Spotter v0.4

Ronny Eichler, *University of Eastern Finland*

Closed-loop experimentation is increasingly common in basic neuroscience. Even simple modalities can provide meaningful feedback. Here I describe a simple to use, modify and inexpensive solution for online video tracking of laboratory rodents. The system provides analog and digital signals which can be used to control and automate behavioral experiments.

A repository of the code described is hosted under:
`http://github.com/wonkoderverstaendige/Spotter`

1 Introduction

A theme of increasing importance in neuroscience is the shift from open- to closed-loop in experimental designs. Instead of feed-forward interaction with the subject, feedback allows to continuously and rapidly adept to changes in the measured variables. For example the efficacy of perturbations greatly depends on the ongoing electrical activity (e.g. spike or LFP feature triggered stimulation [1, 2] or the behavioral state of the test subject. By carefully selecting observed variables used for conditional perturbation the power of experiments can be increased over indiscriminate interactions and subsequent offline analysis.

The modalities used for closing loops can in principally be any observed variable. However the ability to provide meaningful feedback depends on the task at hand and has to be sampled appropriately. A scientific question involving spike timing depending plasticity demands low identification and signal emission latencies on the order of few milliseconds and below. Feedback on sleep states or behavioral performance requires integration times over seconds and only sub-second precision, but might require a much larger set of features.

For a large majority of experiments however feedback can provide support for researchers without being a core modality. Automation of parts of the paradigm can greatly reduce time requirements, allow running of a great number of experiments in parallel and decrease effects introduced by the investigator, from handling variability to imprecise execution timings.

As the number of variables involved in feedback loops increases, so does complexity of setup, maintenance and control. This requires components to be robust, flexible and lend to tight integration and interoperability. Many commercially available products provide high performance, but are often hard to adapt to individual demands, if possible at all (e.g. NeuroNexus or Noldus products).

One of the most versatile experimental variables of experiments involving awake behaving animals is the acquisition of video. It allows detailed analysis of behavior to provide context for recorded internal states, like electrical activity. Here, I will describe a system for online video tracking of laboratory rodents that can be used as a component in closed-loop experimentation, using affordable off-the-shelf components and open source hard- and software.

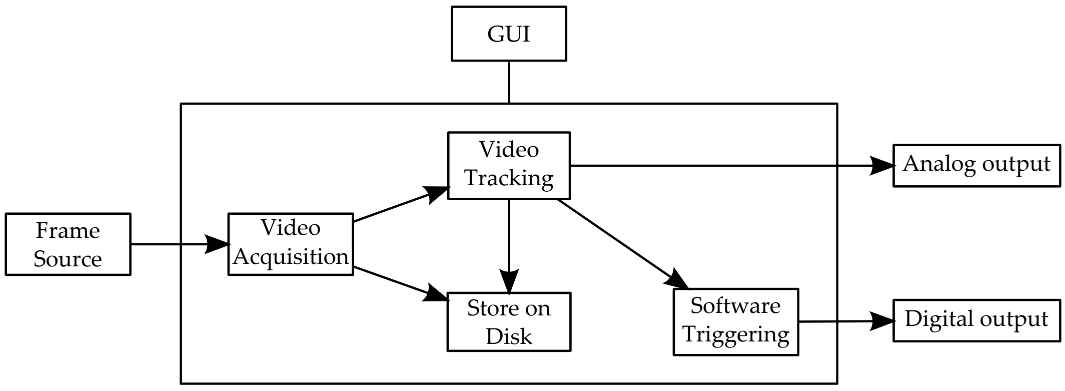


Figure 1: *Design outline for online video tracking with physical output.*

2 Design goals

The goal is to provide a system that can bridge the gap between video streams with high information density to defined physical outputs based on flexibly adjustable criteria.

To provide a hardware interface that can be interpreted by experimental setups the system should give out pseudo-continuous signals as analog voltage values. These can be used for analog thresholding and filtering or by more proficiently by systems with analog-digital converters (ADCs). Secondly, digital outputs should provide signals for fast triggering based on selectable conditions minimizing computational load downstream. This reduces complex states down to binary signals. These signals, analog and digital, should be externalized in a way suitable for simultaneous monitoring on common electrophysiology data acquisition systems.

To prevent data loss in case of improper set up and for offline analysis in general, the video data, detections and trigger events as well as the parameter states should be recorded suitable for archiving and fault tracing 1.

Most commercially available consumer hardware provides video signals at 30 frames per second. This is sufficient for a large range of tasks. The delay between image acquisition and signal emission should be low enough to provide feedback within a theta cycle, as demanded by the experiment originally starting development.

Software and hardware should be easy to obtain, inspect, set up and modify without advanced knowledge.

3 Implementation Considerations

3.1 Software

Many researchers are familiar with the MATLAB numerical computing environment (The MathWorks). MATLAB has support for image processing and can interface cameras through the Image Acquisition Toolbox. However, MATLAB and its toolboxes are expensive, building graphical user interfaces can be a jarring experience and is a rather bulky application.

The most widely used library for computer vision is OpenCV. The open source framework is available for all major operating systems (Windows, Linux, OSX and iOS). While originally a C/C++ library, bindings for other languages are available (e.g. Python, Java). The library provides basic handling of input and output of image data (image and video files, many camera devices, streams) via the FFMPEG or GStreamer libraries at high de- and encoding speeds. OpenCV also exposes basic graphical user interface (GUI) methods for prototyping and a large number of numerical methods for most routine computer vision and machine learning tasks.

To make the resulting code of this project accessible for modification and maintenance, I decided to use the Python language bindings over C/C++ or Java. Python is a popular [4] open source general purpose programming language that is easy to learn, has a focus on readability, can be used to write self-documenting code and is highly extensible. The SciPy/NumPy libraries provide mature support of numerical computation, linear algebra and machine learning. More importantly, recent OpenCV Python bindings are based on the linear algebra methods provided by NumPy. Several Python distributions focused on scientific computing (e.g. Enthought, PythonXY, Anaconda) provide bundles of the core requirements with integrated development environments (IDEs). This provides familiarity to MATLAB users while offering the advantages of a general purpose language. However, Python is a dynamically typed interpreted language. It is often slower than compiled languages (e.g. C/C++), although a number of ways exist to alleviate this difference (see section 6). Modern multi-core desktop computers should nevertheless have enough computational capacity to allow realtime processing of video at moderate frame rates (≤ 60 fps) and resolutions even in this non-optimal case.

In listing 1 a minimal example of Python code is provided. The script uses the OpenCV library to open a video stream, here from a device with $id = 0$ (for example a camera) and displayed at 30 fps until the user presses a key. This example uses the basic interface methods provided by OpenCV to draw the frame. However, these methods are insufficient for building a more advanced

```python
from cv2 import *
vc = VideoCapture(0)

while waitKey(30) < 1:
    rv, frame = vc.read()
    imshow('preview', frame)
```

Listing 1: *A minimal example of using the OpenCV Python bindings to receive and display frames from a video source. The library is imported and a video capture object created. The while loop is querying the capture object for new frames every* 30 ms *until a key is pressed.*

graphical user interfaces. Instead, the Spotter user interface is built with the open source Qt library. It is available on most platforms, has a large range of language bindings including C/C++ and Python, carries no license fees for non-commercial applications and is used by a number of common neuroscience tools (e.g. NeuroScope, NDManager, KlustaViewa).

Qt provides development tools to quickly build the layout (e.g. Qt Designer) and store these definitions in a language independent format separating structure and function. This way existing user interface designs can be ported quickly between different language implementations. This allows to transfer existing GUI designs from one language implementation to another. If the Python bindings of OpenCV turn out to not provide the required performance, this would greatly ease the transition to a C/C++ implementation.

3.2 Hardware

Three hardware building blocks are to be considered. Firstly, the video source device. While the system should be able to process video files for offline analysis, for realtime applications a low latency video stream of the subject to track is required. OpenCV provides a versatile `VideoCapture` (as well as `VideoWriter`) method that uses the FFMPEG multimedia library to decode video device streams from IP-cameras and USB-cameras compatible with the UVC (USB video device class) standard. Consumer-grade USB 2.0 camera devices are typically limited in resolution, frame rate and image quality, especially under poor lighting conditions. For many experiments a temporal resolution of less than 30 ms is sufficient, and tracking can be simplified by if animals carry easy to track hardware (like LEDs). Professional computer vision cameras (e.g. PointGrey Gigabit Ethernet/USB3 devices) can go beyond these basic parameters, but may not be supported by the OpenCV frame aquisition methods as they often require the use of vendor specific SDKs. Python is able to interface with external modules written in C, allowing the integration of these devices with moderate effort if needed.

This software was developed using USB2.0 attached LifeCam Studio 1080p HD Webcam (Microsoft) cameras. They have good resolution and reasonable image quality at a moderate price ($75 at the time of writing). However, USB2.0 is fairly bandwidth limited (480 Mbps in theory) for high definition video. When purchasing a new camera, more recent standards (USB3.0, Thunderbolt) should be preferred, and are increasingly common.

Secondly, A desktop computer or laptop is used as the main processing device. While being the central device in the framework, its requirements are rather loose. A fairly modern computer (< 5 years old) has sufficient processing power and connectivity (USB). All libraries used are available on the major operating systems (Microsoft Windows 7, Linux 2.7+, Apple MacOSX) and the software has successfully been tested on all of them (however, the installation process is easiest on up-to-date distributions, and most involved on Apple machines),

A microcontroller is used to produce the physical output signals. While most PCs possess some ability to generate digital and analog signals (parallel ports, sound devices) they are limited in the number of channels, increasingly rare and burdened by other limitations (high-pass filters, difficult to control reliably). Microcontrollers are devices independent from the main processing machine (the PC) which are controlled via serial interfaces. A large variety of microcontroller ecosystems exist, but most have complicated toolchains, require familiarity with the underlying architecture and an intimate relationship with their datasheets or the assembly of the required peripherals. One of the most popular beginner-friendly ecosystems is the Arduino platform, providing open source hardware and software together. Its central element is a standardized range of microcontroller boards, typically based on 8 bit microcontrollers (AVR ATmega328; Atmel). These are available from a large number of distributors word-wide ($20-50). The standard board implementation has USB connectivity and is pre-programmed with a bootloader which handles USB communication and acts as the self-programming in-circuit serial programmer (ICSP). The second component of the Arduino platform are an IDE that acts as a multi-platform toolchain to compile programs for the microcontroller, and a universal device driver. Typically microcontrollers are programmed using very low level languages, C or assembly, and require the user to reach an advanced understanding of the hardware. Arduino uses an abstraction and superset of C, providing higher-level access to the most common functions and applications. Additionally a large number of libraries for common devices and applications exist, in addition to beginner friendly learning resources. In summary, the Arduino microcontroller platform allows to easily read and write programs to be run on standardized hardware, optionally interfaced through a via USB connection from a host computer.

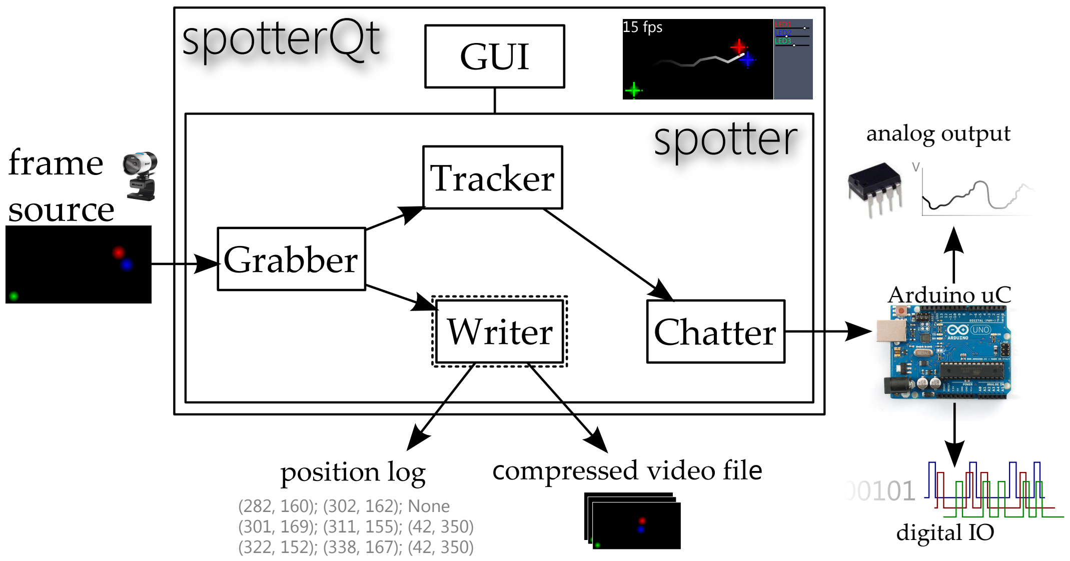
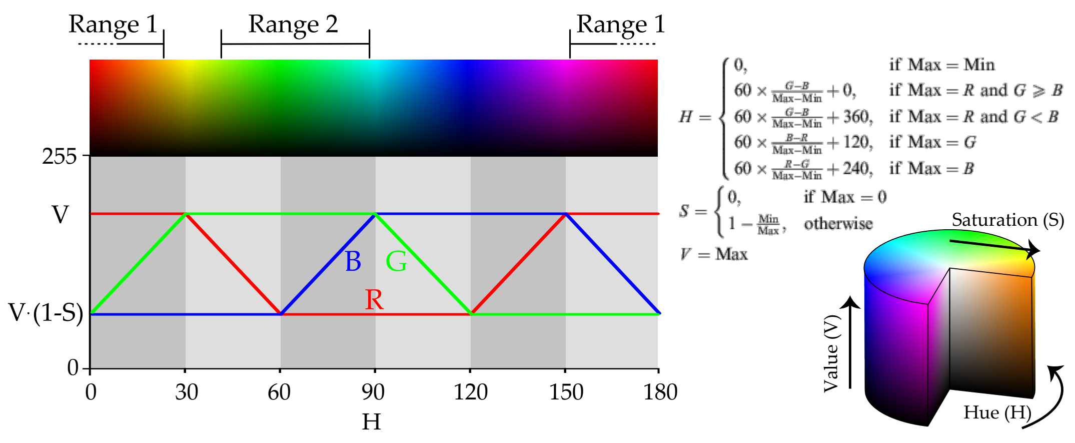


Figure 2: *`spotterQt` is the graphical user interface based on the Qt framework. It wraps around the core `spotter` module to provide interactive control and visual feedback. The core module consists of four sub-modules, separated by task. The frame `Grabber` handles the capture object and returns a new frame when called. The `Tracker` uses defined feature sets to track the position of objects in the frames. For example bright spots from the position LEDs on the implant of a laboratory rodent. The position data is processed and defined results sent through the `Chatter` module over a serial connection to an Arduino microcontroller, which provides digital or analog output of the data. Frame and position data are stored on disk by a `Writer` module as compressed video and text log files. The writer is running in a separate process (dashed box) to provide data-handling without blocking the main thread.*

4 Implementation

4.1 The `spotter.py` main loop

The main loop of the script, referred to as the `spotter` module, can be run independently from the graphical user interface. This provides command line functionality to run headless on a server or be controlled from a remote machine. Additionally, this allows offline batch processing of recorded video data. Each loop iteration processes one frame from the video source, except when processing is paused. Within the `spotter` module, the core functionality is handled by four sub-modules, instances of the aptly named `Grabber`, `Tracker`, `Chatter`



$$H = \begin{cases} 0, & \text{if Max} = \text{Min} \\ 60 \times \frac{G-B}{\text{Max}-\text{Min}} + 0, & \text{if Max} = R \text{ and } G \geqslant B \\ 60 \times \frac{G-B}{\text{Max}-\text{Min}} + 360, & \text{if Max} = R \text{ and } G < B \\ 60 \times \frac{B-R}{\text{Max}-\text{Min}} + 120, & \text{if Max} = G \\ 60 \times \frac{R-G}{\text{Max}-\text{Min}} + 240, & \text{if Max} = B \end{cases}$$

$$S = \begin{cases} 0, & \text{if Max} = 0 \\ 1 - \frac{\text{Min}}{\text{Max}}, & \text{otherwise} \end{cases}$$

$$V = \text{Max}$$

Figure 3: *RGB to HSV color space conversion.* Left: *Ranging of HSV values wraps around the origin if stepping over red (hue 0).* Right: *Formula for the conversion of RGB values to the cylindrical HSV color space.*

and `Writer` classes (Figure 2).

4.1.1 Grabber

The `Grabber` initiates, handles and holds the frame source (either a camera device or a video file) capture object reference (`vc` in Listing 1). Many cameras compress frames and encode them in color spaces other than RGB to overcome bandwidth limitations. Grabber reads and decodes new frames. OpenCV returns a boolean return value if a frame was read successfully, and a 3-dimensional NumPy array. The array contains a 2D array slice with *width x height* pixel elements for each of the RGB (red, green, blue) color channels. The elements of the array are stored as unsigned 8-bit integers (uint8), ranging from 0 to 255. Together, each pixel is can be represented by 24 bit RGB value, which OpenCV handles in B-G-R order. For futher processing, these arrays can be accessed and handled like typical objects in numerical computation software.

Follow frame acquisition several pre-processing steps might be necessary. To reduce computational load images can be scaled, as the number of pixels decreases with the square of the scaling factor. Cheap consumer-grade cameras are noisy, especially under low light conditions. Low-pass filtering and and morphological operations are common pre-processing steps that can emphasize features of interest. They require careful use, however, as they tend to be computationally expensive and can reduce the signal fidelity. Example steps include smoothing (removes high frequency noise), dilation (exaggerates foreground features) and erosion (decreases features from their boundaries). Depending on the tracking algorithm employed, the frame may have to be converted to a suitable color space.

The features typically tracked in the intended application consists of brightly colored spots of the LEDs attached to the headstage of implants on laboratory rodents. Using RGB values however, these properties are not immediately accessible. Separation by the independent color channel values alone is often impractical. A bright red pixel (e.g. R=255, G=110, B=110) has a higher blue component than a dark blue pixel (e.g. R=0, G=0, B=100). More generally, the brightness, saturation and color are setting the off from the background (Figure 4). To easily access these features, the frames are transformed from the RGB to the HSV color space (Figure 3). HSV is a cylindrical three channel color space. Hue (color tone) is encoded in a circular mapping, and value (brightness) and saturation (color intensity) are encoded linearly. Commonly the hue range is defined between $0 - 360°$. However, the image data is stored as 8 bit values per channel. To fit the hue value into this range, OpenCV represents the hue space as half values ranging from 0 to 179. Distinctly colored pixels with defined saturation and brightness can now be separated by simple thresholding operations on the corresponding channels.

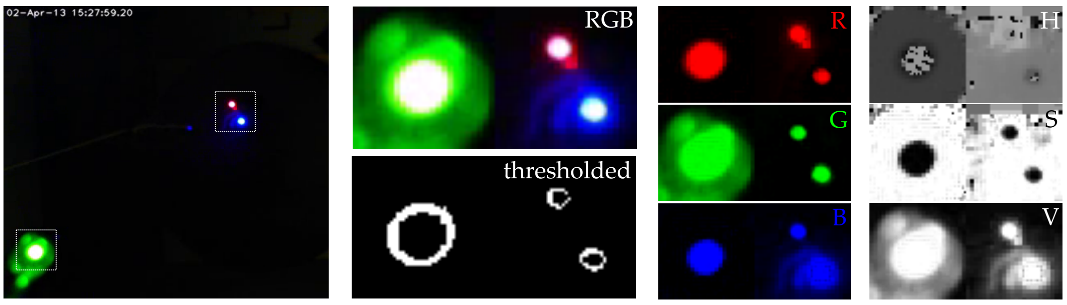


Figure 4: *Example image of colored LED spots.* Left: *Example frame with a green LED for synchronization, and two LEDs (red and blue) carried by the animal.* Center top: *Magnified RGB image of the three spots.* Right: *The separate R, G and B channels before, and separate H, S and V channels after the color space conversion.* Center bottom: *Binary masks after each spot had thresholds applied to saturation and value. As center pixels are overexposed, they have no meaningful hue value and have to be excluded.*

4.1.2 Tracker

Three helper classes (features, regions of interest, objects of interest, defined as `Trackables`) are used as representations for different objectives.

Features Features are the elementary unit for tracking and represent descriptive sets of parameters. These are used by the appropriate tracking algorithm to find the feature in frames. For LEDs, the typical use case in this project, these sets are hue, saturation and value ranges, as well as maximum and minimum area constraints. Additionally, features can be flagged as static or moving (for example a synchronization LED appears at a fixed location) by the user.

Region of Interest Regions of interest (ROIs) are parts of the image that can interact with Objects for conditional signaling. ROIs consist of one or more geometric shape descriptors. Shapes (rectangles, circles, lines, polygons) consist of at least two points. The shape type defines how the points are interpreted. ROIs can be linked with Objects, and each Object/ROI connection tied to a physical output pin on the microcontroller. Collision or traversal of the Object through a linked ROI triggers a state change on the associated digital channel.

Objects of Interest Objects of interest (OOIs, the name is chosen to separate the naming from programmatic objects) are representations of real-world objects to be tracked. Objects consist of a set of one or more features. For example the subject in figure 4 is represented by two features, one for a red, and one for a blue LED. A second object, the synchronization LED, is represented by a single static green LED feature. After all active features have been tracked (returning either coordinates or `None`) the position of an OOI is calculated as the mean of all its linked features. If no feature of the Object was detected, the position of the Object for that frame becomes `None`. This allows distinction of a zero position from a tracking failure in the output signals. From the history of positions a range of parameters can be calculated. For single frames, Objects with multiple features can have a direction of an arbitrary axis, calculated as the normal vector of two of its features. For multiple frames, velocity and acceleration can be computed. These values, together with digital triggers based on the collision with specified ROIs, can be sent to the microcontroller to be represented as digital or analog electrical signals.

Tracking The `Tracker` iterates over a list of features. For the purpose of this report, I will limit the description to simple tracking by application of thresholds in the HSV space.

To reduce the time spent on this step, a simple adaptive search window is applied by slicing the frame array during tracking. The window is calculated by stepping through the position history until a valid position is found. Centering on this position a rectangle is set up, increasing in size with the number of

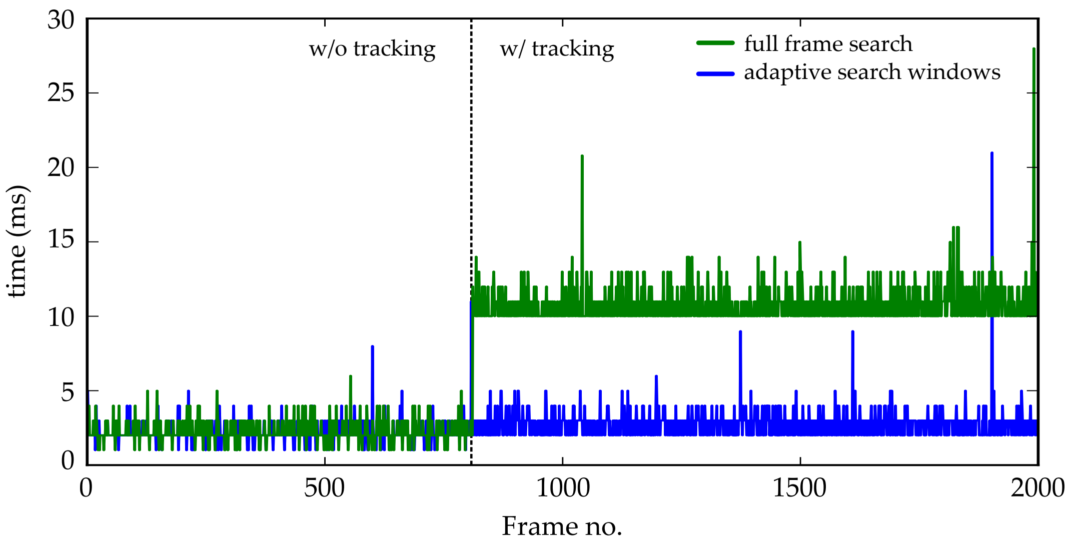


Figure 5: *Tracker computation time for HSV thresholding operation is greatly reduced for adaptive region of interest constrained search. Tracking was started at dashed line. With adaptive search windows (blue) tracking time only slightly increases over pre-processing time. Without search constraints total tracking time is four times higher (green).*

invalid positions preceding the valid one. This incrementally expands the window if a feature is lost. In the current state of development, not using information about directionality, velocity or acceleration or other prediction improvements to further reduce window sizes, this yields dramatically reduced search times for features with reliable detection. For example in figure 5 three LEDs are tracked with minimally $100 \times 100\ px^2$ windows from $640 \times 360\ px^2$ frames. The time per iteration is reduced from 11 ms to under 3 ms, including the unchanged pre-processing fixed time costs.

A binary mask array of pixels matching the feature description is created (Figure 4, bottom center). The element-wise application of range thresholds onto the three color channels of the converted frame array yields three binary arrays which are joined with the AND operator. To find contiguous pixel areas, a border following algorithm extracts contours [3]. The first raw moments ($m_{ij} = \sum_{x,y}(frame[x,y] \cdot x^i \cdot y^j)$) are used to calculate the area (m_{00}) and the centroid for contours within the maximum and minimum area ($\bar{x} = \frac{m_{10}}{m_{00}}$, $\bar{y} = \frac{m_{01}}{m_{00}}$). The resulting coordinates of the contour with the largest area represent the geometric center of the detected pixel group for the feature.

A drawback of the naive thresholding method for detection colored pixel groups is the flooring effect of over-exposed values. The RGB to HSV conversion algorithm can not return meaningful values of the hue if all color components are equally present, and defaults to zero (red). This equates to the longitudinal axis of the color space cylinder (Figure 3). For the most part these are rare cases and can be ignored. However, if the brightness of a light source saturates the image sensor, these pixels will be whitened, and fall back to Hue value of zero when converted to the HSV color space. To avoid this, the upper limit of the Value channel is set close to this maximum, and white pixels are ignored. This can lead to ring-shaped contours being found if performed aggressively (Figure 4, bottom center). For the calculation of the centroid this is of minor importance and will be accepted. It is preferable to have slightly reduced tracking precision over missing the spot altogether. A second drawback is the ranging over intervals crossing the origin (e.g. compare Range 1 and 2 in figure 3, top) wraps around. This requires to either perform two operations of the two sub-intervals, or one operation of the inverted interval and the negation of the returned mask. Depending on the implementation of the ranging operation (separate or combined iteration over channels), either can be advantageous.

If no valid position for a feature is found at a time point, the Python NULL value `None` is returned instead, to distinguish numeric values from the fault case.

4.1.3 Chatter

The `Chatter` module handles all connectivity with the microcontroller. Most importantly it holds and wraps reference to the instance of an `arduino`-class that takes care of low-level communication and opening the serial port connection by bootstrapping the `pySerial` library. Establishing the connection requires finding the port the microcontroller is connected to, which varies greatly between operating systems. Excluding unlikely ports (e.g. Bluetooth serial devices) ports are scanned during automatic port selection by opening and polling all listed ports sequentially. If the right port is found, the microcontroller will respond with a fixed message. To further validate the connection, a handshake is performed. First, `Chatter` will send a string of data which will be mirrored back by the controller. If the send-and-receive test was successful, the microcontroller will report the device capabilities (number of DAC, digital output, digital input and PWM pins).

After each frame was processed and features/objects were tracked, selected signals (see the interface section (4.2) on linking detections with signal pins) are transmitted to the microcontroller using a simple instruction-data protocol. A four byte packet is sent for each signal per frame, consisting of one instruction byte and two data bytes, followed by a newline symbol (Figure 6). The last

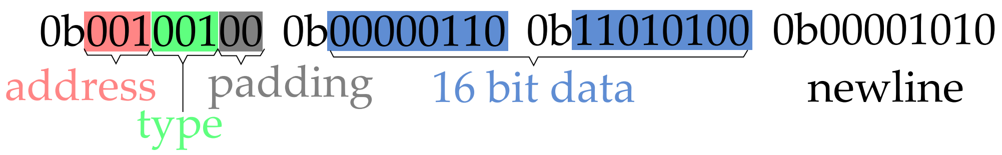
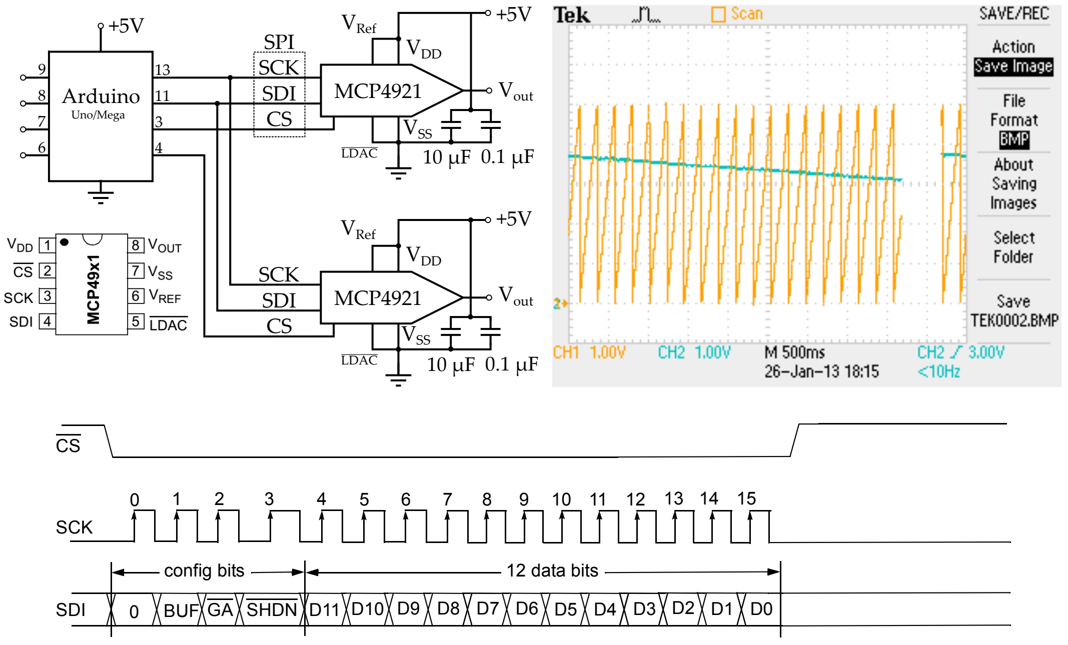


Figure 6: *Structure of a four byte example packet employed in the Spotter-Arduino communication protocol. The example sets DAC 2 to 2.1 V (Figure 9, right).*

byte marks the packet end, adding redundancy to allow recovery from packet alignment loss. The instruction byte contains 3 address bits and 3 type bits indicating the signal type (e.g. an analog signal on DAC channel 2, see figure 6). Two padding bits are filling the instruction to a full byte, adding debugging information and shifting the instruction to values representing human readable symbols in the ASCII table (e.g 'I', 2nd DAC channel).

Analog output Digital to analog conversion (DAC) of pseudo-continuous values (i.e. position, speed, head direction, acceleration) is performed by 12 bit DAC ICs (MCP4921; Microchip), interfaced by the microcontroller via SPI (Serial Peripheral Interface, for protocol details please see datasheet; Figure 7). SPI is a bidirectional serial bus protocol with master-slave architecture. It requires at least three lines (four for bidirectional communication, not required for the DACs) for a clock line (SCK), and a master-to-slave data line (SDI). Additionally, for each slave device a single device-select line is required to address a data stream to a specific device (CS). The MCP4921 uses an external voltage reference (V_{ref}), which is tied to the supply voltage, providing near rail-to-rail range (40 mV offset at both ends, effective V_{range} of 0.04 V to 4.96 V for a perfect 5 V supply). Two optional bypass capacitors (0.1 µF ceramic and 10 µF tantalum) close to the voltage supply of the DACs may reduce high-frequency noise in the output signals.

To convert the position to an analog voltage value, three conversation constants are computed based on the DAC specifications and the frame size. To clearly separate invalid values (e.g. position tracking failed), the zero position is offset by 1/16th of the range. First, the DAC factor converts pixels units to arbitrary DAC units: $(\max_{DAC} - \frac{\max_{DAC}}{16}) \times framesize_{\max}$, with $\max_{DAC} = 4096$ for a 12 bit DAC. A frame size of 640×360 pixel has a DAC factor of 6 units per pixel. A DAC unit is converted to voltage values by the factor $\frac{V_{range}}{\max_{DAC}}$, which in the case of MCP4921 and a 5 V supply and reference voltage corresponds



Write Command for MCP4921 (12-bit DAC).

Figure 7: *Arduino and DAC connections.* Top left: *Two MCP4921 12 bit DACs are interfaced to the Arduino via SPI protocol, requiring two common lines for clock and data signals, and one line per device for destination selection.* Top right: *Screenshot of a test pattern, quickly scanning the available range. Data is sent from a python program to the Arduino, which updates the DAC values accordingly.* Bottom: *SPI protocol for the MCP4921 DAC. 2 bytes (12 data bits, 4 configuration bits) are sent aligned to the clock line, when the device selection line (CS) for the specific DAC is pulled low. Voltage values are updated immediately when a valid write command is received, as the LDAC buffer pin is permanently pulled low.*

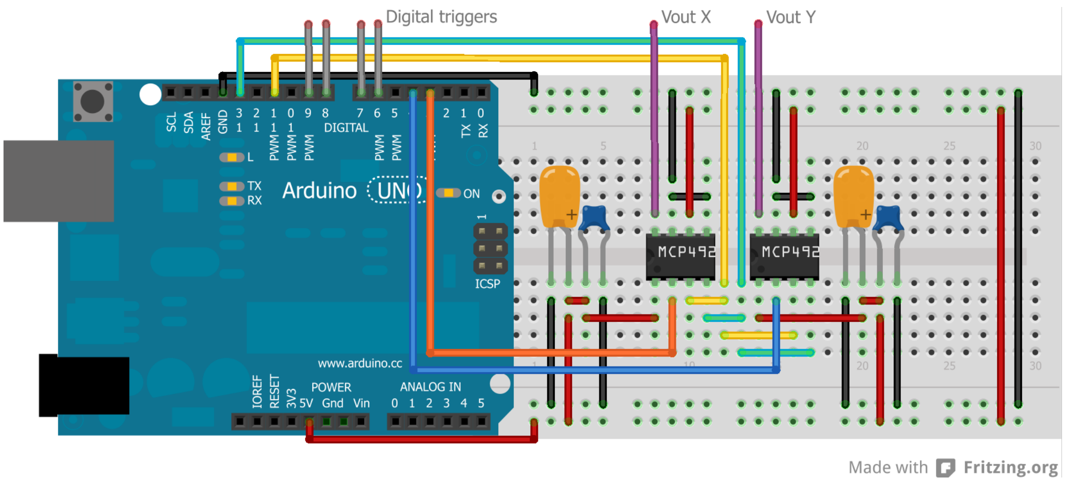


Figure 8: *Schematic of the assembled hardware.* Left: *Arduino Uno microcontroller board. The Arduino is connected to a PC via USB and provides regulated power, SPI data to the DACs (CS: blue/orange; SCK: yellow; SDI: green) and digital trigger outputs (gray).* Right: *Breadboard with wire connections and bypass capacitors to the MCP4921 DAC ICs producing analog output signals (Vout X/Y: violet).*

to 1.2 mV per unit. In total, pixel values have 7.2 mV resolution. To represent images with non-unary aspect ratios, the minor axis is translated into the space of the major axis, maintaining an absolute pixel-to-voltage ratio (Figure 9).

Digital output Digital signals are updated in a straight-forward update loop. Each frame all digital pins are updated to either HIGH or LOW states. In the current version, a full packet is sent per digital pin. All non-zero data values are interpreted as HIGH, zero-data as LOW.

4.1.4 Writer

To store experimental data for offline analysis, the video stream can be stored as a compressed video file (codec choice depends on the installed OpenCV backend), together with a log file of timestamped position and event data. Writing to hard drives can cause blocking long-latency operations, especially if other processes are accessing the hard drive with long seek times (e.g. the computer is performing electrophysiology data acquisition in parallel). To prevent the main process from waiting for video data to be written, `Writer` is running as a separate process. To maintain thread-safety, the inter-process communication is handled by two queues, allowing the exchange of control

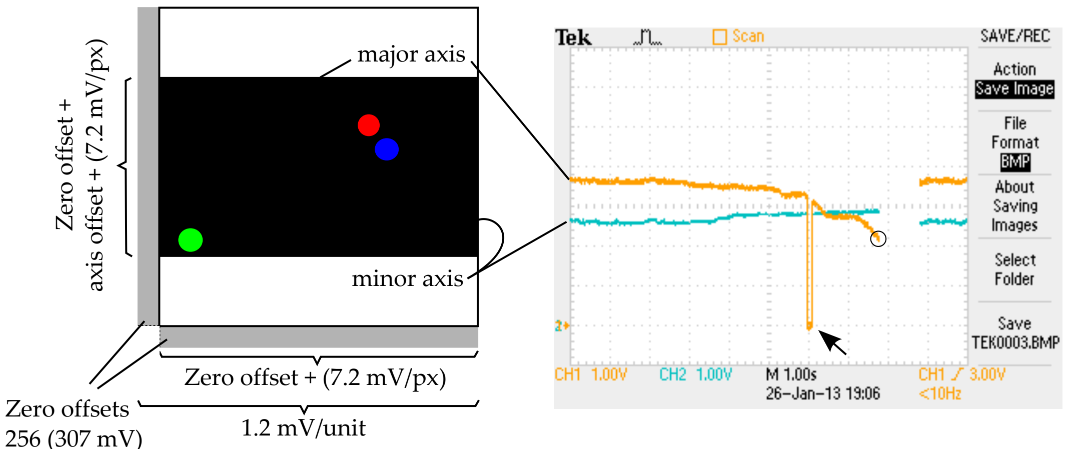


Figure 9: *Conversion of image location to voltage values.* Left: *Relation of frame to voltage values.* Right: *Example screenshot of the analog position signal (x: orange, y: blue), as the animal is starting to traverse the arena after the half mark. Shortly after tracking is lost for a frame and the signal defaults to a fault value (arrow). The example value of previous figures is shown as the current state of the major axis signal (circle).*

messages and data.

4.2 User Interface

The user interface was designed with Qt Designer, which produces language independent definitions. These were translated into Python code. The resulting modules are the structural description of the interface components. These were linked with functional code, defining the behavior of the interface elements. In its current state, starting the program requires providing information via command line parameters, most importantly the frame source origin. As Spotter is intended to be operable purely as a command line tool, interface equivalents are added successively, but as a secondary priority.

The core element of the interface is an OpenGL context, displaying the video frames as they are grabbed from the frame source. As frames captured by OpenCV are NumPy arrays, frames had to be transformed to be accessible by OpenGL without major performance hits. This is achieved by casting the whole arrays into strings and using the OpenGL `DrawPixels` method to read these strings as byte strings. The OpenGL frame also handles user input (mouse and keyboard) to allow interactions with the video frame (e.g. drawing and moving ROIs, selecting areas to extract features from). Onto the OpenGL

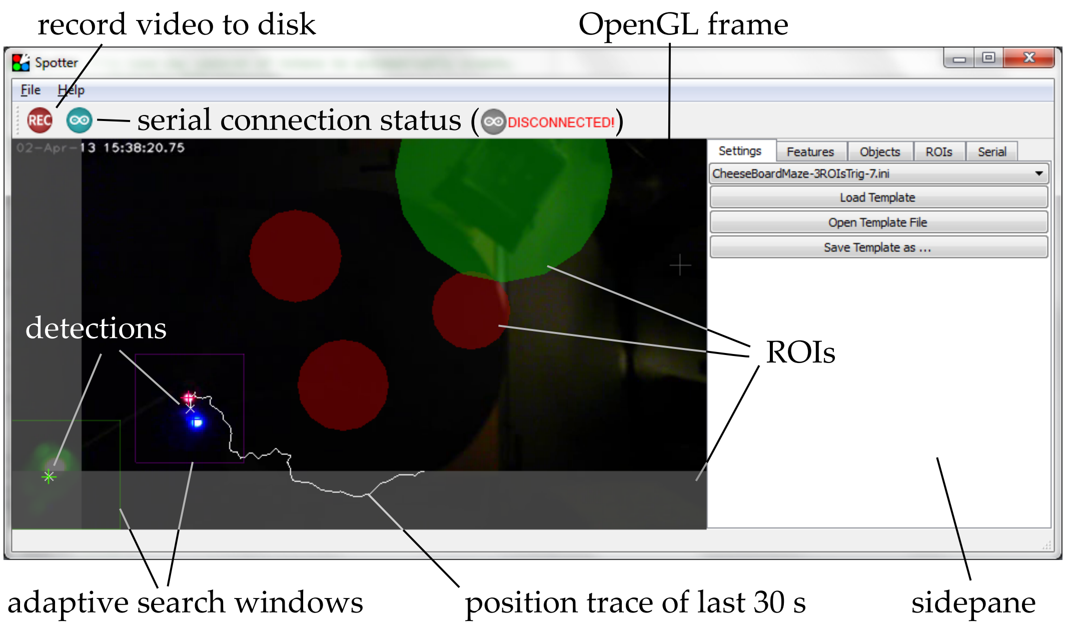
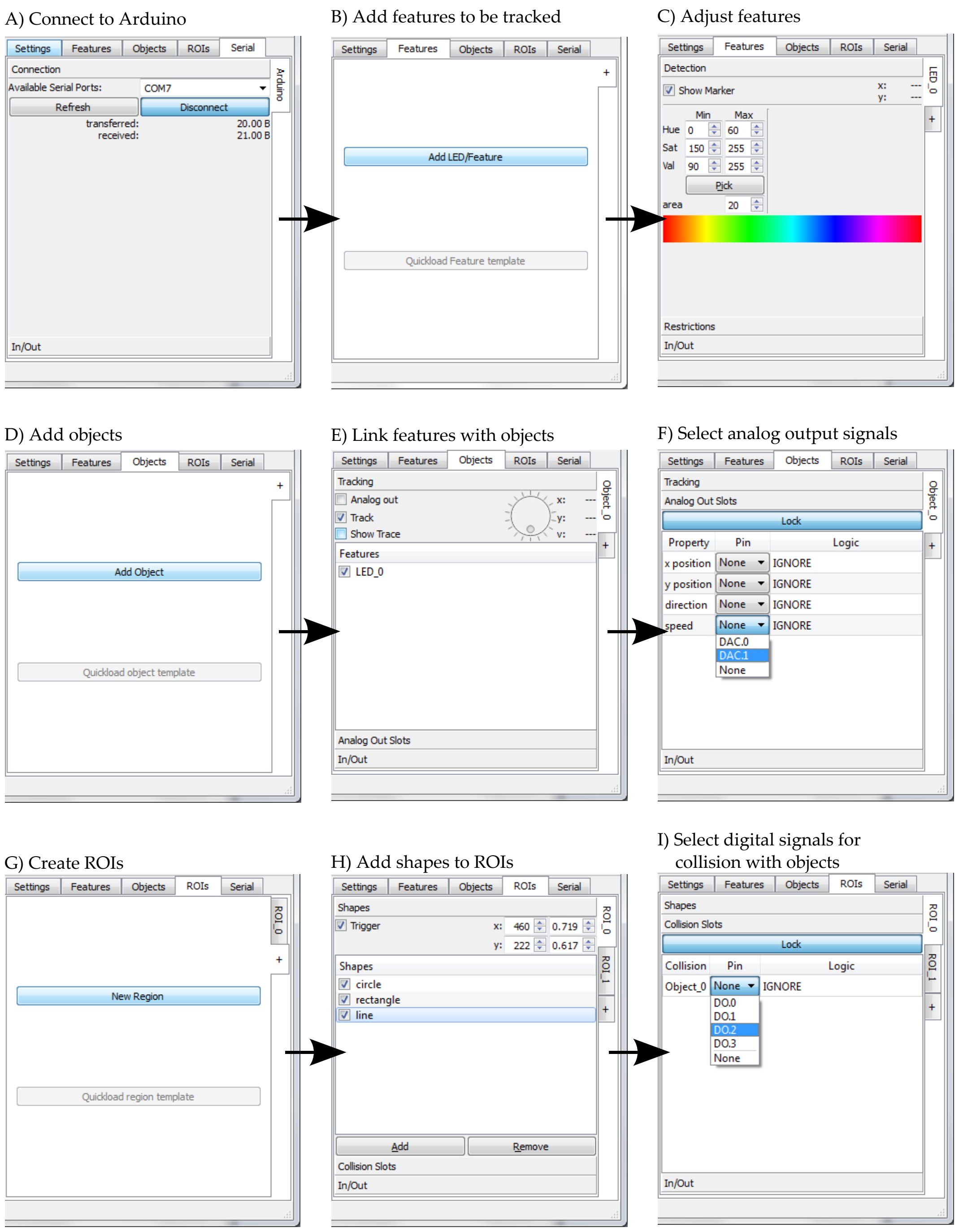


Figure 10: *Overview of the Spotter GUI elements.*

frame, frame-context interface elements are drawn (ROIs, tracking markers, debugging information) using OpenGL primitive methods. Currently, rendering and processing are performed sequentially.

I will refrain from detailing the user interface, but give a short overview of its main use pattern. In brief, the typical workflow starts by establishing a connection of the program with the microcontroller, which can happen automatically, or require manual adjustment of the connection parameters (Figure 11, A). To prevent data loss, a clearly visible indicator in the toolbar shows the current connection status (Figure 10, top). Next, features are added and adjusted to yield robust tracking performance (Figure 11, B and C). Secondly, objects are added and linked with one or multiple features (Figure 11, D and E). Object positions are calculated as the mean position of all its linked features. As long as free devices are available, object properties can be tied to analog output slots (Figure 11, F). Next, regions-of-interest are added. Each ROI can consist of multiple geometric shapes (Figure 11, G and H). To allow conditional triggering, objects and regions-of-interests can interact. ROIs can be linked with objects, and collision of ROI and object trigger digital output pin transitions (Figure 11, I).

Once the tracking and triggering is set up with, settings can be stored as template files for easy recovery and offline analysis (not shown). During the experimental run, video data, position and event logs can be written to the hard


A) Connect to Arduino
B) Add features to be tracked
C) Adjust features
D) Add objects
E) Link features with objects
F) Select analog output signals
G) Create ROIs
H) Add shapes to ROIs
I) Select digital signals for collision with objects
Settings
Features
Objects
ROIs
Serial
Refresh
Disconnect
Add LED/Feature
Quickload Feature template
Pick
Add Object
Quickload object template
Lock
New Region
Quickload region template
Add
Remove

Available Serial Ports: COM7
transferred: 20.00 B
received: 21.00 B

Show Marker

	Min	Max
Hue	0	60
Sat	150	255
Val	90	255
area		20

Tracking: Analog out, Track ✓, Show Trace
Features: LED_0 ✓

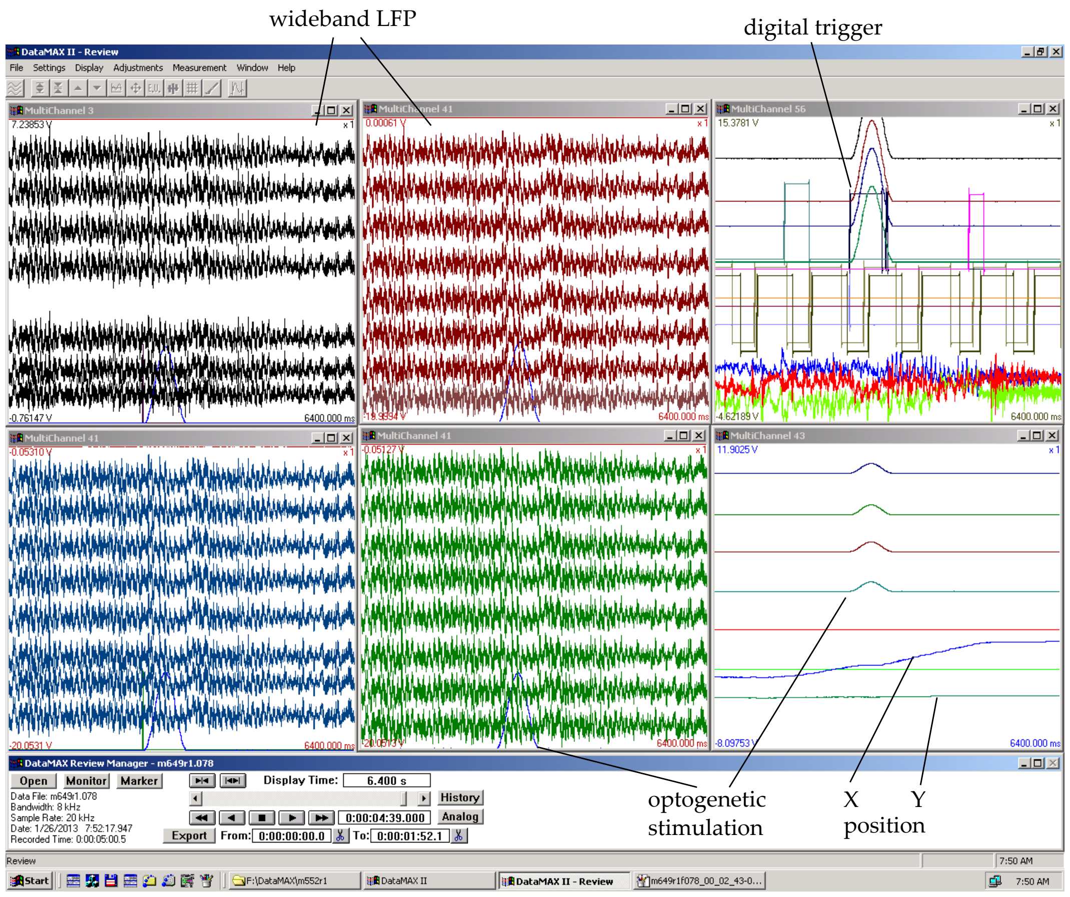
Property	Pin	Logic
x position	None	IGNORE
y position	None	IGNORE
direction	None	IGNORE
speed	None	IGNORE

Trigger ✓ x: 460 0.719 y: 222 0.617
Shapes: circle ✓, rectangle ✓, line ✓

Collision	Pin	Logic
Object_0	None	IGNORE

Figure 11: *Simplified workflow to set up a new experiment.*

drive by toggling the record button (Figure 10). During writing, a timestamp is added to the video frame, additionally to the timestamps in the log files.



Data File: m649r1.078
Bandwidth: 8 kHz
Sample Rate: 20 kHz
Date: 1/26/2013 7:52:17.947
Recorded Time: 0:00:05:00.5
From: 0:00:00:00.0 To: 0:00:01:52.1

Figure 12: *Example screenshot of position triggered optogenetic stimulation together with 32-channel wideband LFP signals. Position data is additionally to the digital triggered recorded as an analog voltage value (blue and green traces, lower right corner).*

5 Results

Online video tracking with Spotter is now actively and routinely used in Prof. Buzsaki's group to trigger optogenetic stimulation for closed-loop experiments (Figure 12). As the application is under development, several bugs exist, and the hardware is not finalized. Initial analysis of the spatial and temporal characteristics shows good spatial, but limited temporal precision.

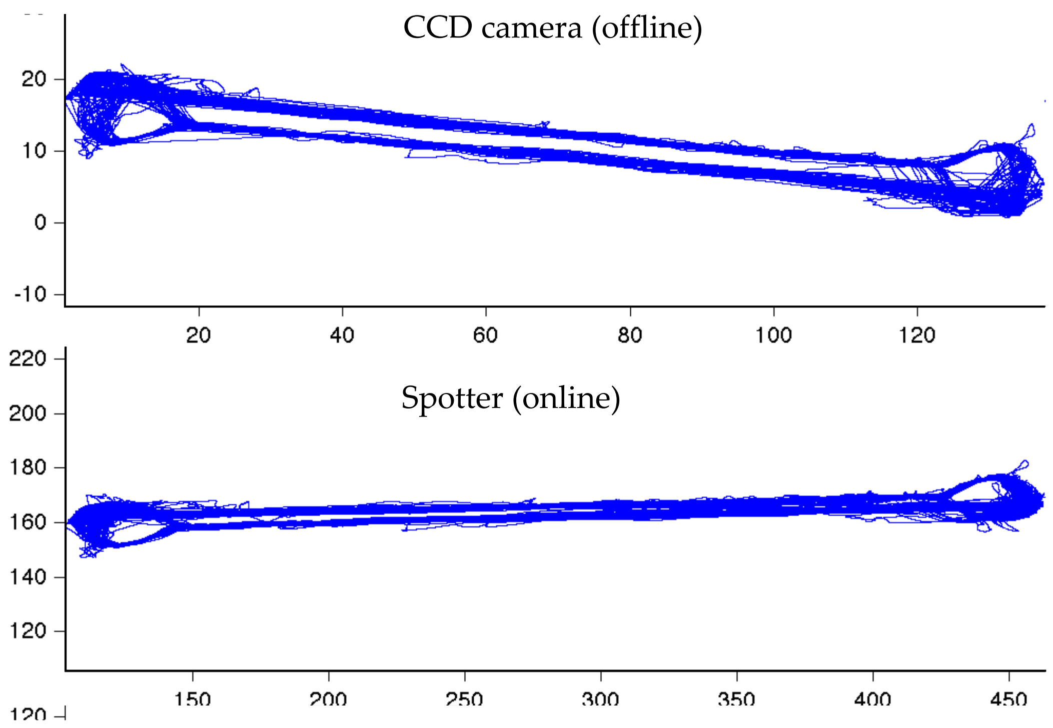


Figure 13: *Comparison of offline computed position with Spotter position tracking as computed from the analog position signal recorded concurrently with the electrophysiology data acquisition system. (units are arbitrary (CCD camera) or pixel (Spotter))*

5.1 Spatial precision

While the spatial resolution is on the order of the frame resolution if the image was not scaled for tracking, the effective precision greatly depends on the tracked feature and the feature definition. Partial detection, caused for example by overlap of the tether with an LED, complete coverage and failed detection, as well as partial rejection of the feature due to overexposure or improper parameter definitions can cause small changes in the computed position of objects. For objects with multiple features, failed detection shifts the mean position and can cause jitter in the signal. Compared to the offline analysis of the previously used camera, Spotter yields highly coherent data compared to the same experiment recorded with a CCD camera (Figure 13). Both show near identical movement patterns, even though they have been recorded from slightly different angles.

5.2 Temporal precision

The temporal properties of the online system have to be measured relative to a signal with high temporal precision. In figure 14 the signal of a green flashing synchronization LED is tracked by having an object, linked to a single feature, collide with a ROI. Upon collision detection, a digital pin of the Arduino is updated after transmission via serial port. This allows precise estimation of the detection and triggering delay. Measuring the peak cross-correlation shows about 109 ms in lag between the current applied to the LED and the digital signal being recorded.

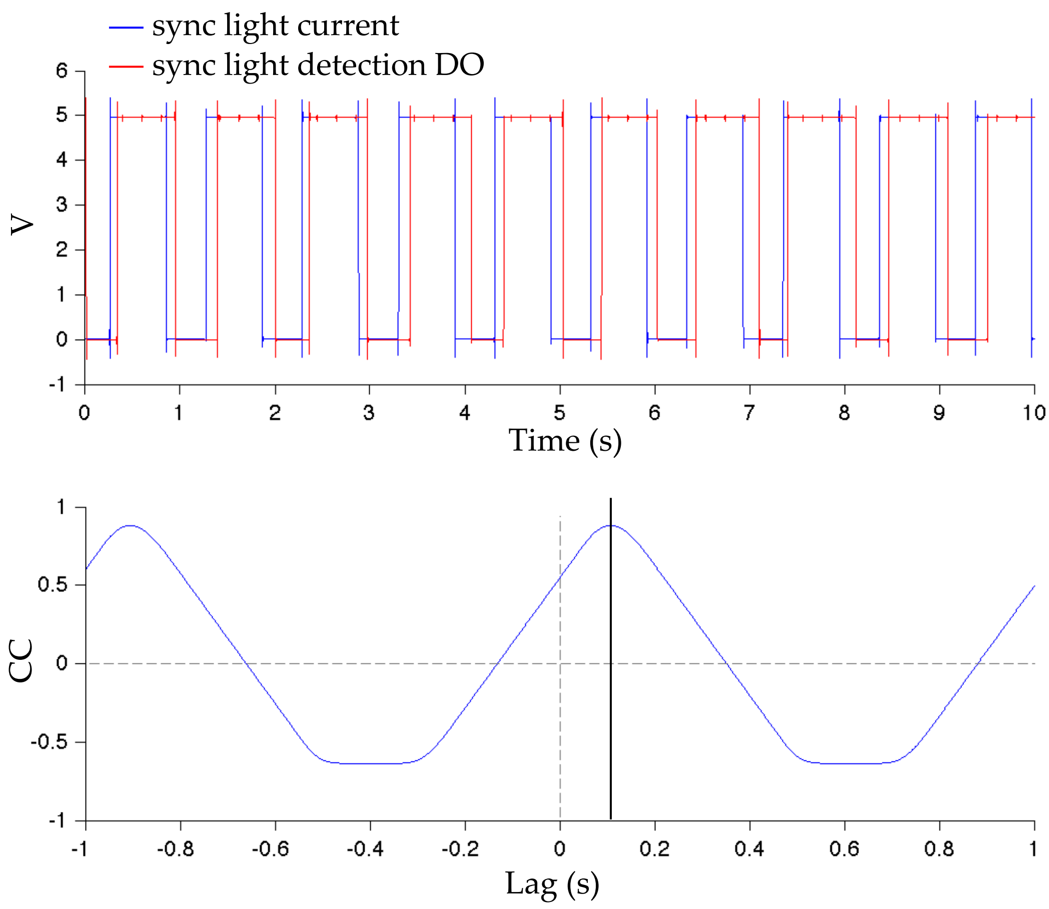


Figure 14: *Temporal correlation between monitored current of a flashing synchronization LED and digital signals emitted by Spotter detecting the light.* Top: *Time course of the signals. The digital signal of Spotter lags behind the actual current applied to the LED.* Bottom: *Cross-correlation of the two signals. The Spotter signal lags by about 109 ms.*

6 Discussion

In the current state of development Spotter allows robust video tracking of laboratory rodents. While designed for animals carrying hardware (i.e. head mounted LEDs), tracking is generalizable for any object which can be distinguished within the HSV color space. Furthermore, multiple objects can be tracked independently at the same time. By reducing the search window dynamically, additional features add little computation time and the tracking delay is dominated by fixed components of the analysis pipeline. In future iterations of the software, the option to integrate modules with new features and tracking algorithms could greatly extent versatility and tracking performance.

For any system with a low sampling rate, delays are unavoidable. This is most obvious for binary signals like flashing LEDs, where the timing of frame acquisition to signal onset is critical. In the worst case, the LED lights up during frame acquisition, leading to incomplete exposure and missing the light although already visible in the frame, adding another full frame length to the detection time (e.g. 33 ms at 30 fps). In the best case, the signal light is active just before frame acquisition. This jittering can only be improved by an increase in frame rate, or by triggering frame acquisition on external signals correlated with the feature of interest (e.g. current applied to the LED).

For continuous signals however, this limitation is of less importance. Here, the buffer and processing delay is critical. The reported lag times are for an older version of the program not using adaptive search windows. One can expect from the decrease in processing time by about 10 ms the cross-correlations to improve accordingly. However, this still leaves about 100 ms of delay. While experiments to pinpoint the origin of the delay are underway, I can speculate on the probable causes.

Using inexpensive video acquisition hardware like an off-the-shelf webcam is cheap, but comes with engineering trade-offs. To allow transmission of high-resolution video over a bandwidth limited port like USB 2.0, the video stream has to be encoded on the remote device. For this purpose, a framebuffer on the camera is likely holding at least one frame for subsequent encoding. At least one more framebuffer is employed by the video acquisition library, followed by a decoding buffer. Each of the buffers can increase the time between acquisition on the camera sensor to processing availability by a full frame length, e.g. 33 ms. Thus the buffering of three frames along the acquisition pipeline would fully explain the delay. Finding the location of the buffers could provide means to mitigate the lag. It is open whether the use of Python bindings versus the C/C++ interface introduces additional buffering delays, or whether buffering is dependent on the video acquisition backend used by OpenCV.

While minor compared to the lag introduced by processing and data acquisition, the serial connection to the microcontroller can be improved. Several Arduino-compatible microcontrollers exist that use a 32 bit ARM based chip, instead of the 8 bit Atmel. These typically have much faster data transmission with lower latencies. Additionally, they are available with built-in DACs (e.g. Arduino Due), reducing the complexity of the hardware. The controller choice aside, the current protocol transmits a lot of overhead per packet. Two DAC channels can be served by a single packet and updating up to 8 digital pins requires a single byte of data. Using the padding bits and reducing the address/device type space to 2 bits each can free enough bits to allow packet length encoding, reducing the overall message length to 3 or 4 bytes. Combining digital and analog updates can result in a total message length of 5 bytes, at the cost of protocol flexibility.

The number of analog and digital channels used can easily be extended. Both the serial and SPI connection have sufficient bandwidth for dozens of devices and the communication protocols are designed with wide address spaces. Furthermore, the Arduino reports its hardware capabilities to the Spotter software. Adding new channels only requires adding the controlling pin to an array in the Arduino code and may be of use when tracking multiple objects.

The widespread introduction of multiplexed high-channel electrophysiology systems based on Intan hardware introduces new challenges for concurrent recording of electrophysiology and external signals, as these systems employ high pass filters (typically 0.1 Hz), rendering DC signals like the slowly changing analog output and rarely changing digital signals useless. To overcome the high-pass filter, multiple future options are under investigation. Primarily, signals can be alternated quickly using bipolar DACs. However, this would increase the difficulty of using these signals externally. Alternatively, encoding analog signals with digital patterns could sidestep high pass filtering using pulse modulation techniques (e.g. PWM, PCM, PDM). But all similarly increase the difficulty of using signals by components other than for mirroring. Likely, adding additional DAC channels to allow both, pure DC signals and coded signals to be emitted concurrently will be required. Additionally to the existing output states and devices, a useful addition would be the ability to encode arbitrary numerical variables as digital, and to a lesser extent analog, signals. As an example, encoding frame IDs as digital PCM or PWM patterns would allow precise alignment of video to electrophysiology recordings during offline analysis.

While the interface in general will require a lot of continued effort, a main step towards increased performance is to separate rendering and processing. Currently, these two are performed sequentially. Any delay in rendering can delay the processing of frames, including frame skips, reducing the temporal resolution and precision. As with the `Writer` module the `Grabber` could be placed into a separate process, or at least a separate thread. This can be generalized to a producer-multiple consumer architecture. Having a separate thread/process for these modules would allow to move between Python and C modules, the latter could provide a great boost to performance while keeping the ease-of-use of Python for the remainder of the code. Using C modules would also allow to integrate libraries to interface non-standard cameras that require vendor specific software development kits, like most cameras better suited for realtime computer vision than webcams. These can achieve higher frame rates (100 Hz and higher) at low latencies and well defined frame acquisition timings. Together with hardware triggering of shutter and sensor signal conversion, this could allow extremely low latency, high speed position tracking with the existing program.

Acknowledgements

I greatly appreciate all the help I received in Prof. Buzsàki's group working on this project. Especially, I want to thank Eran Stark for technical advice. He created the figures for spatial and temporal correlation. And I want to thank Lisa Roux for suggestions and her thorough continuous beta (or rather alpha) testing. The example screenshot in figure 10 is made with one of her recordings. Sven Ehrlich provided a long-term rental of his Arduino when mine was left behind in New York...

Appendix

#	Part name	Part #	Price [$]
1	Arduino Uno R3	78T1601	25.00
1	USB A to B cable	PSG90003	2.00
2	MCP4921 12 bit DAC	08J8743	4.00
2	0.1 µF ceramic capacitor	97M4140	0.20
2	10 µF tantalum capacitor	25M8461	1.00
1	Microsoft LifeCam Studio 1080p HD Webcam	–	50.00
Total			**82.20**

Table 1: *Bill of materials for a basic 2-channel implementation of the Spotter hardware. Part numbers are examples, retrieved July 2013 from newark.com. Does not include the PC required.*

References

[1] Antal Berényi, Mariano Belluscio, Dun Mao, and György Buzsáki. Closed-loop control of epilepsy by transcranial electrical stimulation. *Science (New York, N.Y.)*, 337(6095):735–7, August 2012.

[2] Gabrielle Girardeau, Karim Benchenane, Sidney I Wiener, György Buzsáki, and Michaël B Zugaro. Selective suppression of hippocampal ripples impairs spatial memory. *Nature neuroscience*, 12(10):1222–3, October 2009.

[3] Satoshi Suzuki and KeiichiA Be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, April 1985.

[4] Tiobe. TIOBE Programming Community Index for June 2013, 2013.