improv: A flexible software platform for adaptive neuroscience experiments

Anne Draelos[1,2*], Maxim Nikitchenko[3], Chaichontat Sriworarat[1,2], Daniel Sprague[1,2], Matthew D. Loring[3], Eftychios Pnevmatikakis[4], Andrea Giovannucci[5], Eva A. Naumann[3†*], & John M. Pearson[1,2,3,6,7†]

[1]Department of Biostatistics & Bioinformatics, Duke University, Durham NC 27710, USA,

[2]Center for Cognitive Neuroscience, Duke University, Durham NC 27708, USA,

[3]Department of Neurobiology, Duke University, Durham NC 27710, USA,

[4]Center for Computational Mathematics, Flatiron Institute, New York, NY 10010, USA,

[5]Joint Department of Biomedical Engineering, University of North Carolina at Chapel Hill / North Carolina State University, Chapel Hill NC 27599, USA,

[6]Department of Electrical & Computer Engineering, Duke University, Durham NC 27708, USA,

[7]Department of Psychology & Neuroscience, Duke University, Durham NC 27708, USA.

* Corresponding authors: amw73@duke.edu, eva.naumann@duke.edu

[†]These authors jointly supervised this work.

Abstract

Neuroscientists now routinely record the activity of large numbers of neurons at high temporal and spatial resolution. With these capabilities comes the promise of causally intervening during these recordings by perturbing neurons or changing experimental conditions, requiring tight integration between data acquisition, analysis, and manipulation. Unfortunately, solutions for real-time interventions are rare, difficult to design and implement, and remain largely unused. Here, we introduce *improv*, a software platform that allows users to flexibly specify and manage adaptive experiments to integrate data collection, preprocessing, visualization, and user-defined analytics. Using *improv* for streaming data analysis for two photon calcium imaging and behavior we demonstrate how access to online information can be used for automated, integrated experimentation.

Main

Recent developments in biosensors (Dana, 2019) combined with advanced microscopy techniques (Rumyantsev, 2020) have led to dramatic increases in our ability to record from large populations of neurons (Ji, 2016; Song, 2017). Combined with high-resolution behavioral tracking, this brain activity could be linked to behavior (Dombeck 2010; Huber, 2012; Krakauer, 2017; Markowitz, 2018). In particular, advanced methods for optogenetic photostimulation such as computer-generated holography (Pégard, 2017; Chen, 2018; Yang, 2021) promise precise and fast activation or inhibition of genetically or even functionally defined neural ensembles (Marshel, 2019), thereby enabling direct, cellular resolution causal interventions in the brain (dal Maschio, 2017; Zhang, 2018), even in behaving animals (Robinson, 2020). These causal manipulations of neural activity are essential to reveal the functional contributions of individual neurons to circuit computations and behavior (Grosenick, 2015).

Yet most experiments rely on stimulating neurons selected in advance, thereby limiting the possible experimental parameters to those known (or assumed) at the start (Packer, 2015). Moreover, as more data types are recorded (e.g., number of neurons, behavioral variables) and overall data volume increases, our ability to predetermine relevant parameters for stimulation at the beginning on an experiment necessarily decreases. For example, as behaviorally relevant neurons are often widely distributed (Naumann, 2016; Stringer, 2019), their location may not be known in advance. Similarly, the organization of behavioral states (Marques, 2018), which behavioral variables are associated with neural activity, or which neural dynamics are most relevant to behavior (Sani, 2021), must also be learned. Unfortunately, analyses are often performed hours after data acquisition, precluding any interventions that would benefit from information collected during an ongoing experiment (Vladimirov, 2018). This lack of integration between data collection and informative analysis directly impedes effective experimentation. Access to real-time information throughout an experiment will permit rapid early discovery relevant variables and behavioral metrics, enabling new adaptive and closed-loop approaches to circuit dissection in systems neuroscience.

For instance, in experiments that aim to mimic endogenous neural activity via stimulation, researchers must first record from a population of neurons, stop the experiment to analyze the activity (Pégard, 2017; dal Maschio, 2017; Zhang, 2018; Marshel, 2019; Robinson, 2020), determine which neurons are coactive or fit other response criteria, note their locations, and calculate the necessary photostimulation intensity per neuron for future interventions. While pausing acquisition periodically for offline analysis has been a successful approach,

it breaks down with more complex analyses, when fluctuations in system states arise, neurons drift, or when information supplied by online data analysis is required to inform the next stimulation (Bolus, 2020).

Thus, ideally, data analysis should not be independent from data acquisition. Tight computational integration of data acquisition and interpretable analyses will not only speed up analysis, but also provide a way to use incoming data to inform the next step in an experiment. When instantaneous information from the experiment is available, it can produce strategies for actionable interventions (Grosenick, 2015; Zhang, 2018; Vladimirov, 2018). In fact, for large circuits composed of thousands of neurons, establishing fine-grained causal connections between neurons may prove infeasible without methods of narrowing the set of candidate mechanisms or circuit hypotheses on the fly. Therefore, we argue that adaptive experiments, those whose parameters or structure can change in response to incoming data, hold tremendous promise for understanding neural circuits. But adaptive experiments, by definition, require analysis as the data arrive. Although modern computing and new analysis algorithms have made online preprocessing of large-scale recordings feasible (Friedrich, 2016; Giovannucci, 2019; Issar, 2020), significant technical barriers have prevented their integration into routine experimental pipelines. Existing algorithms and software are not constructed to operate under streaming paradigms with many parallel tasks, as is required for live experimentation. To facilitate complex adaptive paradigms requires software that flexibly and transparently handles data sharing, concurrent execution, and pipeline specification, while remaining easily configurable and extensible.

Here we present *improv*, a modular software platform for the construction and orchestration of adaptive experimental designs (**Fig. 1a**). *improv* manages the backend engineering of data flow and task execution for all steps in an experimental pipeline in real time, without requiring user oversight. Users need only define their particular processing pipeline with simple text files and are free to define their own streaming analyses via Python classes, allowing for rapid prototyping of adaptive experiments. Any type of input or output data stream can be defined and integrated into the setup (e.g., behavioral or neural variables), enabling a wide variety of possible analyses and interventions during an ongoing experiment. In addition, *improv* is designed to be highly stable, ensuring data integrity through intensive logging and high fault tolerance. It offers out-of-the box parallelization, visualization, and user-interaction via a lightweight Python application programming interface. The result is a flexible real-time preprocessing and analysis platform that can prototype new experimental designs in only a few lines of code.

improv's design is based on a streamlined version of the 'actor model' of concurrent systems (Hewitt, 1973). In this model, system components, called actors, interact via message passing, without the need for a

central broker (**Fig. 1b**). Data pipelines are defined by processing steps (actors) and message queues, which correspond to nodes and edges, respectively, in a directed acyclic graph (DAG) (**Fig. 1c**). Actors are implemented as user-defined Python classes that inherit from *improv*'s **Actor** class, which defines the queues for message passing and orchestrates process execution and error handling. Messages between actors are composed of keys that correspond to items in a shared, in-memory data store built atop the Plasma library from Apache Arrow (Apache: arrow.apache.org). Thus, communication overhead and data copying between processes is minimized.

As an illustrative case study, we used *improv* to implement online functional connectivity estimation between neurons via two-photon calcium imaging of neural activity in larval zebrafish expressing the genetically encoded calcium indicator in almost all neurons (**Fig. 2**). The processing pipeline acquires fluorescence image frames, preprocesses them using CaImAn (Giovannucci, 2019), analyzes the resulting spike trains to estimate response properties and functional connectivity, and visualizes the result in a graphical user interface (GUI, see **Supplementary Fig. S4**). As an example, we demonstrate that *improv* is capable of analyzing streaming neural activity across the brain of zebrafish expressing the calcium sensor GCaMP6s (Naumann, 2016). Specifically, *improv* continuously identifies active neurons in response to visual motion presented to the zebrafish as sinusoidal gratings projected from below (**Methods**). Our continuous analysis through *improv* results in frame-to-frame updates of estimated neuron locations and their response to each motion direction online, constructing a continuously updated tuning curve for each detected neuron. By identifying each neuron's preferred direction of motion, we produce up-to-the-moment maps of all motion-sensitive neurons in a plane (**Fig. 2a**). For software prototyping, testing and comparison with offline methods, we used prerecorded datasets streamed from disk at the experimental acquisition rate (3.6 Hz); however, we now routinely use *improv* to stream live, raw fluorescence images as they are acquired (**Methods)**.

To estimate functional connectivity between identified neurons online, during the experiment, we fit a linear-nonlinear-Poisson (LNP) model (Pillow, 2008), a widely used statistical model for neural firing. Since the model is convex, convergence to a global optimum is guaranteed over sufficiently long experiments, even fitting online, one datum at a time, using stochastic gradient descent. As a result, we were able to estimate functional connections in addition to the maps of motion sensitivity for each neuron (**Fig. 2a**, green lines). Our online model fit converged quickly towards the value obtained by fitting the model offline, with all data considered at once (**Fig. 2b,c)**. As *improv* provides readily accessible locations of all neurons, future closed-loop experiments could target specific neurons for optical stimulation based on these up-to-the-minute functional characteristics and their hypothesized connections, rather than on conditions specified *a priori*.

Our optimized system proved capable of preprocessing, analyzing, and visualizing results at rates faster than a simulated acquisition frequency of 30 Hz, which is roughly 6.7 MB/s for one test data set (**Fig. 3a**). Moreover, we show that even faster rates than this are possible. Whereas a serial implementation of the analysis exceeds the per cycle time budget of 33ms, *improv*'s inbuilt computational concurrency accomplishes all the same steps well within these time constraints, suggesting that this approach can also scale to volumetric light-sheet microscopy, e.g., for the entire zebrafish brain at 0.8 Hz for 41 planes (roughly 7 MB/s) (Ahrens, 2013). And while short periods of heightened preprocessing time can occur, any lag between acquisition and final analysis quickly dissipates (**Fig. 3b**). In fact, *improv* was able to process data continuously for more than twenty-four hours without crashing or accumulating more than a single frame of processing lag (**Supplementary Fig. S2**). Furthermore, concurrency allows *improv* to easily ignore "bad actors," i.e., missing or slow frames that frequently lag or crash, and maintain the processing the pipeline (**Fig. 3c**). To illustrate system stability, we simulated an imaging experiment in which the 'acquisition' actor drops frames: rather than await missing frames that may never appear (suspending the experiment) or raising errors (halting the experiment), each actor continues its own execution loop, processing any future data when they become available. Importantly, once acquisition resumes, the entire system recovers automatically, without delays or increased processing time.

improv is not the first method to target the analysis of calcium imaging data (Mitani, 2018; Pachitariu, 2017; Scanbox: scanbox.org). As summarized in **Table S1**, multiple options exist for data acquisition, preprocessing, and offline analysis. Yet *improv* is uniquely suited for adaptive experiments, allowing users to rapidly prototype new online analyses, visualizations, and closed-loop designs with a minimum of code. Furthermore, it was explicitly designed to be sufficiently general to accommodate various data streams—not just for calcium fluorescence imaging, but any kind of neural or behavioral measurements. Rather than a cumbersome, complex, integrated, monolithic analysis solution, it adopts the Unix philosophy (McIlroy, 1978) of modular, composable tools. *improv* can also be readily used with a variety of other common tools. For example, Suite2p (Pachitariu, 2017) can be used as a batch preprocessor for groups of images as a drop-in replacement for CaImAn (**Supplementary Fig. S3**) and this functionality can be implemented in just a few lines of code. Similarly, interoperability between languages can be leveraged to implement analysis algorithms in the Julia language (Bezanson, 2012), which offers better performance through just-in-time compilation (**Methods**).

As data collection capacities in neuroscience continue to grow, the key question for experimentalists may no longer be which data they can afford to collect, but which they can afford to ignore. Adaptive experiments offer the promise of time- and statistically efficient data collection, but they pose unique challenges

for both real-time software and algorithms. With *improv*, we are not only able to provide online feedback about the ongoing experimental parameters, but also supply the technical advances necessary to interpret and act based on instantaneous neural activity. Therefore, *improv* opens the door for unprecedented experiments to define the functional connectivity of a neuron by adaptively narrowing down which neurons to target next based on just-updated hypotheses as data collection continues without interruptions. We expect *improv* to facilitate rapid prototyping of any number of such adaptive experiments, providing both a set of tools and a research platform for the integration of data collection with analysis and intervention.

Acknowledgements

We would like to thank Eric Thomson for helpful comments and discussions, and the Duke Innovation Co-Lab for technical support. We are grateful to Catherine Seitz, Jim Burris, and Karina Olivera for zebrafish husbandry. We thank Misha Ahrens for generously sharing transgenic zebrafish lines.

A.D., M.N., C.S., D.S., M.D.L., E.A.N., and J.M.P. were supported by the BRAIN Initiative R34-EB026951-01A1 and a Duke Institute for Brain Sciences (DIBS) incubator award. A.D. was supported by a Ruth K. Broad Biomedical Research Foundation Postdoctoral Fellowship Award and a Swartz Foundation Postdoctoral Fellowship for Theory in Neuroscience. A.G. was supported by the Beckman Young Investigator Program. E.A.N. and M.D.L. were also supported by the Whitehall Foundation.

Author contributions

E.A.N. and J.M.P. conceived of this project. A.D. and J.M.P. designed and implemented the *improv* framework. A.D., C.S., and D.S. wrote software components for the various use cases and interoperability with other programs. E.P. and A.G. contributed to calcium imaging analysis software for online use. M.N. and M.D.L. built the two-photon microscope and associated visual stimuli software and performed the imaging experiments. A.D., E.A.N., and J.M.P. wrote the manuscript with input from all authors.

Competing interests

The authors declare no competing financial interests.

Software availability

This software package can be downloaded from github.com/pearsonlab/improv or through the Python Package Index pypi.org/project/improv.

Data availability

All data will be made available upon reasonable request.

Figures and legends

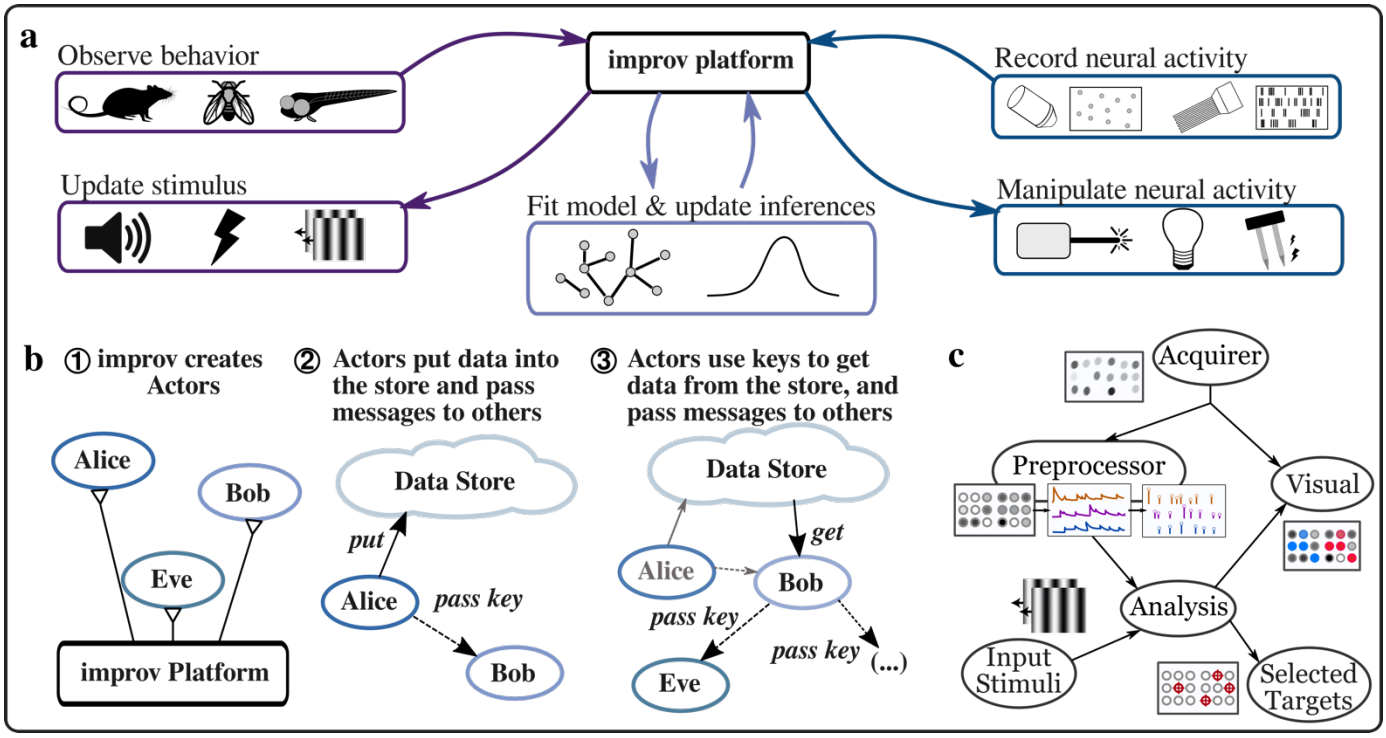

Fig. 1 | Design architecture of *improv.*

a, Schematic for the many use cases of *improv*. Input and output data streams are managed independently and asynchronously, for all kinds of behavioral, neural, or modeling data.

b, Schematic for the actor model. (1) *improv* creates and manages actors, separate concurrent processes. (2) Actors can access a shared data store and pass messages to other actors. (3) Actors communicate only by passing addresses of data items, minimizing data copies.

c, Example actor graph, a two-photon calcium imaging pipeline that acquires images, preprocesses them online with CaImAn, analyzes the resultant neural activity based on input stimuli, and visualizes the result. Nodes (ovals) are actors corresponding to steps in the processing pipeline; arrows indicate logical dependencies between actors.

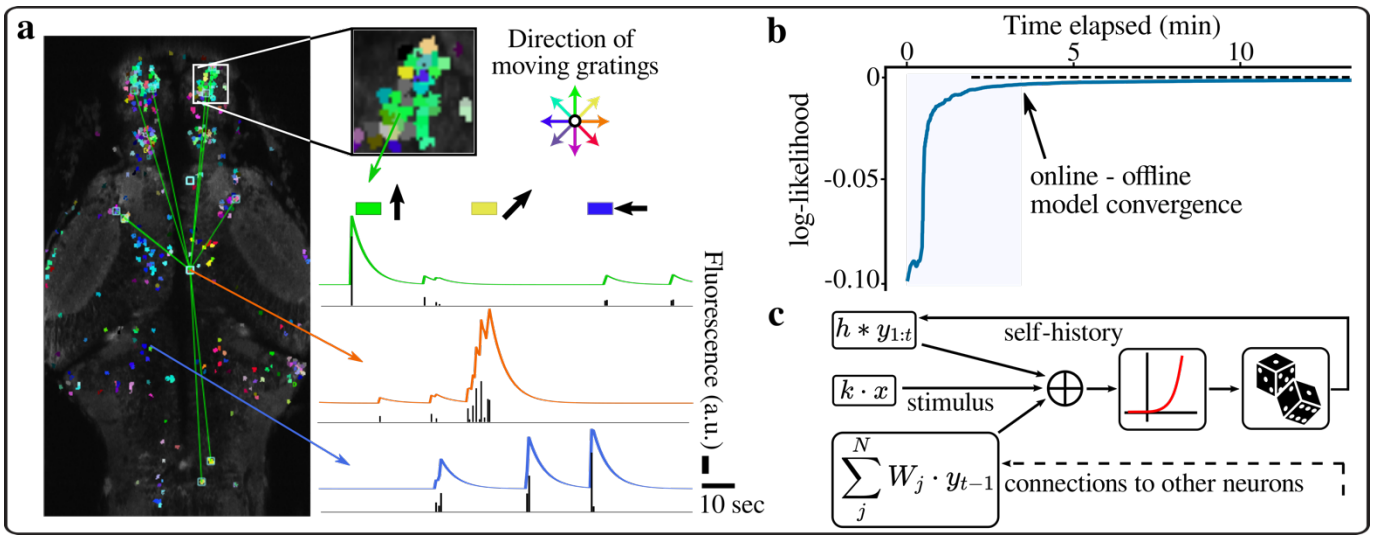

Fig. 2 | Use case of *improv* for online neural activity analysis.

a, Left, an average projection of a two-photon time series of a single plane across the zebrafish brain (grey). Superimposed motion direction tuning map was calculated with custom analysis code. Each shape represents motion-sensitive neurons discovered by online analysis with color indicating each unit's preferred motion direction (inset, see color wheel). Green lines show inferred functional connections from the selected unit (white box) to other neurons obtained from the model fit. Right, average fluorescence intensity for 3 example neurons with calculated spike frequency (black).

b, Log-likelihood of the model fit over time, as more frames are fed into *improv*. The online model fit converges to the offline prediction after just a few hundred frames. The shaded region corresponds to the first complete set of stimuli, after which the stimuli are repeated.

c, Diagram of our model incorporating stimuli, self-history, and weighted connection terms.

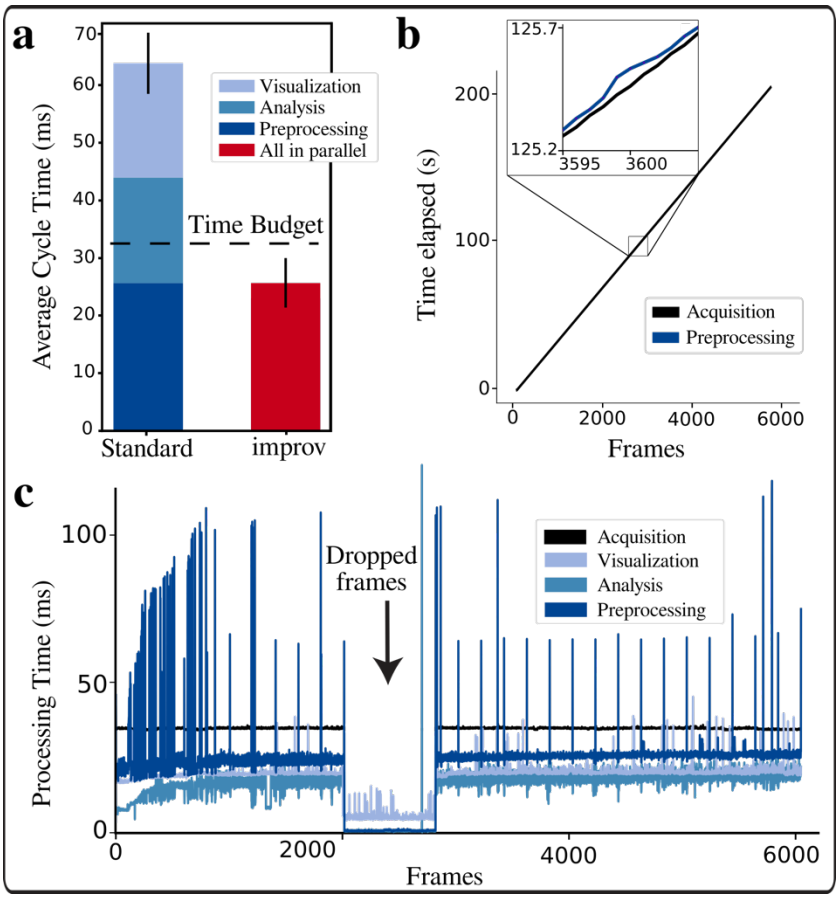

Fig. 3 | Performance & benchmarking of *improv*.

a, *improv* parallelizes operations wherever possible. Real-time fluorescence image preprocessing, analysis, and visualization can be performed by *improv* in a total cycle time well below the time budget of 33 ms for data acquisition at 30 Hz. Executing each process in a standard offline processing sequence exceeds this benchmark, rendering online analysis impossible. Error bars show standard deviation for total average cycle times.

b, Timestamps for acquisition and preprocessing show stable processing time per frame. While some frames take longer than others to process (inset) the difference in data acquired and data processed does not accumulate.

c, Processing time per frame does not appreciably increase throughout the experiment. Even if frames are dropped or corrupted, actors can resume processing when data acquisition recovers.

Methods and Supplemental Information

A. Improv configuration

Configuration in *improv* is streamlined and simple, requiring only that a user define (1) what processing steps (i.e., actors) are part of the pipeline, and (2) in what order they should be executed. The pipeline definition is lightweight: DAGs are specified by text-based configuration files with two sections (**Supplementary Fig. S1**): In the first section, actors are specified by the names of their respective Python classes, with any additional information passed as keyword arguments to class constructors. In the second section, connections among actors are specified by listing the consumers for each actor's outputs. On startup, *improv* handles construction for each of these queues and links them with their respective actors. *improv* can handle both single and multiple output queues.

Importantly, all instantiation, memory-sharing, execution flow, and logging are handled by *improv*, leaving the user free to focus on code within an actor. Furthermore, any Python code (or code executable in a Python environment) is acceptable within an actor: we do not supply a library of functions or require that a user follow particular implementation. Thus, almost anything a user wants to compute or run within an actor is available, including integration with other software tools (see below, section G).

Figure S1b illustrates this for a simple **MeanAnalysis** class. Here, users define the **run** method, which receives a data store ID, called the **key**, from the input queue, uses it to retrieve the datum (in this example, an array, called **estimates,** of neural activity estimates), computes single-neuron and population activity averages, and places the results back into the data store. The keys to identify those results are then published in the supplied output queue **self.q_out**. In our example pipeline, these are retrieved for display by the visual actor (the graphical user interface; see section I).

improv only specifies the semantics of data pipelines, freeing users from the need to manage implementation details. This is facilitated by two key components: a shared, in-memory data store, discussed above, and a central controller program, dubbed *Nexus*. Once users have defined each processing step in the pipeline and the dependency relationships between them, *Nexus* is responsible for the actual orchestration of experiments. At runtime, *Nexus* instantiates each actor, configures information and execution flow, and handles both error signaling and user interactions. On startup, it takes in a list of desired actors and associated attributes from the configuration file, instantiates each class, and provides it access to the shared data store. *improv*'s user

contract is thus both extensible and simple: users write custom Python classes for any new analyses they require, but they do not need to know the internals of classes implementing other pipeline steps.

Once the experimental pipeline is started, *Nexus* executes each class in a separate process and monitors its progress. Each process corresponding to an actor is kept alive continuously, waking as data become available in its input queue. In this way, the system leverages concurrency to overcome delays due to serial processing or input/output overhead. Communication is handled asynchronously, using a custom class combining Python's **asyncio** and **multiprocessing** libraries. As a result, the entire pipeline is robust to failures: no one actor or internal task can suspend the system, which would effectively cause the experiment to terminate. In addition, each object placed into the data store and every parameter change can be logged to disk, effectively creating a snapshot of the system state at each moment in the experiment. This ensures a robust audit trail, such that any data or associated analysis can be reproduced by later offline analyses.

B. Zebrafish

For all experiments, we used 6 days post-fertilization (dpf) zebrafish in a *nacre* -/- (also known as mitfa-/-) background; *nacre* -/- mutants lack dark skin pigmentation but retain wild-type eye pigmentation, vastly increasing imaging quality across the brain while maintaining normal visual functioning. Adult zebrafish were maintained on a 14 hrs. light /10 hrs. dark cycle and fertilized eggs were collected and raised at 28.5 °C. Embryos were kept in E3 solution (5 mM NaCl, 0.17 mM KCl, 0.33 mM CaCl2, 0.33 mM MgSO4). All experiments were approved by Duke University's standing committee on the use of animals in research and training. All new imaging experiments in this study were performed on transgenic zebrafish *Tg*(*elavl3:GCaMP6s*) a generous gifts from Dr. Misha Ahrens (Vladimirov, 2014).

C. Two-photon imaging

In vivo two-photon fluorescence imaging was performed using a custom-built two-photon laser-scanning microscope, equipped with a pulsed Ti-sapphire laser tuned to 950 nm (Spectra Physics, USA). In order to minimize movement artifacts, larvae were embedded in low melting point agarose (2% w/v). However, their tails were freed to allow for observation of behavioral responses. In addition, viability monitoring was supplemented by observing the heartbeat and blood flow through brain vasculature before and after imaging. All data acquisition were performed using custom Labview and Python codes (National Instruments, USA).

Typically, the images were obtained in raster scanning mode with 512x512 pixels scanned at 400 kHz, and frames were acquired at 3.6 Hz. Visual stimuli were projected from below with the red LED of a DLP projector (AAXA Technologies P300), which allowed for simultaneous visual stimulation and detection of green fluorescence. Full contrast square wave gratings drifting in 8 directions moving at 1 mm/cm for 8.5 seconds were interleaved in duration. Details for previously acquired data shown in **Fig. 2** can be found in (Naumann, 2016).

The integration of *improv* into this experimental setup required only a method for transferring streaming data from the acquisition system to our analysis system in real time. We ran *improv* on a separate computer, but in general a separate computer is not required, and analysis can be done on the same machine. While many networking solutions exist, we chose to use ZeroMQ, a widely used universal messaging library that has interfaces in many languages, including Python and LabVIEW. Images obtained in LabVIEW were thus directly streamed via ZeroMQ to our **Acquirer** actor in Python. Similarly, messages and timestamps for the visual stimulus being displayed were also streamed directly from the Python script running the visual stimuli to the same Acquirer actor. And with *improv's* flexibility, should we want to run a simulated experiment using data streamed from disk (rather than live via ZeroMQ), only the **Acquirer** actor would need to be changed, leaving the rest of the pipeline intact.

D. Two-photon calcium fluorescence analysis

Spatial and temporal traces were extracted from the calcium images using the CaImAn Online algorithm within CaImAn (Giovannucci, 2019), modified with custom code to allow for single frame-by-frame processing of incoming data streams without prespecifying the data length. Parameters supplied to CaImAn Online for motion correction and source extraction are specified in a configuration file to the preprocessing actor and are available in our codebase. Our analyses utilized both the estimated fluorescence traces as well as the extracted spike counts for subsequent model fitting. Extracted spatial traces were used for visualization in the user interface (**Supplementary Fig. S4**) with openCV to fill in and color each neuron by its responses to visual stimuli.

To compute each neuron's responses to visual stimuli, a running mean was kept for the responses to each of the 8 directions of grating drift. A window of 10 frames prior to onset of each stimulus was used for baseline subtraction, and a window of 15 frames was used to compute an average response after the onset of each stimulus. These average responses were then used with a custom coloring scheme to visually label directional tuning in the graphical user interface (**Fig. 2a**, see arrow wheel for color code).

E. LNP model fitting

We used a form of the well-known linear-nonlinear-Poisson model to estimate functional connections among all observed neurons. Our generalized linear model included terms for: (1) the stimulus responses, simply modeled as a vector of length 8 corresponding to the current stimuli, without using any spike-triggered averages (STA); (2) the self-history effect, modeled using a history vector of activity up to 4 frames prior to the current frame; and (3) weighted functional connections to all other neurons, modeled using the prior frame activity information. An exponential nonlinear term was used to compute firing rates.

To fit this model online, we used stochastic gradient descent with windows of data ranging from 10 – 100 frames of prior data held in memory. Step sizes were chosen based on data, but in general a value of 1e-5 was used successfully. For visualization purposes, each neuron's top 10 connections (determined by magnitude) were sorted and displayed in the graphical user interface as both a matrix (right) and green lines (center, if a neuron is selected).

F. Benchmarking

Our example implementation has been tested on three main computers: (1) a 2015 Macbook laptop with 8 GB of RAM and a 2 core 2.2 GHz Intel i7 processor; (2) a 2018 custom-built (total cost < $4k) Ubuntu desktop machine with 128 GB of RAM and a 14 core 3.1 GHz Intel i9 processor; and (3) a 2019 custom-built (< $4k) Windows desktop machine with the same specifications as (2). All benchmarking data presented in the main text were taken from (2). The software has been confirmed to be operational with all major operating systems (Windows 10, Ubuntu, Mac OS X).

G. Interoperability with other tools

While *improv* represents the only fully extensible tool dedicated in the online setting, there are a host of excellent tools available for offline analysis, as well as two that offer some online functionality (**Table S1**). As a first step toward integration with these tools, we have implemented proof-of-concept interfaces between *improv* and both Suite2p (via its Python command-line interface) and ScanBox (via the MATLAB execution environment) (**Supplementary Fig. S2**).

Other pipeline or workflow generating software packages have also tackled this problem of efficient pipelining, even specifically for the neuroscience community (Gorgolewski, 2011). Our system by contrast does not attempt to directly bundle collections of applications for a multi-use toolbox, but instead is a lightweight scaffolding approach that provides containers (actors) to accommodate most any application the user needs. This also ensures our code does not fall into maintenance traps, but rather easily enables updates or adding new functionality.

H. Julia computations

Both Python and Julia have packages designed for interfacing with one another. Using the Python package PyJulia, we can compile and execute a piece of Julia code from within a Python program. Combining this with the Julia package PyCall to do the reverse, we can flexibly transfer data from *improv* to Julia for quick analysis (e.g., gradient descent of an LNP model) and transfer the results back into *improv* once again. Notably, this transfer between languages can be accomplished with time-efficient no-copy wrappers if using NumPy arrays. For an example implementation using Julia see the **julia** branch at github.com/pearsonlab/improv.

I. Graphical interface

For our specific experimental integration of *improv* with a two-photon calcium imaging setup, we also constructed a simple graphical user interface (GUI) to provide user control and real-time updates of all images and analyses (**Supplementary Fig. S4**). For the paradigm described in the main text, we plot fluorescence or extracted spikes as a function of the last 500 frames (scrolling window) for both the population average and a user-selected neuron. Response profiles are plotting as circular tuning curves adjacent to those line plots. Below, raw images acquired from the setup are shown to the left, and the processed and analyzed frame is shown to the right. Neurons in the processed frame are colored by their tuned responses and can be selected via mouse click by a user to display its data above. On the right side we display the online results of the LNP model fit. The top plot shows the negative log-likelihood function being minimized as more frames are analyzed, and the bottom plot displays the inferred weight matrix of connections among the top 10 most connected neurons. Again, by selecting a neuron in the center processed frame plot, the connections associated with that neuron are displayed as green lines.

Supplemental Figures and legends

a Configuration File

```
Modules:

  Acquirer:
    class: TwoPAcquirer
    camera: /dev/USB0

  Processor:
    class: CaimanProcessor
    config: caiman_config.txt

  Visual:
    class: MeanAnalysis

  (...)

Connections:

  Acquirer.q_out:  [Visual.q_in,
                    Processor.q_in]
  Processor.q_out: Analysis.q_in
  Analysis.q_out:  [Visual.q_analysis,
                    OutputStim.q_in]
  InputStim.q_out:  Analysis.q_stim
```

b Class Definition File

```
import numpy as np
from nexus.actor import Actor

class MeanAnalysis(Actor):

def __init__(self):
    (...)

  def run(self):
    # Get key for data
    key = self.q_in.get()
    # Get data from the store
    estimates = self.client.getID(key)

    # Compute mean activity
    avg = self.calcAvg(estimates)
    global = np.array(np.mean(avg))

    # Put result into the store
    retKeys.append(self.client.put(avg))
    retKeys.append(self.client.put(global))
    # Publish keys
    self.q_out.put(retKeys)
```

Fig. S1 | Code examples

a, Example configuration file defining a set of actors and a set of connections. Actors are defined by Python classes. Connections form a directed, acyclic graph, specified by listing the children of each actor node.

b, Example class file defining an actor whose operation computes mean activity. The run method fetches data from the store based on keys passed from other actors; processes these data using custom Python code; and places the results into the data store.

Software	Acquire data	Process data	Analyze data	Visualize (GUI)	Extend (user code)	Online algorithm	Open source
Suite2p	✗	✓	✓	✓	✗	✗	✓
ScanImage	✓	✓	✓	✓	✗	~✓	✗
CaImAn	✗	✓	✗	✓	✓	✓	✓
Scanbox	✓	✓	✓	✓	✗	✗	✗
Thunder	✗	✓	✓	✗	✓	✗	✓
SIMA	✗	✓	✗	✓	✓	✗	✓

Table S1 | Comparisons to other tools

Comparison of software platforms for calcium image processing. Only CaImAn offers end-to-end online preprocessing, and it does no subsequent online analysis. Our platform *improv* fills this gap and offers easy extensibility and compatibility with many other tools.

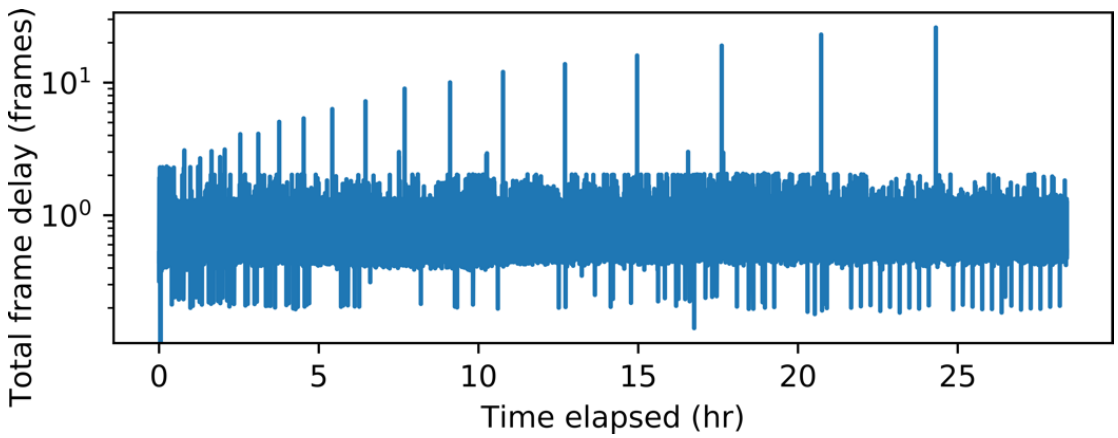

Fig. S2 | Frame lag over 24 hours.

Total frame lag, the time between processing one frame and acquiring the next frame, shown over more than 24 hours running continuously. On average, there is less than 1 frame of total lag even after running for more than a day, demonstrating the robustness and feasibility of streaming calcium data analysis.

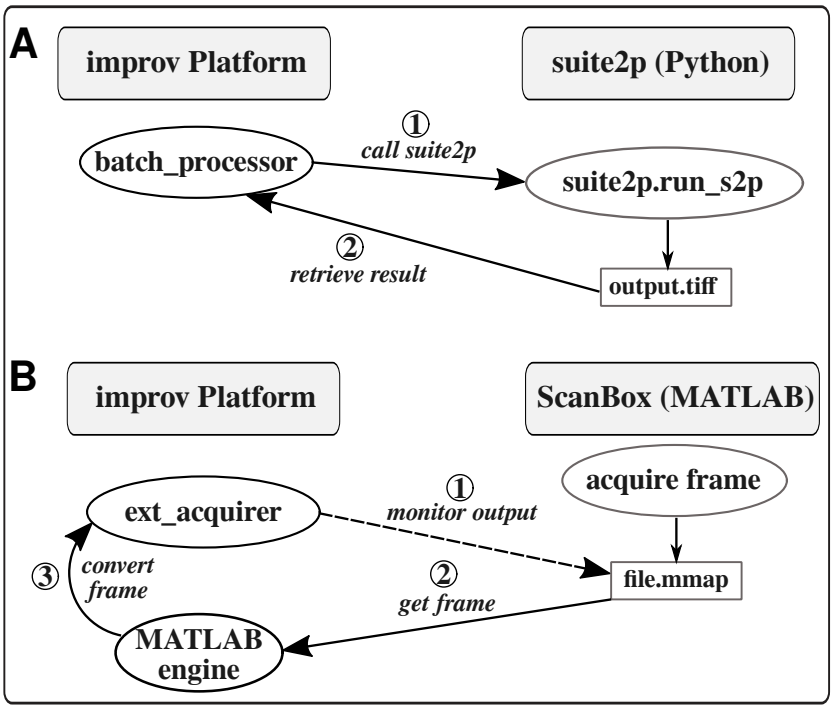

Fig. S3 | Integration with other tools.

a, *improv* can use suite2p for analysis by using our own batch_processor actor that calls suite2p via its Python interface. Outputs from suite2p are saved to disk and read back into *improv's* data store for further processing.

b, *improv* can be used to analyze data acquired by ScanBox via our ext_acquirer actor to monitor the memory-mapped (mmap) file used by ScanBox to write data to disk. When needed, it can access these data by calling MATLAB from Python via the MATLAB engine.

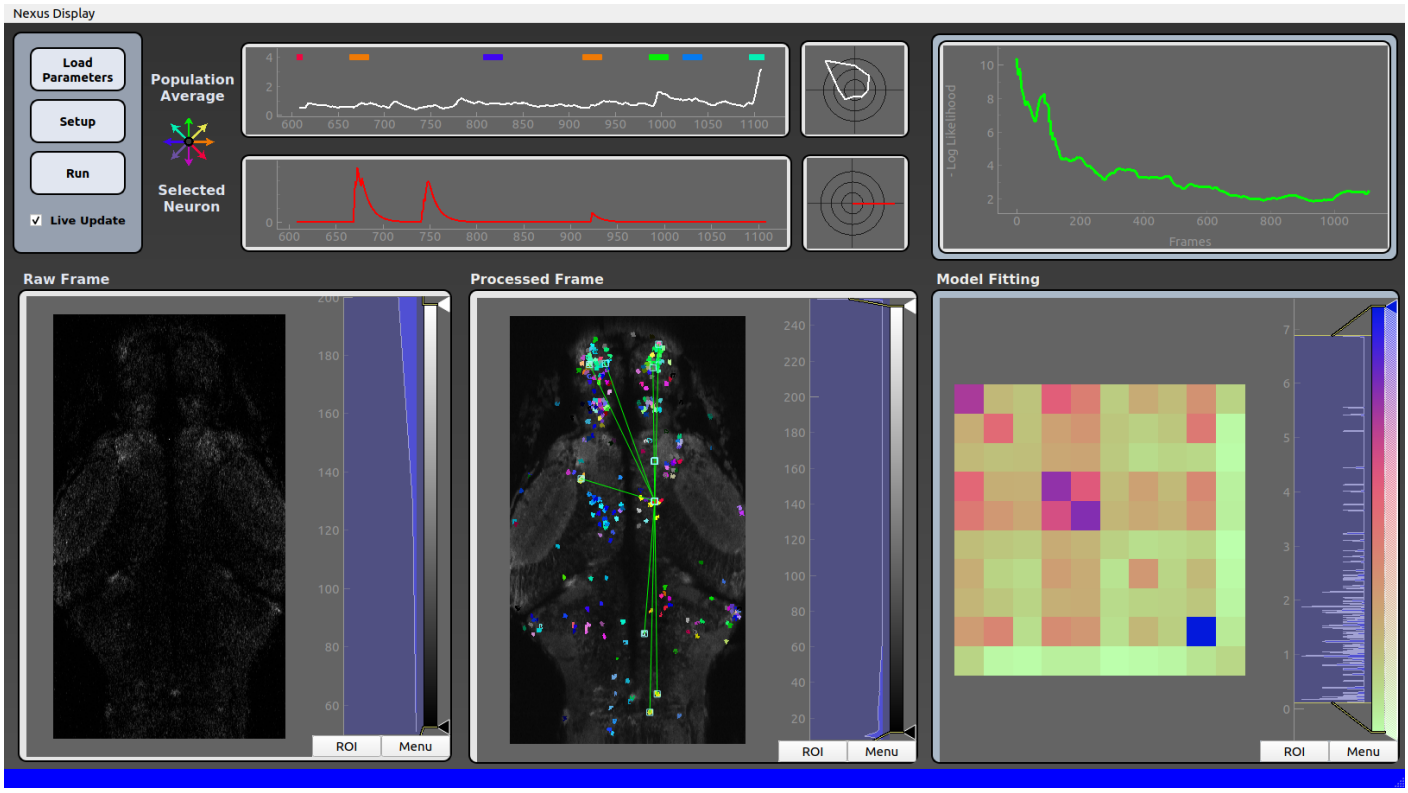

Fig. S4 | Example Graphical User Interface.

Top, left controls serve to configure, run, or pause the experiment. Extracted neural activity or spike trains for both the population average and for user-selected neurons are displayed in real time alongside their polar tuning curves.

Bottom, raw and processed images are displayed in real time. Processed images include colored neurons based on their directional tuning curves, and each identified neuron can be selected (via mouse click) to display their neural activity and tuning curve above, and green lines show their inferred functional connections.

Right, the model fit (negative log-likelihood) and weight matrix are displayed in real time. The top 10 neurons with the most connections (rows) and their top connections (columns) are dynamically ordered for visualization in the matrix, colored by absolute magnitude of the strength of the connection.

References

Ahrens, Misha B., Michael B. Orger, Drew N. Robson, Jennifer M. Li, and Philipp J. Keller. "Whole-brain functional imaging at cellular resolution using light-sheet microscopy." *Nature methods* 10, no. 5 (2013): 413-420.

Apache Arrow, arrow.apache.org.

Bezanson, Jeff, Stefan Karpinski, Viral B. Shah, and Alan Edelman. "Julia: A fast dynamic language for technical computing." *arXiv* 1209.5145 (2012).

Bolus, Michael F., Adam A. Willats, Christopher J. Rozell, and Garrett B. Stanley. "State-space optimal feedback control of optogenetically driven neural activity." *Journal of Neural Engineering* (2020).

dal Maschio, Marco, Joseph C. Donovan, Thomas O. Helmbrecht, and Herwig Baier. "Linking neurons to network function and behavior by two-photon holographic optogenetics and volumetric imaging." *Neuron* 94, no. 4 (2017): 774-789.

Deisseroth, Karl, and Mark J. Schnitzer. "Engineering approaches to illuminating brain structure and dynamics." *Neuron* 80, no. 3 (2013): 568-577.

Friedrich, Johannes, and Liam Paninski. "Fast active set methods for online spike inference from calcium imaging." In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 1992-2000. 2016.

Giovannucci, Andrea, Johannes Friedrich, Pat Gunn, Jeremie Kalfon, Brandon L. Brown, Sue Ann Koay, Jiannis Taxidis et al. "CaImAn an open source tool for scalable calcium imaging data analysis." *eLife* 8 (2019): e38173.

Gorgolewski, Krzysztof, Christopher D. Burns, Cindee Madison, Dav Clark, Yaroslav O. Halchenko, Michael L. Waskom, and Satrajit S. Ghosh. "Nipype: a flexible, lightweight and extensible neuroimaging data processing framework in python." *Frontiers in neuroinformatics* 5 (2011): 13.

Grosenick, Logan, James H. Marshel, and Karl Deisseroth. "Closed-loop and activity-guided optogenetic control." *Neuron* 86, no. 1 (2015): 106-139.

Hewitt, Carl, Peter Bishop, and Richard Steiger. "A Universal Modular ACTOR Formalism for Artificial Intelligence" *IJCAI* (1973): 235-245.

Issar, Deepa, Ryan C. Williamson, Sanjeev B. Khanna, and Matthew A. Smith. "A neural network for online spike classification that improves decoding accuracy." *Journal of Neurophysiology* 123, no. 4 (2020): 1472-1485.

Kane, Gary A., Gonçalo Lopes, Jonny L. Saunders, Alexander Mathis, and Mackenzie W. Mathis. "Real-time, low-latency closed-loop feedback using markerless posture tracking." *eLife* 9 (2020): e61909.

Krakauer, John W., Asif A. Ghazanfar, Alex Gomez-Marin, Malcolm A. MacIver, and David Poeppel. "Neuroscience needs behavior: correcting a reductionist bias." *Neuron* 93, no. 3 (2017): 480-490.

Lu, Rongwen, Wenzhi Sun, Yajie Liang, Aaron Kerlin, Jens Bierfeld, Johannes D. Seelig, Daniel E. Wilson et al. "Video-rate volumetric functional imaging of the brain at synaptic resolution." *Nature neuroscience* 20, no. 4 (2017): 620-628.

Markowitz, Jeffrey E., Winthrop F. Gillis, Celia C. Beron, Shay Q. Neufeld, Keiramarie Robertson, Neha D. Bhagat, Ralph E. Peterson et al. "The striatum organizes 3D behavior via moment-to-moment action selection." *Cell* 174, no. 1 (2018): 44-58.

Marques, João C., Simone Lackner, Rita Félix, and Michael B. Orger. "Structure of the zebrafish locomotor repertoire revealed with unsupervised behavioral clustering." Current Biology 28, no. 2 (2018): 181-195.

Marshel, James H., Yoon Seok Kim, Timothy A. Machado, Sean Quirin, Brandon Benson, Jonathan Kadmon, Cephra Raja et al. "Cortical layer–specific critical dynamics triggering perception." *Science* 365, no. 6453 (2019): eaaw5202.

McIlroy, Malcolm D., Elliot N. Pinson, and Berkley A. Tague. "UNIX Time-Sharing System: Foreword." *Bell System Technical Journal* 57, no. 6 (1978): 1899-1904.

Mitani, Akinori, and Takaki Komiyama. "Real-time processing of two-photon calcium imaging data including lateral motion artifact correction." *Frontiers in Neuroinformatics* 12 (2018): 98.

Naumann, Eva A., James E. Fitzgerald, Timothy W. Dunn, Jason Rihel, Haim Sompolinsky, and Florian Engert. "From whole-brain data to functional circuit models: the zebrafish optomotor response." *Cell* 167, no. 4 (2016): 947-960.

Pachitariu, Marius, Carsen Stringer, Mario Dipoppa, Sylvia Schröder, L. Federico Rossi, Henry Dalgleish, Matteo Carandini, and Kenneth D. Harris. "Suite2p: beyond 10,000 neurons with standard two-photon microscopy." *Biorxiv* (2017).

Packer, Adam M., Lloyd E. Russell, Henry WP Dalgleish, and Michael Häusser. "Simultaneous all-optical manipulation and recording of neural circuit activity with cellular resolution in vivo." *Nature methods* 12, no. 2 (2015): 140-146.

Pégard, Nicolas C., Alan R. Mardinly, Ian Antón Oldenburg, Savitha Sridharan, Laura Waller, and Hillel Adesnik. "Three-dimensional scanless holographic optogenetics with temporal focusing (3D-SHOT)." *Nature communications* 8, no. 1 (2017): 1-14.

Pereira, Talmo D., Diego E. Aldarondo, Lindsay Willmore, Mikhail Kislin, Samuel S-H. Wang, Mala Murthy, and Joshua W. Shaevitz. "Fast animal pose estimation using deep neural networks." *Nature methods* 16, no. 1 (2019): 117-125.

Pillow, Jonathan W., Jonathon Shlens, Liam Paninski, Alexander Sher, Alan M. Litke, E. J. Chichilnisky, and Eero P. Simoncelli. "Spatio-temporal correlations and visual signalling in a complete neuronal population." *Nature* 454, no. 7207 (2008): 995-999.

Sani, Omid G., Hamidreza Abbaspourazad, Yan T. Wong, Bijan Pesaran, and Maryam M. Shanechi. "Modeling behaviorally relevant neural dynamics enabled by preferential subspace identification", *Nature Neuroscience* 24, (2021): 140-149.

Scanbox, Los Angeles, CA, scanbox.org.

Vladimirov, Nikita, Chen Wang, Burkhard Höckendorf, Avinash Pujala, Masashi Tanimoto, Yu Mu, Chao-Tsung Yang et al. "Brain-wide circuit interrogation at the cellular level guided by online analysis of neuronal function." *Nature methods* 15, no. 12 (2018): 1117-1125.

Zhang, Zihui, Lloyd E. Russell, Adam M. Packer, Oliver M. Gauld, and Michael Häusser. "Closed-loop all-optical interrogation of neural circuits in vivo." *Nature methods* 15, no. 12 (2018): 1037-1040.