applied optics

Real-time optical imaging acquisition and processing in Python: a practical guide using CAS

Michael R. Hughes

Applied Optics Group, School of Engineering, Mathematics and Physics, University of Kent, Canterbury, UK (m.r.hughes@kent.ac.uk)

Received 8 April 2025; revised 12 June 2025; accepted 17 June 2025; posted 17 June 2025; published 9 July 2025

Real-time data acquisition and processing is an important step in the development of new approaches to optical imaging in research laboratories. Python is increasingly used for scientific computing and allows for the straightforward application of artificial intelligence models using popular frameworks such as PyTorch. However, achieving high-speed image capture and processing in real time is challenging and requires extensive development work, a particular problem for academic labs where research teams may lack specialist expertise in software development. This note provides guidelines for achieving high performance in Python for optical imaging applications and introduces an open-source framework "CAS" for rapid prototyping of imaging system software. CAS includes a hardware abstraction layer for cameras, a ready-made GUI, which can easily be customized, as well as support for using multiple CPU cores for parallelism. By providing an open-source and flexible Python-based solution, CAS can support research teams to more quickly develop real-time imaging systems.

https://doi.org/10.1364/AO.564458

1. INTRODUCTION

Optical imaging system development is a major topic of research, particularly for applications in biosciences and medicine. Work ranges from the development of new imaging concepts to the customization or adaptation of systems for specific applications. A common theme is the need to acquire images (or image-like data) from a camera or beam-scanning system, perform either single- or multi-frame processing, and then display the processed images to the user. While proof-of-concept work may simply capture and store images for subsequent processing and analysis, even in the basic research phase of development, it is often useful to perform image processing and display results live in a graphical user interface (GUI), both as a more convincing demonstration of the technology's potential and to allow alignment and optimization of the optical system and to speed up the process of testing.

A variety of software platforms and programming languages are used in optical imaging system development. While developers of commercial systems will typically adopt low-level languages such as C++, particularly for later-stage demonstrators and product development, early-stage prototypes in academic labs will often avoid C++ due to the long development time and technical expertise required. LabVIEW (National Instruments) is a popular alternative due to the very short time required for novices to develop simple imaging applications and a graphical programming style that is well-suited to beginners. While LabVIEW allows for the rapid development of simple GUIs for hardware interaction and control, developing more complex and scalable applications requires greater programming expertise, and there is also a requirement to pay recurring software licensing costs, particularly if additional modules beyond the base version are required. MATLAB (MathWorks) is also sometimes used for simple GUIs and benefits from a very large range of toolkits for image processing and related operations, but as a scripting language, it is not well-designed for building real-time GUIs, and it too incurs licensing costs.

Python is widely used for post-processing and data analysis across a wide range of scientific fields, including imaging, and is now dominant in a number of areas, particularly deep learning. There is a huge ecosystem of packages available to handle scientific applications, and bindings are available for major open-source libraries such as OpenCV [1]. As of 2024, Python is the most popular language on GitHub, and a large fraction of researchers joining labs will have had at least some exposure to Python during their education, making it a natural choice for a lab's codebase. However, it is comparatively less widely used for real-time image acquisition and display. As an interpreted language, it is sometimes considered slow, and the global interpreter lock (GIL) makes use of multiple cores in real-time applications reasonably complex. Nevertheless, with the correct approach and architecture, performant Python user interfaces for real-time image capture, processing, and display can be developed. This has the advantage that code developed in Python for offline processing can then easily be integrated into

online systems, provided it is optimized for speed through the use of external libraries such as NumPy [2] and SciPy [3].

There are several examples of open-source Python applications for real-time optical image capture, particularly for microscopy, such as *Tormenta* [4] for fluorescence microscopy, which is over 10 years old, and *ImSwitch* [5], which allows acquisition from cameras and control of hardware such as laser scanners and spatial light modulators. For applications that closely match the intended use cases, they are potentially a convenient solution. However, they are more complex to modify or customize for other types of imaging without extensive knowledge of both Python and the specific open-source package.

This note provides guidance on how Python GUI development for imaging applications can be approached by those who are not experienced in this field. It gives specific details of an open-source framework, CAS [6], which can either be used as an example GUI or as a starting point to accelerate the development of software applications for optical imaging, allowing researchers with only basic expertise in Python to produce functional and high-performance software interfaces. General guidelines for a proven approach to real-time image acquisition and processing in Python are first provided, and then the CAS framework is introduced with a discussion of how it implements these principles to provide a high-performance graphical user interface (GUI).

2. PRINCIPLES OF REAL-TIME ACQUISITION AND PROCESSING

In this section, a high-level overview of one possible set of approaches to image acquisition and processing is presented and summarized in Fig. 1.

The entry point to the program should be a class that implements the main user interface (UI). This handles all user interaction, including both display of images and handling user input such as changes to imaging settings. Acquisition of camera images (or equivalent data) should be handled by an *image acquisition thread* separate from the UI thread, e.g., by having an *image acquisition class* that subclasses *threading.Thread*. This can be started by the UI thread and run on the same core as the UI thread; data acquisition is an example of an input/output (I/O) bound operation, meaning that the thread will spend chunks of time waiting for data, during which other operations in the UI thread, such as handling user interactions, can continue. This thread should grab images from the camera and place them in a queue. Individual images can be stored as simple NumPy arrays, or as more complex data structures as required. Python queues can be used as a circular buffer by monitoring the queue size and removing the oldest element once the queue reaches the maximum desired size. The optimal queue size involves a trade-off between the importance of not dropping frames when temporary slow-downs occur, for example, due to operating system operations, and avoiding visual latency.

The code for handling data transfer can be separated from the camera-specific code to capture images and perform other hardware operations; this will allow the use of different cameras with minimal modification to the acquisition logic. One way to achieve this is to create a class for interfacing a particular camera with a set of standard methods such as *open_camera* and *get_frame*. The image acquisition thread can create an instance of the camera class when started and store a reference to this camera. The UI thread can then directly access methods of the camera class via the stored reference, or a messaging system can be implemented if direct access is not desired. The image acquisition thread should keep a copy of the latest image separate from the queue, so that the UI class can periodically, using a timer or event loop, update the GUI with the latest raw image without pulling images off the *raw image queue*.

The code for processing raw images should run in either a second thread on the same core as the UI and image acquisition threads or as a thread on a separate core. The latter is more complex to handle but may be necessary to improve performance for more intensive operations. In both cases, the thread can be created in the UI thread, and the queue for raw images can be directly shared between the acquisition and processing threads; the processor thread can then monitor the queue for raw images, process them once available, and then place the processed images in a second queue. The UI thread can then monitor this *processed image queue*, either using a timer or event loop and update the displayed image. The algorithms for processing images should be implemented as a separate class from the thread handling messaging and queues, allowing different algorithms to be exchanged as required.

A. Processing on a Separate CPU Core

In some cases, it is desirable to run the processing code on a separate core, as this can lead to a significant increase in speed in some cases. In order to do this, the Python global interpreter

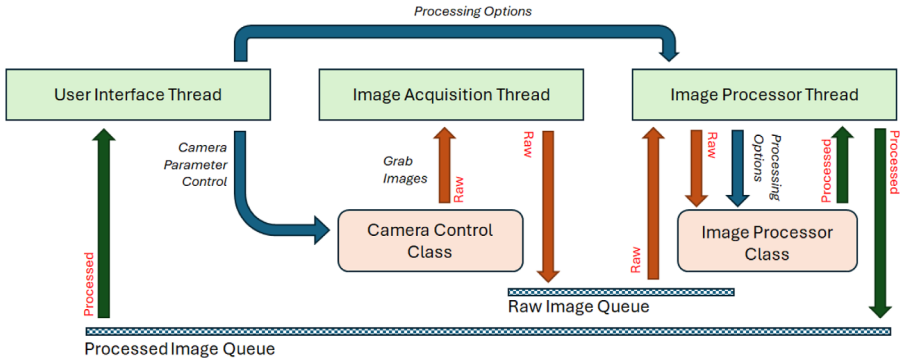

Fig. 1. Schematic of high-level concept for acquisition and processing.

lock (GIL) limitation must be overcome. In essence, a separate instance of the Python interpreter must be started on the second core using the *multiprocessing* module purely to run the image processing code. Inter-process communication between the user interface and the processing code, for example, to send processed images for display and to change parameters that control the processing, must then be handled via messaging systems, which can be implemented using queues from *multiprocessing.queue*. This additional complexity should only be introduced when the speed gain is necessary for the correct functioning of the software. There is also, depending on the means by which image communication between cores is handled, the potential to introduce additional overheads which actually reduce the speed of processing.

To illustrate how this may be implemented and the impact of these effects, a pair of example scripts are included in the repository accompanying this note, which are similar to the minimal working example GUI. One script uses threads in a single process, while the second performs the processing in a separate process. It should be noted that a significant increase in processing speed will likely only be achieved where the GUI or image acquisition presents a significant CPU load, and where the processing does not employ external libraries that make use of multiple cores in any case. For example, the minimal GUI discussed above does not present a significant overhead, and the processing already runs at close to the best-case time (i.e., a simple loop with no GUI), so there would be no significant improvement expected in this case. Therefore, in the scripts comparing single- and multiple-core operation, CPU-intensive operations (multiple square root calculations) were added to both the GUI thread and the image processing thread. As a result, when the Gaussian filter size was set to 15 px, the frame rate was only approximately 4 fps using a single core and better than 6 fps using two cores.

Furthermore, sending large images over queues between cores (not between threads on the same core) may become a limiting factor on the displayed frame rate, particularly for large images or high frame rates. In this case, shared memory is a more efficient solution, provided that the dropping of some frames is acceptable. An area of shared memory is defined, and the processed images are written to this memory, which can then be accessed directly by the user interface thread to update the displayed image. A particular use-case of this method is for applications such as mosaicking, where a large processed image (the mosaic) is being constantly updated with the addition of new image frames, but the GUI requires only periodic updates (i.e., it does not require a new processed mosaic with the addition of each frame). Updating the mosaic in an area of shared memory can therefore be an effective solution.

B. Optimizing Processing Algorithms for Real-Time Use

The field of algorithm optimization in general is beyond the scope of this note, but there are some considerations specific to optical imaging processing. Generally, all processing of images should be performed using external libraries for speed and to avoid the need to re-implement well-tested algorithms. Images are naturally implemented as arrays, which can be manipulated directly using the NumPy package [2]. The Pillow package [7] provides higher-level support for working with images, including loading and saving from the disk in standard formats, and Pillow images can be converted to and from NumPy arrays straightforwardly, although with some time cost. SciPy [3] implements a wide range of useful functions for image processing, such as filtering and transforms, and is directly compatible with NumPy arrays. OpenCV [1], which has a Python wrapper, also provides a wide range of functionality focused on machine vision but with many fast functions relevant to general image processing. These four packages are sufficient for many image processing requirements. Machine learning frameworks, such as PyTorch [8], are also straightforward to use with Python. With all these packages, the aim should be to achieve as much of the processing with as few calls to external functions as possible. To avoid unnecessary optimization, which should be avoided during rapid prototyping, profiling tools can help identify bottlenecks in the code.

For cases where external libraries do not provide all the required functionality, one possibility is to write the code in C++, compile this into a dynamic link library (DLL), or equivalents on the non-Windows system, and then call this library from Python. However, similar performance can often be obtained using the Numba package [9], which provides a just-in-time (JIT) compiler for a specific subset of Python functions and can be used to accelerate a subset of Python code, although with some restrictions and the need to profile code to check that speedups are obtained in practice. An example of using Numba JIT for optical imaging can be seen in the PyFibreBundle package [10], where it allows for state-of-the-art performance for triangular linear interpolation between fiber bundle cores. Finally, although a detailed discussion is again beyond the scope of this note, packages such as CuPy [11] can be used to offload processing to a graphics processing unit (GPU) and can achieve substantial speed-up without the developer needing detailed knowledge of GPU architectures or CUDA.

C. Developing the GUI

While proof-of-concept systems do not require polished GUIs, a functional and intuitive system is useful both for system demonstration and to allow other research team members or collaborators to make use of the imaging device. A range of GUI frameworks can be used in Python, including tkinter [12] and the more feature-rich Qt [13]. Neither of these libraries has extensive support for displaying live scientific images, unlike, for example, LabVIEW, which provides built-in image display options with a broad range of features such as pixel inspection, regions of interest, overlays, and color maps. A *QGraphicsView* can be used in Qt GUIs but would require significant amounts of development work if more advanced visualization is required. Matplotlib [14] allows the display of both images and a large range of other plots and allows for extensive customization, and while not originally intended for the live display of video, it may be useful in many applications, particularly where a mix of images and scientific plots need to be displayed. OpenCV [1] can be used to generate a highly performant image display window, but this is not integrated into a wider GUI and does not provide any additional features. Other Python packages are

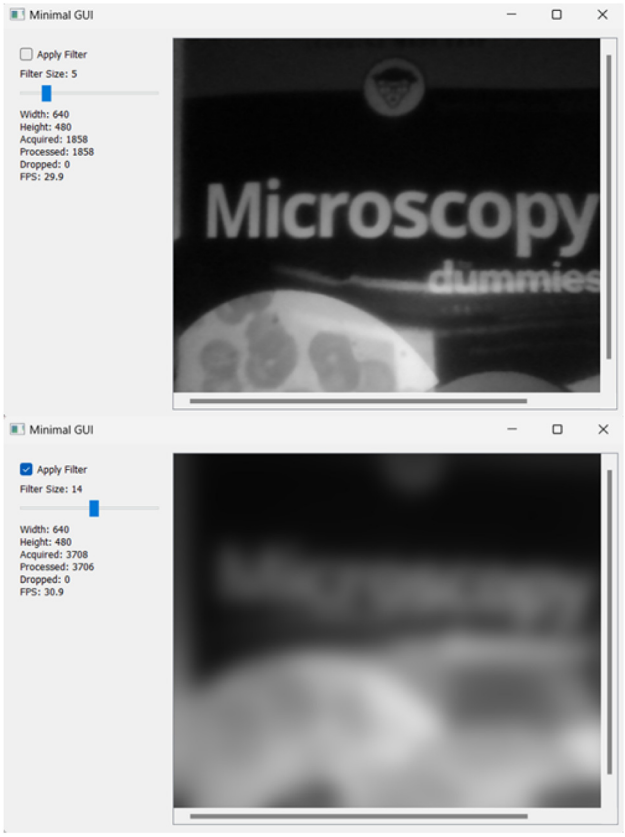

Fig. 2. Screenshots of minimal working GUI, showing the live image from the webcam without processing applied (top) and with processing applied (bottom).

available specifically for image display in Qt GUIs, including PyQtGraph [15] and ImageDisplayQt [16], the latter of which was developed specifically for CAS and offers features including pan and zoom, pixel inspection, regions of interest, and a system for drawing overlays.

D. Minimal Working Example

The repository accompanying this note [17] includes *minimal_gui.py*, a minimal working GUI that illustrates the principles described above and could be used as the basis of more complete GUIs rather than using CAS. This minimal GUI, which contains approximately 200 lines of code, includes the GUI class which inherits from *QMainWindow* to create a Qt GUI as well as two threads, one to acquire images from a web camera and a second to perform image processing (application of a Gaussian filter). Images are stored in a *raw_image_queue* and a *processed_image_queue*, and processed images are displayed using a *QGraphicsView*. Screenshots of the minimal GUI are shown in Fig. 2.

3. DEVELOPING A USER-INTERFACE USING CAS

CAS implements the approach described above, alongside a simple user interface and a library of functions for common operations such as saving images to disk and recording. In particular, it is possible to record raw or processed images directly to disk as video files or TIFF stacks or stream raw images to a memory buffer, which can later be saved. CAS allows the processing of individual images or a set of images, and the processing can run on a separate core if required.

GUIs are created using the Qt library, which provides a rich set of widgets, and image display is via an *ImageDisplayQT* widget. By default, this allows the user to inspect individual pixels, draw regions of interest to obtain mean, maximum and minimum pixel values, and zoom and pan. Custom GUIs are created by sub-classing the base class of CAS (*CAS_GUI*) and overriding methods and attributes as needed. The GUI includes a simple system of expanding menus, and it is straightforward to add additional widgets to control cameras or processing settings or entirely new menus as needed. In many cases, it is possible to have a functioning system running with only a few tens of lines of new code, while more significant customization is possible by over-riding a larger number of methods.

Most processing can be implemented by sub-classing the *ImageProcessorClass* and over-riding the *process* method, which accepts a raw image, or set of raw images, as a NumPy array and returns a processed image, also as a NumPy array. For example, to apply a simple Gaussian filter with $\sigma = 5$ px using the SciPy *gaussian_filter* method:

```
from scipy.ndimage import gaussian_filter

class Filter(ImageProcessorClass):
    def process(self, inputFrame):
        return gaussian_filter(inputFrame,
sigma=5)
```

The CAS base class, *CAS_GUI*, is then sub-classed:

```
from cas_gui.base import CAS_GUI
class ExampleGUI(CAS_GUI):
  def__init__(self):
    super().__init__()
        self.processor=Filter
```

where in this case, we have only made one change: to assign the image processor class. These classes can then be placed in a single file, together with some imports:

```
from PyQt5.QtWidgets import *
from    cas_gui.threads.image_processor_class
import ImageProcessorClass
from scipy.ndimage import gaussian_filter
```

and the code to launch the example GUI:

```
if __name__=="__main__":
    app=QApplication([])
    window=ExampleGUI()
    window.show()
    app.exec()
```

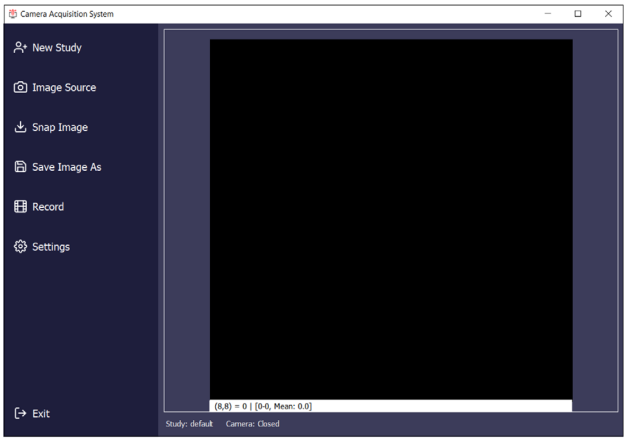

Fig. 3. CAS GUI default appearance.

This code is included in the repository accompanying this note [17] as *cas_example.py*. When executed, this will display a GUI, as shown in Fig. 3. Further examples are available in a copy of the CAS package, which is also contained in the repository, and the CAS documentation provides further details [18].

A. Camera Interfaces Using CAS

CAS provides a common interface for cameras, allowing camera interface classes for specific cameras to be exchanged without altering other GUI logic. Several camera interfaces are included in the CAS package, including commonly used cameras such as those from Thorlabs. In general, a user will need to implement an interface for the specific camera they wish to use. At a minimum, a camera interface must provide methods to open the camera, grab images, and close the camera. Other camera functions, such as control of exposure or gain, can be implemented as required. CAS provides a set of standard functions for this purpose, which can be implemented where the functionality is required. For example, to add exposure control, the developer first implements the *is_exposure_enabled()* function to return *True*; this informs CAS that the exposure on this camera can be controlled, and the exposure controls in the GUI will be enabled. The developer should then implement *set_exposure()*, which accepts a float and sets the exposure to that value, and *get_exposure()*, which should return the current exposure. Finally, *get_exposure_range()* should be implemented and return a tuple of the lowest and highest possible exposures; these control the range of the exposure control slider in the GUI. Full details of how to implement exposure and other property controls for a new camera, such as gain and frame rate, are provided in the CAS documentation, and several examples are included with CAS, including for cameras from FLIR and Thorlabs.

When implementing a new camera, it is convenient to implement a test routine to allow bugs to be identified and fixed more easily than when using the GUI. This can be achieved by adding a block at the end of the file using the standard *if __name__ == "__main__"*: Python technique. If the Python file containing the camera class is run as a script, this will then run the code in the block. An example of how to use this to test the *WebCamera* camera interface is provided in the repository accompanying this note.

The concept of a camera can be extended to laser scanning systems or other pseudo-cameras quite straightforwardly. For example, the *File* camera interface treats an image file as an image source; this allows a GUI created for real-time processing to also be used for offline processing of images, avoiding the need for duplicate GUIs for this purpose.

4. PERFORMANCE

To illustrate that GUIs based on the principles described here offer a high level of performance, a comparison was performed between the minimal GUI described above, CAS, and a simple loop performing only the processing. Testing was performed on a Windows 10 laptop (11th Gen Intel Core i5-1135G7, 4 core, 2.4 GHz processor, with 16 GB RAM) with Python v3.10.14. The application of a Gaussian filter was used as an exemplar image processing operation. Using the SciPy v1.13.1 *gaussian_filter* function with a sigma of 25 pixels, applied to an 8 bit monochrome image of 640 × 480 pixels, when running in a loop, the average frame rate (i.e., frames processed per second) was approximately 20 Hz. In the minimal GUI described above, including acquisition from a web camera, processing, and display of the images in the GUI, the frame rate typically fluctuated between 18 and 20 Hz, suggesting only a small additional overhead from the GUI. In CAS, running the processing on the same processor as the acquisition and user interface leads to a slightly slower frame rate of typically 17–18 Hz, likely due to the increased overhead of the more complex GUI, while running the processing on a separate core restores a frame rate of typically 19 Hz. These values fluctuate with system load and exact settings in the GUI, such as the update rate of the displayed images, but indicate that the GUIs can potentially have only a small impact on the achievable processing speed.

A. Comparative Performance

To provide an indicative comparison of the performance of Python GUIs with MATLAB and LabVIEW, a set of programs was created to perform a 1D Fourier transform on images. For each language, two programs were created. The first simply repeatedly processes a random image in a loop, with no image queues, display, or other GUI elements, in order to determine the absolute best-case speed. The second is a minimal GUI that includes separate acquisition and processing loops and the display of the processed image. Copies of all programs are available in the repository.

The LabVIEW GUI uses two parallel loops, one of which places copies of a randomly generated image into a queue at the required frame rate and a second which pulls images out of the queue, performs the processing, and displays the image. Since LabVIEW handles multi-threading implicitly, these two loops may run on different threads or processes. Since MATLAB does not provide a simple way to produce multi-threaded GUIs out-of-the-box, the MATLAB GUI implementation uses cell arrays in place of queues, and timers to call functions for simulated image acquisition and processing. A minimalist GUI was created in Python, similar to the one described above, and a fully featured GUI was created using the CAS framework.

Table 1. Effective Frame Rates for 1D Fourier Transform of Images

	Frame Rate (Hz)		
Image Size (px)	512 × 512	1024 × 1024	2048 × 2048
MATLAB, no GUI	370	101	25
MATLAB, GUI	30	12	4
LabVIEW, no GUI	250	40	9
LabVIEW, GUI	222	37	8
Python, no GUI	225	54	12
Python, minimal GUI	180	52	12
Python, CAS	200	52	12

All programs perform the same basic operation. A square image is generated using random double-precision floats to simulate the acquisition of an image from a camera. Processing consists of taking a 1D FFT of the image and then taking the absolute value. In the GUIs, the processed image is then displayed live, with the display updated every 50 ms (20 fps), regardless of the underlying processing frame rate. Effective maximum frame rates for different image sizes for each GUI and no-GUI loop are shown in Table 1. Testing was conducted on a Windows 10 PC with an Intel Core i7-7700 quad-core CPU and 24 GB RAM using LabVIEW 2016 and MATLAB R2016a.

This comparison should not be taken as a general indication of the relative speeds of GUIs created using the different software platforms, as performance will vary depending on the application and the exact implementation of the GUI. Rather, the comparison serves to illustrate that Python-based GUIs are not inherently slow compared with other common approaches and hence are suitable for many applications in the imaging laboratory. It can be seen that the simple loop in MATLAB provides the highest speed, while the MATLAB GUI performs very poorly (although further optimization or redesign of the MATLAB GUI may be possible), while the Python and LabVIEW implementations are broadly comparable for all image sizes. It should be noted that the GUI based on the CAS framework is by far the most feature-rich of the tested options (with the ability to inspect the image pixels or save images, for example) and requires only 17 lines of code.

5. CONCLUSION

This note provides a high-level conceptual framework for developing imaging system GUIs in Python, particularly aimed at rapid prototyping of research systems. These principles can be used by teams with a good working knowledge of Python to develop systems from scratch. CAS is provided as an example of how a simple GUI can be built using these principles. As an alternative to developing an original GUI, CAS can be used as a framework to allow rapid development of imaging system software without extensive knowledge of Python.

Funding. Biotechnology and Biological Sciences Research Council (BB/X003744/1); Engineering and Physical Sciences Research Council (EP/X000125/1, EP/R019274/1); Royal Society.

Acknowledgment. Thanks to the members of the Applied Optics Group who have used and provided feedback on the CAS framework.

Disclosures. The author declares no conflicts of interest.

Data availability. A copy of the CAS Python package was archived at the time of submission [17]. No data were generated or analyzed in the presented research.

REFERENCES

1. OpenCV team, "Home," OpenCV, 2025, [accessed 2 April 2025] https://opencv.org/.
2. C. R. Harris, K. J. Millman, S. J. Van Der Walt, *et al.*, "Array programming with NumPy," Nature **585**, 357–362 (2020).
3. P. Virtanen, R. Gommers, T. E. Oliphant, *et al.*, "SciPy 1.0: fundamental algorithms for scientific computing in Python," Nat. Methods **17**, 261–272 (2020).
4. F. M. Barabas, L. A. Masullo, and F. D. Stefani, "Note: Tormenta: an open source Python-powered control software for camera based optical microscopy," Rev. Sci. Instrum. **87**, 126103 (2016).
5. X. Casas Moreno, S. Al-Kadhimi, J. Alvelid, *et al.*, "ImSwitch: generalizing microscope control in Python," J. Open Source Softw. **6**, 3394 (2021).
6. M. Hughes, "Camera acquisition systems (CAS) and CAS GUI," GitHub, 2025 [accessed 6 April 2025], https://github.com/MikeHughesKent/cas.
7. Fredrik Lundh and contributors, "Pillow (PIL Fork) 11.3.0 documentation," Pillow, 2011 [accessed 4 April 2025], https://pillow.readthedocs.io/en/stable/.
8. A. Paszke, S. Gross, F. Massa, *et al.*, "PyTorch: an imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems*, Vol. **32** (2019), pp. 1–12.
9. S. K. Lam, A. Pitrou, and S. Seibert, "Numba: a LLVM-based python JIT compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (2015), pp. 1–6.
10. M. R. Hughes, "Real-timing processing of fiber bundle endomicroscopy images in Python using pyfibrebundle," Appl. Opt. **62**, 9041–9050 (2023).
11. CuPy, "NumPy/SciPy-compatible array library for GPU-accelerated computing with Python," Preferred Networks, Inc., 2025 [accessed 2 April 2025], https://cupy.dev/.
12. tkinter, "tkinter—Python interface to Tcl/Tk," Python Software Foundation, 2025 [accessed 1 April 2025], https://docs.python.org/3/library/tkinter.html.
13. Qt Group, "Qt | Tools for Each Stage of Software Development Lifecycle," The Qt Company, 2025 [accessed 3 April 2025], https://www.qt.io/.
14. The Matplotlib development team, "Matplotlib: visualization with Python," Matplotlib, 2025 [accessed 4 April 2025], https://matplotlib.org/.
15. PyQtGraph, "Scientific Graphics and GUI Library for Python," PyQtGraph, 2021 [accessed 2 April 2025], https://www.pyqtgraph.org/.
16. M. Hughes, "ImageDisplayQT: a PyQT widget for scientific image display in Python GUIs," version 1.1.0, GitHub, 2024 [accessed 2 April 2025], https://github.com/MikeHughesKent/ImageDisplayQT.
17. M. Hughes, "Real-time optical imaging acquisition and processing in Python: a practical guide using CAS: code repository," figshare, 2025, https://doi.org/10.6084/m9.figshare.28736513.v2.
18. "CAS and CAS GUI documentation," CAS-GUI, 2024 [accessed 5 April 2025], https://cas-gui.readthedocs.io/en/latest/.