A low-cost, modular, environment for imaging calcium in neurons and collecting simultaneous motor information

Introduction

Calcium imaging is a burgeoning technique used for imaging and assessing the collective activity of hundreds of neurons simultaneously. A new challenge is the development of techniques that allow for concomitant execution of different tasks and experimental paradigms along with calcium imaging. Of recent interest include examining the motor output of mice while imaging from relevant parts of the striatum (Barbera et al., 2016 and Klaus et al. 2017), and imaging the hippocampus during operant conditioning.

An ideal experimental setup requires several components. First and most importantly, it must have high temporal fidelity. Calcium imaging has strict temporal requirements due to the fact that GCaMP must first be exposed to an LED for a fixed amount of time before an image is captured. A common digital design is to set up a CMOS or other imaging device to capture a frame every time it receives a digital pulse from the device responsible for organizing and synchronizing the experiment. Therefore, substantial jitter in digital pulse delivery can cause potentially substantial frame loss. That is, if the camera has not finished with the previous imaging cycle, it could skip a frame if the next pulse signal occurs too early. Also, in order to train a mouse to respond to a conditioned stimulus, repetition of stimulus and response must occur in a highly regular temporal fashion.

Secondly, the experimental setup must be easy to manipulate or alter. Technical skillsets vary widely in the field of neuroscience, and to be adapted widely experimental designs must accommodate these widely varying backgrounds. It is infeasible and inefficient to rely on a technician every time one must subtly tweak or disturb an experimental paradigm. Ideally, the experimental setup would enable a user to quickly translate or implement an idea they have in mind, and be simple enough to encourage the user to build novel experimental designs instead of conforming to preexisting designs. Experimental setups should accelerate and not impede the pace of research and discovery.

Finally, the experimental setup should be both widely accessible and open-source. These requirements have several sub-components that go hand-in-hand. First, it should be affordable, for reasons that are obvious. Current environments and programming environments can be exceedingly expensive [GIVE EXAMPLES HERE]. The Teensy 3.2 itself costs only $19.80 (<https://www.pjrc.com/store/teensy32.html>). The most expensive experimental component that we use in our setup is the ADNS-9800 sensor, which costs only $27.50. (<https://www.tindie.com/products/jkicklighter/adns-9800-laser-motion-sensor/>). The Arduino and Teensyduino programming environments are free, with the option of leaving a donation for continued development. Wide accessibility is necessary to maximize the effect of an open-source environment. Even if money is not an object to academic audiences, the lower the cost of an item, the more readily hobbyists will adopt the product. As they do more and more, we will see the development of new open source libraries accelerate. Cost can be prohibitive without grant money; therefore, if an open-source programming or design environment existed but were expensive, this would preclude wide-spread contribution of new software libraries or hardware components to the existing system by pricing out hobbyists. For example, in our implementation of a motion-sensing calcium imaging paradigm, we utilize the ADNS-9800 sensor, which is produced by a small company (Jack Enterprises, LLC) in Cookeville, Tennessee. This sensor affords us easy and affordable access to a high-speed, high-fidelity gaming sensor. Open-source products potentially offer faster, highly parallel development by taking advantage of the global village.

Here, we introduce two specific implementations of calcium imaging experimental designs, implemented via a Teensy 3.2 microcontroller in conjunction with several simple code scripts, and thereby demonstrate the ease and usefulness of adopting such a design for future experiments.

**Methods**

There are a number of ways in which people have attempted to observe motor output while imaging from the striatum. In one particular technique, experimenters mount a fluorescence microscope on the head of a mouse, and allow the mouse to move freely while recording activity via video (Barbera et al. 2016) or via video in addition to an accelerometer (Klaus et al. 2017). However, resting a microscope, no matter how light, on the head of a mouse restricts its normal range of movement for the mouse, limiting its peak velocity and introducing a confound variable to the experiment. For example, bearing additional weight recruits more muscle fibers and potentially supportive architecture, which could blur distinctions between neural representations of high and low motor patterns, particularly in motion-related regions of the brain such as the striatum.

Another technique utilizes a “three-dimensional treadmill” setup, initially proposed by Dombeck et al. (2007) and utilized widely elsewhere (Aronov and Tank, 2014; Gritton et al. (2018) (in review). In this setting, the mouse is fitted with a head plate and imaging window, and is suspended atop a Styrofoam ball that is supported by compressed air (Figure 1). This type of imaging offers small image jitter primarily in-plane, which is advantageous because it can easily be corrected by standard cross-correlation-derived motion-correction methods. It also offers a setting in which mouse must apply similar forces to begin or to terminate a motor sequence as it would in a freely-moving setting (Dombeck et al. (2007). Therefore, the mouse able to move at normal velocities. Generally, two computer mice are fit at the equator of the styrofoam ball at an angle of 90 degrees, which provides the experimenter with linear movement in the X-Y plane, as well as rotational information. Most of these techniques utilize LabView to obtain voltage readings from the computer mice (Dombeck et al., 2007, Aronov and Tank, 2014), which, though a comprehensive piece of software, is expensive proprietary. In our own lab, implementing high-level MATLAB implementations of TTL pulse-based data acquisition using a National Instruments data acquisition board in conjunction with ViRMEN software led to temporal delays. As described above, we needed a platform that was low-cost, scalable, and had high temporal fidelity.

Here we introduce a system for simultaneous wide-field calcium imaging and simultaneous motion three-dimensional treadmill tracking that necessitates only an Teensy 3.2 microcontroller (~$20.00), and two ADNS-9800 laser motion sensors (~$27.00x2) (<https://www.tindie.com/products/jkicklighter/adns-9800-laser-motion-sensor/>). This system offers an affordable, modular, open-source method of tracking mouse movement with high fidelity, temporal accuracy and without introducing confounding motor variables.

**Methods**

*Brief overview of design architecture*

We first establish the need for a modular, low-cost and open-source software platform. A strength of neuroscience is that it is a “lowest common denominator” field: practitioners enter with backgrounds as diverse as electrical engineering, biology, and even non-scientific domains, affording it a multifaceted approach to addressing questions in neuroscience. A downside to this is that certain experimental designs might be prohibitively complicated for those coming from specific fields. In an effort to accommodate these varied skillsets, the best design for a novel data acquisition device should be modular, and allow those coming from as many different backgrounds as possible access to experimental designs. Here, we offer two approaches for recording motion data synchronously with calcium imaging data: one via a Raspberry Pi 3B and two computer mice, and one via a Teensy 3.2 microcontroller and two ADNS-9800 sensor modules. The former approach necessitates very little familiarity with electronic devices, and some minor familiarity with a Unix-based operating system. The second approach requires slightly more comfort with electronics and is marginally more expensive, but is capable of providing very high accuracy (8200 counts per inch and up to 12,000 frames per second) motion readings.

In order to control these devices, the user interfaces primarily with a MATLAB [AND PYTHON-> add this!] graphical user interface on a desktop or laptop, where they enter the length of the experiment and the frequency of data acquisition. This frequency will determine the frequency with which TTL pulses are sent to notify the CMOS camera to capture a TIFF image, and also the frequency with which accumulated motor information will be recorded by this PC.

The user’s options entered via the graphical user interface are then sent to an Arduino (in our case, an Arduino Uno, though in principle any type of Arduino-like microcontroller could be adopted to our design), via a universal serial bus (USB) which keeps a master clock and synchronizes motion data acquisition and TTL pulses. Using the serial input from the desktop, the Arduino initializes the experiment, generates a number of loops with a user-defined frequency and, and every loop will send a TTL-pulse (encoded as a digital “1”) to a CMOS camera recording calcium fluorescence and generate an interrupt on the motor data acquisition device. This lets either the Raspberry Pi or Teensy know that it needs to send accumulated motor information to the desktop via a serial connection. The Raspberry Pi utilizes a UART to USB connector, and the Teensy utilizes a simple microUSB-to-USB connector. This motor information is stored in a file with X and Y readings from both sensors, along with a corresponding time stamp. Each TTL pulse notifies the CMOS camera to capture an image and store it on the computer as a TIFF image, each of which corresponds to a unique motor data point. While not sending motor data to the original PC, each motor acquisition device is programmed to asynchronously capture and accumulate motor information from its sensors.

*Raspberry Pi*

A Raspberry Pi is a very inexpensive (~$35) computer that has a 1.4 gigahertz quad-core processor, 4 USB ports, and HDMI port, and 1GB of RAM, and importantly for our purposes, a 40 pin general-purpose input-output (GPIO) header (https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/). Our Raspberry Pi is programmed to run the standard Raspbian operating system, which can be installed via NOOBS— or, New Out of Box Software—available here (<https://www.raspberrypi.org/downloads/>) and easy to install. The main advantage to using this device is that it is a fully-functioning computer. Also, it has GPIO capabilities, which is highly advantageous for a novice in electronics. Linux provides all of the necessary drivers to read information from standard USB computer mice at rates up to 1000 Hz. This means that no proprietary drivers need to be written. Additionally, the GPIO-library specifically written in Python for Raspberry PIs allows easy control over the input-output pins, even providing interrupt capability.

To optimize the ability of the Raspberry Pi to function as a user-friendly microcontroller, a couple of minor, easy optimizations can be made to the operating system. First, one has to enable serial capability on the device. This will allow the computer to send serial information (in this case, motion data) via its default serial pins. Secondly, one can change the polling rate to 1000 Hz, higher than the standard 125 Hz. Depending on the highest polling rate that the computer mouse is capable of, this will make the computer look for mouse events at a higher rate, providing a better time resolution to mouse sensor readings. Lastly, the Python script can be placed or referenced in a location such that it will begin at startup. In our design, we simply added the line “sudo python /home/pi/rpi\_mouse\_synchronous.py &” to the last line of the file “ /etc/rc.local”. A complete tutorial is available at https://github.com/mfromano/micro-control.

The Python script, rpi\_mouse\_synchronous.py is a simple, multi-threaded python script. Once in experiment mode (dictated by a digital 1 signal from the Arduino to one of the GPIO pins), the python script runs single threads for each of the computer mice, and asynchronously captures and accumulates motion data. When it receives another input signal from the Arduino during each frame, it resets its accumulated motion information after sending the accumulated data to the big PC. In order to prioritize this Python script over other processes that the computer might be running, we give this script a high priority using the command “os.nice(1)[CHANGE THIS TO A HIGHER PRIORITY???]”

*Teensy 3.2*

The Teensy 3.2 (<https://www.pjrc.com/store/teensy32.html>) is a less well-known microcontroller with several advantages compared with the Arduino. First, it has a higher clock rate than the Arduino (72 MHz vs 16 MHz), allowing for faster data acquisition rates. Second, it has an output voltage of 3.3 Volts, compared to the Arduino’s 5 Volt output. This offers a small practical advantage, as activating 5 Volt mode on the ADNS-9800 sensors requires additional soldering and modifications to the sensors. While in principle the entire experimental protocol could be run on the Teensy alone, this microcontroller has a limited number of pins, and many of them are utilized for the incorporation of the necessary wiring for both ADNS-9800 sensors. We find that using the Teensy in a modular fashion yields a less cluttered workspace, and facilitates the option of swapping in or out a Raspberry Pi or a different novel motion-sensing device as needed.

*ADNS-9800 Sensors*

The ADNS-9800 Sensors are highly sensitive and have high maximum sampling rates, with a maximum read rate of 12000 frames per second, and 8200 counts per inch resolution (<https://datasheet.octopart.com/ADNS-9800-Avago-datasheet-10666463.pdf>). We have included in our software package drivers for these sensors that allow for easy interfacing and reading from the “motion burst” register in these sensors, which returns displacement in the x and y directions. Currently, accumulated displacements are stored in the sensors, as opposed to the Raspberry Pi, which accumulates sensor readings from standard computer mice. This is possible because ADNS-9800 sensors store motion data in 16 bits instead of the standard 128 bits.

Results

[Demonstration of accurate timing for teensy]

[demonstration of accurate timing for python script]

[simple plot of fluorescence acquisition?]

Discussion