

Novel Tools and Methods

Rigbox: An Open-Source Toolbox for Probing Neurons and Behavior

 Jai Bhagat,^{1,*} Miles J. Wells,^{1,2,*}  Kenneth D. Harris,¹ Matteo Carandini,² and Christopher P. Burgess²

<https://doi.org/10.1523/ENEURO.0406-19.2020>

¹UCL Queen Square Institute of Neurology, University College London, London WC2N 5DU, United Kingdom and

²UCL Institute of Ophthalmology, University College London, London EC1V 9EL, United Kingdom

Abstract

Setting up an experiment in behavioral neuroscience is a complex process that is often managed with ad hoc solutions. To streamline this process, we developed Rigbox, a high-performance, open-source software toolbox that facilitates a modular approach to designing experiments (<https://github.com/cortex-lab/Rigbox>). Rigbox simplifies hardware input-output, time aligns datastreams from multiple sources, communicates with remote databases, and implements visual and auditory stimuli presentation. Its main submodule, Signals, allows intuitive programming of behavioral tasks. Here we illustrate its function with the following two interactive examples: a human psychophysics experiment, and the game of Pong. We give an overview of running experiments in Rigbox, provide benchmarks, and conclude with a discussion on the extensibility of the software and comparisons with similar toolboxes. Rigbox runs in MATLAB, with Java components to handle network communication, and a C library to boost performance.

Key words: behavioral; control; experimental; software; toolbox

Significance Statement

Configuring the hardware and software components required to run a behavioral neuroscience experiment and manage experiment-related data are a complex process. In a typical experiment, software is required to design a behavioral task, present stimuli, read hardware input sensors, trigger hardware outputs, record subject behavior and neural activity, and transfer data between local and remote servers. Here we introduce Rigbox, which, to the best of our knowledge, is the only software toolbox that integrates all of the aforementioned software requirements necessary to run an experiment. This MATLAB-based package provides a platform to rapidly prototype experiments. Multiple laboratories have adopted this package to run experiments in cognitive, behavioral, systems, and circuit neuroscience.

Introduction

In behavioral neuroscience, much time is spent setting up hardware and software, and ensuring compatibility between them. Experiments often require configuring disparate software to interface with distinct hardware, and integrating these components is no trivial task. Furthermore, there are often separate software components for designing a behavioral task, running the task, and acquiring, processing, and logging the data. This requires learning the fundamentals of different software

packages and how to make them communicate appropriately.

Consider a typical experiment focused on decision-making, in which a subject chooses a stimulus among a set of possibilities and obtains a reward if the choice was correct (Carandini and Churchland, 2013). The software setup for this experiment may seem simple: ostensibly, all that is required is software to run the behavioral task, and software to handle experiment data. However, when considering

Received October 4, 2019; accepted May 25, 2020; First published June 3, 2020.

The authors declare no competing financial interests.

Author contributions: K.D.H., M.C., and C.P.B. designed research; J.B., M.J.W., and C.P.B. performed research; C.P.B. contributed unpublished reagents/analytic tools; J.B. and M.J.W. analyzed data; J.B., M.J.W., and C.P.B. wrote the paper.

implementation details for these two types of software, the setup can grow quite complex. Running the behavioral task requires software for starting, stopping, and transitioning between task states, presenting stimuli, reading input devices, and triggering output devices. Handling experiment data requires software for acquiring, processing, and logging stimulus history, response history, and subject physiology, and transferring data between servers and databases.

To address this variety of needs in a single software toolbox, we designed Rigbox (github.com/cortex-lab/Rigbox). Rigbox is modular, high-performance, open-source software for running behavioral neuroscience experiments and acquiring experiment-related data. Rigbox facilitates acquiring, time aligning, and managing data from a variety of sources. Furthermore, Rigbox allows users to programmatically and intuitively design and parametrize behavioral tasks via a framework called Signals.

We begin by giving a general overview of Signals, the core package of Rigbox. We illustrate the following two simple interactive examples of its use: an experiment in visual psychophysics, and the game of Pong. Next, we describe how Rigbox runs Signals experiments and manages experiment data. We then discuss the design considerations of Rigbox and the various types of experiments that have been implemented using Rigbox. Last, we detail the requirements of Rigbox and provide benchmarking results. If you wish to try out the code examples used in this paper, please install Rigbox by following the information in the github repository README file (<http://github.com/cortex-lab/Rigbox>).

Signals

Signals is a framework designed for building bespoke behavioral tasks. In Signals, an experiment is built from a reactive network whose nodes (“signals”) represent experiment parameters. This simplifies problems that deal with how experiment parameters change over time by representing relationships between these parameters with straightforward, self-documenting operations. For example, to define a drifting grating, a user could create a signal that changes the phase of a grating as a function of time (Fig. 1). This is shown in the following code below:

```
theta = 2 * pi; % angle of phase in radians
freq = 3; % frequency of phase in Hz
```

This work was funded by the Medical Research Council (Doctoral Training Award to C.P.B.), the Royal Society (Newton International Fellowship to A.J.P.), EMBO (Fellowship to A.J.P.), the Human Frontier Science Program (Fellowship to A.J.P.), and the Wellcome Trust (Grant 205093 to M.C. and K.D.H.). M.C. holds the GlaxoSmithKline/Fight for Sight Chair in Visual Neuroscience, University College London.

*J.B. and M.J.W. contributed equally to writing the manuscript.

C.P. Burgess's present address: DeepMind, 5 New Street Square, London EC4A 3HQ, United Kingdom.

Acknowledgments: We thank Andy Peters, Nick Steinmetz, Max Hunter, Peter Zarka-Haas, Kevin Miller, Hamish Forrest, and other members of the laboratory for troubleshooting, feedback, inspiration, and code contribution.

Correspondence should be addressed to Jai Bhagat at j.bhagat@ucl.ac.uk.
<https://doi.org/10.1523/ENEURO.0406-19.2020>

Copyright © 2020 Bhagat et al.

This is an open-access article distributed under the terms of the [Creative Commons Attribution 4.0 International license](https://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution and reproduction in any medium provided that the original work is properly attributed.

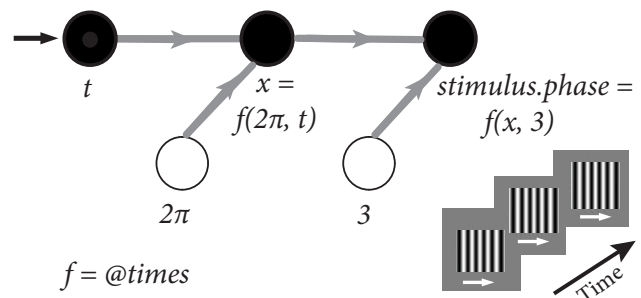


Figure 1. A representation of the time-dependent phase of a visual stimulus in Signals using a clock signal, t . t represents time in seconds since experiment start (its value therefore constantly increases). An unfilled circle represents a constant value: it becomes a node in the network when combined with another signal in an operation (in this instance, via multiplication, represented by the MATLAB function, `times`). The bottom right shows how the phase of the grating changes over time: the white arrow indicates the phase shift direction.

```
stimulus.phase = theta * freq * t; % phase that cycles at 3 Hz
for given stimulus
```

Whenever the clock signal, t , is updated (e.g., by a MATLAB timer callback function), the values of all its dependent signals are then recalculated asynchronously via callbacks. This paradigm is known as functional reactive programming (see D. Lew, “An Introduction to Functional Reactive Programming,” <https://blog.danlew.net/2017/07/27/an-introduction-to-functional-reactive-programming/>).

The operations that can be performed on signals are not just limited to basic arithmetic. Many built-in MATLAB functions (including logical, trigonometric, casting, and array operations) have been overloaded to work on signals as they would on basic numeric or char types. Furthermore, a number of classical functional programming functions (e.g., “map” and “scan”) can be used on signals (Fig. 2). These allow signals to gate, trigger, filter, and accumulate other signals to define a complete experiment.

Example 1: a psychophysics experiment

Our first example of a human-interactive Signals experiment is a script that recreates a psychophysics experiment to study the mechanisms that underlie the discrimination of a visual stimulus (Ringach, 1998). In this experiment, the observer looks at visual gratings (Fig. 3a) that change rapidly and randomly in orientation and phase. The gratings change so rapidly that they summate in the visual system, and the observer tends to perceive two or three of them as superimposed. The task of the observer is to hit the “ctrl” key whenever the orientation of the grating is vertical. At key press, the probability of detection is plotted as a function of stimulus orientation in the recent past. Typically, this exposes a center-surround type of organization, with orientations near vertical eliciting responses, but orientations further away suppressing responses (Fig. 3b). The Signals network representation of this experiment is shown in Figure 4.

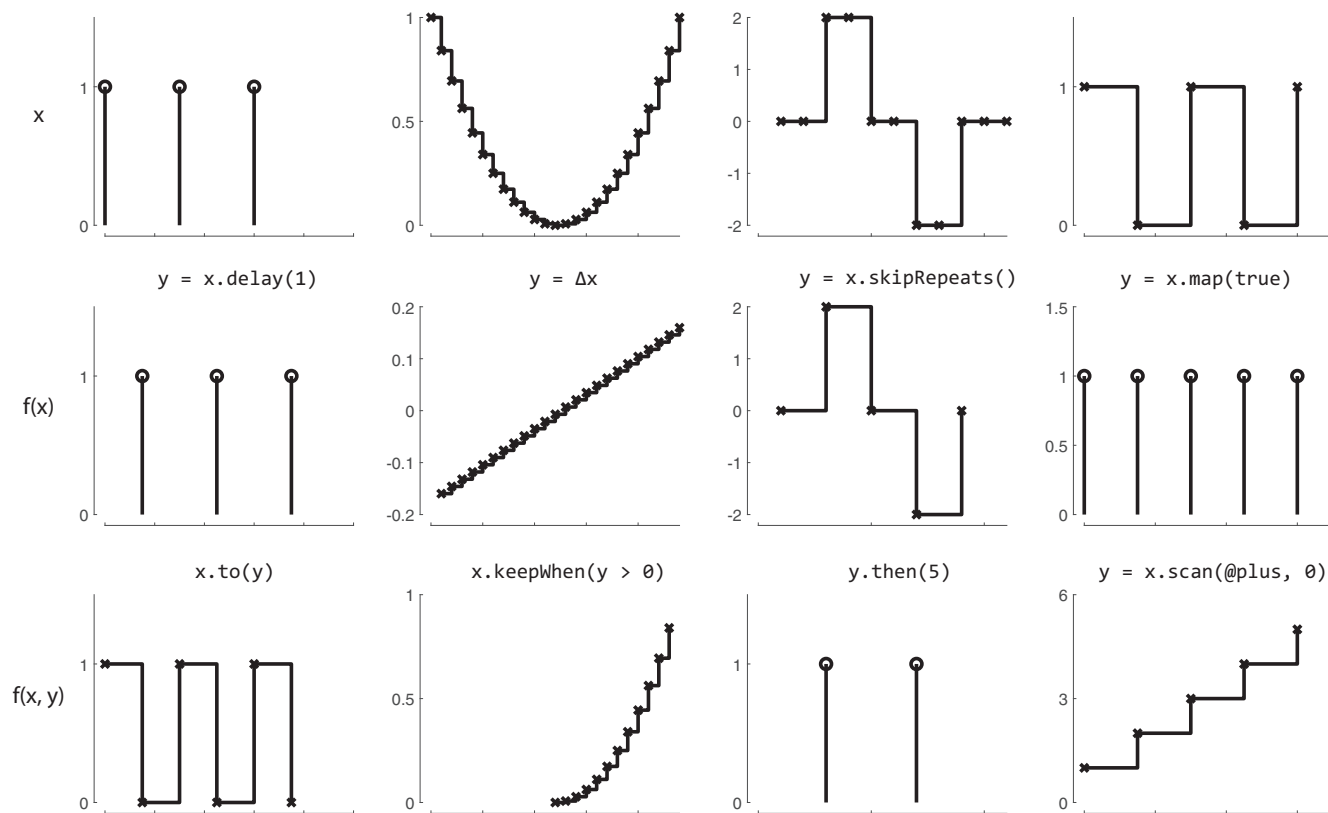


Figure 2. The creation of new signals via example signals methods. Each panel, in which the x-axis represents time and the y-axis represents value, contains a signal. Each column depicts a set of related transformations. The top row contains four arbitrary signals. The second row depicts a signal that results from applying an operation on the signal in the panel above. The third row depicts a signal that results from applying an operation on the signals in the two panels above. Conceptually, each signal can be thought of as both a continuous stream of discrete values, and as a discrete representation whose value changes over time.

To run this experiment, simply run `ringach98` script <https://github.com/cortex-lab/signals/blob/master/docs/examples/ringach98.m> after installing `Rigbox` and press the “Play” button. Below is a breakdown of the 30-odd lines of code:

```

First, some constants are defined:
oris = 0:18:162; % set of orientations, deg
phases = 90:90:360; % set of phases, deg
presentationRate = 10; % Hz
    
```

```

winlen = 10; % length of histogram window, frames
Next, we create a figure:
figh = figure('Name', 'Press "ctrl" key on vertical grating',...
'Position', [680 250 560 700], 'NumberTitle', 'off'); vbox =
uix.
VBox('Parent', figh); % container for the play/pause button
and axes% Create axes for the histogram plot.
axh = axes('Parent', vbox, 'NextPlot', 'replacechildren',
'XTick', oris);
    
```

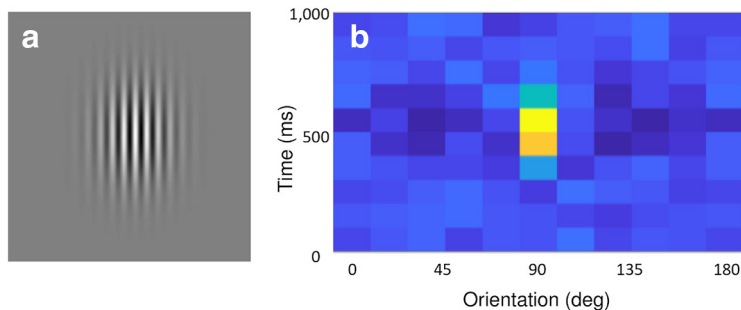


Figure 3. Output shown when running “`ringach98.m`” **a**, A sample grating that the subject is required to respond to via a “ctrl” key press. **b**, A heatmap showing the grating orientations for the 10 frames immediately preceding the key press, summed over all of the key presses for the duration of the experiment. After a few minutes, the distribution of orientations that were presented at each key press resembles a 2D Mexican Hat wavelet, centered on the orientation the subject was reporting at the subject’s average reaction time. In this example, the subject was reporting a vertical grating orientation (90°) with an average reaction time of ~600 ms.

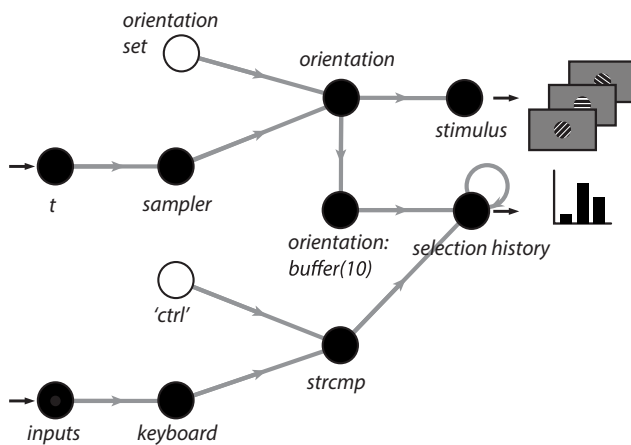


Figure 4. A simplified Signals network diagram of the Ringach experiment. Each circle represents a node in the network that carries out an operation on its direct input. The left-most nodes are inputs to the network, and the values from the right-most layer are used to update the stimulus and the histogram plot. An unfilled circle represents a constant value.

```
xlabel(axh, 'Orientation');
ylabel(axh, 'Time (frames)');
ylim([0 winlen] + 0.5);
vbox.Heights = [30 -1]; % 30 px for the button, the rest for the plot
```

Next, we create our Signals network. The function `playgroundPTB` creates a new Signals network and one input signal, `t`. It creates a start button, which, when pressed, starts a MATLAB timer that periodically updates `t` with the time. Finally, it returns an anonymous function, `setElemsFn`, that, when called with a visual stimulus object, adds the textures to a stimulus renderer:

```
% Create a new Psychtoolbox stimulus window and renderer,
returning a timing
% signal, 't', and function, 'setElemsFn', to load the visual
elements.
```

```
[t, setElemsFn] = sig.test.playgroundPTB(vbox);
net = t.Node.Net; % handle to the network
```

Now, we derive some new signals from UI key press events and the clock signal:

```
% Create a signal from the key board presses.
keyPresses = net.fromUIEvent(figh, 'WindowKeyPress
Fcn');
% Filter it, keeping only 'ctrl' key presses. Turn into logical
signal.
```

```
reports = strcmp(keyPresses.Key, 'ctrl');
% Sample the current time at 'presentationRate'.
sampler = skipRepeats(floor(presentationRate * t));
```

To change the orientation and phase at a given frequency, we derive some indexing signals that will select a value from the orientation and phase sets. The `map` method calls a function with the value of a signal each time it changes. `@(~) foo` is the MATLAB syntax for creating an anonymous function. Each time the sampler signal changes, a new random integer is generated.

```
% Randomly sample orientations and phases by generating
new indices for selecting values from 'oris' and 'phases' each
time 'sampler' updates.
```

```
oriIdx = sampler.map(@(~) randi(numel(oris))); % index for
'oris' array
phaseIdx = sampler.map(@(~) randi(numel(phases))); %
index for @phases@ array
currPhase = phaseIdx.map(@(@idx) phases(idx)); % get
current phase
currOri = oriIdx.map(@(@idx) oris(idx)); % get current ori
```

Next, we derive some signals for updating our plot of reaction times. First, a Boolean array the size of our orientation set is created, then we derive a matrix from these vectors, storing the last 10 orientations presented.

```
% Create a signal to indicate the current orientation (a
Boolean column vector)
oriMask = oris' == currOri;
% Record the last few orientations presented (i.e., 'buffer' the
last few values that 'oriMask' has taken.) as a MxN matrix
where M is the number of orientations (the length of 'oris') and
N is the number of frames ('winlen')
oriHistory = oriMask.buffer(winlen);
```

Each time the user presses the “ctrl” key (represented by the reports signal), the values in the `oriHistory` matrix are added to the histogram via the `scan` method, which initializes the histogram with zeros.

```
% After each keypress, add the @oriHistory@ snapshot to an
accumulating histogram.
```

```
histogram = oriHistory.at(reports).scan(@plus, zeros(numel
(oris), winlen));
```

Now, each time the histogram updates, we call `imagesc` with its value, updating the plot axes.

```
% Plot histogram surface each time it changes.
```

```
histogram.onValue(@(@data) imagesc(oris, 1:winlen, flipud
(data)', 'Parent', axh));
```

Finally, we create the visual stimulus signal and send it to the renderer. The `vis.grating` function returns a subscriptable signal, which has parameter fields related to visual grating properties. When the values of these signal fields are updated, the underlying textures are rerendered by `setElemsFn`.

```
% Create a Gabor with changing orientations and phases.
```

```
grating = vis.grating(t, 'sinusoid', 'Gaussian');
grating.show = true; % set grating to be always visible
grating.orientation = currOri; % assign orientation
grating.phase = currPhase; % assign phase
grating.spatialFreq = 0.2; % cyc/deg
% Add the grating to the renderer.
setElemsFn(struct('grating', grating));
```

With this powerful framework, a user can easily define complex relationships between stimuli, actions, and outcomes to create a complete experiment protocol. This protocol takes the form of a user-written MATLAB function, which we refer to as an “experiment definition” (“exp def”).

When `Rigbox` initializes an experiment, a new Signals network is created with input layer signals representing time, experiment epochs (e.g., new trials), and hardware input devices (e.g., position sensors). These input signals are passed into the `exp def` function, and the code in the `exp def` operates on these signals to create new signals that are added to the network (Fig. 5). The `exp def` is called just once to set up this network.

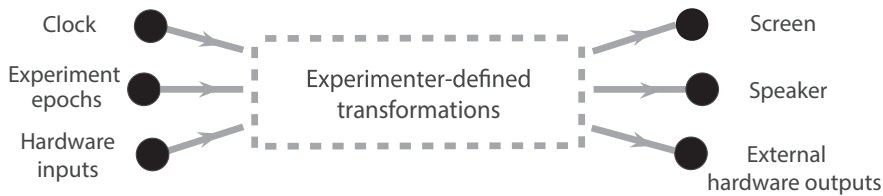


Figure 5. A Signals representation of an experiment. There are three types of input signals in the network, representing a clock, experiment epochs (e.g., new trials and experiment start and end conditions), and hardware input devices (e.g., an optical mouse, keyboard, rotary encoder, lever). In an exp def, the user defines transformations that create new signals (data not shown) from these input signals, which ultimately drive outputs (e.g., a screen, speaker, and external hardware such as a reward valve). The exp def is called once to create these experimenter-defined signals, which are updated during experiment runtime as the input signals they depend on are updated.

At experiment start, values are posted to the input signals of the network. During experiment runtime, these input signals are continuously updated within the main of the experiment while loop or through user interface UI and timer callbacks occur. For example, a position sensor input device may be read from continuously in a while loop to update the signal representing this device. These input signal updates asynchronously propagate to the dependent signals that were created in the exp def. The experiment ends when the “experiment stop” signal is updated (i.e., when all trial conditions have occurred or after a specified duration of time).

The following is a brief overview of the structure of an exp def. An exp def takes up to seven input arguments:

```
function expDef(t, events, params, visStim, inputs, outputs, audio)
```

In order, these are as follows: (1) the clock signal; (2) an events structure containing signals that define experiment epochs, and signals—from those created within the exp def—which the experimenter wishes to log; (3) a signal parameters structure that defines session- or trial-specific signals whose values can be changed directly within a graphical UI (GUI) before starting an experiment—signal parameter defaults are set within the exp def, and parameter sets can be saved and loaded across subjects and experiments; (4) the visual stimuli handler, which contains as fields all signals that parametrize the display of visual stimuli—any visual stimulus signal can be assigned various elements, which the viewing model allows to be defined in visual degrees, for being rendered to a screen, and a visual stimulus can be loaded directly from a saved image file; (5) an inputs structure containing signals that map to hardware input devices; (6) an outputs structure containing signals that map to external hardware output devices; and (7) the audio stimuli handler, which can contain as fields signals that map to available audio devices.

Tutorials on creating an exp def, examples of exp defs, and standalone scripts (including those mentioned in this article), and an in-depth overview of Signals can be found in the signals/docs folder <https://github.com/cortex-lab/signals/tree/master/docs>.

Example 2: Pong

A second human-interactive Signals experiment contained in the Rigbox repository is an exp def that runs the classic computer game Pong (Fig. 6). The signal that sets

the player’s paddle position is mapped to the optical mouse. The epoch structure is set so that a trial ends on a score, and the experiment ends when either the player or cpu (central processing unit) reaches a target score. The code is divided into the following three sections: (1) initializing the game; (2) updating the game; and (3) creating visual elements and defining exp def signal parameters. To run this exp def, follow the directions in the header of the signalsPong file <https://github.com/cortex-lab/signals/blob/master/docs/examples/signalsPong.m>. Because the file itself (including copious documentation) is >300 lines, we will share only an over-

view here; however, readers are encouraged to look through the full file at their leisure. `function signalsPong(t, events, p, visStim, inputs, outputs, audio)`

In this first section, we define constants for the game, arena, ball, and paddles:

```
%% Initialize the game
% how often to update the game in secs
[...]
% initial scores and target score
[...]
% size of arena, ball, and paddle: [w h] in visual degrees
[...]
% ball angle, and ball velocity in visual degrees per second
[...]
% cpu and player paddle X-axis positions in visual degrees
[...]
```

The helper function, `getYPos`, returns the y-position of the cursor, which will be used to set the player paddle: `function yPos = getYPos()`

```
[...]end
% get cursor’s initial y-position
cursorInitialY = events.expStart.map(@(~) getYPos);
```

In the second section, we define how the ball and paddle interactions update the game:

```
%% Update game
% create a signal that will update the y-position of the player’s paddle using ‘getYPos’
‘playerPaddleYUpdateVal = (cursor.map(@(~) get YPos)-cursorInitialY)*cursorGain
% make sure the y-value of the player’s paddle is within the screen bounds,
playerPaddleBounds = cond(...
playerPaddleYUpdateVal > arenaSz(2)/2, arenaSz(2)/2, ...
playerPaddleYUpdateVal < -arenaSz(2)/2, -arenaSz(2)/2, ...
true, playerPaddleYUpdateVal);
% and only updates every @tUpdate secs
playerPaddleY = playerPaddleBounds.at(tUpdate);
% Create a struct, @gameDataInit, holding the initial game state
gameDataInit = struct;
```

```
...
% Create a subscriptable signal, @gameData, whose fields represent the current
```

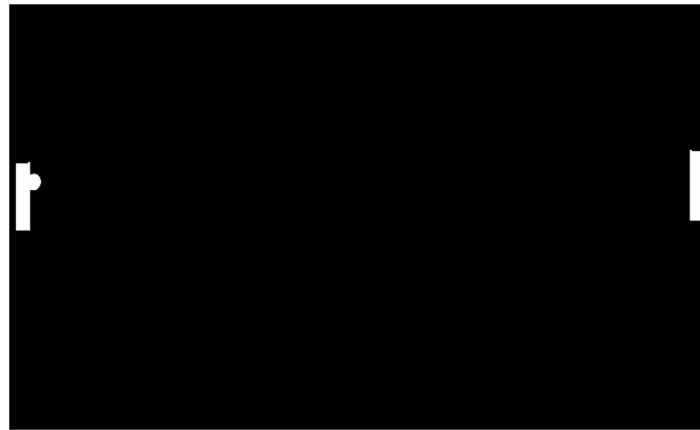


Figure 6. A screenshot of Pong run in Signals. The top shows the paddles and ball during gameplay. The bottom shows the GUI used to launch the game. The paddle colors (represented by an RGB vector) and target score are examples of global signal parameters that can be set once before starting the game. The ball color is an example of a conditional signal parameter that changes randomly after every trial (in this case, after a score) between the arrays indicated in each row (which in this case specify the colors white, red, and blue).

% game state (total scores, etc.), and which will be updated every @tUpdate secs

```
gameData = playerPaddleY.scan(@updateGame,
gameDataInit).subscriptable;
```

The helper function, updateGame, updates gameData. Specifically, it updates the data structure with ball angle,

```
velocity, position, cpu paddle position, and player and cpu
scores, based on the current ball position, which is updated at
each sampled timestep, as follows:
function gameData =
updateGame(gameData, playerPaddleY)
[...]end
% define trial end (when a score occurs)
```

```

anyScored = playerScore | cpuScore;
events.endTrial = anyScored.then(true);
% define game end (when player or cpu score reaches target
score)
endGame = (playerScore == targetScore) | (cpuScore ==
targetScore);
events.expStop = endGame.then(true);
[...]
In the final section, we create the visual elements representing
the arena, ball, and paddles, and define the exp def signal pa-
rameters, as follows:
%% Define the visual elements and the experiment signal
parameters
% create the arena, ball, and paddles as 'vis.patch' subscrip-
table signals
arena = vis.patch(t, 'rectangle');
ball = vis.patch(t, 'circle');
ball.color = p.ballColor;
playerPaddle = vis.patch(t, 'rectangle');
cpuPaddle = vis.patch(t, 'rectangle');
% assign the arena, ball, and paddles to the 'visStim' sub-
scriptable signal handler
Stim.arena = arena;
visStim.ball = ball;
visStim.playerPaddle = playerPaddle;
visStim.cpuPaddle = cpuPaddle;
% define parameters that will be displayed in the GUI
try
% 'p.ballColor' is a conditional signal parameter: on any
given trial, the ball
% color will be chosen at random among three colors: white,
red, blue
p.ballColor = [1 1 1; 1 0 0; 0 0 1]; % RGB color vector array
% 'p.targetScore' is a global signal parameter: it can be
changed via the GUI used
% to run this exp def before starting the game
p.targetScore = 5;
catch
end

```

Running experiments and managing data in Rigbox

Rigbox contains a suite of packages for interfacing with hardware, acquiring and managing data, communicating with a remote database, time aligning events from a variety of sources, and implementing a user interface for managing experiments.

Rigbox simplifies experiments by providing an abstract interface for hardware interactions. All hardware devices, including screens and speakers, are represented by abstract classes that provide a basic set of interface methods. Methods for initializing, configuring, and communicating with a particular device are handled by specific subclasses. This design choice avoids the creation of device-specific dependencies within the toolbox and the experiment code of the user. In this way, hardware devices can be swapped without modifying code or affecting the experiment workflow, and adding support for new devices is straightforward. For example, to support a new

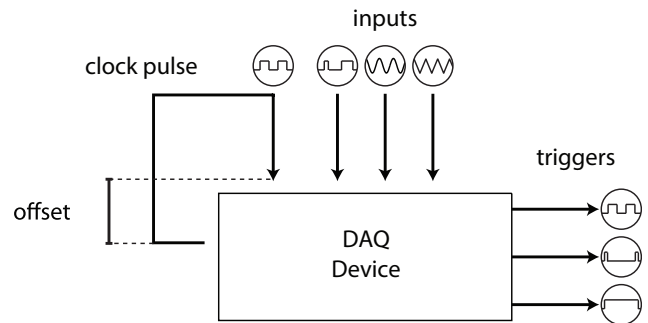


Figure 7. A representation of a Timeline object. The topmost signal is the main timing signal, “chrono,” which is used to unify all time stamps across computers during an experiment. The “inputs” represent different hardware and software input signals read by a NI-DAQ, and the “triggers” represent different hardware output signals, triggered by a NI-DAQ.

multifunction input/output (I/O) device (e.g., an Arduino or other microcontroller), one could simply extend the `+hw/DaqController` class, and to support a new hardware input sensor (e.g., a lever or joystick), one could simply subclass the `+hw/PositionSensor` class.

Intuitive and robust data management is another essential feature of Rigbox. Simple function wrappers save and locate data via human-readable experiment reference strings that reflect straightforward experiment directory structures (i.e., `subject/date/session`). Data can be saved both locally and remotely, and even distributed across multiple servers. Rigbox uses a single paths config file, making it simple to change the location of data and configuration files. Furthermore, this code can be easily integrated with the personal code of a user to generate read and write paths for arbitrary datasets. A Parameters class, which sets, validates, and asserts experiment conditions for each experiment, simplifies data analysis across experiments by standardizing parameterization. Rigbox can also communicate with an Alyx database to query and post data related to a subject or session. Alyx is a lightweight meta-database that can be hosted on an internal server or in the cloud (e.g., via Amazon Web Services). Alyx allows users to organize experiment sessions and their associated files, and to keep track of subject information, such as diet, breeding, and surgeries (Bonacchi et al., 2020).

Experiments typically involve recording simultaneously from many devices, and temporal alignment of these recordings can be challenging. Rigbox contains a class called Timeline, which manages the acquisition and generation of clocking pulses via a National Instruments multifunction I/O data acquisition device (NI-DAQ; Fig. 7). The main clocking pulse of Timeline, “chrono,” is a digital square wave sent out from the NI-DAQ that can flip each time a new chunk of data and are available to the NI-DAQ. A callback function to this flip event collects the NI-DAQ time stamp of the scan where the flip occurred. The difference between this time stamp and the system time recorded when the flip

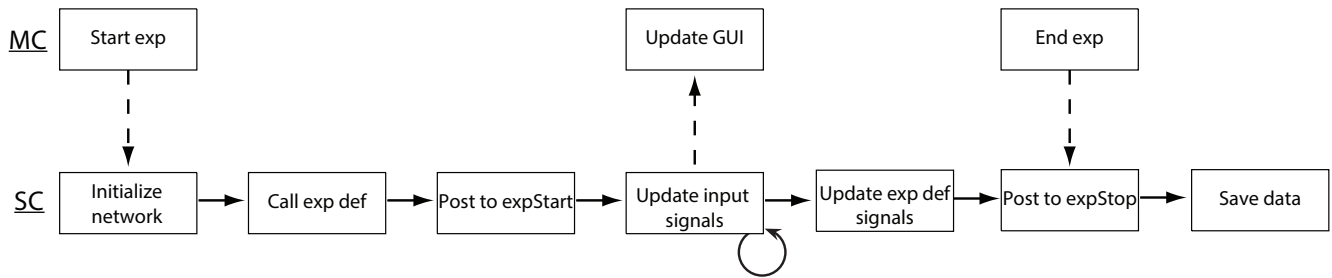


Figure 8. A simplified chronology of events that occur when starting an experiment via the MC GUI. Pushing the “Start” button on the MC GUI sends a message to SC to initialize a Signals network, then call the user’s Signals exp def to create new signals within the network, then post to the ‘expStart’ signal to start the experiment. After starting the experiment, the network input signals are continuously updated via callbacks (e.g., via a MATLAB timer callback, or by reading from hardware input devices), which update the rest of the signals in the network (i.e., those signals defined in the user’s exp def). These updates can then be displayed back to the user on the MC GUI. This continues until the experiment is either ended from the MC GUI, or a condition is met within the user’s exp def that updates the ‘expStop’ signal. After the experiment is ended, experiment data are saved.

command was sent is recorded as an offset time. This offset time can be used to unify all event time stamps across computers: all event time stamps are recorded in time relative to chrono. A Timeline object can acquire any number of hardware or software events [e.g., from hardware inputs directly wired to the NI-DAQ, or UDP (user datagram protocol) messages sent from another computer] and record their values with respect to this offset. For example, a Timeline object can record when a reward valve or laser shutter is opened, a sensor is interacted with, or a screen displaying visual stimuli is updated. In addition to chrono, a Timeline object can also output TTL (transistor–transistor logic) and clock pulses for triggering external devices (e.g., to acquire frames at a specific rate).

Last, Rigbox provides an intuitive yet powerful user interface for running experiments. For this, two

computers are required. An experiment is started from a GUI on one computer, referred to as the “Master Computer” (MC), which runs the experiment on a recording rig, referred to as the “Stimulus Computer” (SC; Fig. 8). An SC is responsible for stimuli presentation, rig hardware interaction, and data acquisition. The MC GUI is used to select, parameterize, and start experiments (Fig. 9). Customizable experiment panels can also be displayed within a different tab in the MC GUI to monitor experiments (Fig. 10). MC and SC communicate during runtime via TCP/IP (transmission control protocol/internet protocol; using WebSockets), and MC can communicate with multiple SCs simultaneously to run multiple experiments in parallel.

Instructions for installation and configuration can be found in the README file and the docs/setup folder

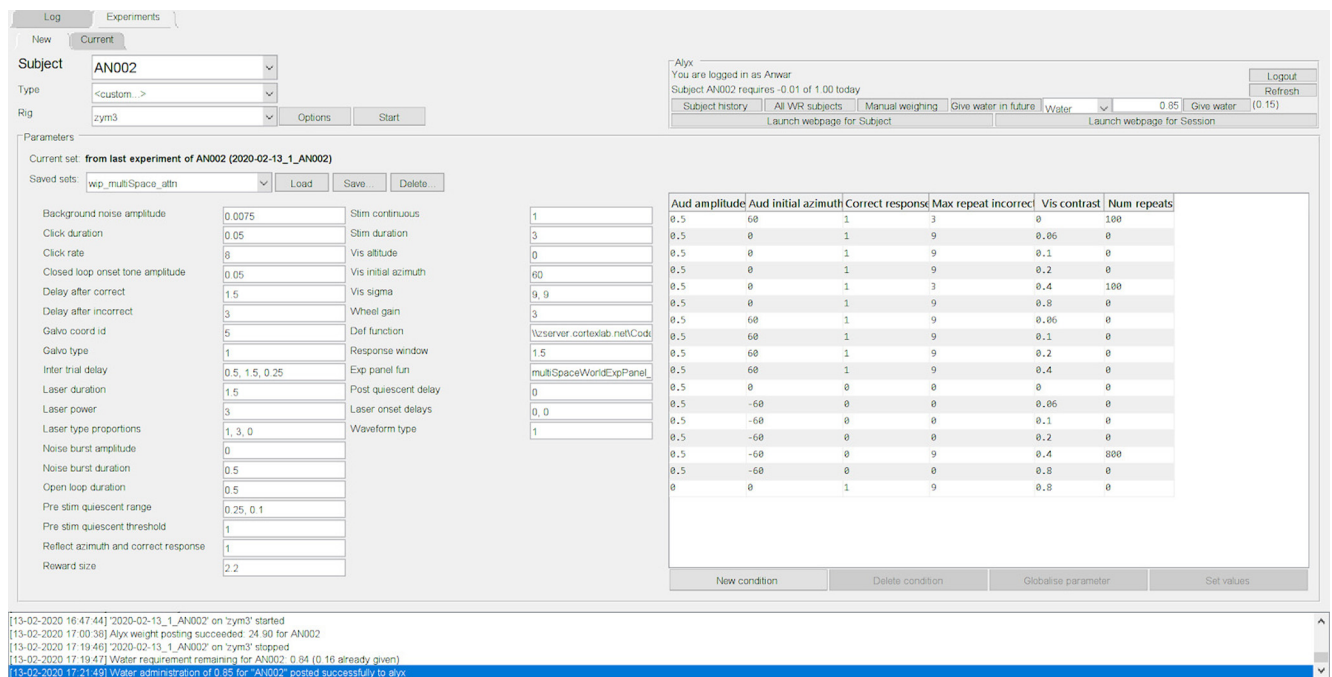


Figure 9. The new experiments tab within the MC GUI. This tab allows a user to select a subject, experiment type, and rig on which to run an experiment. Additionally, rig-specific options can be set via the “Options” button, and signal parameters for the behavioral task can be set via the editable parameter fields.

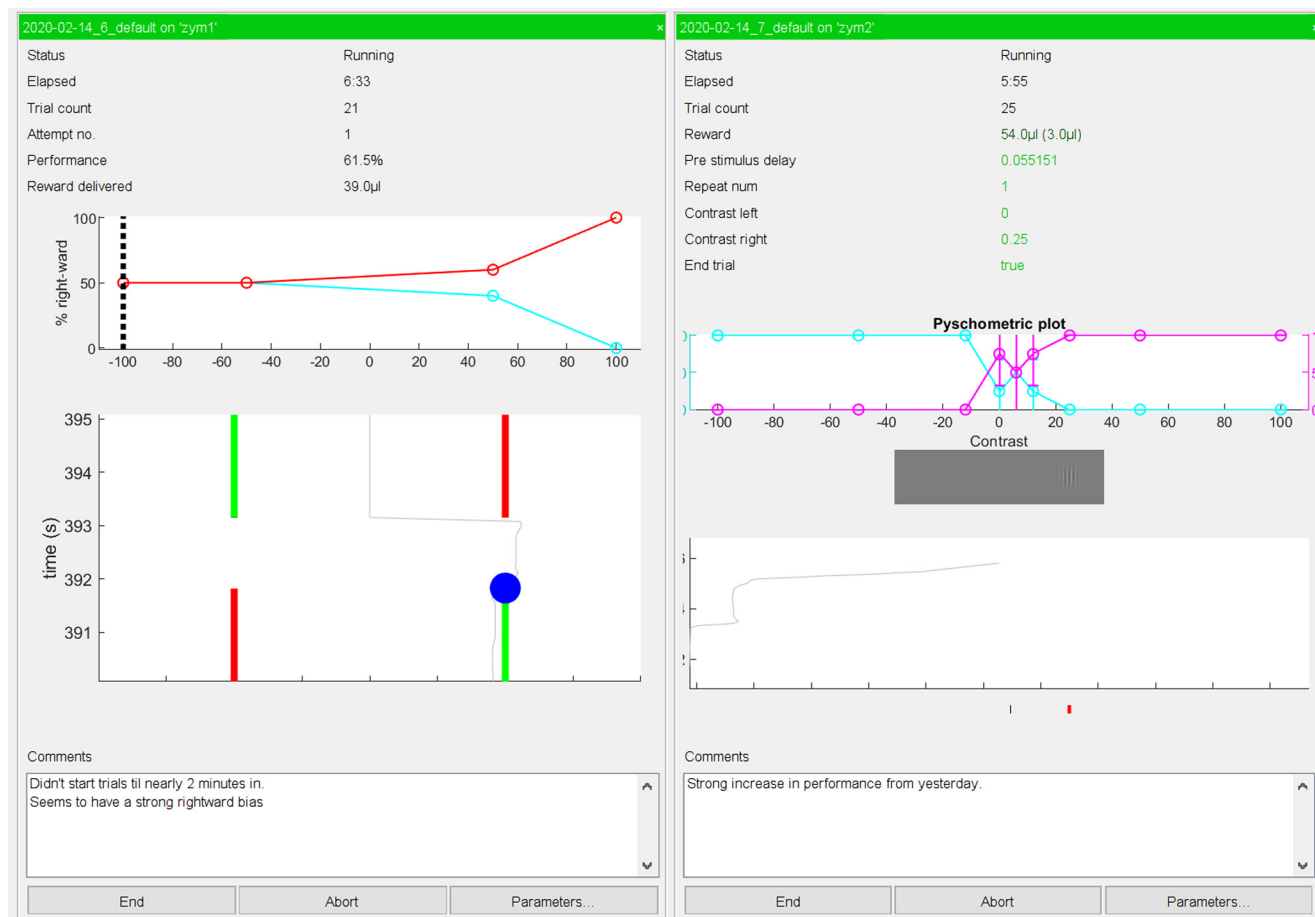


Figure 10. Experiment panels with live updates for two experiments. The top text fields in each panel display experiment information such as elapsed time, trial number, and the current running total of delivered reward. Below the text fields is a psychometric plot showing task performance for specific types of trials, and below this is a plot showing the real-time trace of a hardware input device (the panel on the left shows a two-alternative unforced choice task for which the green bar indicates the direction of the action the subject must make to receive a reward). There is also a text field for logging comments that can be immediately posted to an Alyx database. These experiment panels are highly customizable.

<https://github.com/cortex-lab/Rigbox/tree/master/docs/setup>. This includes information on required dependencies, setting data repository locations, configuring hardware devices, and enabling communication between the MC and SC computers. Hardware and software requirements can also be found in the repository README and the Requirements and benchmarking section of this article.

Data availability

Rigbox is currently under active, test-driven development. All our code is open source, distributed under the Apache 2.0 license at <https://github.com/cortex-lab/Rigbox> (Extended Data 1), and we encourage users to contribute. Please see the contributing section of the README for information on contributing code and reporting issues. When using Rigbox to run behavioral tasks and/or acquire data, please cite this publication.

Discussion

In our laboratory, Rigbox is at the core of our operant, passive, and conditioning experiments. The principal

behavioral task we use is a two-alternative forced choice visual stimulus discrimination task (Burgess et al., 2017). Using Rigbox, we have been able to rapidly prototype multiple variants of this task, including unforced choice, multisensory choice, behavior matching, and bandit tasks, using wheels, levers, balls, and lick detectors. The Signals exp defs for each variant act as a concise and intuitive record of the task design. In addition, Rigbox has made it easy to combine these tasks with a variety of recording techniques, including electrode recordings, 2-photon imaging, and fiber photometry; and neural perturbations, such as scanning laser inactivation and dopaminergic stimulation (Jun et al., 2017; Jacobs et al., 2018; Zátka-Haas et al., 2018; Shimaoka et al., 2019; Steinmetz et al., 2019; Armin et al., 2020). Rigbox has also enabled us to scale our behavioral training: because one MC can control multiple SCs, we run and manage many experiments simultaneously.

Often, experiments are iterative: task parameters are added or modified many times over, and finding an ideal parameter set can be an arduous process. Rigbox allows a user to develop and test an experiment without having

Table 1: Comparison of major features across behavioral control system toolboxes

	BControl	pyBpod	pyControl	VirMen	Rigbox
Behavioral task design paradigm	Procedural	Procedural	Procedural	Object oriented	Functional reactive
Presents visual stimuli? 3D/VR environments?	No	No	No	Yes, yes	Yes, no
Interfaces with hardware?	Yes	Yes	Yes	Yes	Yes
Time aligns multiple datastreams?	Yes	Yes	Yes	No	Yes
Communicates with a remote database?	Yes	Yes	No	No	Yes
Contains unit and integration tests?	?	?	Yes	?	Yes

The top row contains the toolbox names, and the first column contains information on the implementation of a feature. Note: the toolboxes and features mentioned in this table are not exhaustive.

to worry about boilerplate code and UI modifications, as these are handled by Rigbox packages in a modular fashion. Much of the code is object oriented with most aspects of the system represented as configurable objects. Given the modular nature of Rigbox, new features and hardware support may be easily added, provided there is driver support in MATLAB.

To the best of our knowledge, Rigbox is the most complete behavioral control software toolbox currently available in the neuroscience community; however, several other toolboxes implement similar features in different ways (Aronov and Tank, 2014; see also Bpod Wiki, <https://sites.google.com/site/bpodddocumentation/home>; BControl Behavioral Control System, https://brodywiki.princeton.edu/bcontrol/index.php?title=Main_Page; T. Akam, <https://pycontrol.readthedocs.io/en/latest/>; Table 1). Some of these toolboxes also include some features not currently available in Rigbox, for example, microsecond precision triggering of within-trial events, and creating 3D virtual environments. Indeed, the features used by a particular toolbox have advantages (and disadvantages) depending on the user’s desired experiment.

There are pros and cons to following different programming paradigms for software developers who decide how users will design behavioral tasks. Generally, three main paradigms exist: procedural, object oriented, and functional reactive. Here, in the context of programmatic task design, we briefly discuss the differences between these paradigms and in which scenarios one may be favored over the others. Note that here we discuss only the aspect of a toolbox that deals with behavioral task design, not the overall structure of a toolbox (e.g., Rigbox is built on an object-oriented paradigm, but Signals provides a functional reactive paradigm in which to implement a behavioral task).

A procedural approach to task design is probably the most familiar to behavioral neuroscientists. This approach focuses on “how to execute” a task by explicitly defining a control flow that moves a task from one state to the next. The Bcontrol, pyBpod, and pyControl toolboxes follow this paradigm by using real-time finite state machines (RTFSMs), which control the state of a task (e.g., initial state, reward, punishment) during each trial. Some advantages of this approach are that it is simple and intuitive, and guarantees event timing precision down to the minimum cycle of the state machine (e.g., Bcontrol RTFSMs run at a minimum cycle of 6 kHz). Some disadvantages of this approach are that the memory for task parameters are limited by the number of states in the RTFSM, and that the discrete implementation of states is not amenable to experiments that seek to

control parameters continuously (e.g., a task that uses continuous hardware input signals).

Like the procedural approach to task design, an object-oriented approach also tends to be intuitive: objects can neatly represent the state of an experiment via datafields. Objects representing experimental parameters can easily pass information to each other and trigger experimental states via event callbacks. The VirMen toolbox implements this approach by treating everything in the virtual environment as an object and having a runtime function update the environment by performing method calls on the objects based on input sensor signals from a subject performing a task. Some disadvantages of this approach are that the speed of experimental parameter updates are limited by the speed at which the programming language performs dynamic binding (which is often much slower than the RTFSM approach discussed above), and that operation “side effects” (which can alter an experiment’s state in unintended ways) are more likely to occur due to the emphasis on mutability, when compared with a pure procedural or functional reactive approach.

By contrast, Signals follows a functional reactive approach to task design. As we have seen, some advantages of this approach include simplifying the process of updating experiment parameters over time, endowing

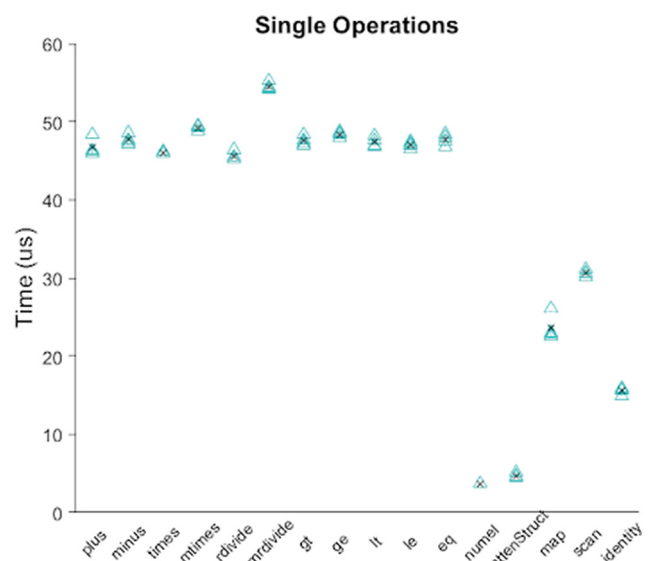


Figure 11. Benchmarking results for operations (specified by the x-axis) on a single signal. The black “x” shows the mean value per group.

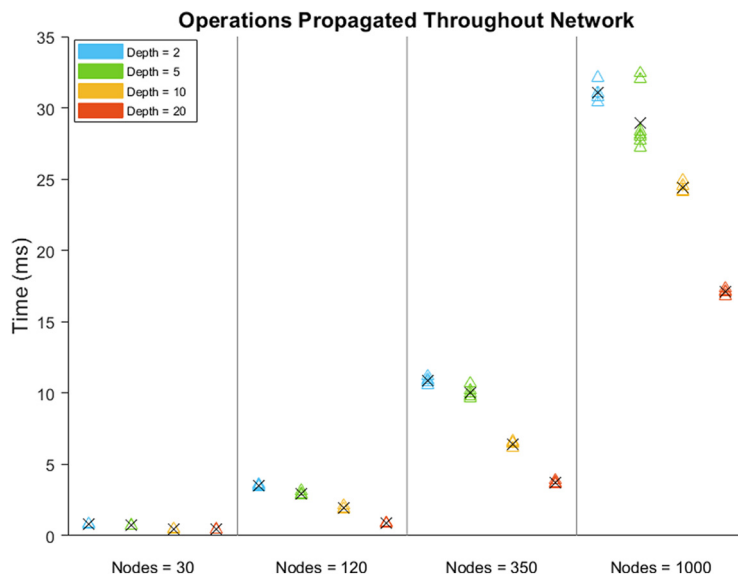


Figure 12. Benchmarking results for updating every signal in a network, for networks of various number of signals (nodes) spread over various number of layers (depth). The black “x” shows the mean value per group.

parameters with memory, and facilitating discrete and continuous event updates with equal ease. In general, a task specification in this paradigm is declarative, which can often make it clearer and more concise than in other paradigms, where control flow and event handling code can obscure the semantics of the task. Some disadvantages are that it suffers from similar speed limitations as in an object-oriented approach, and programmatically designing a task in a functional reactive paradigm is probably unfamiliar to most behavioral neuroscientists. When initially thinking about how a functional reactive network runs a behavioral task, it may be helpful to think of experiment parameters as nodes in the network that get updated via callbacks; there are no procedural calls to the network during experiment runtime.

When considering the entire set of behavioral tasks, no single programming paradigm is perfect, and it is therefore important for a user to consider the goals for the implementation of their task accordingly.

Hardware requirements

For most experiments, typical, contemporary, factory-built desktops running Windows 10 with dedicated graphics cards should suffice. Specific requirements of an SC will depend on the complexity of the experiment. For example, running an audiovisual integration task on three screens requires quality graphics and sound cards. SCs may additionally require a multifunction I/O device to communicate with external rig hardware, of which only NI-DAQs (e.g., NI-DAQ USB 6211) are currently supported.

Hardware Delay Times

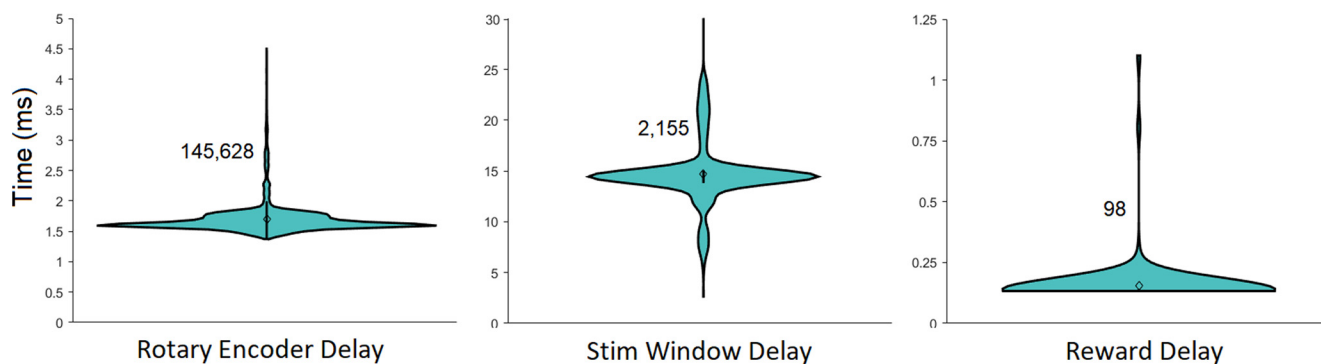


Figure 13. Delay times for specific updates when running a 2AFC (two-alternative forced choice) visual contrast discrimination task. The number next to each violin plot indicates the number of samples in the group. “Rotary Encoder delay” is the time between polling consecutive position values from a rotary encoder. “Stim Window Delay” is the time between triggering a display to be rendered, and its complete render on a screen. “Reward Delay” is the time between triggering and opening a reward valve. The 99th percentile outliers were not included in the plot for “Rotary Encoder delay”: there were 98 instances in which the delay took between 200 and 600 ms, due to the execution time of the NI-DAQmx MATLAB package when sending analog output (reward delivery) via the USB-6211 DAQ.

Below are some minimum hardware specifications required for computers that run Rigbox:

- CPU: 4 logical processors @ 3.0 GHz base speed (e.g., Intel Core i5-6500)
- RAM: DDR4 16 GB @ 2133 MHz (e.g., Corsair Vengeance 16 GB)
- GPU: 2 GB @ 1000 MHz base and memory speed (e.g., NVIDIA Quadro P400).

Software requirements

Similar to the hardware requirements, software requirements for an SC will depend on the experiment. For example, if acquiring data through a NI-DAQ, the SC will require the MATLAB NI-DAQmx support package in addition to the following minimum requirements:

- OS: 64 Bit Windows 7 (or later)
- Libraries: Visual C++ Redistributable Packages for Visual Studio 2013 and 2015
- MATLAB: 2018b or later, including the Data Acquisition Toolbox
- Community MATLAB toolboxes:
 - GUI Layout Toolbox (version 2 or later)
 - Psychophysics Toolbox (version 3 or later).

Benchmarking

Fast execution of experiment runtime code is crucial for performing and accurately analyzing results from a behavioral experiment. Here we provide benchmarking results for the Signals framework. We include results for individual operations on a signal and for operations that propagate through each signal in a network. Single built-in MATLAB operations and Signals-specific methods are consistently executed in the microsecond range (Fig. 11). The network used in a typical two-alternative forced stimulus discrimination task (<https://github.com/cortex-lab/signals/blob/558170702aa2e6962c58f8b2a7b603a96b2c6b1a/docs/examples/advancedChoiceWorld.m>) contains 338 signals spread over 10 layers; a similar network of 350 signals spread over 20 layers can update all signals in under 5 ms, and a network of 120 signals spread over 20 layers can update all signals with submillisecond precision (Fig. 12). Last, we include results for reading from and triggering hardware devices in the above-mentioned stimulus discrimination task.

Updates of the position of a rotary encoder used to indicate choice typically took <2 ms, the time between rendering and displaying the visual stimulus typically took <15 ms, and the delay between triggering and delivering a reward was typically under 0.2 ms (Fig. 13).

All of the results in the Benchmarking section were obtained from running MATLAB 2018b on a Windows 10 64 bit OS with an Intel core i7 8700 processor and 16 GB DDR4 dual-channel RAM clocking at a double data rate of 2133 MHz. Because single executions of signals operations were too quick for MATLAB to measure

precisely, we repeated operations 1000 times and divided the MATLAB returned measured time by 1000. The Performance Testing Framework in MATLAB 2018b was used to obtain these results. https://github.com/cortex-lab/signals/blob/558170702aa2e6962c58f8b2a7b603a96b2c6b1a/tests/Signals_perftest.m contains the code used to generate the results shown in Figures 11 and 12; <https://github.com/cortex-lab/signals/tree/558170702aa2e6962c58f8b2a7b603a96b2c6b1a/tests/results> contains a table of these data; and <https://github.com/cortex-lab/signals/tree/558170702aa2e6962c58f8b2a7b603a96b2c6b1a/tests/results> contains the data used to generate the results shown in Figure 13A. National Instruments USB-6211 was used as the data acquisition I/O device.

References

- Abbott LF, Angelaki DE, Carandini M, Churchland AK, Dan Y, Dayan P, Deneve S, Fiete I, Ganguli S, Harris KD, Häusser M, Hofer S, Latham PE, Mainen ZF, Mrsic-Flogel T, Paninski L, Pillow JW, Pouget A, Svoboda K, Witten IB, et al. (2017) An international laboratory for systems and computational neuroscience. *Neuron* 96:1213–1218.
- Armin L, Okun M, Moss MM, Gurnani H, Farrell K, Wells MJ, Reddy CB, Kepecs A, Harris KD, Carandini M (2020) Dopaminergic and Prefrontal Basis of Learning from Sensory Confidence and Reward Value. *Neuron* 105:700–711.e6.
- Aronov D, Tank DW (2014) Engagement of neural circuits underlying 2D spatial navigation in a rodent virtual reality system. *Neuron* 84:442–456.
- Bonacchi N, Chapuis G, Churchland AK, Harris KD, Hunter M, Rossant C (2020) Data architecture for a large-scale neuroscience collaboration. *BioRxiv*. Advance online publication. Retrieved June 4, 2020. doi:10.1101/827873.
- Burgess CP, Lak A, Steinmetz NA, Zatzka-Haas P, Bai Reddy C, Jacobs EAK, Linden JF, Paton JJ, Ranson A, Schröder S, Soares S, Wells MJ, Wool LE, Harris KD, Carandini M (2017) High-yield methods for accurate two-alternative visual psychophysics in head-fixed mice. *Cell Rep* 20:2513–2524.
- Carandini M, Churchland AK (2013) Probing perceptual decisions in rodents. *Nat Neurosci* 16:824–831.
- Jacobs EAK, Steinmetz NA, Carandini M, Harris KD (2018) Cortical state fluctuations during sensory decision making. *BioRxiv* 348193. doi: 10.1101/348193.
- Jun JJ, Steinmetz NA, Siegle JH, Denman DJ, Bauza M, Barbarits B, Lee AK, Anastassiou CA, Andrei A, Aydin Ç, Barbic M, Blanche TJ, Bonin V, Couto J, Dutta B, Gratiy SL, Gutnisky DA, Häusser M, Karsh B, Ledochowitsch P, et al. (2017) Fully integrated silicon probes for high-density recording of neural activity. *Nature* 551:232–236.
- Ringach DL (1998) Tuning of orientation detectors in human vision. *Vision Res* 38:963–972.
- Shimaoka D, Steinmetz NA, Harris KD, Carandini M (2019) The impact of bilateral ongoing activity on evoked responses in mouse cortex. *Elife* 8:e43533.
- Steinmetz NA, Zatzka-Haas P, Carandini M, Harris KD (2019) Distributed correlates of visually-guided behavior across the mouse brain. *Nature* 576:266–273.
- Zatzka-Haas P, Steinmetz NA, Carandini M, Harris KD (2018) Distinct contributions of mouse cortical areas to visual discrimination. *BioRxiv*. Advance online publication. Retrieved December 21, 2018. doi: 10.1101/501627.