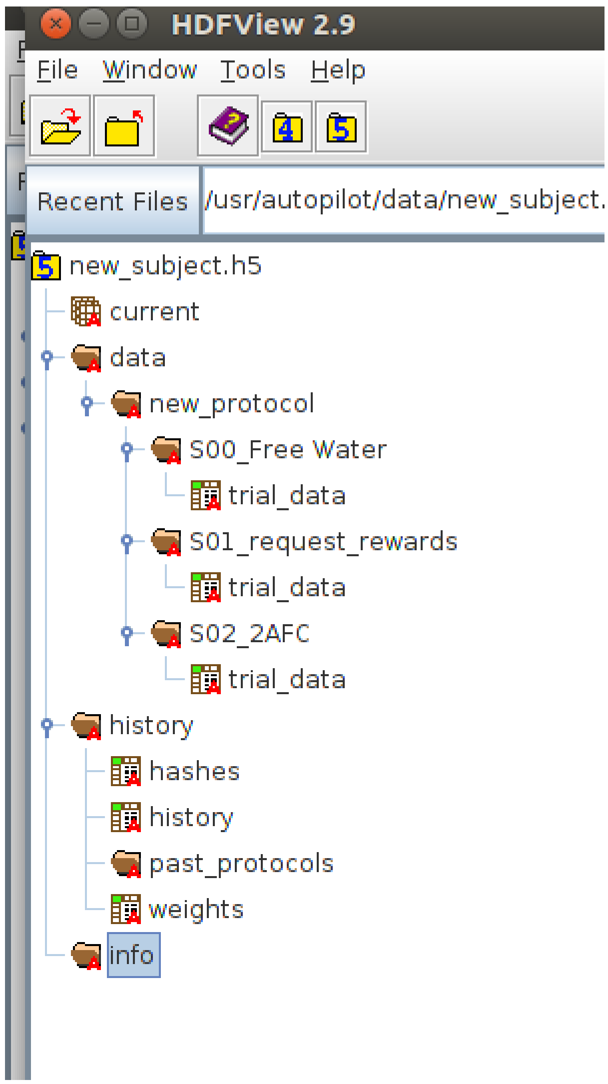


Todo: Currently the biographical fields are hardcoded in the Subject class. In the future we will allow users to create their own Subject schema where, for example, 'genotype' may not be as relevant.

3. Assign the subject to a protocol and step. Notice how the task we created earlier is here!

By creating one, we create an HDF5 file that stores a serialized version of the `.json` protocol file that was generated above, as well as the basic directory and table structure to enable the subject to store data from running the task.


Setup New Subject
Biography
Protocol
Protocols:
speech_discrim_stepped
gap
tone_discrim
test_accuracy_grad
test_speech
test
new_protocol
speech_test
gap_left
test_amplitude
gap_tone
speech_discrim
free_water
proportional_discrim
Free Water
Cancel
OK


HDFView 2.9
File Window Tools Help
Recent Files /usr/autopilot/data/new_subject.
new_subject.h5
current
data
new_protocol
S00_Free Water
trial_data
S01_request_rewards
trial_data
S02_2AFC
trial_data
history
hashes
history
past_protocols
weights
info

6.4 Running the Task

1. Select the subject's name and press the start button! The Terminal will send a `START` message to the Pilot that includes the parameter dictionary for the current step, and if the Pilot is configured with the hardware required in the `HARDWARE` dictionary of the task, it should run.
2. The Terminal will initialize the Pilot's plot using the parameters in the task's `PLOT` dictionary and display data as it is received.

6.5 Debugging a Task

If a Pilot doesn't start the task appropriately, if you have installed the Pilot as a system daemon you can retrieve the logs and see the stack trace by accessing the pilot via SSH:

```
ssh pi@your.pi.ip.address
```

Note: Because Raspberry Pis are common prey on the internet, we strongly advise changing the default password, installing RSA keys to access the pi, and disabling password access via SSH.

and then printing the end of the logs with `journalctl`:

```
# print the -end of the logs for system -unit autopilot
journalctl -u autopilot -e
```

Important: This guide and guide_hardware are lightly out of date with v0.4.0 of autopilot, but still largely reflect the program design and its operation. For a simpler task, see *Blink* .

Many of these things can be done more elegantly, more simply, etc. now but we are a very small team and can only do so much work between releases! We'd be happy to get documentation requests or even a pull request or two to help us out until we can get to it :)

CHAPTER

SEVEN

WRITING A TASK

Some concepts of task design are also discussed in section 3.1 of the whitepaper.

7.1 The Nafc Task

The `Nafc` class serves as an example for new task designs.

To demonstrate the general structure of Autopilot tasks, let's build it from scratch.

7.1.1 The `Task` class

We start by subclassing the *`Task`* class and initializing it.

```
from autopilot.tasks import Task

class Nafc(Task):

    def __init__(self):
        super(Nafc, self).__init__()
```

This gives our new task some basic attributes and methods, including the *`init_hardware()`* method for initializing the `HARDWARE` dictionary and the *`handle_trigger()`* method for handling GPIO triggers.

7.1.2 Four Task Attributes

We then add the four elements of a task description:

1. A **PARAMS** dictionary defines what parameters are needed to define the task
2. A **Data** (`tables.IsDescription`) descriptor describes what data will be returned from the task
3. A **PLOT** dictionary that maps the data output to graphical elements in the GUI.
4. A **HARDWARE** dictionary that describes what hardware will be needed to run the task.

PARAMS

Each parameter needs a human readable `tag` that will be used for GUI elements, and a `type`, currently one of:

- `int`: integers
- `bool`: boolean (checkboxes in GUI)
- `list`: list of possible values in {'Name':int} pairs
- `sounds`: a *autopilot.core.gui.Sound_Widget* to define sounds.

To maintain order when opened by the GUI we use a `odict` rather than a normal dictionary.

```
from collections import odict

PARAMS = odict()
PARAMS['reward']          = {'tag':'Reward Duration (ms)',
                             'type':'int'}
PARAMS['req_reward']      = {'tag':'Request Rewards',
                             'type':'bool'}
PARAMS['punish_stim']     = {'tag':'White Noise Punishment',
                             'type':'bool'}
PARAMS['punish_dur']      = {'tag':'Punishment Duration (ms)',
                             'type':'int'}
PARAMS['correction']      = {'tag':'Correction Trials',
                             'type':'bool'}
PARAMS['correction_pct'] = {'tag':'% Correction Trials',
                             'type':'int',
                             'depends':{'correction':True}}
PARAMS['bias_mode']       = {'tag':'Bias Correction Mode',
                             'type':'list',
                             'values':{'None':0,
                                       'Proportional':1,
                                       'Thresholded Proportional':2}}
PARAMS['bias_threshold'] = {'tag': 'Bias Correction Threshold (%)',
                             'type':'int',
                             'depends':{'bias_mode':2}}
PARAMS['stim']            = {'tag':'Sounds',
                             'type':'sounds'}
```

Note: See the *Nafc* class for descriptions of the task parameters.

These will be taken as key-value pairs when the task is initialized. ie.:

```
PARAMS['correction'] = {'tag':  'Correction Trials',
                        'type': 'bool'}
```

will be used to initialize the task like:

```
Nafc(correction=True) # or False
```

Data

There are two types of data,

- TrialData - where a single value for several variables is returned per 'trial', and
- ContinuousData - where values and timestamps are taken continuously, with either a fixed or variable interval

Both are defined by pytables tables.IsDescription objects. Specify each variable that will be returned and its type using a tables.Col object:

Note: See the pytables documentation for a list of Col types

```
import tables

class TrialData(tables.IsDescription):
    trial_num    = tables.Int32Col()
    target       = tables.StringCol(1)
    response     = tables.StringCol(1)
    correct      = tables.Int32Col()
    correction   = tables.Int32Col()
    RQ_timestamp = tables.StringCol(26)
    DC_timestamp = tables.StringCol(26)
    bailed       = tables.Int32Col()
```

The column types are names with their type and their bit depth except for the StringCol which takes a string length in characters.

The TrialData object is used by the *Subject* class when a task is assigned to create the data storage table.

PLOT

The PLOT dictionary maps the data returned from the task to graphical elements in the *Terminal*'s *Plot*. Specifically, when the task is started, the *Plot* object creates the graphical element (eg. a *Point*) and then calls its update method with any data that is received through its Node.

Data-to-graphical mappings are defined in a data subdictionary, and additional parameters can be passed to the plot – in the below example, for example, a chance_bar is drawn as a horizontal line across the plot. By default it is drawn at 0.5, but its height can be set with an additional parameter chance_level. Available graphical primitives are registered in the plots.PLOT_LIST, and additional parameters are documented in the *Plot* class.

Data is plotted either by trial (default) or by timestamp (if PLOT['continuous'] != True). Numerical data is plotted (on the y-axis) as expected, but further mappings can be defined by extending the graphical element's update method – eg. 'L'(eft) maps to 0 and 'R'(ight) maps to 1 by default.

```
PLOT = {
    'data': {
        'target'   : 'point',
        'response' : 'segment',
        'correct'  : 'rollmean'
    },
    'chance_bar'  : True, # Draw a red bar at 50%
    'roll_window' : 50    # n trials to take rolling mean over
}
```

The above PLOT dictionary produces this pretty little plot:

Todo: screenshot of default nafc plot

HARDWARE

The HARDWARE dictionary maps a hardware type (eg. POKES) and identifier (eg. 'L') to a Hardware object. The task uses the hardware parameterization in the *prefs* file (also see setup_pilot) to instantiate each of the hardware objects, so their naming system must match (ie. there must be a prefs.PINS['POKES']['L'] entry in prefs for a task that has a task.HARDWARE['POKES']['L'] object).

```
from autopilot.core import hardware

HARDWARE = {
    'POKES':{
        'L': hardware.Beambreak,
        'C': hardware.Beambreak,
        'R': hardware.Beambreak
    },
    'LEDS':{
        'L': hardware.LED_RGB,
        'C': hardware.LED_RGB,
        'R': hardware.LED_RGB
    },
    'PORTS':{
        'L': hardware.Solenoid,
        'C': hardware.Solenoid,
        'R': hardware.Solenoid
    }
}
```

7.1.3 Initialization

First, the parameters that are given to the task when it is initialized are stored as attributes, either by unpacking **kwargs...

```
class Nafc(Task):

    def __init__(**kwargs):
        for key, value in kwargs.items():
            setattr(self, key, value)
```

Or explicitly, which is recommended as it is more transparent:

```
class Nafc(Task):

    def __init__(self, stage_block=None, stim=None, reward=50, req_reward=False,
                 punish_stim=False, punish_dur=100, correction=False, correction_pct=50.,
                 bias_mode=False, bias_threshold=20, current_trial=0, **kwargs):

```

(continues on next page)

(continued from previous page)

```
        self.req_reward     = bool(req_reward)
        self.punish_stim    = bool(punish_stim)
        self.punish_dur     = float(punish_dur)
        self.correction     = bool(correction)
        self.correction_pct = float(correction_pct)/100
        self.bias_mode      = bias_mode
        self.bias_threshold = float(bias_threshold)/100

        # etc...
```

Then the hardware is instantiated using a method inherited from the *Task* class:

```
self.init_hardware()
```

Stimulus managers need to be instantiated separately. Currently, stimulus management details like correction trial percentage or bias correction are given as separate parameters, but will be included in the `stim` parameter in the future:

```
# use the init_manager wrapper to choose the correct stimulus manager
self.stim_manager = init_manager(stim)

# give the sounds a function to call when they end
self.stim_manager.set_triggers(self.stim_end)

if self.correction:
    self.stim_manager.do_correction(self.correction_pct)

if self.bias_mode:
    self.stim_manager.do_bias(mode=self.bias_mode,
                              thresh=self.bias_threshold)
```

There are a few attributes that can be set at initialization that are unique:

- **stage_block** - if the task is structured such that the *Pilot* calls each stage method and returns the resulting data, this `threading.Event` is used to wait between stages – an example will be shown below.
- **stages** - an iterator or generator that yields stage methods.

In this example we have structured the task such that its stages (described below) are called in an endless cycle:

```
# This allows us to cycle through the task by just repeatedly calling self.stages.next()
stage_list = [self.request, self.discrim, self.reinforcement]
self.stages = itertools.cycle(stage_list)
```

7.1.4 Stage Methods

The logic of a task is implemented in one or several **stages**. This example Nafc class uses three:

1. `request` - precomputes the target and distractor ports, caches the stimulus, and sets the stimulus to play when the center port is entered
2. `discrim` - sets the reward and punishment triggers for the target and distractor ports
3. `reinforcement` - computes the trial result and readies the task for the next trial.

This task does not call its own stage methods, as we will see in the Wheel task example, but allows the `Pilot` to control them, and advances through stages using a `stage_block` that allows passage whenever a GPIO trigger is activated. Data is returned from each of the stage methods and is then returned to the `Terminal` by the `Pilot`.

Request

First, the `stage_block` is cleared so that the task will not advance until one of the triggers is called. The target and distractor ports are yielded by the `stim_manager` along with the stimulus object.

```
def request(self, *args, **kwargs):
    # Set the event block
    self.stage_block.clear()

    # get next stim
    self.target, self.distractor, self.stim = self.stim_manager.next_stim()
    # buffer it
    self.stim.buffer()
```

Then triggers are stored under the name of the trigger (eg. *'C'* for a trigger that comes from the center poke). All triggers need to be callable, and can be set either individually or as a series, as in this example. A `lambda` function is used to set a trigger with arguments – the center LED is set from green to blue when the stimulus starts playing.

A single task class can support multiple operating modes depending on its parameters. If the task has been asked to give request rewards (see *Training a Subject*), it adds an additional trigger to open the center solenoid.

```
# set the center light to green before the stimulus is played.
self.set_leds({'C': [0, 255, 0]})

# Set sound trigger and LEDs
# We make two triggers to play the sound and change the light color
change_to_blue = lambda: self.pins['LEDS']['C'].set_color([0,0,255])

# set triggers
if self.req_reward is True:
    self.triggers['C'] = [self.stim.play,
                          self.stim_start,
                          change_to_blue,
                          self.pins['PORTS']['C'].open]
else:
    self.triggers['C'] = [self.stim.play,
                          self.stim_start,
                          change_to_blue]
```

Finally, the data for this stage of the trial is gathered and returned to the Pilot. Since stimuli have variable numbers and names of parameters, both the table set up by the `Subject` and the data returning routine here extract stimulus parameters programmatically.

```
self.current_trial = self.trial_counter.next()
data = {
    'target'      : self.target,
    'trial_num'   : self.current_trial,
    'correction'  : self.correction_trial
}
# get stim info and add to data dict
```

(continues on next page)

(continued from previous page)

```
sound_info = {k:getattr(self.stim, k) for k in self.stim.PARAMS}
data.update(sound_info)
data.update({'type':self.stim.type})

return data
```

At the end of this function, the center LED is green, and if the subject pokes the center port the stimulus will play and then the next stage method will be called.

The center LED also turns from green to blue when the stimulus begins to play and then turns off when it is finished. This relies on additional methods that will be explained below.

Discrim

The discrim method simply sets the next round of triggers and returns the request timestamp from the current trial. If either the `target` or `distractor` ports are triggered, the appropriate solenoid is opened or the `punish` method is called.

The `trial_num` is returned each stage for an additional layer of redundancy in data alignment.

```
def discrim(self,*args,**kwargs):
    # clear stage block to wait for triggers
    self.stage_block.clear()

    # set triggers
    self.triggers[self.target]     = [lambda: self.respond(self.target),
                                      self.pins['PORTS'][self.target].open]
    self.triggers[self.distractor] = [lambda: self.respond(self.distractor),
                                      self.punish]

    # Only data is the timestamp
    data = {'RQ_timestamp' : datetime.datetime.now().isoformat(),
            'trial_num'    : self.current_trial}
    return data
```

Todo: pigpio can give us 5 microsecond measurement precision for triggers, currently we just use `datetime.datetime.now()` for timestamps, but highly accurate timestamps can be had by stashing the `ticks` argument given by pigpio to the *`handle_trigger()`* method. We will implement this if you don't first :)

Reinforcement

This method computes the results of the tasks and returns them with another timestamp. This stage doesn't clear the `stage_block` because we want the next trial to be started immediately after this stage completes.

The results of the current trial are given to the stimulus manager's `update()` method so that it can keep track of trial history and do things like bias correction, etc.

The `TRIAL_END` flag in the data signals to the *`Subject`* class that the trial is finished and its row of data should be written to disk. This, along with providing the `trial_num` on each stage, ensure that data is not misaligned between trials.

```
def reinforcement(self,*args,**kwargs):

    if self.response == self.target:
        self.correct = 1
    else:
        self.correct = 0

    # update stim manager
    self.stim_manager.update(self.response, self.correct)

    data = {
        'DC_timestamp' : datetime.datetime.now().isoformat(),
        'response'     : self.response,
        'correct'      : self.correct,
        'trial_num'    : self.current_trial,
        'TRIAL_END'    : True
    }
    return data
```

7.1.5 Additional Methods

Autopilot doesn't confine the logic of a task to its stage methods, instead users can use additional methods to give their task additional functionality.

These can range from trivial methods that just store values, such as the `respond` and `stim_start` methods:

```
def respond(self, pin):
    self.response = pin

def stim_start(self):
    self.discrim_playing = True
```

To more complex methods that operate effectively like stages, like the `punish` method, which flashes the LEDs and plays a punishment stimulus like white noise if it has been configured to do so:

```
def punish(self):
    # clear the punish block to the task doesn't advance while
    # punishment is delivered
    self.punish_block.clear()

    # if there is some punishment stimulus, play it
    if self.punish_stim:
        self.stim_manager.play_punishment()

    # flash LEDs and then clear the block once they are finished.
    self.flash_leds()
    threading.Timer(self.punish_dur / 1000.,
                    self.punish_block.set).start()
```

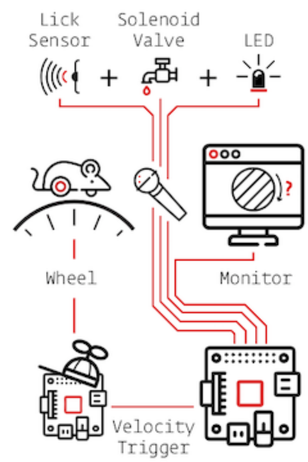
Additionally, since we gave the stimulus manager a trigger method that is called when the stimulus ends, we can turn the light blue when a stimulus is playing, and turn it off when it finishes

```python
def stim_end(self):
    """
    called by stimulus callback

    set outside lights blue
    """
    # Called by the discrim sound's table trigger when playback is finished
    # Used in punishing leaving early
    self.discrim_playing = False
    #if not self.bailed and self.current_stage == 1:
    self.set_leds({'L':[0,255,0], 'R':[0,255,0]})
```

7.2 Distributed Go/No-Go - Using Child Agents

To demonstrate the use of Child agents, we'll build the distributed Go/No-Go task described in section 4.3 of the Autopilot whitepaper.



In short, a subject runs on a circular running wheel whose velocity is measured by a laser computer mouse. When the subject 'fixates' by slowing below a threshold velocity, an drifting Gabor grating is presented. If the grating changes angles, the subject is rewarded if they lick in an IR beambreak sensor. If the grating doesn't change angles, the subject is rewarded if they refrain from licking until the stimulus has ended.

7.2.1 Additional Prefs

To use a Child with this task, we will need to have a second Raspberry Pi setup with the same routine as a Pilot, except it needs the following values in its `prefs.json` file:

Child Prefs

```json
{
    "NAME" : "wheel_child",
    "LINEAGE" : "CHILD",
    "PARENTID" : "parent_pilot",
    "PARENTIP" : "ip.of.parent.pilot",
    "PARENTPORT": "<MSGPORT of parent>",
}
```

And the parent pilot needs to have

Parent Prefs

```json
{
    "NAME": "parent_pilot",
    "CHILDID": "wheel_child",
    "LINEAGE": "PARENT"
}
```

7.2.2 Go/No-Go Parameterization

The parameterization for this task is similar to that of the Nafc task above with a few extensions…

```
from autopilot.tasks import Task

class GoNoGo(Task):


    # Task parameterization
    PARAMS = odict()
    PARAMS['reward'] = {'tag': 'Reward Duration (ms)',
                        'type': 'int'}
    PARAMS['timeout']        = {'tag':'Delay Timeout (ms)',
                                'type':'int'}
    PARAMS['stim'] = {'tag':  'Visuals',
                      'type': 'visuals'}

    # Plot parameterization
    PLOT = {
        'data': {
            'x': 'shaded',
            'target': 'point',
            'response': 'segment'
        },
        # our plot will use time as its x-axis rather than the␣
↪trial number
        'continuous': True
    }

    # TrialData descriptor
    class TrialData(tables.IsDescription):
        trial_num    = tables.Int32Col()
        target       = tables.BoolCol()
        response     = tables.StringCol(1)
        correct      = tables.Int32Col()
        RQ_timestamp = tables.StringCol(26)
        DC_timestamp = tables.StringCol(26)
        shift        = tables.Float32Col()
        angle        = tables.Float32Col()
        delay        = tables.Float32Col()
```

We add one additional data descriptor that describes the continuous data that will be sent from the `Wheel` object:

```
class ContinuousData(tables.IsDescription):
    x = tables.Float64Col()
    y = tables.Float64Col()
    t = tables.Float64Col()
```

The hardware specification is also similar, with one additional `Flag` object which behaves identically to the `Beambreak` object with reversed logic (triggered by 0->1 rather than 1->0).

```
HARDWARE = {
    'POKES': {
        'C': hardware.Beambreak,
    },
    'LEDS': {
        'C': hardware.LED_RGB,
    },
    'PORTS': {
        'C': hardware.Solenoid,
    },
    'FLAGS': {
        'F': hardware.Flag
    }
}
```

Finally, we add an additional `CHILDREN` dictionary to specify the type of Child that we need to run the task, as well as any additional parameters needed to configure it.

The `task_type` must refer to some key in the `autopilot.tasks.CHILDREN_LIST`.

Note: The `Child` agent is a subconfiguration of the `Pilot` agent, they will be delineated more explicitly as the agent framework is solidified.

```
CHILDREN = {
    'WHEEL': {
        'task_type': "Wheel Child",
    }
}
```

7.2.3 Initialization

When initializing this task, we need to make our own `Net_Node` object as well as initialize our child. Assuming that the child is connected to the parent and appropriately configured (see the additional params above), then things should go smoothly.

Warning: Some of the parameters – most egregiously the Grating stimulus – are hardcoded in the initialization routine. **This is bad practice** but an unfortunately necessary evil because the visual stimulus infrastructure is not well developed yet.

```
from autopilot.stim.visual.visuals import Grating

def __init__(self, stim=None, reward = 50., timeout = 1000., stage_block = None,
             punish_dur = 500., **kwargs):
    super(GoNoGo, self).__init__()

    # we receive a stage_block from the pilot as usual, we won't use it
    # for task operation though.
    self.stage_block = stage_block
    self.trial_counter = itertools.count()
```

(continues on next page)

(continued from previous page)

```
    # save parameters passed to us as arguments
    self.punish_dur = punish_dur
    self.reward = reward
    self.timeout = timeout
    self.subject = kwargs['subject']


    # init hardware and set reward as before
    self.init_hardware()
    self.set_reward(self.reward)

    # hardcoding stimulus while visual stim still immature
    self.stim = Grating(angle=0, freq=(4,0), rate=1, size=(1,1), debug=True)

    self.stages = itertools.cycle([self.request, self.discrim, self.reinforce])
```

Initializing the Net Node.

The `Net_Node` gets the following arguments:

- `id`: The name that is used to identify the task's networking object so other networking objects can send it messages. We prefix the pilot's `prefs.NAME` with `T_` because it is a task, though this is not required.
- `upstream`: The name of the network node that is directly upstream from us, we will be sending our messages to the *Pilot* that is running us – and thus address it by its name
- `port`: The port of our upstream mode, most commonly the `prefs.MSGPORT`
- `listens`: A dictionary that maps messages with different ``KEY``s to specific handling methods. Since we don't need to receive any data for this task, this is blank,
- `instance`: Optional, denotes whether this node shouldn't be the only node that exists within the Agent – ie. it uses the same instance of the `tornado` IOLoop as other nodes.

```
self.node = Net_Node(id="T_{}".format(prefs.NAME),
                     upstream=prefs.NAME,
                     port=prefs.MSGPORT,
                     listens={},
                     instance=True)
```

And then to initialize our Child we construct a message to send along to it.

Note that we send the message to `prefs.NAME` – we don't want to have to know the IP address/etc. for our child because it connects to us – so the `Station` object handles sending it along with its *Pilot_Station.l_child()* `listen`.

```
# construct a message to send to the child
value = {
    'child': {'parent': prefs.NAME, 'subject': self.subject},
    'task_type': self.CHILDREN['WHEEL']['task_type'],
    'subject': self.subject
}

# send to the station object with a 'CHILD' key
self.node.send(to=prefs.NAME, key='CHILD', value=value)
```

7.2.4 The Child Task

The *Wheel_Child* task is a very thin wrapper around a `Wheel` object, which does most of the work.

It creates a `stages` iterator with a function that returns nothing to fit in with the general task structure.

```
class Wheel_Child(object):
    STAGE_NAMES = ['collect']

    PARAMS = odict()
    PARAMS['fs'] = {'tag': 'Velocity Reporting Rate (Hz)',
                    'type': 'int'}
    PARAMS['thresh'] = {'tag': 'Distance Threshold',
                        'type': 'int'}

    HARDWARE = {
        "OUTPUT": Digital_Out,
        "WHEEL":  Wheel
    }



    def __init__(self, stage_block=None, fs=10, thresh=100, **kwargs):
        self.fs = fs
        self.thresh = thresh

        self.hardware = {}
        self.hardware['OUTPUT'] = Digital_Out(prefs.PINS['OUTPUT'])
        self.hardware['WHEEL'] = Wheel(digi_out = self.hardware['OUTPUT'],
                                       fs       = self.fs,
                                       thresh   = self.thresh,
                                       mode     = "steady")
        self.stages = cycle([self.noop])
        self.stage_block = stage_block

    def noop(self):
        # just fitting in with the task structure.
        self.stage_block.clear()
        return {}

    def end(self):
        self.hardware['WHEEL'].release()
        self.stage_block.set()
```

7.2.5 A Very Smart Wheel

Most of the Child's contribution to the task is performed by the `Wheel` object.

The Wheel accesses a USB mouse connected to the Pilot, continuously collects its movements, and reports them back to the Terminal with a specified frequency (`fs`) with an internal `Net_Node`

An abbreviated version…

```
from inputs import devices

class Wheel(Hardware):

    def __init__(self, mouse_idx=0, fs=10, thresh=100, thresh_type='dist', start=True,
                 digi_out = False, mode='vel_total', integrate_dur=5):

        self.mouse = devices.mice[mouse_idx]
        self.fs = fs
        self.thresh = thresh
        # time between updates
        self.update_dur = 1./float(self.fs)
```

The Wheel has three message types,

- `'MEASURE'` - the main task is telling us to monitor for a threshold crossing, ie. previous trial is over and it's ready for another one.
- `'CLEAR'` - stop measuring for a threshold crossing event!
- `'STOP'` - the task is over, clear resources and shut down.

```
    # initialize networking
    self.listens = {'MEASURE': self.l_measure,
                    'CLEAR'  : self.l_clear,
                    'STOP'   : self.l_stop}

    self.node = Net_Node('wheel_{}'.format(mouse_idx),
                         upstream=prefs.NAME,
                         port=prefs.MSGPORT,
                         listens=self.listens,
                         )

    # if we are being used in a child object,
    # we send our trigger via a GPIO pin
    self.digi_out = digi_out

    self.thread = None

    if start:
        self.start()

def start(self):
    self.thread = threading.Thread(target=self._record)
    self.thread.daemon = True
    self.thread.start()
```

The wheel starts two threads, one that captures mouse movement events and puts them in a queue, and another that processes movements, transmits them to the Terminal, and handles the threshold triggers when the subject falls below a certain velocity.

```
def _mouse(self):
    # read mouse movements and put them in a queue
    while self.quit_evt:
        events = self.mouse.read()
        self.q.put(events)

def _record(self):

    threading.Thread(target=self._mouse).start()

    # a threading.Event is used to terminate the wheel's operation
    while not self.quit_evt.is_set():

    # ... mouse movements are collected into a 2d numpy array ...

    # if the main task has told us to measure for a velocity threshold
    # we check if our recent movements (move) trigger the threshold
    if self.measure_evt.is_set():
        do_trigger = self.check_thresh(move)
        if do_trigger:
            self.thresh_trig()
            self.measure_evt.clear()

    # and we report recent movements back to the Terminal
    # the recent velocities and timestamp have been calculated as
    # x_vel, y_vel, and nowtime
    self.node.send(key='CONTINUOUS',
                   value={
                       'x':x_vel,
                       'y':y_vel,
                       't':nowtime
                   })
```

If the threshold is triggered, a method (...``thresh_trig``...) is called that sends a voltage pulse through the `Digital_Out` given to it by the Child task.

```
def thresh_trig(self):
    if self.digi_out:
        self.digi_out.pulse()
```

7.2.6 Go/No-Go Stage Methods

After the child is initialized, the Parent pilot begins to call the three stage functions for the task in a cycle

Very similar to the Nafc task above…

- `request` - Tell the Child to begin measuring for a velocity threshold crossing, prepare the stimulus for delivery
- `discrim` - Present the stimulus
- `reinforce` - Reward the subject if they were correct

The code here has been abbreviated for the purpose of the example:

```
def request(self):
    # Set the event lock
    self.stage_block.clear()
    # wait on any ongoing punishment stimulus
    self.punish_block.wait()



    # set triggers
    self.triggers['F'] = [
        lambda: self.stim.play('shift', self.shift )
    ]

    # tell our wheel to start measuring
    self.node.send(to=[prefs.NAME, prefs.CHILDID, 'wheel_0'],
                   key="MEASURE",
                   value={'mode':'steady',
                          'thresh':100})

    # return data from current stage
    self.current_trial = self.trial_counter.next()
    data = {
        'target': self.target, # whether to 'go' or 'not go'
        'shift': self.shift,   # how much to shift the
                               # angle of the stimulus
        'trial_num': self.current_trial
    }

    return data


def discrim(self):
    # if the subject licks on a good trial, reward.
    # set a trigger to respond false if delay time elapses
    if self.target:
        self.triggers['C'] = [lambda: self.respond(True), self.pins['PORTS']['C'].open]
        self.triggers['T'] = [lambda: self.respond(False), self.punish]
    # otherwise punish
    else:
        self.triggers['C'] = [lambda: self.respond(True), self.punish]
        self.triggers['T'] = [lambda: self.respond(False), self.pins['PORTS']['C'].open]
```

(continues on next page)

(continued from previous page)

```
    # the stimulus has just started playing, wait a bit and then shift it (if we're gonna
    # choose a random delay
    delay = 0.0
    if self.shift != 0:
        delay = (random()*3000.0)+1000.0
        # a delay timer is set that shifts the stimulus after
        # <delay> milliseconds
        self.delayed_set(delay, 'shift', self.shift)

    # trigger the timeout in 5 seconds
    self.timer = threading.Timer(5.0, self.handle_trigger, args=('T', True, None)).
↪start()

    # return data to the pilot
    data = {
        'delay': delay,
        'RQ_timestamp': datetime.datetime.now().isoformat(),
        'trial_num': self.current_trial
    }

    return data


def reinforce(self):

    # stop timer if it's still going
    try:
        self.timer.cancel()
    except AttributeError:
        pass
    self.timer = None

    data = {
        'DC_timestamp': datetime.datetime.now().isoformat(),
        'response': self.response,
        'correct': self.correct,
        'trial_num': self.current_trial,
        'TRIAL_END': True
    }
    return data
```

Viola.

Important: This guide and guide_task are lightly out of date with v0.4.0 of autopilot, but still largely reflect the program design and its operation. This guide in particular became obsolete because most extensions to hardware objects are now done by subclassing generic hardware classes like *hardware.gpio.GPIO* and their descendents, which make it relatively clear what parts of the object need to be modified.

As such, this part of the docs was deprecated in v0.3.0 and has been mostly removed in v0.4.0 pending a fuller rewrite.

For now, see the API documentation section for *hardware* for more details on how to extend hardware classes :)

Sorry for the inconvenience, we are a very small team and can only do so much work between releases! We'd be happy to get documentation requests or even a pull request or two to help us out until we can get to it :)

CHAPTER

EIGHT

WRITING A HARDWARE CLASS

There are precious few requirements for `Hardware` objects in Autopilot.

- Each class should have a `release()` method that stops any running threads and releases any system resources – especially those held by pigpio.
- **Each class should define a handful of class attributes when relevant**
  - `trigger` (bool) - whether the device is used to trigger an event. if `True`, `assign_cb()` must be defined and the device will be given a callback function by the instantiating *Task* class
  - `type` (str) - what this device should be known as in `prefs`. Not enforced currently, but will be.
  - `input` and `output` (bool) - whether the device is an input or output device, if either
- When making threaded methods, care should be taken not to spawn an excessive number of running threads, but this is a performance rather than a structural limit.

To use a hardware object in a task, its parameters (especially the pin number for pigpio-based hardware) should be stored in `prefs.json`.

A few basic Hardware classes are dissected in this section to illustrate basic principles of their design, but we expect Hardware objects to be extremely variable in their implementation and application.

8.1 GPIO with pigpio

Autopilot uses pigpio to interface with the Raspberry Pi's GPIO pins. All pigpio objects require that a pigpiod daemon is running as a background process. This used to be done by a launch script that started the pilots, but is now typically launched by `autopilot.external.start_pigpiod()`, which is called by *GPIO.init_pigpio()* so in general you shouldn't need to worry about it. If `pigpiod` is open in a separate process, or left open from a previous crashed run of Autopilot, you will likely need to kill that process before you can use more GPIO-based autopilot objects.

When instantiating a piece of hardware, it must connect to pigpiod by creating a pigpio.pi object, which allows communication with the GPIO. This is provided by the `GPIO.pig` property. The rest of the methods of GPIO-based objects are built around abstractions of commands to the `pig`. See *gpio.LED_RGB* for an example of a subclass that overrides some methods from the *gpio.GPIO* metaclass to be able to control three PWM objects with a similar syntax as other GPIO outputs.

CHAPTER

NINE

PLUGINS & THE WIKI

Autopilot is integrated with a semantic wiki, a powerful tool that merges human-readable text with computer-readable structured information, and blurs the lines between the two in the empowering interface of a wiki that allows anyone to edit it. The autopilot wiki is available at:

https://wiki.auto-pi-lot.com

In addition to a system for storing, discussing, and knitting together a library of technical knowledge, the wiki is used to manage Autopilot's plugin system. The integrated plugin/wiki system is designed to

- make it easier to **extend** and hack existing autopilot classes, particularly Hardware and Task classes, without needing to modify any of the core library code
- make it easier to **share code** across multiple rigs-in-use by allowing you to specify the name of the plugin on the autopilot wiki so you don't need to manually keep the code updated on all computers it's used on
- make a gentler **scaffold between using and contributing to the library** – by developing in a plugin folder, your code is likely very close, if it isn't already, ready to integrate back into the main autopilot library. In the meantime, anyone that is curious
- make it possible to **encode semantic metadata about the plugin** so that others can **discover, modify, and improve** on it. eg. your plugin might control an array of stepper motors, and from that someone can cherrypick code to run a single one, even if it wasn't designed to do that.
- **decentralize the development of autopilot**, allowing anyone to extend it in arbitrary ways without needing to go through a fork/merge process that is ultimately subject to the whims of the maintainer(s) (me), or even an approval process to submit or categorize plugins. Autopilot seeks to be as noncoercive as possible while embracing and giving tools to support the heterogeneity of its use.
- make it trivial for users to not only contribute *plugins* but design new *types of plugin-like public interfaces*. For example, if you wanted to design an interface where users can submit the parameters they use for different tasks, one would only need to build the relevant semantic mediawiki template and form, and then program the API calls to the wiki to index them.
- `todo` — fully realize the vision of decentralized development by allowing plugins to replace existing core autopilot modules…

9.1 Plugins

Plugins are now the recommended way to use Autopilot! They make very few assumptions about the structure of your code, so they can be used like familiar script-based experimental tools, but they also encourage the development of modular code that can easily be used by others and cumulatively contribute to a shared body of tools.

Using plugins is simple! Anything inside of the directory indicated by `prefs.get('PLUGINDIR')` is a plugin! Plugins provide objects that inherit from Autopilot classes supported by an entry in `registry.REGISTRIES` .

For example, we want to write a task that uses some special hardware that we need. We could start by making a directory within `'PLUGINDIR'` like this:

```
plugins
├── my-autopilot-plugin
│   ├── README.md
│   ├── test_hardware.py
│   └── test_task.py
```

Where within `test_hardware.py` you define some custom hardware class that inherits from *gpio.Digital_Out*

```
from autopilot.hardware.gpio import Digital_Out

class Only_On_Pin(Digital_Out):
    """
    you can only turn this GPIO pin on
    """
    def __init__(self, pin, *args, **kwargs):
        super(Only_On_Pin, self).__init__(pin=pin, *args, **kwargs)
        self.set(1)

    def set(self, val):
        """override base class"""
        if val not in (1, True, 'on'):
            raise ValueError('This pin only turns on')
        else:
            super(Only_On_Pin, self).set(val)

    def release(self):
        print('I release nothing. the pin stays on.')
```

You can then use it in some task! Autopilot will use its registry `autopilot.get()` methods to find it after importing all your plugins. For example, we can refer to it as a string in our `HARDWARE` dictionary in our special task:

```
from datetime import datetime
import threading
import numpy as np
from autopilot.tasks import Task
from tables import IsDescription, StringCol

class My_Task(Task):
    """
    I will personally subject myself to the labor of science and through careful hours␣
↪spent meditating on an LED powered by an unsecured Raspberry Pi with the default␣
↪password i will become attuned to the dance of static pixels fluctuating on the␣
↪fundamentalfrequencies of ransomware and ssh bombardment to harnessthe power of both␣
↪god and anime
```

(continues on next page)

(continued from previous page)

```
    """

    PARAMS = {'infinite_light': {
                'tag': 'leave the light on indefinitely? are you sure you want to leave␣
↪the rest of the world behind and never cease your pursuit of this angelic orb?',
                'type': 'bool'}}

    HARDWARE = {'esoterica': {'the_light': 'Only_On_Pin'}}

    class TrialData(IsDescription):
        ontime = StringCol(26)

    def __init__(self, infinite_light:bool=True, *args, **kwargs):
        super(My_Task, self).__init__(*args, **kwargs)
        self.init_hardware()
        self.hardware['esoterica']['the_light'].set(True)

        if not infinite_light:
            infinite_light = True
        self.infinite_light = infinite_light

        self.stages = [self.only_on]

    def only_on(self):
        self.stage_block.clear()

        if not self.infinite_light:
            threading.Timer(np.random.rand()*10e100, self.cease_your_quest).start()

        return {'ontime': datetime.now().isoformat()}

    def cease_your_quest(self):
        self.stage_block.set()
        self.hardware['esoterica']['the_light'].release()
```

Both your hardware object and task will be available to the rest of Autopilot, including in the GUI elements that let you easily parameterize and assign it to your experimental subjects.

Todo: We are still working on formalizing the rest of a plugin architecture, specifically dependency resolution among python packages, autopilot scripts, and dependencies on other plugins. All this in time! For now the wiki asks for a specific autopilot version that a plugin supports when they are submitted, so we will be able to track plugins that need to be updated for changes in the plugin API as it is developed.

9.2 Registries

Plugins are supported by the functions in the *utils.registry* module. Registries allow us to make definite but abstract references to classes of objects that can therefore be extended with plugins.

Since for now Autopilot objects are not guaranteed to have a well-defined inheritance structure, registries are available to the classes of objects listed in the `registry.REGISTRIES` enum. Currently they are:

```
class REGISTRIES(str, Enum):
    """
    Types of registries that are currently supported,
    ie. the possible values of the first argument of :func:`.registry.get`

    Values are the names of the autopilot classes that are searched for
    inheriting classes, eg. ``HARDWARE == "autopilot.hardware.Hardware"`` for␣
↪:class:`autopilot.Hardware`
    """
    HARDWARE = "autopilot.hardware.Hardware"
    TASK = "autopilot.tasks.Task"
    GRADUATION = "autopilot.tasks.graduation.Graduation"
    TRANSFORM = "autopilot.transform.transforms.Transform"
    CHILDREN = "autopilot.tasks.children.Child"
    SOUND = "autopilot.stim.sound.sounds.BASE_CLASS"
```

Each entry in the enum refers to the absolute package.module.class name of the topmost metaclass that is to be searched.

The `autopilot.get()` method first gets the base class with *find_class()*, ensures that plugins have been imported with *import_plugins()* , and searches for a subclass with a matching name with *recurse_subclasses()* . If none is found in the currently imported files, it parses the ast of any files below the base class in the path hierarchy. The distinction is because while we *do* assume that we can import anything we have made/put in our plugins directory, we currently *don't* make that assumption of the core library of autopilot – we want to be able to offer the code for tasks and hardware that have diverse dependencies while giving ourselves some protection against writing squirrelly edge cases everywhere.

In practice, anywhere you go to make an explicit import of an autopilot class that is suported by a registry, it is good practice to use `autopilot.get` instead. It is called like:

```
# autopilot.get('registry_name', 'object_name')
# eg.
autopilot.get('hardware', 'Digital_Out')
```

Note how the registry name is not case sensitive but the object name is. There are a few convenience methods/calling patterns here too. Eg. to list all available objects in a registry:

```
autopilot.get('hardware')
```

or to list just a list of strings instead of the objects themselves:

```
autopilot.get_names('hardware')
```

or you can pass an object itself as the registry type in order to only find subclasses of that class:

```
GPIO = autopilot.get('hardware', 'GPIO')
autopilot.get(GPIO)
```

Todo: In the future, we will extend registries to all autopilot objects by implementing a unitary inheritance structure. This will also clean up a lot of the awkward parts of the library and pave the way to rebuilding eg. the networking modules to be much simpler to use.

That work will be the defining feature of v0.5.0, you can track progress and contribute by seeing the relevant issue: https://github.com/wehr-lab/autopilot/issues/31

as well as the issues in the v0.5.0 milestone: https://github.com/wehr-lab/autopilot/milestone/2

9.3 The Wiki API

The wiki's semantic information can be accessed with the functions in the *utils.wiki* module.

Specifically, we make a function that wraps the Semantic Mediawiki Ask API that consists of a

- **query** or a set of **filters** that select relevant pages using their **categories** and **properties**, and then
- the **properties** to retrieve from those pages.

You can see a list of the categories and properties that can be used on the wiki.

For **Filters**:

- **Both** types of filters are specified with the `[[Double Brackets]]` of mediawiki
- **Categories** are specified with a single colon[1] like `[[Category:Hardware]]`
- **Properties** are specified with double colons, and take a property and a value like `[[Created By::Jonny Saunders]]`

The **queried properties** are specified with a list of strings like `['Has Datasheet', 'Has STL']`

So, for example, one could query the manufacturer, price, and url of the audio hardware documented in the wiki like:

```
from autopilot.utils import wiki

wiki.ask(
    filters=[
        "[[Category:Hardware]]",
        "[[Modality::Audio]]"
    ],
    properties=[
        "Manufactured By",
        "Has Product Page",
        "Has USD Price"
    ]
)
```

which would return a list of dictionaries like:

```
[{
    'Has Product Page': 'https://www.hifiberry.com/shop/boards/hifiberry-amp2/',
    'Has USD Price': 49.9,
    'Manufactured By': 'HiFiBerry',
```

(continues on next page)

[1] This is because categories are a part of mediawiki itself, but properties are implemented by semantic mediawiki. The two have slightly different meanings – categories denote the "type of something that a page is" and properties denote "the attributes that a page has"

(continued from previous page)

```
    'name': 'HiFiBerry Amp2',
    'url': 'https://wiki.auto-pi-lot.com/index.php/HiFiBerry_Amp2'
},
{
    'Has Datasheet': 'https://wiki.auto-pi-lot.com/index.php/File:HiVi-RT13WE-spec-sheet.
↪pdf',
    'Has Product Page': 'https://www.parts-express.com/HiVi-RT1.3WE-Isodynamic-Tweeter-
↪297-421',
    'Has USD Price': 37.98,
    'Is Part Type': 'Speakers',
    'Manufactured By': 'HiVi',
    'name': 'HiVi RT1.3WE',
    'url': 'https://wiki.auto-pi-lot.com/index.php/HiVi_RT1.3WE'
}]
```

These functions can be used on their own to provide interactive, programmatic access to the wiki, but maybe more importantly it serves as a bridge between the wiki and Autopilot's software. By building API calls into the various modules of autopilot that can query structured information from the wiki, the software can be made to take advantage of communally curated experimental and technical knowledge.

Additionally, since it is relatively simple to create new templates and forms (see the Page Forms and Page Schemas extensions that are used to create and manage them)t o accept different kinds of submissions and link them to the rest of the wiki, and the plugin and registry system allow anyone to build the classes needed to take advantage of them, it becomes possible for anyone to create **new kinds of public knowledge interfaces to autopilot.** For example, if there was desire to share and describe parameterizations of a particular `Task` along with summaries of the data, then it would be possible to make a form and template on the wiki to accept them, and provide a GUI plugin to select *empirically optimal parameters for a given outcome measurement* , which would make all the *hard-won rules of thumb and superstition that guides a lot of the fine decisions in behavioral research obsolete in an afternoon.*

The use of the wiki to have communal control over plugins and interfaces makes it possible for us to move autopilot to a model of **decentralized governance** where the "official" repository becomes one version among many, but the plugins remain integrated with the system rather than live on as unrelated forks.

9.4 Plugins on the Wiki

Autopilot plugins can be found on the wiki here: https://wiki.auto-pi-lot.com/index.php/Autopilot_Plugins

(at the moment the cupboard is relatively bare, but it always starts that way.)

Within Autopilot, you can use the `utils.plugins.list_wiki_plugins()` function to list the available functions and return their basic metadata, which is a *very* thin wrapper around `utils.wiki.ask()`

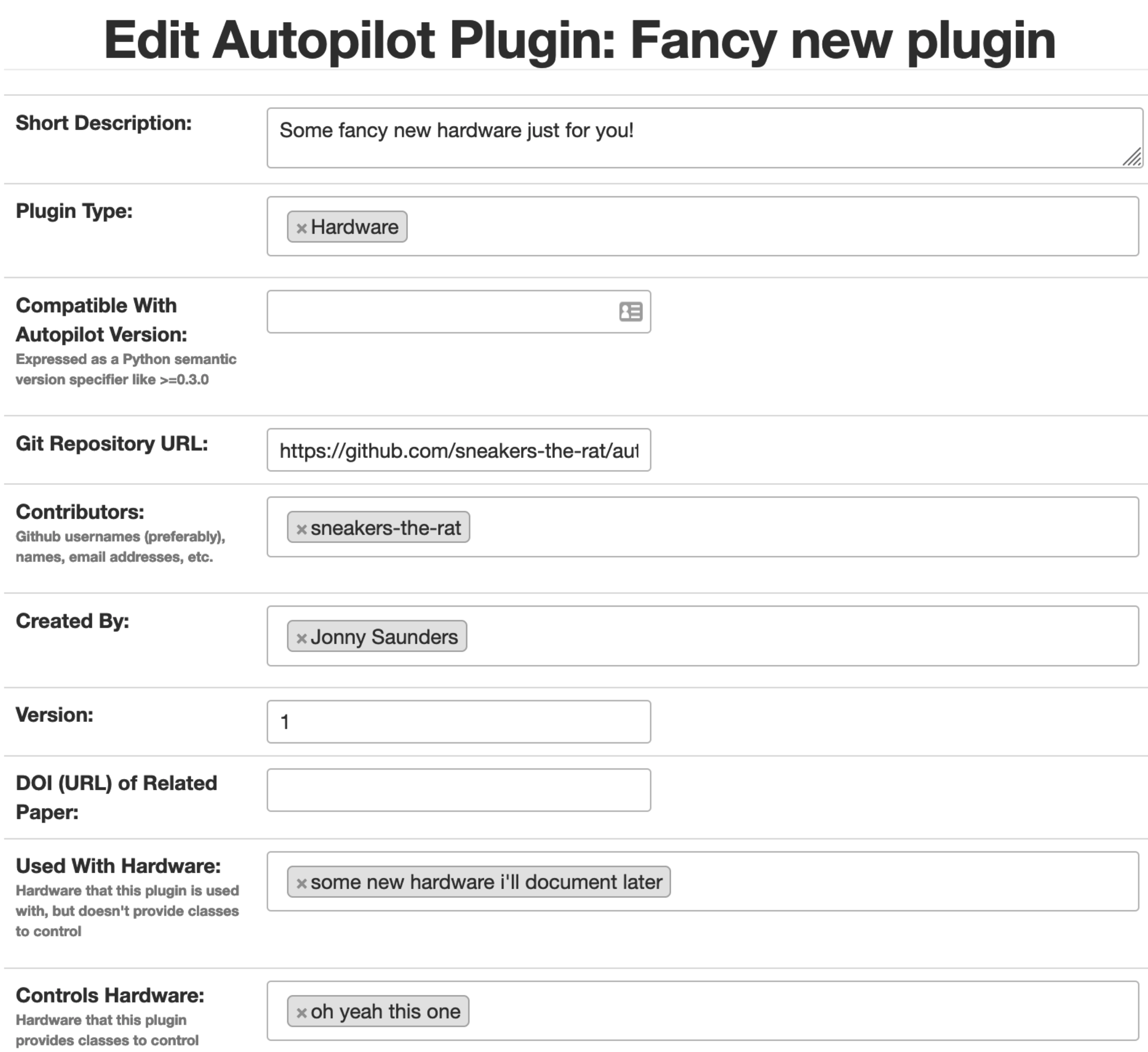
To submit new plugin, one would use the relevant form: https://wiki.auto-pi-lot.com/index.php/Form:Autopilot_Plugin

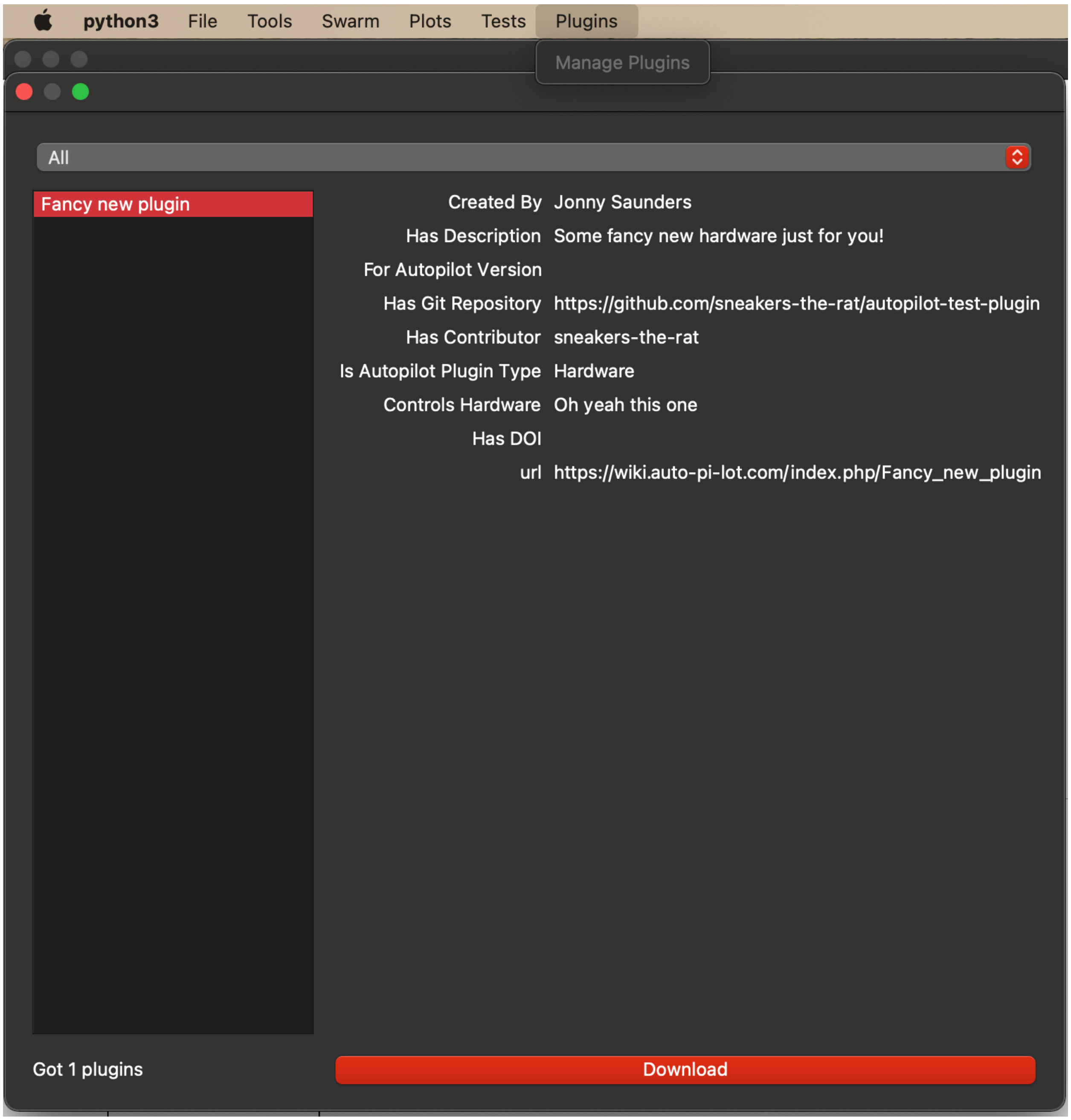
So we might submit our plugin "Fancy New Plugin" (by entering that on the form entry page), and filling in the fields in the form as requested:

Where we provide a description and other metadata – most important some git repository url – that describes the plugin. There are free text fields where appropriate, but also autocompleting token fields that let us keep some semblance of consistency in the semantic links we create. At the end you are then given a free-text field that accepts all common wiki markup as well as free declaration of any semantic links that aren't asked for in the form.

After you submit, it's immediately available in the `gui.Plugins` manager!

Each plugin has one or multiple `Plugin Type(s)` that corresponds to a particular entry in `REGISTRIES` for filtering plugins that provide different types of objects.


Edit Autopilot Plugin: Fancy new plugin
Short Description:
Some fancy new hardware just for you!
Plugin Type:
Hardware
Compatible With Autopilot Version:
Expressed as a Python semantic version specifier like >=0.3.0
Git Repository URL:
https://github.com/sneakers-the-rat/aut
Contributors:
Github usernames (preferably), names, email addresses, etc.
sneakers-the-rat
Created By:
Jonny Saunders
Version:
1
DOI (URL) of Related Paper:
Used With Hardware:
Hardware that this plugin is used with, but doesn't provide classes to control
some new hardware i'll document later
Controls Hardware:
Hardware that this plugin provides classes to control
oh yeah this one


python3
File
Tools
Swarm
Plots
Tests
Plugins
Manage Plugins
All
Fancy new plugin
Created By Jonny Saunders
Has Description Some fancy new hardware just for you!
For Autopilot Version
Has Git Repository https://github.com/sneakers-the-rat/autopilot-test-plugin
Has Contributor sneakers-the-rat
Is Autopilot Plugin Type Hardware
Controls Hardware Oh yeah this one
Has DOI
url https://wiki.auto-pi-lot.com/index.php/Fancy_new_plugin
Got 1 plugins
Download

Todo: Currently the plugin manager is just a proof of concept, though it would require relatively little to add a routine to clone the git repo into the plugins directory, as mentioned above, we are working on integrating dependency management in a way that's unified throughout the package (instead of, say, needing to manually run `python -m autopilot.setup.run_script picamera` to enable the camera, objects are able to specify and request that their dependencies be met automatically).

For now just `git clone <plugin_url> ~/autopilot/plugins` or wherever your `PLUGINDIR` is!

CHAPTER

TEN

EXAMPLES

We're working on writing more examples! Please let us know in the discussion board what you'd like to see :)

Also see the `examples` folder in the repository for jupyter notebooks we haven't set up Sphinx rendering for yet ;)

10.1 Blink

A very simple task: Blink an LED

Written by @mikewehr in the `mike` branch: https://github.com/wehr-lab/autopilot/blob/mike/autopilot/tasks/blink.py

Demonstrates the basic structure of a task with one stage, described in the comments throughout the task.

This page is rendered in the docs here in order to provide links to the mentioned objects/classes/etc., but it was written as source code initially and translated to .rst, so the narrative flow is often inverted: text follows code as comments, rather than text introducing and narrating code.

10.1.1 Preamble

```
import itertools
import tables
import time
from datetime import datetime

from autopilot.hardware import gpio
from autopilot.tasks import Task
from collections import OrderedDict as odict

class Blink(Task):
    """
    Blink an LED.

    Args:
        pulse_duration (int, float): Duration the LED should be on, in ms
        pulse_interval (int, float): Duration the LED should be off, in ms

    """
```

Note that we subclass the `Task` class (`Blink(Task)`) to provide us with some methods useful for all Tasks.

Tasks need to have a few class attributes defined to be integrated into the rest of the system See here for more about class vs. instance attributes https://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide

Params

```
STAGE_NAMES = ["pulse"] # type: list
"""
An (optional) list or tuple of names of methods that will be used as stages for the task.

See ``stages`` for more information
"""

PARAMS = odict()
PARAMS['pulse_duration'] = {'tag': 'LED Pulse Duration (ms)', 'type': 'int'}
PARAMS['pulse_interval'] = {'tag': 'LED Pulse Interval (ms)', 'type': 'int'}
```

`PARAMS` - A dictionary that specifies the parameters that control the operation of the task – each task presumably has some range of options that allow slight variations (eg. different stimuli, etc.) on a shared task structure. This dictionary specifies each `PARAM` as a human-readable `tag` and a `type` that is used by the gui to create an appropriate input object. For example:

```
PARAMS['pulse_duration'] = {'tag': 'LED Pulse Duration (ms)', 'type': 'int'}
```

When instantiated, these params are passed to the `__init__` method.

A `collections.OrderedDict` is used so that parameters can be presented in a predictable way to users.

TrialData

```
class TrialData(tables.IsDescription):
    trial_num = tables.Int32Col()
    timestamp_on = tables.StringCol(26)
    timestamp_off = tables.StringCol(26)
```

`TrialData` declares the data that will be returned for each "trial" – or complete set of executed task stages. It is used by the *`Subject`* object to make a data table with the correct data types. Declare each piece of data using a pytables Column descriptor (see https://www.pytables.org/usersguide/libref/declarative_classes.html#col-sub-classes for available data types, and the pytables guide: https://www.pytables.org/usersguide/tutorials.html for more information)

For each trial, we'll return two timestamps, the time we turned the LED on, the time we turned it off, and the trial number. Note that we use a 26-character `tables.StringCol` for the timestamps,

Hardware

```
HARDWARE = {
    'LEDS': {
        'dLED': gpio.Digital_Out
    }
}
```

Declare the hardware that will be used in the task. Each hardware object is specified with a `group` and an `id` as nested dictionaries. These descriptions require a set of hardware parameters in the autopilot `prefs.json` (typically generated by *`autopilot.setup.setup_autopilot`*) with a matching `group` and `id` structure. For example, an LED declared like this in the `HARDWARE` attribute:

```
HARDWARE = {'LEDS': {'dLED': gpio.Digital_Out}}
```

requires an entry in `prefs.json` like this:

```
"HARDWARE": {"LEDS": {"dLED": {
    "pin": 1,
    "polarity": 1
}}}
```

that will be used to instantiate the *hardware.gpio.Digital_Out* object, which is then available for use in the task like:

```
self.hardware['LEDS']['dLED'].set(1)
```

10.1.2 Initialization

first we call the superclass ('Task')'s initialization method. All tasks should accept `*args` and `**kwargs` to pass parameters not explicitly specified by subclass up to the superclass.:

```
def __init__(self, stage_block=None, pulse_duration=100, pulse_interval=500, *args,␣
↪**kwargs):
    super(Blink, self).__init__(*args, **kwargs)

    # store parameters given on instantiation as instance attributes
    self.pulse_duration = int(pulse_duration)
    self.pulse_interval = int(pulse_interval)
    self.stage_block = stage_block # type: "threading.Event"

    # This allows us to cycle through the task by just repeatedly calling self.stages.
↪next()
    self.stages = itertools.cycle([self.pulse])
```

Some generator that returns the stage methods that define the operation of the task.

To run a task, the *pilot.Pilot* object will call each stage function, which can return some dictionary of data (see `pulse()`) and wait until some flag (`stage_block`) is set to compute the next stage. Since in this case we want to call the same method (`pulse()`) over and over again, we use an `itertools.cycle` object (if we have more than one stage to call in a cycle, we could provide them like `itertools.cycle([self.stage_method_1, self.stage_method_2])` . More complex tasks can define a custom generator for finer control over stage progression.:

```
self.trial_counter = itertools.count()
"""
Some counter to keep track of the trial number
"""
```

Hardware is initialized by the superclass's *Task.init_hardware()* method, which creates all the hardware objects defined in `HARDWARE` according to their parameterization in `prefs.json` , and makes them available in the `hardware` dictionary.:

```
self.init_hardware()
self.logger.debug('Hardware initialized')
```

All task subclass objects have an `logger` – a `logging.Logger` that allows users to easily debug their tasks and see feedback about their operation. To prevent stdout from getting clogged, logging messages are printed and stored according to the `LOGLEVEL` pref – so this message would only appear if `LOGLEVEL == "DEBUG"`:

```
self.stage_block.set()
```

We set the stage block and never clear it so that the *Pilot* doesn't wait for a trigger to call the next stage – it just does it as soon as the previous one completes.

See *run_task()* for more detail on this loop.

10.1.3 Stage Methods

```
def pulse(self, *args, **kwargs):
    """
    Turn an LED on and off according to :attr:`~examples.tasks.Blink.pulse_duration` and␣
↪:attr:`~examples.tasks.Blink.pulse_interval`

    Returns:
        dict: A dictionary containing the trial number and two timestamps.
    """
    # -------------
    # turn light on

    # use :meth:`.hardware.gpio.Digital_Out.set` method to turn the LED on
    self.hardware['LEDS']['dLED'].set(1)
    # store the timestamp
    timestamp_on = datetime.now().isoformat()
    # log status as a debug message
    self.logger.debug('light on')
    # sleep for the pulse_duration
    time.sleep(self.pulse_duration / 1000)

    # ------------
    # turn light off, same as turning it on.

    self.hardware['LEDS']['dLED'].set(0)
    timestamp_off = datetime.now().isoformat()
    self.logger.debug('light off')
    time.sleep(self.pulse_interval / 1000)

    # count and store the number of the current trial
    self.current_trial = next(self.trial_counter)


    data = {
        'trial_num': self.current_trial,
        'timestamp_on': timestamp_on,
        'timestamp_off': timestamp_off
    }
    return data
```

Create the data dictionary to be returned from the stage. Note that each of the keys in the dictionary must correspond to the names of the columns declared in the `TrialData` descriptor.

At the conclusion of running the task, we will be able to access the data from the run with *Subject.get_trial_data()*, which will be a `pandas.DataFrame` with a row for each trial, and a column for each of the fields here.

10.1.4 Full Source

```
"""
A very simple task: Blink an LED

Written by @mikewehr in the ``mike`` branch: https://github.com/wehr-lab/autopilot/blob/mike/autopilot/tasks/blink.py

Demonstrates the basic structure of a task with one stage,
described in the comments throughout the task.

See the main tutorial for more detail: https://docs.auto-pi-lot.com/en/latest/guide.task.html#

This page is rendered in the docs here in order to provide links to the mentioned objects/classes/etc., but
this example was intended to be read as source code, as some comments will only be visible there.
"""
import itertools
import tables
import time
from datetime import datetime

from autopilot.hardware import gpio
from autopilot.tasks import Task
from collections import OrderedDict as odict

class Blink(Task):
    """
    Blink an LED.

    Note that we subclass the :class:`~autopilot.tasks.Task` class (``Blink(Task)``) to provide us with some methods
    useful for all Tasks.

    Args:
        pulse_duration (int, float): Duration the LED should be on, in ms
        pulse_interval (int, float): Duration the LED should be off, in ms

    """
    # Tasks need to have a few class attributes defined to be integrated into the rest of the system
    # See here for more about class vs. instance attributes https://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide

    STAGE_NAMES = ["pulse"] # type: list
    """
```

(continues on next page)

(continued from previous page)

```
    An (optional) list or tuple of names of methods that will be used as stages for the task.

    See :attr:`~examples.tasks.Blink.stages` for more information
    """

    PARAMS = odict()
    """
    A dictionary that specifies the parameters that control the operation of the task -- each task presumably has some
    range of options that allow slight variations (eg. different stimuli, etc.) on a shared task structure. This
    dictionary specifies each ``PARAM`` as a human-readable ``tag`` and a ``type`` that is used by the gui to
    create an appropriate input object. For example::

        PARAMS['pulse_duration'] = {'tag': 'LED Pulse Duration (ms)', 'type': 'int'}

    When instantiated, these params are passed to the ``__init__`` method.

    A :class:`collections.OrderedDict` is used so that parameters can be presented in a predictable way to users.
    """
    PARAMS['pulse_duration'] = {'tag': 'LED Pulse Duration (ms)', 'type': 'int'}
    PARAMS['pulse_interval'] = {'tag': 'LED Pulse Interval (ms)', 'type': 'int'}

    class TrialData(tables.IsDescription):
        """
        This class declares the data that will be returned for each "trial" -- or complete set of executed task
        stages. It is used by the :class:`~autopilot.core.subject.Subject` object to make a data table with the
        correct data types. Declare each piece of data using a pytables Column descriptor
        (see https://www.pytables.org/usersguide/libref/declarative_classes.html#col-sub-classes for available
        data types, and the pytables guide: https://www.pytables.org/usersguide/tutorials.html for more information)

        For each trial, we'll return two timestamps, the time we turned the LED on, the time we turned it off,
        and the trial number. Note that we use a 26-character :class:`tables.StringCol` for the timestamps,
        which are given as an isoformatted string like ``'2021-02-16T18:11:35.752110'``
        """
        trial_num = tables.Int32Col()
        timestamp_on = tables.StringCol(26)
        timestamp_off = tables.StringCol(26)


    HARDWARE = {
        'LEDS': {
            'dLED': gpio.Digital_Out
```

(continues on next page)

(continued from previous page)

```
        }
    }
    """
    Declare the hardware that will be used in the task. Each hardware object is␣
↪specified with a ``group`` and
    an ``id`` as nested dictionaries. These descriptions require a set of hardware␣
↪parameters in the autopilot
    ``prefs.json`` (typically generated by :mod:`autopilot.setup.setup_autopilot` ) with a␣
↪matching ``group`` and
    ``id`` structure. For example, an LED declared like this in the :attr:`~examples.tasks.
↪Blink.HARDWARE` attribute::

        HARDWARE = {'LEDS': {'dLED': gpio.Digital_Out}}

    requires an entry in ``prefs.json`` like this::

        "HARDWARE": {"LEDS": {"dLED": {
            "pin": 1,
            "polarity": 1
        }}}

    that will be used to instantiate the :class:`.hardware.gpio.Digital_Out` object,␣
↪which is then available for use
    in the task like::

        self.hardware['LEDS']['dLED'].set(1)
    """

    def __init__(self, stage_block=None, pulse_duration=100, pulse_interval=500, *args,␣
↪**kwargs):
        # first we call the superclass ('Task')'s initialization method. All tasks should␣
↪accept ``*args``
        # and ``**kwargs`` to pass parameters not explicitly specified by subclass up to␣
↪the superclass.
        super(Blink, self).__init__(*args, **kwargs)

        # store parameters given on instantiation as instance attributes
        self.pulse_duration = int(pulse_duration)
        self.pulse_interval = int(pulse_interval)
        self.stage_block = stage_block # type: "threading.Event"

        # This allows us to cycle through the task by just repeatedly calling self.
↪stages.next()
        self.stages = itertools.cycle([self.pulse])
        """
        Some generator that returns the stage methods that define the operation of the␣
↪task.

        To run a task, the :class:`.pilot.Pilot` object will call each stage function,␣
↪which can return some dictionary
        of data (see :meth:`~examples.tasks.Blink.pulse` ) and wait until some flag␣
↪(:attr:`~examples.tasks.Blink.stage_block` ) is set to compute the
```

(continues on next page)

(continued from previous page)

```
        next stage. Since in this case we want to call the same method (:meth:`~examples.tasks.Blink.pulse` ) over and over again,
        we use an :class:`itertools.cycle` object (if we have more than one stage to call in a cycle, we could provide
        them like ``itertools.cycle([self.stage_method_1, self.stage_method_2])`` . More complex tasks can define a custom
        generator for finer control over stage progression.
        """

        self.trial_counter = itertools.count()
        """
        Some counter to keep track of the trial number
        """


        self.init_hardware()

        """
        Hardware is initialized by the superclass's :meth:`.Task.init_hardware` method, which creates all the
        hardware objects defined in :attr:`~examples.tasks.Blink.HARDWARE` according to their parameterization in
        ``prefs.json`` , and makes them available in the :attr:`~examples.tasks.Blink.hardware` dictionary.
        """

        self.logger.debug('Hardware initialized')

        """
        All task subclass objects have an :attr:`~autopilot.tasks.Task.logger` -- a :class:`logging.Logger` that allows
        users to easily debug their tasks and see feedback about their operation. To prevent stdout from
        getting clogged, logging messages are printed and stored according to the ``LOGLEVEL`` pref -- so this
        message would only appear if ``LOGLEVEL == "DEBUG"``
        """

        self.stage_block.set()

        """
        We set the stage block and never clear it so that the :class:`.Pilot` doesn't wait for a trigger
        to call the next stage -- it just does it as soon as the previous one completes.

        See :meth:`~autopilot.core.pilot.Pilot.run_task` for more detail on this loop.
        """


    ##################################################################################
    # Stage Functions
    ##################################################################################
```

(continues on next page)

(continued from previous page)

```
    def pulse(self, *args, **kwargs):
        """
        Turn an LED on and off according to :attr:`~examples.tasks.Blink.pulse_duration`␣
↪and :attr:`~examples.tasks.Blink.pulse_interval`

        Returns:
            dict: A dictionary containing the trial number and two timestamps.
        """
        # -------------
        # turn light on

        # use :meth:`.hardware.gpio.Digital_Out.set` method to turn the LED on
        self.hardware['LEDS']['dLED'].set(1)
        # store the timestamp
        timestamp_on = datetime.now().isoformat()
        # log status as a debug message
        self.logger.debug('light on')
        # sleep for the pulse_duration
        time.sleep(self.pulse_duration / 1000)

        # ------------
        # turn light off, same as turning it on.

        self.hardware['LEDS']['dLED'].set(0)
        timestamp_off = datetime.now().isoformat()
        self.logger.debug('light off')
        time.sleep(self.pulse_interval / 1000)

        # count and store the number of the current trial
        self.current_trial = next(self.trial_counter)


        data = {
            'trial_num': self.current_trial,
            'timestamp_on': timestamp_on,
            'timestamp_off': timestamp_off
        }

        """
        Create the data dictionary to be returned from the stage. Note that each of the␣
↪keys in the dictionary
        must correspond to the names of the columns declared in the :attr:`~examples.
↪tasks.Blink.TrialData` descriptor.

        At the conclusion of running the task, we will be able to access the data from␣
↪the run with
        :meth:`.Subject.get_trial_data`, which will be a :class:`pandas.DataFrame` with a␣
↪row for each trial, and
        a column for each of the fields here.
        """

        # return the data dictionary from the stage method  and yr done :)
```

(continues on next page)

(continued from previous page)

```
        return data
```

CHAPTER

ELEVEN

CORE

11.1 gui

These classes implement the GUI used by the Terminal.

The GUI is built using PySide2, a Python wrapper around Qt5.

These classes are all currently used only by the `Terminal`.

If performing any GUI operations in another thread (eg. as a callback from a networking object), the method must be decorated with *@gui_event* which will call perform the update in the main thread as required by Qt.

Note: Currently, the GUI code is some of the oldest code in the library – in particular much of it was developed before the network infrastructure was mature. As a result, a lot of modules are interdependent (eg. pass objects between each other). This will be corrected before v1.0

Data:

`_MAPS`	Maps of shorthand names for objects to the objects themselves.

Functions:

`gui_event`(fn)	Wrapper/decorator around an event that posts GUI events back to the main thread that our window is running in.
`pop_dialog`(message[, details, buttons, …])	Convenience function to pop a :class:`.QtGui.QDialog window to display a message.

Classes:

`Control_Panel`(subjects, start_fn, ping_fn, …)	A `QtWidgets.QWidget` that contains the controls for all pilots.
`Subject_List`([subjects, drop_fn])	A trivial modification of `QListWidget` that updates `pilots` when an item in the list is dragged to another location.
`Pilot_Panel`([pilot, subject_list, start_fn, …])	A little panel with
`Pilot_Button`([pilot, subject_list, …])	A subclass of (toggled) `QtWidgets.QPushButton` that incorporates the style logic of a start/stop button - ie.

continues on next page

Table 3 – continued from previous page

`New_Subject_Wizard`()	A popup that prompts you to define variables for a new `subject.Subject` object
`Protocol_Wizard`()	A dialog window to create a new protocol.
`Graduation_Widget`()	A widget used in `Protocol_Wizard` to define graduation parameters.
`Drag_List`()	A `QtWidgets.QListWidget` that is capable of having files dragged & dropped.
`Sound_Widget`()	A widget that allows sounds to be parameterized.
`Bandwidth_Test`(pilots)	Test the limits of the rate of messaging from the connected Pilots.
`Calibrate_Water`(pilots)	A window to calibrate the volume of water dispensed per ms.
`Pilot_Ports`(pilot[, n_clicks, click_dur])	Created by `Calibrate_Water`, Each pilot's ports and buttons to control repeated release.
`Reassign`(subjects, protocols)	A dialog that lets subjects be batch reassigned to new protocols or steps.
`Weights`(subject_weights, subjects)	A table for viewing and editing the most recent subject weights.
`Plugins`()	Dialog window that allows plugins to be viewed and installed.
`Psychometric`(subjects_protocols)	A Dialog to select subjects, steps, and variables to use in a psychometric curve plot.
`Stream_Video`(pilots, *args, **kwargs)	Dialogue to stream, display, and save video.

`_MAPS = { 'dialog': { 'icon': { 'error': PySide2.QtWidgets.QMessageBox.Icon.Critical, 'info': PySide2.QtWidgets.QMessageBox.Icon.Information, 'question': PySide2.QtWidgets.QMessageBox.Icon.Question, 'warning': PySide2.QtWidgets.QMessageBox.Icon.Warning}, 'modality': { 'modal': PySide2.QtCore.Qt.WindowModality.NonModal, 'nonmodal': PySide2.QtCore.Qt.WindowModality.WindowModal}}}`

Maps of shorthand names for objects to the objects themselves.

Grouped by a rough use case, intended for internal (rather than user-facing) use.

`gui_event`(*fn*)

Wrapper/decorator around an event that posts GUI events back to the main thread that our window is running in.

Parameters **fn** (*callable*) – a function that does something to the GUI

`class Control_Panel`(*subjects*, *start_fn*, *ping_fn*, *pilots*)

Bases: `PySide2.QtWidgets.QWidget`

A `QtWidgets.QWidget` that contains the controls for all pilots.

Parameters

- **subjects** (*dict*) – See `Control_Panel.subjects`
- **start_fn** (`toggle_start()`) – the Terminal's toggle_start function, propagated down to each `Pilot_Button`
- **pilots** – Usually the Terminal's `pilots` dict. If not passed, will try to load `params.PILOT_DB`

Variables

- **`subjects`** (*dict*) – A dictionary with subject ID's as keys and `core.subject.Subject` objects as values. Shared with the Terminal object to manage access conflicts.

- **start_fn** (*toggle_start()*) – See Control_Panel.start_fn
- **pilots** (*dict*) – A dictionary with pilot ID's as keys and nested dictionaries containing subjects, IP, etc. as values
- **subject_lists** (*dict*) – A dict mapping subject ID to subject_List
- **layout** (QGridLayout) – Layout grid for widget
- **panels** (*dict*) – A dict mapping pilot name to the relevant *Pilot_Panel*

Specifically, for each pilot, it contains

- one subject_List: A list of the subjects that run in each pilot.
- one *Pilot_Panel*: A set of button controls for starting/stopping behavior

This class should not be instantiated outside the context of a *Terminal* object, as they share the subjects dictionary.

Methods:

init_ui()	Called on init, creates the UI components.
add_pilot(pilot_id[, subjects])	Add a *Pilot_Panel* for a new pilot, and populate a *Subject_List* for it :Parameters: * **pilot_id** (*str*) – ID of new pilot * **subjects** (*list*) – Optional, list of any subjects that the pilot has.
create_subject(pilot)	Becomes Pilot_Panel.create_fn.
update_db([pilots])	Gathers any changes in *Subject_List* s and dumps *pilots* to prefs.get('PILOT_DB')

Attributes:

staticMetaObject

init_ui()
Called on init, creates the UI components.

Specifically, for each pilot in *pilots*, make a subject_List: and *Pilot_Panel*:, set size policies and connect Qt signals.

add_pilot(*pilot_id: str*, *subjects: Optional[list] = None*)
Add a *Pilot_Panel* for a new pilot, and populate a *Subject_List* for it :Parameters: * **pilot_id** (*str*) – ID of new pilot

- **subjects** (*list*) – Optional, list of any subjects that the pilot has.

Returns:

create_subject(*pilot*)
Becomes Pilot_Panel.create_fn. Opens a *New_Subject_Wizard* to create a new subject file and assign protocol. Finally, adds the new subject to the pilots database and updates it.

Parameters **pilot** (*str*) – Pilot name passed from *Pilot_Panel*, added to the created Subject object.

update_db(*pilots: Optional[dict] = None*, ***kwargs*)
Gathers any changes in *Subject_List* s and dumps *pilots* to prefs.get('PILOT_DB')

Parameters **kwargs** – Create new pilots by passing a dictionary with the structure *new={'pilot_name':'pilot_values'}*

where *'pilot_values'* can be nothing, a list of subjects, or any other information included in the pilot db

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce74608e60>

class Subject_List(*subjects=None*, *drop_fn=None*)

Bases: `PySide2.QtWidgets.QListWidget`

A trivial modification of `QListWidget` that updates *pilots* when an item in the list is dragged to another location.

Should not be initialized except by *Control_Panel* .

Variables

- **subjects** (*list*) – A list of subjects ID's passed by *Control_Panel*
- **drop_fn** (*Control_Panel.update_db()*) – called on a drop event

Parameters

- **subjects** – see `subjects`. Can be *None* for an empty list
- **drop_fn** – see `drop_fn()`. Passed from *Control_Panel*

Methods:

populate_list()	Adds each item in `Subject_List.subjects` to the list.
dropEvent(event)	A trivial redefinition of `QtWidgets.QListWidget.dropEvent()` that calls the parent *dropEvent* and then calls `drop_fn`

Attributes:

staticMetaObject

populate_list()

Adds each item in `Subject_List.subjects` to the list.

dropEvent(*event*)

A trivial redefinition of `QtWidgets.QListWidget.dropEvent()` that calls the parent *dropEvent* and then calls `drop_fn`

Parameters **event** – A `QtCore.QEvent` simply forwarded to the superclass.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce743c5e60>

class Pilot_Panel(*pilot=None*, *subject_list=None*, *start_fn=None*, *ping_fn=None*, *create_fn=None*)

Bases: `PySide2.QtWidgets.QWidget`

A little panel with

- the name of a pilot,
- A *Pilot_Button* to start and stop the task
- Add and remove buttons to *create_subject()* and *Pilot_Panel.remove_subject()*

Note: This class should not be instantiated except by *Control_Panel*

Parameters

- **pilot** (*str*) – The name of the pilot this panel controls
- **subject_list** (*Subject_List*) – The *Subject_List* we control
- **start_fn** (*toggle_start()*) – Passed by *Control_Panel*
- **create_fn** (*Control_Panel.create_subject()*) – Passed by *Control_Panel*

Variables

- **layout** (QtWidgets.QGridLayout) – Layout for UI elements
- **button** (*Pilot_Button*) – button used to control a pilot

Methods:

`init_ui()`	Initializes UI elements - creates widgets and adds to `Pilot_Panel.layout` .
`remove_subject()`	Remove the currently selected subject in `Pilot_Panel.subject_list`, and calls the *Control_Panel.update_db()* method.
`create_subject()`	Just calls *Control_Panel.create_subject()* with our *pilot* as the argument

Attributes:

`staticMetaObject`

init_ui()
Initializes UI elements - creates widgets and adds to `Pilot_Panel.layout` . Called on init.

remove_subject()
Remove the currently selected subject in `Pilot_Panel.subject_list`, and calls the *Control_Panel.update_db()* method.

create_subject()
Just calls *Control_Panel.create_subject()* with our *pilot* as the argument

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce743c5d70>

class **Pilot_Button**(*pilot=None*, *subject_list=None*, *start_fn=None*, *ping_fn=None*)
Bases: `PySide2.QtWidgets.QPushButton`

A subclass of (toggled) `QtWidgets.QPushButton` that incorporates the style logic of a start/stop button - ie. color, text.

Starts grayed out, turns green if contact with a pilot is made.

Parameters

- **pilot** (*str*) – The ID of the pilot that this button controls
- **subject_list** (*Subject_List*) – The Subject list used to determine which subject is starting/stopping

- **start_fn** (*toggle_start()*) – The final resting place of the toggle_start method

Variables state (*str*) – The state of our pilot, reflected in our graphical properties. Mirrors `state` , with an additional "DISCONNECTED" state for before contact is made with the pilot.

Methods:

`toggle_start`()	Minor window dressing to call the `start_fn()` with the appropriate pilot, subject, and whether the task is starting or stopping
`set_state`(state)	Set the button's appearance and state

Attributes:

`staticMetaObject`	

toggle_start()
Minor window dressing to call the `start_fn()` with the appropriate pilot, subject, and whether the task is starting or stopping

set_state(*state*)
Set the button's appearance and state

Parameters state (*str*) – one of ``('IDLE', 'RUNNING', 'STOPPING', 'DISCONNECTED')

Todo: There is some logic duplication in this class, ie. if the button state is changed it also emits a start/stop signal to the pi, which is undesirable. This class needs to be reworked.

Returns:

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce743c5d20>

class New_Subject_Wizard
Bases: `PySide2.QtWidgets.QDialog`

A popup that prompts you to define variables for a new *subject.Subject* object

Called by *Control_Panel.create_subject()* , which handles actually creating the subject file and updating the *Terminal.pilots* dict and file.

Contains two tabs - *Biography_Tab* - to set basic biographical information about a subject - *Task_Tab* - to set the protocol and step to start the subject on

Variables

- **protocol_dir** (*str*) – A full path to where protocols are stored, received from `prefs.get('PROTOCOLDIR')`
- **bio_tab** (*Biography_Tab*) – Sub-object to set and store biographical variables
- **task_tab** (*Task_Tab*) – Sub-object to set and store protocol and step assignment

Classes:

`Biography_Tab`()	A widget that allows defining basic biographical attributes about a subject

continues on next page

Table 12 – continued from previous page

Task_Tab()	A tab for selecting a task and step to assign to the subject.

Attributes:

staticMetaObject	

class Biography_Tab

Bases: PySide2.QtWidgets.QWidget

A widget that allows defining basic biographical attributes about a subject

Creates a set of widgets connected to *update_return_dict()* that stores the parameters.

> **Warning:** The below attributes are **not** the object attributes, but are descriptions of the parameters available in the values dictionary. The attributes themselves are PySide Widgets that set the values.

Variables

- **id** (*str*) – A Subject's ID or name
- **start_date** (*str*) – The date the subject started the task. Automatically filled by datetime.date.today().isoformat()
- **blmass** (*float*) – The subject's baseline mass
- **minmass_pct** (*int*) – The percentage of baseline mass that a water restricted subject is allowed to reach
- **minmass** (*float*) – The subject's minimum mass, automatically calculated *blmass * (minmass_pct / 100.)*
- **genotype** (*str*) – A string describing the subject's genotype
- **expt** (*str*) – A tag to describe what experiment this subject is a part of

Methods:

update_return_dict(key, val)	Called by lambda functions by the widgets, eg..
calc_minmass()	Calculates the minimum mass for a subject based on its baseline mass and the allowable percentage of that baseline

Attributes:

staticMetaObject	

update_return_dict(*key, val*)

Called by lambda functions by the widgets, eg.:

```
self.id.editingFinished.connect(lambda: self.update_return_dict('id', self.
↪id.text()))
```

Parameters
- **key** (*str*) – The key of the value being stored
- **val** – The value being stored.

calc_minmass()
Calculates the minimum mass for a subject based on its baseline mass and the allowable percentage of that baseline

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1370>

class Task_Tab
Bases: PySide2.QtWidgets.QWidget

A tab for selecting a task and step to assign to the subject.

Reads available tasks from *prefs.get('PROTOCOLDIR')* , lists them, and creates a spinbox to select from the available steps.

> **Warning:** Like *Biography_Tab* , these are not the actual instance attributes. Values are stored in a *values* dictionary.

Variables
- **protocol** (*str*) – the name of the assigned protocol, filename without .json extension
- **step** (*int*) – current step to assign.

Methods:

update_step_box()	Clears any steps that might be in the step selection box, loads the protocol file and repopulates it.
protocol_changed()	When the protocol is changed, save the value and call *update_step_box()*.
step_changed()	When the step is changed, save it.

Attributes:

staticMetaObject	

update_step_box()
Clears any steps that might be in the step selection box, loads the protocol file and repopulates it.

protocol_changed()
When the protocol is changed, save the value and call *update_step_box()*.

step_changed()
When the step is changed, save it.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1320>

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d12d0>

class Protocol_Wizard
Bases: PySide2.QtWidgets.QDialog

A dialog window to create a new protocol.

Warning: This is a heavily overloaded class, and will be split into separate objects to handle parameters separately. For now this is what we got though and it works.

Protocols are collections of multiple tasks (steps) with some graduation criterion for moving between them.

This widget is composed of three windows:

- **left**: possible task types from `autopilot.get_task()`
- **center**: current steps in task
- **right**: `Parameters` for currently selected step.

The parameters that are used are of the form used by *Task.PARAMS* (see *Nafc.PARAMS* for an example).

Todo: Make specific parameter class so this definition is less squishy

its general structure is:

```
{'parameter_key': {'tag':'Human Readable Name',
                   'type':'param_type'}}
```

while some parameter types have extra items, eg.:

```
{'list_param': {'tag':'Select from a List of Parameters',
                'type': 'list',
                'values': {'First Option':0, 'Second Option':1}}
```

where k:v pairs are still used with lists to allow parameter values (0, 1) be human readable.

The available types include:

- **int** - integer
- **float** - floating point number
- **bool** - boolean boolbox
- **list** - a list of *values* to choose from
- **sounds** - a *Sound_Widget* that allows sounds to be defined.
- **graduation** - a *Graduation_Widget* that allows graduation criteria to be defined

Variables

- **task_list** (`QtWidgets.QListWidget`) – The leftmost window, lists available tasks
- **step_list** (`QtWidgets.QListWidget`) – The center window, lists tasks currently in protocol
- **param_layout** (`QtWidgets.QFormLayout`) – The right window, allows changing available parameters for currently selected step.
- **steps** (*list*) – A list of dictionaries defining the protocol.

Methods:

add_step()	Loads *PARAMS* from task object, adds base parameters to steps list
rename_step()	When the step name widget's text is changed, fire this function to update step_list which updates steps
remove_step()	Remove step from step_list and steps
populate_params()	Calls clear_params() and then creates widgets to edit parameter values.
clear_params()	Clears widgets from parameter window
reorder_steps(*args)	When steps are dragged into a different order, update the step dictionary
set_param()	Callback function connected to the signal each widget uses to signal it has changed.
set_sounds()	Stores parameters that define sounds.
set_graduation()	Stores parameters that define graduation criteria in *self.steps*
check_depends()	Handle dependencies between parameters, eg.

Attributes:

staticMetaObject

add_step()
Loads *PARAMS* from task object, adds base parameters to steps list

rename_step()
When the step name widget's text is changed, fire this function to update step_list which updates

steps

remove_step()
Remove step from step_list and steps

populate_params()
Calls clear_params() and then creates widgets to edit parameter values. Returns:

clear_params()
Clears widgets from parameter window

reorder_steps(**args*)
When steps are dragged into a different order, update the step dictionary

Parameters ***args** – Input from our step_list 's QtWidgets.QListModel 's reorder signal.

set_param()
Callback function connected to the signal each widget uses to signal it has changed.

Identifies the param that was changed, gets the current value, and updates *self.steps*

set_sounds()
Stores parameters that define sounds.

Sound parameters work a bit differently, specifically we have to retrieve Sound_Widget.sound_dict.

set_graduation()
Stores parameters that define graduation criteria in *self.steps*

Graduation parameters work a bit differently, specifically we have to retrieve `Graduation_Widget.param_dict`.

check_depends()
Handle dependencies between parameters, eg. if "correction trials" are unchecked, the box that defines the correction trial percentage should be grayed out.

Todo: Not implemented.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1280>

class Graduation_Widget
Bases: `PySide2.QtWidgets.QWidget`

A widget used in *Protocol_Wizard* to define graduation parameters.

See *tasks.graduation* .

A protocol is composed of multiple tasks (steps), and graduation criteria define when a subject should progress through those steps.

eg. a subject should graduate one stage after 300 trials, or after it reaches 75% accuracy over the last 500 trials.

Variables

- **type_selection** (`QtWidgets.QComboBox`) – A box to select from the available graduation types listed in `autopilot.get_task()` . Has its *currentIndexChanged* signal connected to *Graduation_Widget.populate_params()*
- **param_dict** (*dict*) – Stores the type of graduation and the relevant params, fetched by *Protocol_Wizard* when defining a protocol.
- **set_graduation** (*Protocol_Wizard.set_graduation()*) – Passed to us after we're inited.

Methods:

`populate_params`([params])	Repopulate the widget with fields to edit graduation parameters, fill fields if we are passed *params*.
`clear_params`()	Clear any parameter widgets we have.
`store_param`()	When a parameter is edited, save it in our param_dict, and also call our *set_graduation* method, which should be *Protocol_Wizard.set_graduation()* passed to us after instantiation.

Attributes:

`staticMetaObject`

populate_params(*params=None*)
Repopulate the widget with fields to edit graduation parameters, fill fields if we are passed *params*.

Each `QtWidgets.QLineEdit` 's `QLineEdit.editingFinished()` signal is connected to *Graduation_Widget.store_param()* .

Todo: For now we assume all parameters are defined with a text edit box, so it's not clear how we'd do boolean parameters for example. This will be fixed with refactoring the parameter scheme.

Parameters **params** (*dict*) – In the case that `Protocol_Wizard` switches us back to a step where we have already defined graduation parameters, it will pass them so we can repopulate the relevant widgets with them.

clear_params()
Clear any parameter widgets we have.

store_param()
When a parameter is edited, save it in our param_dict, and also call our *set_graduation* method, which should be `Protocol_Wizard.set_graduation()` passed to us after instantiation.

If we were not passed *set_graduation*, just saves in *param_dict*.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1230>

class Drag_List
Bases: `PySide2.QtWidgets.QListWidget`

A `QtWidgets.QListWidget` that is capable of having files dragged & dropped.

copied with much gratitude from stackoverflow

Primarily used in `Sound_Widget` to be able to drop sound files.

To use: connect *fileDropped* to a method, that method will receive a list of files dragged onto this widget.

Variables **fileDropped** (`QtCore.Signal`) – A Qt signal that takes a list

Attributes:

`fileDropped`(*args, **kwargs)	Call self as a function.
`staticMetaObject`	

Methods:

`dragEnterEvent`(e)	When files are dragged over us, if they have paths in them, accept the event.
`dragMoveEvent`(event)	If the *dragEnterEvent* was accepted, while the drag is being moved within us, *setDropAction* to `QtCore.Qt.CopyAction`
`dropEvent`(event)	When the files are finally dropped, if they contain paths, emit the list of paths through the *fileDropped* signal.

fileDropped(**args*, ***kwargs*)
Call self as a function.

dragEnterEvent(*e*)
When files are dragged over us, if they have paths in them, accept the event.

Parameters **e** (`QtCore.QEvent`) – containing the drag information.

dragMoveEvent(*event*)
If the *dragEnterEvent* was accepted, while the drag is being moved within us, *setDropAction* to `QtCore.Qt.CopyAction`

Parameters **event** (QtCore.QEvent) – containing the drag information.

dropEvent(*event*)

When the files are finally dropped, if they contain paths, emit the list of paths through the *fileDropped* signal.

Parameters **event** (QtCore.QEvent) – containing the drag information.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d11e0>

class Sound_Widget

Bases: PySide2.QtWidgets.QWidget

A widget that allows sounds to be parameterized.

Used in *Protocol_Wizard* .

Has two *Drag_List* s for left and right sounds (for a 2afc task), given Buttons beneath them allow adding and removing sounds.

Adding a sound will open a Add_SoundDialog

Todo: Sounds will eventually be more elegantly managed by a … sound manager.. For now sound managers are rudimentary and only support random presentation with correction trials and bias correction.

Variables **sound_dict** (*dict*) – Dictionary with the structure:

```
{'L': [{'param_1':'param_1', ... }], 'R': [...]}
```

where multiple sounds can be present in either 'L' or 'R' list.

Methods:

pass_set_param_function(set_param_fnxn)	Receives *Protocol_Wizard.set_sounds()*
add_sound(side)	When the "+" button on either side is pressed, open an *Add_Sound_Dialog*.
remove_sound(side)	When the "-" button is pressed, remove the currently highlighted sound.
populate_lists(sound_dict)	Populates the sound lists after re-selecting a step.
files_dropped(files)	

Classes:

Add_Sound_Dialog()	Presents a dialog to define a new sound.

Attributes:

staticMetaObject

pass_set_param_function(*set_param_fnxn*)

Receives *Protocol_Wizard.set_sounds()*

Parameters **set_param_fnxn** (*Protocol_Wizard.set_sounds()*) – Called when sounds are changed.

add_sound(*side*)

When the "+" button on either side is pressed, open an *Add_Sound_Dialog*.

Parameters **side** (*str*) – The buttons are connected with a lambda function, this will be either 'L' or 'R'. Used to add sounds to the *sound_dict*

remove_sound(*side*)

When the "-" button is pressed, remove the currently highlighted sound.

Parameters **side** (*str*) – The buttons are connected with a lambda function, this will be either 'L' or 'R'. Selects that list so we can remove the currently selected row.

populate_lists(*sound_dict*)

Populates the sound lists after re-selecting a step.

Parameters **sound_dict** (*dict*) – passed to us by *Protocol_Wizard* upon reselecting a step.

files_dropped(*files*)

> **Warning:** This was programmed hastily and is pretty idiosyncratic to my use.
>
> It does work for general files but has some extra logic built in to handle my stimuli.
>
> To be made more general in v0.3

> **Note:** Sounds must be in the folder specified in *prefs.get('SOUNDDIR')*.

When files are dropped on the lists, strips *prefs.get('SOUNDDIR')* from them to make them relative paths, adds them to the *sound_dict*

Parameters **files** (*list*) – List of absolute paths.

class Add_Sound_Dialog

Bases: `PySide2.QtWidgets.QDialog`

Presents a dialog to define a new sound.

Makes a selection box to choose the sound type from `autopilot.get_names('sound')`, and then populates edit boxes so we can fill in its *PARAMS* .

Variables

- **type_selection** (`QtWidgets.QComboBox`) – Select from a list of available sounds
- **param_dict** (*dict*) – Parameters that are retreived by the calling *Sound_Widget*.

Methods:

`populate_params`()	When a sound type is selected, make a `QtWidgets.QLineEdit` for each *PARAM* in its definition.
`clear_params`()	Clear all current widgets
`store_param`()	When one of our edit boxes is edited, stash the parameter in *param_dict*

Attributes:

staticMetaObject

populate_params()
When a sound type is selected, make a QtWidgets.QLineEdit for each *PARAM* in its definition.

clear_params()
Clear all current widgets

store_param()
When one of our edit boxes is edited, stash the parameter in *param_dict*

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1190>

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1140>

class **Bandwidth_Test**(*pilots*)
Bases: PySide2.QtWidgets.QDialog

Test the limits of the rate of messaging from the connected Pilots.

Asks pilots to send messages at varying rates and with varying payload sizes, and with messages with/without receipts.

Measures drop rates and message latency

Variables

- **rate_list** (*list*) – List of rates (Hz) to test
- **payload_list** (*list*) – List of payload sizes (KB) to test
- **messages** (*list*) – list of messages received during test

Methods:

init_ui()	Look we're just making the stuff in the window over here alright? relax.
start()	Start the test!!!
send_test(rate, payload, n_msg, confirm)	Send a message describing the test to each of the pilots in Bandwidth_Test.test_pilots
process_test(rate, n_msg, confirm)	Process the results of the test and update the plot window.
save()	Select save file location for test results (csv) and then save them there
register_msg(value)	Receive message from pilot, stash timestamp, number and pilot
update_pbar(val)	
validate_list()	Checks that the entries in Bandwidth_Test.rates and Bandwidth_Test.payloads are well formed.

Attributes:

staticMetaObject

`init_ui()`
Look we're just making the stuff in the window over here alright? relax.

`start()`
Start the test!!!

`send_test`(*rate*, *payload*, *n_msg*, *confirm*)
Send a message describing the test to each of the pilots in `Bandwidth_Test.test_pilots`

Parameters

- **rate** (*int*) – Rate of message sending in Hz
- **payload** (*int*) – Size of message payload in bytes
- **n_msg** (*int*) – Number of messages to send
- **confirm** (*bool*) – If True, use message confirmation, if False no confirmation.

Returns:

`process_test`(*rate*, *n_msg*, *confirm*)
Process the results of the test and update the plot window.

Reads message results from `messages`, appends computed results to `results`, and starts the next test if any remain.

Parameters

- **rate** (*int*) – Rate of current test in Hz
- **n_msg** (*int*) – Number of expected messages in this test
- **confirm** (*bool*) – Whether message confirmations were enabled for this test.

`save()`
Select save file location for test results (csv) and then save them there

`register_msg`(*value*)
Receive message from pilot, stash timestamp, number and pilot

Parameters **value** (*dict*) – Value should contain

- Pilot
- Timestamp
- Message number
- Payload

`update_pbar`(*val*)

`validate_list()`
Checks that the entries in `Bandwidth_Test.rates` and `Bandwidth_Test.payloads` are well formed.

ie. that they are of the form 'integer, integer, integer'…

pops a window that warns about ill formed entry and clears line edit if badly formed

If the list validates, stored as either `Bandwidth_Test.rate_list` or `Bandwidth_Test.payload_list`

`staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d10f0>`

class `Calibrate_Water`(*pilots*)
Bases: `PySide2.QtWidgets.QDialog`

A window to calibrate the volume of water dispensed per ms.

Parameters

- **pilots** (*Terminal.pilots*) – A dictionary of pilots
- **message_fn** (*Net_Node.send()*) – The method the Terminal uses to send messages via its net node.

Methods:

init_ui()	

Attributes:

staticMetaObject	

init_ui()

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744d1050>

class Pilot_Ports(*pilot*, *n_clicks=1000*, *click_dur=30*)

Bases: PySide2.QtWidgets.QWidget

Created by *Calibrate_Water*, Each pilot's ports and buttons to control repeated release.

Parameters

- **pilot** (*str*) – name of pilot to calibrate
- **n_clicks** (*int*) – number of times to open the port during calibration
- **click_dur** (*int*) – how long to open the port (in ms)

Methods:

init_ui()	Init the layout for one pilot's ports:
update_volumes()	Store the result of a volume calibration test in volumes
start_calibration()	Send the calibration test parameters to the *Pilot*
l_progress(value)	Value should contain

Attributes:

staticMetaObject	

init_ui()

Init the layout for one pilot's ports:

- pilot name
- port buttons
- 3 times and vol dispersed

Returns

update_volumes()

Store the result of a volume calibration test in `volumes`

start_calibration()

Send the calibration test parameters to the *Pilot*

Sends a message with a `'CALIBRATE_PORT'` key, which is handled by *Pilot.l_cal_port()*

l_progress(*value*)

Value should contain

- Pilot
- Port
- Current Click (click_num)

Parameters **value** –

Returns

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744bcfa0>

class **Reassign**(*subjects*, *protocols*)

Bases: `PySide2.QtWidgets.QDialog`

A dialog that lets subjects be batch reassigned to new protocols or steps.

Parameters

- **subjects** (*dict*) –

  A dictionary that contains each subject's protocol and step, ie.:

  ```
  {'subject_id':['protocol_name', step_int], ... }
  ```

- **protocols** (*list*) – list of protocol files in the *prefs.get('PROTOCOLDIR')*. Not entirely sure why we don't just list them ourselves here.

Methods:

`init_ui`()	Initializes graphical elements.
`populate_steps`(subject)	When a protocol is selected, populate the selection box with the steps that can be chosen.
`set_protocol`()	When the protocol is changed, stash that and call *Reassign.populate_steps()* .
`set_step`()	When the step is changed, stash that.

Attributes:

`staticMetaObject`	

init_ui()

Initializes graphical elements.

Makes a row for each subject where its protocol and step can be changed.

populate_steps(*subject*)

When a protocol is selected, populate the selection box with the steps that can be chosen.

Parameters **subject** (*str*) – ID of subject whose steps are being populated

set_protocol()

When the protocol is changed, stash that and call *Reassign.populate_steps()* . Returns:

set_step()

When the step is changed, stash that.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744bcf50>

class Weights(*subject_weights*, *subjects*)

Bases: `PySide2.QtWidgets.QTableWidget`

A table for viewing and editing the most recent subject weights.

Parameters

- **subject_weights** (*list*) – a list of weights of the format returned by `Subject.get_weight(baseline=True)()`.
- **subjects** (*dict*) – the Terminal's `Terminal.subjects` dictionary of *Subject* objects.

Methods:

`init_ui`()	Initialized graphical elements.
`set_weight`(row, column)	Updates the most recent weights in `gui.Weights.subjects` objects.

Attributes:

`staticMetaObject`

init_ui()

Initialized graphical elements. Literally just filling a table.

set_weight(*row*, *column*)

Updates the most recent weights in `gui.Weights.subjects` objects.

Note: Only the daily weight measurements can be changed this way - not subject name, baseline weight, etc.

Parameters

- **row** (*int*) – row of table
- **column** (*int*) – column of table

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744bceb0>

class Plugins

Bases: `PySide2.QtWidgets.QDialog`

Dialog window that allows plugins to be viewed and installed.

Works by querying the wiki , find anything in the category `Autopilot Plugins` , clone the related repo, and reload plugins.

At the moment this widget is a proof of concept and will be made functional asap :)

Methods:

init_ui()	
list_plugins()	
download_plugin()	
select_plugin_type()	
select_plugin()	

Attributes:

staticMetaObject	

init_ui()

list_plugins()

download_plugin()

select_plugin_type()

select_plugin()

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744bce60>

class **Psychometric**(*subjects_protocols*)

Bases: PySide2.QtWidgets.QDialog

A Dialog to select subjects, steps, and variables to use in a psychometric curve plot.

See *Terminal.plot_psychometric()*

Parameters **subjects_protocols** (*dict*) – The Terminals Terminal.subjects_protocols dict

Variables **plot_params** (*list*) – A list of tuples, each consisting of (subject_id, step, variable) to be given to viz.plot_psychometric()

Methods:

init_ui()	
populate_steps(subject)	When a protocol is selected, populate the selection box with the steps that can be chosen.
populate_variables()	Fill selection boxes with step and variable names
check_all()	Toggle all checkboxes on or off

Attributes:

plot_params	Generate parameters for plot to be passed to viz.plot_psychometric()

continues on next page

Table 42 – continued from previous page

staticMetaObject

init_ui()

populate_steps(*subject*)
When a protocol is selected, populate the selection box with the steps that can be chosen.

Parameters **subject** (*str*) – ID of subject whose steps are being populated

populate_variables()
Fill selection boxes with step and variable names

check_all()
Toggle all checkboxes on or off

property plot_params
Generate parameters for plot to be passed to `viz.plot_psychometric()`

Returns (subject_name, step_name, x_var_name, n_trials_back)

Return type tuple

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744bce10>

class Stream_Video(*pilots: dict*, **args*, ***kwargs*)
Bases: `PySide2.QtWidgets.QDialog`

Dialogue to stream, display, and save video.

Parameters **pilots** (*dict*) – The `Terminal.pilot_db` with the `prefs` of each pilot (given by `Pilot.handshake()`)

Methods:

init_ui()
populate_cameras()
camera_selected()
toggle_start()
write_video()
l_frame(value)
closeEvent(self, arg__1)

Attributes:

current_pilot
current_camera

continues on next page

Table 44 – continued from previous page

staticMetaObject

init_ui()

property current_pilot: str

property current_camera: str

populate_cameras()

camera_selected()

toggle_start()

write_video()

l_frame(*value*)

closeEvent(*self*, *arg__1: PySide2.QtGui.QCloseEvent*) → None

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce744bcdc0>

pop_dialog(*message: str*, *details: str = ''*, *buttons: tuple = ('Ok',)*, *modality: str = 'nonmodal'*, *msg_type: str = 'info'*) → PySide2.QtWidgets.QMessageBox

Convenience function to pop a :class:`.QtGui.QDialog window to display a message.

Note: This function does *not* call *.exec_* on the dialog so that it can be managed by the caller.

Examples

box = pop_dialog(message='Hey what up', details='i got something to tell you', buttons = ('Ok', 'Cancel'))

ret = box.exec_() if ret == box.Ok:

print("user answered 'Ok'")

else: print("user answered 'Cancel'")

Parameters

- **message** (*str*) – message to be displayed
- **details** (*str*) – Additional detailed to be added to the displayed message
- **buttons** (*list*) – A list specifying which `QtWidgets.QMessageBox.StandardButton` s to display. Use a string matching the button name, eg. "Ok" gives `QtWidgets.QMessageBox.Ok`

  The full list of available buttons is:

  ```
  ['NoButton', 'Ok', 'Save', 'SaveAll', 'Open', 'Yes', 'YesToAll',
   'No', 'NoToAll', 'Abort', 'Retry', 'Ignore', 'Close', 'Cancel',
   'Discard', 'Help', 'Apply', 'Reset', 'RestoreDefaults',
   'FirstButton', 'LastButton', 'YesAll', 'NoAll', 'Default',
   'Escape', 'FlagMask', 'ButtonMask']
  ```

- **modality** (*str*) – Window modality to use, one of "modal", "nonmodal" (default). Modal windows block nonmodal windows don't.
- **msg_type** (*str*) – "info" (default), "question", "warning", or "error" to use `QtGui.QMessageBox.information()`, `QtGui.QMessageBox.question()`, `QtGui.QMessageBox.warning()`, or `QtGui.QMessageBox.error()`, respectively

Returns QtWidgets.QMessageBox

11.2 loggers

Data:

_LOGGERS	List of instantiated loggers, used in `init_logger()` to return existing loggers without modification

Functions:

`init_logger`([instance, module_name, . . .])	Initialize a logger

_LOGGERS = []
List of instantiated loggers, used in `init_logger()` to return existing loggers without modification

init_logger(*instance=None*, *module_name=None*, *class_name=None*, *object_name=None*) → logging.Logger
Initialize a logger

Loggers are created such that. . .

- There is one logger per module (eg. all gpio objects will log to hardware.gpio)
- If the passed object has a `name` attribute, that name will be prefixed to its log messages in the file
- The loglevel for the file handler and the stdout is determined by `prefs.get('LOGLEVEL')`, and if none is provided `WARNING` is used by default
- logs are rotated according to `prefs.get('LOGSIZE')` (in bytes) and `prefs.get('LOGNUM')` (number of backups of `prefs.get('LOGSIZE')` to cycle through)

Logs are stored in `prefs.get('LOGDIR')`, and are formatted like:

```
"%(asctime)s - %(name)s - %(levelname)s : %(message)s"
```

Loggers can be initialized either by passing an object to the first `instance` argument, or by specifying any of `module_name` , `class_name` , or `object_name` (at least one must be specified) which are combined with periods like `module.class_name.object_name`

Parameters

- **instance** – The object that we are creating a logger for! if None, at least one of `module, class_name, or object_name` must be passed
- **module_name** (*None, str*) – If no `instance` passed, the module name to create a logger for
- **class_name** (*None, str*) – If no `instance` passed, the class name to create a logger for
- **object_name** (*None, str*) – If no `instance` passed, the object name/id to create a logger for

Returns `logging.logger`

11.3 pilot

Classes:

Pilot([splash])	Drives the Raspberry Pi

class Pilot(*splash=True*)

Bases: object

Drives the Raspberry Pi

Coordinates the hardware and networking objects to run tasks.

Typically used with a connection to a *Terminal* object to coordinate multiple subjects and tasks, but a high priority for future releases is to do the (trivial amount of) work to make this class optionally standalone.

Called as a module with the -f flag to give the location of a prefs file, eg:

```
python pilot.py -f prefs_file.json
```

if the -f flag is not passed, looks in the default location for prefs (ie. */usr/autopilot/prefs.json*)

Needs the following prefs (typically established by `setup.setup_pilot`):

- **NAME** - The name used by networking objects to address this Pilot
- **BASEDIR** - The base directory for autopilot files (/usr/autopilot)
- **PUSHPORT** - Router port used by the Terminal we connect to.
- **TERMINALIP** - IP Address of our upstream Terminal.
- **MSGPORT** - Port used by our own networking object
- **HARDWARE** - Any hardware and its mapping to GPIO pins. No pins are required to be set, instead each task defines which pins it needs. Currently the default configuration asks for
  - POKES - `hardware.Beambreak`
  - LEDS - `hardware.LED_RGB`
  - PORTS - `hardware.Solenoid`
- **AUDIOSERVER** - Which type, if any, audio server to use (*'jack'*, *'pyo'*, or *'none'*)
- **NCHANNELS** - Number of audio channels
- **FS** - Sampling rate of audio output
- **JACKDSTRING** - string used to start the jackd server, see the jack manpages eg:

```
jackd -P75 -p16 -t2000 -dalsa -dhw:sndrpihifiberry -P -rfs -n3 -s &
```

- **PIGPIOMASK** - Binary mask of pins for pigpio to control, see the pigpio docs , eg:

```
1111110000111111111111110000
```

- **PULLUPS** - Pin (board) numbers to pull up on boot
- **PULLDOWNS** - Pin (board) numbers to pull down on boot.

Variables

- **name** (*str*) – The name used to identify ourselves in *networking*
- **task** (tasks.Task) – The currently instantiated task
- **running** (threading.Event) – Flag used to control task running state
- **stage_block** (threading.Event) – Flag given to a task to signal when task stages finish
- **file_block** (threading.Event) – Flag used to wait for file transfers
- **state** (*str*) – 'RUNNING', 'STOPPING', 'IDLE' - signals what this pilot is up to
- **pulls** (*list*) – list of Pull objects to keep pins pulled up or down
- **server** – Either a *pyo_server()* or *JackClient* , sound server.
- **node** (networking.Net_Node) – Our Net_Node we use to communicate with our main networking object
- **networking** (networking.Pilot_Station) – Our networking object to communicate with the outside world
- **ip** (*str*) – Our IPv4 address
- **listens** (*dict*) – Dictionary mapping message keys to methods used to process them.
- **logger** (logging.Logger) – Used to log messages and network events.

Attributes:

server	
logger	
running	
stage_block	
file_block	
quitting	mp.Event to signal when process is quitting
networking	
node	

Methods:

get_ip()	Get our IP
handshake()	Send the terminal our name and IP to signal that we are alive
update_state()	Send our current state to the Terminal, our Station object will cache this and will handle any future requests.
l_start(value)	Start running a task.
l_stop(value)	Stop the task.
l_param(value)	Change a task parameter mid-run
l_cal_port(value)	Initiate the *calibrate_port()* routine.

continues on next page

Table 49 – continued from previous page

`calibrate_port`(port_name, n_clicks, ...)	Run port calibration routine
`l_cal_result`(value)	Save the results of a port calibration
`l_bandwidth`(value)	Send messages with a poissonian process according to the settings in value
`l_stream_video`(value)	Start or stop video streaming
`calibration_curve`([path, calibration])	# compute curve to compute duration from desired volume
`init_pigpio`()	
`init_audio`()	Initialize an audio server depending on the value of *prefs.get('AUDIOSERVER')*
`blank_LEDs`()	If any 'LEDS' are defined in *prefs.get('HARDWARE')* , instantiate them, set their color to [0,0,0], and then release them.
`open_file`()	Setup a table to store data locally.
`run_task`(task_class, task_params)	Called in a new thread, run the task.

server = None

logger = None

running = None

stage_block = None

file_block = None

quitting = None
mp.Event to signal when process is quitting

networking = None

node = None

get_ip()
Get our IP

handshake()
Send the terminal our name and IP to signal that we are alive

update_state()
Send our current state to the Terminal, our Station object will cache this and will handle any future requests.

l_start(*value*)
Start running a task.

Get the task object by using *value['task_type']* to select from `autopilot.get_task()` , then feed the rest of *value* as kwargs into the task object.

Calls `autopilot.run_task()` in a new thread

Parameters **value** (*dict*) – A dictionary of task parameters

l_stop(*value*)
Stop the task.

Clear the running event, set the stage block.

Todo: Do a coherence check between our local file and the Terminal's data.

Parameters value – ignored

l_param(*value*)
Change a task parameter mid-run

> **Warning:** Not Implemented

Parameters value

l_cal_port(*value*)
Initiate the `calibrate_port()` routine.

Parameters value (*dict*) – Dictionary of values defining the port calibration to be run, including - `port` - which port to calibrate - `n_clicks` - how many openings should be performed - `open_dur` - how long the valve should be open - `iti` - 'inter-trial interval`, or how long should we wait between valve openings.

calibrate_port(*port_name*, *n_clicks*, *open_dur*, *iti*)
Run port calibration routine

Open a `hardware.gpio.Solenoid` repeatedly, measure volume of water dispersed, compute lookup table mapping valve open times to volume.

Continuously sends progress of test with `CAL_PROGRESS` messages

Parameters

- **port_name** (*str*) – Port name as specified in `prefs`
- **n_clicks** (*int*) – number of times the valve should be opened
- **open_dur** (*int, float*) – how long the valve should be opened for in ms
- **iti** (*int, float*) – how long we should `sleep()` between openings

l_cal_result(*value*)
Save the results of a port calibration

l_bandwidth(*value*)
Send messages with a poissonian process according to the settings in value

l_stream_video(*value*)
Start or stop video streaming

Parameters value (*dict*) –

a dictionary of the form:

```
{
    'starting': bool, # whether we're starting (True) or stopping
    'camera': str, # the camera to start/stop, of form 'group.camera_
↪id'
    'stream_to': node id that the camera should send to
}
```

calibration_curve(*path=None*, *calibration=None*)
compute curve to compute duration from desired volume

Parameters

- **calibration**

- **path** – If present, use calibration file specified, otherwise use default.

init_pigpio()

init_audio()

Initialize an audio server depending on the value of *prefs.get('AUDIOSERVER')*

- 'pyo' = `pyoserver.pyo_server()`
- 'jack' = `jackclient.JackClient`

blank_LEDs()

If any 'LEDS' are defined in *prefs.get('HARDWARE')* , instantiate them, set their color to [0,0,0], and then release them.

open_file()

Setup a table to store data locally.

Opens *prefs.get('DATADIR')/local.h5*, creates a group for the current subject, a new table for the current day.

Todo: This needs to be unified with a general file constructor abstracted from `Subject` so it doesn't reimplement file creation!!

Returns (`tables.File`, `tables.Table`, `tables.tableextension.Row`): The file, table, and row for the local data table

run_task(*task_class*, *task_params*)

Called in a new thread, run the task.

Opens a file with `open_file()` , then continually calls *task.stages.next* to process stages.

Sends data back to the terminal between every stage.

Waits for the task to clear *stage_block* between stages.

11.4 plots

Classes to plot data in the GUI.

Todo: Add all possible plot objects and options in list.

Note: Plot objects need to be added to `PLOT_LIST` in order to be reachable.

Functions:

`gui_event`(fn)	Wrapper/decorator around an event that posts GUI events back to the main thread that our window is running in.

Classes:

Plot_Widget()	Main plot widget that holds plots for all pilots
Plot(pilot[, x_width, parent])	Widget that hosts a pyqtgraph.PlotWidget and manages graphical objects for one pilot depending on the task.
Point([color, size])	A simple point.
Line([color, size])	A simple line
Segment(**kwargs)	A line segment that draws from 0.5 to some endpoint.
Roll_Mean([winsize])	Shaded area underneath a rolling average.
Shaded(**kwargs)	Shaded area for a continuous plot
Timer()	A simple timer that counts.
Video(videos[, fps])	Display Video data as it is collected.
HLine()	A Horizontal line.
ImageItem_TimedUpdate(*args, **kwargs)	Reclass of pyqtgraph.ImageItem to update with a fixed fps.

Data:

PLOT_LIST	A dictionary connecting plot keys to objects.

gui_event(*fn*)

Wrapper/decorator around an event that posts GUI events back to the main thread that our window is running in.

Parameters fn (*callable*) – a function that does something to the GUI

class Plot_Widget

Bases: PySide2.QtWidgets.QWidget

Main plot widget that holds plots for all pilots

Essentially just a container to give plots a layout and handle any logic that should apply to all plots.

Variables

- **logger** (*logging.Logger*) – The 'main' logger
- **plots** (*dict*) – mapping from pilot name to *Plot*

Methods:

init_plots(pilot_list)	For each pilot, instantiate a *Plot* and add to layout.

Attributes:

staticMetaObject

init_plots(*pilot_list*)

For each pilot, instantiate a *Plot* and add to layout.

Parameters pilot_list (*list*) – the keys from *Terminal.pilots*

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce75968640>

class Plot(*pilot*, *x_width=50*, *parent=None*)

Bases: PySide2.QtWidgets.QWidget

Widget that hosts a pyqtgraph.PlotWidget and manages graphical objects for one pilot depending on the task.

listens

Key	Method	Description
'START'	`l_start()`	starting a new task
'DATA'	`l_data()`	getting a new datapoint
'STOP'	`l_stop()`	stop the task
'PARAM'	`l_param()`	change some parameter

Plot Parameters

The plot is built from the `PLOT={data:plot_element}` mappings described in the `Task` class. Additional parameters can be specified in the `PLOT` dictionary. Currently:

- **continuous** (bool): whether the data should be plotted against the trial number (False or NA) or against time (True)
- **chance_bar** (bool): Whether to draw a red horizontal line at chance level (default: 0.5)
- **chance_level** (float): The position in the y-axis at which the `chance_bar` should be drawn
- **roll_window** (int): The number of trials `Roll_Mean` take the average over.

Variables

- **pilot** (*str*) – The name of our pilot, used to set the identity of our socket, specifically:

```
'P_{pilot}'
```

- **infobox** (`QtWidgets.QFormLayout`) – Box to plot basic task information like trial number, etc.
- **info** (*dict*) – Widgets in infobox:
  - 'N Trials': `QtWidgets.QLabel`,
  - 'Runtime' : `Timer`,
  - 'Session' : `QtWidgets.QLabel`,
  - 'Protocol': `QtWidgets.QLabel`,
  - 'Step' : `QtWidgets.QLabel`
- **plot** (`pyqtgraph.PlotWidget`) – The widget where we draw our plots
- **plot_params** (*dict*) – A dictionary of plot parameters we receive from the Task class
- **data** (*dict*) – A dictionary of the data we've received
- **plots** (*dict*) – The collection of plots we instantiate based on *plot_params*
- **node** (`Net_Node`) – Our local net node where we listen for data.
- **state** (*str*) – state of the pilot, used to keep plot synchronized.

Parameters

- **pilot** (*str*) – The name of our pilot
- **x_width** (*int*) – How many trials in the past should we plot?

Methods:

`init_plots()`	Make pre-task GUI objects and set basic visual parameters of *self.plot*
`l_start(value)`	Starting a task, initialize task-specific plot objects described in the `Task.PLOT` attribute.
`l_data(value)`	Receive some data, if we were told to plot it, stash the data and update the assigned plot.
`l_stop(value)`	Clean up the plot objects.
`l_param(value)`	**Warning:** Not implemented
`l_state(value)`	Pilot letting us know its state has changed.

Attributes:

`staticMetaObject`

`init_plots()`
Make pre-task GUI objects and set basic visual parameters of *self.plot*

`l_start`(*value*)
Starting a task, initialize task-specific plot objects described in the `Task.PLOT` attribute.

Matches the data field name (keys of `Task.PLOT`) to the plot object that represents it, eg, to make the standard nafc plot:

```
{'target'   : 'point',
 'response' : 'segment',
 'correct'  : 'rollmean'}
```

Parameters **value** (*dict*) – The same parameter dictionary sent by `Terminal.toggle_start()`, including

- current_trial
- step
- session
- step_name
- task_type

`l_data`(*value*)
Receive some data, if we were told to plot it, stash the data and update the assigned plot.

Parameters **value** (*dict*) – Value field of a data message sent during a task.

`l_stop`(*value*)
Clean up the plot objects.

Parameters **value** (*dict*) – if "graduation" is a key, don't stop the timer.

`l_param`(*value*)

> **Warning:** Not implemented

Parameters **value**

`l_state`(*value*)

Pilot letting us know its state has changed. Mostly for the case where we think we're running but the pi doesn't.

Parameters **value** (`Pilot.state`) – the state of our pilot

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c0a0>

class Point(*color=(0, 0, 0)*, *size=5*, ***kwargs*)

Bases: `pyqtgraph.graphicsItems.PlotDataItem.PlotDataItem`

A simple point.

Variables

- **brush** (`QtWidgets.QBrush`) –
- **pen** (`QtWidgets.QPen`) –

Parameters

- **color** (*tuple*) – RGB color of points
- **size** (*int*) – width in px.

Methods:

`update`(data)	**Parameters** **data** (`numpy.ndarray`) – an x_width x 2 array where

Attributes:

`staticMetaObject`	

`update`(*data*)

Parameters **data** (`numpy.ndarray`) – an x_width x 2 array where column 0 is trial number and column 1 is the value, where value can be "L", "C", "R" or a float.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c0f0>

class Line(*color=(0, 0, 0)*, *size=1*, ***kwargs*)

Bases: `pyqtgraph.graphicsItems.PlotDataItem.PlotDataItem`

A simple line

There are many different ways to create a PlotDataItem:

Data initialization arguments: (x,y data only)

PlotDataItem(xValues, yValues)	x and y values may be any sequence (including ndarray) of real numbers
PlotDataItem(yValues)	y values only – x will be automatically set to range(len(y))
PlotDataItem(x=xValues, y=yValues)	x and y given by keyword arguments
PlotDataItem(ndarray(Nx2))	numpy array with shape (N, 2) where `x=data[:,0]` and `y=data[:,1]`

Data initialization arguments: (x,y data AND may include spot style)

PlotDataItem(recarray)	numpy array with `dtype=[('x', float), ('y', float), ...]`
PlotDataItem(list-of-dicts)	`[{'x': x, 'y': y, ...}, ...]`
PlotDataItem(dict-of-lists)	`{'x': [...], 'y': [...], ...}`
PlotDataItem(MetaArray)	1D array of Y values with X sepecified as axis values OR 2D array with a column 'y' and extra columns as needed.

Line style keyword arguments:

connect	Specifies how / whether vertexes should be connected. See `arrayToQPath()`
pen	Pen to use for drawing line between points. Default is solid grey, 1px width. Use None to disable line drawing. May be any single argument accepted by `mkPen()`
shadowPen	Pen for secondary line to draw behind the primary line. disabled by default. May be any single argument accepted by `mkPen()`
fillLevel	Fill the area between the curve and fillLevel
fillOutline	(bool) If True, an outline surrounding the *fillLevel* area is drawn.
fillBrush	Fill to use when fillLevel is specified. May be any single argument accepted by `mkBrush()`
stepMode	If True, two orthogonal lines are drawn for each sample as steps. This is commonly used when drawing histograms. Note that in this case, `len(x) == len(y) + 1` (added in version 0.9.9)

Point style keyword arguments: (see `ScatterPlotItem.setData()` for more information)

symbol	Symbol to use for drawing points OR list of symbols, one per point. Default is no symbol. Options are o, s, t, d, +, or any QPainterPath
symbol-Pen	Outline pen for drawing points OR list of pens, one per point. May be any single argument accepted by `mkPen()`
symbol-Brush	Brush for filling points OR list of brushes, one per point. May be any single argument accepted by `mkBrush()`
symbol-Size	Diameter of symbols OR list of diameters.
pxMode	(bool) If True, then symbolSize is specified in pixels. If False, then symbolSize is specified in data coordinates.

Optimization keyword arguments:

an-tialias	(bool) By default, antialiasing is disabled to improve performance. Note that in some cases (in particluar, when pxMode=True), points will be rendered antialiased even if this is set to False.
deci-mate	deprecated.
down-sam-ple	(int) Reduce the number of samples displayed by this value
down-sam-pleMethod	'subsample': Downsample by taking the first of N samples. This method is fastest and least accurate. 'mean': Downsample by taking the mean of N samples. 'peak': Downsample by drawing a saw wave that follows the min and max of the original data. This method produces the best visual representation of the data but is slower.
autoDown-sam-ple	(bool) If True, resample the data before plotting to avoid plotting multiple line segments per pixel. This can improve performance when viewing very high-density data, but increases the initial overhead and memory usage.
clip-ToView	(bool) If True, only plot data that is visible within the X range of the containing ViewBox. This can improve performance when plotting very large data sets where only a fraction of the data is visible at any time.
iden-tical	*deprecated*

Meta-info keyword arguments:

name	name of dataset. This would appear in a legend

Methods:

`update`() -> None)

Attributes:

`staticMetaObject`

update(*self*, *rect: PySide2.QtCore.QRectF = Default(QRectF)*) → None
update(*self*, *x: float*, *y: float*, *width: float*, *height: float*) → None

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c140>

class Segment(***kwargs*)

Bases: `pyqtgraph.graphicsItems.PlotDataItem.PlotDataItem`

A line segment that draws from 0.5 to some endpoint.

There are many different ways to create a PlotDataItem:

Data initialization arguments: (x,y data only)

PlotDataItem(xValues, yValues)	x and y values may be any sequence (including ndarray) of real numbers
PlotDataItem(yValues)	y values only – x will be automatically set to range(len(y))
PlotDataItem(x=xValues, y=yValues)	x and y given by keyword arguments
PlotDataItem(ndarray(Nx2))	numpy array with shape (N, 2) where `x=data[:,0]` and `y=data[:,1]`

Data initialization arguments: (x,y data AND may include spot style)

PlotDataItem(recarray)	numpy array with `dtype=[('x', float), ('y', float), ...]`
PlotDataItem(list-of-dicts)	`[{'x': x, 'y': y, ...}, ...]`
PlotDataItem(dict-of-lists)	`{'x': [...], 'y': [...], ...}`
PlotDataItem(MetaArray)	1D array of Y values with X sepecified as axis values OR 2D array with a column 'y' and extra columns as needed.

Line style keyword arguments:

connect	Specifies how / whether vertexes should be connected. See `arrayToQPath()`
pen	Pen to use for drawing line between points. Default is solid grey, 1px width. Use None to disable line drawing. May be any single argument accepted by `mkPen()`
shadowPen	Pen for secondary line to draw behind the primary line. disabled by default. May be any single argument accepted by `mkPen()`
fillLevel	Fill the area between the curve and fillLevel
fillOutline	(bool) If True, an outline surrounding the *fillLevel* area is drawn.
fillBrush	Fill to use when fillLevel is specified. May be any single argument accepted by `mkBrush()`
stepMode	If True, two orthogonal lines are drawn for each sample as steps. This is commonly used when drawing histograms. Note that in this case, `len(x) == len(y) + 1` (added in version 0.9.9)

Point style keyword arguments: (see `ScatterPlotItem.setData()` for more information)

symbol	Symbol to use for drawing points OR list of symbols, one per point. Default is no symbol. Options are o, s, t, d, +, or any QPainterPath
symbol-Pen	Outline pen for drawing points OR list of pens, one per point. May be any single argument accepted by `mkPen()`
symbol-Brush	Brush for filling points OR list of brushes, one per point. May be any single argument accepted by `mkBrush()`
symbol-Size	Diameter of symbols OR list of diameters.
pxMode	(bool) If True, then symbolSize is specified in pixels. If False, then symbolSize is specified in data coordinates.

Optimization keyword arguments:

an-tialias	(bool) By default, antialiasing is disabled to improve performance. Note that in some cases (in particluar, when pxMode=True), points will be rendered antialiased even if this is set to False.
deci-mate	deprecated.
down-sam-ple	(int) Reduce the number of samples displayed by this value
down-sam-pleMethod	'subsample': Downsample by taking the first of N samples. This method is fastest and least accurate. 'mean': Downsample by taking the mean of N samples. 'peak': Downsample by drawing a saw wave that follows the min and max of the original data. This method produces the best visual representation of the data but is slower.
autoDown-sam-ple	(bool) If True, resample the data before plotting to avoid plotting multiple line segments per pixel. This can improve performance when viewing very high-density data, but increases the initial overhead and memory usage.
clip-ToView	(bool) If True, only plot data that is visible within the X range of the containing ViewBox. This can improve performance when plotting very large data sets where only a fraction of the data is visible at any time.
iden-tical	*deprecated*

Meta-info keyword arguments:

name	name of dataset. This would appear in a legend

Methods:

`update`(data)	data is doubled and then every other value is set to 0.5, then `setData()` is used with *connect='pairs'* to make line segments.

Attributes:

`staticMetaObject`

update(*data*)

data is doubled and then every other value is set to 0.5, then `setData()` is used with *connect='pairs'* to make line segments.

Parameters **data** (`numpy.ndarray`) – an x_width x 2 array where column 0 is trial number and column 1 is the value, where value can be "L", "C", "R" or a float.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c190>

class Roll_Mean(*winsize=10*, ***kwargs*)

Bases: `pyqtgraph.graphicsItems.PlotDataItem.PlotDataItem`

Shaded area underneath a rolling average.

Typically used as a rolling mean of corrects, so area above and below 0.5 is drawn.

Parameters **winsize** (*int*) – number of trials in the past to take a rolling mean of

Methods:

`update`(data)	**Parameters** **data** (`numpy.ndarray`) – an x_width x 2 array where

Attributes:

`staticMetaObject`

update(*data*)

Parameters **data** (`numpy.ndarray`) – an x_width x 2 array where column 0 is trial number and column 1 is the value.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c1e0>

class Shaded(***kwargs*)

Bases: `pyqtgraph.graphicsItems.PlotDataItem.PlotDataItem`

Shaded area for a continuous plot

There are many different ways to create a PlotDataItem:

Data initialization arguments: (x,y data only)

PlotDataItem(xValues, yValues)	x and y values may be any sequence (including ndarray) of real numbers
PlotDataItem(yValues)	y values only – x will be automatically set to range(len(y))
PlotDataItem(x=xValues, y=yValues)	x and y given by keyword arguments
PlotDataItem(ndarray(Nx2))	numpy array with shape (N, 2) where `x=data[:,0]` and `y=data[:,1]`

Data initialization arguments: (x,y data AND may include spot style)

PlotDataItem(recarray)	numpy array with `dtype=[('x', float), ('y', float), ...]`
PlotDataItem(list-of-dicts)	`[{'x': x, 'y': y, ...}, ...]`
PlotDataItem(dict-of-lists)	`{'x': [...], 'y': [...], ...}`
PlotDataItem(MetaArray)	1D array of Y values with X sepecified as axis values OR 2D array with a column 'y' and extra columns as needed.

Line style keyword arguments:

connect	Specifies how / whether vertexes should be connected. See `arrayToQPath()`
pen	Pen to use for drawing line between points. Default is solid grey, 1px width. Use None to disable line drawing. May be any single argument accepted by `mkPen()`
shadowPen	Pen for secondary line to draw behind the primary line. disabled by default. May be any single argument accepted by `mkPen()`
fillLevel	Fill the area between the curve and fillLevel
fillOutline	(bool) If True, an outline surrounding the *fillLevel* area is drawn.
fillBrush	Fill to use when fillLevel is specified. May be any single argument accepted by `mkBrush()`
stepMode	If True, two orthogonal lines are drawn for each sample as steps. This is commonly used when drawing histograms. Note that in this case, `len(x) == len(y) + 1` (added in version 0.9.9)

Point style keyword arguments: (see `ScatterPlotItem.setData()` for more information)

symbol	Symbol to use for drawing points OR list of symbols, one per point. Default is no symbol. Options are o, s, t, d, +, or any QPainterPath
symbolPen	Outline pen for drawing points OR list of pens, one per point. May be any single argument accepted by `mkPen()`
symbolBrush	Brush for filling points OR list of brushes, one per point. May be any single argument accepted by `mkBrush()`
symbolSize	Diameter of symbols OR list of diameters.
pxMode	(bool) If True, then symbolSize is specified in pixels. If False, then symbolSize is specified in data coordinates.

Optimization keyword arguments:

antialias	(bool) By default, antialiasing is disabled to improve performance. Note that in some cases (in particluar, when pxMode=True), points will be rendered antialiased even if this is set to False.
decimate	deprecated.
downsample	(int) Reduce the number of samples displayed by this value
downsampleMethod	'subsample': Downsample by taking the first of N samples. This method is fastest and least accurate. 'mean': Downsample by taking the mean of N samples. 'peak': Downsample by drawing a saw wave that follows the min and max of the original data. This method produces the best visual representation of the data but is slower.
autoDownsample	(bool) If True, resample the data before plotting to avoid plotting multiple line segments per pixel. This can improve performance when viewing very high-density data, but increases the initial overhead and memory usage.
clipToView	(bool) If True, only plot data that is visible within the X range of the containing ViewBox. This can improve performance when plotting very large data sets where only a fraction of the data is visible at any time.
identical	*deprecated*

Meta-info keyword arguments:

name	name of dataset. This would appear in a legend

Methods:

`update`(data)	**Parameters data** (`numpy.ndarray`) – an x_width x 2 array where

Attributes:

`staticMetaObject`

`update`(*data*)

Parameters data (`numpy.ndarray`) – an x_width x 2 array where column 0 is time and column 1 is the value.

`staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c230>`

`class Timer`

Bases: `PySide2.QtWidgets.QLabel`

A simple timer that counts… time…

Uses a `QtCore.QTimer` connected to *`Timer.update_time()`* .

Methods:

start_timer([update_interval])	**Parameters** **update_interval** (*float*) – How often (in ms) the timer should be updated.
stop_timer()	you can read the sign ya punk
update_time()	Called every (update_interval) milliseconds to set the text of the timer.

Attributes:

staticMetaObject

start_timer(*update_interval=1000*)

Parameters **update_interval** (*float*) – How often (in ms) the timer should be updated.

stop_timer()
you can read the sign ya punk

update_time()
Called every (update_interval) milliseconds to set the text of the timer.

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c280>

class Video(*videos*, *fps=None*)
Bases: `PySide2.QtWidgets.QWidget`

Display Video data as it is collected.

Uses the *ImageItem_TimedUpdate* class to do timed frame updates.

Parameters

- **videos** (*list, tuple*) – Names of video streams that will be displayed
- **fps** (*int*) – if None, draw according to `prefs.get('DRAWFPS')`. Otherwise frequency of widget update

Variables

- **videos** (*list, tuple*) – Names of video streams that will be displayed
- **fps** (*int*) – if None, draw according to `prefs.get('DRAWFPS')`. Otherwise frequency of widget update
- **ifps** (*int*) – 1/fps, duration of frame in s
- **qs** (*dict*) – Dictionary of :class:`~queue.Queue`s in which frames will be dumped
- **quitting** (`threading.Event`) – Signal to quit drawing
- **update_thread** (`threading.Thread`) – Thread with target=:meth:~.*Video._update_frame*
- **layout** (`PySide2.QtWidgets.QGridLayout`) – Widget layout
- **vid_widgets** (*dict*) – dict containing widgets for each of the individual video streams.

Methods:

`init_gui()`	
`_update_frame()`	Pulls frames from `Video.qs` and feeds them to the video widgets.
`update_frame(video, data)`	Put a frame for a video stream into its queue.
`release()`	

Attributes:

`staticMetaObject`	

init_gui()

_update_frame()

Pulls frames from `Video.qs` and feeds them to the video widgets.

Internal method, run in thread.

update_frame(*video*, *data*)

Put a frame for a video stream into its queue.

If there is a waiting frame, pull it from the queue first – it's old now.

Parameters

- **video** (*str*) – name of video stream
- **data** (`numpy.ndarray`) – video frame

release()

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c2d0>

class HLine

Bases: `PySide2.QtWidgets.QFrame`

A Horizontal line.

Attributes:

`staticMetaObject`	

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c320>

class ImageItem_TimedUpdate(**args*, ***kwargs*)

Bases: `pyqtgraph.graphicsItems.ImageItem.ImageItem`

Reclass of `pyqtgraph.ImageItem` to update with a fixed fps.

Rather than calling `update()` every time a frame is updated, call it according to the timer.

fps is set according to `prefs.get('DRAWFPS')`, if not available, draw at 10fps

Variables **timer** (`QTimer`) – Timer held in `globals()` that synchronizes frame updates across image items

See `setImage` for all allowed initialization arguments.

Methods:

`setImage`([image, autoLevels])	Update the image displayed by this item.
`update_img`()	Call `update()`

Attributes:

`staticMetaObject`

`setImage`(*image=None*, *autoLevels=None*, ***kargs*)

Update the image displayed by this item. For more information on how the image is processed before displaying, see `makeARGB`

Arguments:	
image	(numpy array) Specifies the image data. May be 2D (width, height) or 3D (width, height, RGBa). The array dtype must be integer or floating point of any bit depth. For 3D arrays, the third dimension must be of length 3 (RGB) or 4 (RGBA). See *notes* below.
autoLevels	(bool) If True, this forces the image to automatically select levels based on the maximum and minimum values in the data. By default, this argument is true unless the levels argument is given.
lut	(numpy array) The color lookup table to use when displaying the image. See `setLookupTable`.
levels	(min, max) The minimum and maximum values to use when rescaling the image data. By default, this will be set to the minimum and maximum values in the image. If the image array has dtype uint8, no rescaling is necessary.
opacity	(float 0.0-1.0)
compositionMode	See `setCompositionMode`
border	Sets the pen used when drawing the image border. Default is None.
autoDownsample	(bool) If True, the image is automatically downsampled to match the screen resolution. This improves performance for large images and reduces aliasing. If autoDownsample is not specified, then ImageItem will choose whether to downsample the image based on its size.

Notes:

For backward compatibility, image data is assumed to be in column-major order (column, row). However, most image data is stored in row-major order (row, column) and will need to be transposed before calling setImage():

```
imageitem.setImage(imagedata.T)
```

This requirement can be changed by calling `image.setOpts(axisOrder='row-major')` or by changing the `imageAxisOrder` global configuration option.

`staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce7468c3c0>`

update_img()
Call update()

PLOT_LIST = { 'line': <class 'autopilot.core.plots.Line'>, 'point': <class 'autopilot.core.plots.Point'>, 'rollmean': <class 'autopilot.core.plots.Roll_Mean'>, 'segment': <class 'autopilot.core.plots.Segment'>, 'shaded': <class 'autopilot.core.plots.Shaded'>}
A dictionary connecting plot keys to objects.

Todo: Just reference the plot objects.

11.5 styles

Qt Stylesheets for Autopilot GUI widgets

See: https://doc.qt.io/qt-5/stylesheet-reference.html#

11.6 subject

Classes for managing data and protocol access and storage.

Currently named subject, but will likely be refactored to include other data models should the need arise.

Classes:

Subject(name, dir, file, new, biography)	Class for managing one subject's data and protocol.

class Subject(*name: Optional[str] = None, dir: Optional[str] = None, file: Optional[str] = None, new: bool = False, biography: Optional[dict] = None*)
Bases: object

Class for managing one subject's data and protocol.

Creates a tables hdf5 file in *prefs.get('DATADIR')* with the general structure:

```
/ root
|--- current (tables.filenode) storing the current task as serialized JSON
|--- data (group)
|    |--- task_name  (group)
|         |--- S##_step_name
|         |    |--- trial_data
|         |    |--- continuous_data
|         |--- ...
|--- history (group)
|    |--- hashes - history of git commit hashes
|    |--- history - history of changes: protocols assigned, params changed, etc.
|    |--- weights - history of pre and post-task weights
|    |--- past_protocols (group) - stash past protocol params on reassign
|         |--- date_protocol_name - tables.filenode of a previous protocol's params.
|         |--- ...
|--- info - group with biographical information as attributes
```

Variables

- **lock** (threading.Lock) – manages access to the hdf5 file
- **name** (*str*) – Subject ID
- **file** (*str*) – Path to hdf5 file - usually *{prefs.get('DATADIR')}/{self.name}.h5*
- **current** (*dict*) – current task parameters. loaded from the 'current' filenode of the h5 file
- **step** (*int*) – current step
- **protocol_name** (*str*) – name of currently assigned protocol
- **current_trial** (*int*) – number of current trial
- **running** (*bool*) – Flag that signals whether the subject is currently running a task or not.
- **data_queue** (queue.Queue) – Queue to dump data while running task
- **thread** (threading.Thread) – thread used to keep file open while running task
- **did_graduate** (threading.Event) – Event used to signal if the subject has graduated the current step
- **STRUCTURE** (*list*) – list of tuples with order:
  - full path, eg. '/history/weights'
  - relative path, eg. '/history'
  - name, eg. 'weights'
  - type, eg. *Subject.Weight_Table* or 'group'
- **locations** (*node*) – tables.IsDescriptor for tables.

Parameters

- **name** (*str*) – subject ID
- **dir** (*str*) – path where the .h5 file is located, if *None*, *prefs.get('DATADIR')* is used
- **file** (*str*) – load a subject from a filename. if *None*, ignored.
- **new** (*bool*) – if True, a new file is made (a new file is made if one does not exist anyway)
- **biography** (*dict*) – If making a new subject file, a dictionary with biographical data can be passed

Methods:

open_hdf([mode])	Opens the hdf5 file.
close_hdf(h5f)	Flushes & closes the open hdf file.
new_subject_file(biography)	Create a new subject file and make the general filestructure.
ensure_structure()	Ensure that our h5f has the appropriate baseline structure as defined in *self.STRUCTURE*
update_biography(params)	Change or make a new biographical attribute, stored as attributes of the *info* group.
update_history(type, name, value[, step])	Update the history table when changes are made to the subject's protocol.
assign_protocol(protocol[, step_n])	Assign a protocol to the subject.

continues on next page

Table 75 – continued from previous page

flush_current()	Flushes the 'current' attribute in the subject object to the current filenode in the .h5
stash_current()	Save the current protocol in the history group and delete the node
prepare_run()	Prepares the Subject object to receive data while running the task.
data_thread(queue)	Thread that keeps hdf file open and receives data while task is running.
save_data(data)	Alternate and equivalent method of putting data in the queue as *Subject.data_queue.put(data)*
stop_run()	puts 'END' in the data_queue, which causes `data_thread()` to end.
to_csv(path[, task, step])	Export trial data to .csv
get_trial_data([step, what])	Get trial data from the current task.
apply_along([along, step])	
get_step_history([use_history])	Gets a dataframe of step numbers, timestamps, and step names as a coarse view of training status.
get_timestamp([simple])	Makes a timestamp.
get_weight([which, include_baseline])	Gets start and stop weights.
set_weight(date, col_name, new_value)	Updates an existing weight in the weight table.
update_weights([start, stop])	Store either a starting or stopping mass.
graduate()	Increase the current step by one, unless it is the last step.

Classes:

History_Table()	Class to describe parameter and protocol change history
Weight_Table()	Class to describe table for weight history
Hash_Table()	Class to describe table for hash history

open_hdf(*mode='r+'*)

Opens the hdf5 file.

This should be called at the start of every method that access the h5 file and `close_hdf()` should be called at the end. Otherwise the file will close and we risk file corruption.

See the pytables docs here and here

Parameters **mode** (*str*) – a file access mode, can be:

- 'r': Read-only - no data can be modified.
- 'w': Write - a new file is created (an existing file with the same name would be deleted).
- 'a' Append - an existing file is opened for reading and writing, and if the file does not exist it is created.
- 'r+' (default) - Similar to 'a', but file must already exist.

Returns Opened hdf file.

Return type `tables.File`

close_hdf(*h5f*)

Flushes & closes the open hdf file. Must be called whenever `open_hdf()` is used.

Parameters **h5f** (`tables.File`) – the hdf file opened by `open_hdf()`

`new_subject_file`(*biography*)

Create a new subject file and make the general filestructure.

If a file already exists, open it in append mode, otherwise create it.

Parameters **biography** (*dict*) – Biographical details like DOB, mass, etc. Typically created by `Biography_Tab`.

`ensure_structure`()

Ensure that our h5f has the appropriate baseline structure as defined in *self.STRUCTURE*

Checks that all groups and tables are made, makes them if not

`update_biography`(*params*)

Change or make a new biographical attribute, stored as attributes of the *info* group.

Parameters **params** (*dict*) – biographical attributes to be updated.

`update_history`(*type*, *name*, *value*, *step=None*)

Update the history table when changes are made to the subject's protocol.

The current protocol is flushed to the past_protocols group and an updated filenode is created.

Note: This **only** updates the history table, and does not make the changes itself.

Parameters

- **type** (*str*) – What type of change is being made? Can be one of
  - 'param' - a parameter of one task stage
  - 'step' - the step of the current protocol
  - 'protocol' - the whole protocol is being updated.
- **name** (*str*) – the name of either the parameter being changed or the new protocol
- **value** (*str*) – the value that the parameter or step is being changed to, or the protocol dictionary flattened to a string.
- **step** (*int*) – When type is 'param', changes the parameter at a particular step, otherwise the current step is used.

`assign_protocol`(*protocol*, *step_n=0*)

Assign a protocol to the subject.

If the subject has a currently assigned task, stashes it with `stash_current()`

Creates groups and tables according to the data descriptions in the task class being assigned. eg. as described in `Task.TrialData`.

Updates the history table.

Parameters

- **protocol** (*str*) – the protocol to be assigned. Can be one of
  - the name of the protocol (its filename minus .json) if it is in *prefs.get('PROTOCOLDIR')*
  - filename of the protocol (its filename with .json) if it is in the *prefs.get('PROTOCOLDIR')*

  – the full path and filename of the protocol.

- **step_n** (*int*) – Which step is being assigned?

flush_current()
Flushes the 'current' attribute in the subject object to the current filenode in the .h5

Used to make sure the stored .json representation of the current task stays up to date with the params set in the subject object

stash_current()
Save the current protocol in the history group and delete the node

Typically this is called when assigning a new protocol.

Stored as the date that it was changed followed by its name if it has one

prepare_run()
Prepares the Subject object to receive data while running the task.

Gets information about current task, trial number, spawns `Graduation` object, spawns `data_queue` and calls `data_thread()`.

Returns

the parameters for the current step, with subject id, step number, current trial, and session number included.

Return type Dict

data_thread(*queue*)
Thread that keeps hdf file open and receives data while task is running.

receives data through `queue` as dictionaries. Data can be partial-trial data (eg. each phase of a trial) as long as the task returns a dict with 'TRIAL_END' as a key at the end of each trial.

each dict given to the queue should have the *trial_num*, and this method can properly store data without passing *TRIAL_END* if so. I recommend being explicit, however.

Checks graduation state at the end of each trial.

Parameters **queue** (`queue.Queue`) – passed by `prepare_run()` and used by other objects to pass data to be stored.

save_data(*data*)
Alternate and equivalent method of putting data in the queue as *Subject.data_queue.put(data)*

Parameters **data** (*dict*) – trial data. each should have a 'trial_num', and a dictionary with key 'TRIAL_END' should be passed at the end of each trial.

stop_run()
puts 'END' in the data_queue, which causes `data_thread()` to end.

to_csv(*path*, *task='current'*, *step='all'*)
Export trial data to .csv

Parameters

- **path** (*str*) – output path of .csv
- **task** (*str, int*) – not implemented, but in the future pull data from 'current' or other named task
- **step** (*str, int, list, tuple*) – Step to select, see `Subject.get_trial_data()`

get_trial_data(*step: Union[int, list, str] = - 1*, *what: str = 'data'*)
Get trial data from the current task.

Parameters

- **step** (*int, list, 'all'*) – Step that should be returned, can be one of
  - -1: most recent step
  - int: a single step
  - list of two integers eg. [0, 5], an inclusive range of steps.
  - string: the name of a step (excluding S##_)
  - 'all': all steps.
- **what** (*str*) – What should be returned?
  - 'data' : Dataframe of requested steps' trial data
  - 'variables': dict of variables *without* loading data into memory

Returns DataFrame of requested steps' trial data.

Return type `pandas.DataFrame`

apply_along(*along='session'*, *step=- 1*)

get_step_history(*use_history=True*)
Gets a dataframe of step numbers, timestamps, and step names as a coarse view of training status.

Parameters **use_history** (*bool*) – whether to use the history table or to reconstruct steps and dates from the trial table itself. compatibility fix for old versions that didn't stash step changes when the whole protocol was updated.

Returns `pandas.DataFrame`

get_timestamp(*simple=False*)
Makes a timestamp.

Parameters **simple** (*bool*) –

if True: returns as format '%y%m%d-%H%M%S', eg '190201-170811'

if False: returns in isoformat, eg. '2019-02-01T17:08:02.058808'

Returns basestring

get_weight(*which='last'*, *include_baseline=False*)
Gets start and stop weights.

Todo: add ability to get weights by session number, dates, and ranges.

Parameters

- **which** (*str*) – if 'last', gets most recent weights. Otherwise returns all weights.
- **include_baseline** (*bool*) – if True, includes baseline and minimum mass.

Returns dict

`set_weight(date, col_name, new_value)`
Updates an existing weight in the weight table.

Todo: Yes, i know this is bad. Merge with update_weights

Parameters

- **date** (*str*) – date in the 'simple' format, %y%m%d-%H%M%S
- **col_name** (*'start', 'stop'*) – are we updating a pre-task or post-task weight?
- **new_value** (*float*) – New mass.

`update_weights(start=None, stop=None)`
Store either a starting or stopping mass.

start and *stop* can be passed simultaneously, *start* can be given in one call and *stop* in a later call, but *stop* should not be given before *start*.

Parameters

- **start** (*float*) – Mass before running task in grams
- **stop** (*float*) – Mass after running task in grams.

`graduate()`
Increase the current step by one, unless it is the last step.

`class History_Table`
Bases: `tables.description.IsDescription`

Class to describe parameter and protocol change history

Variables

- **time** (*str*) – timestamps
- **type** (*str*) – Type of change - protocol, parameter, step
- **name** (*str*) – Name - Which parameter was changed, name of protocol, manual vs. graduation step change
- **value** (*str*) – Value - What was the parameter/protocol/etc. changed to, step if protocol.

Attributes:

columns

```
columns = { 'name':  StringCol(itemsize=256, shape=(), dflt=b'', pos=None),
'time':  StringCol(itemsize=256, shape=(), dflt=b'', pos=None), 'type':
StringCol(itemsize=256, shape=(), dflt=b'', pos=None), 'value':
StringCol(itemsize=4028, shape=(), dflt=b'', pos=None)}
```

`class Weight_Table`
Bases: `tables.description.IsDescription`

Class to describe table for weight history

Variables

- **start** (*float*) – Pre-task mass

- **stop** (*float*) – Post-task mass
- **date** (*str*) – Timestamp in simple format
- **session** (*int*) – Session number

Attributes:

`columns`

columns = { 'date': StringCol(itemsize=256, shape=(), dflt=b'', pos=None), 'session': Int32Col(shape=(), dflt=0, pos=None), 'start': Float32Col(shape=(), dflt=0.0, pos=None), 'stop': Float32Col(shape=(), dflt=0.0, pos=None)}

class Hash_Table

Bases: `tables.description.IsDescription`

Class to describe table for hash history

Variables

- **time** (*str*) – Timestamps
- **hash** (*str*) – Hash of the currently checked out commit of the git repository.

Attributes:

`columns`

columns = { 'hash': StringCol(itemsize=40, shape=(), dflt=b'', pos=None), 'time': StringCol(itemsize=256, shape=(), dflt=b'', pos=None)}

11.7 terminal

Methods for running the Terminal GUI

Classes:

Terminal()	Central host to a swarm of *Pilot* s and user-facing *gui* objects.

class Terminal

Bases: `PySide2.QtWidgets.QMainWindow`

Central host to a swarm of *Pilot* s and user-facing *gui* objects.

Called as a module with the -f flag to give the location of a prefs file, eg:

```
python terminal.py -f prefs_file.json
```

if the -f flag is not passed, looks in the default location for prefs (ie. */usr/autopilot/prefs.json*)

**Listens used by the internal *Net_Node* **

Key	Method	Description
'STATE'	*l_state()*	A Pi has changed state
'PING'	*l_ping()*	Someone wants to know if we're alive
'DATA'	*l_data()*	Receiving data to store
'HANDSHAKE'	*l_handshake()*	Pilot first contact, telling us it's alive and its IP

Note: See *autopilot.prefs* for full list of prefs needed by terminal!

Variables

- **node** (Net_Node) – Our Net_Node we use to communicate with our main networking object
- **networking** (Terminal_Station) – Our networking object to communicate with the outside world
- **subjects** (*dict*) – A dictionary mapping subject ID to *Subject* object.
- **layout** (QtWidgets.QGridLayout) – Layout used to organize widgets
- **control_panel** (*Control_Panel*) – Control Panel to manage pilots and subjects
- **data_panel** (*Plot_Widget*) – Plots for each pilot and subject.
- **logo** (QtWidgets.QLabel) – Label holding our beautiful logo ;X
- **logger** (logging.Logger) – Used to log messages and network events.
- **settings** (PySide2.QtCore.QSettings) – QSettings used to store pyside configuration like window size, stored in prefs.get("TERMINAL_SETTINGS_FN")

Methods:

initUI()	Initializes graphical elements of Terminal.
reset_ui()	Clear Layout and call *initUI()* again
ping_pilot(pilot)	
heartbeat([once])	Periodically send an INIT message that checks the status of connected pilots
toggle_start(starting, pilot[, subject])	Start or Stop running the currently selected subject's task.
l_data(value)	A Pilot has sent us data.
l_ping(value)	**Todo:** Reminder to implement heartbeating.
l_state(value)	A Pilot has changed state, keep track of it.
l_handshake(value)	Pilot is sending its IP and state on startup.
new_pilot([name, ip, pilot_prefs])	Make a new entry in *Terminal.pilots* and make appropriate GUI elements.
new_protocol()	Open a *gui.Protocol_Wizard* to create a new protocol.
subject_weights()	Gets recent weights from all subjects and open a *gui.Weights* window to view or set weights.

continues on next page

Table 81 – continued from previous page

update_protocols()	If we change the protocol file, update the stored version in subject files
reassign_protocols()	Batch reassign protocols and steps.
calibrate_ports()	Calibrate *hardware.gpio.Solenoid* objects.
test_bandwidth()	Test bandwidth of Pilot connection with variable sized arrays as paylods
plot_psychometric()	Select subject, step, and variables to plot a psychometric curve
manage_plugins()	
stream_video()	Open a window to stream videos from a connected pilot.
closeEvent(event)	When Closing the Terminal Window, close any running subject objects, 'KILL' our networking object.

Attributes:

pilots	A dictionary mapping pilot ID to its attributes, including a list of its subjects assigned to it, its IP, etc.
protocols	List of protocol names available in `PROTOCOLDIR`
subject_protocols	Returns: subject_protocols (dict): a dictionary of subjects: [protocol, step]
subject_list	Get a list of all subject IDs
staticMetaObject	

initUI()

Initializes graphical elements of Terminal.

Including…

- Toolbar
- *gui.Control_Panel*
- *plots.Plot_Widget*

reset_ui()

Clear Layout and call *initUI()* again

property pilots: collections.OrderedDict

A dictionary mapping pilot ID to its attributes, including a list of its subjects assigned to it, its IP, etc.

Returns like `self.pilots['pilot_id'] = {'subjects': ['subject_0', 'subject_1'], 'ip': '192.168.0.101'}`

Return type dict

property protocols: list

List of protocol names available in `PROTOCOLDIR`

Returns list of protocol names in `prefs.get('PROTOCOLDIR')`

Return type list

property subject_protocols: dict

Returns: subject_protocols (dict): a dictionary of subjects: [protocol, step] `property subject_list: list`

Get a list of all subject IDs

Returns list of all subject IDs present in `Terminal.pilots`

Return type list

`ping_pilot(pilot)`

`heartbeat(once=False)`

Periodically send an `INIT` message that checks the status of connected pilots

sent with frequency according to `Terminal.heartbeat_dur`

Parameters **once** (*bool*) – if True, do a single heartbeat but don't start a thread to do more.

`toggle_start(starting, pilot, subject=None)`

Start or Stop running the currently selected subject's task. Sends a message containing the task information to the concerned pilot.

Each `Pilot_Panel` is given a lambda function that calls this one with the arguments specified See `Pilot_Button`, as it is what calls this function.

Parameters

- **starting** (*bool*) – Does this button press mean we are starting (True) or stopping (False) the task?
- **pilot** – Which Pilot is starting or stopping?
- **subject** – Which Subject is currently selected?

`l_data(value)`

A Pilot has sent us data.

value field of message should have *subject* and *pilot* added to dictionary for identification.

Any key in *value* that matches a column in the subject's trial data table will be saved.

If the subject graduates after receiving this piece of data, stop the current task running on the Pilot and send the new one.

Parameters **value** (*dict*) – A dict of field-value pairs to save

`l_ping(value)`

Todo: Reminder to implement heartbeating.

Note: Currently unused, as Terminal Net_Node stability hasn't been a problem and no universal system of heartbeating has been established (global stability has not been an issue).

Parameters **value** – (unused)

`l_state(value)`

A Pilot has changed state, keep track of it.

Parameters **value** (*dict*) – dict containing *state* .

l_handshake(*value*)
Pilot is sending its IP and state on startup.

If we haven't heard of this pilot before, make a new entry in `pilots` and `gui.Control_Panel.update_db()` .

Parameters **value** (*dict*) – dict containing *ip* and *state*

new_pilot(*name: Optional[str] = None*, *ip: str = ''*, *pilot_prefs: Optional[dict] = None*)
Make a new entry in `Terminal.pilots` and make appropriate GUI elements.

Parameters

- **ip** (*str*) – Optional. if given, stored in db.
- **name** (*str*) – If None, prompted for a name, otherwise used for entry in pilot DB.

new_protocol()
Open a `gui.Protocol_Wizard` to create a new protocol.

Prompts for name of protocol, then saves in *prefs.get('PROTOCOLDIR')*

subject_weights()
Gets recent weights from all `subjects` and open a `gui.Weights` window to view or set weights.

update_protocols()
If we change the protocol file, update the stored version in subject files

reassign_protocols()
Batch reassign protocols and steps.

Opens a `gui.Reassign` window after getting protocol data, and applies any changes made in the window.

calibrate_ports()
Calibrate `hardware.gpio.Solenoid` objects.

See `gui.Calibrate_Water`.

After calibration routine, send results to pilot for storage.

test_bandwidth()
Test bandwidth of Pilot connection with variable sized arrays as paylods

See `gui.Bandwidth_Test`

plot_psychometric()
Select subject, step, and variables to plot a psychometric curve

manage_plugins()

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce743c59b0>

stream_video()
Open a window to stream videos from a connected pilot.

Choose from connected pilots and configured `Camera` objects (`prefs.json` sent by Pilots in `Pilot.handshake()`). Stream video, save to file.

Todo: Configure camera parameters!!!

closeEvent(*event*)
When Closing the Terminal Window, close any running subject objects, 'KILL' our networking object.

Since the *:class:*.Net_Node` keeping us alive is a *daemon*, no need to explicitly kill it.

CHAPTER

TWELVE

HARDWARE

Classes that manage hardware logic.

Each hardware class should be able to operate independently - ie. not be dependent on a particular task class, etc. Other than that there are very few design requirements:

- Every class should have a .release() method that releases any system resources in use by the object, eg. objects that use pigpio must have their *pigpio.pi* client stopped; LEDs should be explicitly turned off.
- The very minimal class attributes are described in the `Hardware` metaclass.
- Hardware methods are typically called in their own threads, so care should be taken to make any long-running operations internally threadsafe.

Note: This software was primarily developed for the Raspberry Pi, which has two types of numbering schemes , "board" numbering based on physical position (e.g. pins 1-40, in 2 rows of 20 pins) and "bcm" numbering based on the broadcom chip numbering scheme (e.g. GPIO2, GPIO27).

Board numbering is easier to use, but pigpio , which we use as a bridge between Python and the GPIOs, uses the BCM scheme. As such each class that uses the GPIOs takes a board number as its argument and converts it to a BCM number in the __init__ method.

If there is sufficient demand to make this more flexible, we can implement an additional *pref* to set the numbering scheme, but the current solution works without getting too muddy.

Warning: In order to use pigpio, the pigpio daemon must be running. See the docs Usually `Pilot` s should be started by the bash script or systemd service generated by `setup.setup_pilot`, which starts pigpiod.

Data:

BOARD_TO_BCM	Mapping from board (physical) numbering to BCM numbering.
BCM_TO_BOARD	The inverse of *BOARD_TO_BCM*.

Classes:

Hardware([name, group])	Generic class inherited by all hardware.

BOARD_TO_BCM = { 3: 2, 5: 3, 7: 4, 8: 14, 10: 15, 11: 17, 12: 18, 13: 27, 15: 22, 16: 23, 18: 24, 19: 10, 21: 9, 22: 25, 23: 11, 24: 8, 26: 7, 29: 5, 31: 6, 32: 12, 33: 13, 35: 19, 36: 16, 37: 26, 38: 20, 40: 21}

Mapping from board (physical) numbering to BCM numbering.

See this pinout.

Hardware objects take board numbered pins and convert them to BCM numbers for use with *pigpio*.

Type dict

`BCM_TO_BOARD = { 2: 3, 3: 5, 4: 7, 5: 29, 6: 31, 7: 26, 8: 24, 9: 21, 10: 19, 11: 23, 12: 32, 13: 33, 14: 8, 15: 10, 16: 36, 17: 11, 18: 12, 19: 35, 20: 38, 21: 40, 22: 15, 23: 16, 24: 18, 25: 22, 26: 37, 27: 13}`

The inverse of *BOARD_TO_BCM*.

Type dict

`class Hardware(name=None, group=None, **kwargs)`

Bases: `object`

Generic class inherited by all hardware. Should not be instantiated on its own (but it won't do anything bad so go nuts i guess).

Primarily for the purpose of defining necessary attributes.

Also defines *__del__* to call *release()* so objects are always released even if not explicitly.

Variables

- **name** (*str*) – unique name used to identify this object within its group.
- **group** (*str*) – hardware group, corresponds to key in prefs.json `"HARDWARE": {"GROUP": {"ID": {**params}}}`
- **is_trigger** (*bool*) – Is this object a discrete event input device? or, will this device be used to trigger some event? If *True*, will be given a callback by *Task*, and *assign_cb()* must be redefined.
- **pin** (*int*) – The BCM pin used by this device, or None if no pin is used.
- **type** (*str*) – What is this device known as in *.prefs*? Not required.
- **input** (*bool*) – Is this an input device?
- **output** (*bool*) – Is this an output device?

Attributes:

is_trigger	
pin	
type	
input	
output	
calibration	Calibration used by the hardware object.

Methods:

release()	Every hardware device needs to redefine *release()*, and must
assign_cb(trigger_fn)	Every hardware device that is a `trigger` must redefine this to accept a function (typically *Task.handle_trigger()*) that is called when that trigger is activated.
get_name()	Usually Hardware is only instantiated with its pin number, but we can get its name from prefs
init_networking([listens])	Spawn a *Net_Node* to `Hardware.node` for streaming or networked command

is_trigger = False

pin = None

type = ''

input = False

output = False

release()

Every hardware device needs to redefine *release()*, and must

- Safely unload any system resources used by the object, and
- Return the object to a neutral state - eg. LEDs turn off.

When not redefined, a warning is given.

assign_cb(*trigger_fn*)

Every hardware device that is a `trigger` must redefine this to accept a function (typically *Task.handle_trigger()*) that is called when that trigger is activated.

When not redefined, a warning is given.

get_name()

Usually Hardware is only instantiated with its pin number, but we can get its name from prefs

init_networking(*listens=None*, ***kwargs*)

Spawn a *Net_Node* to `Hardware.node` for streaming or networked command

Parameters

- **listens** (*dict*) – Dictionary mapping message keys to handling methods
- ****kwargs** – Passed to *Net_Node*

Returns:

property calibration: dict

Calibration used by the hardware object.

Attempt to read from `prefs.get('CALIBRATIONDIR')/group.name.json` , if `Hardware.group` is `None`, attempt to read from `prefs.get('CALIBRATIONDIR')/name.json`

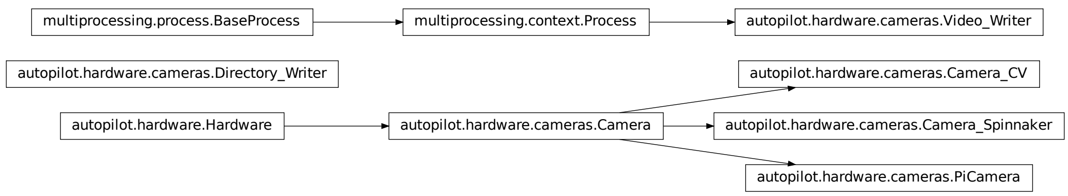
Setting the attribute (over)writes the calibration to disk as a *.json* file

Will be different for each hardware type, subclasses should document this property separately (eg. by overwriting `Hardware.calibration.__doc__`

Returns if calibration is found, a dictionary of calibration for each property. Empty if no calibration found

Return type (dict)

12.1 cameras



Classes:

Camera([fps, timed, crop])	Metaclass for Camera objects.
PiCamera(camera_idx, sensor_mode, …)	Interface to the Raspberry Pi Camera Module via picamera
Camera_CV([camera_idx])	Capture Video from a webcam with OpenCV
Camera_Spinnaker([serial, camera_idx])	Capture video from a FLIR brand camera with the Spinnaker SDK.
Video_Writer(q, path[, fps, timestamps, blosc])	Encode frames as they are acquired in a separate process.

Functions:

list_spinnaker_cameras()	List all available Spinnaker cameras and their `DeviceInformation`

OPENCV_LAST_INIT_TIME = <Synchronized wrapper for c_double(0.0)>

Time the last OpenCV camera was initialized (seconds, from time.time()).

v4l2 has an extraordinarily obnoxious …feature – if you try to initialize two cameras at ~the same time, you will get a neverending stream of informative error messages: `VIDIOC_QBUF: Invalid argument`

The workaround seems to be relatively simple, we just wait ~2 seconds if another camera was just initialized.

class Camera(*fps=None*, *timed=False*, *crop=None*, *rotate: int = 0*, ***kwargs*)

Bases: `autopilot.hardware.Hardware`

Metaclass for Camera objects. Should not be instantiated on its own.

Parameters

- **fps** (*int*) – Framerate of video capture
- **timed** (*bool, int, float*) – If False (default), camera captures indefinitely. If int or float, captures for this many seconds
- **rotate** (*int*) – Number of times to rotate image clockwise (default 0). Note that image rotation should happen in `_grab()` or be otherwise implemented in each camera subclass, because it's a common enough operation many cameras have some optimized way of doing it.

- ****kwargs** –

  Arguments to *stream()*, *write()*, and *queue()* can be passed as dictionaries, eg.:

  ```
  stream={'to':'T', 'ip':'localhost'}
  ```

When the camera is instantiated and *capture()* is called, the class uses a series of methods that should be overwritten in subclasses. Further details for each can be found in the relevant method documentation.

It is highly recommended to instantiate Cameras with a `Hardware.name`, as it is used in *output_filename* and to identify the network stream

Three methods are required to be overwritten by all subclasses:

- *init_cam()* - **required** - used by *cam*, instantiating the camera object so that it can be queried and configured
- *_grab()* - **required** - grab a frame from the *cam*
- *_timestamp()* - **required** - get a timestamp for the frame

The other methods are optional and depend on the particular camera:

- *capture_init()* - *optional* - any required routine to prepare the camera after it is instantiated but before it begins to capture
- *_process()* - *optional* - the wrapper around a full acquisition cycle, including streaming, writing, and queueing frames
- *_write_frame()* - *optional* - how to write an individual frame to disk
- *_write_deinit()* - *optional* - any required routine to finish writing to disk after acquisition
- *capture_deinit()* - *optional* - any required routine to stop acquisition but not release the camera instance.

Variables

- **frame** (*tuple*) – The current captured frame as a tuple (timestamp, frame).
- **shape** (*tuple*) – Shape of captured frames (height, width, channels)
- **blosc** (*bool*) – If True (default), use blosc compression when
- **cam** – The object used to interact with the camera
- **fps** (*int*) – Framerate of video capture
- **timed** (*bool, int, float*) – If False (default), camera captures indefinitely. If int or float, captures for this many seconds
- **q** (*Queue*) – Queue that allows frames to be pulled by other objects
- **queue_size** (*int*) – How many frames should be buffered in the queue.
- **initialized** (*threading.Event*) – Called in *init_cam()* to indicate the camera has been initialized
- **stopping** (*threading.Event*) – Called to signal that capturing should stop. when set, ends the threaded capture loop
- **capturing** (*threading.Event*) – Set when camera is actively capturing
- **streaming** (*threading.Event*) – Set to indicate that the camera is streaming data over the network
- **writing** (*threading.Event*) – Set to indicate that the camera is writing video locally

- **queueing** (*threading.Event*) – Indicates whether frames are being put into q
- **indicating** (*threading.Event*) – Set to indicate that capture progress is being indicated in stdout by tqdm

Parameters

- **fps**
- **timed**
- **crop** (*tuple*) – (x, y of top left corner, width, height)
- ****kwargs**

Attributes:

input	test documenting input
type	what are we anyway?
cam	Camera object.
output_filename	Filename given to video writer.

Methods:

capture([timed])	Spawn a thread to begin capturing.
_capture()	Threaded capture method started by capture().
_process()	A full frame capture cycle.
stream([to, ip, port, min_size])	Enable streaming frames on capture.
l_start(val)	Begin capturing by calling *Camera.capture()*
l_stop(val)	Stop capture by calling *Camera.release()*
write([output_filename, timestamps, blosc])	Enable writing frames locally on capture
_write_frame()	Put frame into the _write_q, optionally compressing it with blosc.pack_array()
_write_deinit()	End the *Video_Writer*.
queue([queue_size])	Enable stashing frames in a queue for a local consumer.
_grab()	Capture a frame and timestamp.
_timestamp([frame])	Generate a timestamp for each *_grab()*
init_cam()	Method to initialize camera object
capture_init()	Optional: Prepare *cam* after initialization, but before capture
capture_deinit()	Optional: Return *cam* to an idle state after capturing, but before releasing
stop()	Stop capture by setting stopping
release()	Release resources held by Camera.

input = True
test documenting input

type = 'CAMERA'
what are we anyway?

Type (str)

capture(*timed=None*)
Spawn a thread to begin capturing.

Parameters **timed** (*None, int, float*) – if None, record according to `timed` (default). If numeric, record for `timed` seconds.

_capture()

Threaded capture method started by `capture()`.

Captures until `stopping` is set.

Calls capture methods, in order:

- `capture_init()` - any required routine to prepare the camera after it is instantiated but before it begins to capture
- `_process()` - the wrapper around a full acquisition cycle, including streaming, writing, and queueing frames
- `_grab()` - grab a frame from the `cam`
- `_timestamp()` - get a timestamp for the frame
- `_write_frame()` - how to write an individual frame to disk
- `_write_deinit()` - any required routine to finish writing to disk after acquisition
- `capture_deinit()` - any required routine to stop acquisition but not release the camera instance.

_process()

A full frame capture cycle.

`_grab`s the :attr:()`.frame`, then handles streaming, writing, queueing, and indicating according to `stream()`, `write()`, `queue()`, and `indicating`, respectively.

stream(*to='T'*, *ip=None*, *port=None*, *min_size=5*, ***kwargs*)

Enable streaming frames on capture.

Spawns a `Net_Node` with `Hardware.init_networking()`, and creates a streaming queue with `Net_Node.get_stream()` according to args.

Sets `Camera.streaming`

Parameters

- **to** (*str*) – ID of the recipient. Default 'T' for Terminal.
- **ip** (*str*) – IP of recipient. If None (default), 'localhost'. If None and `to` is 'T', `prefs.get('TERMINALIP')`
- **port** (*int, str*) – Port of recipient socket. If None (default), `prefs.get('MSGPORT')`. If None and `to` is 'T', `prefs.get('TERMINALPORT')`.
- ****kwargs** – passed to `Hardware.init_networking()` and thus to `Net_Node`

l_start(*val*)

Begin capturing by calling `Camera.capture()`

Parameters **val** – unused

l_stop(*val*)

Stop capture by calling `Camera.release()`

Parameters **val** – unused

write(*output_filename=None*, *timestamps=True*, *blosc=True*)

Enable writing frames locally on capture

Spawns a `Video_Writer` to encode video, sets `writing`

Parameters

- **output_filename** (*str*) – path and filename of the output video. extension should be `.mp4`, as videos are encoded with libx264 by default.
- **timestamps** (*bool*) – if True, (timestamp, frame) tuples will be put in the `_write_q`. if False, timestamps will be generated by `Video_Writer` (not recommended at all).
- **blosc** (*bool*) – if true, compress frames with `blosc.pack_array()` before putting in `_write_q`.

`_write_frame`()

Put `frame` into the `_write_q`, optionally compressing it with `blosc.pack_array()`

`_write_deinit`()

End the `Video_Writer`.

Blocks until the `_write_q` is empty, holding the release of the object.

`queue`(*queue_size=128*)

Enable stashing frames in a queue for a local consumer.

Other objects can get frames as they are acquired from `q`

Parameters **queue_size** (*int*) – max number of frames that can be held in `q`

`property cam`

Camera object.

If `_cam` hasn't been initialized yet, use `init_cam()` to do so

Returns Camera object, different for each camera.

`property output_filename`

Filename given to video writer.

If explicitly set, returns as expected.

If None, or path already exists while the camera isn't capturing, a new filename is generated in the user directory.

Returns (str) `_output_filename`

`_grab`()

Capture a frame and timestamp.

Method must be overridden by subclass

Returns

(str, `numpy.ndarray`) Tuple of isoformatted (str) or numeric timestamp returned by `_timestamp()`, and captured frame

`_timestamp`(*frame=None*)

Generate a timestamp for each `_grab()`

Must be overridden by subclass

Parameters **frame** – If needed by camera subclass, pass the frame or image object to get timestamp

Returns (str, int, float) Either an isoformatted (str) or numeric timestamp

`init_cam`()

Method to initialize camera object

Must be overridden by camera subclass

Returns camera object

capture_init()

Optional: Prepare *cam* after initialization, but before capture

Returns None

capture_deinit()

Optional: Return *cam* to an idle state after capturing, but before releasing

Returns None

stop()

Stop capture by setting `stopping`

release()

Release resources held by Camera.

Must be overridden by subclass.

Does not raise exception in case some general camera release logic should be put here...

class PiCamera(*camera_idx: int = 0, sensor_mode: int = 0, resolution: Tuple[int, int] = (1280, 720), fps: int = 30, format: str = 'rgb', *args, **kwargs*)

Bases: `autopilot.hardware.cameras.Camera`

Interface to the Raspberry Pi Camera Module via picamera

Parameters of the `picamera.PiCamera` class can be set after initialization by modifying the `PiCamera.cam` attribute, eg `PiCamera().cam.exposure_mode = 'fixedfps'` – see the `picamera.PiCamera` documentation for full documentation.

Note that some parameters, like resolution, can't be changed after starting `capture()` .

The Camera Module is a slippery little thing, and `fps` and `resolution` are just requests to the camera, and aren't necessarily followed with 100% fidelity. The possible framerates and resolutions are determined by the `sensor_mode` parameter, which by default tries to guess the best sensor mode based on the fps and resolution. See the Sensor Modes documentation for more details.

This wrapper uses a subclass, `PiCamera.PiCamera_Writer` to capture frames decoded by the gpu directly from the preallocated buffer object. Currently the restoration from the buffer assumes that RGB, or generally `shape[2] == 3`, images are being captured. See this stackexchange post by Dave Jones, author of the picamera module, for a strategy for capturing grayscale images quickly.

This class also currently uses the default `Video_Writer` object, but it could be more performant to use the `picamera.PiCamera.start_recording()` method's built-in ability to record video to a file — try it out!

Todo: Currently timestamps are constructed with `datetime.datetime.now.isoformat()`, which is not altogether accurate. Timestamps should be gotten from the `frame` attribute, which depends on the `clock_mode`

References

- https://blog.robertelder.org/recording-660-fps-on-raspberry-pi-camera/
- Fast capture from the author of picamera - https://raspberrypi.stackexchange.com/a/58941/112948
- More on fast capture and processing, see last example in section - https://picamera.readthedocs.io/en/release-1.12/recipes2.html#rapid-capture

Parameters

- **camera_idx** (*int*) – Index of picamera (default: 0, >=1 only supported on compute module)
- **sensor_mode** (*int*) – Sensor mode, default 0 detects automatically from resolution and fps, note that sensor_mode will affect the available resolutions and framerates, see Sensor Modes for more information
- **resolution** (*tuple*) – a tuple of (width, height) integers, but mind the note in the above documentation regarding the sensor_mode property and resolution
- **fps** (*int*) – frames per second, but again mind the note on sensor_mode
- **format** (*str*) – Format passed to :class`picamera.PiCamera.start_recording` one of `('rgb' (default), 'grayscale')` The `'grayscale'` format uses the `'yuv'` format, and extracts the luminance channel
- ***args ()** – passed to superclass
- ****kwargs ()** – passed to superclass

Attributes:

`sensor_mode`	Sensor mode, default 0 detects automatically from resolution and fps, note that sensor_mode will affect the available resolutions and framerates, see Sensor Modes for more information.
`resolution`	A tuple of ints, (width, height).
`fps`	Frames per second
`rotation`	Rotation of the captured image, derived from `Camera.rotate` * 90.

Methods:

`init_cam()`	Initialize and return the `picamera.PiCamera` object.
`capture_init()`	Spawn a `PiCamera.PiCamera_Writer` object to `PiCamera._picam_writer` and `start_recording()` in the set `format`
`_grab()`	Wait on the `grab_event` to be set, then clear it before returning the frame.
`capture_deinit()`	`stop_recording()` and `close()` the camera, releasing its resources.
`release()`	Release resources held by Camera.

Classes:

PiCamera_Writer(resolution, int], format)	Writer object for processing individual frames, see: https://raspberrypi.stackexchange.com/a/58941/112948

property sensor_mode: int
Sensor mode, default 0 detects automatically from resolution and fps, note that sensor_mode will affect the available resolutions and framerates, see Sensor Modes for more information.

When set, if the camera has been initialized, will change the attribute in `PiCamera.cam`

Returns int

property resolution: Tuple[int, int]
A tuple of ints, (width, height).

Resolution can't be changed while the camera is capturing.

See Sensor Modes for more information re: how resolution relates to `picamera.PiCamera.sensor_mode`

Returns tuple of ints, (width, height)

property fps: int
Frames per second

See Sensor Modes for more information re: how fps relates to `picamera.PiCamera.sensor_mode`

Returns int - fps

property rotation: int
Rotation of the captured image, derived from `Camera.rotate` * 90.

Must be one of `(0, 90, 180, 270)`

Rotation can be changed during capture

Returns int - Current rotation

init_cam() → picamera.PiCamera
Initialize and return the `picamera.PiCamera` object.

Uses the stored `camera_idx`, *resolution*, *fps*, and *sensor_mode* attributes on init.

Returns `picamera.PiCamera`

capture_init()
Spawn a *PiCamera.PiCamera_Writer* object to `PiCamera._picam_writer` and `start_recording()` in the set `format`

_grab() → Tuple[str, numpy.ndarray]
Wait on the `grab_event` to be set, then clear it before returning the frame.

Returns (timestamp, frame) tuple

capture_deinit()
`stop_recording()` and `close()` the camera, releasing its resources.

release()
Release resources held by Camera.

Must be overridden by subclass.

Does not raise exception in case some general camera release logic should be put here…

class PiCamera_Writer(*resolution: Tuple[int, int], format: str = 'rgb'*)

Bases: `object`

Writer object for processing individual frames, see: https://raspberrypi.stackexchange.com/a/58941/112948

Parameters resolution (*tuple*) – (width, height) tuple used when making numpy array from buffer

Variables

- **grab_event** (`threading.Event`) – Event set whenever a new frame is captured, cleared by the parent class when the frame is consumed.
- **frame** (`numpy.ndarray`) – Captured frame
- **timestamp** (*str*) – Isoformatted timestamp of time of capture.

Methods:

`write`(buf)	Reconstutute the buffer into a numpy array in `PiCamera_Writer.frame` and make a timestamp in `PiCamera_Writer.timestamp`, then set the `PiCamera_Writer.grab_event`

write(*buf*)

Reconstutute the buffer into a numpy array in `PiCamera_Writer.frame` and make a timestamp in `PiCamera_Writer.timestamp`, then set the `PiCamera_Writer.grab_event`

Parameters buf () – Buffer given by PiCamera

class Camera_CV(*camera_idx=0, **kwargs*)

Bases: `autopilot.hardware.cameras.Camera`

Capture Video from a webcam with OpenCV

By default, OpenCV will select a suitable backend for the indicated camera. Some backends have difficulty operating multiple cameras at once, so the performance of this class will be variable depending on camera type.

Parameters

- **camera_idx** (*int*) – The index of the desired camera
- ****kwargs** – Passed to the `Camera` metaclass.

Variables

- **camera_idx** (*int*) – The index of the desired camera
- **last_opencv_init** (*float*) – See `OPENCV_LAST_INIT_TIME`
- **last_init_lock** (`threading.Lock`) – Lock for setting `last_opencv_init`

Note: OpenCV must be installed to use this class! A Prebuilt opencv binary is available for the raspberry pi, but it doesn't take advantage of some performance-enhancements available to OpenCV. Use the `install_opencv.sh` script in the setup directory to compile OpenCV with these enhancements.

If your camera isn't working, to print debugging information you can run:

```
echo 3 > /sys/class/video4linux/videox/dev_debug
```

(continues on next page)

(continued from previous page)

```
# check logs
dmesg
```

Attributes:

fps	Attempts to get FPS with `cv2.CAP_PROP_FPS`, uses 30fps as a default
shape	Attempts to get image shape from `cv2.CAP_PROP_FRAME_WIDTH` and `HEIGHT` :returns: (width, height) :rtype: tuple
backend	capture backend used by OpenCV for this camera
v4l_info	Device information from `v4l2-ctl`

Methods:

_grab()	Reads a frame with `cam.read()`
_timestamp([frame])	Attempts to get timestamp with `cv2.CAP_PROP_POS_MSEC`.
init_cam()	Initializes OpenCV Camera
release()	Release resources held by Camera.

property fps

Attempts to get FPS with `cv2.CAP_PROP_FPS`, uses 30fps as a default

Returns framerate

Return type int

property shape

Attempts to get image shape from `cv2.CAP_PROP_FRAME_WIDTH` and `HEIGHT` :returns: (width, height) :rtype: tuple

_grab()

Reads a frame with `cam.read()`

Returns (timestamp, frame)

Return type tuple

_timestamp(*frame=None*)

Attempts to get timestamp with `cv2.CAP_PROP_POS_MSEC`. Frame does not need to be passed to this method, as timestamps are retrieved from *cam*

Todo: Convert this float timestamp to an isoformatted system timestamp

Returns milliseconds since capture start

Return type float

property backend

capture backend used by OpenCV for this camera

Returns name of capture backend used by OpenCV for this camera

Return type str

init_cam()

Initializes OpenCV Camera

To avoid overlapping resource allocation requests, checks the last time any *Camera_CV* object was instantiated and makes sure it has been at least 2 seconds since then.

Returns camera object

Return type `cv2.VideoCapture`

release()

Release resources held by Camera.

Must be overridden by subclass.

Does not raise exception in case some general camera release logic should be put here…

property v4l_info

Device information from `v4l2-ctl`

Returns Information for all devices available through v4l2

Return type dict

class Camera_Spinnaker(*serial=None*, *camera_idx=None*, ***kwargs*)

Bases: `autopilot.hardware.cameras.Camera`

Capture video from a FLIR brand camera with the Spinnaker SDK.

Parameters

- **serial** (*str*) – Serial number of desired camera
- **camera_idx** (*int*) – If no serial provided, select camera by index. Using `serial` is HIGHLY RECOMMENDED.
- ****kwargs** – passed to *Camera* metaclass

Note: PySpin and the Spinnaker SDK must be installed to use this class. Please use the `install_pyspin.sh` script in `setup`

See the documentation for the Spinnaker SDK and PySpin here:

https://www.flir.com/products/spinnaker-sdk/

Variables

- **serial** (*str*) – Serial number of desired camera
- **camera_idx** (*int*) – If no serial provided, select camera by index. Using `serial` is HIGHLY RECOMMENDED.
- **system** (`PySpin.System`) – The PySpin System object
- **cam_list** (`PySpin.CameraList`) – The list of PySpin Cameras available to the system
- **nmap** – A reference to the nodemap from the GenICam XML description of the device
- **base_path** (*str*) – The directory and base filename that images will be written to if object is `writing`. eg:

  base_path = '/home/user/capture_directory/**capture_**' image_path = base_path + 'image1.png'
- **img_opts** (`PySpin.PNGOption`) – Options for saving .png images, made by *write()*

Attributes:

ATTR_TYPES	Conversion from data types to pointer types
ATTR_TYPE_NAMES	Conversion from data types to human-readable names
RW_MODES	bool, 'write':bool} descriptor
bin	Camera Binning.
exposure	Set Exposure of camera
fps	Acquisition Framerate
frame_trigger	Set camera to lead or follow hardware triggers
acquisition_mode	Image acquisition mode
readable_attributes	All device attributes that are currently readable with get()
writable_attributes	All device attributes that are currently writeable wth set()
device_info	Get all information about the camera

Methods:

init_cam()	Initialize the Spinnaker Camera
capture_init()	Prepare the camera for acquisition
capture_deinit()	De-initializes the camera after acquisition
_process()	Modification of the Camera._process() method for Spinnaker cameras
_grab()	Get next timestamp and PySpin Image
_timestamp([frame])	Get the timestamp from the passed image
write([output_filename, timestamps, blosc])	Sets camera to save acquired images to a directory for later encoding.
_write_frame()	Write frame to `base_path` + timestamp + '.png' with `PySpin.Image.Save()`
_write_deinit()	After capture, write images in `base_path` to video with `Directory_Writer`
get(attr)	Get a camera attribute.
set(attr, val)	Set a camera attribute
list_options(name)	List the possible values of a camera attribute.
release()	Release all PySpin objects and wait on writer, if still active.

ATTR_TYPES = {}
Conversion from data types to pointer types

ATTR_TYPE_NAMES = {}
Conversion from data types to human-readable names

RW_MODES = {}
bool, 'write':bool} descriptor

Type Conversion from read/write mode to {'read'

init_cam()
Initialize the Spinnaker Camera

Initializes the camera, system, cam_list, node map, and the camera methods and attributes used by `get()` and `set()`

Returns The Spinnaker camera object

Return type `PySpin.Camera`

capture_init()

Prepare the camera for acquisition

calls the camera's `BeginAcquisition` method and populate *shape*

capture_deinit()

De-initializes the camera after acquisition

_process()

Modification of the *Camera._process()* method for Spinnaker cameras

Because the objects returned from the *_grab()* method are image *pointers* rather than :class:`numpy.ndarray`s, they need to be handled differently.

More details on the differences are given in the *_write_frame()*,

_grab()

Get next timestamp and PySpin Image

Returns (timestamp, `PySpin.Image`)

Return type tuple

_timestamp(*frame=None*)

Get the timestamp from the passed image

Parameters **frame** (`PySpin.Image`) – Currently grabbed image

Returns PySpin timestamp

Return type float

write(*output_filename=None*, *timestamps=True*, *blosc=True*)

Sets camera to save acquired images to a directory for later encoding.

For performance, rather than encoding during acquisition, save each image as a (lossless) .png image in a directory generated by *output_filename*.

After capturing is complete, a `Directory_Writer` encodes the images to an x264 encoded .mp4 video.

Parameters

- **output_filename** (*str*) – Directory to write images to. If None (default), generated by *output_filename*
- **timestamps** (*bool*) – Not used, timestamps are always appended to filenames.
- **blosc** (*bool*) – Not used, images are directly saved.

_write_frame()

Write frame to `base_path` + timestamp + '.png' with `PySpin.Image.Save()`

_write_deinit()

After capture, write images in `base_path` to video with `Directory_Writer`

Camera object will remain open until writer has finished.

property bin

Camera Binning.

Attempts to bin on-device, and use averaging if possible. If averaging not available, uses summation.

Parameters **tuple** – tuple of integers, (Horizontal, Vertical binning)

Returns (Horizontal, Vertical binning)

Return type tuple

property exposure

Set Exposure of camera

Can be set with

- `'auto'` - automatic exposure control. note that this will limit framerate
- `float` from 0-1 - exposure duration proportional to fps. eg. if fps = 10, setting exposure = 0.5 means exposure will be set as 50ms
- `float` or `int` >1 - absolute exposure time in microseconds

Returns If exposure has been set, return set value. Otherwise return `.get('ExposureTime')`

Return type str, float

property fps

Acquisition Framerate

Set with integer. If set with None, ignored (superclass sets FPS to None on init)

Returns from `cam.AcquisitionFrameRate.GetValue()`

Return type int

property frame_trigger

Set camera to lead or follow hardware triggers

If `'lead'`, Camera will send TTL pulses from Line 2.

If `'follow'`, Camera will follow triggers from Line 3.

See also:

- https://www.flir.com/support-center/iis/machine-vision/application-note/configuring-synchronized-capture-with-multiple-cameras
- https://www.flir.com/support-center/iis/machine-vision/knowledge-base/what-external-iidc-trigger-modes-are-supported-by-my-camera/

property acquisition_mode

Image acquisition mode

One of

- `'continuous'` - continuously acquire frame camera
- `'single'` - acquire a single frame
- `'multi'` - acquire a finite number of frames.

> **Warning:** Only `'continuous'` has been tested.

property readable_attributes

All device attributes that are currently readable with *`get()`*

Returns A dictionary of attributes that are readable and their current values

Return type dict

property writable_attributes
All device attributes that are currently writeable wth *set()*

Returns A dictionary of attributes that are writeable and their current values

Return type dict

get(*attr*)
Get a camera attribute.

Any value in *readable_attributes* can be read. Attempts to get numeric values with `.GetValue`, otherwise gets a string with `.ToString`, so be cautious with types.

If `attr` is a method (ie. in `._camera_methods`, execute the method and return the value

Parameters **attr** (*str*) – Name of a readable attribute or executable method

Returns Value of `attr`

Return type float, int, str

set(*attr*, *val*)
Set a camera attribute

Any value in `writeable_attributes` can be set. If attribute has a `.SetValue` method, (ie. accepts numeric values), attempt to use it, otherwise use `.FromString`.

Parameters

- **attr** (*str*) – Name of attribute to be set
- **val** (*str, int, float*) – Value to set attribute

list_options(*name*)
List the possible values of a camera attribute.

Parameters **name** (*str*) – name of attribute to query

Returns Dictionary with {available options: descriptions}

Return type dict

property device_info
Get all information about the camera

Note that this is distinct from camera *attributes* like fps, instead this is information like serial number, version, firmware revision, etc.

Returns {feature name: feature value}

Return type dict

release()
Release all PySpin objects and wait on writer, if still active.

class Video_Writer(*q*, *path*, *fps=None*, *timestamps=True*, *blosc=True*)
Bases: `multiprocessing.context.Process`

Encode frames as they are acquired in a separate process.

Must call `start()` after initialization to begin encoding.

Encoding continues until 'END' is put in `q`.

Timestamps are saved in a .csv file with the same path as the video.

Parameters

- **q** (Queue) – Queue into which frames will be dumped
- **path** (*str*) – output path of video
- **fps** (*int*) – framerate of output video
- **timestamps** (*bool*) – if True (default), input will be of form (timestamp, frame). if False, input will just be frames and timestamps will be generated as the frame is encoded (**not recommended**)
- **blosc** (*bool*) – if True, frames in the q will be compressed with blosc. if False, uncompressed

Variables **timestamps** (*list*) – Timestamps for frames, written to .csv on completion of encoding

Methods:

run()	Open a skvideo.io.FFmpegWriter and begin processing frames from q

run()

Open a skvideo.io.FFmpegWriter and begin processing frames from q

Should not be called by itself, overwrites the multiprocessing.Process.run() method, so should call Video_Writer.start()

Continue encoding until 'END' put in queue.

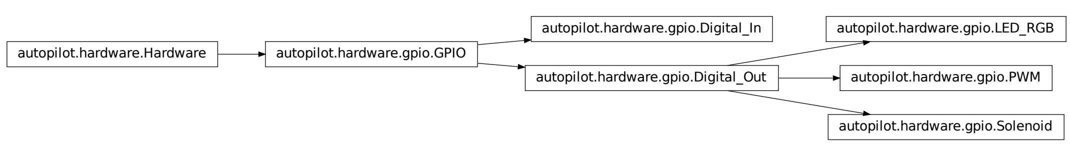
list_spinnaker_cameras()

List all available Spinnaker cameras and their DeviceInformation

Returns list of dictionaries of device information for each camera.

Return type list

12.2 gpio



Hardware that uses the GPIO pins of the Raspi. These classes rely on pigpio, whose daemon (pigpiod) must be running in the background – typically this is handled with a launch script/system daemon (see the launch_pilot.sh script generated by setup_autopilot.py)

Autopilot uses a custom version of pigpio (https://github.com/sneakers-the-rat/pigpio) that returns isoformatted timestamps rather than tick numbers in callbacks. See the setup_pilot.sh script.

Note: Autopilot uses the "Board" rather than "Broadcom" numbering system, see *the numbering note*. *GPIO* objects convert internally between board and bcm numbers using *GPIO.pin* , GPIO.pin_bcm , *BOARD_TO_BCM* , and *BCM_TO_BOARD* .

Note: This module does not include hardware that uses the GPIO pins over a specific protocol like i2c

Data:

TRIGGER_MAP	Maps user input descriptions of triggers to the corresponding pigpio object.
INVERSE_TRIGGER_MAP	Inverse of *TRIGGER_MAP*.
PULL_MAP	Maps user input descriptions of internal resistor pullups/downs to the corresponding pigpio object.
INVERSE_PULL_MAP	Inverse of *PULL_MAP*, mapping pigpio objects for internal resistor pullups/downs to their canonical form ('U', 'D', None for pullup, pulldown, or no pull)
ENABLED	False if pigpio cannot be imported – and GPIO devices cannot be used.

Functions:

clear_scripts([max_scripts])	Stop and delete all scripts running on the pigpio client.

Classes:

GPIO([pin, polarity, pull, trigger])	Metaclass for hardware that uses GPIO.
Digital_Out([pin, pulse_width, polarity])	TTL/Digital logic out through a GPIO pin.
Digital_In(pin[, event, record])	Record digital input and call one or more callbacks on logic transition.
PWM(pin[, range])	PWM output from GPIO.
LED_RGB([pins, r, g, b, polarity, blink])	An RGB LED, wrapper around three *PWM* objects.
Solenoid(pin[, polarity, duration, vol])	Solenoid valve for water delivery.

TRIGGER_MAP = {0: 1, 1: 0, 'B': 2, 'D': 1, 'U': 0, (0, 1): 2}

Maps user input descriptions of triggers to the corresponding pigpio object.

INVERSE_TRIGGER_MAP = {0: 'U', 1: 'D', 2: 'B'}

Inverse of *TRIGGER_MAP*. Used to assign canonical references to triggers – ie. it is possible to take multiple params (1, True, 'U') -> pigpio trigger objects, but there is one preferred way to refer to a pigpio object.

PULL_MAP = {None: 0, 0: 1, 1: 2, 'D': 1, 'U': 2}

Maps user input descriptions of internal resistor pullups/downs to the corresponding pigpio object.

INVERSE_PULL_MAP = {0: None, 1: 'D', 2: 'U'}

Inverse of *PULL_MAP*, mapping pigpio objects for internal resistor pullups/downs to their canonical form ('U', 'D', None for pullup, pulldown, or no pull)

ENABLED = True

False if pigpio cannot be imported – and GPIO devices cannot be used.

True if pigpio can be imported

clear_scripts(*max_scripts=256*)

Stop and delete all scripts running on the pigpio client.

To be called, eg. between tasks to ensure none are left hanging by badly behaved GPIO devices

Parameters **max_scripts** (*int*) – maximum number of scripts allowed by pigpio. Set in `pigpio.c` and not exported to the python module, so have to hardcode it again here, default for pigpio fork is 256

class GPIO(*pin=None*, *polarity=1*, *pull=None*, *trigger=None*, ***kwargs*)

Bases: `autopilot.hardware.Hardware`

Metaclass for hardware that uses GPIO. Should not be instantiated on its own.

Handles initializing pigpio and wraps some of its commonly used methods

Parameters

- **pin** (*int*) – The Board-numbered GPIO pin of this object.
- **polarity** (*int*) – Logic direction. if 1: on=High=1, off=Low=0; if 0: off=Low=0, on=High=1
- **pull** (*str, int*) – state of pullup/down resistor. Can be set as 'U'/'D' or 1/0 to pull up/down. See `PULL_MAP`
- **trigger** (*str, int, bool*) – whether callbacks are triggered on rising ('U', 1, True), falling ('D', 0, False), or both edges ('B', (0,1))
- **kwargs** – passed to the `Hardware` superclass.

Variables

- **pig** (`pigpio.pi`) – An object that manages connection to the pigpio daemon. See docs at http://abyz.me.uk/rpi/pigpio/python.html
- **CONNECTED** (`bool`) – Whether the connection to pigpio was successful
- **pigpiod** – Reference to the pigpiod process launched by `external.start_pigpiod()`
- **pin** (`int`) – The Board-numbered GPIO pin of this object.
- **pin_bcm** (`int`) – The BCM number of the connected pin – used by pigpio. Converted from pin passed as argument on initialization, which is assumed to be the board number.
- **pull** (`str, int`) – state of pullup/down resistor. Can be set as 'U'/'D' or 1/0 to pull up/down
- **polarity** (`int`) – Logic direction. if 1: on=High=1, off=Low=0; if 0: off=Low=0, on=High=1
- **on** (`int`) – if polarity == 1, high/1. if polarity == 0, low/0
- **off** (`int`) – if polarity == 1, low/0. if polarity == 0, high/1
- **trigger** (`str, int, bool`) – whether callbacks are triggered on rising ('U', 1, True), falling ('D', 0, False), or both edges ('B', (0,1))
- **trigger_edge** – The pigpio object representing RISING_EDGE, FALLING_EDGE, BOTH_EDGES. Set by :attr`.trigger`

Methods:

`init_pigpio`()	Create a socket connection to the pigpio daemon and set as `GPIO.pig`
`release`()	Release the connection to the pigpio daemon.

Attributes:

pin	//raspberrypi.stackexchange.com/a/12967>`_ GPIO pin.
state	Instantaneous state of GPIO pin, on (`True`) or off (`False`)
pull	State of internal pullup/down resistor.
polarity	on=High=1, off=Low=0; if 0: off=Low=0, on=High=1.
trigger	Maps strings ((‘U’,1,True), (‘D’,0,False), (‘B’,[0,1])) to pigpio edge types (RISING_EDGE, FALLING_EDGE, EITHER_EDGE), respectively.

init_pigpio() → bool

Create a socket connection to the pigpio daemon and set as `GPIO.pig`

Returns True if connection was successful, False otherwise

Return type bool

property pin

//raspberrypi.stackexchange.com/a/12967>`_ GPIO pin.

When assigned, also updates *pin_bcm* with the BCM-numbered pin.

Type `Board-numbered <https

property state: bool

Instantaneous state of GPIO pin, on (`True`) or off (`False`)

Returns bool

property pull

State of internal pullup/down resistor.

See *PULL_MAP* for possible values.

Returns ‘U’/’D’/None for pulled up, down or not set.

Return type int

property polarity

on=High=1, off=Low=0; if 0: off=Low=0, on=High=1.

When set, updates `on` and `off` accordingly

Type Logic direction. if 1

property trigger

Maps strings ((‘U’,1,True), (‘D’,0,False), (‘B’,[0,1])) to pigpio edge types (RISING_EDGE, FALLING_EDGE, EITHER_EDGE), respectively.

Type dict

release()

Release the connection to the pigpio daemon.

Note: the Hardware metaclass will call this method on object deletion.

class Digital_Out(*pin=None*, *pulse_width=100*, *polarity=1*, ***kwargs*)

Bases: `autopilot.hardware.gpio.GPIO`

TTL/Digital logic out through a GPIO pin.

Parameters

- **pin** (*int*) – The Board-numbered GPIO pin of this object
- **pulse_width** (*int*) – Width of digital output `pulse()` (us). range: 1-100
- **polarity** (*bool*) – Whether 'on' is High (1, default) and pulses bring the voltage High, or vice versa (0)

Variables

- **scripts** (`dict`) – maps script IDs to pigpio script handles
- **pigs_function** (`bytes`) – when using pigpio scripts, what function is used to set the value of the output? (eg. 'w' for digital out, 'gdc' for pwm, more info here: http://abyz.me.uk/rpi/pigpio/pigs.html)
- **script_counter** (`itertools.count`) – generate script IDs if not explicitly given to `series()`. generated IDs are of the form 'series_#'

Attributes:

`output`
`type`
`pigs_function`

Methods:

`set`(value)	Set pin logic level.
`turn`([direction])	Change output state using on/off parlance.
`toggle`()	If pin is High, set Low, and vice versa.
`pulse`([duration])	Send a timed on pulse.
`_series_script`(values[, durations, unit, …])	Create a pigpio script to set a pin to a series of values for a series of durations.
`store_series`(id, **kwargs)	Create, and store a pigpio script for a series of output values to be called by `series()`
`series`([id, delete])	Execute a script that sets the pin to a series of values for a series of durations.
`delete_script`(script_id)	spawn a thread to delete a script with id `script_id`
`delete_all_scripts`()	Stop and delete all scripts
`stop_script`([id])	Stops a running pigpio script
`release`()	Stops and deletes all scripts, sets to `off`, and calls `GPIO.release()`

output = True

type = 'DIGITAL_OUT'

pigs_function = b'w'

set(*value: bool*)

Set pin logic level.

Default uses `pigpio.pi.write()`, but can be overwritten by inheriting classes

Stops the last running script when called.

Parameters **value** (*int, bool*) – (1, True) to set High, (0, False) to set Low.

`turn`(*direction='on'*)
Change output state using on/off parlance. logic direction varies based on `Digital_Out.polarity`

Stops the last running script when called.

Parameters **direction** (*str, bool*) – 'on', 1, or True to turn to `on` and vice versa for `off`

`toggle`()
If pin is High, set Low, and vice versa.

Stops the last running script when called.

`pulse`(*duration=None*)
Send a timed `on` pulse.

Parameters **duration** (*int*) – If None (default), uses `duration`, otherwise duration of pulse from 1-100us.

`_series_script`(*values*, *durations=None*, *unit='ms'*, *repeat=None*, *finish_off=True*)
Create a pigpio script to set a pin to a series of values for a series of durations.

Typically shouldn't be called by itself, is used by `series()` or `store_series()`

For more information on pigpio scripts, see: http://abyz.me.uk/rpi/pigpio/pigs.html#Scripts

Parameters

- **values** (*list*) – A list of tuples of (value, duration) or a list of values in (1,0) to set self.pin_bcm to.
- **durations** (*list*) – If `values` is not a list of tuples, a list of durations. len(durations) must be either == len(values) or else len(durations) == 1, in which case the duration is repeated.
- **unit** (*"ms", "us"*) – units of durations in milliseconds or microseconds
- **repeat** (*int*) – If the script should be repeated, how many times? A value of 2 results in the script being run 2 times total, not 2 *additional* times (or, 3 total times)
- **finish_off** (*bool*) – If true, the script ends by turning the pin to `off`

Returns the constructed script string

Return type (str)

`store_series`(*id*, ***kwargs*)
Create, and store a pigpio script for a series of output values to be called by `series()`

Parameters

- **id** (*str*) – shorthand key used to call this series with `series()`
- **kwargs** – passed to `_series_script()`

`series`(*id=None*, *delete=None*, ***kwargs*)
Execute a script that sets the pin to a series of values for a series of durations.

See `_series_script()` for series parameterization.

Ideally one would use `store_series()` and use the returned id to call this function. Otherwise, this method calls `store_series()` and runs it.

Parameters

- **id** (*str, int*) – ID of the script, if not already created, created with `store_script()`. If None (default), an ID is generated with `script_counter` of the form `'script_#'`

- **kwargs** – passed to `_series_script()`

`delete_script(script_id)`
spawn a thread to delete a script with id `script_id`

This is a 'soft' deletion – it checks if the script is running, and waits for up to 10 seconds before actually deleting it.

The script is deleted from the pigpio daemon, from `script_handles` and from `scripts`

Parameters **script_id** (*str*) – a script ID in `Digital_Out.script_handles`

`delete_all_scripts()`
Stop and delete all scripts

This is a "hard" deletion – the script will be immediately stopped if it's running.

`stop_script(id=None)`
Stops a running pigpio script

Parameters **id** (*str, none*) – If None, stops the last run script. if str, stops script with that id.

`release()`
Stops and deletes all scripts, sets to `off`, and calls `GPIO.release()`

`class Digital_In(pin, event=None, record=True, **kwargs)`
Bases: `autopilot.hardware.gpio.GPIO`

Record digital input and call one or more callbacks on logic transition.

Parameters

- **pin** (*int*) – Board-numbered GPIO pin.
- **event** (`threading.Event`) – For callbacks assigned with `assign_cb()` with `evented = True`, set this event whenever the callback is triggered. Can be used to handle stage transition logic here instead of the `Task` object, as is typical.
- **record** (*bool*) – Whether all logic transitions should be recorded as a list of ('EVENT', 'Timestamp') tuples.
- ****kwargs** – passed to `GPIO`

Sets the internal pullup/down resistor to `Digital_In.off` and `Digital_In.trigger` to `Digital_In.on` upon instantiation.

Note: pull and trigger are set by polarity on initialization in digital inputs, unlike other GPIO classes. They are not mutually synchronized however, ie. after initialization if any one of these attributes are changed, the other two will remain the same.

Variables

- **pig** (`pigpio.pi()`) – The pigpio connection.
- **pin** (`int`) – Broadcom-numbered pin, converted from the argument given on instantiation
- **callbacks** (`list`) – A list of :meth:`pigpio.callback`s kept to clear them on exit
- **polarity** (`int`) – Logic direction, if 1: off=0, on=1, pull=low, trigger=high and vice versa for 0
- **events** (`list`) – if `record` is True, a list of ('EVENT', 'TIMESTAMP') tuples

Attributes:

is_trigger	
type	
input	

Methods:

assign_cb(callback_fn[, add, evented, . . .])	Sets `callback_fn` to be called when `Digital_In.trigger` is detected.
clear_cb()	Tries to call *.cancel()* on each of the callbacks in `callbacks`
record_event(pin, level, timestamp)	On either direction of logic transition, record the time
release()	Clears any callbacks and calls *GPIO.release()*

is_trigger = True

type = 'DIGI_IN'

input = True

assign_cb(*callback_fn*, *add=True*, *evented=False*, *manual_trigger=None*)

Sets `callback_fn` to be called when `Digital_In.trigger` is detected.

`callback_fn` must accept three parameters:

- GPIO (int, 0-31): the BCM number of the pin that was triggered
- level (0-2):
  - 0: change to low (falling)
  - 1: change to high (rising)
  - 2: no change (watchdog timeout)
- timestamp (str): If using the Autopilot version of pigpio, an isoformatted timestamp

Parameters

- **callback_fn** (*callable*) – The function to be called when triggered
- **add** (*bool*) – Are we adding another callback? If False, the previous callbacks are cleared.
- **evented** (*bool*) – Should triggering this event also set the internal `event`? Note that `Digital_In.event` must have been passed.
- **manual_trigger** (*'U', 'D', 'B'*) – Override `Digital_In.trigger` if needed.

clear_cb()

Tries to call *.cancel()* on each of the callbacks in `callbacks`

record_event(*pin*, *level*, *timestamp*)

On either direction of logic transition, record the time

Parameters

- **pin** (*int*) – BCM numbered pin passed from pigpio

- **level** (*bool*) – High/Low status of current pin
- **timestamp** (*str*) – isoformatted timestamp

release()
Clears any callbacks and calls *GPIO.release()*

class PWM(*pin*, *range=255*, ***kwargs*)
Bases: *autopilot.hardware.gpio.Digital_Out*

PWM output from GPIO.

Parameters

- **pin** (*int*) – Board numbered GPIO pin
- **range** (*int*) – Maximum value of PWM duty-cycle. Default 255.
- ****kwargs** – passed to *Digital_Out*

Attributes:

output	
type	
pigs_function	
range	Maximum value of PWM dutycycle.
polarity	Logic direction.

Methods:

set(value)	Sets PWM duty cycle normalized to *polarity* and transformed by `_clean_value()`
release()	Turn off and call *Digital_Out.release()*

output = True

type = 'PWM'

pigs_function = b'pwm'

set(*value*)
Sets PWM duty cycle normalized to *polarity* and transformed by `_clean_value()`

Stops the last running script

Parameters value (*int, float*) –

- if int > 1, sets value (or *PWM.range*-value if *PWM.polarity* is inverted).
- if 0 <= float <= 1, transforms to a proportion of *range* (inverted if needed as well).

property range
Maximum value of PWM dutycycle.

Doesn't set duration of PWM, but set values will be divided by this range. eg. if `range == 200`, calling `PWM.set(100)()` would result in a 50% duty cycle

Parameters (int) – 25-40000

property polarity
Logic direction.

- if 1: on=High=:attr:~*PWM.range*, off=Low=0;
- if 0: off=Low=0, on=High=:attr:~*PWM.range*.

When set, updates `on` and `off`

release()
Turn off and call *Digital_Out.release()*

Returns:

class LED_RGB(*pins=None*, *r=None*, *g=None*, *b=None*, *polarity=1*, *blink=True*, ***kwargs*)
Bases: *autopilot.hardware.gpio.Digital_Out*

An RGB LED, wrapper around three *PWM* objects.

Parameters

- **pins** (*list*) – A list of (board) pin numbers. Either *pins* OR all *r*, *g*, *b* must be passed.
- **r** (*int*) – Board number of Red pin - must be passed with *g* and *b*
- **g** (*int*) – Board number of Green pin - must be passed with *r* and *b*
- **b** (*int*) – Board number of Blue pin - must be passed with *r* and *g*:
- **polarity** (*0, 1*) – 0: common anode (low turns LED on) 1: common cathode (low turns LED off)
- **blink** (*bool*) – Flash RGB at the end of init to show we're alive and bc it's real cute.
- ****kwargs** – passed to *Digital_Out*

Variables channels (*dict*) – The three PWM objects, {'r':PWM, ... etc}

Attributes:

output	
type	
range	Returns: dict: ranges for each of the `LED_RGB.channels`
pin	Dict of the board pin number of each channel, ``{'r' : self.channels['r'].pin, .
pin_bcm	Dict of the broadcom pin number of each channel, ``{'r' : self.channels['r'].pin_bcm, .
pull	State of internal pullup/down resistor.

Methods:

set([value, r, g, b])	Set the color of the LED.
toggle()	If pin is High, set Low, and vice versa.
pulse([duration])	Send a timed on pulse.
_series_script(colors[, durations, unit, ...])	Create a script to flash a series of colors.
flash(duration[, frequency, colors])	Specify a color series by total duration and flash frequency.

continues on next page

Table 30 – continued from previous page

release()	Release each channel and stop pig without calling superclass.

output = True

type = 'LEDS'

property range: dict
Returns: dict: ranges for each of the `LED_RGB.channels`

set(*value=None*, *r=None*, *g=None*, *b=None*)
Set the color of the LED.

Can either pass

- a full (R, G, B) tuple to `value`,
- a single `value` that is applied to each channel,
- if `value` is not passed, individual `r`, `g`, or `b` values can be passed (any combination can be set in a single call)

Stops the last run script

Parameters

- **value** (*int, float, tuple, list*) – If list or tuple, an (R, G, B) color. If float or int, applied to each color channe. Can be set with floats 0-1, or ints >= 1 (See `PWM.range`). If None, use `r`, `g`, and `b`.
- **r** (*float, int*) – value to set red channel
- **g** (*float, int*) – value to set green channel
- **b** (*float, int*) – value to set blue channel

toggle()
If pin is High, set Low, and vice versa.

Stops the last running script when called.

pulse(*duration=None*)
Send a timed on pulse.

Parameters **duration** (*int*) – If None (default), uses `duration`, otherwise duration of pulse from 1-100us.

_series_script(*colors*, *durations=None*, *unit='ms'*, *repeat=None*, *finish_off=True*)
Create a script to flash a series of colors.

Like `Digital_Out._series_script()`, but sets all pins at once.

Parameters

- **colors** (*list*) – a list of (R, G, B) colors, or a list of ((R,G,B),duration) tuples.
- **durations** (*int, list*) – Duration of each color. if a single value, used for all `colors`. if a list, `len(durations) == len(colors)`. If None, colors must be `((R,G,B),duration)` tuples.
- **unit** (*'ms', 'us'*) – unit of durations, milliseconds or microseconds
- **repeat** (*int*) – Number of repetitions. If None, script runs once.
- **finish_off** (*bool*) – Whether the channels should be set to `off` when the script completes

Returns constructed pigpio script string.

Return type str

`flash`(*duration*, *frequency=10*, *colors=((1, 1, 1), (0, 0, 0))*)
Specify a color series by total duration and flash frequency.

Largely a convenience function for on/off flashes.

Parameters

- **duration** (*int, float*) – Duration of flash in ms.
- **frequency** (*int, float*) – Frequency of flashes in Hz
- **colors** (*list*) –

  A list of RGB values 0-255 like:

  ```
  [[255,255,255],[0,0,0]]
  ```

`release`()
Release each channel and stop pig without calling superclass.

property pin
Dict of the board pin number of each channel, `{'r' : self.channels['r'].pin, ... }`

property pin_bcm
Dict of the broadcom pin number of each channel, `{'r' : self.channels['r'].pin_bcm, ... }`

property pull
State of internal pullup/down resistor.

See *PULL_MAP* for possible values.

Returns 'U'/'D'/None for pulled up, down or not set.

Return type int

class Solenoid(*pin*, *polarity=1*, *duration=20*, *vol=None*, ***kwargs*)
Bases: `autopilot.hardware.gpio.Digital_Out`

Solenoid valve for water delivery.

Parameters

- **pin** (*int*) – Board pin number, converted to BCM on init.
- **polarity** (*0, 1*) – Whether HIGH opens the port (1) or closes it (0)
- **duration** (*int, float*) – duration of open, ms.
- **vol** (*int, float*) – desired volume of reward in uL, must have computed calibration results, see `calibrate_ports()`
- ****kwargs** – passed to `Digital_Out`

Only NC solenoids should be used, as there is no way to guarantee that a pin will maintain its voltage when it is released, and you will spill water all over the place.

Variables

- **calibration** (*dict*) – Dict with with line coefficients fitting volume to open duration, see `calibrate_ports()`. Retrieved from prefs, specifically `prefs.get('PORT_CALIBRATION')[name]`

- **mode** (*'DURATION', 'VOLUME'*) – Whether open duration is given in ms, or computed from calibration
- **duration** (*int, float*) – Duration of valve opening, in ms. When set, creates a script 'open' that is used to open the valve for a precise amount of time

Attributes:

output	
type	
DURATION_MIN	Minimum allowed duration in ms
duration	

Methods:

dur_from_vol(vol)	Given a desired volume, compute an open duration.
open([duration])	Open the valve.

output = True

type = 'SOLENOID'

DURATION_MIN = 2

Minimum allowed duration in ms

property duration

dur_from_vol(*vol*)

Given a desired volume, compute an open duration.

Must have calibration available in prefs, see `calibrate_ports()`.

Parameters **vol** (*float, int*) – desired reward volume in uL

Returns computed opening duration for given volume

Return type int

open(*duration=None*)

Open the valve.

Uses the 'open' script created when assigning duration.

Parameters **duration** (*float*) – If provided, open for this duration instead of the duration stored on instantiation.

12.3 i2c

Classes:

I2C_9DOF(accel, gyro, mag, gyro_hpf[, ...])	A Sparkfun 9DOF combined accelerometer, magnetometer, and gyroscope.
MLX90640([fps, integrate_frames, interpolate])	A MLX90640 Temperature sensor.

class I2C_9DOF(*accel: bool = True, gyro: bool = True, mag: bool = True, gyro_hpf: float = 0.2, accel_range=16, kalman_mode: str = 'both', invert_gyro=False, *args, **kwargs*)

Bases: `autopilot.hardware.Hardware`

A Sparkfun 9DOF combined accelerometer, magnetometer, and gyroscope.

Sensor Datasheet: https://cdn.sparkfun.com/assets/learn_tutorials/3/7/3/LSM9DS1_Datasheet.pdf

Hardware Datasheet: https://github.com/sparkfun/9DOF_Sensor_Stick

Documentation on calculating position values: https://arxiv.org/pdf/1704.06053.pdf

This device uses I2C, so must be connected accordingly:

- VCC: 3.3V (pin 2)
- Ground: (any ground pin
- SDA: I2C.1 SDA (pin 3)
- SCL: I2C.1 SCL (pin 5)

This class uses code from the Adafruit Circuitfun library, modified to use pigpio

Note: use this for processing?? https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6111698/

Parameters

- **accel** (*bool*) – Whether the accelerometer should be made active (default: True)
- **gyro** (*bool*) – Whether the gyroscope should be made active (default: True) – accel must be true if gyro is true
- **mag** (*bool*) – Whether the magnetomete should be made active (default: True)
- **gyro_hpf** (*int, float*) – Highpass filter cutoff for onboard gyroscope filter. One of *GYRO_HPF_CUTOFF* (default: 4), or `False` to disable
- **kalman_mode** (*'both', 'accel', None*) – Whether to use a kalman filter that integrates accelerometer and gyro readings ('both', default), a kalman filter with just the accelerometer values ('accel'), or just return the raw calculated orientation values from `rotation`
- **invert_gyro** (*list, tuple*)

Attributes:

ACCELRANGE_2G
ACCELRANGE_16G

continues on next page

Table 34 – continued from previous page

ACCELRANGE_4G	
ACCELRANGE_8G	
MAGGAIN_4GAUSS	
MAGGAIN_8GAUSS	
MAGGAIN_12GAUSS	
MAGGAIN_16GAUSS	
GYROSCALE_245DPS	
GYROSCALE_500DPS	
GYROSCALE_2000DPS	
GYRO_HPF_CUTOFF	Highpass-filter cutoff frequencies (keys, in Hz) mapped to binary flag.
`accel_range`	The accelerometer range.
`mag_gain`	The magnetometer gain.
`gyro_scale`	The gyroscope scale.
`gyro_filter`	Set the high-pass filter for the gyroscope.
`gyro_polarity`	
`acceleration`	The calibrated x, y, z acceleration in m/s^2
`magnetic`	The magnetometer X, Y, Z axis values as a 3-tuple of gauss values.
`gyro`	The gyroscope X, Y, Z axis values as a 3-tuple of degrees/second values.
`rotation`	Return roll (rotation around x axis) and pitch (rotation around y axis) computed from the accelerometer
`temperature`	Returns: float: Temperature in Degrees C

Methods:

`calibrate`([what, samples, sample_dur])	Calibrate sensor readings to correct for bias and scale errors

ACCELRANGE_2G = 0

ACCELRANGE_16G = 8

ACCELRANGE_4G = 16

ACCELRANGE_8G = 24

MAGGAIN_4GAUSS = 0

MAGGAIN_8GAUSS = 32

MAGGAIN_12GAUSS = 64

MAGGAIN_16GAUSS = 96

`GYROSCALE_245DPS = 0`

`GYROSCALE_500DPS = 8`

`GYROSCALE_2000DPS = 24`

`GYRO_HPF_CUTOFF = {0.1: 9, 0.2: 8, 0.5: 7, 1: 6, 2: 5, 4: 4, 8: 3, 15: 2, 30: 1, 57: 0}`

Highpass-filter cutoff frequencies (keys, in Hz) mapped to binary flag.

> **Note:** the frequency of a given binary flag is dependent on the output frequency (952Hz by default, changing frequency is not currently exposed in this object). See Table 52 of the sensor datasheet for more.

property accel_range

The accelerometer range. Must be one of: - *I2C_9DOF.ACCELRANGE_2G* - *I2C_9DOF.ACCELRANGE_4G* - *I2C_9DOF.ACCELRANGE_8G* - *I2C_9DOF.ACCELRANGE_16G*

property mag_gain

The magnetometer gain. Must be a value of: - *I2C_9DOF.MAGGAIN_4GAUSS* - *I2C_9DOF.MAGGAIN_8GAUSS* - *I2C_9DOF.MAGGAIN_12GAUSS* - *I2C_9DOF.MAGGAIN_16GAUSS*

property gyro_scale

The gyroscope scale. Must be a value of: - *I2C_9DOF.GYROSCALE_245DPS* - *I2C_9DOF.GYROSCALE_500DPS* - *I2C_9DOF.GYROSCALE_2000DPS*

property gyro_filter: Union[int, float, bool]

Set the high-pass filter for the gyroscope.

> **Note:** the frequency of a given binary flag is dependent on the output frequency (952Hz by default, changing frequency is not currently exposed in this object). See Table 52 of the sensor datasheet for more.

Parameters **gyro_filter** (*int, float, False*) – Filter frequency (in *GYRO_HPF_CUTOFF*) or False to disable

Returns current HPF cutoff or `False` if disabled

Return type float, bool

property gyro_polarity

property acceleration

The calibrated x, y, z acceleration in m/s^2

Returns x, y, z acceleration

Return type accel (tuple)

property magnetic

The magnetometer X, Y, Z axis values as a 3-tuple of gauss values.

Returns x, y, z gauss values

Return type (tuple)

property gyro

The gyroscope X, Y, Z axis values as a 3-tuple of degrees/second values.

property rotation

Return roll (rotation around x axis) and pitch (rotation around y axis) computed from the accelerometer

Uses `transform.geometry.IMU_Orientation` to fuse accelerometer and gyroscope with Kalman filter

Returns np.ndarray - [roll, pitch]

property temperature

Returns: float: Temperature in Degrees C

calibrate(*what: str = 'accelerometer', samples: int = 10000, sample_dur: Optional[float] = None*) → dict

Calibrate sensor readings to correct for bias and scale errors

Note: Currently only calibrating the accelerometer is implemented.

The accelerometer is calibrated by rotating the sensor slowly in all three rotational dimensions in such a way that minimizes linear acceleration (not due to gravity). A perfect sensor would output a sphere of points centered at 0

Parameters

- **what** (*str*) – which sensor is to be calibrated (currentlty only "accelerometer" implemented)
- **samples** (*int*) – number of samples that should be used to compute the calibration
- **sample_dur** (*float*) – number of seconds to sample for, overrides `samples` if not None (default)

Returns calibration dictionary (also saved to disk using `Hardware.calibration`)

Return type dict

class MLX90640(*fps=64, integrate_frames=64, interpolate=3, **kwargs*)

Bases: `autopilot.hardware.cameras.Camera`

A MLX90640 Temperature sensor.

Parameters

- **fps** (*int*) – Acquisition framerate, must be one of `MLX90640.ALLOWED_FPS`
- **integrate_frames** (*int*) – Number of frames to average over
- **interpolate** (*int*) – Interpolation multiplier – 3 "increases the resolution" 3x
- ****kwargs** – passed to `Camera`

Variables

- **shape** (`tuple`) – :attr:`~MLX90640.SHAPE_SENSOR
- **integrate_frames** (`int`) – Number of frames to average over
- **interpolate** (`int`) – Interpolation multiplier – 3 "increases the resolution" 3x
- **_grab_event** (`threading.Event`) – capture thread sets every time it gets a frame, _grab waits every time, keeps us from returning same frame twice

This device uses I2C, so must be connected accordingly:

- VCC: 3.3V (pin 2)
- Ground: (any ground pin
- SDA: I2C.1 SDA (pin 3)
- SCL: I2C.1 SCL (pin 5)

Uses a modified version of the MLX90640 Library that is capable of outputting 64fps. You must install the library separately, see the `setup_mlx90640.sh` script.

Capture works a bit differently from other Cameras – the *capture_init()* method spawns a *_threaded_capture()* thread, which continually puts frames in the `_frames` array which serves as a ring buffer. The *_grab()* method then awaits the `_grab_event` to be set by the capture thread, and when it is set returns the mean across frames of the ring buffer.

Note: The setup script modifies the systemwide i2c baudrate to 1MHz, which may interfere with other I2C devices. It can be returned to 400kHz (default) by editing `/config/boot.txt` to read `dtparam=i2c_arm_baudrate=400000`

Attributes:

type	what are we anyway?
ALLOWED_FPS	FPS must be one of these
SHAPE_SENSOR	(H, W) Output shape of this sensor is always the same.
fps	
integrate_frames	
interpolate	

Methods:

init_cam()	Set the camera object to use our *MLX90640.fps*
capture_init()	Spawn a *_threaded_capture()* thread
_threaded_capture()	Continually capture frames into the `_frames` ring buffer
_grab()	Await the `_grab_event` and then average over the frames stored in `_frames`
_timestamp([frame])	Just gets Python timestamps for now. . .
interpolate_frame(frame)	Interpolate frame according to *interpolate* using `scipy.interpolate.griddata()`
release()	Stops the capture thread, cleans up the camera, and calls the superclass release method.

type = 'MLX90640'

what are we anyway?

Type (str)

ALLOWED_FPS = (1, 2, 4, 8, 16, 32, 64)

FPS must be one of these

SHAPE_SENSOR = (32, 24)

(H, W) Output shape of this sensor is always the same. May differ from `MLX90640.shape` if interpolate >1

property fps

property integrate_frames

property interpolate

init_cam()
Set the camera object to use our *MLX90640.fps*

capture_init()
Spawn a *_threaded_capture()* thread

_threaded_capture()
Continually capture frames into the `_frames` ring buffer

Stops when `stopping` is set.

_grab()
Await the `_grab_event` and then average over the frames stored in `_frames`

Returns (`ndarray`) Averaged and interpolated frame

_timestamp(*frame=None*)
Just gets Python timestamps for now...

Returns Isoformatted timestamp from datetime

Return type str

interpolate_frame(*frame*)
Interpolate frame according to *interpolate* using `scipy.interpolate.griddata()`

Parameters **frame** (`numpy.ndarray`) – Frame to interpolate

Returns Interpolated Frame

Return type (`numpy.ndarray`)

release()
Stops the capture thread, cleans up the camera, and calls the superclass release method.

12.4 usb

Hardware that uses USB

Classes:

Wheel([mouse_idx, fs, thresh, thresh_type, ...])	A continuously measured mouse wheel.
Scale([model, vendor_id, product_id])	

class Wheel(*mouse_idx=0, fs=10, thresh=100, thresh_type='dist', start=True, digi_out=False, mode='vel_total', integrate_dur=5*)
Bases: *autopilot.hardware.Hardware*

A continuously measured mouse wheel.

Uses a USB computer mouse.

> **Warning:** 'vel' thresh_type not implemented

Parameters

- **mouse_idx** (*int*)
- **fs** (*int*)
- **thresh** (*int*)
- **thresh_type** (*'dist'*)
- **start** (*bool*)
- **digi_out** (*Digital_Out*, bool)
- **mode** (*'vel_total'*)
- **integrate_dur** (*int*)

Attributes:

input
type
trigger
THRESH_TYPES
MODES
MOVE_DTYPE

Methods:

start()	
check_thresh(move)	Updates thresh_val and checks whether it's above/below threshold
calc_move(move[, thresh_type])	Calculate distance move depending on type (x, y, total dist)
thresh_trig()	
assign_cb(trigger_fn)	Every hardware device that is a `trigger` must redefine this to accept a function (typically *Task.handle_trigger()*) that is called when that trigger is activated.
l_measure(value)	Task has signaled that we need to start measuring movements for a trigger
l_clear(value)	Stop measuring!
l_stop(value)	Stop measuring and clear system resources :Parameters: **value** ()
release()	Every hardware device needs to redefine *release()*, and must

input = True

type = 'Wheel'

`trigger = False`

`THRESH_TYPES = ['dist', 'x', 'y', 'vel']`

`MODES = ('vel_total', 'steady', 'dist', 'timed')`

`MOVE_DTYPE = [('vel', 'i4'), ('dir', 'U5'), ('timestamp', 'f8')]`

`start()`

`check_thresh(`*move*`)`

Updates thresh_val and checks whether it's above/below threshold

Parameters **move** (*np.array*) – Structured array with fields ('vel', 'dir', 'timestamp')

Returns:

`calc_move(`*move*, *thresh_type=None*`)`

Calculate distance move depending on type (x, y, total dist)

Parameters

- **move** ()
- **thresh_type** ()

Returns:

`thresh_trig()`

`assign_cb(`*trigger_fn*`)`

Every hardware device that is a `trigger` must redefine this to accept a function (typically `Task.handle_trigger()`) that is called when that trigger is activated.

When not redefined, a warning is given.

`l_measure(`*value*`)`

Task has signaled that we need to start measuring movements for a trigger

Parameters **value** ()

`l_clear(`*value*`)`

Stop measuring!

Parameters **value** ()

Returns:

`l_stop(`*value*`)`

Stop measuring and clear system resources :Parameters: **value** ()

Returns:

`release()`

Every hardware device needs to redefine *release()*, and must

- Safely unload any system resources used by the object, and
- Return the object to a neutral state - eg. LEDs turn off.

When not redefined, a warning is given.

`class Scale(`*model='stamps.com'*, *vendor_id=None*, *product_id=None*`)`

Bases: `autopilot.hardware.Hardware`

Note: Not implemented, working on using a digital scale to make weighing faster.

Parameters

- **model**
- **vendor_id**
- **product_id**

Attributes:

MODEL

```
MODEL = {'stamps.com': {'product_id': 27251, 'vendor_id': 5190}}
```

CHAPTER

THIRTEEN

NETWORKING

Classes for network communication.

There are two general types of network objects -

- **autopilot.networking.Station and its children are independent processes that should only be instantiated once** per piece of hardware. They are used to distribute messages between Net_Node s, forward messages up the networking tree, and responding to messages that don't need any input from the Pilot or Terminal.
- **Net_Node is a pop-in networking class that can be given to any other object that** wants to send or receive messages.

The Message object is used to serialize and pass messages. When sent, messages are JSON serialized (with some special magic to compress/encode numpy arrays) and sent as zmq multipart messages.

Each serialized message, when sent, can have n frames of the format:

```
[hop_0, hop_1, ... hop_n, final_recipient, serialized_message]
```

Or, messages can have multiple "hops" (a typical message will have one 'hop' specified by the to field), the second to last frame is always the final intended recipient, and the final frame is the serialized message. Note that the to field of a Message object will always be the final recipient even if a list is passed for to when sending. This lets Station objects efficiently forward messages without deserializing them at every hop.

Functions:

serialize_array(array)	Pack an array with blosc.pack_array() and serialize with base64.b64encode()

serialize_array(*array*)

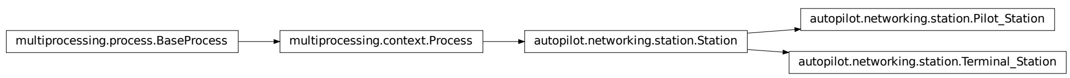
Pack an array with blosc.pack_array() and serialize with base64.b64encode()

Parameters **array** (numpy.ndarray) – Array to serialize

Returns {'NUMPY_ARRAY': base-64 encoded, blosc-compressed array.}

Return type dict

13.1 station



Classes:

Station(id, push_ip, push_port, push_id, ...)	Independent networking class used for messaging between computers.
Terminal_Station(pilots)	`Station` object used by *Terminal* objects.
Pilot_Station()	`Station` object used by *Pilot* objects.

class Station(*id: Optional[str] = None*, *push_ip: Optional[str] = None*, *push_port: Optional[int] = None*, *push_id: Optional[str] = None*, *pusher: bool = False*, *listen_port: Optional[int] = None*, *listens: Optional[Dict[str, Callable]] = None*)

Bases: `multiprocessing.context.Process`

Independent networking class used for messaging between computers.

These objects send and handle `networking.Message` s by using a dictionary of `listens`, or methods that are called to respond to different types of messages.

Each sent message is given an ID, and a thread is spawned to periodically resend it (up until some time-to-live, typically 5 times) until confirmation is received.

By default, the only listen these objects have is `l_confirm()`, which responds to message confirmations. Accordingly, *listens* should be added by using `dict.update()` rather than reassigning the attribute.

Station objects can be made with or without a `pusher`, a `zmq.DEALER` socket that connects to the `zmq.ROUTER` socket of an upstream Station object.

This class can be instantiated on its own if all of the required arguments are supplied, but the intended pattern of use is to subclass it with any custom `listen` methods for handling message types and other logic that would be specific for an agent type that uses it.

Note: This object will likely be deprecated in v0.5.0, as the gains of a separate messaging process are not as great as the complications caused by having two different kinds of networking object in the system. In the future we will move to having a single type of networking object that can either be spawned as a separate process or as a thread.

Args are similar to the documented Attributes, and so only those that differ from attributes are documented here

Parameters pusher (*bool*) – If `True`, create a `zmq.DEALER` socket connected to `push_ip`, `push_port`, and `push_id`. (Default: `False`).

Variables

- **context** (`zmq.Context`) – zeromq context
- **loop** (`tornado.ioloop.IOLoop`) – a tornado ioloop
- **pusher** (`zmq.Socket`) – pusher socket - a dealer socket that connects to other routers

- **push_ip** (*str*) – If we have a dealer, IP to push messages to
- **push_port** (*str*) – If we have a dealer, port to push messages to
- **push_id** (*str*) – `identity` of the Router we push to
- **listener** (`zmq.Socket`) – The main router socket to send/recv messages
- **listen_port** (*str*) – Port our router listens on
- **logger** (`logging.Logger`) – Used to log messages and network events.
- **id** (*str*) – What are we known as? What do we set our `identity` as?
- **ip** (*str*) – Device IP
- **listens** (*dict*) – Dictionary of functions to call for different types of messages. keys match the `Message.key`.
- **senders** (*dict*) – Identities of other sockets (keys, ie. directly connected) and their state (values) if they keep one
- **push_outbox** (*dict*) – Messages that have been sent but have not been confirmed to our `Station.pusher`
- **send_outbox** (*dict*) – Messages that have been sent but have not been confirmed to our `Station.listener`
- **timers** (*dict*) – dict of `threading.Timer` s that will check in on outbox messages
- **msg_counter** (`itertools.count`) – counter to index our sent messages
- **file_block** (`threading.Event`) – Event to signal when a file is being received.

Attributes:

repeat_interval	

Methods:

run()	A `zmq.Context` and `tornado.IOLoop` are spawned, the listener and optionally the pusher are instantiated and connected to `handle_listen()` using `on_recv()` .
prepare_message(to, key, value[, repeat, flags])	If a message originates with us, a *Message* class is instantiated, given an ID and the rest of its attributes.
send([to, key, value, msg, repeat, flags])	Send a message via our `listener` , ROUTER socket.
push([to, key, value, msg, repeat, flags])	Send a message via our `pusher` , DEALER socket.
repeat()	Periodically (according to *repeat_interval*) resend messages that haven't been confirmed
l_confirm(msg)	Confirm that a message was received.
l_stream(msg)	Reconstitute the original stream of messages and call their handling methods
handle_listen(msg)	Upon receiving a message, call the appropriate listen method in a new thread.
get_ip()	Find our IP address
release()	

continues on next page

Table 4 – continued from previous page

_check_stop()	periodic callback called by the IOLoop to check if the *closing* flag has been set, and closing process if so

repeat_interval = 5.0

run()

A zmq.Context and tornado.IOLoop are spawned, the listener and optionally the pusher are instantiated and connected to *handle_listen()* using on_recv() .

The process is kept open by the tornado.IOLoop .

prepare_message(*to*, *key*, *value*, *repeat=True*, *flags=None*)

If a message originates with us, a *Message* class is instantiated, given an ID and the rest of its attributes.

Parameters

- **flags**
- **repeat**
- **to** (*str*) – The identity of the socket this message is to
- **key** (*str*) – The type of message - used to select which method the receiver uses to process this message.
- **value** – Any information this message should contain. Can be any type, but must be JSON serializable.

send(*to=None*, *key=None*, *value=None*, *msg=None*, *repeat=True*, *flags=None*)

Send a message via our listener , ROUTER socket.

Either an already created *Message* should be passed as *msg*, or at least *to* and *key* must be provided for a new message created by *prepare_message()* .

A threading.Timer is created to resend the message using *repeat()* unless *repeat* is False.

Parameters

- **flags**
- **to** (*str*) – The identity of the socket this message is to
- **key** (*str*) – The type of message - used to select which method the receiver uses to process this message.
- **value** – Any information this message should contain. Can be any type, but must be JSON serializable.
- **msg** (*.Message*) – An already created message.
- **repeat** (*bool*) – Should this message be resent if confirmation is not received?

push(*to=None*, *key=None*, *value=None*, *msg=None*, *repeat=True*, *flags=None*)

Send a message via our pusher , DEALER socket.

Unlike *send()* , *to* is not required. Every message is always sent to push_id . *to* can be included to send a message further up the network tree to a networking object we're not directly connected to.

Either an already created *Message* should be passed as *msg*, or at least *key* must be provided for a new message created by *prepare_message()* .

A threading.Timer is created to resend the message using *repeat()* unless *repeat* is False.

Parameters

- **flags**
- **to** (*str*) – The identity of the socket this message is to. If not included, sent to `push_id()` .
- **key** (*str*) – The type of message - used to select which method the receiver uses to process this message.
- **value** – Any information this message should contain. Can be any type, but must be JSON serializable.
- **msg** (*.Message*) – An already created message.
- **repeat** (*bool*) – Should this message be resent if confirmation is not received?

repeat()
Periodically (according to `repeat_interval`) resend messages that haven't been confirmed

TTL is decremented, and messages are resent until their TTL is 0.

l_confirm(*msg*)
Confirm that a message was received.

Parameters msg (`Message`) – A confirmation message - note that this message has its own unique ID, so the value of this message contains the ID of the message that is being confirmed

l_stream(*msg*)
Reconstitute the original stream of messages and call their handling methods

The `msg` should contain an `inner_key` that indicates the key, and thus the handling method.

Parameters msg (*dict*) – Compressed stream sent by `Net_Node._stream()`

handle_listen(*msg: List[bytes]*)
Upon receiving a message, call the appropriate listen method in a new thread.

If the message is `to` us, send confirmation.

If the message is not `to` us, attempt to forward it.

Parameters msg (*str*) – JSON `Message.serialize()` d message.

get_ip()
Find our IP address

returns (str): our IPv4 address.

release()

_check_stop()
periodic callback called by the IOLoop to check if the *closing* flag has been set, and closing process if so

class **Terminal_Station**(*pilots*)
Bases: `autopilot.networking.station.Station`

`Station` object used by `Terminal` objects.

Spawned without a `pusher`.

Listens

Key	Method	Description
'PING'	*l_ping()*	We are asked to confirm that we are alive
'INIT'	*l_init()*	Ask all pilots to confirm that they are alive
'CHANGE'	*l_change()*	Change a parameter on the Pi
'STOPALL'	*l_stopall()*	Stop all pilots and plots
'KILL'	*l_kill()*	Terminal wants us to die :(
'DATA'	*l_data()*	Stash incoming data from a Pilot
'STATE'	*l_state()*	A Pilot has changed state
'HANDSHAKE'	*l_handshake()*	A Pi is telling us it's alive and its IP
'FILE'	*l_file()*	The pi needs some file from us

Parameters pilots (*dict*) – The `Terminal.pilots` dictionary.

Attributes:

plot_timer
sent_plot

Methods:

start_plot_timer()	Start a timer that controls how often streamed video frames are sent to `gui.Video` plots.
l_ping(msg)	We are asked to confirm that we are alive
l_init(msg)	Ask all pilots to confirm that they are alive
l_change(msg)	Change a parameter on the Pi
l_stopall(msg)	Stop all pilots and plots
l_kill(msg)	Terminal wants us to die :(
l_data(msg)	Stash incoming data from a Pilot
l_continuous(msg)	Handle the storage of continuous data
l_state(msg)	A Pilot has changed state.
l_handshake(msg)	A Pi is telling us it's alive and its IP.
l_file(msg)	A Pilot needs some file from us.

plot_timer = None

sent_plot = {}

start_plot_timer()

Start a timer that controls how often streamed video frames are sent to `gui.Video` plots.

l_ping(*msg: autopilot.networking.message.Message*)

We are asked to confirm that we are alive

Respond with a blank 'STATE' message.

Parameters msg (*Message*)

l_init(*msg: autopilot.networking.message.Message*)

Ask all pilots to confirm that they are alive

Sends a "PING" to everyone in the pilots dictionary.

Parameters msg (*Message*)

l_change(*msg: autopilot.networking.message.Message*)
Change a parameter on the Pi

> **Warning:** Not Implemented

Parameters **msg** (*Message*)

l_stopall(*msg: autopilot.networking.message.Message*)
Stop all pilots and plots

Parameters **msg** (*Message*)

l_kill(*msg: autopilot.networking.message.Message*)
Terminal wants us to die :(

Stop the `Station.loop`

Parameters **msg** (*Message*)

l_data(*msg: autopilot.networking.message.Message*)
Stash incoming data from a Pilot

Just forward this along to the internal terminal object ('_T') and a copy to the relevant plot.

Parameters **msg** (*Message*)

l_continuous(*msg: autopilot.networking.message.Message*)
Handle the storage of continuous data

Forwards all data on to the Terminal's internal `Net_Node`, send to *Plot* according to update rate in `prefs.get('DRAWFPS')`

Parameters **msg** (*Message*) – A continuous data message

l_state(*msg: autopilot.networking.message.Message*)
A Pilot has changed state.

Stash in 'state' field of pilot dict and send along to _T

Parameters **msg** (*Message*)

l_handshake(*msg: autopilot.networking.message.Message*)
A Pi is telling us it's alive and its IP.

Send along to _T

Parameters **msg** (*Message*)

l_file(*msg: autopilot.networking.message.Message*)
A Pilot needs some file from us.

Send it back after `base64.b64encode()` ing it.

> **Todo:** Split large files into multiple messages…

Parameters **msg** (*Message*) – The value field of the message should contain some relative path to a file contained within *prefs.get('SOUNDDIR')* . eg. *'/songs/sadone.wav'* would return *'os.path.join(prefs.get('SOUNDDIR')/songs.sadone.wav'*

class Pilot_Station

Bases: *autopilot.networking.station.Station*

`Station` object used by *Pilot* objects.

Spawned with a `pusher` connected back to the *Terminal* .

Listens

Key	Method	Description
'STATE' 'COHERE' 'PING' 'START' 'STOP' 'PARAM' 'FILE'	*l_state()* *l_cohere()* *l_ping()* *l_start()* *l_stop()* *l_change()* *l_file()*	Pilot has changed state Make sure our data and the Terminal's match. The Terminal wants to know if we're listening We are being sent a task to start We are being told to stop the current task The Terminal is changing some task parameter We are receiving a file

Methods:

_pinger()	Periodically ping the terminal with our status
l_noop(msg)	
l_state(msg)	Pilot has changed state
l_cohere(msg)	Send our local version of the data table so the terminal can double check
l_ping([msg])	The Terminal wants to know our status
l_start(msg)	We are being sent a task to start
l_stop(msg)	Tell the pi to stop the task
l_change(msg)	The terminal is changing a parameter
l_file(msg)	We are receiving a file.
l_continuous(msg)	Forwards continuous data sent by children back to terminal.
l_child(msg)	Tell one or more children to start running a task.
l_forward(msg)	Just forward the message to the pi.

_pinger()

Periodically ping the terminal with our status

Calls its own timer to replace it

Returns:

l_noop(*msg*)

l_state(*msg: autopilot.networking.message.Message*)

Pilot has changed state

Stash it and alert the Terminal

Parameters **msg** (*Message*)

l_cohere(*msg: autopilot.networking.message.Message*)

Send our local version of the data table so the terminal can double check

> **Warning:** Not Implemented

Parameters **msg** (`Message`)

`l_ping`(*msg: Optional[*autopilot.networking.message.Message*] = None*)
The Terminal wants to know our status

Push back our current state.

Parameters **msg** (`Message`)

`l_start`(*msg:* autopilot.networking.message.Message)
We are being sent a task to start

If we need any files, request them.

Then send along to the pilot.

Parameters **msg** (`Message`) – value will contain a dictionary containing a task description.

`l_stop`(*msg:* autopilot.networking.message.Message)
Tell the pi to stop the task

Parameters **msg** (`Message`)

`l_change`(*msg:* autopilot.networking.message.Message)
The terminal is changing a parameter

> **Warning:** Not implemented

Parameters **msg** (`Message`)

`l_file`(*msg:* autopilot.networking.message.Message)
We are receiving a file.

Decode from b64 and save. Set the file_block.

Parameters **msg** (`Message`) – value will have 'path' and 'file', where the path determines where in *prefs.get('SOUNDDIR')* the b64 encoded 'file' will be saved.

`l_continuous`(*msg:* autopilot.networking.message.Message)
Forwards continuous data sent by children back to terminal.

Continuous data sources from this pilot should be streamed directly to the terminal.

Parameters **msg** (`Message`) – Continuous data message

`l_child`(*msg:* autopilot.networking.message.Message)
Tell one or more children to start running a task.

By default, the *key* argument passed to *self.send* is 'START'. However, this can be overriden by providing the desired string as *msg.value['KEY']*.

This checks the pref *CHILDID* to get the names of one or more children. If that pref is a string, sends the message to just that child. If that pref is a list, sends the message to each child in the list.

Parameters **msg** () – A message to send to the child or children.

Returns nothing

l_forward(*msg: autopilot.networking.message.Message*)
Just forward the message to the pi.

13.2 node

Classes:

Net_Node(id, upstream, port, listens, ...)	Drop in networking object to be given to any sub-object behind some external-facing Station object.

class Net_Node(*id: str, upstream: str, port: int, listens: Dict[str, Callable], instance: bool = True, upstream_ip: str = 'localhost', router_port: Optional[int] = None, daemon: bool = True, expand_on_receive: bool = True*)

Bases: `object`

Drop in networking object to be given to any sub-object behind some external-facing *Station* object.

To minimize the complexity of the network topology, the typical way to use ``Net_Node``s is through a *Station* ROUTER, rather than

addressing each other directly. Practically, this means that all messages are sent first to the parent `networking.Station` object, which then handles them, forwards them, etc. This proved to be horribly misguided and will be changed in v0.5.0 to support simplified messaging to a `agent_id.netnode_id` address. Until then the networking modules will be in a bit of flux.

To receive messages directly at this Net_Node, pass the `router_port` which will bind a `zmq.ROUTER` socket, and messages will be handled as regular 'listens' Note that Net_Nodes assume that they are the final recipients of messages, and so don't handle forwarding messages (unless a `listen` method explicitly does so), and will automatically deserialize them on receipt.

Note: Listen methods currently receive only the `value` of a message, this will change in v0.5.0, where they will receive the full message like `networking.Station` objects.

Parameters

- **id** (*str*) – What are we known as? What do we set our `identity` as?
- **upstream** (*str*) – The identity of the ROUTER socket used by our upstream *Station* object.
- **port** (*int*) – The port that our upstream ROUTER socket is bound to
- **listens** (*dict*) – Dictionary of functions to call for different types of messages. keys match the *Message.key*.
- **instance** (*bool*) – Should the node try and use the existing zmq context and tornado loop?
- **upstream_ip** (*str*) – If this Net_Node is being used on its own (ie. not behind a *Station*), it can directly connect to another node at this IP. Otherwise use 'localhost' to connect to a station.
- **router_port** (*int*) – Typically, Net_Nodes only have a single Dealer socket and receive messages from their encapsulating *Station*, but if you want to take this node offroad and use it independently, an int here binds a Router to the port.
- **daemon** (*bool*) – Run the IOLoop thread as a `daemon` (default: `True`)

Variables

- **context** (zmq.Context) – zeromq context
- **loop** (tornado.ioloop.IOLoop) – a tornado ioloop
- **sock** (zmq.Socket) – Our DEALER socket.
- **id** (*str*) – What are we known as? What do we set our identity as?
- **upstream** (*str*) – The identity of the ROUTER socket used by our upstream *Station* object.
- **port** (*int*) – The port that our upstream ROUTER socket is bound to
- **listens** (*dict*) – Dictionary of functions to call for different types of messages. keys match the *Message.key*.
- **outbox** (*dict*) – Messages that have been sent but have not been confirmed
- **timers** (*dict*) – dict of threading.Timer s that will check in on outbox messages
- **logger** (logging.Logger) – Used to log messages and network events.
- **msg_counter** (itertools.count) – counter to index our sent messages
- **loop_thread** (threading.Thread) – Thread that holds our loop. initialized with *daemon=True*

Attributes:

repeat_interval	
ip	Find our IP address

Methods:

init_networking()	Creates socket, connects to specified port on localhost, and starts the *threaded_loop()* as a daemon thread.
threaded_loop()	Run in a thread, either starts the IOLoop, or if it is already started (ie.
handle_listen(msg)	Upon receiving a message, call the appropriate listen method in a new thread and send confirmation it was received.
send([to, key, value, msg, repeat, flags, …])	Send a message via our sock , DEALER socket.
repeat()	Periodically (according to *repeat_interval*) resend messages that haven't been confirmed
l_confirm(value)	Confirm that a message was received.
l_stream(msg)	Reconstitute the original stream of messages and call their handling methods
prepare_message(to, key, value, repeat[, flags])	Instantiate a *Message* class, give it an ID and the rest of its attributes.
get_stream(id, key[, min_size, upstream, …])	Make a queue that another object can dump data into that sends on its own socket.
release()	

repeat_interval = 5

init_networking()
Creates socket, connects to specified port on localhost, and starts the *threaded_loop()* as a daemon thread.

threaded_loop()
Run in a thread, either starts the IOLoop, or if it is already started (ie. running in another thread), breaks.

handle_listen(*msg: List[bytes]*)
Upon receiving a message, call the appropriate listen method in a new thread and send confirmation it was received.

Note: Unlike *Station.handle_listen()* , only the *Message.value* is given to listen methods. This was initially intended to simplify these methods, but this might change in the future to unify the messaging system.

Parameters **msg** (*list*) – JSON *Message.serialize()* d message.

send(*to: Optional[Union[str, list]] = None*, *key: Optional[str] = None*, *value: Optional[Any] = None*, *msg: Optional[autopilot.networking.message.Message] = None*, *repeat: bool = True*, *flags=None*, *force_to: bool = False*)
Send a message via our `sock` , DEALER socket.

to is not required.

- If the node doesn't have a router, (or the recipient is not in the `Net_Node.senders` dict) every message is always sent to `upstream` . *to* can be included to send a message further up the network tree to a networking object we're not directly connected to.
- If the node has a router, since messages can only be sent on router sockets after the recipient has first sent us a message, if the `to` is in the `senders` dict, it will be directly sent via `Net_Node.router`
- If the `force_to` arg is `True`, send to the `to` recipient directly via the dealer `Net_Node.sock`
- If `to` is a list, or is intended to be sent as a multihop message with an explicit path, then networking objects will attempt to forward it along that path (disregarding implicit topology).

Either an already created *Message* should be passed as *msg*, or at least *key* must be provided for a new message created by *prepare_message()* .

A `threading.Timer` is created to resend the message using *repeat()* unless *repeat* is False.

Parameters

- **to** (*str, list*) – The identity of the socket this message is to. If not included, sent to `upstream()` .
- **key** (*str*) – The type of message - used to select which method the receiver uses to process this message.
- **value** – Any information this message should contain. Can be any type, but must be JSON serializable.
- **msg** (*.Message*) – An already created message.
- **repeat** (*bool*) – Should this message be resent if confirmation is not received?
- **flags** (*dict*)
- **force_to** (*bool*) – If we really really want to use the 'to' field to address messages (eg. node being used for direct communication), overrides default behavior of sending to upstream.

repeat()
Periodically (according to *repeat_interval*) resend messages that haven't been confirmed

TTL is decremented, and messages are resent until their TTL is 0.

l_confirm(*value*)
Confirm that a message was received.

Parameters **value** (*str*) – The ID of the message we are confirming.

l_stream(*msg*)
Reconstitute the original stream of messages and call their handling methods

The msg should contain an inner_key that indicates the key, and thus the handling method.

Parameters **msg** (*dict*) – Compressed stream sent by Net_Node._stream()

prepare_message(*to*, *key*, *value*, *repeat*, *flags=None*)
Instantiate a *Message* class, give it an ID and the rest of its attributes.

Parameters

- **flags**
- **repeat**
- **to** (*str*) – The identity of the socket this message is to
- **key** (*str*) – The type of message - used to select which method the receiver uses to process this message.
- **value** – Any information this message should contain. Can be any type, but must be JSON serializable.

get_stream(*id*, *key*, *min_size=5*, *upstream=None*, *port=None*, *ip=None*, *subject=None*, *q_size: Optional[int] = None*)
Make a queue that another object can dump data into that sends on its own socket. Smarter handling of continuous data than just hitting 'send' a shitload of times. :returns: Place to dump ur data :rtype: Queue

property ip: str
Find our IP address

Todo: this is a copy of the *Station.get_ip()* method – unify this in v0.5.0

returns (str): our IPv4 address.

release()

13.3 Message

Classes:

Message([msg, expand_arrays])	A formatted message.

class Message(*msg=None*, *expand_arrays=False*, ***kwargs*)
Bases: object

A formatted message.

id, *to*, *sender*, and *key* are required attributes, but any other key-value pair passed on init is added to the message's attributes and included in the message.

Can be indexed and set like a dictionary (message['key'], etc.)

Variables

- **id** (*str*) – ID that uniquely identifies a message. format {sender.id}_{number}
- **to** (*str*) – ID of socket this message is addressed to
- **sender** (*str*) – ID of socket where this message originates
- **key** (*str*) – Type of message, used to select a listen method to process it
- **value** – Body of message, can be any type but must be JSON serializable.
- **timestamp** (*str*) – Timestamp of message creation
- **ttl** (*int*) – Time-To-Live, each message is sent this many times at max, each send decrements ttl.

Parameters

- ***args**
- ****kwargs**

Attributes:

id
to
sender
key
value
changed
flags
timestamp
ttl
serialized

Methods:

__getitem__(key)	**Parameters key**

continues on next page

Table 13 – continued from previous page

`__setitem__`(key, value)	**Parameters** • **key**
`_serialize_numpy`(array)	Serialize a numpy array for sending over the wire
`expand`()	Don't decompress numpy arrays by default for faster IO, explicitly expand them when needed
`__delitem__`(key)	**Parameters** **key**
`__contains__`(key)	**Parameters** **key**
`get_timestamp`()	Get a Python timestamp
`validate`()	Checks if *id*, *to*, *sender*, and *key* are all defined.
`serialize`()	Serializes all attributes in *__dict__* using json.

id = None

to = None

sender = None

key = None

value = None

changed = False

flags = {}

timestamp = None

ttl = 2

serialized = None

__getitem__(*key*)

Parameters **key**

__setitem__(*key*, *value*)

Parameters

- **key**
- **value**

_serialize_numpy(*array*)

Serialize a numpy array for sending over the wire

Parameters **array**

Returns:

expand()

Don't decompress numpy arrays by default for faster IO, explicitly expand them when needed

Returns

`__delitem__`(*key*)

Parameters **key**

`__contains__`(*key*)

Parameters **key**

`get_timestamp`()

Get a Python timestamp

Returns Isoformatted timestamp from `datetime`

Return type str

`validate`()

Checks if *id*, *to*, *sender*, and *key* are all defined.

Returns Does message have all required attributes set?

Return type bool (True)

`serialize`()

Serializes all attributes in __*dict*__ using json.

Returns JSON serialized message.

Return type str

CHAPTER

FOURTEEN

STIM

Classes:

Stim()	Placeholder stimulus meta-object until full implementation

class Stim

Bases: `object`

Placeholder stimulus meta-object until full implementation

14.1 managers

This is a scrappy first draft of a stimulus manager that will be built out to incorporate arbitrary stimulus logic. For now you can subclass *Stim_Manager* and redefine *next_stim*

Todo: Make this more general, for more than just sounds.

Functions:

`init_manager`(stim)	

Classes:

`Stim_Manager`([stim])	Yield sounds according to some set of rules.
`Proportional`(stim)	Present groups of stimuli with a particular frequency.
`Bias_Correction`([mode, thresh, window])	Basic Bias correction module.

init_manager(*stim*)

class Stim_Manager(*stim=None*)

Bases: `object`

Yield sounds according to some set of rules.

Currently implemented:

- **correction trials - If a subject continually answers to one side incorrectly, keep** the correct answer on the other side until they answer in that direction

- bias correction - above some bias threshold, skew the correct answers to the less-responded side

Variables

- **stimuli** (*dict*) – Dictionary of instantiated stimuli like:

```
{'L': [Tone1, Tone2, ...], 'R': [Tone3, Tone4, ...]}
```

- **target** (*'L', 'R'*) – What is the correct port?
- **distractor** (*'L', 'R'*) – What is the incorrect port?
- **response** (*'L', 'R'*) – What was the last response?
- **correct** (*0, 1*) – Was the last response correct?
- **last_stim** – What was the last stim? (one of *self.stimuli*)
- **correction** (*bool*) – Are we doing correction trials?
- **correction_trial** (*bool*) – Is this a correction trial?
- **last_was_correction** (*bool*) – Was the last trial a correction trial?
- **correction_pct** (*float*) – proportion of trials that are correction trials
- **bias** – False, or a bias correction mode.

Parameters **stim** (*dict*) –

Dictionary describing sound stimuli, in a format like:

```
{
'L': [{'type':'tone',...},{...}],
'R': [{'type':'tone',...},{...}]
}
```

Methods:

do_correction([correction_pct])	Called to set correction trials to True and correction percent.
do_bias(**kwargs)	Instantiate a *Bias_Correction* module
init_sounds(sound_dict)	Instantiate sound objects, using the 'type' value to choose an object from `autopilot.get('sound')` .
set_triggers(trig_fn)	Give a callback function to all of our stimuli for when the stimulus ends.
make_punishment(type, duration)	**Warning:** Not Implemented

continues on next page

Table 4 – continued from previous page

`play_punishment()`	**Warning:** Not Implemented
`next_stim()`	Compute and return the next stimulus
`compute_correction()`	If *self.correction* is true, compute correction trial logic during *next_stim*.
`update(response, correct)`	At the end of a trial, update the status of our internal variables with the outcome of the trial.
`end()`	End all of our stim.

do_correction(*correction_pct=0.5*)
Called to set correction trials to True and correction percent.

Parameters **correction_pct** (*float*) – Proportion of trials that should randomly be set to be correction trials.

do_bias(***kwargs*)
Instantiate a *Bias_Correction* module

Parameters **kwargs** – parameters to initialize *Bias_Correction* with.

init_sounds(*sound_dict*)
Instantiate sound objects, using the 'type' value to choose an object from `autopilot.get('sound')` .

Parameters **sound_dict** (*dict*) –

a dictionary like:: { 'L': [{'type':'tone',. . . },{. . . }], 'R': [{'type':'tone',. . . },{. . . }] }

set_triggers(*trig_fn*)
Give a callback function to all of our stimuli for when the stimulus ends.

Note: Stimuli need a *set_trigger* method.

Parameters **trig_fn** (*callable*) – A function to be given to stimuli via *set_trigger*

make_punishment(*type*, *duration*)

Warning: Not Implemented

Parameters

- **type**
- **duration**

play_punishment()

> **Warning:** Not Implemented

next_stim()
Compute and return the next stimulus

If we are doing correction trials, compute that.

Same thing with bias correction.

Otherwise, randomly select a stimulus to present.

Returns ('L'/'R' Target, 'L'/'R' distractor, Stimulus to present)

compute_correction()
If *self.correction* is true, compute correction trial logic during *next_stim*.

- If the last trial was a correction trial and the response to it wasn't correct, return True
- If the last trial was a correction trial and the response was correct, return False
- If the last trial as not a correction trial, but a randomly generated float is less than *correction_pct*, return True.

Returns whether this trial should be a correction trial.

Return type bool

update(*response*, *correct*)
At the end of a trial, update the status of our internal variables with the outcome of the trial.

Parameters

- **response** (*'L', 'R'*) – How the subject responded
- **correct** (*0, 1*) – Whether the response was correct.

end()
End all of our stim. Stim should have an *.end()* method of their own

class Proportional(*stim*)
Bases: *autopilot.stim.managers.Stim_Manager*

Present groups of stimuli with a particular frequency.

Frequencies do not need to add up to 1, groups will be selected with the frequency (frequency)/(sum(frequencies)).

Parameters **stim** (*dict*) – Dictionary with the structure:

```
{'manager': 'proportional',
 'type': 'sounds',
 'groups': (
     {'name':'group_name',
      'frequency': 0.2,
      'sounds':{
          'L': [{Tone1_params}, {Tone2_params}...],
          'R': [{Tone3_params}, {Tone4_params}...]
      }
    },
    {'name':'second_group',
```

(continues on next page)

(continued from previous page)

```
        'frequency': 0.8,
        'sounds':{
            'L': [{Tone1_params}, {Tone2_params}...],
            'R': [{Tone3_params}, {Tone4_params}...]
        }
      })
}
```

Variables

- **stimuli** (*dict*) – A dictionary of stimuli organized into groups
- **groups** (*dict*) – A dictionary mapping group names to frequencies

Parameters **stim** (*dict*) –

Dictionary describing sound stimuli, in a format like:

```
{
'L': [{'type':'tone',...},{...}],
'R': [{'type':'tone',...},{...}]
}
```

Methods:

`init_sounds_grouped`(sound_stim)	Instantiate sound objects similarly to *Stim_Manager*, just organizes them into groups.
`init_sounds_individual`(sound_stim)	Initialize sounds with individually set presentation frequencies.
`store_groups`(stim)	store groups and frequencies
`set_triggers`(trig_fn)	Give a callback function to all of our stimuli for when the stimulus ends.
`next_stim`()	Compute and return the next stimulus

init_sounds_grouped(*sound_stim*)

Instantiate sound objects similarly to *Stim_Manager*, just organizes them into groups.

Parameters **sound_stim** (*tuple, list*) – an iterator like:

```
(
 {'name':'group_name',
  'frequency': 0.2,
  'sounds': {
      'L': [{Tone1_params}, {Tone2_params}...],
      'R': [{Tone3_params}, {Tone4_params}...]
    }
},
{'name':'second_group',
  'frequency': 0.8,
  'sounds':{
      'L': [{Tone1_params}, {Tone2_params}...],
      'R': [{Tone3_params}, {Tone4_params}...]
  }
})
```

init_sounds_individual(*sound_stim*)
Initialize sounds with individually set presentation frequencies.

Todo: This method reflects the need for managers to have a unified schema, which will be built in a future release of Autopilot.

Parameters **sound_stim** (*dict*) – Dictionary of {'side':[sound_params]} to generate sound stimuli

Returns:

store_groups(*stim*)
store groups and frequencies

set_triggers(*trig_fn*)
Give a callback function to all of our stimuli for when the stimulus ends.

Note: Stimuli need a *set_trigger* method.

Parameters **trig_fn** (*callable*) – A function to be given to stimuli via *set_trigger*

next_stim()
Compute and return the next stimulus

If we are doing correction trials, compute that.

Same thing with bias correction.

Otherwise, randomly select a stimulus to present, weighted by its group frequency.

Returns ('L'/'R' Target, 'L'/'R' distractor, Stimulus to present)

class Bias_Correction(*mode='thresholded_linear'*, *thresh=0.2*, *window=100*)
Bases: `object`

Basic Bias correction module. Modifies the threshold of random stimulus choice based on history of biased responses.

Variables

- **responses** (`collections.deque`) – History of prior responses
- **targets** (`collections.deque`) – History of prior targets.

Parameters

- **mode** – One of the following:
  - ***'thresholded linear'*** [above some threshold, do linear bias correction] eg. if response rate 65% left, make correct be right 65% of the time
- **thresh** (*float*) – threshold above chance, ie. 0.2 means has to be 70% biased in window
- **window** (*int*) – number of trials to calculate bias over

Methods:

`next_bias()`	Compute the next bias depending on *self.mode*

continues on next page

Table 6 – continued from previous page

thresholded_linear()	If we are above the threshold, linearly correct the rate of presentation to favor the rarely responded side.
update(response, target)	Store some new response and target values

next_bias()
Compute the next bias depending on *self.mode*

Returns Some threshold *Stim_Manager* uses to decide left vs right.

Return type float

thresholded_linear()
If we are above the threshold, linearly correct the rate of presentation to favor the rarely responded side.

eg. if response rate 65% left, make correct be right 65% of the time

Returns 0.5-bias, where bias is the difference between the mean response and mean target.

Return type float

update(*response*, *target*)
Store some new response and target values

Parameters

- **response** (*'R', 'L'*) – Which side the subject responded to
- **target** (*'R', 'L'*) – The correct side.

14.2 sound

Module for generating and playing sounds.

This module contains the following files:

sounds.py : Defines classes for generating sounds jackclient.py : Define the interface to the jack client pyoserver.py : Defines the interface to the pyo server

The use of pyoserver is discouraged in favor of jackclient. This is controlled by the pref *AUDIOSERVER*.

14.2.1 jackclient

Client that dumps samples directly to the jack client with the `jack` package.

Data:

SERVER	After initializing, JackClient will register itself with this variable.
FS	Sampling rate of the active server
BLOCKSIZE	Blocksize, or the amount of samples processed by jack per each `JackClient.process()` call.
QUEUE	Queue to be loaded with frames of BLOCKSIZE audio.
Q_LOCK	Lock that enforces a single writer to the *QUEUE* at a time.

continues on next page

Table 7 – continued from previous page

CONTINUOUS	Event that (when set) signals the sound server should play some sound continuously rather than remain silent by default (eg.
CONTINUOUS_QUEUE	Queue that
CONTINUOUS_LOOP	Event flag that is set when frames dropped into the CONTINUOUS_QUEUE should be looped (eg.

Classes:

JackClient([name])	Client that dumps frames of audio directly into a running jackd client.

SERVER = None

After initializing, JackClient will register itself with this variable.

Type *JackClient*

FS = 192000

Sampling rate of the active server

Type int

BLOCKSIZE = 1024

Blocksize, or the amount of samples processed by jack per each *JackClient.process()* call.

Type int

QUEUE = None

Queue to be loaded with frames of BLOCKSIZE audio.

Type multiprocessing.Queue

PLAY = <multiprocessing.synchronize.Event object at 0x7fcec8892a10>

Event used to trigger loading samples from *QUEUE*, ie. playing.

Type multiprocessing.Event

STOP = <multiprocessing.synchronize.Event object at 0x7fce74068b10>

Event that is triggered on the end of buffered audio.

Note: NOT an event used to stop audio.

Type multiprocessing.Event

Q_LOCK = None

Lock that enforces a single writer to the *QUEUE* at a time.

Type multiprocessing.Lock

CONTINUOUS = None

Event that (when set) signals the sound server should play some sound continuously rather than remain silent by default (eg. play a background sound).

Type multiprocessing.Event

CONTINUOUS_QUEUE = None

Queue that

Type multiprocessing.Queue

CONTINUOUS_LOOP = None

Event flag that is set when frames dropped into the CONTINUOUS_QUEUE should be looped (eg. in the case of stationary background noise), otherwise they are played and then discarded (ie. the sound is continuously generating and submitting samples)

Type multiprocessing.Event

class JackClient(*name='jack_client'*)

Bases: `multiprocessing.context.Process`

Client that dumps frames of audio directly into a running jackd client.

When first initialized, sets module level variables above.

Variables

- **name** (*str*) – name of client, default "jack_client"
- **q** (Queue) – Queue that stores buffered frames of audio
- **q_lock** (Lock) – Lock that manages access to the Queue
- **play_evt** (multiprocessing.Event) – Event used to trigger loading samples from *QUEUE*, ie. playing.
- **stop_evt** (multiprocessing.Event) – Event that is triggered on the end of buffered audio.
- **quit_evt** (multiprocessing.Event) – Event that causes the process to be terminated.
- **client** (jack.Client) – Client to interface with jackd
- **blocksize** (*int*) – The blocksize - ie. samples processed per *JackClient.process()* call.
- **fs** (*int*) – Sampling rate of client
- **zero_arr** (numpy.ndarray) – cached array of zeroes used to fill jackd pipe when not processing audio.
- **continuous_cycle** (itertools.cycle) – cycle of frames used for continuous sounds
- **mono_output** (*bool*) – `True` or `False` depending on if the number of output channels is 1 or >1, respectively. detected and set in *JackClient.boot_server()* , initialized to `True` (which is hopefully harmless)

Parameters name

Methods:

boot_server()	Called by *JackClient.run()* to boot the server upon starting the process.
run()	Start the process, boot the server, start processing frames and wait for the end.
quit()	Set the `JackClient.quit_evt`
process(frames)	Process a frame of audio.
write_to_outports(data)	Write the sound in *data* to the outport(s).

boot_server()

Called by *JackClient.run()* to boot the server upon starting the process.

Activates the client and connects it to the physical speaker outputs as determined by *prefs.get('OUTCHANNELS')*.

This is the interpretation of OUTCHANNELS: * empty string

'mono' audio: the same sound is always played to all channels. Connect a single virtual outport to every physical channel. If multi-channel sound is provided, raise an error.

- **a single int (example: J)** This is equivalent to [J]. The first virtual outport will be connected to physical channel J. Note this is NOT the same as 'mono', because only one speaker plays, instead of all speakers.
- **a list (example: [I, J])** The first virtual outport will be connected to physical channel I. The second virtual outport will be connected to physical channel J. And so on. If 1-dimensional sound is provided, play the same to all speakers (like mono mode). If multi-channel sound is provided and the number of channels is different form the length of this list, raise an error.

`jack.Client` s can't be kept alive, so this must be called just before processing sample starts.

run()

Start the process, boot the server, start processing frames and wait for the end.

quit()

Set the `JackClient.quit_evt`

process(*frames*)

Process a frame of audio.

If the `JackClient.play_evt` is not set, fill port buffers with zeroes.

Otherwise, pull frames of audio from the `JackClient.q` until it's empty.

When it's empty, set the `JackClient.stop_evt` and clear the `JackClient.play_evt` .

Parameters **frames** – number of frames (samples) to be processed. unused. passed by jack client

write_to_outports(*data*)

Write the sound in *data* to the outport(s).

If self.mono_output:

If data is 1-dimensional: Write that data to the single outport, which goes to all speakers.

Otherwise, raise an error.

If not self.mono_output:

If data is 1-dimensional: Write that data to every outport

If data is 2-dimensional: Write one column to each outport, raising an error if there is a different number of columns than outports.

14.2.2 pyoserver

Functions:

pyo_server([debug])	Returns a booted and started pyo audio server

pyo_server(*debug=False*)

Returns a booted and started pyo audio server

Warning: Use of pyo is generally discouraged due to dropout issues and the general opacity of the module.

Parameters **debug** (*bool*) – If true, setVerbosity of pyo server to 8.

14.2.3 sounds

This module defines classes to generate different sounds.

These classes are currently implemented: * Tone : a sinuosoidal pure tone * Noise : a burst of white noise * File : read from a file * Speech * Gap

The behavior of this module depends on *prefs.get('AUDIOSERVER')*. * If this is 'jack', or True:

Then import jack, define Jack_Sound, and all sounds inherit from that.

- **If this is 'pyo':** Then import pyo, define PyoSound, and all sounds inherit from that.
- **If this is 'docs':** Then import both jack and pyo, define both Jack_Sound and PyoSound, and all sounds inherit from *object*.
- **Otherwise:** Then do not import jack or pyo, or define either Jack_Sound or PyoSound, and all sounds inherit from *object*.

Todo: Implement sound level and filter calibration

Classes:

`Jack_Sound`()	Base class for sounds that use the `JackClient` audio server.
`BASE_CLASS`	alias of `autopilot.stim.sound.sounds.Jack_Sound`
`Tone`(frequency, duration[, amplitude])	The Humble Sine Wave
`Noise`(duration[, amplitude, channel])	Generates a white noise burst with specified parameters
`File`(path[, amplitude])	A .wav file.
`Gap`(duration, **kwargs)	A silent sound that does not pad its final chunk – used for creating precise silent gaps in a continuous noise.

Data:

`STRING_PARAMS`	These parameters should be given string columns rather than float columns.

Functions:

`int_to_float`(audio)	Convert 16 or 32 bit integer audio to 32 bit float.

class Jack_Sound

Bases: `object`

Base class for sounds that use the `JackClient` audio server.

Variables

- **PARAMS** (*list*) – List of strings of parameters that need to be defined for this sound
- **type** (*str*) – Human readable name of sound type
- **duration** (*float*) – Duration of sound in ms
- **amplitude** (*float*) – Amplitude of sound as proportion of 1 (eg 0.5 is half amplitude)
- **table** (numpy.ndarray) – A Numpy array of samples
- **chunks** (*list*) – `table` split up into chunks of *BLOCKSIZE*
- **trigger** (*callable*) – A function that is called when the sound completes
- **nsamples** (*int*) – Number of samples in the sound
- **padded** (*bool*) – Whether the sound had to be padded with zeros when split into chunks (ie. sound duration was not a multiple of BLOCKSIZE).
- **fs** (*int*) – sampling rate of client from *jackclient.FS*
- **blocksize** (*int*) – blocksize of client from *jackclient.BLOCKSIZE*
- **server** (Jack_Client) – Current Jack Client
- **q** (multiprocessing.Queue) – Audio Buffer queue from *jackclient.QUEUE*
- **q_lock** (multiprocessing.Lock) – Audio Buffer lock from *jackclient.Q_LOCK*
- **play_evt** (multiprocessing.Event) – play event from *jackclient.PLAY*
- **stop_evt** (multiprocessing.Event) – stop event from *jackclient.STOP*
- **buffered** (*bool*) – has this sound been dumped into the `q` ?
- **buffered_continuous** (*bool*) – Has the sound been dumped into the `continuous_q`?

Initialize a new Jack_Sound

This sets sound-specific parameters to None, set jack-specific parameters to their equivalents in jackclient, initializes some other flags and a logger.

Attributes:

PARAMS	list of strings of parameters to be defined
type	string human readable name of sound
server_type	type of server, always 'jack' for *Jack_Sound* s.

Methods:

chunk([pad])	Split our *table* up into a list of `Jack_Sound.blocksize` chunks.
set_trigger(trig_fn)	Set a trigger function to be called when the `stop_evt` is set.
wait_trigger()	Wait for the stop_evt trigger to be set for at least a second after the sound should have ended.
get_nsamples()	given our fs and duration, how many samples do we need?
quantize_duration([ceiling])	Extend or shorten a sound so that it is a multiple of *jackclient.BLOCKSIZE*
buffer()	Dump chunks into the sound queue.

continues on next page

Table 15 – continued from previous page

buffer_continuous()	Dump chunks into the continuous sound queue for looping.
play()	Play ourselves.
play_continuous([loop])	Play the sound continuously.
stop_continuous()	Stop playing a continuous sound
end()	Release any resources held by this sound

PARAMS = []
list of strings of parameters to be defined

Type list

type = None
string human readable name of sound

Type str

server_type = 'jack'
type of server, always 'jack' for *Jack_Sound* s.

Type str

chunk(*pad=True*)
Split our *table* up into a list of `Jack_Sound.blocksize` chunks.

Parameters

- **pad** (*bool*) – If the sound is not evenly divisible into chunks,
- **pad with zeros** (*True, default*)
- **with its continuous sound**

set_trigger(*trig_fn*)
Set a trigger function to be called when the `stop_evt` is set.

Parameters **trig_fn** (*callable*) – Some callable

wait_trigger()
Wait for the stop_evt trigger to be set for at least a second after the sound should have ended.

Call the trigger when the event is set.

get_nsamples()
given our fs and duration, how many samples do we need?

literally:

```
np.ceil((self.duration/1000.)*self.fs).astype(np.int)
```

quantize_duration(*ceiling=True*)
Extend or shorten a sound so that it is a multiple of `jackclient.BLOCKSIZE`

Parameters **ceiling** (*bool*) – If true, extend duration, otherwise decrease duration.

buffer()
Dump chunks into the sound queue.

After the last chunk, a *None* is put into the queue. This tells the jack server that the sound is over and that it should clear the play flag.

buffer_continuous()

Dump chunks into the continuous sound queue for looping.

Continuous shoulds should always have full frames - ie. the number of samples in a sound should be a multiple of *jackclient.BLOCKSIZE*.

This method will call *quantize_duration()* to force duration such that the sound has full frames.

An exception will be raised if the sound has been padded.

play()

Play ourselves.

If we're not buffered, be buffered.

Otherwise, set the play event and clear the stop event.

If we have a trigger, set a Thread to wait on it.

play_continuous(*loop=True*)

Play the sound continuously.

Sound will be paused if another sound has its 'play' method called.

Currently - only looping is implemented: the full sound is loaded by the jack client and repeated indefinitely.

In the future, sound generation methods will be refactored as python generators so sounds can be continuously generated and played.

Parameters **loop** (*bool*) – whether the sound will be stored by the jack client and looped (True), or whether the sound will be continuously streamed (False, not implemented)

Returns:

todo:

```
merge into single play method that changes behavior if continuous or not
```

stop_continuous()

Stop playing a continuous sound

Should be merged into a general stop method

end()

Release any resources held by this sound

BASE_CLASS

alias of *autopilot.stim.sound.sounds.Jack_Sound* **Attributes:**

PARAMS	list of strings of parameters to be defined
server_type	type of server, always 'jack' for *Jack_Sound* s.
type	string human readable name of sound

Methods:

buffer()	Dump chunks into the sound queue.
buffer_continuous()	Dump chunks into the continuous sound queue for looping.
chunk([pad])	Split our *table* up into a list of Jack_Sound.blocksize chunks.
end()	Release any resources held by this sound

continues on next page

Table 17 – continued from previous page

get_nsamples()	given our fs and duration, how many samples do we need?
play()	Play ourselves.
play_continuous([loop])	Play the sound continuously.
quantize_duration([ceiling])	Extend or shorten a sound so that it is a multiple of *jackclient.BLOCKSIZE*
set_trigger(trig_fn)	Set a trigger function to be called when the stop_evt is set.
stop_continuous()	Stop playing a continuous sound
wait_trigger()	Wait for the stop_evt trigger to be set for at least a second after the sound should have ended.

class Tone(*frequency*, *duration*, *amplitude=0.01*, ***kwargs*)

Bases: *autopilot.stim.sound.sounds.Jack_Sound*

The Humble Sine Wave

Parameters

- **frequency** (*float*) – frequency of sin in Hz
- **duration** (*float*) – duration of the sin in ms
- **amplitude** (*float*) – amplitude of the sound as a proportion of 1.
- ****kwargs** – extraneous parameters that might come along with instantiating us

Attributes:

PARAMS	list of strings of parameters to be defined
type	string human readable name of sound

Methods:

init_sound()	Create a sine wave table using pyo or numpy, depending on the server type.

PARAMS = ['frequency', 'duration', 'amplitude']

list of strings of parameters to be defined

Type list

type = 'Tone'

string human readable name of sound

Type str

init_sound()

Create a sine wave table using pyo or numpy, depending on the server type.

class Noise(*duration*, *amplitude=0.01*, *channel=None*, ***kwargs*)

Bases: *autopilot.stim.sound.sounds.Jack_Sound*

Generates a white noise burst with specified parameters

The *type* attribute is always "Noise".

Initialize a new white noise burst with specified parameters.

The sound itself is stored as the attribute *self.table*. This can be 1-dimensional or 2-dimensional, depending on *channel*. If it is 2-dimensional, then each channel is a column.

Parameters

- **duration** (*float*) – duration of the noise
- **amplitude** (*float*) – amplitude of the sound as a proportion of 1.
- **channel** (*int or None*) – which channel should be used If 0, play noise from the first channel If 1, play noise from the second channel If None, send the same information to all channels ("mono")
- ****kwargs** – extraneous parameters that might come along with instantiating us

Attributes:

`PARAMS`	list of strings of parameters to be defined
`type`	string human readable name of sound

Methods:

`init_sound()`	Defines *self.table*, the waveform that is played.

`PARAMS = ['duration', 'amplitude', 'channel']`
list of strings of parameters to be defined

Type list

`type = 'Noise'`
string human readable name of sound

Type str

`init_sound()`
Defines *self.table*, the waveform that is played.

The way this is generated depends on *self.server_type*, because parameters like the sampling rate cannot be known otherwise.

The sound is generated and then it is "chunked" (zero-padded and divided into chunks). Finally *self.initialized* is set True.

`class File(path, amplitude=0.01, **kwargs)`
Bases: `autopilot.stim.sound.sounds.Jack_Sound`

A .wav file.

Todo: Generalize this to other audio types if needed.

Parameters

- **path** (*str*) – Path to a .wav file relative to the *prefs.get('SOUNDDIR')*
- **amplitude** (*float*) – amplitude of the sound as a proportion of 1.
- ****kwargs** – extraneous parameters that might come along with instantiating us

Attributes:

PARAMS	list of strings of parameters to be defined
type	string human readable name of sound

Methods:

init_sound()	Load the wavfile with `scipy.io.wavfile`, converting int to float as needed.

PARAMS = ['path', 'amplitude']
list of strings of parameters to be defined

Type list

type = 'File'
string human readable name of sound

Type str

init_sound()
Load the wavfile with `scipy.io.wavfile`, converting int to float as needed.

Create a sound table, resampling sound if needed.

class Gap(*duration*, ***kwargs*)
Bases: *autopilot.stim.sound.sounds.Jack_Sound*

A silent sound that does not pad its final chunk – used for creating precise silent gaps in a continuous noise.

Parameters **duration** (*float*) – duration of gap in ms

Variables **gap_zero** (*bool*) – True if duration is zero, effectively do nothing on play.

Attributes:

type	string human readable name of sound
PARAMS	list of strings of parameters to be defined

Methods:

init_sound()	Create and chunk an array of zeros according to `Gap.duration`
chunk([pad])	If gap is not duration == 0, call parent `chunk`.
buffer()	Dump chunks into the sound queue.
play()	Play ourselves.

type = 'Gap'
string human readable name of sound

Type str

PARAMS = ['duration']
list of strings of parameters to be defined

Type list

init_sound()
Create and chunk an array of zeros according to `Gap.duration`

chunk(*pad=False*)
If gap is not duration == 0, call parent chunk. :Parameters: **pad** (*bool*) – unused, passed to parent chunk

buffer()
Dump chunks into the sound queue.

After the last chunk, a *None* is put into the queue. This tells the jack server that the sound is over and that it should clear the play flag.

play()
Play ourselves.

If we're not buffered, be buffered.

Otherwise, set the play event and clear the stop event.

If we have a trigger, set a Thread to wait on it.

STRING_PARAMS = ['path', 'type']
These parameters should be given string columns rather than float columns.

Bother Jonny to do this better bc it's really bad.

int_to_float(*audio*)
Convert 16 or 32 bit integer audio to 32 bit float.

Parameters **audio** (numpy.ndarray) – a numpy array of audio

Returns Audio that has been rescaled and converted to a 32 bit float.

Return type numpy.ndarray

CHAPTER

FIFTEEN

TASKS

15.1 task

Classes:

Task(*args, **kwargs)	Generic Task metaclass

class Task(**args*, ***kwargs*)

Bases: object

Generic Task metaclass

Variables

- **PARAMS** (collections.OrderedDict) – Params to define task, like:

```
PARAMS = odict()
PARAMS['reward']          = {'tag':'Reward Duration (ms)',
                             'type':'int'}
PARAMS['req_reward']      = {'tag':'Request Rewards',
                             'type':'bool'}
```

- **HARDWARE** (*dict*) – dict for necessary hardware, like:

```
HARDWARE = {
    'POKES':{
        'L': hardware.Beambreak, ...
    },
    'PORTS':{
        'L': hardware.Solenoid, ...
    }
}
```

- **PLOT** (*dict*) – Dict of plotting parameters, like:

```
PLOT = {
    'data': {
        'target'   : 'point',
        'response' : 'segment',
        'correct'  : 'rollmean'
    },
    'chance_bar'  : True, # Draw a red bar at 50%
```

(continues on next page)

(continued from previous page)

```
    'roll_window' : 50 # number of trials to roll window over
}
```

- **Trial_Data** (tables.IsDescription) – Data table description, like:

```
class TrialData(tables.IsDescription):
    trial_num = tables.Int32Col()
    target = tables.StringCol(1)
    response = tables.StringCol(1)
    correct = tables.Int32Col()
    correction = tables.Int32Col()
    RQ_timestamp = tables.StringCol(26)
    DC_timestamp = tables.StringCol(26)
    bailed = tables.Int32Col()
```

- **STAGE_NAMES** (*list*) – List of stage method names
- **stage_block** (threading.Event) – Signal when task stages complete.
- **punish_stim** (*bool*) – Do a punishment stimulus
- **stages** (*iterator*) – Some generator or iterator that continuously returns the next stage method of a trial
- **triggers** (*dict*) – Some mapping of some pin to callback methods
- **pins** (*dict*) – Dict to store references to hardware
- **pin_id** (*dict*) – Reverse dictionary, pin numbers back to pin letters.
- **punish_block** (threading.Event) – Event to mark when punishment is occuring
- **logger** (logging.Logger) – gets the 'main' logger for now.

Parameters

- **subject** (*str*) – Name of subject running the task
- **current_trial** (*int*) – Current trial number, default 0
- ***args** ()
- ****kwargs** ()

Attributes:

PARAMS
HARDWARE
STAGE_NAMES
PLOT

Classes:

TrialData()

Methods:

init_hardware()	Use the HARDWARE dict that specifies what we need to run the task alongside the HARDWARE subdict in `prefs` to tell us how they're plugged in to the pi
set_reward([vol, duration, port])	Set the reward value for each of the 'PORTS'.
handle_trigger(pin[, level, tick])	All GPIO triggers call this function with the pin number, level (high, low), and ticks since booting pigpio.
set_leds([color_dict])	Set the color of all LEDs at once.
flash_leds()	flash lights for punish_dir
end()	Release all hardware objects

PARAMS = OrderedDict()

HARDWARE = {}

STAGE_NAMES = []

PLOT = {}

class TrialData

Bases: `tables.description.IsDescription`

Attributes:

columns

columns = { 'session': Int32Col(shape=(), dflt=0, pos=None), 'trial_num': Int32Col(shape=(), dflt=0, pos=None)}

init_hardware()

Use the HARDWARE dict that specifies what we need to run the task alongside the HARDWARE subdict in `prefs` to tell us how they're plugged in to the pi

Instantiate the hardware, assign it `Task.handle_trigger()` as a callback if it is a trigger.

set_reward(*vol=None*, *duration=None*, *port=None*)

Set the reward value for each of the 'PORTS'.

Parameters

- **vol** (*float, int*) – Volume of reward in uL
- **duration** (*float*) – Duration to open port in ms
- **port** (*None, Port_ID*) – If *None*, set everything in 'PORTS', otherwise only set *port*

handle_trigger(*pin*, *level=None*, *tick=None*)

All GPIO triggers call this function with the pin number, level (high, low), and ticks since booting pigpio.

Calls any trigger assigned to the pin in *self.triggers* , unless during punishment (returns).

Parameters

- **pin** (*int*) – BCM Pin number
- **level** (*bool*) – True, False high/low
- **tick** (*int*) – ticks since booting pigpio

`set_leds`(*color_dict=None*)
Set the color of all LEDs at once.

Parameters **color_dict** (*dict*) – If None, turn LEDs off, otherwise like:

{'pin': [R,G,B], 'pin2: [R,G,B]}

`flash_leds`()
flash lights for punish_dir

`end`()
Release all hardware objects

15.2 children

Sub-tasks that serve as children to other tasks.

Note: The Child agent will be formalized in an upcoming release, until then these classes remain relatively undocumented as their design will likely change.

Classes:

`Child`()	Just a placeholder class for now to work with `autopilot.get()`
`Wheel_Child`([stage_block, fs, thresh])	
`Video_Child`([cams, stage_block, start_now])	**Parameters** **cams** (*dict, list*) –
`Transformer`(transform, operation[, node_id, ...])	**Parameters** • **transform**

class Child
Bases: `object`

Just a placeholder class for now to work with `autopilot.get()`

class Wheel_Child(*stage_block=None*, *fs=10*, *thresh=100*, ***kwargs*)
Bases: `autopilot.tasks.children.Child`

Attributes:

`STAGE_NAMES`
`PARAMS`

continues on next page

Table 7 – continued from previous page

HARDWARE

Methods:

noop()
end()

STAGE_NAMES = ['collect']

PARAMS = OrderedDict([('fs', {'tag': 'Velocity Reporting Rate (Hz)', 'type': 'int'}), ('thresh', {'tag': 'Distance Threshold', 'type': 'int'})])

HARDWARE = { 'OUTPUT': <class 'autopilot.hardware.gpio.Digital_Out'>, 'WHEEL': <class 'autopilot.hardware.usb.Wheel'>}

noop()

end()

class Video_Child(*cams=None*, *stage_block=None*, *start_now=True*, ***kwargs*)

Bases: *autopilot.tasks.children.Child*

Parameters cams (*dict, list*) –

Should be a dictionary of camera parameters or a list of dicts. Dicts should have, at least:

```
{
    'type': 'string_of_camera_class',
    'name': 'name_of_camera_in_task',
    'param1': 'first_param'
}
```

Attributes:

PARAMS

Methods:

start()
stop()
noop()

PARAMS = OrderedDict([('cams', { 'tag': 'Dictionary of camera params, or list of dicts', 'type': ('dict', 'list')})])

start()

stop()

noop()

class **Transformer**(*transform*, *operation: str = 'trigger'*, *node_id=None*, *return_id='T'*, *return_ip=None*, *return_port=None*, *return_key=None*, *router_port=None*, *stage_block=None*, *value_subset=None*, ***kwargs*)

Bases: `autopilot.tasks.children.Child`

Parameters

- **transform**
- **operation** (*str*) – either
  - "trigger", where the last transform is a `Condition`

  and a trigger is returned to sender only when the return value of the transformation changes, or * "stream", where each result of the transformation is returned to sender
- **return_id**
- **return_ip**
- **return_port**
- **return_key**
- **router_port** (*None, int*) – If not `None` (default), spawn the node with a route port to receieve
- **stage_block**
- **value_subset** (*str*) – Optional - subset a value from from a dict/list sent to `l_process()`
- ****kwargs**

Methods:

`noop`()	
`l_process`(value)	

noop()

l_process(*value*)

15.3 free_water

Classes:

`Free_Water`([stage_block, current_trial, ...])	Randomly light up one of the ports, then dispense water when the subject pokes there

class **Free_Water**(*stage_block=None*, *current_trial=0*, *reward=50*, *allow_repeat=False*, ***kwargs*)

Bases: `autopilot.tasks.task.Task`

Randomly light up one of the ports, then dispense water when the subject pokes there

Two stages:

- waiting for response, and

- reporting the response afterwards

Variables

- **target** ('L', 'C', 'R') – The correct port
- **trial_counter** (itertools.count) – Counts trials starting from current_trial specified as argument
- **triggers** (*dict*) – Dictionary mapping triggered pins to callable methods.
- **num_stages** (*int*) – number of stages in task (2)
- **stages** (itertools.cycle) – iterator to cycle indefinitely through task stages.

Parameters

- **stage_block** (threading.Event) – used to signal to the carrying Pilot that the current trial stage is over
- **current_trial** (*int*) – If not zero, initial number of *trial_counter*
- **reward** (*int*) – ms to open solenoids
- **allow_repeat** (*bool*) – Whether the correct port is allowed to repeat between trials
- ****kwargs**

Attributes:

STAGE_NAMES
PARAMS
DATA
HARDWARE
PLOT

Classes:

TrialData()

Methods:

water(*args, **kwargs)	First stage of task - open a port if it's poked.
response()	Just have to alert the Terminal that the current trial has ended and turn off any lights.
end()	When shutting down, release all hardware objects and turn LEDs off.

STAGE_NAMES = ['water', 'response']

PARAMS = OrderedDict([('reward', {'tag': 'Reward Duration (ms)', 'type': 'int'}), ('allow_repeat', {'tag': 'Allow Repeated Ports?', 'type': 'bool'})])

DATA = { 'target': {'plot': 'target', 'type': 'S1'}, 'timestamp': {'type': 'S26'}, 'trial_num': {'type': 'i32'}}

class TrialData

Bases: `tables.description.IsDescription`

Attributes:

`columns`

columns = { 'target': StringCol(itemsize=1, shape=(), dflt=b'', pos=None), 'timestamp': StringCol(itemsize=26, shape=(), dflt=b'', pos=None), 'trial_num': Int32Col(shape=(), dflt=0, pos=None)}

HARDWARE = { 'LEDS': { 'C': <class 'autopilot.hardware.gpio.LED_RGB'>, 'L': <class 'autopilot.hardware.gpio.LED_RGB'>, 'R': <class 'autopilot.hardware.gpio.LED_RGB'>}, 'POKES': { 'C': <class 'autopilot.hardware.gpio.Digital_In'>, 'L': <class 'autopilot.hardware.gpio.Digital_In'>, 'R': <class 'autopilot.hardware.gpio.Digital_In'>}, 'PORTS': { 'C': <class 'autopilot.hardware.gpio.Solenoid'>, 'L': <class 'autopilot.hardware.gpio.Solenoid'>, 'R': <class 'autopilot.hardware.gpio.Solenoid'>}}

PLOT = {'data': {'target': 'point'}}

water(**args*, ***kwargs*)

First stage of task - open a port if it's poked.

Returns

Data dictionary containing:

```
'target': ('L', 'C', 'R') - correct response
'timestamp': isoformatted timestamp
'trial_num': number of current trial
```

Return type dict

response()

Just have to alert the Terminal that the current trial has ended and turn off any lights.

end()

When shutting down, release all hardware objects and turn LEDs off.

15.4 graduation

Object that implement Graduation criteria to move between different tasks in a protocol.

Classes:

Graduation()	Base Graduation object.
Accuracy([threshold, window])	Graduate stage based on percent accuracy over some window of trials.
NTrials(n_trials[, current_trial])	Graduate after doing n trials

class Graduation

Bases: `object`

Base Graduation object.

All Graduation objects need to populate PARAMS, COLS, and define an *update* method.

Attributes:

PARAMS	list of parameters to be defined
COLS	list of any data columns that this object should be given.

Methods:

update(row)	**Parameters** **:class:`~tables.tableextension.Row`** – Trial row

PARAMS = []

list of parameters to be defined

Type list

COLS = []

list of any data columns that this object should be given.

Type list

update(*row*)

Parameters **:class:`~tables.tableextension.Row`** – Trial row

class Accuracy(*threshold=0.75*, *window=500*, ***kwargs*)

Bases: `autopilot.tasks.graduation.Graduation`

Graduate stage based on percent accuracy over some window of trials.

Parameters

- **threshold** (*float*) – Accuracy above this threshold triggers graduation
- **window** (*int*) – number of trials to consider in the past.
- ****kwargs** – should have 'correct' corresponding to the corrects/incorrects of the past.

Attributes:

PARAMS	list of parameters to be defined
COLS	list of any data columns that this object should be given.

Methods:

update(row)	Get 'correct' from the row object.

PARAMS = ['threshold', 'window']
list of parameters to be defined

Type list

COLS = ['correct']
list of any data columns that this object should be given.

Type list

update(*row*)
Get 'correct' from the row object. If this trial puts us over the threshold, return True, else False.

Parameters **row** (Row) – Trial row

Returns Did we graduate this time or not?

Return type bool

class NTrials(*n_trials*, *current_trial=0*, ***kwargs*)
Bases: *autopilot.tasks.graduation.Graduation*

Graduate after doing n trials

Variables **counter** (itertools.count) – Counts the trials.

Parameters

- **n_trials** (*int*) – Number of trials to graduate after
- **current_trial** (*int*) – If not starting from zero, start from here
- ****kwargs**

Attributes:

PARAMS	list of parameters to be defined

Methods:

update(row)	If we're past n_trials in this trial, return True, else False.

PARAMS = ['n_trials', 'current_trial']
list of parameters to be defined

Type list

update(*row*)
If we're past n_trials in this trial, return True, else False.

Parameters **row** – ignored

Returns Did we graduate or not?

Return type bool

15.5 nafc

Classes:

`Nafc`([stage_block, stim, reward, ...])	A Two-alternative forced choice task.

class Nafc(*stage_block=None*, *stim=None*, *reward=50*, *req_reward=False*, *punish_stim=False*, *punish_dur=100*, *correction=False*, *correction_pct=50.0*, *bias_mode=False*, *bias_threshold=20*, *stim_light=True*, ***kwargs*)

Bases: `autopilot.tasks.task.Task`

A Two-alternative forced choice task.

(can't have number as first character of class.)

Stages

- **request** - compute stimulus, set request trigger in center port.
- **discrim** - respond to input, set reward/punishment triggers on target/distractor ports
- **reinforcement** - deliver reward/punishment, end trial.

Variables

- **target** (*"L", "R"*) – Correct response
- **distractor** (*"L", "R"*) – Incorrect response
- **stim** – Current stimulus
- **response** (*"L", "R"*) – Response to discriminand
- **correct** (*0, 1*) – Current trial was correct/incorrect
- **correction_trial** (*bool*) – If using correction trials, last trial was a correction trial
- **trial_counter** (`itertools.count`) – Which trial are we on?
- **discrim_playing** (*bool*) – Is the stimulus playing?
- **bailed** (*0, 1*) – Subject answered before stimulus was finished playing.
- **current_stage** (*int*) – As each stage is reached, update for asynchronous event reference

Parameters

- **stage_block** (`threading.Event`) – Signal when task stages complete.
- **stim** (*dict*) –

  Stimuli like:

  ```
  "sounds": {
      "L": [{"type": "Tone", ...}],
      "R": [{"type": "Tone", ...}]
  }
  ```

- **reward** (*float*) – duration of solenoid open in ms
- **req_reward** (*bool*) – Whether to give a water reward in the center port for requesting trials
- **punish_stim** (*bool*) – Do a white noise punishment stimulus

- **punish_dur** (*float*) – Duration of white noise in ms
- **correction** (*bool*) – Should we do correction trials?
- **correction_pct** (*float*) – (0-1), What proportion of trials should randomly be correction trials?
- **bias_mode** (*False, "thresholded_linear"*) – False, or some bias correction type (see `managers.Bias_Correction`)
- **bias_threshold** (*float*) – If using a bias correction mode, what threshold should bias be corrected for?
- **current_trial** (*int*) – If starting at nonzero trial number, which?
- **stim_light** (*bool*) – Should the LED be turned blue while the stimulus is playing?
- ****kwargs**

Attributes:

`STAGE_NAMES`
`PARAMS`
`PLOT`
`HARDWARE`

Classes:

`TrialData`()

Methods:

`request`(*args, **kwargs)	Stage 0: compute stimulus, set request trigger in center port.
`discrim`(*args, **kwargs)	Stage 1: respond to input, set reward/punishment triggers on target/distractor ports
`reinforcement`(*args, **kwargs)	Stage 2 - deliver reward/punishment, end trial.
`punish`()	Flash lights, play punishment sound if set
`respond`(pin)	Set self.response
`stim_start`()	mark discrim_playing = true
`stim_end`()	called by stimulus callback
`flash_leds`()	flash lights for punish_dir

STAGE_NAMES = ['request', 'discrim', 'reinforcement']

`PARAMS = OrderedDict([('reward', {'tag': 'Reward Duration (ms)', 'type': 'int'}), ('req_reward', {'tag': 'Request Rewards', 'type': 'bool'}), ('punish_stim', {'tag': 'White Noise Punishment', 'type': 'bool'}), ('punish_dur', {'tag': 'Punishment Duration (ms)', 'type': 'int'}), ('correction', {'tag': 'Correction Trials', 'type': 'bool'}), ('correction_pct', { 'depends': {'correction': True}, 'tag': '% Correction Trials', 'type': 'int'}), ('bias_mode', { 'tag': 'Bias Correction Mode', 'type': 'list', 'values': { 'None': 0, 'Proportional': 1, 'Thresholded Proportional': 2}}), ('bias_threshold', { 'depends': {'bias_mode': 2}, 'tag': 'Bias Correction Threshold (%)', 'type': 'int'}), ('stim', {'tag': 'Sounds', 'type': 'sounds'})])`

`PLOT = { 'chance_bar': True, 'data': {'correct': 'rollmean', 'response': 'segment', 'target': 'point'}, 'roll_window': 50}`

`class TrialData`

Bases: `tables.description.IsDescription`

Attributes:

`columns`

`columns = { 'DC_timestamp': StringCol(itemsize=26, shape=(), dflt=b'', pos=None), 'RQ_timestamp': StringCol(itemsize=26, shape=(), dflt=b'', pos=None), 'bailed': Int32Col(shape=(), dflt=0, pos=None), 'correct': Int32Col(shape=(), dflt=0, pos=None), 'correction': Int32Col(shape=(), dflt=0, pos=None), 'response': StringCol(itemsize=1, shape=(), dflt=b'', pos=None), 'target': StringCol(itemsize=1, shape=(), dflt=b'', pos=None), 'trial_num': Int32Col(shape=(), dflt=0, pos=None)}`

`HARDWARE = { 'LEDS': {'C': 'LED_RGB', 'L': 'LED_RGB', 'R': 'LED_RGB'}, 'POKES': {'C': 'Digital_In', 'L': 'Digital_In', 'R': 'Digital_In'}, 'PORTS': {'C': 'Solenoid', 'L': 'Solenoid', 'R': 'Solenoid'}}`

request(**args*, ***kwargs*)

Stage 0: compute stimulus, set request trigger in center port.

Returns

With fields:

```
{
'target': self.target,
'trial_num' : self.current_trial,
'correction': self.correction_trial,
'type': stimulus type,
**stim.PARAMS
}
```

Return type data (dict)

discrim(**args*, ***kwargs*)

Stage 1: respond to input, set reward/punishment triggers on target/distractor ports

Returns

With fields:: { 'RQ_timestamp': datetime.datetime.now().isoformat(), 'trial_num': self.current_trial, }

Return type data (dict)

reinforcement(**args*, ***kwargs*)
Stage 2 - deliver reward/punishment, end trial.

Returns

With fields:

```
 {
'DC_timestamp': datetime.datetime.now().isoformat(),
'response': self.response,
'correct': self.correct,
'bailed': self.bailed,
'trial_num': self.current_trial,
'TRIAL_END': True
}
```

Return type data (dict)

punish()
Flash lights, play punishment sound if set

respond(*pin*)
Set self.response

Parameters **pin** – Pin to set response to

stim_start()
mark discrim_playing = true

stim_end()
called by stimulus callback

set outside lights blue

flash_leds()
flash lights for punish_dir

CHAPTER

SIXTEEN

TRANSFORMATIONS

Data transformations.

Composable transformations from one representation of data to another. Used as the lubricant and glue between hardware objects. Some hardware objects disagree about the way information should be represented – eg. cameras are very partial to letting position information remain latent in a frame of a video, but some other object might want the actual `[x,y]` coordinates. Transformations help negotiate (but don't resolve their irreparably different worldviews :( )

Transformations are organized by modality, but this API is quite immature.

Transformations have a `process` method that accepts and returns a single object. They must also define the format of their inputs and outputs (`format_in` and `format_out`). That API is also a sketch.

The *__add__()* method allows transforms to be combined, eg.:

```
from autopilot import transform as t
transform_me = t.Image.DLC('model_directory')
transform_me += t.selection.DLCSlice('point')
transform_me.process(frame)
# ... etcetera
```

Todo: This is a first draft of this module and it purely synchronous at the moment. It will be expanded to … * support multiple asynchronous processing rhythms * support automatic value coercion * make recursion checks – make sure a child hasn't already been added to a processing chain. * idk participate at home! list your own shortcomings of this module, don't be shy it likes it.

Functions:

`make_transform`(transforms)	Make a transform from a list of iterator specifications.

make_transform(*transforms: List[dict]*) → *autopilot.transform.transforms.Transform*

Make a transform from a list of iterator specifications.

Parameters **transforms** (*list*) –

A list of `Transform` s and parameterizations in the form:

```
[
    {'transform': Transform,
    'args': (arg1, arg2,), # optional
    'kwargs': {'key1':'val1', ...}, # optional
    {'transform': ...}
]
```

Returns `Transform`

Data transformations.

Experimental module.

Reusable transformations from one representation of data to another. eg. converting frames of a video to locations of objects, or locations of objects to area labels

---

Todo: This is a preliminary module and it purely synchronous at the moment. It will be expanded to … * support multiple asynchronous processing rhythms * support automatic value coercion

The following design features need to be added * recursion checks – make sure a child hasn't already been added to a processing chain.

Classes:

TransformRhythm(value)	**ivar ~TransformRhythm.FIFO** First-in-first-out, process inputs as they are received, potentially slowing down the transformation pipeline
Transform(rhythm, *args, **kwargs)	Metaclass for data transformations

class TransformRhythm(*value*)

Bases: `enum.Enum`

Variables

- **FIFO** – First-in-first-out, process inputs as they are received, potentially slowing down the transformation pipeline
- **FILO** – First-in-last-out, process the most recent input, ignoring previous (lossy transformation)

Attributes:

FIFO
FILO

FIFO = 1

FILO = 2

class Transform(*rhythm: autopilot.transform.transforms.TransformRhythm = <TransformRhythm.FILO: 2>, *args, **kwargs*)

Bases: `object`

Metaclass for data transformations

Each subclass should define the following

- *process()* - a method that takes the input of the transoformation as its single argument and returns the transformed output
- *format_in* - a *dict* that specifies the input format

- *format_out* - a *dict* that specifies the output format

Parameters **rhythm** (*TransformRhythm*) – A rhythm by which the transformation object processes its inputs

Variables **(class** (*child*) – *Transform*): Another Transform object chained after this one

Attributes:

rhythm	
format_in	
format_out	
parent	If this Transform is in a chain of transforms, the transform that precedes it

Methods:

process(input)	
reset()	If a transformation is stateful, reset state.
check_compatible(child)	Check that this Transformation's *format_out* is compatible with another's *format_in*
__add__(other)	Add another Transformation in the chain to make a processing pipeline

property rhythm: autopilot.transform.transforms.TransformRhythm

property format_in: dict

property format_out: dict

property parent: Optional[autopilot.transform.transforms.Transform]

If this Transform is in a chain of transforms, the transform that precedes it

Returns *Transform*, None if no parent.

process(*input*)

reset()

If a transformation is stateful, reset state.

check_compatible(*child: autopilot.transform.transforms.Transform*)

Check that this Transformation's *format_out* is compatible with another's *format_in*

Todo: Check for types that can be automatically coerced into one another and set _coercion to appropriate function

Parameters **child** (*Transform*) – Transformation to check compatibility

Returns bool

__add__(*other*)

Add another Transformation in the chain to make a processing pipeline

Parameters **other** (`Transformation`) – The transformation to be chained

16.1 Coercion

placeholder. . . objects to make type and shape coercion seamless. . . .

16.2 Geometry

Classes:

Distance(pairwise, n_dim, metric, . . .)	Given an n_samples x n_dimensions array, compute pairwise or mean distances
Angle([abs, degrees])	Get angle between line formed by two points and horizontal axis
IMU_Orientation(use_kalman, invert_gyro, . . .)	Compute absolute orientation (roll, pitch) from accelerometer and gyroscope measurements (eg from *hardware.i2c.I2C_9DOF*)
Rotate([dims, rotation_type, degrees, . . .])	Rotate in 3 dimensions using `scipy.spatial.transform.Rotation`
Spheroid([target])	Fit and transform 3d coordinates according to some spheroid.

Functions:

_ellipsoid_func(fit, a, b, c, x, y, z)	Ellipsoid equation for use with `Ellipsoid.fit()`

class Distance(*pairwise: bool = False, n_dim: int = 2, metric: str = 'euclidean', squareform: bool = True, *args, **kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Given an n_samples x n_dimensions array, compute pairwise or mean distances

Parameters

- **pairwise** (*bool*) – If False (default), return mean distance. if True, return all distances
- **n_dim** (*int*) – number of dimensions (input array will be filtered like `input[:,0:n_dim]`
- **metric** (*str*) – any metric acceptable to :func:`scipy.spatial.distance.pdist
- **squareform** (*bool*) – if pairwise is True, if True return square distance matrix, otherwise return compressed distance matrix (dist(X[i], X[j] = y[i*j])
- ***args**
- ****kwargs**

Attributes:

format_in
format_out

Methods:

process(input)

format_in = {'type': <class 'numpy.ndarray'>}

format_out = {'type': <class 'numpy.ndarray'>}

process(*input: numpy.ndarray*)

class Angle(*abs=True*, *degrees=True*, **args*, ***kwargs*)

Bases: *autopilot.transform.transforms.Transform*

Get angle between line formed by two points and horizontal axis

Attributes:

format_in
format_out

Methods:

process(input)

format_in = {'type': <class 'numpy.ndarray'>}

format_out = {'type': <class 'float'>}

process(*input*)

class IMU_Orientation(*use_kalman: bool = True*, *invert_gyro: bool = False*, **args*, ***kwargs*)

Bases: *autopilot.transform.transforms.Transform*

Compute absolute orientation (roll, pitch) from accelerometer and gyroscope measurements (eg from *hardware.i2c.I2C_9DOF*)

Uses a *timeseries.Kalman* filter, and implements [PPT+18] to fuse the sensors

Can be used with accelerometer data only, or with combined accelerometer/gyroscope data for greater accuracy

Parameters

- **invert_gyro** (*bool*) – if the gyroscope's orientation is inverted from accelerometer measurement, multiply gyro readings by -1 before using
- **use_kalman** (*bool*) – Whether to use kalman filtering (True, default), or return raw trigonometric transformation of accelerometer readings (if provided, gyroscope readings will be ignored)

Variables kalman (*transform.timeseries.Kalman*) – If `use_kalman == True` , the Kalman Filter.

References

[PPT+18] [ABCO15]

Methods:

`process`(accelgyro)	**Parameters accelgyro** (tuple, `numpy.ndarray`) – tuple of (accelerometer[x,y,z], gyro[x,y,z]) readings as arrays, or

process(*accelgyro: Union[Tuple[numpy.ndarray, numpy.ndarray], numpy.ndarray]*) → numpy.ndarray

Parameters accelgyro (tuple, `numpy.ndarray`) – tuple of (accelerometer[x,y,z], gyro[x,y,z]) readings as arrays, or an array of just accelerometer[x,y,z]

Returns filtered [roll, pitch] calculations in degrees

Return type `numpy.ndarray`

class Rotate(*dims='xyz'*, *rotation_type='euler'*, *degrees=True*, *inverse=''*, *rotation=None*, **args*, ***kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Rotate in 3 dimensions using `scipy.spatial.transform.Rotation`

Parameters

- **dims** (*"xyz"*) – string specifying which axes the rotation will be around, eg `"xy"` , `"xyz"`\`
- **rotation_type** (*str*) – Format of rotation input, must be one available to the `Rotation` class (but currently only euler angles are supported)
- **degrees** (*bool*) – whether to output rotation in degrees (True, default) or radians
- **inverse** (*"xyz"*) – dimensions in the "rotation" input to `Rotate.process()` to inverse before applying rotation
- **rotation** (tuple, list, `numpy.ndarray`, None) – If supplied, use the same rotation for all processed data. If None, `Rotate.process()` will expect a tuple of (data, rotation).

Methods:

`process`(input)	**Parameters input** (tuple, `numpy.ndarray`) – a tuple of (input[x,y,z], rotation[x,y,z]) where input is to be rotated

process(*input*)

Parameters input (tuple, `numpy.ndarray`) – a tuple of (input[x,y,z], rotation[x,y,z]) where input is to be rotated according to the axes in rotation (indicated in `Rotate.dims`). If only an input array is provided, a static rotation array must have been provided in the constructor (otherwise the most recent rotation will be used)

Returns `numpy.ndarray` - rotated input array

class Spheroid(*target=(1, 1, 1, 0, 0, 0)*, *source: tuple = (None, None, None, None, None, None)*, *fit: Optional[numpy.ndarray] = None*, **args*, ***kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Fit and transform 3d coordinates according to some spheroid.

Eg. for calibrating accelerometer readings by transforming them from their uncalibrated spheroid to the expected sphere with radius == 9.8m/s/s centered at (0,0,0).

Does not estimate/correct for rotation of the spheroid.

Examples

Calibrate an accelerometer by transforming readings to a 9.8-radius sphere centered at 0 >>> sphere = Spheroid(target=(9.8,9.8,9.8,0,0,0)) # take some readings… # imagine we're taking them from some sensor idk # say our sensor slightly exaggerates gravity in the z-axis… >>> readings = np.array((0.,0.,10.5)) # fit our object (need >>1 sample) >>> sphere.fit(readings) # transform to proper gravity >>> sphere.process(readings) [0., 0., 9.8]

Parameters

- **target** (*tuple*) – parameterization of spheroid to transform to, if none is passed, transform to unit circle centered at (0,0,0). parameterized as:

  ```
  (a, # radius of x dimension
  ```

  b, # radius of y dimension c, # radius of z dimension x, # x-offset y, # y-offset z) # z-offset
- **source** (*tuple*) – parameterization of spheroid to transform from in the same 6-tuple form as `target`, if None is passed, assume we will use `Spheroid.fit()`
- **fit** (None, `numpy.ndarray`) – Initialize with values to fit, if None assume fit will be called later.

References

- https://jekel.me/2020/Least-Squares-Ellipsoid-Fit/
- http://www.juddzone.com/ALGORITHMS/least_squares_3D_ellipsoid.html

Methods:

`fit`(points, **kwargs)	Fit a spheroid from a set of noisy measurements
`process`(input)	Transform input (x,y,z) points such that points in `source` are mapped to those in `target`
`generate`(n[, which, noise])	Generate random points from the ellipsoid

fit(*points*, ***kwargs*)

Fit a spheroid from a set of noisy measurements

updates the `_scale` and `_offset` private arrays used to manipulate input data

Note: It's usually important to pass `bounds` to `scipy.optimize.curve_fit()` !!! passed as a 2-tuple of `((min_a, min_b, ...), (max_a, max_b...))` In particular such that a, b, and c are positive. If no bounds are passed, assume at least that much.

Parameters

- **points** (`numpy.ndarray`) – (M, 3) array of points to fit
- ****kwargs** () – passed on to `scipy.optimize.curve_fit()`

Returns parameters of fit ellipsoid (a,b,c,x,y,z)

Return type tuple

`process`(*input: numpy.ndarray*)

Transform input (x,y,z) points such that points in `source` are mapped to those in `target`

Parameters **input** (`numpy.ndarray`) – x, y, and z coordinates

Returns coordinates transformed according to the spheroid requested

Return type `numpy.ndarray`

`generate`(*n: int, which: str = 'source', noise: float = 0*)

Generate random points from the ellipsoid

Parameters

- **n** (*int*) – number of points to generate
- **which** (*'str'*) – which spheroid to generate from? ('source' - default, or 'target')
- **noise** (*float*) – noise to add to points

Returns (n, 3) array of generated points

Return type `numpy.ndarray`

`_ellipsoid_func`(*fit, a, b, c, x, y, z*)

Ellipsoid equation for use with `Ellipsoid.fit()`

Parameters

- **fit** (`numpy.ndarray`) – (M, 3) array of x,y,z points to fit
- **a** (*float*) – X-scale parameter to fit
- **b** (*float*) – Y-scale parameter to fit
- **c** (*float*) – Z-scale parameter to fit
- **x** (*float*) – X-offset parameter to fit
- **y** (*float*) – Y-offset parameter to fit
- **z** (*float*) – Z-offset parameter to fit

Returns result of ellipsoid function, minimize parameters to == 1

Return type float

16.3 Image

Classes:

Image([shape])	Metaclass for transformations of images
DLC(model_dir, model_zoo, *args, **kwargs)	Do pose estimation with DeepLabCut-Live!!!!!

class Image(*shape=None*, **args*, ***kwargs*)

Bases: *autopilot.transform.transforms.Transform*

Metaclass for transformations of images

Attributes:

format_in
format_out
shape

property format_in: dict

property format_out: dict

property shape: Tuple[int, int]

class DLC(*model_dir: Optional[str] = None*, *model_zoo: Optional[str] = None*, **args*, ***kwargs*)

Bases: *autopilot.transform.image.Image*

Do pose estimation with `DeepLabCut-Live`!!!!!

Specify a `model_dir` (relative to `<BASEDIR>/dlc` or absolute) or a model name from the DLC model zoo.

All other args and kwargs are passed on to `dlclive.DLCLive`, see its documentation for details: https://github.com/DeepLabCut/DeepLabCut-live

Variables

- **model_type** (*str*, *'local' or 'zoo'*) – whether a directory (local) or a modelzoo name (zoo) was passed
- **live** (`dlclive.DLCLive`) – the DLCLive object

Must give either `model_dir` or `model_zoo`

Parameters

- **model_dir** (*str*) – directory of model, either absolute or relative to `<BASEDIR>/dlc`. if `None`, use `model_zoo`
- **model_zoo** (*str*) – name of modelzoo model. if `None`, use `model_dir`
- ***args** – passed to DLCLive and superclass
- ****kwargs** – passed to DLCLive and superclass

Methods:

process(input)	
list_modelzoo()	List available modelzoo model names in local deeplabcut version
import_dlc()	
create_modelzoo(model)	
load_model()	
export_model()	

Attributes:

model	
model_dir	
dlc_paths	paths used by dlc in manipulating/using models
dlc_dir	`{prefs.get('BASE_DIR')}/dlc` :returns: str
format_in	
format_out	

process(*input: numpy.ndarray*) → numpy.ndarray

property model: str

property model_dir: str

property dlc_paths: dict

paths used by dlc in manipulating/using models

- config: <model_dir>/config.yaml
- train_pose_cfg: <model_dir>/dlc-models/iteration-<n>/<name>/train/pose_cfg.yaml,
- export_pose_cfg: <model_dir>/exported-models/<name>/pose_cfg.yaml
- export_dir: <model_dir>/exported-models/<name>

Returns dict

property dlc_dir: str

`{prefs.get('BASE_DIR')}/dlc` :returns: str

classmethod list_modelzoo()

List available modelzoo model names in local deeplabcut version

Returns names of available modelzoo models

Return type list

import_dlc()

create_modelzoo(*model*)

`load_model()`

`export_model()`

`property format_in: dict`

`property format_out: dict`

16.4 Logical

Classes:

Condition([minimum, maximum, elementwise])	Compare the input against some condition
Compare(compare_fn, *args, **kwargs)	Compare processed values using some function that returns a boolean

class Condition(*minimum=None*, *maximum=None*, *elementwise=False*, **args*, ***kwargs*)

Bases: *autopilot.transform.transforms.Transform*

Compare the input against some condition

Parameters

- **minimum**
- **maximum**
- **elementwise** (*bool*) – if False, return True only if *all* values are within range. otherwise return bool for each tested value
- ***args**
- ****kwargs**

Methods:

process(input)

Attributes:

minimum
maximum
format_in
format_out

`process(input)`

`property minimum`

`property maximum`

`property format_in: dict`

property format_out: dict

class Compare(*compare_fn: callable*, **args*, ***kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Compare processed values using some function that returns a boolean

ie. process will `return compare_fn(*args)` from `process`.

it is expected that `input` will be an iterable with len > 1

Parameters

- **compare_fn** (*callable*) – Function used to compare the values given to `Compare.process()`
- ***args** ()
- ****kwargs** ()

Methods:

`process`(input)	

process(*input*)

16.5 Selection

Classes:

`Slice`(select, *args, **kwargs)	Generic selection processor
`DLCSlice`(select, tuple, list], ...)	Select x,y coordinates of `DLC` output based on the name of the tracked parts

class Slice(*select*, **args*, ***kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Generic selection processor

Parameters

- **select** (*slice, tuple[slice]*) – a slice or tuple of slices
- ***args**
- ****kwargs**

Attributes:

`format_in`
`format_out`

Methods:

check_slice(select)	
process(input)	

format_in = {'type': 'any'}

format_out = {'type': 'any'}

check_slice(*select*)

process(*input*)

class DLCSlice(*select: Union[str, tuple, list], min_probability: float = 0, *args, **kwargs*)

Bases: *autopilot.transform.selection.Slice*

Select x,y coordinates of *DLC* output based on the name of the tracked parts

note that min_probability is undefined when a list or tuple of part names are defined: the form of the returned array is ambiguous (how to tell which part is which when some might be excluded?)

Parameters

- **select** (*slice, tuple[slice]*) – a slice or tuple of slices
- ***args**
- ****kwargs**

Attributes:

format_in	
format_out	

Methods:

check_slice(select)	
process(input)	

format_in = { 'parent': <class 'autopilot.transform.image.DLC'>, 'type': <class 'numpy.ndarray'>}

format_out = {'type': <class 'numpy.ndarray'>}

check_slice(*select*)

process(*input: numpy.ndarray*)

16.6 Timeseries

Timeseries transformations, filters, etc.

Classes:

Filter_IIR([ftype, buffer_size, coef_type, axis])	Simple wrapper around `scipy.signal.iirfilter()`
Kalman(dim_state, dim_measurement, . . .)	Kalman filter!!!!!
Integrate([decay, dt_scale])	

class Filter_IIR(*ftype='butter'*, *buffer_size=256*, *coef_type='sos'*, *axis=0*, **args*, ***kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Simple wrapper around `scipy.signal.iirfilter()`

Creates a streaming filter – takes in single values, stores them, and uses them to filter future values.

Parameters

- **ftype** (*str*) – filter type, see `ftype` of `scipy.signal.iirfilter()` for available filters
- **buffer_size** (*int*) – number of samples to store when filtering
- **coef_type** (*{'ba', 'sos'}*) – type of filter coefficients to use (see `scipy.signal.sosfilt()` and `scipy.signal.lfilt()`)
- **axis** (*int*) – which axis to filter over? (default: 0 because when passing arrays to filter, want to filter samples over time)
- ****kwargs** – passed on to `scipy.signal.iirfilter()` , eg.
  - `N` - filter order
  - `Wn` - array or scalar giving critical frequencies
  - `btype` - type of band: `['bandpass', 'lowpass', 'highpass', 'bandstop']`

Variables

- **coefs** (*np.ndarray*) – filter coefficients, depending on `coef_type`
- **buffer** (*collections.deque*) – buffer of stored values to filter
- **coef_type** (*str*) – type of filter coefficients to use (see `scipy.signal.sosfilt()` and `scipy.signal.lfilt()`)
- **axis** (*int*) – which axis to filter over? (default: 0 because when passing arrays to filter, want to filter samples over time)
- **ftype** (*str*) – filter type, see `ftype` of `scipy.signal.iirfilter()` for available filters

Methods:

process(input)	Filter the new value based on the values stored in `Filter.buffer`

process(*input: float*)

Filter the new value based on the values stored in `Filter.buffer`

Parameters **input** (*float*) – new value to filter!

Returns the filtered value!

Return type float

class Kalman(*dim_state: int*, *dim_measurement: Optional[int] = None*, *dim_control: int = 0*, **args*, ***kwargs*)

Bases: `autopilot.transform.transforms.Transform`

Kalman filter!!!!!

Adapted from https://github.com/rlabbe/filterpy/blob/master/filterpy/kalman/kalman_filter.py simplified and optimized lovingly <3

Each of the arrays is named with its canonical letter and a short description, (eg. the x_state vector `x_state` is `self.x_state`

Parameters

- **dim_state** (*int*) – Dimensions of the state vector
- **dim_measurement** (*int*) – Dimensions of the measurement vector
- **dim_control** (*int*) – Dimensions of the control vector

Variables

- **x_state** (`numpy.ndarray`) – Current state vector
- **P_cov** (`numpy.ndarray`) – Uncertainty Covariance
- **Q_proc_var** (`numpy.ndarray`) – Process Uncertainty
- **B_control** (`numpy.ndarray`) – Control transition matrix
- **F_state_trans** (`numpy.ndarray`) – State transition matrix
- **H_measure** (`numpy.ndarray`) – Measurement function
- **R_measure_var** (`numpy.ndarray`) – Measurement uncertainty
- **M_proc_measure_xcor** (`numpy.ndarray`) – process-measurement cross correlation
- **z_measure** (`numpy.ndarray`) –
- **K** (`numpy.ndarray`) – Kalman gain
- **y** (`numpy.ndarray`) –
- **S** (`numpy.ndarray`) – System uncertainty
- **SI** (`numpy.ndarray`) – Inverse system uncertainty
- **x_prior** (`numpy.ndarray`) – State prior
- **P_prior** (`numpy.ndarray`) – Uncertainty prior
- **x_post** (`numpy.ndarray`) – State posterior probability
- **P_post** (`numpy.ndarray`) – Uncertainty posterior probability

References

Roger Labbe. "Kalman and Bayesian Filters in Python" - https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python Roger Labbe. "FilterPy" - https://github.com/rlabbe/filterpy

Methods:

`_init_arrays`([state])	Initialize the arrays!
`predict`([u, B, F, Q])	Predict next x_state (prior) using the Kalman filter x_state propagation equations.
`update`(z[, R, H])	Add a new measurement (z_measure) to the Kalman filter.
`_reshape_z`(z, dim_z, ndim)	ensure z is a (dim_z, 1) shaped vector
`process`(z, **kwargs)	Call predict and update, passing the relevant kwargs
`residual_of`(z)	Returns the residual for the given measurement (z_measure).
`measurement_of_state`(x)	Helper function that converts a x_state into a measurement.

Attributes:

`alpha`	Fading memory setting.

`_init_arrays`(*state=None*)

Initialize the arrays!

`predict`(*u=None*, *B=None*, *F=None*, *Q=None*)

Predict next x_state (prior) using the Kalman filter x_state propagation equations.

Update our state and uncertainty priors, `x_prior` and `P_prior`

u [np.array, default 0] Optional control vector.

B [np.array(dim_state, dim_u), or None] Optional control transition matrix; a value of None will cause the filter to use *self.B_control*.

F [np.array(dim_state, dim_state), or None] Optional x_state transition matrix; a value of None will cause the filter to use *self.F_state_trans*.

Q [np.array(dim_state, dim_state), scalar, or None] Optional process noise matrix; a value of None will cause the filter to use *self.Q_proc_var*.

`update`(*z: numpy.ndarray*, *R=None*, *H=None*) → numpy.ndarray

Add a new measurement (z_measure) to the Kalman filter.

If z_measure is None, nothing is computed. However, x_post and P_post are updated with the prior (x_prior, P_prior), and self.z_measure is set to None.

Parameters

- **z** (`numpy.ndarray`) – measurement for this update. z_measure can be a scalar if dim_measurement is 1, otherwise it must be convertible to a column vector.

  If you pass in a value of H_measure, z_measure must be a column vector the of the correct size.

- **R** (`numpy.ndarray`, int, None) – Optionally provide R_measure_var to override the measurement noise for this one call, otherwise self.R_measure_var will be used.

- **H** (numpy.ndarray, None) – Optionally provide H_measure to override the measurement function for this one call, otherwise self.H_measure will be used.

_reshape_z(*z*, *dim_z*, *ndim*)
ensure z is a (dim_z, 1) shaped vector

process(*z*, ***kwargs*)
Call predict and update, passing the relevant kwargs

Parameters

- **z** ()
- ****kwargs** ()

Returns self.x_state

Return type np.ndarray

residual_of(*z*)
Returns the residual for the given measurement (z_measure). Does not alter the x_state of the filter.

measurement_of_state(*x*)
Helper function that converts a x_state into a measurement.

x [np.array] kalman x_state vector

z_measure [(dim_measurement, 1): array_like] measurement for this update. z_measure can be a scalar if dim_measurement is 1, otherwise it must be convertible to a column vector.

property alpha
Fading memory setting. 1.0 gives the normal Kalman filter, and values slightly larger than 1.0 (such as 1.02) give a fading memory effect - previous measurements have less influence on the filter's estimates. This formulation of the Fading memory filter (there are many) is due to Dan Simon [1]_.

class Integrate(*decay=1*, *dt_scale=False*, **args*, ***kwargs*)
Bases: autopilot.transform.transforms.Transform

Methods:

process(input)	

process(*input*)

16.7 Units

For converting between things that are the same thing but have different numbers and shapes

Classes:

Rescale(in_range, float] =, out_range, float] =)	Rescale values from one range to another
Colorspaces(value)	An enumeration.
Color(convert_from, convert_to[, output_scale])	Convert colors using the colorsys module!!

class Rescale(*in_range: Tuple[float, float] = (0, 1)*, *out_range: Tuple[float, float] = (0, 1)*, *clip=False*, **args*, ***kwargs*)
Bases: autopilot.transform.transforms.Transform

Rescale values from one range to another

Attributes:

format_in	
format_out	

Methods:

process(input)	Subtract input minimum, multiple by output/input size ratio, add output minimum

format_in = { 'type': (<class 'numpy.ndarray'>, <class 'float'>, <class 'int'>, <class 'tuple'>, <class 'list'>)}

format_out = {'type': <class 'numpy.ndarray'>}

process(*input*)
Subtract input minimum, multiple by output/input size ratio, add output minimum

class Colorspaces(*value*)
Bases: `enum.Enum`

An enumeration.

Attributes:

HSV
RGB
YIQ
HLS

HSV = 1

RGB = 2

YIQ = 3

HLS = 4

class Color(*convert_from: autopilot.transform.units.Colorspaces = <Colorspaces.HSV: 1>, convert_to: autopilot.transform.units.Colorspaces = <Colorspaces.RGB: 2>, output_scale=255, *args, **kwargs*)
Bases: `autopilot.transform.transforms.Transform`

Convert colors using the colorsys module!!

Note: All inputs must be scaled (0,1) and all outputs will be (0,1)

Attributes:

format_in
format_out
CONVERSIONS

Methods:

process(input, *args)

format_in = {'type': <class 'tuple'>}

format_out = {'type': <class 'tuple'>}

CONVERSIONS = { <Colorspaces.RGB: 2>: { <Colorspaces.HSV: 1>: <function rgb_to_hsv at 0x7fce8b04d830>, <Colorspaces.YIQ: 3>: <function rgb_to_yiq at 0x7fce8b04bef0>, <Colorspaces.HLS: 4>: <function rgb_to_hls at 0x7fce8b04d680>}, <Colorspaces.HSV: 1>: { <Colorspaces.RGB: 2>: <function hsv_to_rgb at 0x7fce8b04d8c0>}, <Colorspaces.YIQ: 3>: { <Colorspaces.RGB: 2>: <function yiq_to_rgb at 0x7fce8b04be60>}, <Colorspaces.HLS: 4>: { <Colorspaces.RGB: 2>: <function hls_to_rgb at 0x7fce8b04d710>}}

process(*input*, **args*)

CHAPTER

SEVENTEEN

VIZ

17.1 trial_viewer

Tools to visulize data after collection.

> **Warning:** this module is unfinished, so it is undocumented.

Functions:

`load_subject_data`(data_dir, subject_name[, ...])	
`load_subject_dir`(data_dir[, steps, grad, which])	**Parameters** • **data_dir** (*str*) – A path to a directory with `Subject` style hdf5 files
`step_viewer`(grad_data)	
`trial_viewer`(step_data[, roll_type, ...])	**Parameters** • **bar**

load_subject_data(*data_dir*, *subject_name*, *steps=True*, *grad=True*)

load_subject_dir(*data_dir*, *steps=True*, *grad=True*, *which=None*)

Parameters

- **data_dir** (*str*) – A path to a directory with `Subject` style hdf5 files
- **steps** (*bool*) – Whether to return full trial-level data for each step
- **grad** (*bool*) – Whether to return summarized step graduation data.
- **which** (*list*) – A list of subjects to subset the loaded subjects to

step_viewer(*grad_data*)

trial_viewer(*step_data*, *roll_type='ewm'*, *roll_span=100*, *bar=False*)

Parameters

- **bar**
- **roll_span**
- **roll_type**
- **step_data**

17.2 psychometric

Functions:

`calc_psychometric`(data, var_x[, var_y])	Calculate a psychometric curve (logistic regression of var_y on var_x)
`plot_psychometric`(subject_protocols)	Plot psychometric curves for selected subjects, steps, and variables

calc_psychometric(*data*, *var_x*, *var_y='response'*)

Calculate a psychometric curve (logistic regression of var_y on var_x)

Parameters

- **data** (`pandas.DataFrame`) – Subject data
- **var_x** (*str*) – name of column to use as the discriminand
- **var_y** (*str*) – name of the column for the response, usually 'response'

Returns parameters for logistic function

Return type params (tup)

plot_psychometric(*subject_protocols*)

Plot psychometric curves for selected subjects, steps, and variables

Typically called by `Terminal.plot_psychometric()`.

Parameters **subject_protocols** (*list*) – A list of tuples, each with

- subject_id (str)
- step_name (str)
- variable (str)

Returns `altair.Chart`

CHAPTER

EIGHTEEN

UTILS

Utility functions!

18.1 Common Utils

Generic utility functions that are used in multiple places in the library that for now don't have a clear other place to be

Functions:

list_classes(module)	List all classes within a module/package without importing by parsing the syntax tree directly with `ast`.
find_class(cls_str)	Given a full package.module.ClassName string, return the relevant class
recurse_subclasses(cls[, leaves_only])	Given some class, find its subclasses recursively
list_subjects([pilot_db])	Given a dictionary of a pilot_db, return the subjects that are in it.
load_pilotdb([file_name, reverse])	Try to load the file_db
coerce_discrete(df, col[, mapping])	Coerce a discrete/string column of a pandas dataframe into numeric values
find_key_recursive(key, dictionary)	Find all instances of a key in a dictionary, recursively.
find_key_value(dicts, key, value[, single])	Find an entry in a list of dictionaries where dict[key] == value.

Classes:

ReturnThread([group, target, name, args, ...])	Thread whose .join() method returns the value from the function thx to https://stackoverflow.com/a/6894023
NumpyEncoder(*[, skipkeys, ensure_ascii, ...])	Allow json serialization of objects containing numpy arrays.
NumpyDecoder(*args, **kwargs)	Allow json deserialization of objects containing numpy arrays.

list_classes(*module*) → List[Tuple[str, str]]

List all classes within a module/package without importing by parsing the syntax tree directly with `ast`.

Parameters **module** (*module, str*) – either the imported module to be queried, or its name as a string. if passed a string, attempt to import with `importlib.import_module()`

Returns list of tuples [('ClassName', 'module1.module2.ClassName')] a la `inspect.getmembers()`

find_class(*cls_str: str*)

Given a full package.module.ClassName string, return the relevant class

Parameters **cls_str** (*str*) – a full package.module.ClassName string, like `'autopilot.hardware.Hardware'`

Returns the class indicated by `cls_str`

recurse_subclasses(*cls*, *leaves_only=False*) → list

Given some class, find its subclasses recursively

See: https://stackoverflow.com/a/17246726/13113166

Parameters **leaves_only** (*bool*) – If True, only include classes that have no further subclasses, if False (default), return all subclasses.

Returns list of subclasses

class **ReturnThread**(*group=None*, *target=None*, *name=None*, *args=()*, *kwargs={}*, *Verbose=None*)

Bases: `threading.Thread`

Thread whose .join() method returns the value from the function thx to https://stackoverflow.com/a/6894023

This constructor should always be called with keyword arguments. Arguments are:

group should be None; reserved for future extension when a ThreadGroup class is implemented.

target is the callable object to be invoked by the run() method. Defaults to None, meaning nothing is called.

name is the thread name. By default, a unique name is constructed of the form "Thread-N" where N is a small decimal number.

args is the argument tuple for the target invocation. Defaults to ().

kwargs is a dictionary of keyword arguments for the target invocation. Defaults to {}.

If a subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.

Methods:

run()	Method representing the thread's activity.
join([timeout])	Wait until the thread terminates.

run()

Method representing the thread's activity.

You may override this method in a subclass. The standard run() method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

join(*timeout=None*)

Wait until the thread terminates.

This blocks the calling thread until the thread whose join() method is called terminates – either normally or through an unhandled exception or until the optional timeout occurs.

When the timeout argument is present and not None, it should be a floating point number specifying a timeout for the operation in seconds (or fractions thereof). As join() always returns None, you must call is_alive() after join() to decide whether a timeout happened – if the thread is still alive, the join() call timed out.

When the timeout argument is not present or None, the operation will block until the thread terminates.

A thread can be join()ed many times.

join() raises a RuntimeError if an attempt is made to join the current thread as that would cause a deadlock. It is also an error to join() a thread before it has been started and attempts to do so raises the same exception.

`list_subjects`(*pilot_db=None*)
Given a dictionary of a pilot_db, return the subjects that are in it.

Parameters pilot_db (*dict*) – a pilot_db. if None tried to load pilot_db with **:method:`.load_pilotdb`**

Returns a list of currently active subjects

Return type subjects (list)

`load_pilotdb`(*file_name=None*, *reverse=False*)
Try to load the file_db

Parameters

- **reverse**
- **file_name**

Returns:

`coerce_discrete`(*df*, *col*, *mapping={'L': 0, 'R': 1}*)
Coerce a discrete/string column of a pandas dataframe into numeric values

Default is to map 'L' to 0 and 'R' to 1 as in the case of Left/Right 2AFC tasks

Parameters

- **df** (`pandas.DataFrame`) – dataframe with the column to transform
- **col** (*str*) – name of column
- **mapping** (*dict*) – mapping of strings to numbers

Returns transformed dataframe

Return type df (`pandas.DataFrame`)

`find_key_recursive`(*key*, *dictionary*)
Find all instances of a key in a dictionary, recursively.

Parameters

- **key**
- **dictionary**

Returns list

`find_key_value`(*dicts: List[dict]*, *key: str*, *value: str*, *single=True*)
Find an entry in a list of dictionaries where dict[key] == value.

Parameters

- **dicts** ()
- **key** ()
- **value** ()
- **single** (*bool*) – if `True` (default), raise an exception if multiple results are matched

class NumpyEncoder(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: `json.encoder.JSONEncoder`

Allow json serialization of objects containing numpy arrays.

Use like `json.dump(obj, fp, cls=NumpyEncoder)`

Deserialize with `NumpyDecoder`

References

- https://stackoverflow.com/a/49677241/13113166
- https://github.com/mpld3/mpld3/issues/434#issuecomment-340255689
- https://gist.github.com/massgh/297a73f2dba017ffd28dbc34b9a40e90

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, float or None. If skipkeys is True, such items are simply skipped.

If ensure_ascii is true, the output is guaranteed to be str objects with all incoming non-ASCII characters escaped. If ensure_ascii is false, the output can contain non-ASCII characters.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation.

If specified, separators should be an (item_separator, key_separator) tuple. The default is (', ', ': ') if *indent* is `None` and (',', ': ') otherwise. To get the most compact JSON representation, you should specify (',', ':') to eliminate whitespace.

If specified, default is a function that gets called for objects that can't otherwise be serialized. It should return a JSON encodable version of the object or raise a `TypeError`.

Methods:

`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`).

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:

```python
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 NumpyDecoder(*args*, ***kwargs*)

Bases: json.decoder.JSONDecoder

Allow json deserialization of objects containing numpy arrays.

Use like json.load(fp, cls=NumpyDecoder)

Serialize with *NumpyEncoder*

References

- https://stackoverflow.com/a/49677241/13113166
- https://github.com/mpld3/mpld3/issues/434#issuecomment-340255689
- https://gist.github.com/massgh/297a73f2dba017ffd28dbc34b9a40e90

object_hook, if specified, will be called with the result of every JSON object decoded and its return value will be used in place of the given dict. This can be used to provide custom deserializations (e.g. to support JSON-RPC class hinting).

object_pairs_hook, if specified will be called with the result of every JSON object decoded with an ordered list of pairs. The return value of object_pairs_hook will be used instead of the dict. This feature can be used to implement custom decoders. If object_hook is also defined, the object_pairs_hook takes priority.

parse_float, if specified, will be called with the string of every JSON float to be decoded. By default this is equivalent to float(num_str). This can be used to use another datatype or parser for JSON floats (e.g. decimal.Decimal).

parse_int, if specified, will be called with the string of every JSON int to be decoded. By default this is equivalent to int(num_str). This can be used to use another datatype or parser for JSON integers (e.g. float).

parse_constant, if specified, will be called with one of the following strings: -Infinity, Infinity, NaN. This can be used to raise an exception if invalid JSON numbers are encountered.

If strict is false (true is the default), then control characters will be allowed inside strings. Control characters in this context are those with character codes in the 0-31 range, including '\t' (tab), '\n', '\r' and '\0'.

Methods:

object_hook(obj)	

object_hook(*obj*)

18.2 GUI Invoker

Classes:

InvokeEvent(fn, *args, **kwargs)	Sends signals to the main QT thread from spawned message threads
Invoker	Wrapper that calls an evoked event made by *InvokeEvent*

Functions:

get_invoker()

class InvokeEvent(*fn*, **args*, ***kwargs*)

Bases: `PySide2.QtCore.QEvent`

Sends signals to the main QT thread from spawned message threads

See stackoverflow

Accepts a function, its args and kwargs and wraps them as a `QtCore.QEvent`

Attributes:

EVENT_TYPE

EVENT_TYPE = PySide2.QtCore.QEvent.Type(65533)

class Invoker

Bases: `PySide2.QtCore.QObject`

Wrapper that calls an evoked event made by *InvokeEvent*

Methods:

event(event)	**Parameters** event

Attributes:

staticMetaObject

event(*event*)

Parameters event

staticMetaObject = <PySide2.QtCore.QMetaObject object at 0x7fce75968f00>

get_invoker()

18.3 Plugins

Utility functions for handling plugins, eg. importing, downloading, listing, confirming, etc.

Functions:

import_plugins([plugin_dir])	Import all plugins in the plugin (or supplied) directory.
unload_plugins()	Un-import imported plugins (mostly for testing purposes)
list_wiki_plugins()	List plugins available on the wiki using *utils.wiki.ask()*

import_plugins(*plugin_dir: Optional[pathlib.Path] = None*) → dict

Import all plugins in the plugin (or supplied) directory.

There is no specific form for a plugin at the moment, so this function will recursively import all modules and packages within the directory.

Plugins can then be accessed by the *get()* registry functions.

Parameters plugin_dir (None, `pathlib.Path`) – Directory to import. if `None` (default), use `prefs.get('PLUGINDIR')`.

Returns of imported objects with form {"class_name": class_object}

Return type dict

unload_plugins()

Un-import imported plugins (mostly for testing purposes)

list_wiki_plugins()

List plugins available on the wiki using *utils.wiki.ask()*

Returns {'plugin_name': {'plugin_prop':'prop_value',. . . }

Return type dict

18.4 Registry

Registry for programmatic access to autopilot classes and plugins

When possible, rather than importing and using an object directly, access it using the `get` methods in this module. This makes it possible for plugins to be integrated across the system.

Classes:

REGISTRIES(value)	Types of registries that are currently supported, ie.

Functions:

get(base_class[, class_name, plugins, ast, . . .])	Get an autopilot object.
get_names(base_class[, class_name, plugins, . . .])	*get()* but return a list of object names instead of the objects themselves
get_hardware([class_name, plugins, ast])	Get a hardware class by name.
get_task([class_name, plugins, ast])	Get a task class by name.

class REGISTRIES(*value*)

Bases: `str`, `enum.Enum`

Types of registries that are currently supported, ie. the possible values of the first argument of *registry.get()*

Values are the names of the autopilot classes that are searched for inheriting classes, eg. `HARDWARE == "autopilot.hardware.Hardware"` for `autopilot.Hardware`

Attributes:

HARDWARE
TASK
GRADUATION
TRANSFORM
CHILDREN
SOUND

`HARDWARE = 'autopilot.hardware.Hardware'`

`TASK = 'autopilot.tasks.Task'`

`GRADUATION = 'autopilot.tasks.graduation.Graduation'`

`TRANSFORM = 'autopilot.transform.transforms.Transform'`

`CHILDREN = 'autopilot.tasks.children.Child'`

`SOUND = 'autopilot.stim.sound.sounds.BASE_CLASS'`

get(*base_class: Union[*autopilot.utils.registry.REGISTRIES*, str, type], class_name: Optional[str] = None, plugins: bool = True, ast: bool = True, include_base: bool = False*) → Union[type, List[type]]

Get an autopilot object.

Parameters

base_class (*REGISTRIES*, str, type) – Class to search its subclasses for the indicated object. One of the values in th or else one of its keys (eg. `'HARDWARE'`). If given a full module.ClassName string (eg. `"autopilot.tasks.Task"`) attempt to get the indicated object. If given an object, use that.

class_name (str, None): Name of class that inherits from base_class that is to be returned. if `None` (default), return all found subclasses of `base_class`

plugins (bool): If `True` (default), ensure contents of PLUGINDIR are loaded (with *import_plugins()*) and are included in results. If `False`, plugins are not explicitly imported, but if any have been imported elsewhere, they will be included anyway because we can't control all the different ways to subclass in Python.

ast (bool): If `True` (default), if an imported object isn't found that matches `class_name`, parse the syntax trees of submodules of `base_class` with *utils.common.list_classes()* without importing to try and find it. If a match is found, it is imported and checked whether or not it is indeed a subclass of the `base_class`. if `False`, do not parse ast trees (will miss any modules that aren't already imported).

include_base (bool): If `False` (default), remove the `base_class` before returning

Returns Either the requested items, or a list of all the relevant items

`get_names`(*base_class: Union[*autopilot.utils.registry.REGISTRIES*, str, type], class_name: Optional[str] = None, plugins: bool = True, ast: bool = True, full_name: bool = False*) → List[str]

`get()` but return a list of object names instead of the objects themselves

See `get()` for documentation of base arguments.

Note: While technically you can call this function with a `class_name`, by default `[class_name] == get_names(base_class, class_name)`, but if `full_name == False` it could be used to get the fully qualified package.module name in a pretty roundabout way.

Parameters **full_name** (*bool*) – if `False` (default), return just the class name. if `True`, return the full `package.subpackage.module.Class_Name` name.

Returns a list of names

Return type List[str]

`get_hardware`(*class_name: Optional[str] = None, plugins: bool = True, ast: bool = True*) → Union[Type[Hardware], List[Type[Hardware]]]

Get a hardware class by name.

Alias for `registry.get()`

Parameters

- **class_name** (*str*) – Name of hardware class to get
- **plugins** (*bool*) – If `True` (default) ensure plugins are loaded and return from them. see `registry.get()` for more details about the behavior of this argument
- **ast** (*bool*) – If `True` (default) parse the syntax tree of all modules within `hardware`. see `registry.get()` for more details about the behavior of this argument

Returns `Hardware`

`get_task`(*class_name: Optional[str] = None, plugins: bool = True, ast: bool = True*) → Union[Type[Task], List[Type[Task]]]

Get a task class by name.

Alias for `registry.get()`

Parameters

- **class_name** (*str*) – Name of task class to get
- **plugins** (*bool*) – If `True` (default) ensure plugins are loaded and return from them. see `registry.get()` for more details about the behavior of this argument
- **ast** (*bool*) – If `True` (default) parse the syntax tree of all modules within `tasks`. see `registry.get()` for more details about the behavior of this argument

Returns `Task`

18.5 Wiki

Utility functions for dealing with the wiki (https://wiki.auto-pi-lot.com).

See the docstrings of the `ask()` function, as well as the guide_wiki_plugins section in the user guide for use.

Functions:

`ask`(filters[, properties])	Perform an API call to the wiki using the ask API and simplify to a list of dictionaries
`browse`(search[, browse_type, params])	Use the browse api of the wiki to search for specific pages, properties, and so on.
`make_ask_string`(filters[, properties, full_url])	Create a query string to request semantic information from the Autopilot wiki
`make_browse_string`(search[, browse_type, . . .])	

ask(*filters: Union[List[str], str], properties: Union[None, List[str], str] = None*) → List[dict]

Perform an API call to the wiki using the ask API and simplify to a list of dictionaries

Parameters

- **filters** (*list, str*) – A list of strings or a single string of semantic mediawiki formatted property filters. See `make_ask_string()` for more information
- **properties** (*None, list, str*) – Properties to return from filtered pages, See `make_ask_string()` for more information

Returns:

browse(*search: str, browse_type: str = 'page', params: Optional[dict] = None*)

Use the browse api of the wiki to search for specific pages, properties, and so on.

Parameters

- **search** (*str*) – the search string! * can be used as a wildcard.
- **browse_type** (*str*) – The kind of browsing we're doing, one of:
  - page
  - subject
  - property
  - pvalue
  - category
  - concept
- **params** (*dict*) – Additional params for the browse given as a dictionary, see the smw docs for usage.

Returns dict, list of dicts of results

make_ask_string(*filters: Union[List[str], str], properties: Union[None, List[str], str] = None, full_url: bool = True*) → str

Create a query string to request semantic information from the Autopilot wiki

Parameters

- **filters** (*list, str*) – A list of strings or a single string of semantic mediawiki formatted property filters, eg `"[[Category:Hardware]]"` or `"[[Has Contributor::sneakers-the-rat]]"`. Refer to the semantic mediawiki documentation for more information on syntax
- **properties** (*None, list, str*) – Properties to return from filtered pages, see the available properties on the wiki and the semantic mediawiki documentation for more information on syntax. If `None` (default), just return the names of the pages
- **full_url** (*bool*) – If `True` (default), prepend `f'{WIKI_URL}api.php?action=ask&query='` to the returned string to make it ready for an API call

Returns the formatted query string

Return type str

`make_browse_string`(*search*, *browse_type='page'*, *params=None*, *full_url: bool = True*)

CHAPTER NINETEEN

SETUP

After initial setup, configure autopilot: create an autopilot directory and a prefs.json file

Functions:

make_dir(adir[, permissions])	Make a directory if it doesn't exist and set its permissions to *0777*
make_alias(launch_script[, bash_profile])	Make an alias so that calling `autopilot` calls `autopilot_dir/launch_autopilot.sh`
parse_manual_prefs(manual_prefs)	
parse_args()	
locate_user_dir(args)	
run_form(prefs)	
make_launch_script(prefs[, prefs_fn, …])	
make_systemd(prefs, launch_file)	
results_string(env_results, config_msgs, …)	
make_ectopic_dirnames(basedir)	
main()	

make_dir(*adir: pathlib.Path*, *permissions: int = 511*)

Make a directory if it doesn't exist and set its permissions to *0777*

Parameters

- **adir** (*str*) – Path to the directory
- **permissions** (*int*) – an octal integer used to set directory permissions (default `0o777`)

make_alias(*launch_script: pathlib.Path*, *bash_profile: Optional[str] = None*) → Tuple[bool, str]

Make an alias so that calling `autopilot` calls `autopilot_dir/launch_autopilot.sh`

Parameters

- **launch_script** (*str*) – the path to the autopilot launch script to be aliased

- **bash_profile** (*str, None*) – Optional, location of shell profile to edit. if None, use `.bashrc` then `.bash_profile` if they exist

parse_manual_prefs(*manual_prefs: List[str]*) → dict

parse_args()

locate_user_dir(*args*) → pathlib.Path

run_form(*prefs: dict*) → Tuple[dict, List[str]]

make_launch_script(*prefs: dict, prefs_fn=None, launch_file=None, permissions: int = 509*) → pathlib.Path

make_systemd(*prefs: dict, launch_file: pathlib.Path*) → Tuple[bool, str]

results_string(*env_results: dict, config_msgs: List[str], error_msgs: List[str], prefs_fn: str, prefs*) → str

make_ectopic_dirnames(*basedir: pathlib.Path*) → dict

main()

19.1 scripts

Scripts used in *run_script* and *setup_autopilot* to install packages and configure the system environment

Scripts are contained in the *scripts.SCRIPTS* dictionary, and each script is of the form:

```
'script_name': {
    'type': 'bool', # always bool, signals that gui elements should present it as a␣
↪checkbox to run or not
    'text': 'human readable description of what the script does',
    'commands': [
        'list of shell commands'
    ]
}
```

The commands in each `commands` list are concatenated with `&&` and run sequentially (see *run_script.call_series()*). Certain commands that are expected to fail but don't impact the outcome of the rest of the script – eg. making a directory that already exists – can be made optional by using the syntax:

```
[
    'required command',
    {'command':'optional command', 'optional': True}
]
```

This concatenates the command with a ``; `` which doesn't raise an error if the command fails and allows the rest of the script to proceed.

Note: The above syntax will be used in the future for additional parameterizations that need to be made to scripts (though being optional is the only paramaterization avaialable now).

Note: An unadvertised feature of `raspi-config` is the ability to run commands frmo the cli – find the name of a command here: https://github.com/RPi-Distro/raspi-config/blob/master/raspi-config and then use it like this: `sudo raspi-config nonint function_name argument` , so for example to enable the camera one just calls sudo `raspi-config nonint do_camera 0` (where turning the camera on, perhaps counterintuitively, is `0` which is true for all commands)

Todo: Probably should have these use `prefs.get('S')copes` as well

Data:

SCRIPTS

```
SCRIPTS = OrderedDict([ ( 'env_pilot', { 'commands':  [ 'sudo apt-get update', 'sudo
apt-get install -y ' 'build-essential cmake git python3-dev ' 'libatlas-base-dev
libsamplerate0-dev ' 'libsndfile1-dev libreadline-dev ' 'libasound-dev i2c-tools '
'libportmidi-dev liblo-dev libhdf5-dev ' 'libzmq-dev libffi-dev'], 'text':  'install
system packages necessary for ' 'autopilot Pilots? (required if they arent ' 'already)',
'type':  'bool'}), ( 'env_terminal', { 'commands':  [ 'sudo apt-get update', 'sudo
apt-get install ' '-y ' 'libxcb-icccm4 ' 'libxcb-image0 ' 'libxcb-keysyms1 '
'libxcb-randr0 ' 'libxcb-render-util0 ' 'libxcb-xinerama0 ' 'libxcb-xfixes0'], 'text':
'install system packages necessary for ' 'autopilot Terminals? (required if they arent '
'already)', 'type':  'bool'}), ( 'performance', { 'commands':  [ 'sudo systemctl disable
raspi-config', "sudo sed -i '/^exit 0/i echo " '"performance" | sudo tee '
"/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor' " '/etc/rc.local', 'sudo sh -c
"echo @audio - memlock ' '256000 >> /etc/security/limits.conf"', 'sudo sh -c "echo @audio
- rtprio 75 ' '>> /etc/security/limits.conf"', 'sudo sh -c "echo vm.swappiness = 10 ' '>>
/etc/sysctl.conf"'], 'text':  'Do performance enhancements? (recommended, ' 'change cpu
governor and give more memory to ' 'audio)', 'type':  'bool'}), ( 'change_pw', {
'commands':  ['passwd'], 'text':  "If you haven't, you should change the default "
'raspberry pi password or you _will_ get your ' 'identity stolen. Change it now?',
'type':  'bool'}), ( 'set_locale', { 'commands':  [ 'sudo dpkg-reconfigure locales',
'sudo dpkg-reconfigure ' 'keyboard-configuration'], 'text':  'Would you like to set your
locale?', 'type':  'bool'}), ( 'hifiberry', { 'commands':  [ { 'command':  'sudo adduser
pi i2c', 'optional':  True}, 'sudo sed -i ' "'s/^dtparam=audio=on/#dtparam=audio=on/g' "
'/boot/config.txt', 'sudo sed -i ' "'$s/$/\\ndtoverlay=hifiberry-dacplus\\
ndtoverlay=i2s-mmap\\ndtoverlay=i2c-mmap\\ndtparam=i2c1=on\\ndtparam=i2c_arm=on/' "
'/boot/config.txt', "echo -e 'pcm.!default {\\n type hw " 'card 0\\n}\\nctl.!default {\\n
type ' "hw card 0\\n}' | sudo tee " '/etc/asound.conf'], 'text':  'Setup Hifiberry
DAC/AMP?', 'type':  'bool'}), ( 'bluetooth', { 'commands':  [ "sudo sed - i '$s/$/\n"
"dtoverlay=pi3-disable-bt/' " '/boot/config.txt', 'sudo systemctl disable '
'hciuart.service', 'sudo systemctl disable ' 'bluealsa.service', 'sudo systemctl disable
' 'bluetooth.service'], 'text':  "Disable Bluetooth? (recommended unless you're " 'using
it <3', 'type':  'bool'}), ( 'systemd', { 'text':  'Install Autopilot as a systemd
service?\n' 'If you are running this command in a virtual ' 'environment it will be used
to launch ' 'Autopilot', 'type':  'bool'}), ( 'alias', { 'text':  'Create an alias to
launch with "autopilot" ' '(must be run from setup_autopilot, calls ' 'make_alias)',
'type':  'bool'}), ( 'jackd_apt', { 'commands':  [ 'sudo apt update && sudo apt install '
'-y jackd2'], 'text':  'Install jack audio from apt repository ' '(required if
AUDIOSERVER == jack)', 'type':  'bool'}), ( 'jackd_source', { 'commands':  [ 'git clone '
'git://github.com/jackaudio/jack2 ' '--depth 1', 'cd jack2', './waf configure --alsa=yes
' '--libdir=/usr/lib/arm-linux-gnueabihf/', './waf build -j6', 'sudo ./waf install',
'sudo ldconfig', 'sudo sh -c "echo @audio - memlock ' '256000 >>
/etc/security/limits.conf"', 'sudo sh -c "echo @audio - rtprio 75 ' '>>
/etc/security/limits.conf"', 'cd ..', 'rm -rf ./jack2'], 'text':  'Install jack audio
from source, try this if ' 'youre having compatibility or runtime issues ' 'with jack
(required if AUDIOSERVER == jack)', 'type':  'bool'}), ( 'opencv', { 'commands':  [ 'sudo
apt-get install -y ' 'build-essential cmake ccache unzip ' 'pkg-config libjpeg-dev
libpng-dev ' 'libtiff-dev libavcodec-dev ' 'libavformat-dev libswscale-dev ' 'libv4l-dev
libxvidcore-dev ' 'libx264-dev ffmpeg libgtk-3-dev ' 'libcanberra-gtk* libatlas-base-dev
' 'gfortran python2-dev python-numpy', 'git clone '
'https://github.com/opencv/opencv.git', 'git clone '
'https://github.com/opencv/opencv_contrib', 'cd opencv', 'mkdir build', 'cd build',
'cmake -D ' 'CMAKE_BUILD_TYPE=RELEASE ' '-D ' 'CMAKE_INSTALL_PREFIX=/usr/local ' '-D '
'OPENCV_EXTRA_MODULES_PATH=/home/pi/git/opencv_contrib/modules ' '-D BUILD_TESTS=OFF -D '
'BUILD_PERF_TESTS=OFF ' '-D BUILD_DOCS=OFF -D ' 'WITH_TBB=ON -D '
'CMAKE_CXX_FLAGS="-DTBB_USE_GCC_BUILTINS=1 ' '-D__TBB_64BIT_ATOMICS=0" ' '-D
WITH_OPENMP=ON -D ' 'WITH_IPP=OFF -D ' 'WITH_OPENCL=ON -D ' 'WITH_V4L=ON -D '
'WITH_LIBV4L=ON -D ' 'ENABLE_NEON=ON -D ' 'ENABLE_VFPV3=ON -D '
'PYTHON3_EXECUTABLE=/usr/bin/python3 ' '-D ' 'PYTHON_INCLUDE_DIR=/usr/include/python3.7
'-D ' 'PYTHON_INCLUDE_DIR2=/usr/include/arm-linux-gnueabihf/python3.7 ' '-D '
'OPENCV_ENABLE_NONFREE=ON ' '-D ' 'INSTALL_PYTHON_EXAMPLES=OFF ' '-D WITH_CAROTENE=ON '
'-D ' "CMAKE_SHARED_LINKER_FLAGS='-latomic' " '-D BUILD_EXAMPLES=OFF ..', 'sudo sed -i '
```

19.2 run_script

Run scripts to setup system dependencies and autopilot plugins

```
> # to list scripts
> python3 -m autopilot.setup.run_script --list

> # to execute one script (setup hifiberry soundcard)
> python3 -m autopilot.setup.run_script hifiberry

> # to execute multiple scripts
> python3 -m autopilot.setup.run_script hifiberry jackd
```

Functions:

call_series(commands[, series_name])	Call a series of commands, giving a single return code on completion or failure
run_script(script_name)	
run_scripts(scripts[, return_all, print_status])	Run a series of scripts, printing results
list_scripts()	

call_series(*commands*, *series_name=None*)

Call a series of commands, giving a single return code on completion or failure

Parameters **commands** –

Returns

run_script(*script_name*)

run_scripts(*scripts*, *return_all=False*, *print_status=True*) → Union[bool, Dict[str, bool]]

Run a series of scripts, printing results

Parameters **scripts** (*list*) – list of script names

Returns success or failure of scripts - `True` if all were successful, `False` otherwise.

Return type bool

list_scripts()

CHAPTER

TWENTY

PREFS

Module to hold module-global variables as preferences.

Upon import, prefs attempts to import a `prefs.json` file from the default location (see *prefs.init()*).

Prefs are then accessed with *prefs.get()* and *prefs.set()* functions. After initialization, if a pref if `set`, it is stored in the `prefs.json` file – prefs are semi-durable and persist across sessions.

When attempting to get a pref that is not set, *prefs.get()* will first try to find a default value (set in `_PREFS` , and if none is found return `None` – accordingly no prefs should be intentionally set to None, as it signifies that the pref is not set.

Prefs are thread- and process-safe, as they are stored and served by a `multiprocessing.Manager` object.

`prefs.json` is typically generated by running *autopilot.setup.setup_autopilot* , though you can freestyle it if you are so daring.

The ``**HARDWARE**`` pref is a little special. It specifies how each of the *hardware* components connected to the system is configured. It is a dictionary with this general structure:

```
'HARDWARE': {
    'GROUP': {
        'ID': {
            'hardware_arg': 'val'
        }
    }
}
```

where there are user-named `'GROUPS'` of hardware objects, like `'LEDS'` , etc. Within a group, each object has its `'ID'` (passed as the `name` argument to the hardware initialization method) which allows it to be identified from the other components in the group. The intention of this structure is to allow multiple categories of hardware objects to be parameterized and used separately, even though they might be the same object type. Eg. we may have three LEDs in our nosepokes, but also have an LED that serves at the arena light. If we wanted to write a command that turns off all LEDs, we would have to explicitly specify their IDs, making it difficult to re-use very common hardware command patterns within tasks. There are obvious drawbacks to this scheme – clunky, ambiguous, etc. and will be deprecated as parameterization continues to congeal across the library.

The class that each element is used with is determined by the *Task.HARDWARE* dictionary. Specifically, the *Task.init_hardware()* method does something like:

```
self.hardware['GROUP']['ID'] = self.HARDWARE['GROUP']['ID'](**prefs.get('HARDWARE')[
↪'GROUP']['ID'])
```

> **Warning:** These are **not** hard coded prefs. *_DEFAULTS* populates the *default* values for prefs, but local prefs are always restored from and saved to `prefs.json` . If you're editing this file and things aren't changing, you're in the wrong place!

This iteration of prefs with respect to work done on the People's Ventilator Project

If a pref has a string for a `'deprecation'` field in *prefs._DEFAULTS* , a `FutureWarning` will be raised with the string given as the message

Classes:

Scopes(value)	Enum that lists available scopes and groups for prefs

Data:

_DEFAULTS	Ordered Dictionary containing default values for prefs.

Functions:

get([key])	Get a pref!
set(key, val)	Set a pref!
save_prefs([prefs_fn])	Dump prefs into the `prefs_fn` .json file
init([fn])	Initialize prefs on autopilot start.
add(param, value)	Add a pref after init
git_version(repo_dir)	Get the git hash of the current commit.
compute_calibration([path, calibration, . . .])	**Parameters** • **path**
clear()	Mostly for use in testing, clear loaded prefs (without deleting prefs.json)

class Scopes(*value*)

Bases: `enum.Enum`

Enum that lists available scopes and groups for prefs

Scope can be an agent type, common (for everyone), or specify some subgroup of prefs that should be presented together (like directories)

COMMON = All Agents DIRECTORY = Prefs group for specifying directory structure TERMINAL = prefs for Terminal Agents Pilot = Prefs for Pilot agents LINEAGE = prefs for networking lineage (until networking becomes more elegant ;) AUDIO = Prefs for configuring the Jackd audio server

Attributes:

COMMON	All agents
TERMINAL	Prefs specific to Terminal Agents
PILOT	Prefs specific to Pilot Agents
DIRECTORY	Directory structure
LINEAGE	Prefs for coordinating network between pilots and children

continues on next page

Table 4 – continued from previous page

AUDIO	Audio prefs. . .

COMMON = 1
All agents

TERMINAL = 2
Prefs specific to Terminal Agents

PILOT = 3
Prefs specific to Pilot Agents

DIRECTORY = 4
Directory structure

LINEAGE = 5
Prefs for coordinating network between pilots and children

AUDIO = 6
Audio prefs. . .

_PREF_MANAGER = <multiprocessing.managers.SyncManager object at 0x7fcecedc9e50>
The `multiprocessing.Manager` that stores prefs during system operation and makes them available and consistent across processes.

```
_DEFAULTS = OrderedDict([ ( 'NAME', { 'scope': <Scopes.COMMON: 1>, 'text': 'Agent
Name:', 'type': 'str'}), ( 'PUSHPORT', { 'default': '5560', 'scope': <Scopes.COMMON:
1>, 'text': 'Push Port - Router port used by the Terminal ' 'or upstream agent:',
'type': 'int'}), ( 'MSGPORT', { 'default': '5565', 'scope': <Scopes.COMMON: 1>,
'text': 'Message Port - Router port used by this agent ' 'to receive messages:',
'type': 'int'}), ( 'TERMINALIP', { 'default': '192.168.0.100', 'scope':
<Scopes.COMMON: 1>, 'text': 'Terminal IP:', 'type': 'str'}), ( 'LOGLEVEL', { 'choices':
('DEBUG', 'INFO', 'WARNING', 'ERROR'), 'default': 'WARNING', 'scope': <Scopes.COMMON:
1>, 'text': 'Log Level:', 'type': 'choice'}), ( 'LOGSIZE', { 'default': 5242880,
'scope': <Scopes.COMMON: 1>, 'text': 'Size of individual log file (in bytes)', 'type':
'int'}), ( 'LOGNUM', { 'default': 4, 'scope': <Scopes.COMMON: 1>, 'text': 'Number of
logging backups to keep of LOGSIZE', 'type': 'int'}), ( 'CONFIG', { 'hidden': True,
'scope': <Scopes.COMMON: 1>, 'text': 'System Configuration', 'type': 'list'}), (
'VENV', { 'default':
'/home/docs/checkouts/readthedocs.org/user_builds/auto-pi-lot/envs/parallax', 'scope':
<Scopes.COMMON: 1>, 'text': 'Location of virtual environment, if used.', 'type':
'str'}), ( 'AUTOPLUGIN', { 'default': True, 'scope': <Scopes.COMMON: 1>, 'text':
'Attempt to import the contents of the plugin ' 'directory', 'type': 'bool'}), (
'PLUGIN_DB', { 'default': '/home/docs/autopilot/plugin_db.json', 'scope':
<Scopes.COMMON: 1>, 'text': 'filename to use for the .json plugin_db that ' 'keeps track
of installed plugins', 'type': 'str'}), ( 'BASEDIR', { 'default':
'/home/docs/autopilot', 'scope': <Scopes.DIRECTORY: 4>, 'text': 'Base Directory',
'type': 'str'}), ( 'DATADIR', { 'default': '/home/docs/autopilot/data', 'scope':
<Scopes.DIRECTORY: 4>, 'text': 'Data Directory', 'type': 'str'}), ( 'SOUNDDIR', {
'default': '/home/docs/autopilot/sounds', 'scope': <Scopes.DIRECTORY: 4>, 'text':
'Sound file directory', 'type': 'str'}), ( 'LOGDIR', { 'default':
'/home/docs/autopilot/logs', 'scope': <Scopes.DIRECTORY: 4>, 'text': 'Log Directory',
'type': 'str'}), ( 'VIZDIR', { 'default': '/home/docs/autopilot/viz', 'scope':
<Scopes.DIRECTORY: 4>, 'text': 'Directory to store Visualization results', 'type':
'str'}), ( 'PROTOCOLDIR', { 'default': '/home/docs/autopilot/protocols', 'scope':
<Scopes.DIRECTORY: 4>, 'text': 'Protocol Directory', 'type': 'str'}), ( 'PLUGINDIR', {
'default': '/home/docs/autopilot/plugins', 'scope': <Scopes.DIRECTORY: 4>, 'text':
'Directory to import ', 'type': 'str'}), ( 'REPODIR', { 'default': PosixPath('/home/
docs/checkouts/readthedocs.org/user_builds/auto-pi-lot/checkouts/parallax'), 'scope':
<Scopes.DIRECTORY: 4>, 'text': 'Location of Autopilot repo/library', 'type': 'str'}), (
'CALIBRATIONDIR', { 'default': '/home/docs/autopilot/calibration', 'scope':
<Scopes.DIRECTORY: 4>, 'text': 'Location of calibration files for solenoids, ' 'etc.',
'type': 'str'}), ( 'PIGPIOMASK', { 'default': '1111110000111111111111110000', 'scope':
<Scopes.PILOT: 3>, 'text': 'Binary mask controlling which pins pigpio ' 'controls
according to their BCM numbering, ' 'see the -x parameter of pigpiod', 'type': 'str'}),
( 'PIGPIOARGS', { 'default': '-t 0 -l', 'scope': <Scopes.PILOT: 3>, 'text': 'Arguments
to pass to pigpiod on startup', 'type': 'str'}), ( 'PULLUPS', { 'scope': <Scopes.PILOT:
3>, 'text': 'Pins to pull up on system startup? (list of ' 'form [1, 2])', 'type':
'list'}), ( 'PULLDOWNS', { 'scope': <Scopes.PILOT: 3>, 'text': 'Pins to pull down on
system startup? (list of ' 'form [1, 2])', 'type': 'list'}), ( 'PING_INTERVAL', {
'default': 5, 'scope': <Scopes.PILOT: 3>, 'text': 'How many seconds should pilots wait
in ' 'between pinging the Terminal?', 'type': 'float'}), ( 'DRAWFPS', { 'default':
'20', 'scope': <Scopes.TERMINAL: 2>, 'text': 'FPS to draw videos displayed during '
'acquisition', 'type': 'int'}), ( 'PILOT_DB', { 'default':
'/home/docs/autopilot/pilot_db.json', 'scope': <Scopes.TERMINAL: 2>, 'text': 'filename
to use for the .json pilot_db that ' 'maps pilots to subjects (relative to BASEDIR)',
'type': 'str'}), ( 'TERMINAL_SETTINGS_FN', { 'default':
'/home/docs/autopilot/terminal.conf', 'scope': <Scopes.TERMINAL: 2>, 'text': 'filename
to store QSettings file for Terminal', 'type': 'str'}), ( 'TERMINAL_WINSIZE_BEHAVIOR', {
'choices': ( 'remember', 'moderate', 'maximum', 'custom'), 'default': 'remember',
'scope': <Scopes.TERMINAL: 2>, 'text': 'Strategy for resizing terminal window on '
'opening', 'type': 'choice'}), ( 'TERMINAL_CUSTOM_SIZE', { 'default': [0, 0, 1000,
400], 'depends': ('TERMINAL_WINSIZE_BEHAVIOR', 'custom'), 'scope': <Scopes.TERMINAL:
2>, 'text': 'Custom size for window, specified as [px from ' 'left, px from top, width,
height]', 'type': 'list'}), ( 'LINEAGE', { 'choices': ('NONE', 'PARENT', 'CHILD'),
```

Ordered Dictionary containing default values for prefs.

An Ordered Dictionary lets the prefs be displayed in gui elements in a predictable order, but prefs are stored in `prefs.json` in alphabetical order and the 'live' prefs used during runtime are stored in _PREFS

Each entry should be a dict with the following structure:

```
"PREF_NAME": {
    "type": (str, int, bool, choice, list) # specify the appropriate GUI input, str␣
↪or int are validators,
    choices are a
        # dropdown box, and lists allow users to specify lists of values like "[0,␣
↪1]"
    "default": If possible, assign default value, otherwise None
    "text": human-readable text that described the pref
    "scope": to whom does this pref apply? see :class:`.Scopes`
    "depends": name of another pref that needs to be supplied/enabled for this one␣
↪to be enabled (eg. don't set sampling rate of audio server if audio server␣
↪disabled)
        can also be specified as a tuple like ("LINEAGE", "CHILD") that enables the␣
↪option when prefs[depends[0]] == depends[1]
    "choices": If type=="choice", a tuple of available choices.
}
```

get(*key: Optional[str] = None*)

Get a pref!

If a value for the given `key` can't be found, prefs will attempt to

Parameters **key** (*str, None*) – get pref of specific `key`, if `None`, return all prefs

Returns value of pref (type variable!), or `None` if no pref of passed `key`

set(*key: str*, *val*)

Set a pref!

Note: Whenever a pref is set, the prefs file is automatically updated – prefs are system-durable!!

(specifically, whenever the module-level `_INITIALIZED` value is set to True, prefs are saved to file to avoid overwriting before loading)

Parameters

- **key** (*str*) – Name of pref to set
- **val** – Value of pref to set (prefs are not type validated against default types)

save_prefs(*prefs_fn: Optional[str] = None*)

Dump prefs into the `prefs_fn` .json file

Parameters

- **prefs_fn** (*str, None*) – if provided, pathname to `prefs.json` otherwise resolve `prefs.json` according the
- **to the normal methods. . . .**

init(*fn=None*)

Initialize prefs on autopilot start.

If passed dict of prefs or location of prefs.json, load and use that

Otherwise

- Look for the autopilot wayfinder `~/.autopilot` file that tells us where the user directory is
- look in default location `~/autopilot/prefs.json`

Todo: This function may be deprecated in the future – in its current form it serves to allow the sorta janky launch methods in the headers/footers of autopilot/core/pilot.py and autopilot/core/terminal.py that will eventually be transformed into a unified agent framework to make launching easier. Ideally one would be able to just import prefs without having to explicitly initialize it, but we need to formalize the full launch process before we make the full lurch to that model.

Parameters **fn** (*str, dict*) – a path to *prefs.json* or a dictionary of preferences

`add`(*param*, *value*)
Add a pref after init

Parameters

- **param** (*str*) – Allcaps parameter name
- **value** – Value of the pref

`git_version`(*repo_dir*)
Get the git hash of the current commit.

Stolen from numpy's setup

and linked by ryanjdillon on SO

Parameters **repo_dir** (*str*) – directory of the git repository.

Returns git commit hash.

Return type unicode

`compute_calibration`(*path=None*, *calibration=None*, *do_return=False*)

Parameters

- **path**
- **calibration**
- **do_return**

Returns:

`clear`()
Mostly for use in testing, clear loaded prefs (without deleting prefs.json)

(though you will probably overwrite prefs.json if you clear and then set another pref so don't use this except in testing probably)

CHAPTER

TWENTYONE

EXTERNAL

Autopilot uses two lightly modified versions of existing libraries that are included in the repository as submodules.

- mlx90640-library - driver for the `hardware.i2c.MLX90640` that correctly sets the baudrate for 64fps capture
- pigpio - pigpio that is capable of returning full timestamps rather than system ticks in gpio callbacks.

CHAPTER

TWENTYTWO

CHANGELOG

For full details, see commit logs and issues at http://github.com/wehr-lab/autpilot

22.1 Version 0.4

22.1.1 v0.4.1 (August 17th)

Bugfixes

- The `autopilot.setup.forms.HARDWARE_FORM` would incorrectly use the class object itself rather than the class name in a few places which caused hardware names to incorrectly display and be impossible to add!

22.1.2 v0.4.0 - Become Multifarious (August 3rd, 2021)

This release is primarily to introduce the new plugin system, the autopilot wiki, and their integration as a way of starting the transformation of Autopilot into a tool with decentralized development and governance (as well as make using the tool a whole lot easier and more powerful).

With humble thanks to Lucas Ott, Tillie Morris, Chris Rodgers, Arne Meyer , Mikkel Roald-Arbøl , David Robbe , *and an anonymous discussion board poster for being part of this release.*

New Features

- Registries & Plugins - Autopilot now supports users writing their code outside of the library as plugins! To support this, a registry system was implemented throughout the program. Plugin objects can be developed as objects that inherit from the Autopilot object tree – eg. implementing a GPIO object by subclassing `hardware.gpio.GPIO` , or a new task by subclassing `Task` . This system is flexible enough to allow any lineage of objects to be included as a plugin – stimuli, tasks, and so on – and we will be working to expand registries to every object in Autopilot, including the ability for plugins to replace core modules to make Autopilot's flexibility verge on ludicrous. The basic syntax of the registry system is simple and doesn't require any additional logic beyond inheritance to be implemented on plugin objects – `autopilot.get('object_type', 'object_name')` is the basic method, with a few aliases for specific object types like `autopilot.get_hardware()`. Also thanks to Arne Meyer for submitting an early draft of the registry system and Mikkel Roald-Arbøl for raising the issue.
- At long last, the Autopilot Wiki is alive!!!! - https://wiki.auto-pi-lot.com/ - The wiki is the place for communal preservation of technical knowledge about using Autopilot, like hardware designs, build guides, parameter sets, and beyond! This isn't any ordinary wiki, though, we got ourselves a *semantic wiki* which augments traditional wikis with a rich system of human and computer-readable linked attributes: a particular type of page will have some set of attributes, like a page about a 3D printed part will have an associated .stl file, but rather than having

these be in plaintext they are specified in a format that is queryable, extensible, and infinitely mutable. The vision for the wiki is much grander (but not speculative! very concrete!) than just a place to take notes, but is intended to blend the use of Autopilot as an experimental tool with body of knowledge that supports it. Autopilot can query the wiki with the `wiki` module like `wiki.ask('[[Category:3D_CAD]]', 'Has STL')` to get links to all .stl files for all 3D parts on the wiki. The integration between the two makes using and submitting information trivial, but *also* makes *designing whole new types of community interfaces* completely trivial. As a first pass, the Wiki will be the place to index plugins, the system for submitting them, querying them, and downloading them only took a few hours and few dozen lines of code to implement. The wiki is infinitely malleable – that's the point – and I am very excited to see how people use it.

- Tests & Continuous Integration with Travis! We are on the board with having nonzero tests! The travis page is here: https://travis-ci.com/github/wehr-lab/autopilot and the coveralls page is here: https://coveralls.io/github/wehr-lab/autopilot . At the moment we have a whopping 27% coverage, but as we build out our testing suite we hope that it will become much easier for people to contribute to Autopilot and be confident that it works!
- **New Hardware Objects**
  - `cameras.PiCamera` - A fast interface to the PiCamera, wrapping the picamera library, and using tips from its developer to juice every bit of speed i could!
  - The I2C_9DOF object was massively improved to take better advantage of its onboard DSP and expose more of its i2c commands.
- **New Transforms**
  - `timeseries.Kalman` - adapted a Kalman filter from the wonderful filterpy package! it's in the new timeseries transform module
  - `geometry.IMU_Orientation` - IMU_Orientation performs a sensor fusion algorithm with the Kalman Filter class to combine gyroscope and accelerometer measurements into a better estimate of earth-centric roll and pitch. This is used by the IMU class, but is made independent so it can be used without an Autopilot hardware object/post-facto/etc.
  - `timeseries.Filter_IIR` - Filter_IIR implements scipy's IIR filter as a transform object.
  - `timeseries.Integrate` - Integrate adds successive numbers together (scaled by dt if requested). not much by itself, but when used with a kalman filter very useful :)
  - `geometry.Rotate` - use scipy to rotate a vector by some angle in x, y, and/or z
  - `geometry.Spheroid` - fit and transform 3d coordinates according to some spheroid - used in the IMU's accelerometer calibration method: given some target spheroid, and some deformed spheroid (eg. a miscalibrated accelerometer might have the x, y, or z axis scaled or offset) either explicitly set or estimated from a series of point measurements, transform future input given that transformation to correct for the deformed source spheroid.
- **New Prefs**
  - `'AUTOPLUGIN'` - Attempt to import the contents of the plugin directory,
  - `'PLUGIN_DB'` - filename to use for the .json plugin_db that keeps track of installed plugins',
  - `'PING_INTERVAL'` - How many seconds should pilots wait in between pinging the Terminal?',
  - `'TERMINAL_SETTINGS_FN'` - filename to store QSettings file for Terminal',
  - `'TERMINAL_WINSIZE_BEHAVIOR'` - Strategy for resizing terminal window on opening',
  - `'TERMINAL_CUSTOM_SIZE'` - Custom size for window, specified as [px from left, px from top, width, height]',

Major Improvements

- Stereo Sound (Thank you Chris Rodgers!) - https://github.com/wehr-lab/autopilot/pull/102
- Multihop messages & direct messaging - https://github.com/wehr-lab/autopilot/pull/99 - it is now possible to send multihop messages through multiple Station objects, as well as easier to send messages directly between net nodes. See the examples in the network tests section of the docs.
- Multiple Children (Thank you Chris Rodgers!) - https://github.com/wehr-lab/autopilot/pull/103 - the `CHILDID` field now accepts a list, allowing a Pilot to initialize child tasks on multiple children. (this syntax and the hierarchical nature of pilots and children will be deprecated as we refactor the networking modules into a general mesh system, but this is lovely to have for now :)
- Programmatic Setup - https://github.com/wehr-lab/autopilot/issues/33 - noninteractive setup of prefs and scripts by using `autopilot.setup -f prefs.json -p PREFNAME=VALUE -s scriptname1 -s scriptname2`
- Widget to stream video, en route to more widgets for direct GUI control of hardware objects connected to pilots
- Support python 3.8 and 3.9 essentially by not insisting that the spinnaker SDK be installable by all users (which at the time was only available for 3.7)

Minor Improvements

- Terminal can be opened maximized, or have its size and position set explicitly, preserve between launches (Thank you Chris Rodgers!) - https://github.com/wehr-lab/autopilot/pull/70
- Pilots will periodically ping the Terminal again, Terminal can manually ping Pilots that may have gone silent - https://github.com/wehr-lab/autopilot/pull/91
- Pilots share their prefs with the Terminal in their initial handshake - https://github.com/wehr-lab/autopilot/pull/91
- Reintroduce router ports for net-nodes to allow them to bind a port to receive messages - https://github.com/wehr-lab/autopilot/pull/115/commits/35be5d634d98a7983ec3d3d6c5b94da6965a2579
- Listen methods are now optional for net_nodes
- Allowed the creation of dataless tasks - https://github.com/wehr-lab/autopilot/pull/115/commits/628e1fb9c8fcd15399b19b351fed87e4826bc9ab
- Allowed the creation of plotless tasks - https://github.com/wehr-lab/autopilot/pull/115/commits/08d99d55a32b45f54e3853813c7c71ea230b25dc
- The `I2C_9DOF` clas uses memoryviews rather than buffers for a small performance boost - https://github.com/wehr-lab/autopilot/pull/115/commits/890f2c500df8010b50d61f64e2755cd2c7a8aeed
- Phasing out using `Queue` s in favor of `collections.deque` for applications that only need thread and not process safety because they are way faster and what we wanted in the first place anyway.
- New Scripts - `i2c`, `picamera`, `env_terminal`
- utils.NumpyEncoder and decoder to allow numpy arrays to be json serialized
- calibrations are now loaded by hardware objects themselves instead of the extraordinarily convoluted system in `prefs` – though some zombie code still remains there.
- Net nodes know their ip now, but this is a lateral improvement pending a reworking of the networking modules.
- `performance` script now sets `swappiness = 10` to discourage the use of swapfiles - see https://www.raspberrypi.org/forums/viewtopic.php?t=198765
- Setting a string in the `deprecation` field of a pref in `_DEFAULTS` prints it as a warning to start actually deprecating responsibly.

- Logging in more places like Subject creation, manipulation, protocol assignation.

Bugfixes

- Loggers would only work for the last object that was instantiated, which was really embarassing. fixed - https://github.com/wehr-lab/autopilot/pull/91
- Graduation criteria were calculated incorrectly when subjects were demoted in stages of a protocol - https://github.com/wehr-lab/autopilot/pull/91
- fix durations in solenoid class (Thank you Chris Rodgers!) - https://github.com/wehr-lab/autopilot/pull/63
- LED_RGB ignores zero - https://github.com/wehr-lab/autopilot/pull/98
- Fix batch assignment window crashing when there are subjects that are unassigned to a task - https://github.com/wehr-lab/autopilot/pull/115/commits/e42fc5802792822ff5a53a2379041a4a8b301e9e
- Catch malformed protocols in batch assignment widget - https://github.com/wehr-lab/autopilot/pull/115/commits/2cc8508a4bf3a6d49512197dc72433c60d0c656e
- Remove broken `Terminal.reset_ui` method and made control panel better at adding/removing pilots - https://github.com/wehr-lab/autopilot/pull/91
- Subject class handles unexpected state a lot better (eg. no task assigned, no step assigned, tasks with no data.) but is still an absolute travesty that needs to be refactored badly.
- The jackclient would crash with long-running continuous sounds as the thread feeding it samples eventually hiccuped. Made more robust by having jackclient store samples locally int he sound server rather than being continuously streamed from the queue.
- PySide2 references still incorrectly used `QtGui` rather than `QtWidgets`
- pigpio scripts would not be stopped and removed when a task was stopped, the *`gpio.clear_scripts()`* function now handles that.
- `xcb` was removed from `PySide2` distributions, so it's now listed in the requirements for the Terminal and made available in the `env_terminal` script.
- `LED_RGB` didn't appropriately raise a `ValueError` when called with a single `pin` - https://github.com/wehr-lab/autopilot/issues/117
- A fistful of lingering Python 2 artifacts

Code Structure

- continuing to split out modules in *`autopilot.core`* - networking this time
- utils is now a separate module instead of being in multiple places
- the npyscreen forms in `setup_autopilot` were moved to a separate module
- `setup_autopilot` was broken into functions instead of a very long and impenetrable script. still a bit of cleaning to do there.
- `autopilot.setup.setup_autopilot` was always extremely awkward, so it's now been aliased as `autopilot.setup`
- the docs have now been split into subfolders rather than period separated names to make urls nicer – eg /dev/hardware/cameras.htm rather than /dev/hardware.cameras.html . this should break some links when switching between versions on readthedocs but other than that be nondestructive.

Docs

- new *Quickstart* documentation with lots of quick examples!

Regressions

- Removed the `check_compatible` method in the Transforms class. We will want to make a call at some point if we want to implement a full realtime pipelining framework or if we want to use something like luigi or joblib or etc. for now this is an admission that type and shape checking was never really implemented but it does raise some exceptions sometimes.

22.2 Version 0.3

22.2.1 v0.3.5 (February 22, 2021)

Bugfixes

- Very minor one, fixes to the way `Terminal` accesses the `pilot_db.json` file to use `Terminal.pilots` property that makes a new pilot_db.json file if one doesn't exist, but otherwise loads the one that is found in `prefs.get('PILOT_DB')`
- Reorganized `Terminal` source to group properties together & minor additions of type hinting
- Fixed some bad fallback behavior looking for files in old hardcoded default directories, eg. in the ye olde `utils.get_pilotdb()`

22.2.2 v0.3.4 (December 13, 2020)

Improvements

- Unify the creation of loggers!!!! See the docs ;) `autopilot.core.loggers` : https://github.com/wehr-lab/autopilot/pull/52/commits/d55638f985ab38044fc95ffeff5945021c2e198e https://github.com/wehr-lab/autopilot/issues/38
- Unify prefs, including sensible defaults, refactoring of scripts into a reasonable format, multiprocess-safety, and just generally a big weight off my mind. Note that this is a **breaking change** to the way prefs are accessed. Previously one would do *prefs.PREF_NAME*, but that made it very difficult to provide default values or handle missing prefs. the new syntax is *prefs.get('PREF_NAME')* which returns defaults with a warning and *None* if the pref is not set: https://github.com/wehr-lab/autopilot/pull/52/commits/c40a212bcaf5f184f2a6a606027fe15b1b4df59c https://github.com/wehr-lab/autopilot/issues/38
- completely clean up scripts, and together that opened the path to clean up setup as well. so all things configuration got a major promotion
- We're on the board with CI and automated testing with a positively massive 3% code coverage!!! https://github.com/wehr-lab/autopilot/pull/52/commits/743bb8fe67a69fcc556fa76e81f72f97f510dff7
- new scripts to eg. create autopilot alias: https://github.com/wehr-lab/autopilot/pull/52/commits/211919b05922e18a85d8ef6216973f4000fd32c5

Bugfixes

- cleanup scripts on object deletion: https://github.com/wehr-lab/autopilot/pull/52/commits/e8218304bd7ef2e13d2adfc236f3e781abea5f78 https://github.com/wehr-lab/autopilot/issues/41
- don't drop 'floats' from gui when we say we can use them…: https://github.com/wehr-lab/autopilot/pull/52/commits/743bb8fe67a69fcc556fa76e81f72f97f510dff7
- pigpio scripts dont like floats: https://github.com/wehr-lab/autopilot/pull/52/commits/9f939cd78a5296db3bf318115bee0213bcd1afc0

Docs

- Clarification of supported systems: https://github.com/wehr-lab/autopilot/pull/52/commits/ce0ddf78b7f59f5487fec2ca7e8fb3c0ad162051
- Solved an ancient sphinx riddle of how to get data objects/constants to pretty-print: https://github.com/wehr-lab/autopilot/pull/52/commits/ec6d5a75dada05688b6bd3c1a53b3d9e5923870f
- Clarify hardware prefs https://github.com/wehr-lab/autopilot/pull/52/commits/f3a7609995c84848004891a0f41c7847cb754aae
- what numbering system do we use: https://github.com/wehr-lab/autopilot/pull/52/commits/64267249d7b1ec1040b522308cd60f928f2b2ee6

Logging

- catch pigpio script init exception: https://github.com/wehr-lab/autopilot/pull/52/commits/3743f8abde7bbd3ed7766bdd75aee52afedf47e2
- more of it idk https://github.com/wehr-lab/autopilot/pull/52/commits/b682d088dbad0f206c3630543e96a5a00ceabe25

22.2.3 v0.3.3 (October 25, 2020)

Bugfixes

- Fix layout in batch reassign gui widget from python 3 float division
- Cleaner close by catching KeyboardInterrupt in networking modules
- Fixing audioserver boot options – if 'AUDIOSERVER' is set even if 'AUDIO' isn't set in prefs, should still start server. Not full fixed, need to make single plugin handler, single point of enabling/disabling optional services like audio server
- Fix conflict between polarity and pull in initializing *pulls* in pilot
- Catch `tables.HDF5ExtError` if local .h5 file corrupt in pilot
- For some reason 'fs' wasn't being replaced in the jackd string, reinstated.
- Fix comparison in LED_RGB that caused '0' to turn on full becuse 'value' was being checked for its truth value (0 is false) rather than checking if value is None.
- `obj.next()` to `next(obj)`` in jackdserver

Improvements

- Better internal handling of pigpiod – you're now able to import and use hardware modules without needing to explicitly start pigpiod!!
- Hopefully better killing of processes on exit, though still should work into unified process manager so don't need to reimplement everything (eg. as is done with launching pigpiod and jackd)
- Environment scripts have been split out into `setup/scripts.py` and you can now run them with `python -m autopilot.setup.run_script` (use `--help` to see how!)
- Informative error when setup is run with too narrow terminal: https://github.com/wehr-lab/autopilot/issues/23
- More loggers, but increased need to unify logger creation!!!

Cleanup

- remove unused imports in main `__init__.py` that made cyclical imports happen more frequently than necessary
- single-sourcing version number from `__init__.py`
- more cleanup of unnecessary meta and header stuff left from early days
- more debugging flags
- filter `NaturalNameWarning` from pytables
- quieter cleanups for hardware objects

22.2.4 v0.3.2 (September 28, 2020)

Bugfixes

- https://github.com/wehr-lab/autopilot/issues/19 - previously, I attempted to package binaries for the lightly modified pigpio and for jackd (the apt binary used to not work), but after realizing that was the worst possible way of going about it I changed install strategies, but didn't entirely remove the vestiges of the prior attempt. The installation expected certain directories to exist (in autopilot/external) that didn't, which crashed and choked install. Still need to formalize a configuration and plugin system, but getting there.
- https://github.com/wehr-lab/autopilot/issues/20 - the jackd binary in the apt repos for the raspi used to not work, so i was in the habit of compiling jackd audio from source. I had build that into the install routine, but something about that now causes the JACK-Client python interface to throw segfaults. Somewhere along the line someone fixed the apt repo version of jackd so we use that now.
- previously I had only tested in a virtual environment, but now the installation routine properly handles not being in a venv.

Cleanup

- remove bulky static files like fonts and css from /docs/ where they were never needed and god knows how they got there
- use a forked sphinx-sass when building docs that doesn't specify a required sphinx version (which breaks sphinx)
- removed skbuild requirements from install
- fixed pigpio install requirement in requirements_pilot.txt
- included various previously missed files in MANIFEST.in

- added installation of system libraries to the pilot configuration menu

22.2.5 v0.3.1 (August 4, 2020)

Practice version!!! still figuring out pypi

22.2.6 v0.3.0 (August 4, 2020)

Major Updates

- **Python 3** - We've finally made it to Python 3! Specifically we have brought Autopilot up to compatibility with Python 3.8 – though the Spinnaker SDK is currently only available through Python 3.7, so we have formally required 3.7 for now while we work on moving acquisition to Aravis. I will *not attempt to keep Autopilot compatible with Python 2*, but no decision has been made about compatibility with other versions of Python 3. Until then, expect that Autopilot will attempt to keep up with major version changes. The switch also let up update PySide (Qt library used for the GUI) to PySide2, which uses Qt5 and has a whole raft of other improvements.
- **Continuous Data Handling** - The `Subject` class and `networking` modules have been improved to handle continuous data (eg. streaming data, generally non-trialwise or non-event-sampled data). Continuous data can be set in a Task description either with a `tables` column descriptor as trial data is, but also can be set as `'infer'`, for which the `Subject` class will wait until it receives the first data and automatically create a `tables` column depending on its type and shape. While previously we intended to nudge users to be explicit about declaring their data, this was necessary to allow for data that might be variable in type and shape to be included in a Task – eg. it should be possible to record video data without needing to specify the resolution or bit depth as a hardcoded parameter in a task class. I have come to like type inference, and may make it a general practice for all types of data. That would potentially allow tasks to be written without explicitly declaring the data that they produce at all, but I haven't decided if that's a good thing or not yet.
- The **GPIO engine** has been rebuilt, relying more on `pigpio`'s function interface. This means that GPIO timing is now ~microsecond precise, important for reward delivery, LED flashing, and a number of other basic infrastructural needs. The reorganization of hardware modules resulted in general `GPIO`, `Digital_In` and `Digital_Out` metaclasses, making common operations like setting polarity, triggers, and pullup/down resistors much easier.
- Setup has been *greatly improved.* This includes proper packaging and installation with setuptools & sk-build, allowing us to finally join PyPI :) https://pypi.org/project/auto-pi-lot/ . Setup has been unified into a single npyscreen-based set of prompts that allow the user to run scripts to install libraries or configure their environment (also see `run_script()` and `list_scripts()`), set `prefs`, configure hardware objects (based on some very fun signature introspection), setup autopilot as a systemd service, etc. Getting started with Autopilot is now three commands!:

```
pip install auto-pi-lot
autopilot.setup.setup_autopilot
~/autopilot/launch_autopilot.sh
```

Minor Updates

- **Logging** level is now set from `prefs`, so where before, eg. every message through the networking modules would be logged to stdout, now only warnings and exceptions are. This gives a surprisingly large performance boost.
  - Logging has also been much improved in `networking` modules, where rather than an awkward `do_logging` flag that was used to avoid logging performance-critical events like streaming data, logging is controlled by log level throughout the system. By default, logging of most messages is set at `debug` level so they don't drown out important messages in the logs as they used to.
- **Networking** modules now only deserialize messages if they are the final recipient, saving lots of processing time – particularly with streamed arrays. *`Message`* objects also only re-serialize messages if they have been changed. Message structure has been changed such that serialized messages are now of the general format:

```
[sender,
 (optional) intermediate_node_1, intermediate_node_2, ...
 final_recipient,
 message_contents]
```

- Configuration will continue to be a point of improvement, but a few minor updates were made:
  - `prefs.CONFIG` will be used to signal multiple, potentially overlapping agent configurations, each of which may have their own system dependencies, external daemons, etc. Eg. a Pilot could be configured to play audio (which requires a jackd daemon to be started before Autopilot) and video (which requires Autopilot to be started in a X session). Checks of `prefs.CONFIG` are now `in` rather than == to reflect that.
  - `prefs.PINS` was renamed `prefs.HARDWARE`, and now allows hardware to be configured with dictionaries rather than integers only. Initially `PINS` was meant to just contain pin numbering for GPIO objects, but having a single point of hardware configuration is preferable. *`Task.init_hardware()`* now respects all parameters set in `prefs`.
- Throughout the code, minimal `get_this` type methods have begun to be replaced with `@property` attributes. This is because a) I love them and think they are magical, but b) will also be building Autopilot's closed-loop infrastructure around a Qt-style signal/slot architecture that wraps `@property` attributes so they can be `.connected` to one another easily.
- Previously it was possible to control presentation by *groups* of stimuli, but now it is possible to control the presentation frequency of individual stimuli.
- `PySide2` has proper support for CSS Stylesheets, so the design of Autopilot's GUI has been marginally improved, a process that will continue in the ceaseless quest for aesthetic perfection.
- Several setup routines have been added to make installation of opencv, pyspin, etc. easier. I also wrote a routine to `download_box()` files from a URL, which is mysteriously hard to do.
- The *To-Do* page now reflects the full ambition of Autopilot, where before this vision was contained only in the whitepaper and a disorganized plaintext file in the repo.
- The *`Subject`* class can now export trial data *`to_csv()`*. A very minor update, but one that is the first in a number of planned improvements to data export.
- I have also opened up a message board in google groups to make feature requests and discuss use and development, hope to see you there :)

  https://groups.google.com/forum/#!forum/autopilot-users

New Features

- **TRANSFORMS** have been introduced!!! *Transform* objects have a *process()* method that, well, transforms data in some way. Multiple transforms can be added together to make a transformation chain. This module is still very young and doesn't have a developed API, but will be built to to automatic type compatibility checking, coercion, parallelization, and rhythm (FIFO/FILO) control. Transforms are implemented with different modalities (image, selection, logical) that imply different types of input and output data structures, but the hierarchical structure of the modules is still quite flat.
  - Autopilot is now integrated with DeepLabCut-live!!!! You can now use realtime pose tracking in your experiments. See the dlclive_example
- **HARDWARE** has been substantially refactored to give objects an appropriate inheritance structure. This substantially reduces effort duplication across hardware objects and makes a bunch of obvious capabilities available to all of them, for example all hardware objects are now network (*init_networking()*) and logging (init_logging()) capable.
  - **Cameras**: The *cameras.Camera_CV* class allows webcams/other simple cameras to be accessed through OpenCV, and the *cameras.Camera_Spinnaker* class allows FLIR and other cameras to be accessed through the Spinnaker SDK. Cameras are capable of encoding videos locally (with x264), streaming frames over the network, and making acquired frames available to other objects on the same computer. The *Camera_Spinnaker* class provides simple @property setter/getter methods for common parameters, but also makes all PySpin attributes available to the user with its *get()* and *set()* methods. The *cameras.Camera* metaclass is written so that new camera types can be added by overriding a few methods. A new *Video_Child* can be used to run a camera on a Child agent.
  - **9DOF Motion Sensor**: The *i2c.I2C_9DOF* class can use the LSM9DS1 sensor to collect accelerometer, magnetometer, and gyroscopic data to compute unambiguous position and orientation information. We will be including calibration and computation routines that make it easier to extract properties of interest – eg. computing vertical motion by combining readings from the three sensors.
  - **Temperature Sensor**: The *i2c.MLX90640* class can use the MLX90640 sensor to measure temperature. The sensor is 32x24px, which the class can *interpolate()*. The class also allows frames to be integrated and averaged over time, substantially reducing noise. I modified the driver library to enable capture at the full 64fps on the Raspberry Pi.
- **NETWORKING** modules can stream continuous data better in a few ways:
  - Net_Node modules were given a *get_stream()* method that lets objects, well, stream data. Specifically, they are given a queue.Queue to shovel data into, which is then picked up by a dedicated zmq.Socket in its own thread, which handles batching, serialization, and load balancing. Streamed messages are batched (ie. contain multiple messages), but behave like normal message when received – they are split and contain an inner_key that is used to call the listen with each message (see l_stream()).
  - *networking* objects also now compress arrays-in-transit with the superfast blosc compression library. This increases their throughput dramatically, as many data streams in neuroscience are relatively low-entropy (eg. the pixels in a video of a mostly-white arena are mostly unchanged frame-to-frame and are thus highly compressible). See the *Message._serialize_numpy()* and Message._deserialize_numpy() methods.
- **STIMULI** - The *JackClient* can now play continuous sounds rather than discrete sounds. An example can be found in the Nafc_Gap task, which plays continuous white noise. All sounds now have a play_continuous() method, which continually dumps samples in a cycle into a queue for the *JackClient*. The continuous sound will be interrupted if another sound has its *Jack_Sound.play()* method called, but the continuous sound will resume seamlessly even if number of samples in the played sound aren't a multiple of the jack buffer size. We use this for gaps in noise (using the new *Gap* class), which we have confirmed are sample-accurate.
- **UI & VIZ**

- A *Video* window has been created to display streaming video. The Terminal_Networking.l_continuous() method meters frames such that even if high-speed video is being acquired, frames are only sent at a rate of prefs.DRAWFPS. The *Video* class uses the *ImageItem_TimedUpdate* object, a slight modification of pyqtgraph.ImageItem, that calls its update method according to a PySide2.QtCore.QTimer.
- A plots_menu menu has been added to the Terminal, and a GUI dialog (*gui.Psychometric*) has been added to create simple psychometric curves with the *viz.psychometric* module, which uses altair. Plans for developing visualization are described in *To-Do*.
- A general *gui.pop_dialog()* function simplifies displaying messages to the user using the Terminal UI. This was an initial step towards improving status/error reporting from other agents, further detailed in *To-Do*.

Bugfixes

- Some objects, particularly several *gui* objects, had the old *mouse/mice* terminology updated to *subject/subjects*.
- *Net_Node* objects were only implicitly destroyed by their release method which ends the threaded loop by setting the closing event.
- Embarassingly, *Pilot* objects were not prevented from running multiple tasks at a time. This led to some very confusing and hard-to-debug problems, as well as frequent conflicts over hardware access and resources. Typically what would happen is the Terminal would send a START message to begin a task, and if it wouldn't received a message receipt quickly enough would resend it, resulting in two tasks being started – but this would happen whenever two START messages were sent to a pilot. This was fixed with a simple check of Pilot.state before a task is initialized. Similar bugs were fixed in *Plot* objects.
- The *Subject* class would sometimes fail to get and increment the trial session. This has been fixed by saving the session number as an attribute in the info node.
- The *Subject* class would reset the session counter even when the same task was being reassigned (eg. if updated), now it preserves session number if the protocol name is unchanged.
- The *update_protocols()* method didn't report which subjects had their protocols updated, and so if there was some exception when setting new protocols it happened silently, making it so a user would never know their task was never updated. This was fixed with a noisier protocol update method for the Subject class and by displaying a list of subjects that were updated after the method is called.
- Correction trials were being calculated incorrectly by the *Stim_Manager*, such that rather than only repeating a stimulus *if the subject got the previous trial incorrect,* the stimulus was always repeated at least once.

Code Structure

- Modified versions of external libraries have been added as git submodules in *autopilot/external*.
- Requirements files have been split out to better differentiate between different agents and use-cases. eg. requirements for Terminal agents are in requirements/requirements_terminal.txt, requirements for build the docs are in requirements/requirements_docs.txt, etc. This is a temporary arrangement, as a future design goal is restructuring setup routines so that they can flexibly install components as-needed (see *To-Do*)
- autopilot.core.hardware has been refactored into its own module, *autopilot.hardware*, and split by device type, currently…
  - autopilot.cameras
  - autopilot.gpio - devices that use the GPIO pins for standard digital I/O logic
  - autopilot.i2c - devices that use the GPIO pins for I2C

    - `autopilot.usb`

- The docs are hosted on readthedocs again, so the docs structure has been collapsed to a single folder without built documentation
- The autopilot user directory is now `~/autopilot` rather than `/usr/autopilot`, which was always a mistake anyway. Autopilot creates a wayfinder `~/.autopilot` file that is used to find the user directory if it's set elsewhere

External Libraries

- External libraries can now be built and packaged along with autopilot using cmake, see CMakeLists.txt. Still uh having a little bit of trouble getting this to work, so code is in place to build and package the custom pigpio repo and jack audio but this will likely need some more work.
- pigpio https://github.com/sneakers-the-rat/pigpio/
    - Added the ability to return absolute timestamps rather than system ticks. pigpio typically returns 1 32-bit integer of ticks since the daemon started, absolute timestamps are 64-bit, so the pigpio daemon and python interface (*pi*) were given two new methods:
        * *synchronize* gets several (default 5) sets of paired timestamps and ticks using *get_sync_time*. It then computes an offset for translating ticks to timestamps
        * *ticks_to_timestamp* converts ticks to timestamps based on the offset found with *synchronize*
        * *get_current_time* sends two requests to the daemon to get the seconds and microseconds of the complete timestamp and returns an isoformatted string
- mlx90640-library https://github.com/pimoroni/mlx90640-library
    - Removed building examples by default which require additional dependencies
    - When using the raspi I2C driver, the baudrate would never be set to 1MHz, which is necessary to achieve full 64fps. This was fixed to use 1MHz by default.

Regressions

- Message confirmation (holding a message to resend if confirmation isn't received) was causing a huge amount of problems and needed to be rethought. There are in general very low rates (near-zero) of messages being dropped without some larger bug causing them, so confirmation has been disabled for now.
- The same is true of `heartbeat()` - which polled for status of connected pilots. this will be repaired and restored, as the terminal currently has a pretty bad idea of the status of what's connected to it. this will be part of a broader networking overhaul

22.3 Version 0.2

22.3.1 v0.2.0 (October 26, 2019)

Can't change what just started existing!

Release version of autopilot consistent with explanation in https://www.biorxiv.org/content/10.1101/807693v1

Development Roadmap, Minor To-dos, and all future plans :)

CHAPTER

TWENTYTHREE

TO-DO

23.1 Visions

The long view: design, ux, and major functionality projects roughly corresponding to minor semantic versions

23.1.1 Integrations

Make autopilot work with…

Open Ephys Integration

- write a C extension to the Rhythm API similar to that used by the OpenEphys Rhythm Node.
- Enable existing OE configuration files to be loaded and used to configure plugin, so ephys data can be collected natively alongside behavioral data.

Multiphoton & High-performance Image Integration

- Integrate the Thorlabs multiphoton imaging SDK to allow 2p image acquisition during behavior
- Integrate the Aravis camera drivers to get away from the closed-source spinnaker SDK

Bonsai Integration

- Write source and sink modules so Bonsai pipelines can be used within Autopilot for image processing, acquisition etc.

23.1.2 Closed-Loop Behavior & Processing Pipelines

- design a signal/slot architecture like Qt so that hardware devices

and data streams can be connected with low latency. Ideally something like:

```
# directly connecting acceleration in x direction
# to an LED's brightness
accelerometer.acceleration.connect('x', LED.brightness)

# process some video frame and use it to control task stage logic
camera.frame.transform(
    DLC, **kwargs
).connect(
    task.subject_position
)
```

- The pipelining framework should be concurrent, but shouldn't rely on `multiprocessing.Queue` s and the like for performance, as transferring data between processes requires it to be pickled/unpickled. Instead it should use shared memory, like `multiprocessing.shared_memory` available in Python 3.8
- The pipelining framework should be evented, such that changes in the source parameter are automatically pushed through the pipeline without polling. This could be done with a decorator around the `setter` method for the sender,
- The pipelining framework need not be written from scratch, and could use one of Python's existing pipelining frameworks, like
  - Joblib
  - Luigi
  - pyperator
  - streamz (love the ux of this but doesn't seem v mature)

- **Agents**
  - The Agent infrastructure is still immature—the terminal, pilot, and child agents are written as independent classes, rather than with a shared inheritance structure. The first step is to build a metaclass for autopilot agents that includes the different prefs setups they need and their runtime requirements. Many of the further improvements are discussed in the setup section
  - Child agents need to be easier to spawn and configure, and child tasks lack any formalization at all.
- **Parameters**
  - Autopilot has a lot of types of parameters, and at the moment they all have their own styles. This makes a number of things difficult, but primarily it makes it hard to predict which style is needed at any particular time. Instead Autopilot needs a generalized ``Param``eter class. It should be able to represent the human readable name of that parameter, the parameter's value, the expected data type, whether that parameter is optional, and so on.
  - The parameter class should also be recursive, so parameter sets are not treated distinctly from an individual parameter – eg. a task needs a set of parameters, one of which is a list of hardware. one hardware object in that list will have its own list of parameters, and so forth.
  - The parameter class should operate in both directions – ie. it should be able to represent *set* parameters, as well as be able to be used as a specifier of parameters that *need to be set*

- The parameter class should be cascading, where parameters apply to lower 'levels' of parameterization unless specified otherwise. For example, one may want to set `correction_trials` on for all stimuli in a task, but be able to turn them off for one stimulus in particular. To avoid needing to manually implement layered logic for all objects, handlers should be able to assume that a parameter will be passed from parent objects to their children.
- GUI elements should be automatically populating – some GUI elements are, like the protocol wizard is capable of populating a list of parameters from a task description, but it is incapable of choosing different types of stimulus managers, reading all their parameters, and so on. Instead it should be possible to descend through all levels of parameters for all objects in all GUI windows without duplicating the effort of implementing the parameterization logic every time.

- **Configuration & Setup**
  - Setup routines and configuration options are currently hard-coded into npyscreen forms (see `PilotSetupForm`). `prefs` setup needs to be separated into a model-view-controller type design where the available prefs and values are made separate from their form.
  - Setup routines should include both the ability to install necessary resources and the ability to check if those resources have been installed so that hardware objects can be instantiated freely without setup and configuration becoming cumbersome.
  - Currently, Autopilot creates a crude bash script with `setup_pilot.sh` to start external processes before Autopilot. This makes handling multiple environment types difficult – ie. one needs to close the program entirely, edit the startup script, and restart in order to switch from a primarily auditory to primarily visual experiment. Management of external processes should be brought into Autopilot, potentially by using sargehttps://sarge.readthedocs.io/en/latest/index.html or some other process management tool.
  - Autopilot should both install to a virtual environment by default and should have docker containers built for it. Further it should be possible to package up your environment for the purposes of experimental replication.
- **UI/UX**
  - The GUI code is now the oldest in the entire library. It needs to be generally overhauled to make use of the tools that have been developed since it was written (eg. use of networking modules rather than passing sets of variables around).
  - It should be much easier to read the status of, interact with, and reconfigure agents that are connected to the terminal. Currently control of Pilots is relatively opaque and limited, and often requires the user to go read the logs stored on each individual pilot to determine what is happening with it. Instead Autopilot should have an additional window that can be used to set the parameters, reconfigure, and test each individual Pilot.
  - There are some data -> graphical object mappings available to tasks, but Autopilot needs a fuller grammar of graphics. It should be possible to reconfigure plotting in the terminal GUI, and it should be possible to modify short-term parameters like bin widths for rolling means.
  - Autopilot shouldn't sprawl into a data visualization library, but it should have some basic post-experiment plotting features like plotting task performance and stages over time.
  - Autopilot should have a web interface for browsing data. We are undecided about building a web interface for controlling tasks, but it should be possible to download data, do basic visualization, and observe the status of the system from a web portal.
- **Tasks**
  - Task design is a bit *too* open at the moment. Tasks need to feel like they have more 'guarantees' on their operation. eg. there should be a generalized callback api for triggering events. the existing `handle_trigger()` is quite limited. There should be an obvious way for users to implement saving/reporting data from their tasks.

    * Relatedly, the creation of triggers is pretty awkward and not strictly threadsafe, it should be possible to identify triggers in subclasses (eg. a superclass creates some trigger, a subclass should be able to unambiguously identify it without having to parse method names, etc)

  - It's possible already to use a python generator to have more complex ordering of task stages, eg. instead of using an `itertools.cycle` one could write a generator function that yields task stages based on some parameters of the task. There should be an additional manager type, the `Trial_Manager`, that implements some common stage schemes – cycles, yes, but also DAGs, timed switches, etc. This way tasks could blend some intuitive features of finite-state machines while also not being beholden by them.

- **Mesh Networking**
  - Autopilot's networking system at the moment risks either a) being bottlenecked by having to route all data through a hierarchical network tree, or b) being indicipherable and impossible to program with as individual objects and streams are capable of setting up arbitrary connections that need to potentially be manually configured. This goal is very abstract, but Autopilot should have a mesh-networking protocol.
  - It should be possible for any object to communicate with any other object in the network without name collisions
  - It should be possible to stream data efficiently both point-to-point but also from one producer to many consumers.
  - It should be possible for networking connections to be recovered automatically in the case a node temporarily becomes unavailable.
  - Accordingly, Autopilot should adapt Zyre for general communications, and improve its file transfer capabilities so that it resembles something like bittorrent.

- **Data**
  - Autopilot's data format shouldn't be yet another standard incompatible with all the others that exist. Autopilot should at least implement data translators for, if not adopt outright the Neurodata Without Borders standard.
  - For distributed data acquisition, it makes sense to use a distributed database, so we should consider switching data collection infrastructure from .hdf5 files to a database system like PostgreSQL.

- **Hardware Library**
  - Populate https://auto-pi-lot.com/hardware with hardware designs, CAD files, BOMs, and assembly instructions
  - Make a 'thingiverse for experimental hardware' that allows users to browse hardware based on application, materials, etc.

23.2 Improvements

The shorter view: smaller, specific tweaks to improve functionality of existing features roughly corresponding to patches in semantic versioning.

- **Logging**
  - ensure that all events worth logging are logged across all objects.
  - ensure that the structure of logfiles is intuitive – one logfile per object type (networking, hardware rather than one per each hardware device)
  - logging of experimental conditions is incomplete – only the git hash of the pilot is stored, but the git hash of *all* relevant agents should be stored, and logging should be expanded to include `params` and system configuration (like `pip freeze`)

  - logs should also be made both human and machine readable – use prettyprint for python objects, and standardize fields present in logger messages.
  - File and Console log handlers should be split so that users can configure what they want to *see* vs. what they want *stored* separately (See https://docs.python.org/3/howto/logging-cookbook.html#multiple-handlers-and-formatters)

- **UI/UX**
  - Batch subject creation.
  - Double-clicking a subject should open a window to edit and view task parameters.
  - Drag-and-drop subjects between pilots.
  - Plot parameters should be editable - window roll size, etc.
  - Make a messaging routine where a pilot can display some message on the terminal. this should be used to alert the user about any errors in task operation rather than having to inspect the logs on the pilot.
  - The `Subject_List` remains selectable/editable once a subject has started running, making it unclear which subject is running. It should become fixed once a subject is running, or otherwise unambiguously indicate which subject is running.
  - Plot elements should have tooltips that give their value – eg. when hovering over a rolling mean, a tooltip should display the current value of the rolling mean as well as other configuration params like how many trials it is being computed over.
  - Elements in the GUI should be smarter about resizing, particularly the main window should be able to use a scroll bar once the number of subjects forces them off the screen.

- **Hardware**
  - Sound calibration - implement a calibration algorithm that allows speakers to be flattened
  - Implement OpenCL for image processing, specifically decoding on acquisition with OpenCV, with VC4CL. See
    * https://github.com/doe300/VC4CL/issues/29
    * https://github.com/thortex/rpi3-opencv/
    * https://github.com/thortex/rpi3-vc4cl/
  - Have hardware objects sense if they are configured on instantiation – eg. when an audio device is configured, check if the system has been configured as well as the hifiberry is in `setup/presetup_pilot.sh`

- **Synchronization**
  - Autopilot needs a unified system to generate timestamps and synchronize events across pilots. Currently we rely on implicit NTP-based synchronization across Pilots, which has ~ms jitter when configured optimally, but is ultimately not ideal for precise alignment of data streams, eg. ephys sampled at 30kHz. `pigpio` should be extended such that a Pilot can generate a clock signal that its children synchronize to. With the recent addition of timestamp generation within pigpio, that would be one parsimonious way of
  - In order to synchronize audio events with behavioral events, the *`JackClient`* needs to add a call to `jack_last_frame_time` in order to get an accurate time of when sound stimuli start and stop (See https://jackaudio.org/api/group__TimeFunctions.html)
  - Time synchronization between Terminal and Pilot agents is less important, but having them synchronized as much as possible is good. The Terminal should be set up to be an NTP server that Pilots follow.

- **Networking**

  - Multihop messages (eg. send to `C` through `A` and `B`) are clumsy. This may be irrelevant if Autopilot's network infrastructure is converted a true meshnet, but in the meantime networking modules should be better at tracking and using trees of connected nodes.
  - The system of zmq routers and dealers is somewhat cumbersome, and the new radio/dish pattern in zmq might be better suited. Previously, we had chosen not to use pub/sub as the publisher is relatively inefficient – it sends every message to every recipient, who filter messages based on their id, but the radio/dish method may be more efficient.
  - Network modules should use a thread pool for handling messages, as spawning a new thread for each message is needlessly costly
- **Data**
  - Data specification needs to be formalized further – currently data for a task is described with `tables` specifiers, `TrialData` and `ContinuousData`, but there are always additional fields – particularly from stimuli. The *`Subject`* class should be able to create columns and tables for
    * Task data as specified in the task description
    * Stimulus data as specified by a stimulus manager that initializes them. eg. the stimulus manager initializes all stimuli for a task, and then is able to yield a description of all columns needed for all initialized stimuli. So, for a task that uses
- **Tests** - Currently Autopilot has *no unit tests* (shocked ghasps, monocles falling into brandy glasses). We need to implement an automated test suite and continuous integration system in order to make community development of Autopilot manageable.
- **Configuration**
  - Rather than require all tasks be developed within the directory structure of Autopilot, Tasks and hardware objects should be able to be added to the system in a way that mimcs tensor2tensor's registry For example, users could specify a list of user directories in `prefs`, and user-created Hardware/Tasks could be decorated with a `@registry.register_task`.
    * This would additionally solve the awkward `tasks.TASK_LIST` method of making tasks available by name that is used now by having a more formal task registry.
- **Cleanliness & Beauty**
  - Intra-autopilot imports are a bit messy. They should be streamlined so that importing one class from one module doesn't spiral out of control and import literally everything in the package.
  - Replace `getter`- and `setter`-type methods throughout with `@properties` when it would improve the object, eg. in the *`JackClient`*, the storage/retrieval of all the global module variables could be made much neater with `@property` methods.
  - Like the *`Hardware`* class, top-level metaclasses should be moved to the `__init__` file for the module to avoid awkward imports and extra files like *`autopilot.tasks.task.Task`*
  - Use `enum.Enum` s all over! eg. things like *`autopilot.hardware.gpio.TRIGGER_MAP`* etc.
- **Concurrency**
  - Autopilot could be a lot smarter about the way it manages threads and processes! It should have a centralized registry of threads and processes to keep track on their status
  - Networking modules and other thread-creating modules should probably create thread pools to avoid the overhead of constantly spawning them
- **Decorators** - specific improvements to make autopilot objects magic!
  - *`hardware.gpio`* - try/catch release decorator so don't have to check for attribute error in every subclass!

23.3 Bugs

Known bugs that have eluded us thus far

- The *Pilot_Button* doesn't always reflect the availability/unavailability of connected pilots. The button model as well as the general heartbeating/status indication routines need to be made robust.
- The `pilot_db.json` and *Subject_List* doesn't check for duplicate subjects across Pilots. That shouldn't be a problem generally, but if a subject is switched between Pilots that may not be reflected in the generated metadata. Pilot ID needs to be more intimately linked to the *Subject*.
- If Autopilot needs to be quit harshly, some pigpio-based hardware objects don't quit nicely, and the pigpiod service can remain stuck on. Resource release needs to be made more robust
- Network connectivity can be lost if the network hardware is disturbed (in our case the router gets kicked from the network it is connected to) and is only reliably recovered by restarting the system. Network connections should be able to recover disturbance.
- The use of *off* and *on* is inconsistent between *Digital_Out* and *PWM* – since the PWM cleans values (inverts logic, expands range),
- There is ambiguity in setting PWM ranges: using *PWM.set()* with 0-1 uses the whole range off to on, but numbers from 0-*PWM.range* can be used as well – 0-1 is the preferred behavior, but should using 0-range still be supported as well?

23.4 Completed

good god we did it

- *v0.3.5 (February 22, 2021)* - Integrate DeepLabCut
- *v0.3.5 (February 22, 2021)* - Unify installation
- *v0.3.5 (February 22, 2021)* - Upgrade to Python 3
- *v0.3.5 (February 22, 2021)* - Upgrade to PySide 2 & Qt5
- *v0.3.5 (February 22, 2021)* - Generate full timestamps from pigpio rather than ticks
- *v0.3.5 (February 22, 2021)* - Continuous data handling
- *v0.3.5 (February 22, 2021)* - GPIO uses pigpio functions rather than python timing
- *v0.3.5 (February 22, 2021)* - networking modules compress arrays before transfer
- *v0.3.5 (February 22, 2021)* - Images can be acquired from cameras

23.5 Lowest Priority

Improvements that are very unimportant or strictly for unproductive joy

- **Classic Mode - in honor of an ancient piece of software that Autopilot may have descended from,** add a hidden key that when pressed causes the entire terminal screen to flicker whenever any subject in any pilot gets a trial incorrect.

CHAPTER

TWENTYFOUR

REFERENCES

CHAPTER

TWENTYFIVE

TESTS

25.1 Networking

Networking Tests.

Assumptions

- In docstring examples, `listens` callbacks are often omitted for clarity

Functions:

test_node(node_params)	*Net_Node* s can be initialized with their default parameters
test_node_to_node(node_params)	*Net_Node* s can directly send messages to each other with ROUTER/DEALER pairs.
test_multihop(node_params, station_params)	*Message* s can be routed through multiple *Station* objects by using a list in the `to` field

test_node(*node_params*)

Net_Node s can be initialized with their default parameters

test_node_to_node(*node_params*)

Net_Node s can directly send messages to each other with ROUTER/DEALER pairs.

```
>>> node_1 = Net_Node(id='a', router_port=5000)
>>> node_2 = Net_Node(id='b', upstream='a', port=5000)
>>> node_2.send('a', 'KEY', 'VALUE')
>>> node_2.send('b', 'KEY', 'VALUE')
```

test_multihop(*node_params*, *station_params*)

Message s can be routed through multiple *Station* objects by using a list in the `to` field

```
# send message:
# node_1 -> station_1 -> station_2 -> station_3 -> node_3
>>> station_1 = Station(id='station_1', listen_port=6000,
        pusher=True, push_port=6001, push_id='station_2')
>>> station_2 = Station(id='station_2', listen_port=6001,
        pusher=True, push_port=6002, push_id='station_3',)
>>> station_3 = Station(id='station_3', listen_port=6002)
>>> node_1 = Net_Node(id='node_1',
        upstream='station_1', port=6000)
>>> node_3 = Net_Node(id='node_3',
```

(continues on next page)

(continued from previous page)

```
        upstream='station_3', port=6002)
>>> node_1.send(key='KEY', value='VALUE',
        to=['station_1', 'station_2', 'station_3', 'node_3'])
```

25.2 Plugins

Functions:

hardware_plugin(default_dirs)	Make a basic plugin that inherits from the Hardware class, clean it up on exit
test_hardware_plugin(hardware_plugin)	A subclass of *autopilot.hardware.Hardware* in the PLUGINDIR can be accessed with `autopilot.get()`.
test_autoplugin()	the *autopilot.utils.registry.get()* function should automatically load plugins if the pref `AUTOPLUGIN` is `True` and the `plugins` argument is True

hardware_plugin(*default_dirs*) → Tuple[pathlib.Path, str]

Make a basic plugin that inherits from the Hardware class, clean it up on exit

Returns path to created plugin file

Return type Path

test_hardware_plugin(*hardware_plugin*)

A subclass of *autopilot.hardware.Hardware* in the `PLUGINDIR` can be accessed with `autopilot.get()`.

For example, for the following class declared in some `.py` file in the plugin dir:

```
from autopilot.hardware import Hardware

class Test_Hardware_Plugin(Hardware):
    def __init__(self, *args, **kwargs):
        super(Test_Hardware_Plugin, self).__init__(*args, **kwargs)

    def release(self):
        pass
```

one would be able to access it throughout autopilot with:

```
autopilot.get('hardware', 'Test_Hardware_Plugin')
# or
autopilot.get_hardware('Test_Hardware_Plugin')
```

test_autoplugin()

the *autopilot.utils.registry.get()* function should automatically load plugins if the pref `AUTOPLUGIN` is `True` and the `plugins` argument is True

25.3 Prefs

Functions:

test_prefs_defaults(default_pref)	
test_prefs_deprecation()	If there is a string in the `'deprecation'` field of a pref in *_DEFAULTS*, a warning is raised printing the string.

test_prefs_defaults(*default_pref*)

test_prefs_deprecation()

If there is a string in the `'deprecation'` field of a pref in *_DEFAULTS*, a warning is raised printing the string.

25.4 Registry

Data:

_EXPECTED_HARDWARE	A list of all the hardware we expect to have at the moment.

Functions:

logger_registry_get(caplog)	
test_get_one(base_class, class_name)	Get one autopilot object with a specified base class and class name using a string, an enum in autopilot.utils.registry.REGISTRIES, or an object itself
test_get_all(base_class)	Test that calling `get` with no `class_name` argument returns all the objects for that registry
test_get_subtree(logger_registry_get, caplog)	Test that calling `get` with a child of a top-level object (eg `GPIO` rather than `Hardware`) gets all its children, (using GPIO as the test case)
test_get_hardware()	use the *autopilot.utils.registry.get_hardware()* alias
test_get_task()	use the *autopilot.utils.registry.get_task()* alias
test_get_equivalence()	Test that the same object is gotten regardless of method of specifying base_class
test_except_on_failure()	Ensure a exceptions are raised for nonsense

_EXPECTED_HARDWARE = ('autopilot.hardware.cameras.Camera', 'autopilot.hardware.cameras.Camera_CV', 'autopilot.hardware.cameras.Camera_Spinnaker', 'autopilot.hardware.gpio.Digital_In', 'autopilot.hardware.gpio.Digital_Out', 'autopilot.hardware.gpio.GPIO', 'autopilot.hardware.gpio.LED_RGB', 'autopilot.hardware.gpio.PWM', 'autopilot.hardware.gpio.Solenoid', 'autopilot.hardware.i2c.I2C_9DOF', 'autopilot.hardware.i2c.MLX90640', 'autopilot.hardware.usb.Scale', 'autopilot.hardware.usb.Wheel')

A list of all the hardware we expect to have at the moment.

This doesn't need to be maintained *exactly*, but is just used as an independent source of expectation for which

Hardware objects we can expect.

So in all tests that use it, this tests a **minimal** expectation, ie. that we get all the values that we should get if this were up to date, knowing that it might not be.

logger_registry_get(*caplog*)

test_get_one(*base_class*, *class_name*)
Get one autopilot object with a specified base class and class name using a string, an enum in autopilot.utils.registry.REGISTRIES, or an object itself

test_get_all(*base_class*)
Test that calling `get` with no `class_name` argument returns all the objects for that registry

test_get_subtree(*logger_registry_get*, *caplog*)
Test that calling `get` with a child of a top-level object (eg `GPIO` rather than `Hardware`) gets all its children, (using GPIO as the test case)

test_get_hardware()
use the `autopilot.utils.registry.get_hardware()` alias

mostly a formality to keep it working since the underlying function is tested elsewhere

test_get_task()
use the `autopilot.utils.registry.get_task()` alias

mostly a formality to keep it working since the underlying function is tested elsewhere

test_get_equivalence()
Test that the same object is gotten regardless of method of specifying base_class

test_except_on_failure()
Ensure a exceptions are raised for nonsense

25.5 Setup

Functions:

`test_make_alias`()	
`test_quiet_mode`()	Autopilot can be setup programmatically by calling setup_autopilot with –quiet and passing prefs and scripts manually

test_make_alias()

test_quiet_mode()
Autopilot can be setup programmatically by calling setup_autopilot with –quiet and passing prefs and scripts manually

25.6 Sounds

Tests for generating sound stimuli.

This script runs tests that generate different sound stimuli and verifies that they are initialized correctly.

Currently these only work if AUDIOSERVER is 'jack'. 'pyo' is not tested. 'docs' doesn't actually generate waveforms.

This doesn't require (or test) a running jackd or even a JackClient. Instead, these tests short-circuit those dependencies by manually setting FS and BLOCKSIZE in autopilot.stim.sound.jackclient.

A TODO is to test the JackClient itself.

Currently only the sound Noise is tested.

These tests cover multiple durations and amplitudes of mono and multi-channel Noise, including some edges cases like very short durations or zero amplitude.

The rest of this docstring addresses the workaround used to short-circuit jackd and JackClient.

Here is the sequence of events that leads to FS and BLOCKSIZE. * If an autopilot.core.pilot.Pilot is initialized: ** autopilot.core.pilot.Pilot.__init__ checks prefs.AUDIOSERVER,

and calls autopilot.core.pilot.Pilot.init_audio.

**** autopilot.core.pilot.Pilot.init_audio calls** autopilot.external.__init__.start_jackd.

**** autopilot.external.__init__.start_jackd takes the JACKDSTRING pref** and replaces the token '-rfs' in it with the FS pref. The jackd process is launched and stored in autopilot.external.JACKD_PROCESS. That process may fail or not, we continue anyway.

**** Next, autopilot.core.pilot.Pilot.init_audio instantiates an** autopilot.stim.sound.jackclient.JackClient()

**** autopilot.stim.sound.jackclient.JackClient.__init__** initalizes a jack.Client

**** autopilot.stim.sound.jackclient.JackClient.fs** is set to jack.Client.samplerate. Note that this is either the requested sample rate, or some default value from jack (not Autopilot) if the client did not actually succeed in booting.

**** autopilot.stim.sound.jackclient.FS (a global variable) is set to** autopilot.stim.sound.jackclient.JackClient.fs

- Later, a sound (e.g., Noise) is initialized.

** autopilot.stim.sound.sounds.Noise.__init__ calls super().__init__, ** which is autopilot.stim.sound.sounds.Jack_Sound.__init__ ** autopilot.stim.sound.sounds.Jack_Sound.__init__

sets *self.fs* to jackclient.FS

**** autopilot.stim.sound.sounds.Noise.__init__ calls** autopilot.stim.sound.sounds.Noise.init_sound

**** autopilot.stim.sound.sounds.Noise.init_sound calls** autopilot.stim.sound.sounds.Jack_Sound.get_nsamples

**** autopilot.stim.sound.sounds.Jack_Sound.get_nsamples** inspects *self.fs*

To remove the dependence on jackd2 and JackClient, the entire first block of code can be circumvented by setting these: autopilot.stim.sound.jackclient.FS autopilot.stim.sound.jackclient.BLOCKSIZE

Functions:

`test_init_noise`(duration_ms, amplitude, ...)	Initialize and check a mono (single-channel) noise.
`test_init_multichannel_noise`(duration_ms, ...)	Initialize and check a multi-channel noise.

test_init_noise(*duration_ms*, *amplitude*, *check_duration_samples*, *check_n_chunks_expected*)
Initialize and check a mono (single-channel) noise.

A mono *Noise* is initialized with specified duration and amplitude. The following things are checked: * The attributes should be correctly set * The *table* should be the right dtype and the right duration,

given the sampling rate

- The chunks should be correct, given the block size. The last chunk should be zero-padded.
- The waveform should not exceed amplitude anywhere
- As long as the waveform is sufficiently long, it should exceed 90% of the amplitude somewhere
- Concatenating the chunks should generate a result equal to the table, albeit zero-padded to a multiple of the block size.
- Specifying channel as None should give identical results to leaving it unspecified.

duration_ms : passed as *duration* amplitude : passed as *amplitude* check_duration_samples : int or None

If not None, the length of the sounds *table* should be this

check_n_chunks_expected [int or None] If not None, the length of the sounds *chunks* should be this

`test_init_multichannel_noise`(*duration_ms*, *amplitude*, *channel*, *check_duration_samples*, *check_n_chunks_expected*)

Initialize and check a multi-channel noise.

A multi-channel *Noise* is initialized with specified duration, amplitude, and channel. The following things are checked: * The attributes should be correctly set * The *table* should be the right dtype and the right duration,

given the sampling rate

- The chunks should be correct, given the block size. The last chunk should be zero-padded.
- The column *channel* should contain non-zero data and all other columns should contain zero data.
- The waveform should not exceed amplitude anywhere
- As long as the waveform is sufficiently long, it should exceed 90% of the amplitude somewhere
- Concatenating the chunks should generate a result equal to the

duration_ms : passed to *Noise* as *duration* amplitude : passed to *Noise* as *amplitude* channel : passed to *Noise* as *channel* check_duration_samples : int or None

If not None, the length of the sounds *table* should be this

check_n_chunks_expected [int or None] If not None, the length of the sounds *chunks* should be this

25.7 Terminal

25.8 Transforms

25.9 Utils

CHAPTER

TWENTYSIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[ABCO15] Fatemeh Abyarjoo, Armando Barreto, Jonathan Cofino, and Francisco R. Ortega. Implementing a Sensor Fusion Algorithm for 3D Orientation Detection with Inertial/Magnetic Sensors. In Tarek Sobh and Khaled Elleithy, editors, *Innovations and Advances in Computing, Informatics, Systems Sciences, Networking and Engineering*, Lecture Notes in Electrical Engineering, 305–310. Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-06773-5_41.

[KLS+20] Gary A Kane, Gonçalo Lopes, Jonny L Saunders, Alexander Mathis, and Mackenzie W Mathis. Real-time, low-latency closed-loop feedback using markerless posture tracking. *eLife*, 9:e61909, December 2020. doi:10.7554/eLife.61909.

[PPT+18] Photis Patonis, Petros Patias, Ilias N. Tziavos, Dimitrios Rossikopoulos, and Konstantinos G. Margaritis. A Fusion Method for Combining Low-Cost IMU/Magnetometer Outputs for Use in Applications on Mobile Devices. *Sensors (Basel, Switzerland)*, August 2018. doi:10.3390/s18082616.

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y