bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

JONNY L. SAUNDERS, MICHAEL WEHR@

University of Oregon
Institute of Neuroscience, Department of Psychology
Eugene, OR 97403, United States

AUTOPILOT

AUTOMATING BEHAVIORAL EXPERIMENTS WITH LOTS OF RASPBERRY PIS

Neuroscience needs behavior, and behavioral experiments require the coordination of large numbers of heterogeneous hardware components and data streams. Currently available tools strongly limit the complexity and reproducibility of experiments. Here we introduce Autopilot, a complete, open-source Python framework for behavioral neuroscience that distributes experiments over networked swarms of Raspberry Pis. Autopilot enables qualitatively greater experimental flexibility by allowing arbitrary numbers of hardware components to be combined in arbitrary experimental designs. Research is made reproducible by documenting all data and task design parameters in a human-readable and publishable format at the time of collection. Autopilot provides an order-of-magnitude performance improvement over existing tools while also being an order of magnitude less costly to implement. Autopilot's flexible, scalable architecture allows neuroscientists to design the next generation of experiments to investigate the behaving brain.

HOMEPAGE

REPOSITORY

DOCUMENTATION

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Contents

1 Introduction 3
- 1.1 Existing Systems for Behavioral Experiments 5
- 1.2 Limitations of Existing Systems 6

2 Design 9
- 2.1 Efficiency 9
- 2.2 Flexibility 11
- 2.3 Reproducibility 14

3 Program Structure 17
- 3.1 Tasks 17
- 3.2 Hardware 22
- 3.3 Stimuli 23
- 3.4 Agents - Terminal, Pilot, and Child 24
- 3.5 Networking 27
- 3.6 GUI & Plots 28

4 Tests 29
- 4.1 Latency 29
- 4.2 Bandwidth 30
- 4.3 Distributed Go/No-go Task 31

5 Limitations and Future Directions 34

6 Glossary 36

Bibliography 38

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

1
Introduction

Animal behavior experiments require precision and repetition, which can best be achieved by computer automation. The complexity of contemporary behavioral experiments, however, presents a stiff methodological challenge. For example, researchers might wish to measure pupil dilation[28, 28], respiration[25], and running speed[24], while tracking the positions of body parts in 3 dimensions[23] and recording the activity of large ensembles of neurons[17], as subjects perform tasks with custom input devices such as a steering wheel[4] while immersed in virtual reality environments using stimuli synthesized in real time[34, 7]. Coordinating the array of necessary hardware into a coherent experimental design—with the millisecond precision required to study the brain—can be daunting.

Historically, researchers have developed software to automate behavior experiments as-needed within their lab or relied on purchasing proprietary software (eg. [11]). Open-source alternatives have emerged recently, often developed in tandem with hardware peripherals available for purchase [13, 30]. However, the diverse hardware and software requirements for behavioral experiments often lead researchers to cobble together multiple tools to perform even moderately complex experiments. Indeed, most software packages do not attempt to simultaneously support custom hardware operation, behavioral task logic, stimulus generation, and data acquisition. Idiosyncratic systems can hinder reproducibility, especially if the level of detail reported in a methods section is sparse[37]. Additionally, development time and proprietary software are expensive, as are the custom hardware peripherals that are required to use most available open-source behavior software.

Here we present Autopilot, a complete open-source software and hardware framework for behavioral experiments. We leverage the power of distributed computing using the surprisingly capable Raspberry Pi 4[1] to allow researchers to coordinate arbitrary numbers of heterogeneous hardware components in arbitrary experimental designs.

[1] See Table 3.2

Autopilot takes a different approach than existing systems to overcome the technical challenges of behavioral research: *just use more computers*. Specifically, the advent of inexpensive single-board computers (ie. the Raspberry Pi) that are powerful enough to run a full Linux operating system allows a unified platform to run on every Pi or other computer in the system so that they can work together seamlessly. At

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

the core of its architecture is a networking protocol (Section 3.5) that is fast enough to stream electrophysiological or imaging data and flexible enough to make the mutual coordination of hardware straightforward.

This distributed design also makes Autopilot extremely scalable, as the Raspberry Pi's $35 price tag makes it an order of magnitude less costly than comparable systems (Section 2.3). Its low cost doesn't come at the expense of performance or useability: Autopilot also has an order of magnitude greater measurement precision and an order of magnitude lower latency than comparable systems (Sections 2.1 and 4).

Autopilot balances experimental flexibility with support. Its task design infrastructure is flexible enough to perform arbitrary experiments, but also provides support for data management, plotting task progress, and custom training regimens. We provide a set of modular tools for users to easily build common tasks (such as the two-alternative forced choice task described in Section 3.1), and have also written complete low-level API documentation to facilitate any tinkering needed to make Autopilot do whatever is needed. Rather than relying on costly proprietary hardware modules, users can take advantage of the wide array of peripherals and extensive community support available for the Raspberry Pi.

Finally, we have designed Autopilot to do reproducible research. Experiments are not written as scripts that are reliant on the particularities of each researcher's hardware configuration. Instead, we have designed the system to encourage users to write reusable, portable experiments that are incorporated into a public central library. Every parameter that defines an experiment is automatically saved in publication-ready format, removing ambiguity in reported methods and facilitating exact replication with a single file.

We begin by defining the requirements of a complete behavioral system and evaluating two current examples (Sections 1.1 and 1.2). We then describe Autopilot's design principles (Section 2) and how they are implemented in the program's structure (Section 3). We close with a demonstration of its current capabilities and our plans to expand them (Sections 4 and 5).

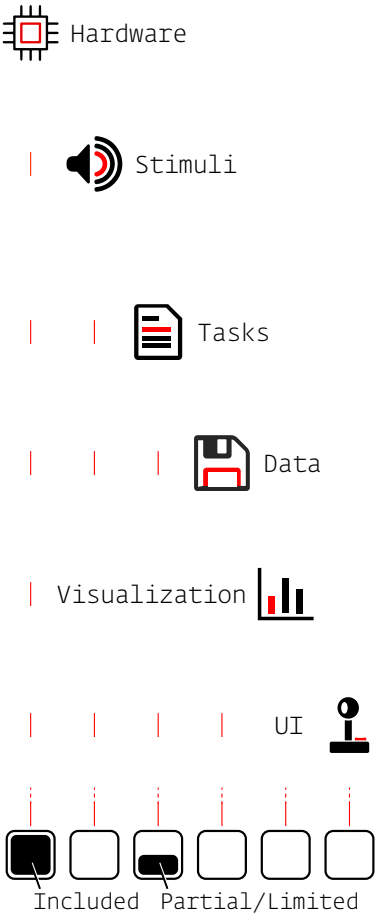
We would like to acknowledge and thank Lucas Ott for doing much of the behavioral training, Brynna Paros and Nick Sattler for their help with constructing our behavioral boxes, Matt Smear and Reese Findley for loaning us their Bpod for far longer than they intended to, Erik Flister whose Ratrix software inspired some of the design features of Autopilot [22], several artists on `flaticon.com` (Freepik, Nikita Golubev, Those Icons) whose work served as stems for many of the figures, and the Janet Smith House for the endless support and relentless criticism of the figures. This material is based on work supported by NIH NIDCD R01 DC-015828, NSF Graduate Research Fellowship No. 1309047, and a University of Oregon Incubating Interdisciplinary Initiatives award.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

1.1 *Existing Systems for Behavioral Experiments*

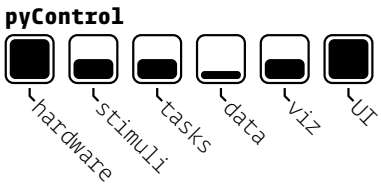
A complete system to automate behavioral experiments has 6 requirements:

1. Hardware to interact with the experimental subject, including **sensors** (eg. photodiodes, cameras, rotary encoders) to receive input and **actuators** (eg. lights, motors, solenoids) to provide feedback.
2. Some capability to synthesize and present sensory stimuli. Ideally both discrete stimuli, like individual tone pips or grating patches, and continuous stimuli, like those used in virtual reality experiments, should be possible.
3. A framework to coordinate hardware and stimuli as a task. Task definition should be flexible such that it facilitates rather than constrains experimental design.
4. A data management system that allows fine control of data collection and format. Data should be human readable and include complete metadata that allows independent analysis and reproduction.
5. Some means of visualizing data as it is collected in order to observe task status. It should be possible to customize visualization to the needs and structure of the task.
6. Finally, a user interface to control task operation. The UI should make it possible for someone who does not program to operate the system.



We will briefly describe two other systems that are "complete" as described above: pyControl and Bpod.

pyControl



pyControl is a behavioral framework built in Python by the Champalimaud Foundation. It uses the micropython microcontroller ("pyboard") as its primary hardware device along with several extension boards sold by openephys. The pyboard has four I/O ports, or eight with a multiplexing expander board. Schematics are available for many other hardware components like solenoid valve drivers and rotary encoders. Multiple pyboards can be connected to a computer via USB and run independent tasks simultaneously with a GUI.

There is limited support for some parametrically defined sound stimuli, presented from a separate amplifier connected using the I2C protocol. Visual stimuli are unsupported.

Like most behavioral software, pyControl uses a finite-state machine formalism to define its tasks. A task is a set of discrete states, each

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

of which has a set of events that transition the task from one state to another. pyControl also allows timed transitions between states, and one function that is called on every event for a rough sort of parallelism.

The facility for data management is limited. All events and states are stored alongside timestamps as a plain text log file, one file per subject per session (Figure 1.1).

```
D 0 2
D 8976 3
D 8976 1
P 8976 Print Statement
D 10162 3
D 10163 2
```

Figure 1.1: pyControl data is stored as plain text, each line having a type (**D**ata or **P**rint), timestamp, and state

There is only one plot type available in the GUI, a raster plot of events, and no facility for varying the plot by task type. The GUI is otherwise quite capable, including the ability to batch run subjects, redefine task variables, and configure hardware.

Bpod

Bpod is primarily a collection of hardware designs and an assembly service run by Sanworks LLC. Similar to pyControl, each Bpod behavior box is based on a finite-state machine microcontroller with four I/O ports. Additional hardware modules provide extended functionality.

The software that runs Bpod is a sparsely documented MATLAB package. A task is implemented as a MATLAB script that constructs a new state machine for each trial, uploads it to the Bpod, and waits for the trial to finish. As a result, only one Bpod can be used per host computer, or at least per MATLAB session. Data are stored as trial-split events in a MATLAB structure.

There are a few basic plots for two-alternative forced choice tasks, but any plotting is done in the main loop so the MATLAB graphics engine blocks the program between trials. Bpod has a reasonably complete GUI for managing the hardware and running tasks, but it is error-prone and highly technical (Figure 1.2).

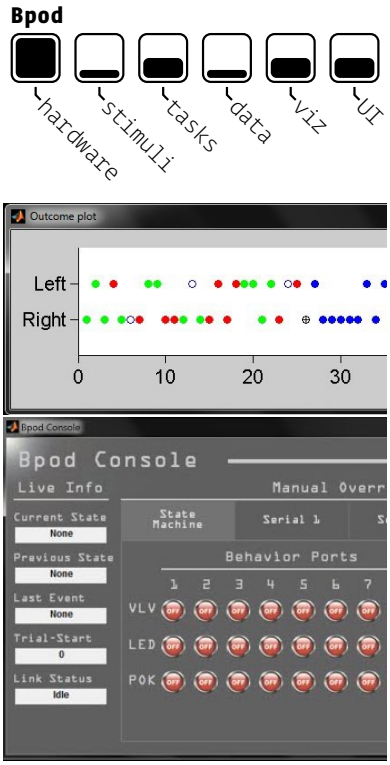


Figure 1.2: A Bpod event plot (above) showing the results of individual behavioral trials, and the Bpod GUI (below).

For brevity we have omitted many other excellent tools that perform some subset of the operations of a complete behavioral system.[2]

1.2 *Limitations of Existing Systems*

We see several limitations with these and other behavioral systems:

- **Hardware** - Both Pycontrol and Bpod strongly encourage users to purchase a limited set of hardware modules and add-ons from their particular hardware ecosystem. If a required part is not available for purchase, neither system provides a clear means of interacting with custom hardware, requiring the user to 'tack on' loosely-integrated components—we found one such lab using a Raspberry Pi to deliver sounds in their Bpod task. There is also a hard limit on the *number* of hardware peripherals that can be used in any given task, as there

[2] **Other tools:**
- Expyriment[20] - site, git
- PsychoPy[27] - site, git
- OpenSesame[21] - site, git
- SMiLE - docs
- ArControl[8] - git
- and see OpenBehavior

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

is no ability to use additional pyboards or Bpod state machines. The microcontrollers used in these systems also impose strong limits on their software: neither run a full, high-level programming language[3]. We will discuss this further in section 2.2. A broader limitation of existing systems is the difficulty of flexibly integrating the diverse hardware and analytical tools necessary to perform the next generation of behavioral neuroscience experiments that study "naturalistic, unrestrained, and minimally shaped behavior"[10].

[3] Bpod runs custom firmware written in C++ on a Teensy 3.6 microcontroller. pyControl's pyboard implements micropython, a subset of Python that excludes canonical libraries like numpy[35] or scipy[16]

- **Stimuli** - Stimuli are not tightly integrated into either of these systems, requiring the user to write custom routines for their synthesis, presentation, and description in the resulting data. Neither are capable of delivering visual stimuli. Bpod only supports raw audio waveforms presented with either a proprietary analog output hardware module or using PsychToolbox from the host computer. Some parametric audio stimuli are included in the pyControl source code but we were unable to find any documentation or examples of their use.

- **Tasks** - Tasks in both systems require a large amount of code and effort duplication. Neither system has a notion of reusable tasks or task 'templates,' so every user needs to rewrite every task from scratch. Bpod's structure in particular tends to encourage users to write long task scripts that are difficult to read (Figure 1.3) because much of its codebase is 'backend' code for compiling and communicating with the state machine, so users have to write basic routines like stimulus creation themselves. Another factor that contributes to the difficulty of task design in these systems is the need to work around the limitations of finite state machines, which we discuss further in section 3.1.

- **Data** - Data storage and formatting is basic, requiring extensive additional processing to make it human readable. For example, to determine whether a subject got a trial correct in an example Bpod experiment, one would use the following code:

  ```
  SessionData.RawEvents.Trial{1,1}.States.Punish(1) ~= NaN
  ```

  As a result, data format is idiosyncratic to each user, making data sharing dependent on manual annotation and metadata curation from investigators. Additionally, since the parameters of experiments are not saved by default—and the GUIs of both systems allow parameters to be changed at will—critical data could be lost and experiments could be made unreproducible unless the user writes custom code to save them.

```
for currentTrial = 1:MaxTrials
% new state matrix every trial
sma = NewStateMatrix();

% add states and transitions
sma = AddState(sma,
    'Name', 'Wait', ...
    'Timer', 0,...
    'StateChangeConditions', ...
    {'Port2In', 'Delay'}, ...
    'OutputActions', ...
    {'AudioPlayer1','*'});
% add more states...

% upload and run task
SendStateMatrix(sma);
RawEvents = RunStateMatrix;

% manually gather data and params
BpodSystem.Data = AddTrialEvents(
    BpodSystem.Data, RawEvents);

% plotting in the main loop
UpdateSideOutcomePlot(...);
UpdateTotalRewardDisplay(...);

% manually save data
SaveBpodSessionData;
end
```

Figure 1.3: Bpod's general task structure.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

- **Visualization & GUI** - The GUIs of each of these systems are highly technical, and are not designed to be easily used by non-programmers. Visualization of task progress is quite rigid in both systems, either a timeseries of task states or plots specific to two-alternative forced choice tasks. There is no obvious way to adapt plots to specific tasks.

In short, existing systems for behavioral experiments are limited by the hardware they can use, the tasks they can implement, and the ease with which they can be implemented. Some of these limitations are cosmetic—fixable with additional code or hardware—but several of the most crucial are intrinsic to the design of these systems.

These systems, among others, have pioneered the development of modern behavioral hardware and software, and are to be commended for being open-source and highly functional. One need look no further for evidence of their usefulness than to their adoption by many labs worldwide. At the time that these systems were developed, a general-purpose single-board computer with performance like the Raspberry Pi 4 was not widely available. The above two systems are not unique in their limitations, but are reflective of broader constraints of developing experimental tools. We are only able to articulate the design principles that differentiate Autopilot by building on their work.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

2
Design

Autopilot distributes experiments across a network of Raspberry Pis,[1] a type of inexpensive single-board computer.

[1] Raspberry Pi model 4B, see Table 3.2

Autopilot has three primary design principles:

1. **Efficiency** - Autopilot should minimize computational overhead and maximize use of hardware resources.
2. **Flexibility** - Autopilot should be transparent in all its operations so that users can expand it to fit their use-case.
3. **Reproducibility** - Autopilot should maximize standardization and minimize the potential for the black-box of local reprogramming. Autopilot should maximize the information it stores about its operation as part of normal data collection.

2.1 *Efficiency*

Though it is a single board, the Raspberry Pi operates more like a computer than an integrated circuit. It most commonly runs a custom Linux distribution, Raspbian, allowing Autopilot to use Python across the whole system. Using an interpreted language like Python running on Linux has inherent performance drawbacks compared to compiled languages running on embedded microprocessors. While Python's overhead is negligible on modern processors, Autopilot is nevertheless designed to maximize computational efficiency.

Concurrency

Most behavioral software is single-threaded (Figure 2.1), meaning the program will only perform a single operation at a time. If the program is busy or waiting for an input, other operations are blocked until it is finished.

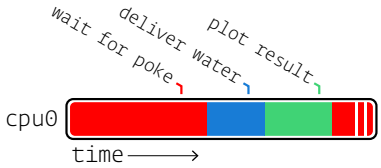


Figure 2.1: A single-threaded program executes all operations sequentially, using a single process and cpu core.

Autopilot distributes computation across multiple processes and threads to take advantage of the Raspberry Pi's four CPU cores. Every object in Autopilot does its work in separate **threads**. Specifically, Autopilot spawns separate threads to process messages and events, an architecture described more fully in section 3.5. Threading does not offer true concurrency[2], but does allow Python to distribute computational time between operations so that, for example, waiting for an

[2] See David Beazley's 'Understanding the Global Interpreter Lock' and associated visualizations.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

event does not block the rest of the program, and events are not missed because the program is busy (Figure 2.2).

Critical operations that are computationally intensive or cannot be interrupted are given their own dedicated **processes**. Linux allows individual cores of a processor to be reserved for single processes, so individual Raspberry Pis are capable of running four truly parallel processing streams. For example, all Raspberry Pis in an Autopilot swarm create a messaging client to handle communication between devices which runs on its own processor core so no messages are missed. Similarly, if an experiment requires sound delivery, a realtime sound engine in a separate process (Figure 2.3) also runs on its own core.

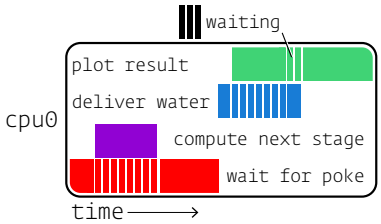


Figure 2.2: A multi-threaded program divides computation time of a single process and cpu core across multiple operations so that, for example, waiting for input doesn't block other operations.

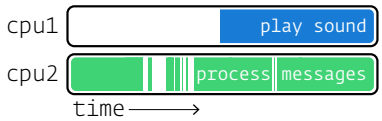


Figure 2.3: A multi-process program is truly concurrent, allowing multiple cpu cores to operate in parallel.

Leveraging Low-Level Libraries

Autopilot uses Python as a "glue" language, where it wraps and coordinates faster low-level compiled code[36]. Performance-critical components of Autopilot are thin wrappers around fast C libraries (Table 2.1).

Table 2.1: A few libraries Autopilot uses

jack	realtime audio
pigpio	GPIO control
ZeroMQ	networking
Qt	GUI

Since Autopilot coordinates its low-level components in parallel rather putting everything inside one "main loop," Autopilot actually has *better* temporal resolution than single-threaded systems like Bpod or pyControl, despite the realtime nature of their dedicated processors (Table 2.2).

Table 2.2: Using pigpio as a dedicated I/O process gives autopilot greater measurement precision

	Precision
Autopilot (pigpio)	$5\mu s$
Bpod	$100\mu s$
pyControl	$1000\mu s$

Caching

Finite-state machines are only aware of the current state and the events that transition it to future states. They are thus incapable of exploiting the often predictable structure of behavioral tasks to precompute future states and precache stimuli. Further, to change task parameters between trials (eg. changing the rewarded side in a two-alternative forced-choice task), state machines need to be fully reconstructed and reuploaded to the device that runs them each time.

Autopilot precomputes and caches as much as possible. Rather than wait "inside" a state, Autopilot prepares each of the next possible events and saves them for immediate execution when the appropriate trigger is received. Static stimuli are prepared once at the beginning of a behavioral session and stored in memory. Before their presentation, they are buffered to minimize latency.

Autopilot's efficient design lets it access the best of both worlds—the speed and responsiveness of compiled code on dedicated microprocessors and the accessibility and flexibility of interpreted code.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

2.2 *Flexibility*

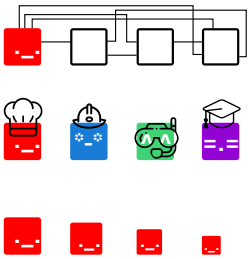
Single-language

Behavior software that uses dedicated microprocessors like Bpod must have some routine for compiling the high-level abstraction of the experiment into machine code. This gives those systems a theoretical advantage in processing speed, but the compiler becomes the bottleneck of complexity: only those things that can be compiled can be included in the experiment. This may in part contribute to the ubiquity of state-machine formalisms in behavior software.

Because Python is used throughout the system, extending Autopilot's functionality is straightforward. Task design (see section 3.1) is effectively arbitrary—anything that can be expressed in Python is a valid task. Hardware can also be implemented arbitrarily, including hardware that makes use of external libraries (eg. ACQ4[6] and our planned integration with OpenEphys).

Modularity

Although Autopilot deeply integrates with the Raspberry Pi's hardware, we have also worked to make its components modular. Modularity has 3 primary advantages:



1. **Modularity makes code more flexible** by reducing the constraints imposed by unstructured code interdependencies
2. **Modularity makes code more intelligible** by logically distributing tasks to discrete classes
3. **Modularity reduces effort-duplication** by allowing multiple, similar classes to be created with inheritance rather than copying and pasting.

There is no such thing as "incompatible hardware" with Autopilot because the classes that control hardware are independent from the code that provides other core functionality. In systems without modular design, hardware implementation is spread across the codebase; for example to add a new type of hardware output to a Bpod system, one would need to write new firmware for it in C, modify Bpod's existing firmware, hunt through the code to modify how states are added and state machines are assembled, add its controls explicitly to the GUI, and so on.

Tasks specify what type of hardware is needed to run them, but are agnostic about the way the hardware is implemented, making their descriptions more portable. Tasks that have the same structure but differ in hardware (eg. a freely moving two-alternative forced

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

choice task in which a mouse visits several IR sensors, or a head-fixed two-alternative forced choice task in which a mouse runs on a wheel to indicate its choice) can be implemented by a trivial subclass that modifies the hardware description rather than completely rewriting the task.

Structured Expansion & Code Transparency

We call Autopilot a software framework because in addition to providing classes and methods to run experiments out of the box, it also provides explicit structure that scaffolds any additional code that is needed by the user. Our goal is to clearly articulate in the documentation how modules should interact so that anyone can write code that works on any apparatus.

Autopilot is designed for users with a range of programming expertise, from those who only want to interact with a GUI, to those who wish to fundamentally rewrite core operations for their particular experiment. As such, it is extensively documented: this paper provides a high-level introduction to its design and structure, its user guide describes how to use the program and provides examples, and its API-level documentation describes in granular detail how the program actually works[3]. Nothing is "off-limits" to the user—there isn't any hidden, undocumented hardware code behind the curtain. We want users to be able to understand how and why everything works the way it does so that Autopilot can be adapted and expanded to any use-case.

[3] The user guide and API documentation are available at `docs.auto-pi-lot.com`

A broader goal of Autopilot is to build a library of flexible task prototypes that can be tweaked and adapted, hopefully reducing the number of times the wheel is reinvented. We have attempted to nudge users to write reusable tasks by designing Autopilot such that rather than writing separate task scripts that are loaded and run by the program, tasks are written into a fork[4] of the library itself. When publishing research that uses a particular task, users are incentivized to pull the changes they have made in their fork back into the central library because doing so makes that task available to anyone using Autopilot. Autopilot's documentation is automatically generated from structured comments[5], which naturally establishes a minimal level of documentation that we will require to have a task accepted into the main library. We hope the combination of these design nudges and explicit development instructions in the user guide encourages users to make contributing well-documented, reusable tasks a normal part of using Autopilot.

[4] Autopilot is version controlled using git. Users develop tasks in a copy, or 'fork' of the library that keeps track of their changes so that they can later be re-integrated, or 'pulled', into the main library.

[5] using Sphinx

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Message Handling

Modular software needs a well-defined protocol to communicate between modules, and Autopilot's is heavily influenced by the concurrency philosophy[6] of ZeroMQ[15]. All communication between computers and modules happens with ZeroMQ messages, and handling those messages is the main way that Autopilot handles events. A key design principle is that Autopilot components should not "share state"—they can communicate, but they are not *dependent* on one another. While this may seem like a trivial detail, having networking and message-handling at its core has three advantages that make Autopilot a fundamental departure from previous behavioral software.

[6] "ZeroMQ [...] has a subversive effect on how you develop network-capable applications. [...] message processing rapidly becomes the central loop, and your application soon breaks down into a set of message processing tasks."

"If there's one lesson we've learned from 30+ years of concurrent programming, it is: *just don't share state.*"

-The ZeroMQ Guide

First, new software modules can be added to any system by simply dropping in a standalone networking object. There is no need to dramatically reorganize existing code to make room for new functionality. Instead new modules can receive, process, and send information by just connecting to a parent module in the swarm. For example, each plot opens a network connection to stream incoming task data independently from the stream that is saving the data.

Second, Autopilot can be made to interact with other software libraries that use ZeroMQ. For example, The OpenEphys GUI for electrophysiology can send and receive ZMQ messages to execute actions such as starting or stopping recordings. Interaction with other software is also useful in the case that some expensive computation needs to happen mid-task. For example, one could send frames captured from a video camera on a Raspberry Pi to a GPU computing cluster for tracking the position of the animal. Since ZeroMQ messages are just TCP packets it is also possible to communicate over the internet for remote control or to communicate with a data server.

Third, making every component network-capable allows tasks to be distributed over multiple Raspberry Pis. Chaining multiple Pis distributes the computational load, allowing, for example, one Raspberry Pi to record and process video while another runs a sound server and delivers rewards. Autopilot expands with the complexity of your task, simultaneously eliminating limitations on quantity of hardware peripherals while ensuring latency is minimal. More interestingly, distributing tasks allows the arbitrary construction of what we call "behavioral topologies," which we describe in section 3.4.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

2.3 *Reproducibility*

Standardized task descriptions

The implementation and fine details of a behavioral experiment matter. Seemingly trivial details like milliseconds of delay between trial phases and microliters of reward volume can be the difference between a successful and unsuccessful task (Figure 2.4). *Reporting* those details can thus be the difference between a reproducible and unreproducible result. Researchers also often use "auxiliary" logic in tasks—such as methods for correcting response bias—that are never completely neutral to the interpretation of results. These too can be easily omitted due to brevity or memory in plain-English descriptions of a task, such as those found in Methods sections. Even if all details of an experiment were faithfully reported, the balkanization of behavioral software into systems peculiar to each lab (or even to individuals within a lab) makes actually performing a replication of a behavior result expensive and technically challenging. Widespread use of experimental tools that are not explicitly designed to preserve every detail of their operation presents a formidable barrier to rigorous and reproducible science[37].

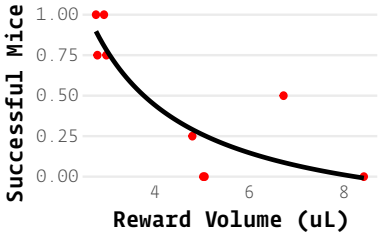


Figure 2.4: "Minor" details have major effects. Proportion of mice (each point, n=4) that were successful learning the first stage of the speech task described in [31] across 10 behavior boxes with variable reward sizes. A $2\mu L$ difference in reward size had a surprisingly large effect on success rate.

Autopilot splits experiments into a) the **code** that runs the experiment, which is intended to be standardized and shared across implementations, and b) the **parameters** (Figure 2.5) that define your particular experiment. For example, two-alternative forced choice tasks have a shared structure regardless of the stimulus modality, but only your task plays pitch-shifted national anthems. Critically, this division of labor enables the possibility of developing a shared library of tasks as described in section 2.2

```
{
"step_name"    : "tone_discrim",
"task_type"    : "2AFC",
"bias_mode"    : 0,
"punish_sound" : false,
"stim" : {
  "sounds" : {
    "L": {
      "duration"  : 100,
      "frequency" : 10000,
      "type"      : "tone",
      "amplitude" : 0.01},
    "R": {"...":"..."}}},
"reward": {
  "type"     : "volume",
  "volume"   : 20},
"graduation" : {
    "type"      : "accuracy",
    "threshold" : 0.75,
    "window"    : 400},
}
```

Figure 2.5: Task parameters are stored as portable JSON, formatting has been abbreviated for clarity.

The practice of reporting exactly the parameter description used by the software to run the experiment removes any chance for incompleteness in reporting. Because all task parameters are included in the produced data files, tasks are fully portable and can be reimplemented exactly by anyone that has comparable hardware to yours.

Self-Documenting Data

A major goal of the open science movement is to normalize publishing well-documented and clearly-formatted data alongside every paper. Typically, data are acquired and stored in formats that are lab-idiosyncratic or ad-hoc. Making data publishable then requires a laborious cleaning process. In the worst-case scenario, this cleaning process unearths some critically missing information about the experiment, requiring awkward caveats in the Methods section. Moreover, without careful version control, any changes made to the task code or parameters can be lost, making it difficult to compare last week's data

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

to last month's.

The best way to make data publishable is to avoid cleaning data altogether and *design good data hygiene practices into the data acquisition process.* Autopilot automatically stores all the information required to fully reconstruct an experiment, including any changes in task parameters or code version that happen throughout training as the task is refined.

Autopilot data is stored in HDF5 files, a hierarchical, high-performance file format. HDF5 files support metadata throughout the file hierarchy, allowing annotations to natively accompany data. Because HDF5 files can store nearly all commonly used data types, data from all collection modalities—trialwise behavioral data, continuous electrophysiological data, imaging data, etc.—can be stored together from the time of its acquisition. Data is always stored with the conditions of its collection, and is ready to analyze and publish immediately (Figure 2.6). No Autopilot-specific scripts are needed to import data into your analysis tool of choice—anything that can read HDF5 files can read Autopilot data.

In future versions we will implement the Neurodata Without Borders standard[29], further enabling Autopilot data to be immediately incorporated into existing processing pipelines (see section 5).

```
root
├── current_task
├── data
│   └── task_name
│       ├── S00_free_water
│       │   └── trial_data
│       └── S01_tone_discrim
│           └── trial_data
├── history
│   ├── git_hashes
│   ├── parameter_history
│   ├── past_protocols
│   │   └── old_task
│   └── weights
└── info
    ├── animal_id
    ├── birth_date
    ├── genotype
    └── etc_additional_info
```

Figure 2.6: Example data structure. All information necessary to reconstruct an experiment is automatically stored in a human-readable HDF5 file.

Expense

Autopilot is an order of magnitude less expensive than comparable behavioral systems (Table 2.3). We think the expense of a system is important for two reasons: scientific equity and statistical power.

The distribution of scientific funding is highly skewed, with a large proportion of research funding concentrated in relatively few labs[18]. Lower research costs benefit all scientists, but lower instrumentation costs directly increase the accessibility of state-of-the-art experiments to labs with less funding. Since well-funded labs also tend to be concentrated at a few (well-funded) institutions, lower research costs also broaden the base of scientists outside traditional research institutions that can stay at the cutting edge[2, 9, 26].

Neuroscience also stands to benefit from the lessons learned from the replication crisis in Psychology[32]. In neuroscience, underpowered experiments are the rule, rather than the exception[5]. Statistical power in neuroscience is arguably even worse than it appears, because large numbers of observations (eg. neural recordings) from a small number of animals are typically pooled, ignoring the nested structure of observations collected within individual animals. Increasing the number of cells recorded from a small number of animals dramatically increases the likelihood of Type I errors (Figure 2.7)—indeed, for val-

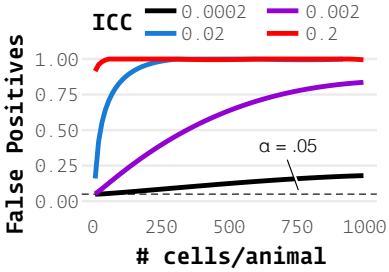


Figure 2.7: When comparing a value across groups, eg. a genetic knockout vs. wildtype, even a modest intra-animal (or, more generally, intra-cluster) correlation (ICC) causes the false positive rate to be far above the nominal $\alpha = 0.05$. Shown are false positive rates for simulated data with various numbers of "cells" recorded for comparisons between two groups of 5 animals each with a real effect size of 0. We note that 741 simultaneously recorded cells were reported in [17] and a mean ICC of 0.19 across 18 neuroscientific datasets was reported in [1]

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

ues of within-animal correlation typical of neuroscientific data, high numbers of observations make Type I errors more likely than not[1]. For this reason, perhaps paradoxically, recent technical advances in multiphoton imaging and silicon-probe recordings will actually make statistical rigor in neuroscience *worse* if we don't use analyses that account for the multilevel structure of the data and correspondingly record from the increased number of animals that they require.

Although the expense of multi-photon imaging and high-density electrophysiology will always impose an experimental bottleneck, behavioral training time is often the greater determinant of study sample size. Typical behavioral experiments require daily training sessions often carried out over weeks and months, while far fewer imaging or electrophysiology sessions are carried out per animal. Training large cohorts of animals in parallel is thus the necessary basis of a well-powered imaging or electrophysiology experiment.

	Autopilot	pyControl	Bpod
Behavior CPU	$35	$284	$745
Nosepoke (3x)	$216	$579	$735
Total for One	**$251**	**$920**	**$1480**
Five Systems	$1255	$4600	$7400
Host CPU(s)	$1000	$5000	$5000
Total for Five	**$2255**	**$9600**	**$12400**
Total for Ten	**$3510**	**$19200**	**$24800**

Table 2.3: Cost for Basic 2AFC System "Nosepoke" includes a solenoid valve, IR sensor, water tube, LED, housing, and any necessary driver PCBs.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

3
Program Structure

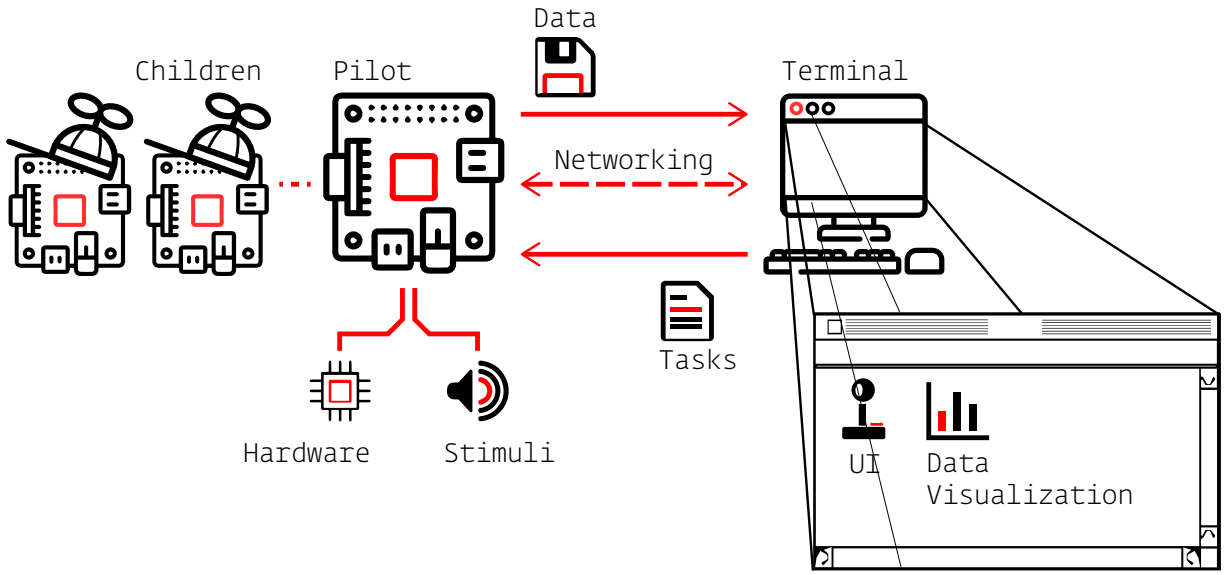


Figure 3.1: Overview of major Autopilot components

Autopilot consists of software and hardware modules that are configured to create a behavioral topology. Independent agents linked by flexible networking objects fill different roles within a topology, such as hosting the user interface, controlling hardware, or delivering stimuli. This infrastructure is ultimately organized to perform a behavioral task.

3.1 Tasks

Behavioral experiments in Autopilot are centered around **tasks**. Tasks are Python classes that describe the parameters, coordinate the hardware, and perform the logic of the experiment. Tasks may consist of one or multiple **stages**, completion of which constitutes a **trial** (Figure 3.2). Stages are analogous to states in the finite state machine formalism.

Multiple tasks are combined to make **protocols**, in which animals move between tasks according to "graduation" criteria like accuracy or number of trials. Training an animal to perform a task typically requires some period of shaping where they are familiarized to the apparatus and the structure of the task. For example, to teach animals about the availability of water from "nosepoke" sensors, we typically begin with

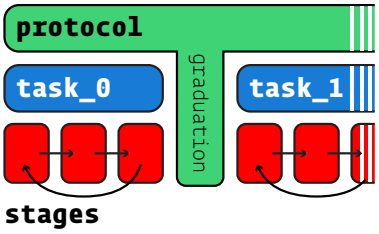


Figure 3.2: Protocols consist of one or multiple tasks, tasks consist of one or multiple stages. Completion of all of a task's stages constitutes a trial, and meeting some graduation criterion like accuracy progresses a subject between tasks.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

a "free water" task that simply gives them water for poking their nose in them. Having a structured protocol system prevents shaping from relying on intuition or ad hoc criteria.

Task Components

The following is a basic two-alternative choice (2AFC) task—a sound is played and an animal is rewarded for poking its nose in a designated target nosepoke. While simple, it is included here in full to show how one can program a task, including an explicit data and plotting structure, in roughly 60 lines of generously spaced Python.

Every task begins by describing four elements:

1) the task's parameters, 2) the data that will be collected, 3) how to plot the data, and 4) the hardware that is needed to run the task.

task - parameters

```
class Nafc(Task):
    PARAMS = {}
    PARAMS['stim']   = {'tag'  : 'Sound Stimuli',
                        'type' : 'sounds'}
    PARAMS['reward'] = {'tag'  : 'Reward Duration (ms)',
                        'type' : 'int'}

    class TrialData(tables.IsDescription):
        target = tables.StringCol(1)
        correct = tables.BoolCol()

    PLOT = {}
    PLOT['data']  =  {'target'  : 'point',
                      'correct' : 'rollmean'},
    # n trials to roll window over
    PLOT['params'] = {'roll_window' : 50}

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

1) A `PARAMS` dictionary defines what parameters are needed to run the task.

A human readable tag and a data type describe each parameter.

2) A (PyTables[14]) `Data` descriptor defines 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.

The specific implementation of the hardware (eg. where it is connected, how to interact with it) is independent of the task. The task just knows about a PORT named C that is a Solenoid.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Created tasks receive some common methods, like input/trigger handling and networking, from an inherited metaclass. Python inheritance can also be used to make small alterations to existing tasks[1] rather than rewriting the whole thing.

[1] An example of subclassing a generic 'Task' class is included in Autopilot's user guide

Stage Methods

The logic of tasks is described in one or a series of methods (stages). The order of stages can be cyclical, as in this example, or can have arbitrary logic governing the transition between stages.

task - methods

```
    def __init__(self, stim, reward=10):
        self.stim_mgr = Stim_Manager(stim)
        self.reward   = Reward_Manager(reward)

        stage_list  = [self.discrim, self.reinforcement]
        self.stages = itertools.cycle(stage_list)

        self.init_hardware()
        self.stages.next()()

    def discrim(self):
        target, wrong, stim = self.stim_mgr.next()
        self.target = target

        self.triggers[target] = [
            self.hardware['PORTS']['C'].open,
            self.stages.next()]
        self.triggers[wrong] = self.stages.next()

        self.node.send('DATA', {'target':target})

        stim.play()

    def reinforcement(self, response):
        if response == self.target:
            self.node.send('DATA', {'correct':True})
        else:
            self.node.send('DATA', {'correct':False})

        self.stages.next()()
```

In Python, **def** defines new methods. The `__init__` method is called when a new object is initialized

Managers control stimulus and reward delivery, so users can, for example, continually synthesize new stimuli or implement adaptive rewards

Stages are combined into an object that (in this case) continually cycles through them when its `next()` method is called.

This starts the task by retrieving the first stage and then calling it.

The stimulus manager returns which port will be the target and the sound to be played.

A sequence of triggers is set: if the target port is poked, a reward will be delivered and the next stage will be called.

The task has a networking object that asynchronously streams data back to the user-facing terminal

In this example, the response port is passed from the trigger handling function. If it matches the stored target variable, the animal answered correctly.

Finally, the task is repeated by calling the next stage.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Autopilot is not prescriptive about how tasks are written. The same task could have two separate methods for correct and incorrect answers rather than a single reinforcement method, or only a single stage that blocks the program while it waits for a response.

Publishing data from this task requires no additional effort: a hash that uniquely identifies the code version (as well as any local changes) is automatically stored at the time of collection, as is the parameter dictionary (Figure 3.3). If this task was incorporated into the central task library, anyone using Autopilot would be able to exactly replicate the experiment from the published data.

```
{
"step_name": "Simple 2AFC",
"stim" : {
  "sounds" : {
    "L": {
      "type"      : "tone",
      "frequency" : 4000},
    "R": {
      "type"      : "tone",
      "frequency" : 8000}
  }
},
"reward": 10
}
```

Figure 3.3: Example parameters for the above task

The limitations of finite state machines

The 2AFC task described above could be easily implemented in a finite-state machine. However, the difficulty of programming a finite-state machine is subject to combinatoric explosion with more complex tasks. Specifically, finite-state machines can't handle any task that requires any notion of "state history."

As an example, consider a maze-based task. In this task, the animal has to learn a particular route through a maze—it is not enough to reach the endpoint, but the animal has to follow a specific path to reach it (Figure 3.4). The arena is equipped with an actimeter that detects when the animal enters each area.

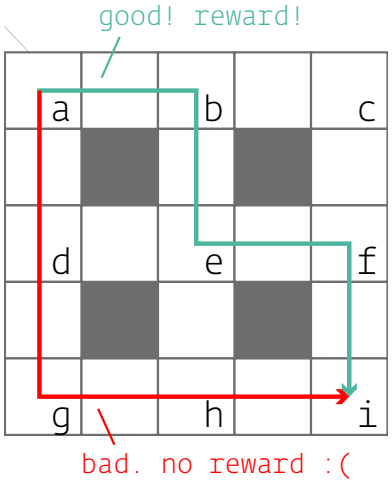


Figure 3.4: The subject must reach point i but only via the correct (green) path.

In Autopilot, we would define a hardware object that logs positions from the actimeter with a `store_position()` method. If the animal has entered the target position ("i" in this example), a `task_trigger()` that advances the task stage is called. The following code is incomplete, but illustrates the principle.

maze - hardware

```
1  class Actimeter(Hardware):
2      def __init__(self):
3          # ... some code to access the hardware ...
4          self.positions = []
5          self.target_position = "i"
6
7      def store_position(self, position):
8          self.positions.append(position)
9
10         if position == self.target_position:
11             self.finished_cb(self.positions)
12             self.positions = []
```

See line 18 below

The task follows, with parameters and network methods for sending data omitted for clarity.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

maze - task

```
class Maze(Task):
    def __init__(self):
        self.target_path = ['a', 'b', 'e', 'f', 'i']

        self.actimeter = Actimeter()
        self.actimeter.finished_cb = self.finished

    def finished(self, positions):
        if positions == self.target_path:
            self.reward()
```

The actimeter is given a reference to the Maze task's `finished()` method, which it calls when the target position is reached

The sequence of `positions` is compared to the `target_path` with `==`. If they match, the subject is rewarded!

How would such a task be programmed in a finite-state machine formalism? Since the path matters, each "state" needs to consist of the current position and all the positions before it. But, since the animal can double back and have arbitrarily many state transitions before reaching the target corner, this task is impossible to represent with a finite-state machine, as a full representation would necessitate infinitely many states (this is one example of the *pumping lemma*, see [19]).

Even if we dramatically simplify the task by 1) assuming the animal never turns back and visits a space twice, and 2) only considering paths that are less than or equal to the length of the correct path, the finite state machine would be as complex as figure 3.5.

While finite-state machines are relatively easy to implement and work well for simple tasks, they quickly become an impediment to even moderately complex tasks. Even for 2AFC tasks, many desirable features are difficult to implement with a finite state machine, such as: (1) graduation to a more difficult task depending on performance history, (2) adjusting reward volume based on learning rate, (3) selecting or synthesizing upcoming stimuli based on patterns of errors[3], etc.

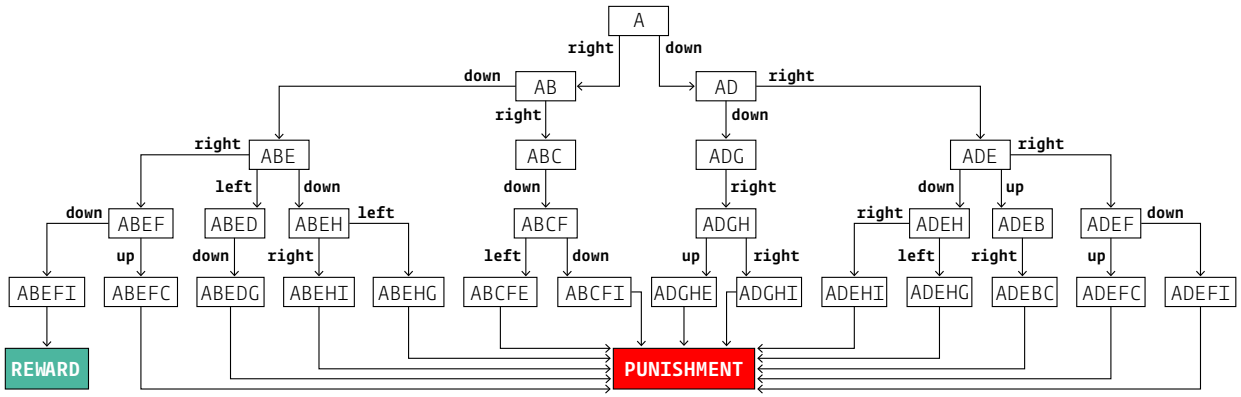


Figure 3.5: State transition tree for a simplified maze task.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

3.2 *Hardware*

The Raspberry Pi can interface with nearly all common hardware, and has an extensive collection of guides, tutorials, and an active forum to support users implementing new hardware. There is also an enormous amount of existing hardware for the Raspberry Pi, including sound cards, motor controllers, sensor arrays, ADC/DACs, and touch-screen displays, largely eliminating the need for expensive proprietary hardware (Table 3.1).

Autopilot uses pigpio to interact with its GPIO pins, giving Autopilot 5μs measurement precision and enabling protocols that require high precision (such as Serial, PWM, and I2C) for nearly all of the pins. Hardware devices in Autopilot are independent Python objects, so their implementation and logic is flexible across installations and tasks. Hardware logic is also reusable, so it doesn't need to be reimplemented for every task, and is intended to be built into a library of hardware objects analogously to tasks.

All hardware objects can be given callback functions to trigger task events, and can be given their own networking object to directly send data and receive configuration input. Time-consuming or continuous operations are run in separate threads and don't block task operation. This makes complex hardware logic easy to implement—for example, if one were using flashing LEDs as an aversive stimulus, the flashes could be delivered with a single method call while the next stage in the task is being computed and other hardware input is still being taken.

Though we expect most users will want to make their own or use existing hardware, we have designed a set of 3D-printable components (Figure 3.6), and include them along with assembly instructions and parts lists on Autopilot's website.

Table 3.1: Cost of common peripherals. The native hardware of the Raspberry Pi and low-level hardware control of Autopilot make most custom-built peripherals unnecessary. While Bpod requires an additional module to decode rotary encoder signals, for example, Autopilot can directly decode them via its GPIO pins with minimal effort by using existing open-source libraries. Inexpensive off-the-shelf hardware is also available to supplement the Pi's native hardware.

Device	Raspi	Bpod
ADC/DAC	$30	$475/$475
I2C	$0	$165
Ethernet	$0	$235
Rotary Encoder	$0	$135

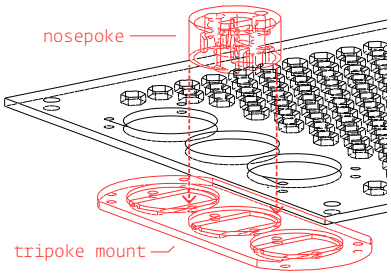


Figure 3.6: We have designed a basic set of easily-assembled hardware available on Autopilot's website.

Table 3.2: Specifications of reviewed behavior hardware

	Raspberry Pi 4B	Bpod (Teensy 3.6)	pyControl (pyboard)
CPU Clock	1.5GHz	180MHz	168MHz
CPU Cores	4	1	1
CPU Architecture	ARMv8-A, 64-bit	ARMv7E-M 32-bit	ARMv7E-M 32-bit
RAM Size	1, 2, or 4GB	256KB	192KB
Storage	MicroSD (any size)	1024KB	1024KB
GPU	Broadcom VideoCore VI	N/A	N/A
GPIO Pins	40	58	29
USB Ports	2x USB 2.0, 2x USB 3.0	2x USB 2.0	1x USB 2.0
Ethernet	1Gbps	100Mbps	N/A
WiFi	2.4/5 GHz 802.11 b/g/n/ac	N/A	N/A
Camera Input	15-pin Serial Interface	N/A	N/A
Bluetooth	✓	N/A	N/A

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

System Adaptability

Rather than attempting to enforce a uniform hardware ecosystem, Autopilot adapts to the radically divergent hardware of different researchers by keeping hardware logic independent from the way it is set up in a particular system. A systemwide `prefs.json` file contains all durable configuration information for your setup (Figure 3.7). Hardware is then accessed by its type and name, so if a task needs an LED named "C" (for "Center"), it would connect to the pins defined in `prefs['PINS']['LEDS']['C']` no matter how the LED was connected or configured in the system. This layer of abstraction allows the task classes to be general enough to maintain a shared task library while also allowing researchers to retain total control over their system.

```
{
"AGENT": "pilot",
"AUDIOSERVER": "jack",
"DATADIR": "/some/data/dir",
"NAME": "example_pilot",
"PINS": {
    "LEDS": {
        "C": [22,18,16],
        "L": [11,13,15],
        "R": [19,21,23]}
},
"MSGPORT": 5565,
"TERMINALIP": "192.168.0.100"
}
```

Figure 3.7: The prefs.json file stores durable system configuration options.

Autopilot also allows the hardware in your particular setup to be used for multiple tasks that may have differing hardware demands. We think a common use-case will be a series of mostly-static behavioral boxes that can be reconfigured without being wholly rebuilt. The hardware schematics we release are modular, so that one could, for example, change a behavior box from a freely-moving two-alternative forced choice task to a head-fixed version by replacing a panel and installing a running wheel. We are working to implement support for multiple hardware configurations that can be kept in the `prefs.json` file and automatically swapped depending on the task being run.

3.3 *Stimuli*

A hardware object would control a speaker, whereas stimulus objects are the individual sounds that the speaker would play. Like tasks and hardware, Autopilot makes stimulus generation portable between users, and is released with a family of common sounds like tones, noises, and sounds from files. The logic of sound presentation is contained in an inherited metaclass, so to program a new stimulus a user only needs to describe how to generate it from its parameters (Figure 3.8). Sound stimuli are better developed than visual stimuli in the current version of Autopilot, but we present a proof-of-concept visual experiment (Section 4.3) using psychopy[27].

```
An Autopilot Tone
my_tone = sounds.Tone(
    frequency = 500,
    duration  = 200)
my_tone.play()
```

```
A Bpod Tone
tone = GenerateSineWave(...
samplingrate, freq, dur);

% load to audio server
server = BPodAudioPlayer;
server.loadSound(1,tone);

% buffer sound after poke
sma = AddState(sma, ...,
'OutputActions',
{'AudioPlayer1', '*'});
% play sound by number
sma = AddState(sma, ...,
'OutputActions',
{'AudioPlayer1', 1});
```

Figure 3.8: Autopilot stimuli are parametrically defined and inherit all the playback logic that makes them easy to integrate in tasks

Autopilot controls the realtime audio server jack from an independent Python process that dumps samples directly into jack's buffer (Figure 3.9), giving it the lowest trigger-to-playback latency of any of the systems we have tested or found benchmarks for (Section 4.1). Sounds can be buffered in system memory or synthesized on demand, and the only limit on the number of stimuli that can be simultaneously buffered is the Pi's generous 4GB of memory. Because the realtime server is independent from the logic of sound synthesis and storage,

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

stimuli can be controlled independently from different threads without interrupting audio or dropping frames.

We use the Hifiberry Amp 2, a combined sound card and amplifier, which is capable of 192kHz/24Bit audio playback. Jack can output to any sound hardware, however, including the builtin audio of the Raspberry Pi if fidelity isn't important. There are no external video cards for the Raspberry Pi, but its embedded video card is capable of presenting video and visual stimuli (Section 4.3) especially if the other computationally demanding parts of the task are distributed to other Raspberry Pis (Section 3.4).

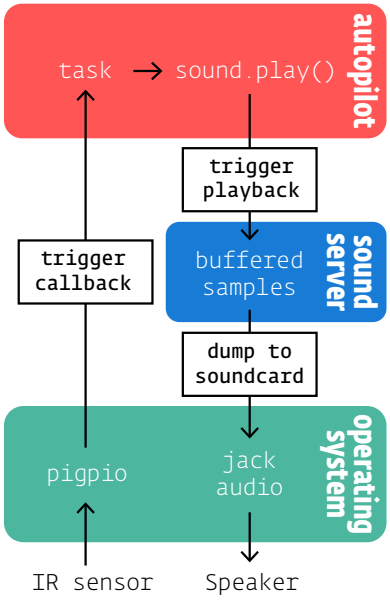


Figure 3.9: Our sound server keeps audio samples buffered until a `.play()` method is called, and then dumps them directly into the jack audio daemon.

Stimulus and Reward Managers

In many tasks, the structure of the stimulus presentation is as important as the structure of the task. Stimulus structure can become complicated quickly—in addition to whatever order is necessitated by the task design, it is common to also include shaping routines like bias correction in the presentation logic. Different types of stimuli also require different degrees of coordination: unitary stimuli that are presented once per trial can be handled independently without fear of them overlapping or interrupting one another, but continuous stimuli that change in response to task performance need to be mutually coordinated.

We separate stimulus presentation logic from task structure by using stimulus managers. Stimulus managers have different 'base' presentation types—eg. random presentation, blocked presentation, etc.—and a set of configurable transformations like bias correction that can be chained together. The stimulus manager can yield prebuffered stimulus objects, synthesize new stimuli according to some task-related rule, and manage a continuous stimulus stream.

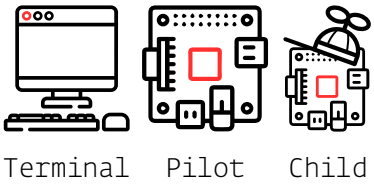
Reward managers behave similarly[2]. Reward managers can implement different calibration schemes—eg. for gravity-fed water delivery, reward can be configured to be delivered for a constant time, constant volume, or use the animal's mass and performance to adaptively deliver a total volume over a period of time.

[2] Reward managers are not yet implemented as independent classes in the current version (0.2) of Autopilot, but are a planned feature of Autopilot v0.3. Different modes of reward delivery are currently implemented by the `Solenoid` class.

3.4 *Agents - Terminal, Pilot, and Child*

All of the above components—tasks, hardware, and stimuli—are organized into a single system as an "agent," the central executable component of Autopilot which a) manages the core operations of the system and b) defines how it interacts with the rest of the agents it is connected to. Specifically, agents are built around an action vocabulary that maps different types of messages to callback methods.

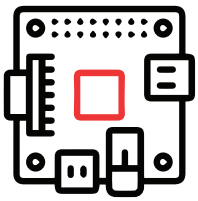
Three Agents:

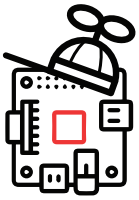


bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Currently, we have implemented three Agent types:

- **Terminal** - The user-facing control agent.
- **Pilot** - A Raspberry Pi that runs tasks, coordinates hardware, and optionally coordinates a set of child Pis.
- **Child** - Subordinate Pis to a pilot that carry out different parts of a task





Terminal agents serve as a root node (see Section 3.5) in an Autopilot swarm. The terminal is the only agent with a GUI, which is used to control its connected pilots and visualize incoming task data. The terminal also manages data and keeps a registry of all active experimental subjects. The terminal is intended to make the day-to-day use of an Autopilot swarm manageable, even for those without programming experience. The terminal GUI is described further in Section 3.6.

Pilot agents are the workhorses of Autopilot—the agents that run the experiments. Pilots are intended to operate as always-on, continuously running system services. Pilots make a network connection to a terminal and wait for further instructions. They maintain the system-level software used for interfacing with the hardware connected to the Raspberry Pi, receive and execute tasks, and continually return data to the terminal for storage.

Each pilot is capable of coordinating one or many **child** agents. The pilot maintains a network connection to its children, and if a task specifies that some of its functionality is to be split between Raspberry Pis, the pilot notifies its children and sends them a specialized subtask description. The pilot serves as the only point of contact between its children and the terminal, so the terminal only needs to keep track of its pilots, and doesn't need separate methods for communicating with all their children, their hardware, etc.

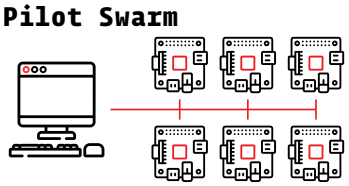
Behavioral topologies

We think one of the most transformative features of Autopilot's distributed structure is the control that users have over what we call "behavioral topology." The logic of hardware and task operation within an agent, the distribution of labor between agents performing a task, and the pattern of connectivity and command within a swarm of agents constitute a topology.

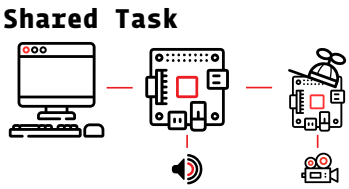
Below we illustrate this idea with a few examples:

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

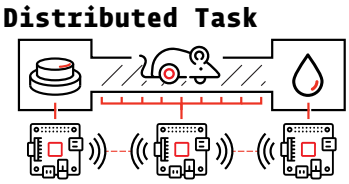
- **Pilot Swarm** - The first and most obvious topological departure from traditional behavioral instrumentation is the use of a single computer to independently coordinate tasks in parallel. Our primary installation of Autopilot is a cluster of 10 behavior boxes that can independently run tasks dispatched from a central terminal which manages data and visualization. This topology highlights the expandability of an Autopilot system: adding new pilots is inexpensive, and the single central terminal makes controlling experiments and managing data simple.



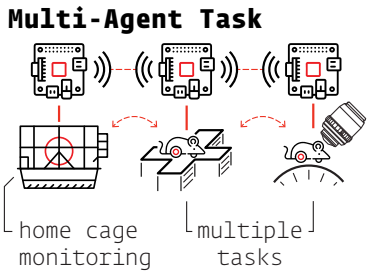
- **Shared Task** - Tasks can be shared across pilots and their (potentially multiple) children to handle tasks with computationally intensive operations. For example, in an open-field navigation task, one pilot can deliver position-dependent sounds while one of its children records and analyzes video of the arena to track the animal's position. The terminal only needs to be configured to connect to the parent pilot, but since networking is handled in an independent process the raw video data can pass through the parent from the child such that sound delivery remains responsive.



- **Distributed Task** - Many pilots with overlapping responsibilities can cooperate to perform distributed tasks. We anticipate this will be useful when the experimental arenas can't be fully contained (such as natural environments), or when experiments require simultaneous input and output from multiple subjects. Distributed tasks can take advantage of the Pi's wireless communication, enabling, for example, experiments that require many networked cameras to observe an area, or experiments that use the Pis themselves as an interface in a multisubject augmented reality experiment.



- **Multi-Agent Task** - Neuroscientific research often consists of multiple mutually interdependent experiments, each with radically different instrumentation. Autopilot provides a framework to unify these experiments by allowing users to rewrite core functionality of the program while maintaining integration between its components. For example, a neuroethologist could build a new "Observer" agent that continually monitors an animal's natural behavior in its home cage to calibrate a parameter in a task run by a pilot. If they wanted to manipulate the behavior, they could build a "Compute" agent that processes Calcium imaging data taken while the animal performs the task to generate and administer patterns of optogenetic stimulation. We think that unifying diverse experimental data streams and hardware into a single framework is the best way to perform experiments that measure natural behavior and its hierarchical organization across multiple timescales in order to understand the naturally behaving brain[10].



bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

3.5 *Networking*

Agents use two types of object to communicate with one another: core **station** objects and peripheral **node** objects (Figure 3.10). Each agent creates one station in a separate process that handles all communication *between* agents. Stations are capable of forwarding data and maintaining agent state so the agent process is not unnecessarily interrupted. Nodes are created by individual modules run within an agent—eg. tasks, plots, hardware—that allow them to send and receive messages within an agent or between agents through the station object. Messages are TCP packets[3], so there is no distinction between sending messages within a computer, a local network, or over the internet.

Both types of networking objects are tailored to their hosts by a set of callback functions—**listens**—that define how to handle each type of message. Messages have a uniform key-value structure, where the key indicates the listen used to process the message and the value is the message payload. This system makes adding new network-enabled components trivial:

A new networked LED

```
class LED_RGB(Hardware):
    def __init__(self):
        # call self.color for a 'COLOR' message
        self.listens = {'COLOR': self.color}
        self.node = networking.Node(
            id      = 'BEST_LED',
            listens = self.listens)

    def color(msg):
        self.set_color(msg.value)

# elsewhere in the code, we change the color to red!
node.send(to='BEST_LED', key='COLOR', value=[255,0,0])
```

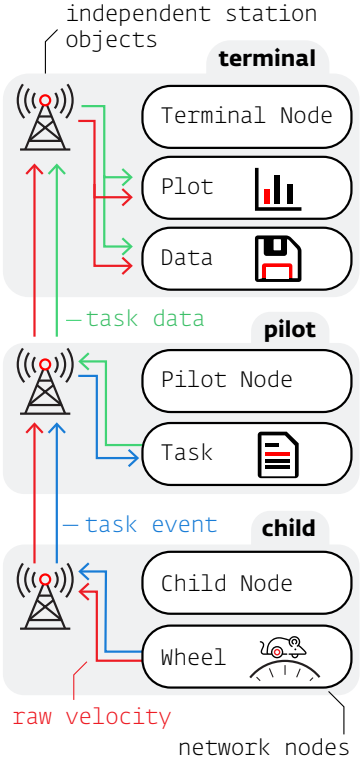


Figure 3.10: Autopilot segregates data streams efficiently—eg. raw velocity (red) can be plotted and saved by the terminal while only the task-relevant events (blue) are sent to the pilot. The pilot then sends trial-summarized data to the terminal (green).

[3] Autopilot uses ZeroMQ[15] and tornado to send and process messages

Network connectivity is treelike (Figure 3.11)—each independent networking object can have many children but at most one parent. This structure makes an implicit assumption about the anisotropy of information flow: 'higher' nodes don't need to send messages to the 'lowest' nodes, and the 'lowest' nodes send all their messages to one or a few 'higher' nodes. It enforces simplified delegation of responsibilities in both directions: a terminal shouldn't need to know about every hardware object connected to all of its connected pilots, it just sends messages to the pilots, who handle it from there. A far-downstream node shouldn't need to know exactly how to send its data back to the terminal, so it pushes it upstream until it reaches a node that does.

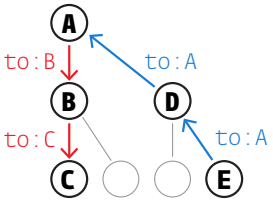


Figure 3.11: Treelike network structure—downstream messages are addressed by successive nodes, but upstream messages can always be pushed until the target is found.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

3.6 GUI & Plots

The terminal's GUI controls day-to-day system operation[4]. It is intended to be a nontechnical frontend that can be used by those without programming experience.

For each pilot, the terminal creates a control panel that manages subjects, task operation, and plots incoming data. Subjects can be managed through the GUI, including creation, protocol assignment, and metadata editing. Protocols can also be created from within the GUI. The PARAMS dictionary from a task is used to programmatically generate a series of fields that the user can fill to describe their particular version of the task. The standardized description of tasks not only allows them to be reused between researchers, but also take advantage of the rest of the infrastructure of Autopilot.

The GUI also has a set of basic maintenance and informational routines in its menus, like calibrating water ports or viewing a history of subject weights. The simple callback design and network infrastructure makes adding new GUI functionality straightforward.

[4] Autopilot uses PySide, a wrapper around Qt, to build its GUI.

Plotting

Realtime data visualization is critical for monitoring training progress and ensuring that the task is working correctly, but each task has different requirements for visualization. A task that has a subject continuously running on a ball requires a continuous readout of running velocity, whereas a trial-based task only needs to show correct/incorrect responses as they happen. Autopilot solves this problem by assigning the data returned by the task to graphical primitives like points, lines, or shaded areas as specified in a task's PLOT dictionary (taking inspiration from Wilkinson's grammar of graphics[39]).

```
Trial Plot
{"data": {
    "target"   : "point",
    "response" : "segment",
    "correct"  : "rollmean"
},
"roll_window" : 50}
```

```
Continuous Plot
{"data": {
    "target"   : "point",
    "response" : "segment",
    "velocity" : "shaded"
},
"continuous": true}
```

Figure 3.12: PLOT parameters for Figure 3.13. In both, "target" and "response" data are mapped to "point" and "segment" graphical primitives, but timestamps rather than trial numbers are used for the x-axis in the "continuous" plot (Figure 3.13, bottom). Additional parameters can be specified, eg. the trial plot (Figure 3.13, top) computes rolling accuracy over the past 50 trials

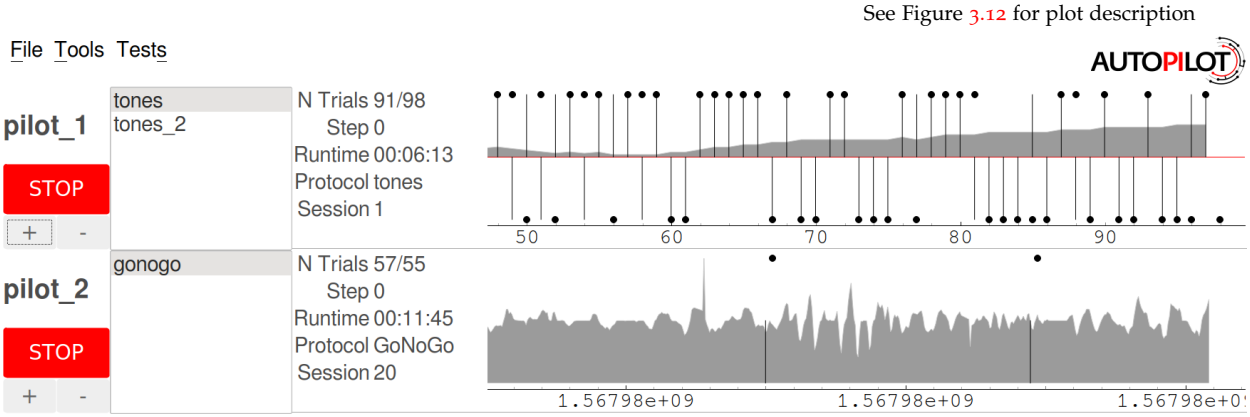


Figure 3.13: Screenshot from a terminal GUI running two different tasks with different plots concurrently. pilot_1 runs 2 subjects: (tones and tones_2). See Figure 3.12 for plot description

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

4
Tests

We have been testing and refining Autopilot since we built our swarm of 10 training boxes 10 months ago. In that time 115 mice[1] have performed over 1.9 million trials on auditory two-alternative forced choice tasks. Our terminal has sent and received more than 42 million messages. While Autopilot is (by definition) immature at release, it is by no means untested.

[1] All procedures were performed in accordance with National Institutes of Health guidelines, as approved by the University of Oregon Institutional Animal Care and Use Committee.

4.1 Latency

Neurons compute at millisecond timescales, so any task that links neural computation to behavior needs to have near-millisecond latency. We measured Autopilot's end-to-end, hardware input to hardware output latency by measuring the delay between a poke in a nosepoke sensor and the onset of a 10kHz pure tone (Table 4.1).

We also measured the latency of a Bpod state machine configured according to the provided instructions and running an example task from their repository. Sound playback was triggered with a 1ms TTL pulse to the state machine's BNC input port. We note that for the Bpod test we used a more recent soundcard from the same manufacturer and Ubuntu 16.04 (running the `lowlatency` kernel) since the recommended Asus Xonar DX is no longer available for purchase and Ubuntu 14.04 is no longer supported.

Autopilot's jack audio backend was configured with a `192kHz` sampling rate and a total buffer size of `128` samples, and Bpod's Psychtoolbox server was configured with a `192kHz` sampling rate with a `32` sample buffer for theoretical minimum latencies of `0.67` and `0.17ms`, respectively.

For both systems we directly measured the input logic and output sound voltage with an oscilloscope and estimated latency with its measurement cursors.

Table 4.1: Latency Test Materials

Autopilot	Raspberry Pi 4
Soundcard	Hifiberry Amp2
IR Break Sensor	TT Electronics OPB901L55
Speaker	HiVi RT1.3WE
Bpod	State Machine R2
Computer	See Table 4.2
Soundcard	ASUS Xonar Essence STX II
Stimulator	Grass S88
Oscilloscope	Tektronix TDS 2004B

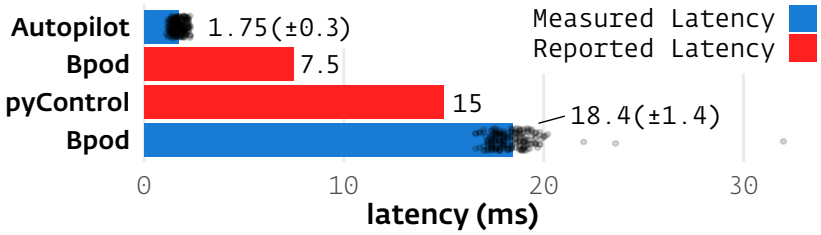


Figure 4.1: For the two systems we measured (blue), mean latency is presented ± standard deviation of all individual measurements (black dots, n=200 for each). Reported latencies (red) of Bpod and pyControl were found online.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Autopilot's `1.75ms` ± `0.3` latency—less than 3x the theoretical minimum—improves upon the measured latency of Bpod and reported latency of pyControl by an order of magnitude (Figure 4.1, `18.4ms` ± `1.4`, `15ms` respectively). This suggests that Autopilot eliminates most perceptible end-to-end latency, which is necessary for tasks that require realtime feedback.

While we did not deeply investigate the reason why Bpod exceeded its theoretical minimum latency by more than 100x, potential sources of latency include a costly serial reading method, or the MATLAB graphics engine being continuously called in the main loop of the program, which are intrinsic to its single-threaded design.

Since Autopilot's event handling infrastructure is shared across tasks and hardware classes, latency for all events should be roughly similar to that of audio playback. One future direction is to improve upon Autopilot's already-low latency by compiling its sound server and event handling methods using Cython.

Table 4.2: Terminal Specs

CPU	AMD FX-4300
CPU Speed	3.8GHz
Memory	8GB
Ethernet	1Gbit/s
Switch	NETGEAR GSS116E

4.2 *Bandwidth*

To support data-intensive tasks like those that require online processing of video or electrophysiological data, the networking modules at the core of Autopilot need high bandwidth and low latency.

We tested network capacity using Autopilot's `Bandwidth_Test` widget. This test requests that a set of selected pilots send messages at a range of selected frequencies and payload sizes back to the terminal. The messages pass through four networking objects en route: the stations and network nodes running the test for both the terminal and pilots (See Figure 3.10). Delay is measured as the duration between the creation of the message at the sender and the processing of the message at the receiver. The Pis and terminal were synchronized on common NTP servers to align timestamps.

First we tested the limits of our terminal's ability to receive messages from the 10 pilots that it controls. Our terminal is a modest desktop (complete with a vintage 2012 CPU, see Table 4.2) with ethernet connections to 10 Raspberry Pi 3b's through a network switch. We first tested the rate at which the Pi 3b's and our terminal could send and process typical (`255 Byte`) messages without a data payload (Figure 4.2, top). A single Pi was capable of sending at a maximum rate of `707 Hz` without exceeding its nominal mean delay of `4.9` (± `0.47`) ms. Adding additional Pis did not cause increased delay until the total sending rate surpassed roughly `2000 Hz`. These are the rate limits of sending and receiving messages, respectively.

As we increased the size of each individual message by including payloads of generated data (Figure 4.2, bottom), the rate of messag-

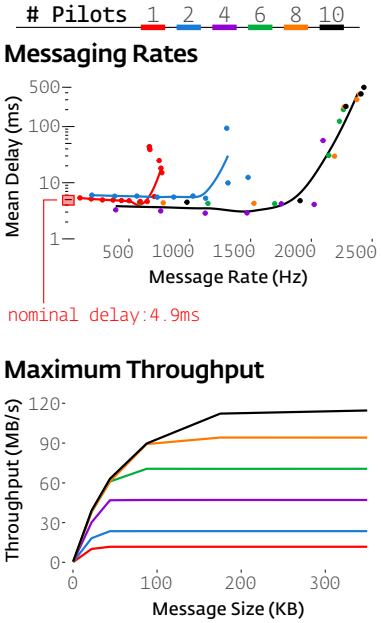


Figure 4.2: Network latency (top) and throughput (bottom) tests. Each point in the latency test represents the mean rate and delay of 5,000 255 Byte messages. Throughput (bottom) was calculated as the product of message rate and message size, and is displayed for a test that requested different numbers of pilots (colors) to send messages of different size to the terminal.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

ing decreased, but the total throughput (`message rate (Hz) * size (Bytes)`) saturated linearly as a multiple of the number of sending Pis. The Raspberry Pi 3b has a shared USB/Ethernet Bus, and thus appears to have a relatively limited `11.8MB/s` throughput.

Fortunately, the Raspberry Pi 4 has an independent gigabit ethernet bus. On a Raspberry Pi 4, Autopilot has a `41MB/s` maximum throughput and a `1,919Hz` maximum messaging rate (Figure 4.3). We observed a slightly higher messaging delay with the Raspberry Pi 4 (`6.9ms` vs. `4.9ms` Raspberry Pi 3B+). We note that the NTP synchronization method we used to measure delays has a margin of error on the order of milliseconds.

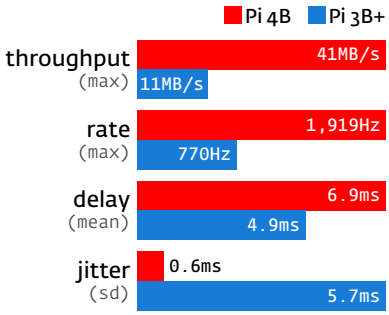


Figure 4.3: The Raspberry Pi 4's gigabit ethernet bus markedly improves network performance.

Autopilot's networking modules are capable of supporting the infrastructure of next-generation behavioral neuroscience experiments. Our humble terminal was capable of receiving the full `114.6MB/s` of 10 Pis without sign of saturation, and a Raspberry Pi 4 is capable of sending data at `41MB/s`. This bandwidth makes Autopilot capable of streaming raw Calcium imaging[2] and electrophysiological data from modern high-density probes[3]. The delay between sending and processing messages over 4 hops in a network (`4.9ms`) is less than the latency with which comparable systems (Figure 4.1) process triggers when connected directly via serial.

[2] 2-Photon: 5.9MB/s
(12 bits * 512x512 resolution * 15Hz)

[3] Neuropixels: 14.4MB/s[17]
(10 bits * 30kHz * 384 channels)

Finally, while Autopilot typically operates in a "TCP-like" protocol—resending messages until they have been confirmed as received—these tests were run with an optional "UDP-like" protocol which does not check for confirmation. Across the approximately 2.5 million messages sent during these tests only `537` were dropped (and only during tests which saturated rate or bandwidth capacity), giving Autopilot a delivery rate of `99.98%` in "UDP" mode. By design, delivery rate is guaranteed to be 100% in "TCP" mode.

4.3 *Distributed Go/No-go Task*

We designed a visual go/no-go task as a proof of concept for distributing task elements across multiple Pis, and also for the presentation of visual stimuli (Figure 4.4). The code for this task is described in greater detail in the user guide.

In this task, a head-fixed subject would[4] be running on a wheel in front of a display with a lick-detecting water port able to deliver reward. Above the port is an LED. Whenever the LED is green, if the subject drops below a threshold velocity for a fixation period, a grating stimulus at a random orientation is presented on the monitor. After a random delay, there is a chance that the grating changes orientation by a random amount. If the subject licks the port in trials when the orientation is changed, or refrains from licking when it is not, the

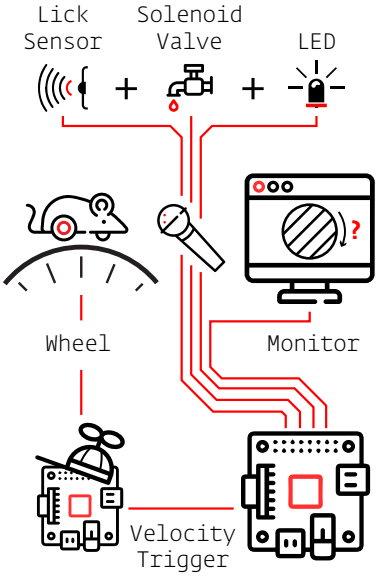


Figure 4.4: Hardware distribution for the distributed go/no-go task

[4] No mice were trained on this task

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

subject is rewarded.

One "parent" pilot controlled the operation of the task, including the coordination of its child[5]. The parent was connected to the LED and solenoid valve for reward delivery, as well as a monitor[6] to display the gratings[7]. The child continuously streamed velocity data (measured with a USB optical mouse against the surface of the wheel) back to the terminal for storage (see also Figure 3.10, which depicts the network topology for this task). The child waited for a message from the parent to initiate measuring velocity, and when a rolling average of recent velocities fell below a given threshold the child sent a TTL trigger back to the parent to start displaying the grating. This split-pilot topology allows us to poll the subject velocity continuously (at `125Hz` in this example) without competing for resources with psychopy's rendering engine.

[5] Both Raspberry Pi 4s

[6] Acer S230HL - (`1920x1080px, 60Hz`)

[7] Visual stimuli were presented with Psychopy (v3.1.5) using the glfw (v1.8.3) backend while Autopilot was run in a dedicated X11 display server.

We measured trigger (TTL pulse from the child) to visual stimulus onset latency using the measurement cursors of our oscilloscope as before. To detect the onset of the visual stimulus, we used a high-speed optical power meter[8] attached to the top-left corner of our display monitor. The stimulus was a drifting Gabor grating drawn to fill half the horizontal and vertical width of the screen (`960 x 540px`), with a spatial frequency of `4cyc/960px` and temporal (drift) frequency of `1Hz`.

[8] Thorlabs PM100D

We observed a bimodal distribution of latencies (Quartiles: `28, 30, 36ms, n=50`, Figure 4.5), presumably because onsets of visual stimuli are quantized to the refresh rate (`60Hz, 16.67ms`) of the monitor. This range of latencies corresponds to the second and third frame after the trigger is sent (2/3 of observations fall in the 2nd frame, 1/3 of observations in the 3rd frame). We observed a median framerate of `36.2 FPS (IQR: 0.7)` across 50 trials (`8863` frames, Figure 4.6).

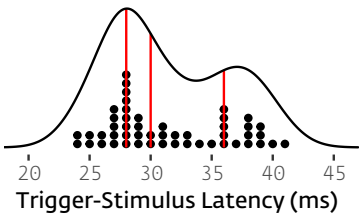


Figure 4.5: Stacked dots are a histogram of individual observations (n=50) underneath the probability density (black line), red lines indicate quartiles.

We further tested the Pi's framerate by using Psychopy's `timeByFrames` test—a script that draws stimuli without any Autopilot components running—to see if the framerate limits were imposed by the hardware of the Raspberry Pi or overhead from Autopilot (Table 4.3). We tested a series of Gabor filters and random dot stimuli (dots travel in random directions with equal velocity, default parameters) at different screen resolutions and stimulus complexities. The Raspberry Pi was capable of moderately high framerates (`>60 FPS`) for smaller, lower resolution stimuli, but struggled (`<30 FPS`) for full HD, fullscreen stimuli.

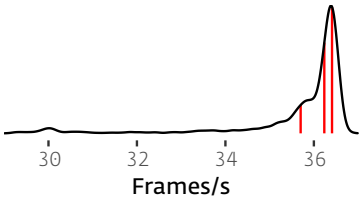


Figure 4.6: Probability density of framerates for 960 x 540px grating rendered at 1080p. Red lines indicate quartiles

Autopilot is appropriate for realtime rendering of simple stimuli, and the proof-of-concept API we built around Psychopy doesn't impose discernible overhead (Mean framerate for a `960 x 540px` grating at `1080p` in Autopilot: `36.2 fps`, vs. `timeByFrames: 35.0 fps`). In the future we will investigate prerendering and caching complex stimuli in order to increase performance. A straightforward option for higher-performance video would be to deploy an Autopilot agent running

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

on a desktop computer with a high-performance GPU, or to use a single-board computer with a GPU like the NVIDIA Jetson ($99).

Stimulus	Resolution	Size / # Dots	Mean FPS	σ FPS
Gabor Filter	1280 x 720	300 x 300px	106.4	5.5
Gabor Filter	1920 x 1080	300 x 300px	75.2	3.5
Gabor Filter	1280 x 720	640 x 360px	53.5	2.2
Gabor Filter	1920 x 1080	960 x 540px	35.0	1.0
Gabor Filter	1280 x 720	720 x 720px	41.5	2.2
Gabor Filter	1920 x 1080	1080 x 1080px	20.1	0.7
Random Dots	1280 x 720	100 dots	98.0	3.8
Random Dots	1920 x 1080	100 dots	67.6	3.0
Random Dots	1280 x 720	1000 dots	20.9	0.25
Random Dots	1920 x 1080	1000 dots	19.5	0.36

Table 4.3: Tests performed over 1000 frames with PsychoPy's `timeByFrames` test.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

5
Limitations and Future Directions

While we believe that Autopilot's order of magnitude increase of performance and decrease in expense, and its qualitative improvements in task design flexibility due to its distributed architecture are already useful contributions to behavioral neuroscience, we do not view Autopilot as "finished." We view Autopilot—like all open-source software—as an evolving project. We are invested in its development, and will be continually working to fix bugs, make its use more elegant, and add new features in collaboration with its users.

We expect that as the codebase matures and other researchers use Autopilot in new, unexpected ways that some fundamental elements of its structure may evolve. We have built version logging into the structure of the system so that changes will not compromise the replicability of experiments (see **Versioning and Containerization** below). While there will inevitably be changes between versions, these will be both transparently documented and announced in release notes in order to alert users and describe how to adapt as needed. Accordingly, potential users should not let the limitations and future directions described below cause them to worry about early adoption or to wait for a stable version—the cost to start using Autopilot is low, and in our experience implementing experiments is already easier and more straightforward than comparable behavior systems.

We see several limitations in the launch version of Autopilot that we will improve on in future versions:

- **Python 3** - We began developing Autopilot while there was still a case to be made for using Python 2. Now, given Python 2's impending end of life in 2020, we will transition Autopilot to Python 3 by the end of 2019. We have already started transitioning with the `Subject` data class and don't see the transition as a great obstacle.
- **Synchronization** - Currently, there is no synchronization engine built into Autopilot. To ensure time-sensitive operations distributed over multiple Raspberry Pis are synchronized (ie. generate near-identical timestamps), we will add the ability for agents to generate and follow a clock signal with pigpio. This synchronization engine will also allow alignment of Autopilot data with external software, such as the proprietary software often used for imaging data acquisition.
- **Integration with Other Software** - We will make Autopilot capable of natively recording electrophysiological data by integrating with Open Ephys[33]. We also are interested in tightly integrating other recent tools like DeepLabCut[23] and MoSeq[40] to make Autopilot a unified platform for complex and naturalistic behavioral experiments.
- **Transformations** - To enable the use of computer vision and other analytical tools within tasks we have begun building a data transformation module. This module will provide a framework to perform high-level data transformations—eg. images from a camera to positions of tracked objects—that convert raw data from hardware objects to processed data useful for designing complex tasks.
- **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. We will be designing a common Agent class schema so that they are easier to design and deploy. We also plan to expand the available

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

agents, specifically by introducing Observer and Compute agents. Observers will be designed for passive observation without supervision from a terminal, eg. for monitoring animals continuously in their home cages. Compute agents will run on high-performance computers in order to facilitate computationally intensive operations like GPU-dependent image analysis, online spike-sorting, etc. A mature agent framework will provide a much more streamlined path to the complex multi-agent experiments alluded to in Section 3.4.

- **Data** - We plan on transitioning our data model to implementing the Neurodata Without Borders[29] standard. Since the Neurodata Without Borders standard is implemented in HDF5 and structurally similar to our data model, this transition should be straightforward. We also plan on adding support for a NoSQL mongoDB database backend to improve reliability, scalability, and performance of data storage and retrieval. Since our data model is standardized, we will ensure all data storage backends are mutually compatible so data stored in a database can be exported to HDF5 files and vice versa. Currently Autopilot only automatically logs changes in task parameters and code version, but in the future we will expand our logging facility to include detailed data on systemwide preferences and connected hardware.
- **Versioning and Containerization** - While Autopilot version and local changes are logged in collected data by default, there is no way to specify that a task should be run using a particular version automatically (ie. the user has to manually check out the specific git commit before running Autopilot). We intend on supporting task parameterizations that specify particular versions of Autopilot. We also will expand Autopilot's version logging system to include the versions of all the other packages in the environment. In our view, the best way to support reproducible software environments is to use a container system like Docker, so we will be building infrastructure to generate containers from task parameterizations.
- **Tasks** - We look forward to collaborating with other researchers to expand the available library of tasks. While the two-alternative forced choice and go/no-go tasks we have implemented are common, we designed Autopilot to be capable of performing *any* behavioral experiment. For example: we have already started a collaboration to build a freely-moving, jumping-based behavior that relies on 16 hardware components and data streams, and have future plans to build hardware and stimulus management extensions for human psychophysical tasks performed in an fMRI.
- **Mesh Networking** - The tree structure of Autopilot's networking was built to enforce simplicity of its messaging protocol, but it limits the ability for data to be shared efficiently between a large number of pilots because communication has to be routed through a hub terminal. We will implement a true mesh network architecture by implementing a distributed hash table, allowing agents to directly communicate with one another without explicit configuration. We also will implement a peer-to-peer data protocol akin to Bittorrent to allow efficient distribution of data across a swarm of agents.
- **Web Interface** - We would like to make a web-compatible UI that allows tasks to be administered and monitored from any computer. A web interface would make continuous experiments much easier to manage—we specifically intend this improvement (along with the Observer agent) to facilitate active sensory enrichment[38, 12] and developmental experiments.
- **Platform Independence** - We have not rigorously tested Autopilot on operating systems other than Raspbian and Ubuntu Linux, though we know the terminal agent and its GUI works on macOS.
- **Unit Tests** - At release, Autopilot has no unit tests. To make the codebase easier to maintain, we aim to reach 100% coverage by the first stable release of the program (v1.0).

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

6 Glossary

Agent	3.4	The executable part of Autopilot. A set of startup routines (eg. opening a GUI or starting an audio server), runtime behavior (eg. opening as a window or running as a background system process), and event handling methods (ie. **listens**) that constitute the role of the particular Autopilot instance in the **swarm**.
Child	3.4	An **agent** that performs some auxiliary, supporting role in a **task**—primarily used for offloading some hardware responsibilities from a **pilot**.
Graduation	3.1	Moving between successive **tasks** in a **protocol** when some criterion is met.
Listen	3.5	A method belonging to the **station** or **node** of a particular **agent** that defines how to process a particular type of message (ie. a message with a particular `key`).
Node	3.5	A networking object that some module (eg. hardware, **tasks**, GUI routines) or method (eg. a **listen**) uses to communicate with other **nodes**. Messages to other **agents** in the swarm are relayed through their **Station**
Pilot	3.4	An **agent** that runs on a Raspberry Pi, the primary experimental agent of Autopilot. Typically runs as a system service, receives **tasks** from a **terminal** and runs them. Can organize a group of **children** if requested by the **task**.
Protocol	3.1	A (`.json`) file that contains a list of **task** parameters and the **graduation** criteria to move between them. The **tasks** in a protocol are also known as its **levels**.
Stage	3.1	**Stages** are methods that implement the logic of a **task**. They can be used analogously to states in a finite-state machine (eg. wait for **trial** initiation, play stimulus, etc.) or asynchronously (whenever x input is received, rotate stimulus by y degrees).
Station	3.5	Each **agent** has a single **station**, a networking object that is run in its own process and is responsible for communication between **agents**. The **station** also routes messages from **children** or other **nodes**.
Swarm		Informally, a group of connected **agents**.
Task	3.1	A formalized description of an experiment: the parameters it takes, the data that it collects, the hardware it needs, and a collection of **stages** that describe what happens during the experiment.
Terminal	3.4	A user-facing **agent** that provides a GUI for operating and maintaining a **swarm**.
Topology	3.4	A particular combination of **agents**, their designated responsibilities, and the networking connections between them invoked by a **task** (eg. task requires one pilot to record video, one to process the video, and one to administer reward) or by usage (eg. 10 pilots are connected to a single terminal and are typically used to run 10 independent tasks, though they could run shared tasks together).
Trial	3.1	If a **task** is structured such that its **stages** form a repeating series, a **trial** is a single completion of that series.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

Bibliography

[1] Emmeke Aarts, Matthijs Verhage, Jesse V. Veenvliet, Conor V. Dolan, and Sophie van der Sluis. A solution to dependency: Using multilevel analysis to accommodate nested data. Nature Neuroscience, 17(4):491–496, April 2014. 2.7, 2.3

[2] Jeremy Ashkenas, Haeyoun Park, and Adam Pearce. Even With Affirmative Action, Blacks and Hispanics Are More Underrepresented at Top Colleges Than 35 Years Ago. The New York Times, August 2017. 2.3

[3] Ji Hyun Bak, Jung Yoon Choi, Athena Akrami, Ilana Witten, and Jonathan W Pillow. Adaptive optimal training of animal behavior. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, Advances in Neural Information Processing Systems 29, pages 1947–1955. Curran Associates, Inc., 2016. 3.1

[4] Christopher P. Burgess, Armin Lak, Nicholas A. Steinmetz, Peter Zatka-Haas, Charu Bai Reddy, Elina A. K. Jacobs, Jennifer F. Linden, Joseph J. Paton, Adam Ranson, Sylvia Schröder, Sofia Soares, Miles J. Wells, Lauren E. Wool, Kenneth D. Harris, and Matteo Carandini. High-Yield Methods for Accurate Two-Alternative Visual Psychophysics in Head-Fixed Mice. Cell Reports, 20(10):2513–2524, September 2017. 1

[5] Katherine S. Button, John P. A. Ioannidis, Claire Mokrysz, Brian A. Nosek, Jonathan Flint, Emma S. J. Robinson, and Marcus R. Munafò. Power failure: Why small sample size undermines the reliability of neuroscience. Nature Reviews Neuroscience, 14(5):365–376, May 2013. 2.3

[6] Luke Campagnola, Megan B. Kratz, and Paul B. Manis. ACQ4: An open-source software platform for data acquisition and analysis in neurophysiology research. Frontiers in Neuroinformatics, 8, 2014. 2.2

[7] Anna R. Chambers, Kenneth E. Hancock, Kamal Sen, and Daniel B. Polley. Online stimulus optimization rapidly reveals multidimensional selectivity in auditory cortical neurons. The Journal of Neuroscience, 34(27):8963–8975, July 2014. 1

[8] Xinfeng Chen and Haohong Li. ArControl: An Arduino-Based Comprehensive Behavioral Platform with Real-Time Performance. Frontiers in Behavioral Neuroscience, 11, 2017. 2

[9] Aaron Clauset, Samuel Arbesman, and Daniel B. Larremore. Systematic inequality and hierarchy in faculty hiring networks. Science Advances, 1(1):e1400005, February 2015. 2.3

[10] Sandeep Robert Datta, David J. Anderson, Kristin Branson, Pietro Perona, and Andrew Leifer. Computational Neuroethology: A Call to Action. Neuron, 104(1):11–24, October 2019. 1.2, 3.4

[11] Chance Elliott, Vipin Vijayakumar, Wesley Zink, and Richard Hansen. National Instruments LabVIEW: A Programming Environment for Laboratory Automation and Measurement. JALA: Journal of the Association for Laboratory Automation, 12(1):17–24, February 2007. 1

[12] Navzer D. Engineer, Cherie R. Percaccio, Pritesh K. Pandya, Raluca Moucha, Daniel L. Rathbun, and Michael P. Kilgard. Environmental Enrichment Improves Response Strength, Threshold, Selectivity, and Latency of Auditory Cortex Neurons. Journal of Neurophysiology, 92(1):73–82, July 2004. 5

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

[13] Open Ephys. pyControl. http://www.open-ephys.org/store/pycontrol, November 2019. 1

[14] Alted Francesc and Ivan Vilata. PyTables: Hierarchical Datasets in Python, 2002. 3.1

[15] Pieter Hintjens. ZeroMQ: Messaging for Many Applications. O'Reilly Media, Beijing, 1 edition edition, March 2013. 2.2, 3

[16] Eric Jones, Travis Oliphant, and Pearu Peterson. SciPy: Open Source Scientific Tools for Python. http://www.scipy.org/, 2001. 3

[17] James J. Jun, Nicholas A. Steinmetz, Joshua H. Siegle, Daniel J. Denman, Marius Bauza, Brian Barbarits, Albert K. Lee, Costas A. Anastassiou, Alexandru Andrei, Çağatay Aydın, Mladen Barbic, Timothy J. Blanche, Vincent Bonin, João Couto, Barundeb Dutta, Sergey L. Gratiy, Diego A. Gutnisky, Michael Häusser, Bill Karsh, Peter Ledochowitsch, Carolina Mora Lopez, Catalin Mitelut, Silke Musa, Michael Okun, Marius Pachitariu, Jan Putzeys, P. Dylan Rich, Cyrille Rossant, Wei-Lung Sun, Karel Svoboda, Matteo Carandini, Kenneth D. Harris, Christof Koch, John O'Keefe, and Timothy D. Harris. Fully integrated silicon probes for high-density recording of neural activity. Nature, 551(7679):232–236, August 2017. 1, 2.7, 3

[18] Yarden Katz and Ulrich Bernhard Matter. On the Biomedical Elite: Inequality and Stasis in Scientific Knowledge Production. Berkman Klein Center for Internet & Society Research, November 2017. 2.3

[19] Dexter C. Kozen. Limitations of Finite Automata. In Dexter C. Kozen, editor, Automata and Computability, Undergraduate Texts in Computer Science, pages 67–71. Springer New York, New York, NY, 1997. 3.1

[20] Florian Krause and Oliver Lindemann. Expyriment: A Python library for cognitive and neuroscientific experiments. Behavior Research Methods, 46(2):416–428, June 2014. 2

[21] Sebastiaan Mathôt, Daniel Schreij, and Jan Theeuwes. OpenSesame: An open-source, graphical experiment builder for the social sciences. Behavior Research Methods, 44(2):314–324, June 2012. 2

[22] Philip Meier, Erik Flister, and Pamela Reinagel. Collinear features impair visual detection by rats. Journal of Vision, 11(3), March 2011. 1

[23] Tanmay Nath, Alexander Mathis, An Chi Chen, Amir Patel, Matthias Bethge, and Mackenzie Weygandt Mathis. Using DeepLabCut for 3D markerless pose estimation across species and behaviors. Nature Protocols, 14(7):2152–2176, July 2019. 1, 5

[24] Cristopher M. Niell and Michael P. Stryker. Modulation of Visual Responses by Behavioral State in Mouse Visual Cortex. Neuron, 65(4):472–479, February 2010. 1

[25] Ana Parabucki, Alexander Bizer, Genela Morris, Antonio E. Munoz, Avinash D. S. Bala, Matthew Smear, and Roman Shusterman. Odor Concentration Change Coding in the Olfactory Bulb. eNeuro, 6(1), February 2019. 1

[26] J. M. Pearce, J. C. Molloy, S. Kuznetsov, and S. Dosemagen. Expanding Equitable Access to Experimental Research and STEM Education by Supporting Open Source Hardware Development, January 2019. 2.3

bioRxiv preprint doi: https://doi.org/10.1101/807693; this version posted October 17, 2019. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC 4.0 International license.

[27] Jonathan Peirce, Jeremy R. Gray, Sol Simpson, Michael MacAskill, Richard Höchenberger, Hiroyuki Sogo, and Erik Kastman. PsychoPy2: Experiments in behavior made easy. Behavior Research Methods, 51(1):195–203, February 2019. 2, 3.3

[28] Jacob Reimer, Matthew J. McGinley, Yang Liu, Charles Rodenkirch, Qi Wang, David A. McCormick, and Andreas S. Tolias. Pupil fluctuations track rapid changes in adrenergic and cholinergic activity in cortex. Nature Communications, 7:13289, November 2016. 1

[29] Oliver Rübel, Andrew Tritt, Benjamin Dichter, Thomas Braun, Nicholas Cain, Nathan Clack, Thomas J. Davidson, Max Dougherty, Jean-Christophe Fillion-Robin, Nile Graddis, Michael Grauer, Justin T. Kiggins, Lawrence Niu, Doruk Ozturk, William Schroeder, Ivan Soltesz, Friedrich T. Sommer, Karel Svoboda, Ng Lydia, Loren M. Frank, and Kristofer Bouchard. NWB:N 2.0: An Accessible Data Standard for Neurophysiology. bioRxiv, January 2019. 2.3, 5

[30] Josh Sanders. Sanworks - BPod. https://www.sanworks.io/shop/products.php?productFamily=bpod. 1

[31] Jonny L. Saunders and Michael Wehr. Mice can learn phonetic categories. The Journal of the Acoustical Society of America, 145(3):1168–1177, March 2019. 2.4

[32] Patrick E. Shrout and Joseph L. Rodgers. Psychology, Science, and Knowledge Construction: Broadening Perspectives from the Replication Crisis. Annual Review of Psychology, 69(1):487–510, 2018. 2.3

[33] Joshua H. Siegle, Aarón Cuevas López, Yogi A. Patel, Kirill Abramov, Shay Ohayon, and Jakob Voigts. Open Ephys: An open-source, plugin-based platform for multichannel electrophysiology. Journal of Neural Engineering, 14(4):045003, June 2017. 5

[34] Kay Thurley and Aslı Ayaz. Virtual reality systems for rodents. Current Zoology, 63(1):109–119, February 2017. 1

[35] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. Computing in Science Engineering, 13(2):22–30, March 2011. 3

[36] Guido van Rossum. Glue It All Together With Python. https://www.python.org/doc/essays/omg-darpa-mcc-position/, June 1998. 2.1

[37] Matthew B. Wall. Reliability starts with the experimental tools employed. Cortex, 113:352–354, April 2019. 1, 2.3

[38] Deborah L. Wells. Sensory stimulation as environmental enrichment for captive animals: A review. Applied Animal Behaviour Science, 118(1):1–11, April 2009. 5

[39] Leland Wilkinson. The Grammar of Graphics. In James E. Gentle, Wolfgang Karl Härdle, and Yuichi Mori, editors, Handbook of Computational Statistics: Concepts and Methods, Springer Handbooks of Computational Statistics, pages 375–414. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 3.6

[40] Alexander B. Wiltschko, Matthew J. Johnson, Giuliano Iurilli, Ralph E. Peterson, Jesse M. Katon, Stan L. Pashkovski, Victoria E. Abraira, Ryan P. Adams, and Sandeep Robert Datta. Mapping Sub-Second Structure in Mouse Behavior. Neuron, 88(6):1121–1135, December 2015. 5