PyQtGraph - High Performance Visualization for All Platforms

Ognyan Moore[‡*], Nathan Jessurun[§], Martin Chase[§], Nils Nemitz[§], Luke Campagnola[¶]

Abstract—PyQtGraph is a plotting library with high performance, cross-platform support and interactivity as its primary objectives. These goals are achieved by connecting the Qt GUI framework and the scientific Python ecosystem. The end result is a plotting library that supports using native python data types and NumPy arrays to drive interactive visualizations on all major operating systems. Whereas most scientific visualization tools for Python are oriented around publication-quality plotting and browser-based user interaction, PyQtGraph occupies a niche for applications in data analysis and hardware control that require real-time visualization and interactivity in a desktop environment.

The well-established framework supports line plots, scatter plots, and images, including time-series 3D data represented as 4D arrays, in addition to the basic drawing primitives provided by Qt.

For datasets up to several hundred thousand points, real-time rendering speed is achieved by optimized interaction with the Python bindings of the Qt framework. For enhanced image processing capabilities, PyQtGraph optionally integrates with CUDA. This ensures rendering capabilities are scalable with increasing data demands. Moreover, this improvement is enabled simply by installing the CuPy[1] library, i.e. requiring no in-depth user configurations.

PyQtGraph provides interactivity not only for panning and scaling, but also through mouse hover, click, drag events and other common native interactions. Since PyQtGraph uses the Qt framework, the user can substitute their own intended application behavior to those events if they feel the library defaults are not appropriate. This flexibility allows the development of customized and streamlined user interfaces for data manipulation.

The included parameter tree framework allows straightforward interactions with arbitrary user functions and configuration settings. Callbacks execute on changing parameter values, even asynchronously if requested.

An active developer community and regular release cycles indicate and encourage further library development. PyQtGraph's support cycle is synchronized with the NEP-29[2] standard, ensuring most popular scientific python modules are continually compatible with each release.

PyQtGraph is available through pypi.org (https://pypi.org/project/pyqtgraph/), conda-forge (https:/ anaconda.org/conda-forge/pyqtgraph) and GitHub (https://github.com/pyqtgraph/pyqtgraph).

Index Terms—Visualization, Qt, NumPy, PyData, Python

Introduction

The benefits of interactive exploration of scientific data were recognized as soon as computer systems gained graphical displays. While early implementations like the PRIM-9 system[3] of the Stanford Linear Accelerator Center were only available to large installations, more affordable microcomputers soon found their place in smaller laboratories also[4][5], controlling experiments and recording data.

Software packages designed to acquire and process this data soon appeared, with MATLAB[6] and LabView[7] both implementing graphical representation of data from their very first versions. The latter was designed to enable data acquisition, processing and visualization all in the framework of a single program. This approach remains common in fields like statistics where the tools for interaction with data are reasonably well-defined. In other areas, the advent of high-level programming languages like Java and Python has enabled researchers to create the tools for their specific needs with reasonable time investment. This is facilitated by a continuously growing open-source infrastructure that provides resources addressing anything from mathematical methods[8] to full-scale laboratory data infrastructure[9], [10].

With less need to recreate existing solutions, it becomes feasible to implement software aiming to reduce turn-around times of iterated experiments: A traditional view of the scientific method envisions a sequence of detailed experiment design, pain-staking note-taking, followed by an exhaustive evaluation resulting in a revised experiment. However, when experiments can be optimized over a wide parameter space, the evaluation quickly becomes the dominant factor. Even for established experimental parameters, external factors such as degraded performance of equipment result in a significant loss of time if they are discovered only in subsequent evaluation.

The solution is to provide immediate feedback to the researcher throughout the experiments, and data visualization has long proven its effectiveness in this regard [Friendly2008]. A challenge lies in providing tools for a detailed inspection of interesting data while new information continues to arrive at rates that for extreme cases are counted in Gb/s even after preselection[11]. These tools also need to provide the flexibility to handle data that falls outside the range expected in design, as this is the most likely to indicate failures or to provide the sought-after discovery.

Here we present a visualization library created with these goals in mind. Although written in Python to allow for easy expansion, a close integration with the cross-platform Qt UI framework[12] it provides the capability to interactively handle datasets of hundreds of thousands of points, or live representation of high-resolution camera data.

* *Corresponding author: ognyan.moore@gmail.com*
‡ *Hobu Inc.*
§ *Unaffiliated*
¶ *Allen Institute*

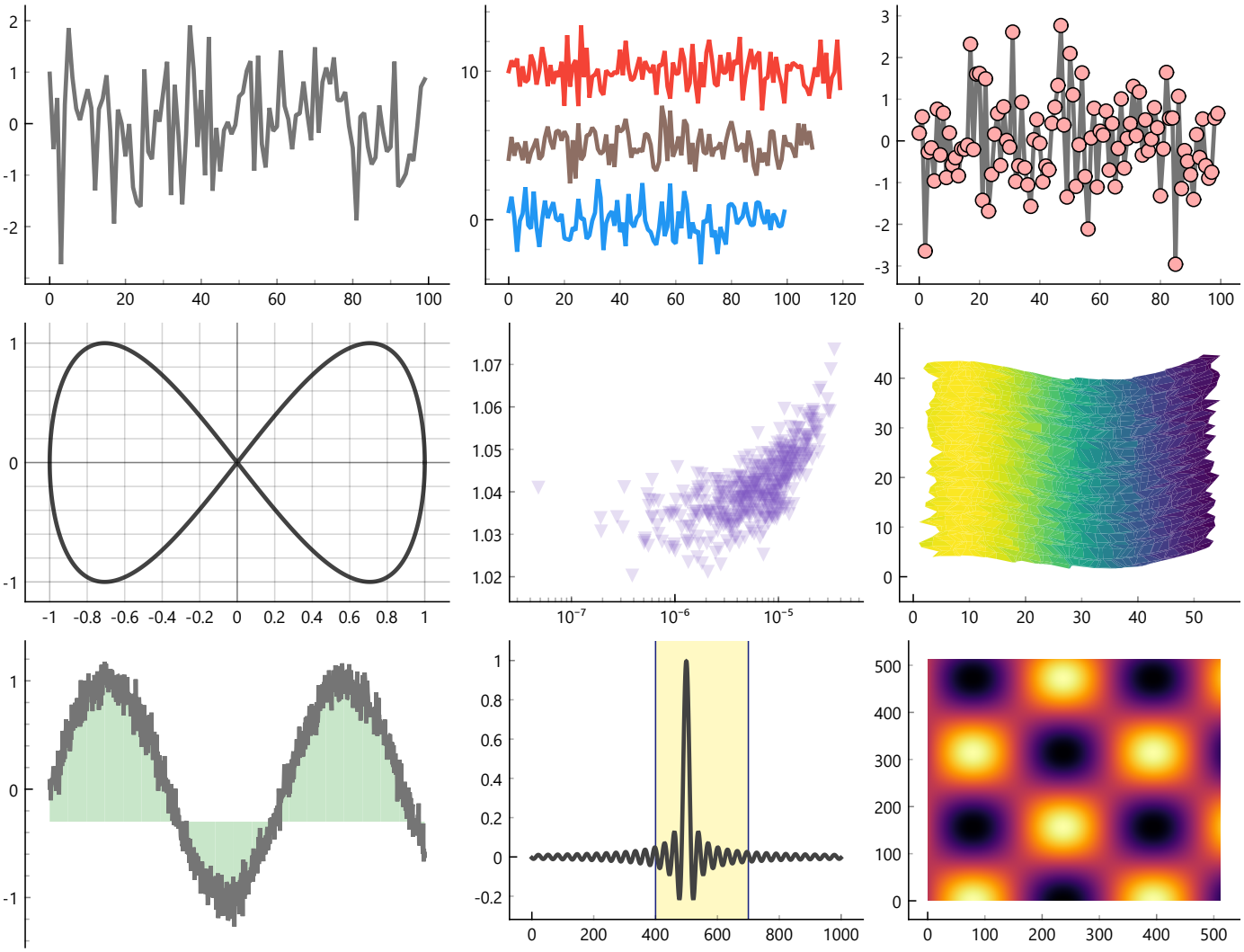

Fig. 1: *A selection of basic plots from PyQtGraph's suite of examples.*

Approach

Python

The Python programming language enjoys a large popularity in scientific research due ease of entry and a robust standard library combined with access to very comprehensive numerical computing packages. This makes Python an attractive alternative to established computational tools such as MATLAB[6] and Mathematica.

The set of most commonly used scientific computing tools in Python are commonly referred to as the SciPy stack. This refers to SciPy, NumPy, and a variety of other libraries that use the NumPy `ndarray` data structure as a container for vectorized operations. The `ndarray` gives developers a high level API to low-level operations with excellent performance. This API allows NumPy and SciPy to provide a wide variety of standard numerical computing operations, all of which are very efficient and help overcome the performance penalty of working with Python as a cross-platform, interpreted, dynamically typed language.

Qt

The Qt framework is a GUI platform written in C++ that allows the creation of cross-platform applications with a single shared code-base. Comprehensive Python bindings (PyQt) expose the complete Qt API. Here, the specific section of interest is the `GraphicsView` framework, which provides a surface for managing and interacting with a large number of custom-made 2D graphical items, with support for zooming and rotation[13]. PyQtGraph is built on this foundation to extend the SciPy stack with performant cross-platform visualization.

Implementation

`GraphicsView` renders line segments in a freely scaled coordinate system through `QPainterPath` objects. The rendering performance of PyQtGraph results from optimized code to create such paths directly from NumPy ndarrays describing sets of x and y coordinates. One illustrative example tightly interfaces with Qt's internal pointers through `QPolygonF` objects to offer significant speedups for `QPainterPath` generation. They use NumPy's structured array functionality to efficiently create a binary compatible structure that can serve as an input stream to a `QPainterPath` item (see the Appendix section for details). This `QPainterPath` is then drawn to the screen by the Graphics View framework. Note that while `arrayToQPolgyonF` is a trivial example of NumPy/Qt integration, a much more complex usage can be found here.

CAPABILITIES

All 2d line rendering functions that handle large quantities start with NumPy arrays and become painter paths through the powerful `arrayToQPath` conversion. This generic NumPy-to-Qt data translator covers all common plotting requirements. Figure 1 shows a demonstration from the suite of examples. All graphs included in this paper were generated using PyQtGraph's interactive export functions, which can store both bitmaps and vector formats, or provide access to the raw plotted data.

Plot Types

PyQtGraph shows all plots within a `PlotItem` object consisting of a `ViewBox` equipped with a set of axes. This allows dynamic pan and zoom through the transforms of Qt's `GraphicsView`, with no need to regenerate the `QPainterPath` objects. Individual elements of the plot are represented by graphics items that share the same coordinate systems and shown in any combination and drawing order.

PyQtGraph represents line plots as `PlotCurveItem` objects and offers typical functionality such as color, width and dashing. "Shadow pen" lines can be underlaid for additional contrast.

Scatter plot items are assigned a default shape, color and size per data set, but each point can also have a unique attributes. Shapes are pre-rendered and cached to optimize performance when the underlying dataset is updated. Depending on the application, symbols can be set to scale with the view or maintain constant size. Functionality is included for items in scatter plots to recognize mouse hover events.

Plots can be extended by both horizontal and vertical error bars and annotated by text labels. Built in routines can also transform the plotted data to provide logarithmic scaling, Fourier transforms, and to show the gradient dy/dt directly over t or as a phase map over y.

Bar graphs and images also make use of this framework and can be added to the same PlotItem, although they are more commonly used separately. Users can also create `QPainterPath` objects to add their own graphical elements using the well documented methods of the Qt Graphics View framework. PyQtGraph's suite of examples[14] illustrates this with some demonstrations.

Performance

We evaluated the plotting performance for line plots of randomly generated datasets of different length. Figure 2 shows the time taken from setting new data to the completion of the drawing process for 1 to 100 separate curves ranging from 100 to 10 million points. We find that even on common hardware, a curve with 10,000 points can be drawn in less than 10 ms, and an update rate of 60 Hz can be maintained up to approximately 30,000 points. Adding more curves introduces additional overhead, such that the same number of 10,000 points, plotted over 100 curves of 100 points each, increases the update time to just below 40 ms. Nevertheless, the number of points in each of the 100 curves can be increased to close to 3,000 before the update rate falls below 10 frames per second (FPS). At this point, the majority of time is spent processing line segments, and their distribution across different numbers of curves is of secondary importance: A single curve allows for 200,000 points to be displayed at 10 FPS.

Plotting the results as the update time divided by the overall number of data points further illustrates this. As the total number of points approaches 100,000, where the more or less fixed overhead of the Qt drawing process is no longer significant, the update times converges to approximately 200 ns per point drawn for both 10 curves and 100 curves. We attribute the increased update time for a single curve to the larger set of data that needs to be handled simultaneously, which may lead to caching issues.

Although the detailed result will vary with platform, system, and data, we consider these results to provide a good reference for the performance that can be expected from PyQtGraph.

Images and Regions of Interest

PyQtGraph also provides the means to display images and other multi-dimensional data. Handling streams of such data, as in live video, is similarly enabled by efficient NumPy methods that convert the input data into a binary representation that can be used directly by the Qt framework. Various analysis and processing tools interact with the image arrays, for example regions of interest (ROIs), look-up tables (LUT) for color-mapped display, or histograms.

Image Views

The principal object in displaying images, `ImageItem`, accepts 2-dimensional (interpreted as grayscale) or 3-dimensional (either color or color and alpha) data of any numeric type. Stored in NumPy arrays, this data can be pre-processed efficiently using any available functions in the SciPy stack. Subtracting a background, for instance, is simply a matter of subtracting the reference frame. This input is then processed by `ImageItem` and converted into Qt's `QImage` format for rapid display. A range of colormaps are provided to enhance detail perception, and can be altered interactively in levels and colors through a `HistogramLUTItem`.

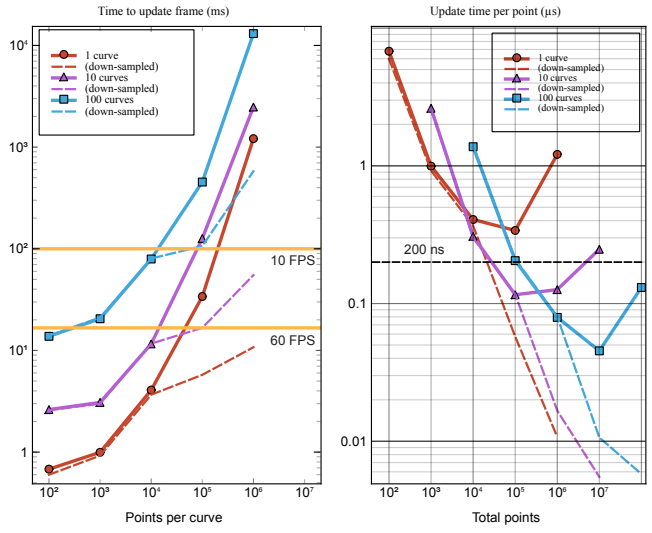

Fig. 2: *Line speed benchmark. The time to render 1, 10 or 100 lines of data is shown for varying numbers of points per line. All data was created using an AMD 5900x Ryzen 9 CPU. Left: Time per update over points per curve. The thresholds for achieving 10 and 60 frames/s are shown by horizontal lines. Right: Update time per point, plotted over the total number of points. For more than 100,000 points, the line-plotting time becomes dominant, and the results converge to 200 ns per point for both 10 and 100 curves, while plotting all points as a single curve increases the time to 500–600 ns per point.*

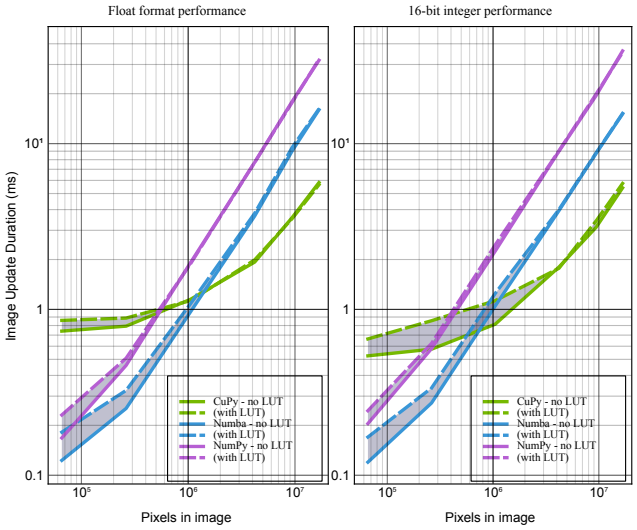

Fig. 3: *Image speed benchmark. The time to update an image frame is shown for different data formats. Left: Using optimized NumPy processing (purple lines), the drawing time is log-scale linear with the number of pixels over a wide range. GPU accelerated CUDA processing using CuPy (green lines) describe a more complex relationship with image size. The need to copy data to and from the GPU creates additional overhead, but as image size grows, the faster processing speed becomes sufficient to compensate for that overhead. The choice of various extra processing tasks like LUTs (dashed lines) show the same basic trends. Alternatively, PyQtGraph's image rendering pipeline can be accelerated in Numba is available on the system. Benchmarks with Numba (blue lines) can be seen as have performance between that of CuPy and NumPy only. Right: For input data in uint16 format, CUDA processing is particularly advantageous and can provide an almost four-fold reduction in drawing time. Benchmarks were performed on an AMD 5900x Ryzen 9 CPU and an NVIDIA RTX 3080 discrete GPU.*

ROIs

A common image analysis task is to define a ROI in a larger original image. This is supported by multiple interactive objects (`LineROI`, `CircleROI`, `PolygonROI`, and others), which provide NumPy slice objects that reference the selected region within the image array. Once extracted, the relevant data can then be further processed. Magnification, live plotting, FFTs and custom analysis are all simple to implement. Multiple ROIs can be bound together in groups to provide background correction or region comparisons both within a single image stream or across many. These ROI objects remain interactive while attached to the image, so that resizing, moving and rotating a ROI can prompt immediate updates of all subsequent plotting and analysis interfaces.

Performance

Numerous factors play into the final performance of a video stream. Data type conversions, LUTs, scaling, and any custom preprocessing all need to occur for each frame, and the computational effort typically scales with image size. A minimum of 20 FPS is generally required for a usable interactive video stream, although 60 FPS is preferred in many applications. In some cases, data can be directly passed to the built-in methods of Qt's `QImage`. Otherwise `ImageItem` relies on the core function `makeARGB` to efficiently convert data types, order data properly, rescale levels and apply a LUT if desired (see the Appendix section for details).

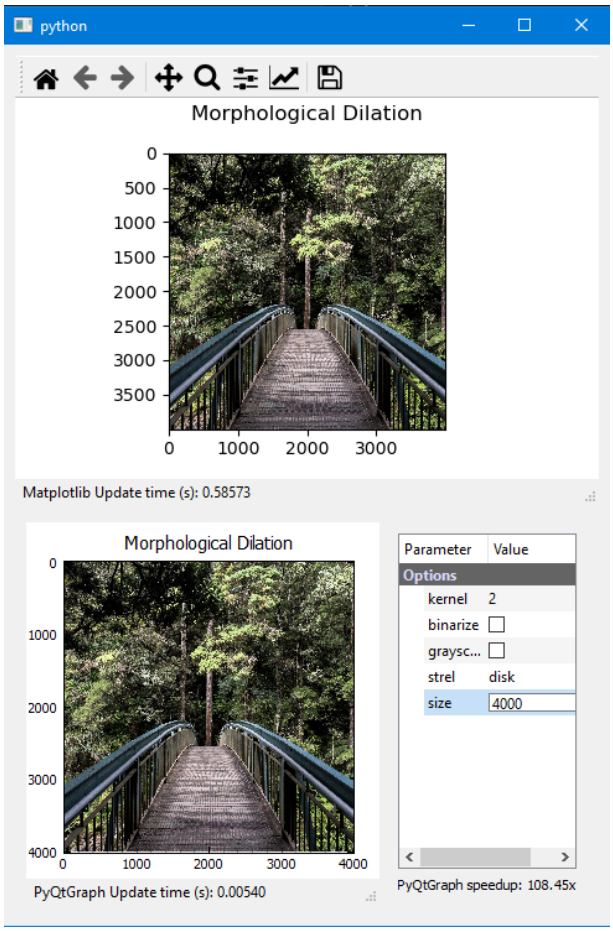

Fig. 4: *Performance test with PyQtGraph and Matplotlib widgets embedded in a Qt5 application. Over a wide range of image sizes, PyQtGraph completes drawing approximately 75–150 times faster, taking only 5.4 ms in this example of a 4000×4000 image. The test is performed without GPU acceleration in a Microsoft Windows environment, and both libraries are set to sub-sample without interpolation. Free-to-use test images are provided by the "Unsplash" service.*

When integrated as a widget in a Qt application (Figure 4), we typically find `ImageItem` to display an image 75–150 times faster than the `FigureCanvas` provided by Matplotlib, a plotting library that emphasizes graphical quality over speed.

Some share of the image processing is by necessity done in the primary event thread of the Qt application, as that thread requires full access to the data to be displayed. Other calculations can be moved to other threads to improve performance and maintain the responsiveness of the UI. For example, larger images can be downsampled before handing them to the main thread for display. This multi-threading consideration extends throughout the application, and any excessive use of the event thread will impact image display performance.

To further accelerate the handling of large datasets, PyQtGraph can make use of a GPU substrate in one of two ways: `GLImageItem` or CuPy. `GLImageItem`, while limited in its interactivity, employs OpenGL for rendering. The CuPy library, a drop-in replacement for NumPy, moves array processing tasks to a CUDA-enabled GPU. This is not beneficial in all applications, since the cost of copying the image data between system memory and the GPU needs to be amortized by a sufficient number of calculations. In the context of image processing, we find that CuPy

provides an advantage for images with several hundred thousands of pixels (Figure 3), depending on target hardware.

Interactivity

Event Driven GUI

The Qt framework is event driven, which allows PyQtGraph to provide seamless mouse interaction. This also enables users to develop their own desired behavior in response to mouse move, hover, leave, enter, double-click, zoom, or drag events. Almost every aspect of PyQtGraph interacts with the Qt events, or provides its own in response to e.g. axis adjustments or changes to a selected region. This interactivity is a core component of the Qt framework, and adding such behavior to a plot in PyQtGraph is no more complicated than generating the plot in the first place.

Responsiveness at scale

Recognizing zoom events enables resolution-aware downsampling of the plotted data. Multiple available methods provide different trade-offs of accuracy against performance, and include a "peak" display that precisely captures the minima and maxima of the data, a "mean" over the down-sampled interval, and a fast "sub-sample" that displays only 1 in N data points. Zooming into the view automatically reveals more detail of the dataset.

Parameter Trees

Another common requirement for user interaction is a mechanism to interact with for configuration settings or algorithm parameters. PyQtGraph provides this capability through the `ParameterTree` object which hosts any number of `Parameters`. Similar to `traitlets`, PyQtGraph's `Parameter` objects encapsulate a value type and allow registering callbacks, performing input validation, and more. However, `Parameters` are different in that most are coupled to a widget representation, i.e., allowing users to easily update the values graphically. `Parameter` objects can be created through a simple Python dictionary listing its specifications (type, value, and traits such as 'readonly' or value range). It is then bound to a Qt widget for editing text, numeric, list-like, and custom data depending on the parameter type. Parameters can be grouped, linked, and dynamically instantiated or removed. Callbacks for user actions or value changes allow results to be recalculated immediately (i.e., while a spinbox is changing its value) or after the value has settled. Parameter trees can save and load their states hierarchically to easily create persistent configuration files. Since accessing parameters mimics a Python dictionary, they can function as a drop-in replacement for programmatically adjusted settings rather than forcing users to interface through widgets alone.

EXAMPLES

Rapid iteration of processing parameters

Figure 5 shows such a parameter tree in use. In applications such as image processing, immediate feedback for a choice of algorithmic parameters can help to rapidly reduce the exploration space in the search for viable solutions. For instance, it might be difficult to tell the appropriate kernel size for a morphological operation without testing multiple combinations of image types, parameter values, and more. These factors often make fine-tuning a laborious process. Parameter trees assist in creating a tool to integrate the user with the testing space, quickly and without large amounts of boilerplate code. Using callbacks to provide immediate response, workable parameter combinations can be explored, and candidate solutions can be stored to configuration files, both for comparison to alternative approaches and for application to specific data types.

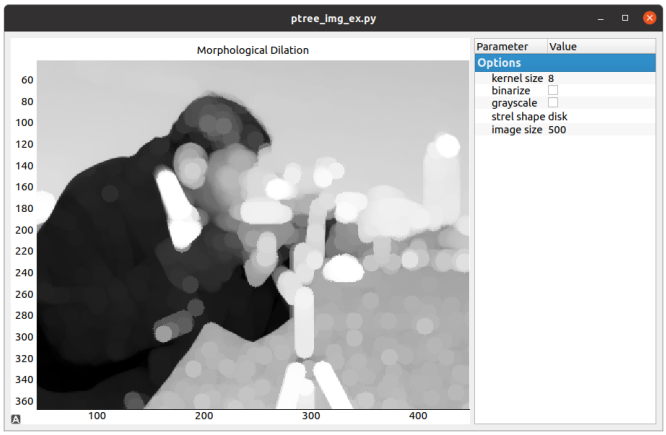

Fig. 5: *Sample use of parameter trees for user interaction, where various image processing parameters can be quickly updated. The displayed image reflects these changes in real-time.*

Model Prototyping

The supplementary information contains a similar application of PyQtGraph's capabilities to a machine learning model. Here the parameter trees allows tuning aspects of the input data, model structure and output formats. The plotting functions provide live feedback for how these changes affect model accuracy, greatly assisting a rapid prototyping process.

Monitoring of real-time data

Visualization can provide immediate feedback on measurement results and the operational state of the equipment involved. Figure 6 shows an application of the opportunities provided by PyQtGraph's interactive facilities in this application. For most applications, no data reduction is necessary to maintain smooth display of a sufficiently large buffer, and no additional code is needed to alternate between monitoring of new data and close inspection of specific events.

Additional examples

The supplementary information includes video demonstrations of two additional applications that make heavy use of PyQtGraph functionality to explore spectral data and to visualize volumetric data representing the 3d structure of multilayer circuit boards.

SOFTWARE DEVELOPMENT

The original motivation for pyqtgraph was in data acquisition software, where there is a need to be able to display video and plots with realtime frame rates and interactivity that allows data exploration. Interactivity was highly important; the established `matplotlib` library already existed and was excellent for visualizing data in a way that tells a particular story. New data, though, doesn't have this story yet. You want to be able to slice it

```
def ndarray_from_qpolygonf(polyline):
    # polyline.data() will be None if the pointer was null.
    # voidptr(None) is the same as voidptr(0).
    vp = Qt.compat.voidptr(polyline.data(), len(polyline)*2*8, True)
    return np.frombuffer(vp, dtype=np.float64).reshape((-1, 2))

def create_qpolygonf(size):
    polyline = QtGui.QPolygonF()
    if hasattr(polyline, 'resize'):
        # (PySide) and (PyQt6 >= 6.3.1)
        polyline.resize(size)
    else:
        polyline.fill(QtCore.QPointF(), size)
    return polyline

def arrayToQPolygonF(x, y):
    """
    Utility function to convert two 1D-NumPy arrays representing curve data
    (X-axis, Y-axis data) into a single open polygon (QtGui.PolygonF) object.
    """
    # Validation asserts both x and y are same-shaped and 1D, not shown here
    size = x.size
    polyline = create_qpolygonf(size)
    memory = ndarray_from_qpolygonf(polyline)
    memory[:, 0] = x
    memory[:, 1] = y
    return polyline
```

TABLE 1: *PyQtGraph source code for the core `arrayToQPolygonF` function.*

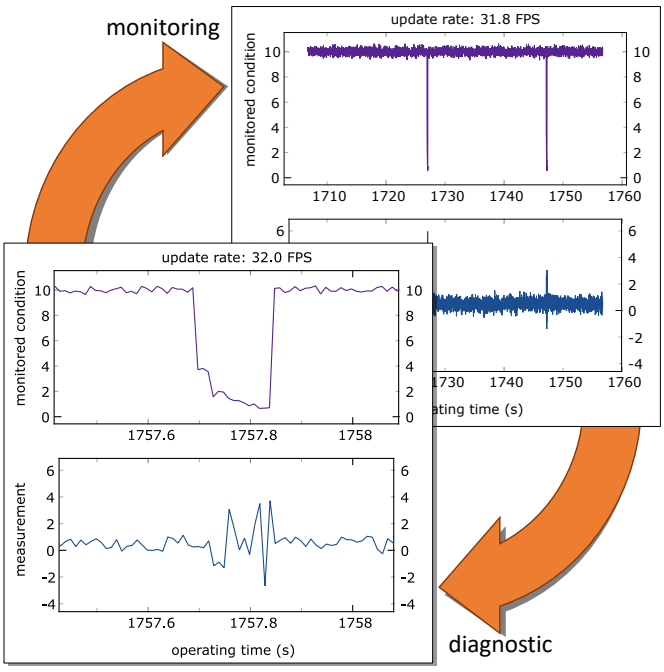

Fig. 6: *Monitoring and diagnostic of a (simulated) experiment with intermittent failures. Incoming data at 100 samples/s for two measurement channels is recorded into a rolling 5,000 point buffer and continuously displayed at 30 frames/s. When a failure is observed, it can quickly be brought into focus with simple mouse interactions (click-and-drag and mousewheel zoom) for inspection, or to record accurate time stamps. Afterwards, a single click returns the view to automatic scaling without loss of any incoming data.*

and stretch it and look at it from every possible angle, quickly and easily, so that you can decide what story to tell.

At the time, most acquisition software would have been written in C/C++ for efficiency. However, newer developments meant python interfaces to Qt's C++ logic provided a good mix between speed and ease of use. PyQwt was perfect for this purpose, but went through a long period without a maintainer (presumably at the time, it was a huge burden maintaining and distributing compiled python packages). So pyqtgraph began as a replacement for PyQwt that would be pure-python, and thus easier to develop and distribute. Following that template, it was also to include UI elements that have common use in acquisition/analysis applications, but are missing from Qt (for example, tools for adjusting image contrast, parameter trees, etc.).

PyQtGraph was first released in 2012, under the open source MIT license. It is known to run on systems ranging from the Raspberry Pi to IBM's s390x architecture. Development is coordinated by volunteer maintainers, with additional code provided by occasional contributors. A continuous integration system asserts that the codebase passes a suite of tests for different combinations of Qt bindings, Python versions and operating systems. PyQtGraph has adopted NEP-29[2] to establish a support timeline for Python and NumPy versions in line with the rest of the Python community and development occurs in close communication with projects such as ACQ4[15] and Orange3[16] that constitute a large part of the user base.

OUTLOOK

With a growing number of both maintainers and contributors, PyQtGraph is well positioned to take advantage of technological developments. The support of hardware acceleration in recent versions of NumPy has already been used to add CUDA integration to some time-critical code, but there is still plenty of potential for further improvements to performance and capabilities. Increased use of multi-threaded patterns is a goal in this respect, both throughout the library, and in user code supported by appropriate

```
import cupy as cp
import numpy as np

def makeARGB(data, lut=None, levels=None, scale=None, useRGBA=False, output=None):
    # condensed variant, full code at:
    # https://github.com/pyqtgraph/pyqtgraph/blob/pyqtgraph-0.12.0/pyqtgraph/functions.py#L1102-L1331
    xp = cp.get_array_module(data) if cp else np

    nanMask = None
    if data.dtype.kind == "f" and xp.isnan(data.min()):
        nanMask = xp.isnan(data)
    # Scaling
    if isinstance(levels, xp.ndarray) and levels.ndim == 2: # rescale each channel independently
        newData = xp.empty(data.shape, dtype=int)
        for i in range(data.shape[-1]):
            minVal, maxVal = levels[i]
            if minVal == maxVal:
                maxVal = xp.nextafter(maxVal, 2 * maxVal)
            rng = maxVal - minVal
            rng = 1 if rng == 0 else rng
            newData[..., i] = (data[..., i] - minVal) * (scale / rng)
        data = newData
    else:
        minVal, maxVal = levels
        rng = maxVal - minVal
        data = (data - minVal) * (scale / rng)
    # LUT
    if xp == cp: # cupy.take only supports "wrap" mode
        data = cp.take(lut, cp.clip(data, 0, lut.shape[0] - 1), axis=0)
    else:
        data = np.take(lut, data, axis=0, mode='clip')

    imgData = output
    if useRGBA:
        order = [0, 1, 2, 3]  # array comes out RGBA
    else:
        order = [2, 1, 0, 3]  # channels line up as BGR in the final image.
    # attempt to use library function to copy data into image array
    fastpath_success = try_fastpath_argb(xp, data, imgData, useRGBA)
    if fastpath_success:
        pass
    elif data.ndim == 2:
        for i in range(3):
            imgData[..., i] = data
    elif data.shape[2] == 1:
        for i in range(3):
            imgData[..., i] = data[..., 0]
    else:
        for i in range(0, data.shape[2]):
            imgData[..., i] = data[..., order[i]]
    if data.ndim != 3 or data.shape[2] != 4:
        imgData[..., 3] = 255
    # apply nan-mask through alpha channel
    if nanMask is not None:
        if xp == cp: # Workaround for https://github.com/cupy/cupy/issues/4693
            imgData[nanMask, :, 3] = 0
        else:
            imgData[nanMask, 3] = 0
    return imgData
```

TABLE 2: *PyQtGraph source code for the core* `makeARGB` *function. For brevity, edge cases and null checks have been omitted.*

documentation, examples and API design. The growing maturity of the Numba just-in-time compiler [17] for Python code provides additional opportunities for acceleration beyond what NumPy's array operations can provide.

APPENDIX

Implementation of arrayToQPolygonF

The function `arrayToQPolygonF` is one of the simpler cases that demonstrates the how PyQtGraph bridges the gap between NumPy and Qt Table 1

Execution takes two `ndarray` objects of the same length, representing *x*- and *y*-coordinates for a series of line segments. A `QPolygonF` object is instantiated and resized to store enough points that represent the *x*- and *y*-coordinates that were passed in. From there, a void-pointer of the `QPolygonF`'s internal memory is retrieved in a NumPy format allowing easy assignment of the user data. Lastly, we fill that NumPy array with the *x*- and *y*-coordinates that were initially provided. In this process, we went from NumPy arrays representing *x*- and *y*- coordinates to a `QPolygonF` object without performing any serialization, iteration or casting.

Implementation of makeARGB

The function `makeARGB` provides the data conversions used in displaying image data. It is included here as Table 2 to show the approach and the integration of CUDA GPU support discussed in section .

The segment of memory within a `QImage` object that will ultimately be displayed on the screen can be accessed and written to as a contiguous, row-major, 3-dimensional NumPy `ndarray` of unsigned 8-bit integers; i.e. the red, green and blue color values and alpha value of each pixel, one row at a time. With this array as the output target, an incoming image data goes through a number of processing steps. Many of the steps are only conditionally executed, depending on the shape and type of the incoming data, as well as the use of LUTs or rescaling. Some of the respective branches and decision trees have been omitted here for brevity. In a best-case scenario, the incoming data is already in the correct format, and the steps converting data type and element order can then also be omitted. The CuPy library provides CUDA support by replicating large sections of NumPy functionality, allowing for near-identical code paths. The two if-statements seen here address the lack of a 'clip' mode in CuPy's 'take' function, as well as differing behavior for masks as indices.

Acknowledgements

The authors wish to thank all prior, present and future contributors to the PyQtGraph project. Their efforts enable all that is presented here. One regular contributor, @pijyoi, has made significant contributions to the NumPy and Qt interoperability, as well as reviewed pull requests from other contributors and maintainers and provided countless bug-fixes. Finally, we would like to thank maintainers and contributors to the NumPy, SciPy and CuPy projects.

References

[1] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "CuPy: A NumPy-compatible library for NVIDIA GPU calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf

[2] *NEP 29 — Recommend Python and NumPy version support as a community policy standard*, available at https://numpy.org/neps/nep-0029-deprecation_policy.html.

[3] J. H. Friedman and W. Stuetzle, "John W. Tukey's work on interactive graphics," *The Annals of Statistics*, vol. 30, pp. 1626–1639, 2002, https://doi.org/10.1214/aos/1043351250.

[4] J. S. Byrd, "Microcomputers for nuclear instrumentation," *presented at Conference and Exhibits on Small Computers, May 23-24 1979, Clemson, USA*, 1 1979, available at https://www.osti.gov/biblio/6060192.

[5] A. V. Reed, "On choosing an inexpensive microcomputer for the experimental psychology laboratory," *Behavior Research Methods & Instrumentation*, vol. 12, pp. 607–613, 1980, https://doi.org/10.3758/BF03201852.

[6] C. Moler and J. Little, "A history of MATLAB," in *Proceedings of the ACM on Programming Languages*, vol. HOPL 4. ACM New York, NY, USA, 2020, pp. 81.1–81.67, https://doi.org/10.1145/3386331.

[7] S. Josifovska, "The father of LabView," *IEE Review*, vol. 49, pp. 30–33, 2003, https://doi.org/10.1049/ir:20030905.

[8] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, 2020, https://doi.org/10.1038/s41586-020-2649-2.

[9] J. L. Johnson, H. tom Wörden, and K. van Wijk, "PLACE: An open-source Python package for laboratory automation, control, and experimentation," *Journal of Laboratory Automation*, vol. 20, pp. 10–16, 2015, https://doi.org/10.1177/2211068214553022.

[10] L. J. Koerner, T. A. Caswell, D. B. Allan, and S. I. Campbell, "A Python instrument control and data acquisition suite for reproducible research," *IEEE Transactions on Instrumentation and Measurement*, vol. 69, pp. 1698–1707, 2020, https://doi.org/10.1109/TIM.2019.2914711.

[11] C. Bozzi, S. Roiser, and the LHCb Collaboration, "The LHCb software and computing upgrade for run 3: opportunities and challenges," in *IOP Conf. Series: Journal of Physics: Conf. Series*, 2017, https://doi.org/10.1088/1742-6596/898/11/112002.

[12] *Qt widget toolkit*, https://www.qt.io.

[13] *QGraphicsView Class*, Qt documentation, March 2021, https://doc.qt.io/qt-5/qgraphicsview.html. [Online]. Available: https://doc.qt.io/qt-5/qgraphicsview.html

[14] *Example Application*, PyQtGraph, can be run after installation by `python -m pyqtgraph.examples`.

[15] L. Campagnola, M. Kratz, and P. Manis, "Acq4: an open-source software platform for data acquisition and analysis in neurophysiology research," *Frontiers in Neuroinformatics*, vol. 8, p. 3, 2014, https://doi.org/10.3389/fninf.2014.00003. [Online]. Available: https://www.frontiersin.org/article/10.3389/fninf.2014.00003

[16] J. Demšar, T. Curk, A. Erjavec, Črt Gorup, T. Hočevar, M. Milutinovič, M. Možina, M. Polajnar, M. Toplak, A. Starič, M. Štajdohar, L. Umek, L. Žagar, J. Žbontar, M. Žitnik, and B. Zupan, "Orange: Data mining toolbox in Python," *Journal of Machine Learning Research*, vol. 14, pp. 2349–2353, 2013. [Online]. Available: http://jmlr.org/papers/v14/demsar13a.html

[17] 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.