Xi-CAM

Release 0+unknown

Ronald J. Pandolfi

Nov 14, 2022

CONTENTS

The `Xi-CAM` package is a plugin-based framework and graphical user interface (GUI) application for synchrotron data management, acquisition, visualization, and analysis. Support for a variety of techniques is provided through its plugin architecture.

GETTING STARTED

Before starting to develop plugins, you will need to install Xi-CAM and some of its dependencies.

Follow the QuickStart guide to get started with installation and exploring an example Xi-CAM plugin (covers GUIPlugin, Workflow, and OperationPlugins).

For more detailed documentation, you may want to visit the following:

- GUIPlugin documentation
- OperationPlugin documentation
- Workflow documentation

The Resources section has useful links to tutorials, examples, and documentation that can help with developing Xi-CAM plugins.

DEVELOPER DOCUMENTATION

2.1 QuickStart Guide

This is a quick-start guide that will help you install Xi-CAM and explore an example plugin that you can experiment with.

This guide does not explore the implementation of the plugin in too much detail. For more in-depth documentation for developing plugins from scratch, see:

2.1.1 GUIPlugin Documentation

This documentation provides information on GUIPlugins and GUILayouts to help with designing your own plugins for Xi-CAM. API reference documentation is also included at the bottom.

If you are new to developing Xi-CAM plugins, it is recommended that you follow the quick-start documentation *first.*

For more general development resources, see the *resources documentation*.

What Is A GUIPlugin?

A GUIPlugin is an interactive user-facing plugin in Xi-CAM. It can be used to visualize and analyze data.

GUIPlugins make use of the `qtpy` Python package for interactive GUI components. See the resources documentation for more information.
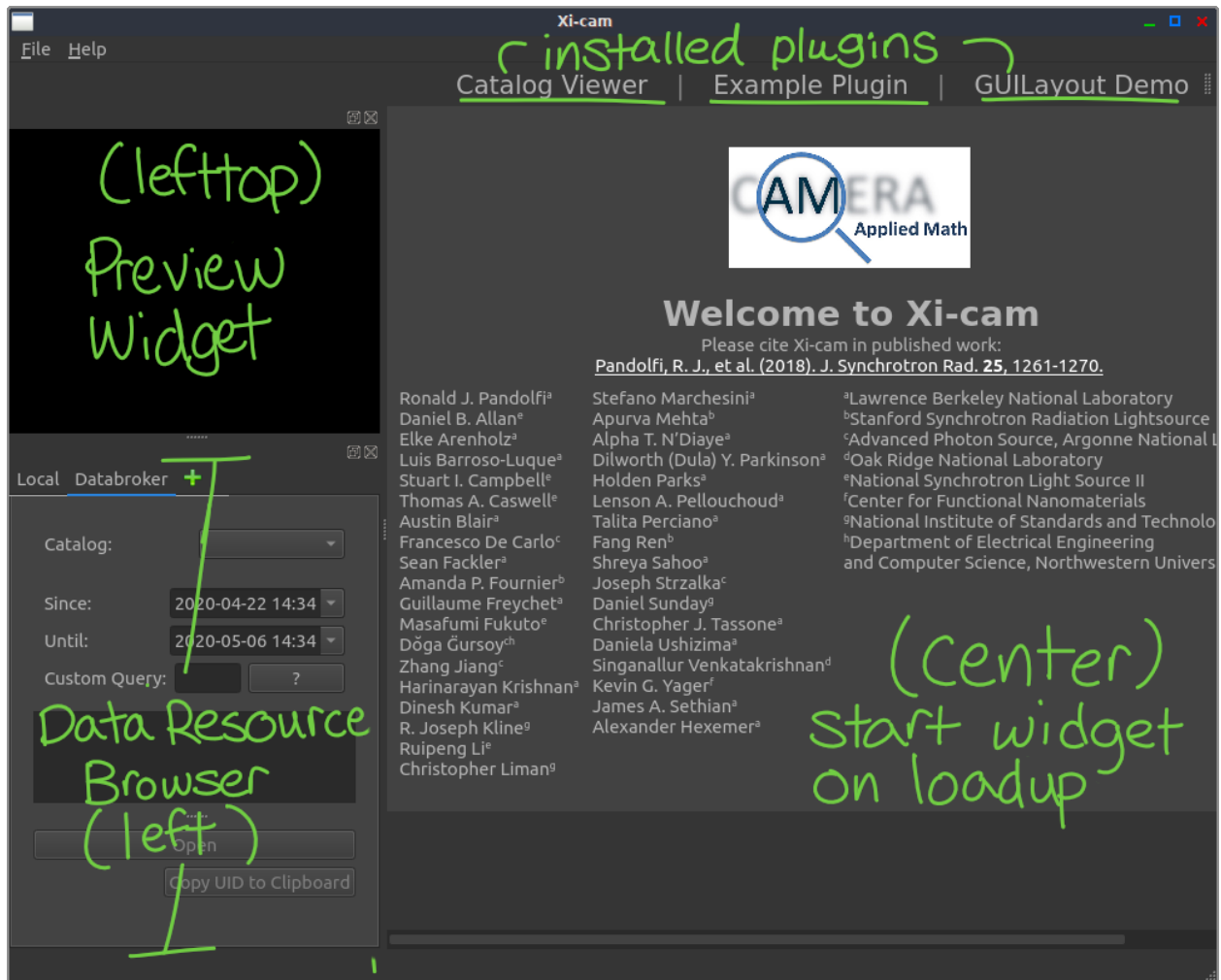
Where is GUIPlugin?

```
xicam.plugins.guiplugin
```

What Does a GUIPlugin Look Like?

First, let's look at what Xi-CAM looks like when you first load it:

Main window of Xi-CAM when running xicam. Note that there are three installed *GUIPlugins* here; if you haven't installed any plugins, you won't see any listed.

As you can see, the main window of Xi-CAM after it has finished loading shows any installed GUIPlugins, a citation / references widget, a preview widget, and a data browser widget. The data browser widget can be used to load data into a `GUIPlugin`. The data preview widget can be used to "preview" data before loading it.

Xi-cam

File   Help

installed plugins

Catalog Viewer | Example Plugin | GUILayout Demo

(lefttop)
Preview
Widget

CAMERA
Applied Math

Welcome to Xi-cam
Please cite Xi-cam in published work:
Pandolfi, R. J., et al. (2018). J. Synchrotron Rad. **25**, 1261-1270.

Local   Databroker   +

Catalog:

Since: 2020-04-22 14:34

Until: 2020-05-06 14:34

Custom Query: ?

Data Resource
Browser
(left)

Open

Copy UID to Clipboard

Ronald J. Pandolfi[a]
Daniel B. Allan[e]
Elke Arenholz[a]
Luis Barroso-Luque[a]
Stuart I. Campbell[e]
Thomas A. Caswell[e]
Austin Blair[a]
Francesco De Carlo[c]
Sean Fackler[a]
Amanda P. Fournier[b]
Guillaume Freychet[a]
Masafumi Fukuto[e]
Doǧa Gursoy[ch]
Zhang Jiang[c]
Harinarayan Krishnan[a]
Dinesh Kumar[a]
R. Joseph Kline[g]
Ruipeng Li[e]
Christopher Liman[g]

Stefano Marchesini[a]
Apurva Mehta[b]
Alpha T. N'Diaye[a]
Dilworth (Dula) Y. Parkinson[a]
Holden Parks[a]
Lenson A. Pellouchoud[a]
Talita Perciano[a]
Fang Ren[b]
Shreya Sahoo[a]
Joseph Strzalka[c]
Daniel Sunday[g]
Christopher J. Tassone[e]
Daniela Ushizima[a]
Singanallur Venkatakrishnan[d]
Kevin G. Yager[f]
James A. Sethian[a]
Alexander Hexemer[a]

[a]Lawrence Berkeley National Laboratory
[b]Stanford Synchrotron Radiation Lightsource
[c]Advanced Photon Source, Argonne National L
[d]Oak Ridge National Laboratory
[e]National Synchrotron Light Source II
[f]Center for Functional Nanomaterials
[g]National Institute of Standards and Technolo
[h]Department of Electrical Engineering
and Computer Science, Northwestern Univers

(Center)
Start widget
on loadup

It is important to keep in mind a few concepts for GUIPlugins:

- A `GUIPlugin` can have one or more `stages`.
- Each `stage` is defined with a `GUILayout`.
- A `GUILayout` is defined with a widget (or multiple widgets).

These concepts are explored in more detail later in this document.

How Do I Create a GUIPlugin?

To create a `GUIPlugin`, you will need:

- a derived class of `GUIPlugin`
- a `setup.py` file with a `xicam.plugins.GUIPlugin` entry point

Although you may structure your plugin's code and support files as you like, we recommend using a `cookiecutter` template that we have created for Xi-CAM's `GUIPlugin`.

What is cookiecutter?

`cookiecutter` is a templating tool that can be used to interactively create python project. For more information, see the cookiecutter documentation.

Install cookiecutter

In your active environment, you will need to `pip install cookiecutter`.

Run cookiecutter with the Xi-CAM GUIPlugin Template

Now, in the directory of your choice (the home directory, ~, should work if you are unsure), run the following:

```
cookiecutter https://github.com/Xi-CAM/Xi-cam.templates.GuiPlugin
```

This will download the template for creating a `GUIPlugin`, then present you with a series of prompts.

A prompt will look like `prompt [default value]:` . If you want to use the default value specified, hit the enter key. Otherwise, respond to the prompt with the value you would like.

Here are the prompts with their descriptions:

This will create a python package with some files and code to get started developing a GUIPlugin. **You can always change the names of your plugin, package, etc. later by hand.**

The GUIPlugin you created will be implemented in `xicam.package_name/xicam/package_name/__init__.py`.

For purposes of this documentation, we will refer to these values by their defaults.

Installing Your GUIPlugin

When you create a new plugin package using cookiecutter, one of the files it generates is `setup.py`. This contains meta-information about the package. When you run `pip install` of your package, it uses this information to create a disribution.

`setup.py` also defines entry points, which Xi-CAM uses to find plugins.

For more information about entry points in Xi-CAM, see the *following documentation*.

Navigate to your created package directory and create an editable pip install:

```
cd xicam.my_plugin
pip install -e .
```

This tells pip install your file locally by looking at the `setup.py` file, and the `-e` allows you to make changes to your code without having to reinstall.

If you change an entry point in setup.py, you must reinstall.

Selecting and Activating a GUIPlugin

We can activate any of the installed GUIPlugins by clicking on their name at the top. Let's click on "My Plugin":

Note that this plugin doesn't do much yet; it simply displays the text "Stage 1…" You can also click "Stage 2" at the top, and you will see the text "Stage 2…" in the center.

How is MyPlugin Implemented?

The code for `MyPlugin` is implemented in `xicam.package_name/xicam/package_name/__init__.py`.

```python
from qtpy.QtWidgets import QLabel

from xicam.plugins import GUIPlugin, GUILayout


class MyPlugin(GUIPlugin):
    # Defines the name of the plugin (how it is displayed in Xi-CAM)
    name = "My Plugin"

    def __init__(self, *args, **kwargs):
        # Insert code here

        # Modify stages here
        self.stages = {'Stage 1': GUILayout(QLabel("Stage 1...")),
                       "Stage 2": GUILayout(QLabel("Stage 2..."))}

        # Initialize the parent class, GUIPlugin
        super(MyPlugin, self).__init__(*args, **kwargs)
```

Cookiecutter set up this starter code for us. We have a derived version of `GUIPlugin`, which we call `MyPlugin`. It has the name "My Plugin", which is how it will appear in the Xi-CAM GUI.

We then have an `__init__` method to describe how to create a `MyPlugin`. Notice that there is a `QLabel`, which is simply text, added into two `GUILayouts`. These layouts are then added to the interface via `self.stages`.
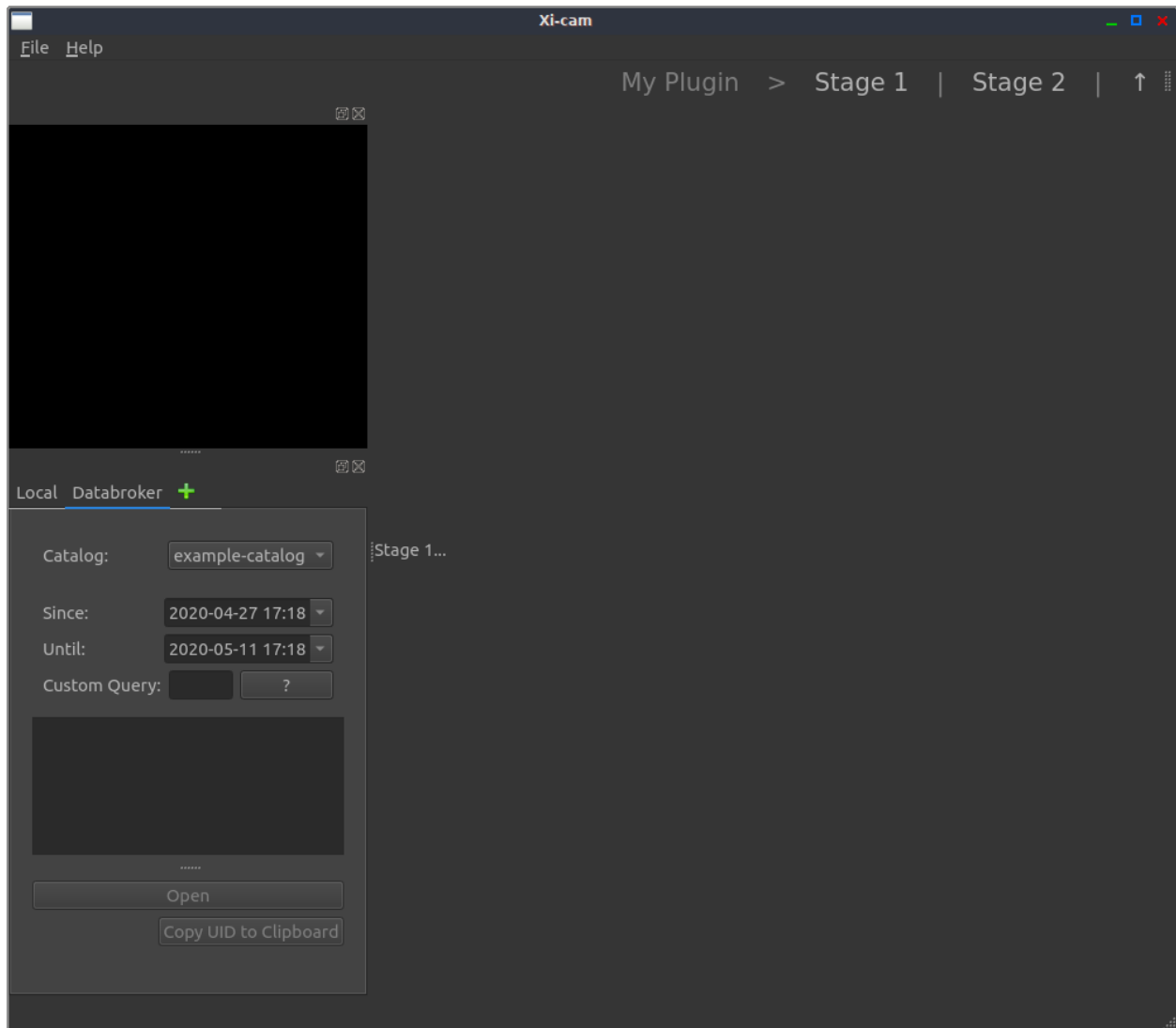
Fig. 1: MyPlugin's interface.

What Is a Stage?

Visually, a stage is a stand-alone interface for a `GUIPlugin`. A `GUIPlugin` must have at least one stage but may have multiple stages. With multiple stages, each stage has its own interface and each stage can be selected in the top bar of Xi-CAM.
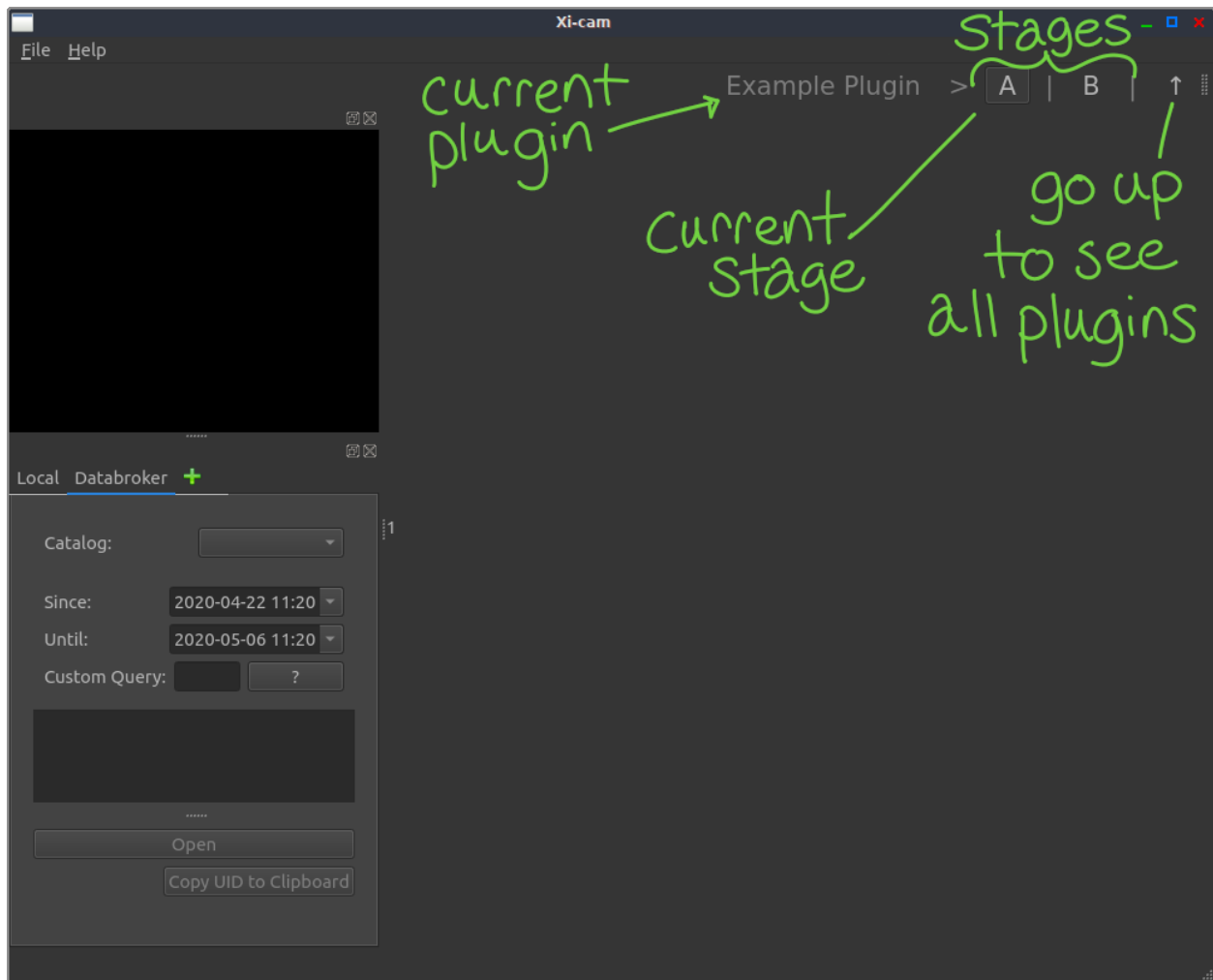
Stages for a `GUIPlugin` are accessible with `self.stages`. `self.stages` is a dictionary where each

- key is the name of the stage
- value is a `GUILayout`s

For example, we might define two stages as:

```
self.stages = {"A": GUILayout(QLabel("1")),
               "B": GUILayout(QLabel("2"))}
```

This will look like:

The interface of a plugin named "My Plugin" with multiple stages, "A" and "B". Note that "A" is currently selected, so we see the label "1" in the middle of the window.

What Is a GUILayout?

A `GUILayout` is a layout used to describe how widgets should be organized in a stage in a GUIPlugin.
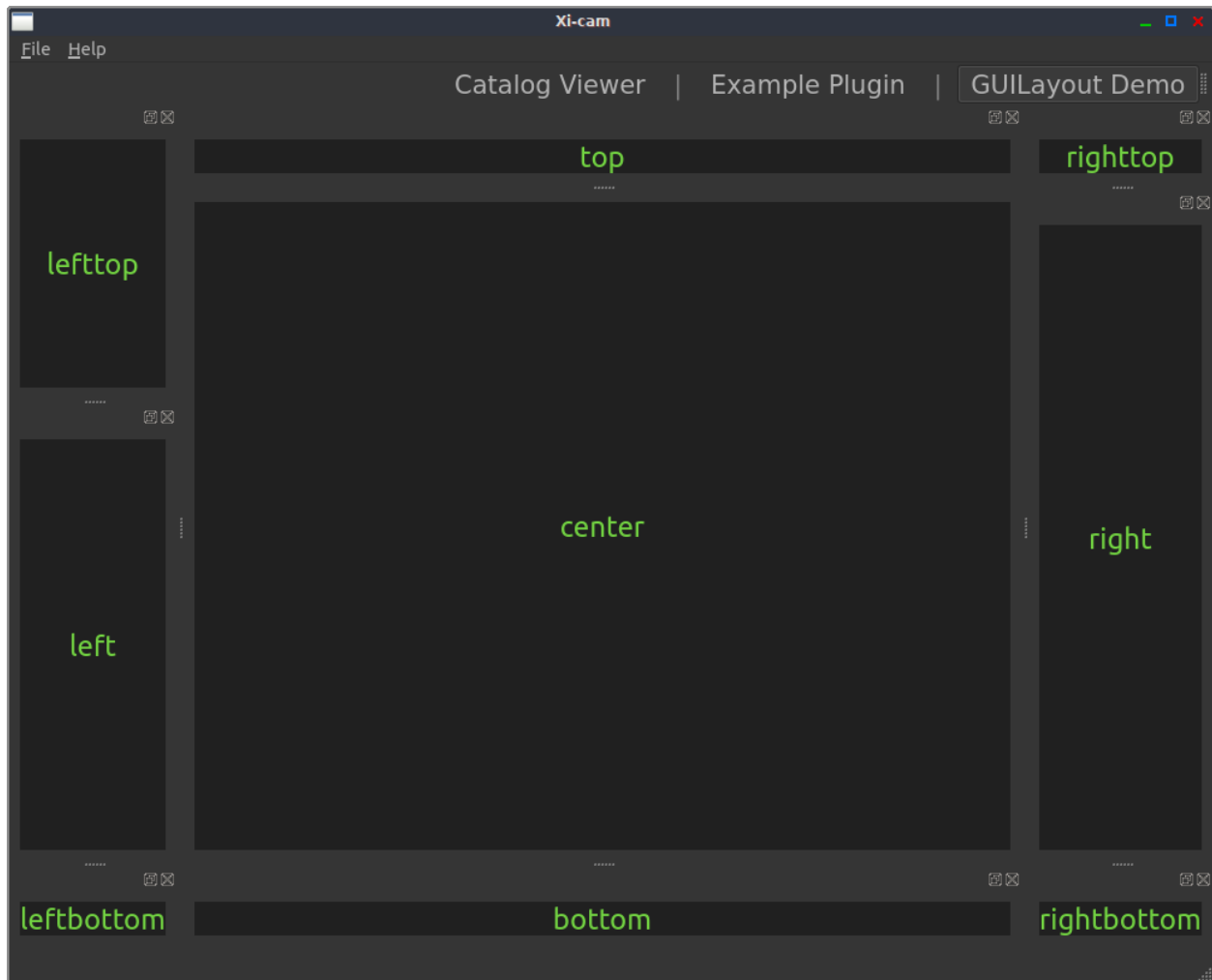
The layout corresponds to a 3x3 grid in the Xi-CAM main window, with the names center, left, right, lefttop, righttop, leftbottom, rightbottom. These names correspond to the arguments you can pass when creating a `GUILayout`.

You **must** provide at least one widget, which will be the center widget.

What Is a QLabel?

`QLabel` is a Qt widget provided by the Qt backend Xi-CAM makes use of. It acts a widget that holds simple text.

For more information on Qt, see Qt for Python Documentation.

How Do I Load Data into My Plugin?

In order to load data into a `GUIPlugin`, you must:

- have access to or configure a databroker catalog
- re-implement `appendCatalog` in your `GUIPlugin` derived class
    - this will need to have access to an internal widget to display the data
- have a `GUIPlugin` selected in Xi-CAM

Configuring a Databroker Catalog

For purposes of this documentation, we will be using a sample msgpack catalog and a starter catalog.yml file you can download.

For general help about databroker, catalogs, and configuration, there is excellent documentation here: https://nsls-ii.github.io/databroker/v2/index.html. Additional documentation about catalogs can be found here: https://intake.readthedocs.io/en/latest/index.html

Download	MD5
`349497da-ead2-4015-8201-4719f8a2de69.msgpack`	3a18341f570b100afbaff1c889e9b4f8
`catalog.yml`	c14814b4537810f14300f8c8d5949285

After downloading these files, we will want to do three things:

1. Decide where to put our data and move it there
2. Update our `catalog.yml paths` to have a path directory the data is in
3. Move our `catalog.yml` to a place it can be discovered

Moving the msgpack Data

You can choose where you'd like to copy or move your data. For purposes of this guide, we will create a new directory in our home called `catalogs` and move the downloaded msgpack file there.

Updating catalog.yml

Now that we've moved / copied our sample catalog msgpack file, we need to update our `catalog.yml` to tell it where it can find that data.

We will want to add a line under `paths` in `catalog.yml` that is the complete file path to the `catalogs` directory we added above.

Making catalog.yml Discoverable

To know where we can put our `catalog.yml` file, we can run the following in a Python interpreter:

```python
from databroker import catalog_search_path
print(catalog_search_path())
```

You can move the `catalog.yml` file in any of the paths listed. Note that typically there will be a more user-oriented path and a more global system-level path for the catalogs to find. You can copy the `catalog.yml` file to either (or both) path depending on how you want a machine set up.

Implementing appendCatalog

Let's implement the `appendCatalog` method in `MyPlugin` so we can load the catalog. We will also be adding a widget to view the loaded catalog.

Inside of the `MyPlugin` class (located in `xicam/my_plugin/__init__.py`), add the `appendCatalog` as follows:

```python
from qtpy.QtWidgets import QLabel

from xicam.core.msg import logMessage
from xicam.plugins import GUILayout, GUIPlugin
from xicam.gui.widgets.imageviewmixins import CatalogView


class MyPlugin(GUIPlugin):
    # Define the name of the plugin (how it is displayed in Xi-CAM)
    name = "My Plugin"

    def __init__(self, *args, **kwargs):
        self._catalog_viewer = CatalogView()
        self._stream = "primary"
        self._field = "img"

        catalog_viewer_layout = GUILayout(self._catalog_viewer)

        # Modify stages here
        # self.stages = {"Stage 1": GUILayout(QLabel("Stage 1..."))}
        self.stages = {"View Catalog": catalog_viewer_layout}

        super(MyPlugin, self).__init__(*args, **kwargs)

    def appendCatalog(self, catalog):
        self._catalog_viewer.setCatalog(catalog, self._stream, self._field)
        logMessage(f"Opening catalog with stream {self._stream} and field {self._field}.
↪")
```

API Reference

2.1.2 OperationPlugin Documentation

This documentation provides information on the foundational aspects of the OperationPlugin class, as well as a more detailed API reference.

If you are new to developing Xi-CAM plugins, it is recommended that you follow the quick-start documentation *first.*

For more general development resources, see the *Resources* page.

What Is an OperationPlugin?

An OperationPlugin can be thought of as a function with some extra annotations attached to it. When we want to define an OperationPlugin, we simply need to define a Python function, then add some additional syntax to the function to define things like inputs, outputs, descriptions of inputs/outputs, units, etc.

To achieve this, The OperationClass makes extensive use of Python decorators.

Where Is OperationPlugin?

```
xicam.plugins.operationplugin
```

What Does an OperationPlugin Look Like?

Let's start off with a simple function that computes the square of its input:

```python
def my_square(n):
    return n**2
```

Now, let's make this an OperationPlugin:

```python
from xicam.plugins.operationplugin import operation, output_names

@operation
@output_names("square")
def my_square(n):
    return n**2
```

That's it!

Notice the two decorators here: `@operation` and `@output_names`.

The `@operation` says that this function is now a Xi-CAM OperationPlugin. Any input arguments for the function will be the input names for the operation. In this case, our input is `n`. (This can actually be overwritten by using a different decorator, `@input_names`, which is described later.)

The `@output_names` allows us to name our outputs, in this case, `square`. This will be useful when connecting multiple operations together in a `Workflow`.

Default Input Values

If you want to provide your operation with default input values, you can use argument defaults in your function:

```python
from xicam.plugins.operationplugin import operation, output_names


@operation
@output_names("square")
def my_square(n = 0):
    return n**2
```

This provides this operation's `n` input with a default value of `0`.

Required and Highly-Used Decorators

In order to make a function an operation, the following decorators *must* be used:

- `@operation` – allows creation of operations from the function
- `@output_names` – defines the name of the output(s)

Additionally, although not required to for an operation, the following decorators are highly-recommended for use:

- `@display_name` – the name of the operation
- `@describe_input` – attach a description to the specified input (can be used multiple times)
- `@describe_output` – attach a description to the specified output (can be used multiple times)

Type Hinting (Optional)

With Python3 (3.5+), you can add type hinting to your code. In the context of Xi-CAM OperationPlugins, this can be used to make your operation code a little easier to read.

Let's use the `my_square` function we defined earlier in this operation:

```python
from xicam.plugins.operationplugin import operation, output_names


@operation
@output_names("square")
def my_square(n: int) -> int:
    return n**2
```

Note the `n:  int` and the `-> int:` here. These *suggest* (but do not mandate) that the input be an integer, and the output expected is an integer.

Again, these are not required, but they can help with readability and debugging your code.

For more information, see Python's typing module.

Example

A simple division operation that returns both the quotient and remainder.

This illustrates the use of multiple input/output descriptions and multiple outputs.

```python
from typing import Tuple
from xicam.plugins.operationplugin import describe_input, describe_output, display_name,
→operation, output_names


@operation
@output_names("quotient", "remainder")
@display_name("Division with Remainder")
@describe_input("dividend", "The number being divided.")
@describe_input("divisor", "The number to divide by.")
@describe_output("quotient", "The result of the division.")
@describe_output("remainder", "The remaining value.")
def my_divide(dividend: int, divisor: int = 1) -> Tuple[int, int]:
    quotient = int(dividend // divisor)
    remainder = dividend % divisor
    return quotient, remainder
```

How Do I Use an OperationPlugin?

Now that we've defined an operation, how do we actually use it?

When we define an operation using the `@operation` decorator around a function, we are defining a new operation class.

We can then create an operation object by using the syntax `func()`, where `func` is the name of the function in the operation.

Let's take our `my_square` operation (defined above) and create one:

```python
from xicam.plugins.operationplugin import operation, output_names


@operation
@output_names("square")
def my_square(n):
    return n**2


op = my_square()
```

Now that we have an operation object (instance), `op`, we can use it within a `Workflow`.

Let's create a `Workflow`, add our operation to it, then execute it.

```python
from xicam.core.execution import Workflow
from xicam.plugins.operationplugin import operation, output_names


@operation
@output_names("square")
def my_square(n):
    return n**2
```

```
op = my_square()
workflow = Workflow()
workflow.add_operation(op)
result = workflow.execute(n=11).result()
print(result)
```

We create a `my_square` operation, create a `Workflow`, and add the operation to the `Workflow`. Then, we execute the `Workflow`, sending in the input n=11, wait for the result, and print it.

(For purposes of this document, we won't cover `Workflow` in depth. More information about `Workflow` can be found in the *Workflow Documentation*.)

API Documentation

2.1.3 Workflow Documentation

This documentation provides information on the `Worfklow` class and its API reference.

If you are new to developing Xi-CAM plugins, it is recommended that you follow the quick-start documentation *first.*

For more general development resources, see the *Resources* page.

Note that the examples in this documentation can be run in a python interpreter outside of Xi-CAM (for demonstration purposes). Auxiliary support code to be able to do this is marked with a comment `# Only need if not running xicam`. When developing within Xi-CAM, you will **not** need the lines of code marked with that comment.

What Is a Workflow?

In Xi-CAM, a `Workflow` is represents a sequence of one or more `OperationPlugins` to execute. Basically, it allows you to process data through some pipeline of operations. Multiple operations can be linked together in a `Workflow`, provided that the connection between any two operations is compatible (based on inputs and outputs). Execution can be performed asynchronously or synchronously.

Where Is Workflow?

```
xicam.core.execution.Workflow
```

What Does a Workflow Look Like?

As mentioned previously, a Workflow can be thought of as a graph-like structure. We can add operations (*nodes*) and connect them with links (*edges*).

Example

```python
from xicam.core import execution  # Only need if not running xicam
from xicam.core.execution import localexecutor  # Only need if not running xicam
from xicam.core.execution import Workflow
from xicam.plugins.operationplugin import operation, output_names

execution.executor = localexecutor.LocalExecutor()  # Only need if not running xicam

# Define our operations
@operation
@output_names("sum")
def my_add(x, y):
    return x + y

@operation
@output_names("square_root")
def my_sqrt(n):
    from math import sqrt
    return sqrt(n)

# Instanciate operations
add_op = my_add()
sqrt_op = my_sqrt()

# Create a Workflow and add our operation instances to it
workflow = Workflow()
workflow.add_operations(add_op, sqrt_op)

# Link the "sum" output of add_op to the "n" input of sqrt_op
workflow.add_link(add_op, sqrt_op, "sum", "n")

# Execute the workflow, sending 1 and 3 as initial inputs to add_op (the first operation)
# This should give us sqrt(1 + 3) -> 2.0.
result = workflow.execute_synchronous(x=1, y=3)
print(result)  # Should be ({"square_root": 2.0},)
```

In this example, we use an addition operation and a square root operation in our Workflow. We want to add two numbers, then take the square root of the sum.

First, we instanciate our two operation types. This gives us an `add_op` operation object and a `sqrt_op` operation object.

Next, we add our operations to the workflow.

We then want to link the operations together so we first add two numbers, then take the square root of the result. We do this by connecting `add_op`'s "sum" output to `sqrt_op`'s "n" input.

Now that we have added our operations and connected them as we like, we can run our workflow. In this case, we will use `execute_synchronous` (there are other methods for execution which will be explained later).

However, if we just were to try `workflow.execute_synchronous()`, the workflow wouldn't know what the "x" and "y" inputs are supposed to be for the first operation, `add_op`.

We can either:

1. pass in data into the *first* operation(s)' inputs when we call an execute method on the workflow

2. have a GUI widget that exposes the operations through the GUI (such as `WorkflowEditor`), which can provide values directly to the operations' inputs

In this example, we used option 1 (for an example of option 2, see the `ExamplePlugin`'s use of `WorkflowEditor` in the *quick-start documentation*). To do this, we passed `x=1` and `y=3` to our `execute_synchronous` call, which provided values for the `invert` operation's `x` and `y` input arguments.

Useful Methods for Modifying the Workflow

Here is a condensed version of the various ways to modify a Workflow's operation and links. For more information, see the *API Reference*.

Adding, Inspecting, and Removing Operations

Adding operations:

- `add_operation` – add an operation to the `Workflow`
- `add_operations` – add multiple operations to the `Workflow`
- `insert_operation` – insert an operation at a specific index in the `Workflow`

Inspecting operations:

- `operations` – get the operations currently in the `Workflow`

Removing operations:

- `remove_operation` – remove an operation from the `Workflow`
- `clear_operations` – remove *all* operations from the `Workflow`

Adding, Inspecting, and Removing Links

Adding links:

- `add_link` – add a link between one operation's output and another's input
- `auto_connect_all` – try to automatically connect all the operations based on input/output names

Inspecting links:

- `links` – get all links in the `Workflow`
- `operation_links` – get all links connected to a specific operation in the `Workflow`
- `get_inbound_links` – get all incoming links to a specific operation in the `Workflow`
- `get_outbound_links` – get all outgoing links from a specific operation in the `Workflow`

Removing links:

- `remove_link` – remove a link from the `Workflow`
- `clear_operation_links` – remove all links for a specified operation in the `Workflow`
- `clear_links` – remove all links in the `Workflow`

Enabling and Disabling an Operation

It is possible to enable or disable operations. By default, all operations added to a `Workflow` are enabled. For more information, see the *API Reference*.

Executing a Workflow

When you execute a Workflow, the operations are executed based on how they are linked together.

There are a few ways to run a Workflow: `execute`, `execute_synchronous`, and `execute_all`.

Synchronous Execution

As we saw in our example earlier, we can use `execute_synchronous` to run a Workflow as a normal snippet of Python code. When this method is run, the we wait until we get a result back before the interpreter can continue running code.

Asynchronous Execution (Recommended)

The `execute` and `execute_all` methods are asynchronous, so they run in a separate thread. *This is highly beneficial in a GUI environment like Xi-CAM, since we don't want to block Xi-CAM's UI from responding,* and we could potentially offload execution onto a remote device. These methods take in several parameters; for now, we will focus on three of these parameters:

- `callback_slot` – Function to execute when the results of the Workflow are ready. The callback_slot gives you access to these results as a positional argument. **This is invoked for each result.** For example, let's say you have a crop operation that takes in an image (array) as an input parameter. You could pass in a list of images to crop to `Workflow.execute_all()`, and the callback_slot will be invoked for each of the images in the passed list. Basically, you will get a cropped image for each image sent into the workflow.

- `finished_slot` – Function to execute when the internal thread in the Workflow has finished its execution (all of the operations are done). **This occurs once during a Workflow's execution.**

- `kwargs` – Any additional keyword arguments to pass into the method; these usually correspond with the entry operations' inputs (as we saw in our example earlier).

The primary difference between `Workflow.execute` and `Workflow.execute_all` is that `execute_all` will run multiple times for the `kwargs` passed in. This means the `kwargs` must have an iterable value. Let's look at some examples.

Example for execute

Let's revisit our addition and square root workflow from earlier but make it asynchronous:

```python
from qtpy.QtWidgets import QApplication  # Only need if not running xicam
from xicam.core import execution  # Only need if not running xicam
from xicam.core.execution import localexecutor  # Only need if not running xicam
from xicam.core.execution import Workflow
from xicam.plugins.operationplugin import operation, output_names

qapp = QApplication([])  # Only need if not running xicam
execution.executor = localexecutor.LocalExecutor()  # Only need if not running xicam
```

```python
# Define our operations
@operation
@output_names("sum")
def my_add(x, y):
    return x + y


@operation
@output_names("square_root")
def my_sqrt(n):
    from math import sqrt
    return sqrt(n)

# Define callback slot (when a result is ready)
def print_result(*results):
    print(results)


# Define finished slot (when the workflow is entirely finished)
def finished():
    print("Workflow finished.")

# Instanciate operations
add_op = my_add()
sqrt_op = my_sqrt()

# Create a Workflow and add our operation instances to it
workflow = Workflow()
workflow.add_operations(add_op, sqrt_op)

# Link the "sum" output of add_op to the "n" input of sqrt_op
workflow.add_link(add_op, sqrt_op, "sum", "n")

# Execute the workflow, sending 1 and 3 as initial inputs to add_op (the first operation)
# This should give us sqrt(1 + 3) -> 2.0.
workflow.execute(callback_slot=print_result,
                 finished_slot=finished,
                 x=1,
                 y=3)
```

This will print out:

```
({'square_root': 2.0},)
Workflow finished.
```

Notice that we've added two new functions for our callback slot and our finished slot. `print_result` will be called when the workflow has finished its execution and the result is ready. `finished` will be called when the workflow has finished execution for all of its input data. In this case, we have only one set of input data, x=1 and y=3.

(Also note that we have an additional import and that we are creating a QApplication; this is not needed when working within Xi-CAM).

Example for execute_all

Now, let's say we want to do this addition and square root workflow for multiple sets of x and y inputs. We can use `execute_all` to do this:

```python
from qtpy.QtWidgets import QApplication  # Only need if not running xicam
from xicam.core import execution  # Only need if not running xicam
from xicam.core.execution import localexecutor  # Only need if not running xicam
from xicam.core.execution import Workflow
from xicam.plugins.operationplugin import operation, output_names

qapp = QApplication([])  # Only need if not running xicam
execution.executor = localexecutor.LocalExecutor()  # Only need if not running xicam

# Define our operations
@operation
@output_names("sum")
def my_add(x, y):
    return x + y


@operation
@output_names("square_root")
def my_sqrt(n):
    from math import sqrt
    return sqrt(n)

# Define callback slot (when a result is ready)
def print_result(*results):
    print(results)

# Define finished slot (when the workflow is entirely finished)
def finished():
    print("Workflow finished.")

# Instanciate operations
add_op = my_add()
sqrt_op = my_sqrt()

# Create a Workflow and add our operation instances to it
workflow = Workflow()
workflow.add_operations(add_op, sqrt_op)

# Link the "sum" output of add_op to the "n" input of sqrt_op
workflow.add_link(add_op, sqrt_op, "sum", "n")

# Execute the workflow, sending the inputs x=1,y=3; x=10,y=15; x=50,y=50.
# This should give us 2.0, 5.0, and 10.0.
workflow.execute_all(callback_slot=print_result,
                     finished_slot=finished,
                     x=[1, 10, 50],
                     y=[3, 15, 50])
```

This will print out:

```
({'square_root': 2.0},)
({'square_root': 5.0},)
({'square_root': 10.0},)
Workflow finished.

Notice that we've just changed `execute` to `execute_all`, and we've modified the `x`␣
↪and `y` values to be lists.
Now, we will have three executions: `x=1 y=3`, `x=10 y=15`, and `x=50 y=50`.
Each time one of these executions finishes, our callback slot `print_result` is called.
When the workflow is finished executing everything, then our finished slot `finished` is␣
↪called.
```

API Reference

2.1.4 Install Xi-CAM

If you haven't already installed Xi-CAM, follow the installation instructions for your operating system:

Installing Xi-CAM for Linux

Installing Xi-CAM requires a few system components to be installed. After successfully installing these components and Xi-CAM, you will be ready to start developing Xi-CAM plugins!

Install python3

First, ensure that you have **python3.8** installed on your system.

Consult your specific distribution's package manager for installing python3.

Create and Activate a Virtual Environment

Creating a virtual environment allows you to install and uninstall packages without modifying any packages on your system. This is *highly* recommended.

There are a couple of ways to create a virtual environment:

1. via the **venv** module provided with python3
2. via **conda** (you will need to install this from anaconda.org or miniconda.org)

In the commands below, we will create a new environment called **xicam** in your home directory, and then activate the environment.

Once an environment is activated, any packages installed through pip will be installed into this sequestered xicam environment. *(If using conda, you can install either with pip or conda.)*

virtualenv

If you would like to create a virtual environment, run the following:

```
cd ~
python3 -m venv xicam
source xicam/bin/actviate
```

conda

If you would like to create an environment through conda, run the following:

```
cd ~
conda create -n xicam python=3.8
conda activate xicam
```

Install Python Qt Bindings

Xi-CAM depends on a GUI application framework called Qt; you will need to install one of the python bindings for Qt (PyQt5 or PySide2) in order to run Xi-CAM.

*Make sure that you have activated the **xicam** environment.*

For example, you can install the **PyQt5** pip package as follows:

```
pip install PyQt5
```

Install the Xi-CAM package

Now that we have activated a new **xicam** environment and installed **PyQt5**, we can install Xi-CAM:

```
pip install xicam
```

To ensure everything is installed correctly, you can run Xi-CAM as follows:

```
xicam
```

Where Do I Go from Here?

You are now ready to start developing plugins for Xi-CAM!

To learn about developing plugins for Xi-CAM, see the *Quick Start Guide*.

Installing Xi-CAM for MacOS

Installing Xi-CAM requires a few system components to be installed. After successfully installing these components and Xi-CAM, you will be ready to start developing Xi-CAM plugins!

Install python3

First, ensure that you have **python3.8** installed on your system.

The quickest way to do this is by downloading and running the python.org installer for python3. The python3.8 macOS 64-bit installer can be found here.

Alternatively, you can install XCode and homebrew to manage multiple versions of python on your system.

Create and Activate a Virtual Environment

Creating a virtual environment allows you to install and uninstall packages without modifying any packages on your system. This is *highly* recommended.

There are a couple of ways to create a virtual environment:

1. via the **venv** module provided with python3

2. via **conda** (you will need to install this from anaconda.org or miniconda.org)

Using the **Terminal** application, we will create a new environment called **xicam** in your home directory, and then activate the environment.

Once an environment is activated, any packages installed through pip will be installed into this sequestered xicam environment. *(If using conda, you can install either with pip or conda.)*

virtualenv

If you would like to create a virtual environment, run the following:

```
cd ~
python3 -m venv xicam
source xicam/bin/actviate
```

conda

If you would like to create an environment through conda, run the following:

```
cd ~
conda create -n xicam python=3.8
conda activate xicam
```

Install Python Qt Bindings

Xi-CAM depends on a GUI application framework called Qt; you will need to install one of the python bindings for Qt (PyQt5 or PySide2) in order to run Xi-CAM.

*Make sure that you have activated the **xicam** environment.*

For example, you can install the **PyQt5** pip package as follows:

```
pip install PyQt5
```

Install the Xi-CAM package

Now that we have activated a new **xicam** environment and installed **PyQt5**, we can install Xi-CAM:

```
pip install xicam
```

To ensure everything is installed correctly, you can run Xi-CAM as follows:

```
xicam
```

Where Do I Go from Here?

You are now ready to start developing plugins for Xi-CAM!

To learn about developing plugins for Xi-CAM, see the *Quick Start Guide*.

Installing Xi-CAM for Windows

Installing Xi-CAM requires a few system components to be installed. After successfully installing these components and Xi-CAM, you will be ready to start developing Xi-CAM plugins!

Install python3

On Windows, a great way to manage python installations and packages is through Anaconda. Follow their Windows installation instructions, which will install the conda package manager, Anaconda, Anaconda Prompt, and Anaconda Navigator.

- Anaconda – A package that provides conda and several common python packages
- Anaconda Prompt – A command line shell for managing conda environments and installing packages
- Anaconda Navigator – A GUI for managing conda environments and installing packages

Open the Anaconda Prompt program.

Then, create a new environment called **xicam**. This creates a sequestered space on your system to install xicam and its dependencies without modifying any of your system's libraries.

Next, activate the environment. This tells the system to use the libraries and applications inside the environment.

```
conda create -n xicam python=3.8
conda activate xicam
```

Install Python Qt Bindings

Xi-CAM depends on a GUI application framework called Qt; you will need to install one of the python bindings for Qt (PyQt5 or PySide2) in order to run Xi-CAM.

*Make sure that you have activated the **xicam** environment.*

In your open Anaconda Prompt window, install the **pyqt** conda package as follows:

```
conda install pyqt
```

Install the Xi-CAM package

Now that we have activated a new **xicam** environment and installed **pyqt**, we can install Xi-CAM using a python package management tool called **pip**. Run the following in your open Anaconda Prompt.

```
pip install xicam
```

To ensure everything is installed correctly, you can run Xi-CAM as follows:

```
xicam
```

Where Do I Go from Here?

You are now ready to start developing plugins for Xi-CAM!

To learn about developing plugins for Xi-CAM, see the *Quick Start Guide*.

Copyable Instructions

Anaconda Prompt:

```
cd ~
conda create -n xicam
conda activate xicam

conda install -c conda-forge pyqt

pip install xicam

xicam
```

2.1.5 Overview

In this guide we will:

- Explore the main window of Xi-CAM

- Download and install an Example Plugin

- Configure a sample catalog so we can load data

- Explore the Example Plugin

Key Concepts

Here is a quick overview of some concepts that will be explored in this guide. Note that more documentation is available for each of these concepts.

We have one `GUIPlugin` (`ExamplePlugin`) - this will be a plugin that you will be able to select and see within Xi-CAM. The layout of the `GUIPlugin` is defined by a `GUILayout`.

We have a few `OperationPlugins` (`invert` and `random_noise`) - These plugins are basically functions that take in data and output derived data.

We also need a way to actually run data through the operations. To do this, we have a `Workflow` (`ExampleWorkflow`) - this contains linked operations to execute (can be thought of like a pipeline).

2.1.6 Looking at Xi-CAM's Main Window

Let's look at what the main window in Xi-CAM looks like first:

When Xi-CAM finishes loading, we see the window as shown above. Any installed **GUIPlugins** will be visible (and selectable) at the top (note that you will probably not have any installed yet).

We can also see some of the default widgets provided:

- a welcome widget in the *center* of the window

- a preview widget in the top-left (*lefttop*) of the window, which shows a sample of selected data in the data browser widget

- a data browser widget on the *left* of the window, which can show available databroker catalogs

Quick GUILayout Overview

We mentioned the terms *center*, *lefttop*, and *left* above. These correspond to positions in a `GUILayout`. Here is a *quick* overview of how the Xi-CAM main window is organized:

You can see that the layout of Xi-CAM follows a 3x3 grid, where each section is named according to its orientation in relation to the center of the window.

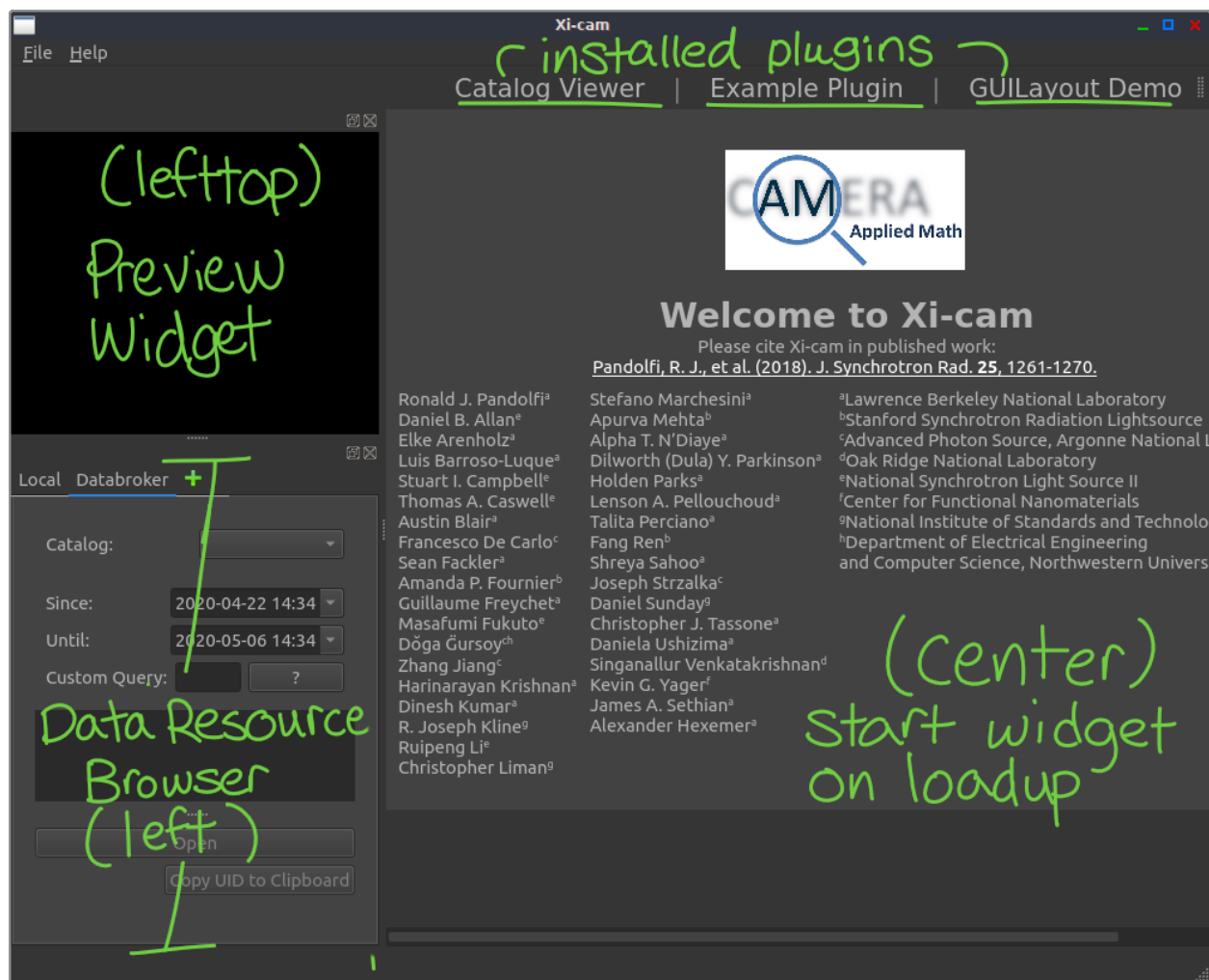(Note that any `GUIPlugins` you create will have one or more of these `GUILayouts`).

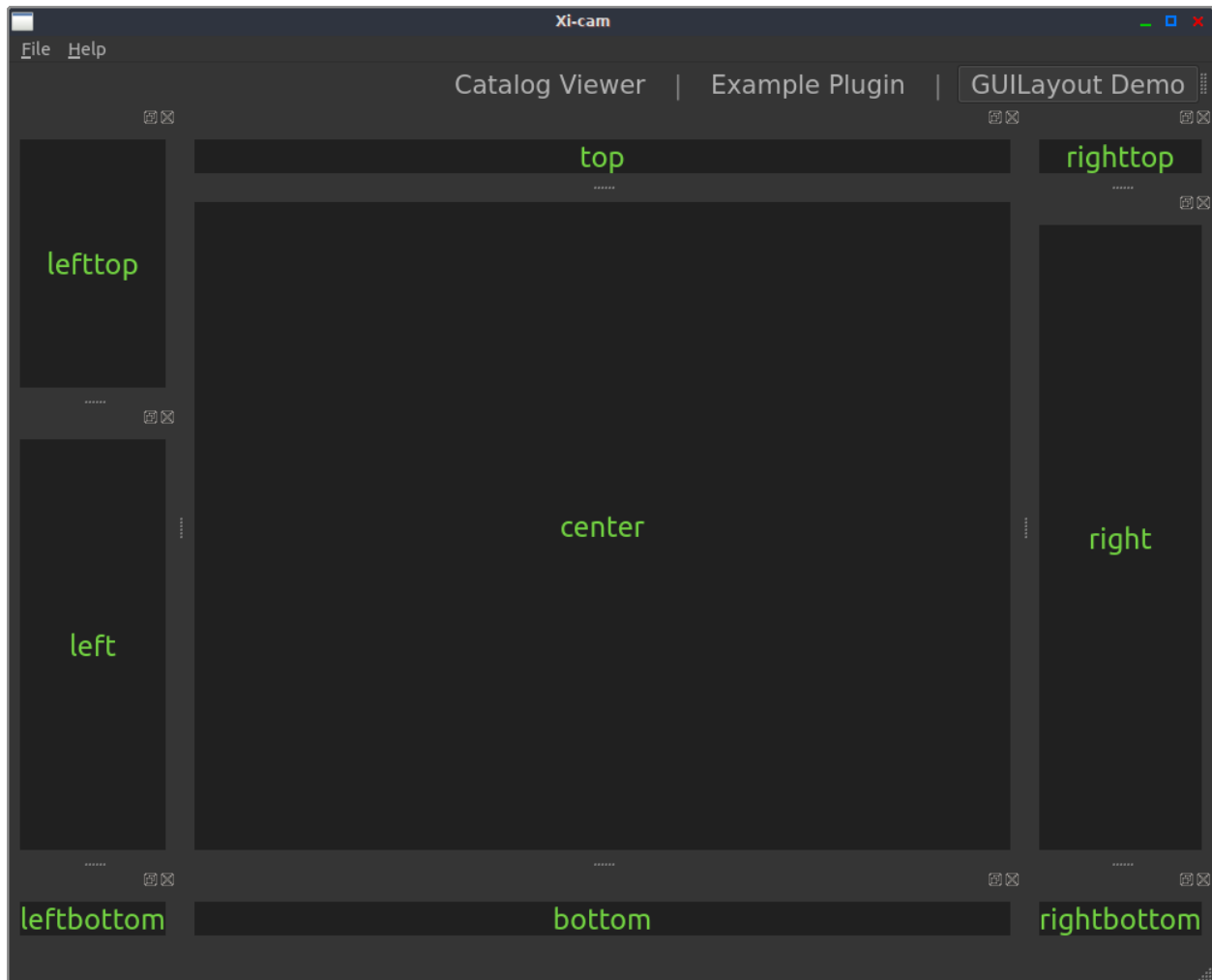Fig. 2: The main window of Xi-CAM after it has finished loading.

Fig. 3: The layout of Xi-CAM's main window.

Xi-CAM Menu Bar

At the top of the main window, there is a menu bar that contains some helpful items.

In the `File` item you can find `Settings` for Xi-CAM. This includes things like:

- Logging configuration - where to find the log files, what type of logging record. . .

- Theme - change the appearance of Xi-CAM

- Device settings - allows managing different devices (detectors) (if you have Acquire or SAXS installed)

In the `Help` item you can find a link to the Xi-CAM documentation, a way to contact the development team, and versioning / licensing information for Xi-CAM.
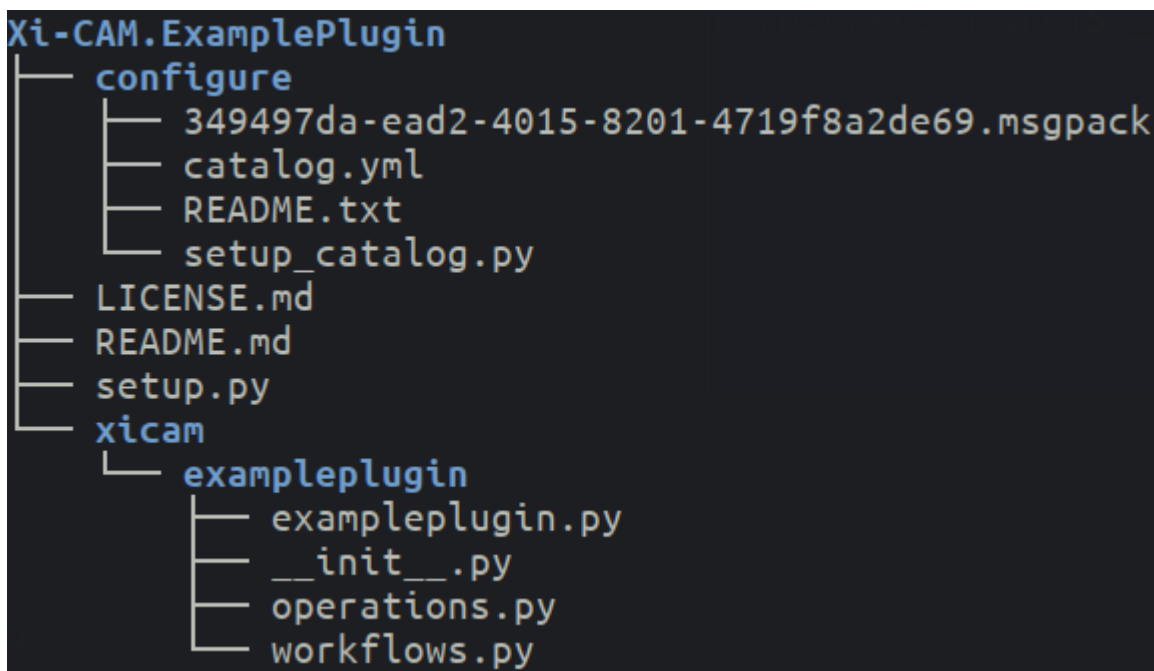
2.1.7 Download and Install the ExamplePlugin

Now that we have looked at the main window and its layout, let's download the Example Plugin.

```
cd ~
git clone https://github.com/Xi-CAM/Xi-CAM.ExamplePlugin
cd Xi-CAM.ExamplePlugin
```

What's Inside the ExamplePlugin Repository

The repository will contain the following:

Fig. 4: The contents of the ExamplePlugin repo when you clone it.

At the top there are a few files and directories:

- `setup.py` - describes how to install this as a python package; **also used to register plugins (via entry points)**.

- `configure` - special directory for this example, helps set up a catalog

---

- `xicam` - directory that acts as a python namespace package

In `xicam`, there is a `exampleplugin` subpackage that contains:

- `__init__.py` - makes `exampleplugin` a python package; also exposes the `ExamplePlugin` class

- `exampleplugin.py` - module that contains the `ExamplePlugin` GUI plugin

- `operations.py` - module that contains the example `OperationPlugins`

- `workflows.py` - module that contains the example `Workflows`

How Do I Install the Example Plugin?

So far, we have only downloaded the Example Plugin - we still need to install it so Xi-CAM can find it and load it.

We can install downloaded plugins using a *pip editable install*:

```
pip install -e .
```

This uses Python's **entry points** mechanism to register plugins for Xi-CAM to see.

Exploring the Example Plugin Interface

When you run `xicam`, you should now see the Example Plugin available at the top right of the main window.

Select it and you should the Example Plugin layout:

In the center, we have a `CatalogView` that will be used to display loaded data. On the right, there is a `WorkflowEditor` that shows the operations in the workflow and allows for running the workflow. At the bottom, there is a `DynImageView`, which will be used to display the results data.

2.1.8 How Do I Load Data?

Now that we have the Example Plugin installed, we need to have data to load into it.

For purposes of this guide, we will be configuring a catalog called "example_catalog."

For more information, see the Bluesky DataBroker documentation.

Configuring a Catalog

There is a `configure/` directory in the repository we cloned. This contains a catalog configuration file, a msgpack catalog, and a script.

Feel free to inspect the script before you run it; it will attempt to set up a msgpack catalog source for Xi-CAM to use:

```
cd configure
python setup_catalog.py
cd ..
```
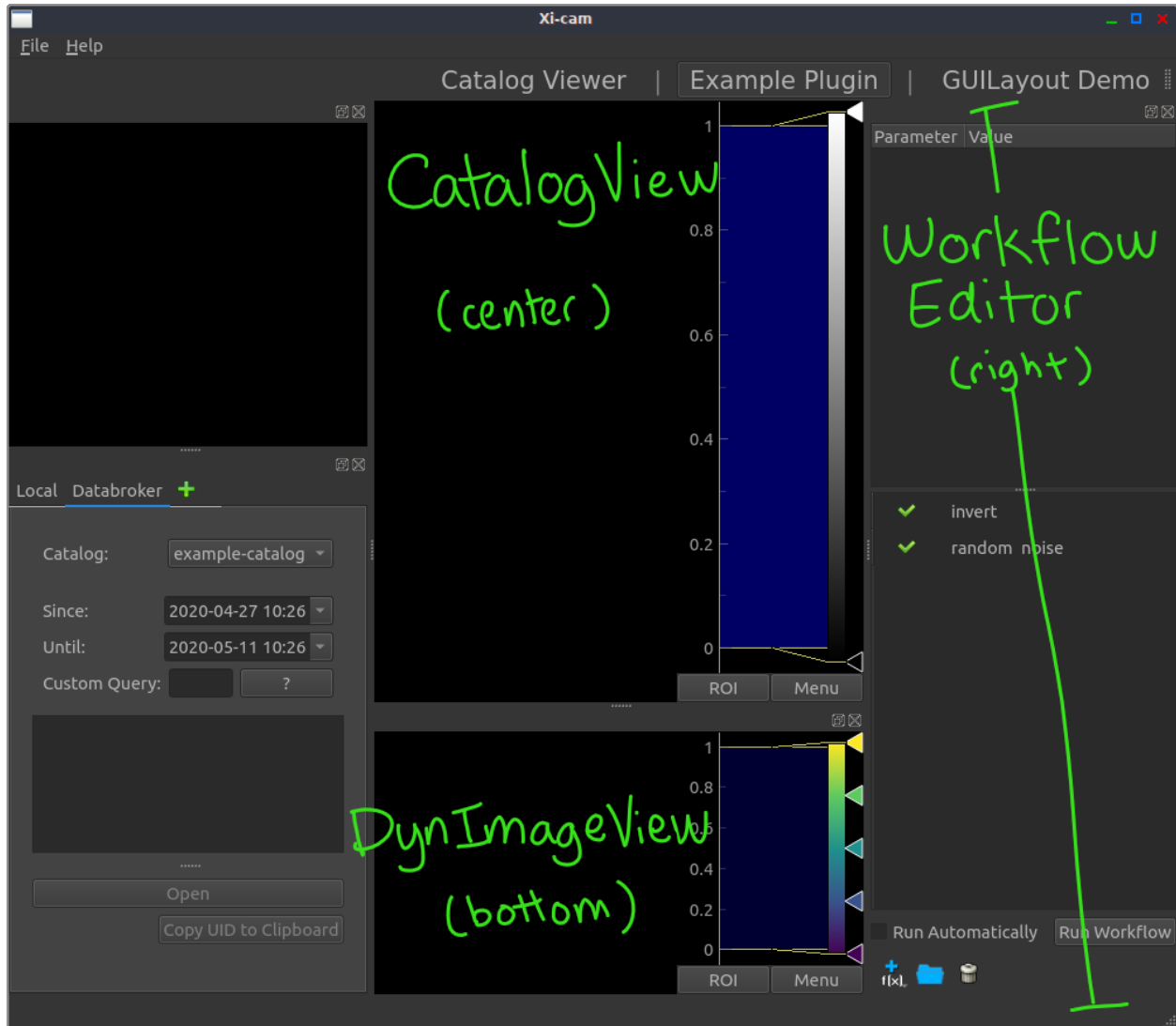
Fig. 5: The Example Plugin. Uses a CatalogView, DynImageView, and WorkflowEditor as widgets in its layout.

Loading a Catalog from the Data Resource Browser

Now that we've configured the catalog, let's make sure that Xi-CAM can see it.

When loading a catalog into Xi-CAM, you must have a GUIPlugin active. Let's select our "Example Plugin."

Look at the *Data Resource Browser* on the left hand side of the window. The *Data Resource Browser* gives us access to two different types of data browsers by default:

- a bluesky browser for catalogs (adapted from work done by NSLS-II)

- a local file browser

After configuring our example catalog, the bluesky catalog browser should have the text "example_catalog" in the *Catalog* drop-down box.

Notice that it also has two text inputs, *Since* and *Until*. Our example catalog was created in the beginning of 2020. In order to see the data (catalogs) our "example_catalog" contains, we need to change the *Since* text input.

Change it's value to "2020-01-01". This will now look for any data that was created since the start of 2020. After making this change, the example_catalog will be re-queried for data created within these new dates.

You should see a catalog show up in the table below with the id *349497da*. If you *single-click* the row in the table to highlight it, more information and a preview of the data should be shown as well. You can then open it with the "Open" button.

You should see Clyde the cat loaded into the center `CatalogView`.

Running a Workflow

Our Example Plugin has one internal workflow, the `ExampleWorkflow`. The `ExampleWorkflow` contains two `OperationPlugins` (operations):

- `invert` - inverts its input image

- `random_noise` - applies random noise to its input image, has a "strength" parameter to define how much noise to apply to the image

This workflow is exposed in the GUI with a `WorkflowEditor` on the right side of the layout.

Now that we have loaded some data, let's run our workflow by clicking the "Run Workflow" button.

You should see an inverted picture with some random noise added to it.

Note that you can adjust the amount of random noise by selecting the "random_noise" text in the `WorkkflowEditor`, then changing the value of "strength" that shows up in the parameter tree above.

2.1.9 Examining the Code

Let's take a quick look at how the code is implemented for our Example Plugin.

The code for this particular plugin is organized into three modules:

- `exampleplugin.py` - Defines the `ExamplePlugin` (the GUIPlugin)

- `operations.py` - Defines two `OperationPlugins`: invert and random_noise

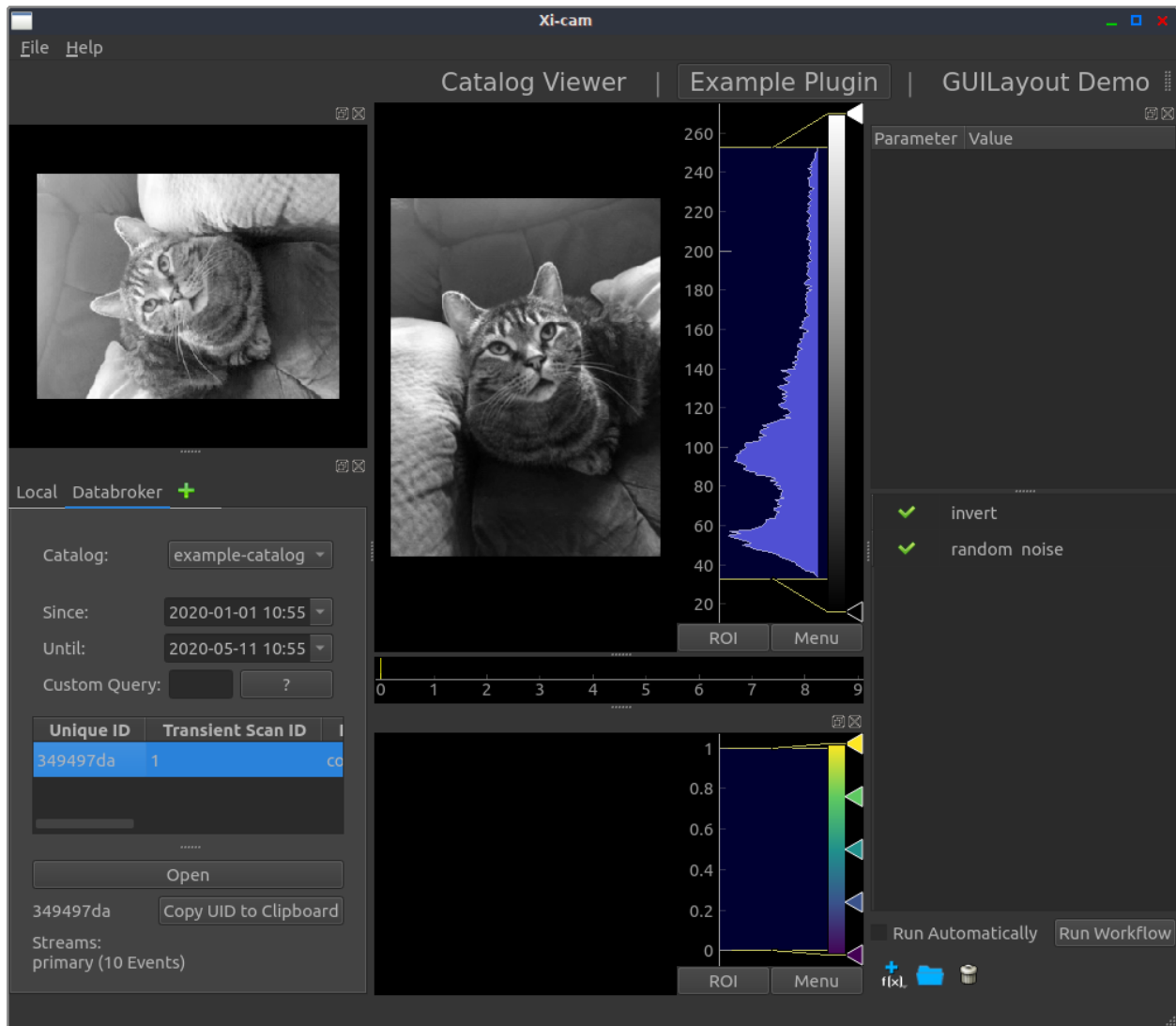- `workflows.py` - Defines an `ExampleWorkflow` with the invert and random_noise operations

Fig. 6: Here we see catalog 349497da in the DataResourceBrowser. It has one stream (primary) with 10 events in it. A preview shows the first frame of the data, and the opened data appears in the center.
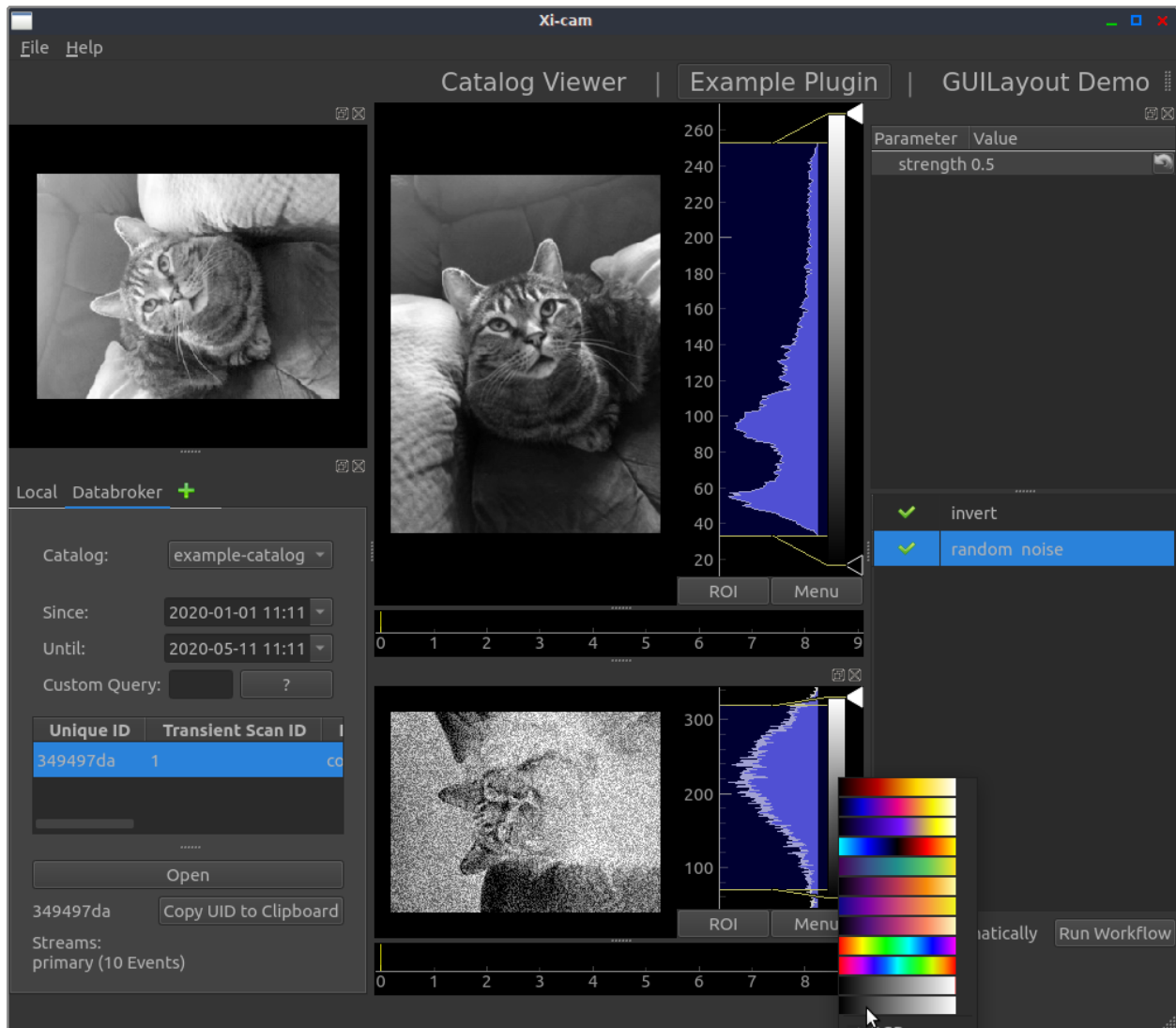
Fig. 7: The result data after running the workflow. Note that the color lookup table can be changed by right-clicking the gradient bar.

operations.py

Here we define `OperationPlugins` (or operation)

An operation can be thought of as a function; input data is sent into the operation, and the operation generates some output with the given input.

When defining an `OperationPlugin`, we use Python decorators (the `@` seen in the code). At the very least, you must provide the `@operation` and `@output_names` decorators for an operation.

workflows.py

Here we define an `ExampleWorkflow`.

We add our two operations to the `ExampleWorkflow`, then connect them so that `invert`'s "output_image" value is sent to `random_noise`'s input image argument.

exampleplugin.py

Here we define the gui plugin `ExamplePlugin`.

We provide a `name` for the plugin, which will display as "Example Plugin" in Xi-CAM.

We define our widgets, our layout, and any internal objects we might need (like the workflow) inside of our `__init__` method. We connect the `WorkflowEditor`'s `sigRunWorkflow` signal to our `run_workflow` method. This means whenever "Run Workflow" is clicked in the WorkflowEditor, our `ExamplePlugin`'s `run_workflow` method will be called.

We also define a `results_ready` method that will be called whenever our workflow has finished executing its operations. Providing `callback_slot=self.results_ready` in our `execute` call sets up this connection for us.

2.2 Entry Points

An entry point is a mechanism that can be used to make objects discoverable by a common interface / name.

2.2.1 Xi-CAM Entry Points

In Xi-CAM, you can define entry points and then run `pip install -e .` in your plugin package directory to register plugins. This allows Xi-CAM to see your plugins when it loads. Entry points are defined in `setup.py` files, in the `entry_points` key.

Let's look at an example repository and `setup.py`:

```
setup.py
xicam/
    myplugin/
        __init__.py          - defines MyGUIPlugin (also marks this directory as a
→Python module)
        operations/
            __init__.py      - marks this directory as a Python module
            edge_detection.py - contains edge detection operations (laplace and sobel)
        workflows/
```

(continues on next page)

```
        __init__.py      - (marks this directory as a Python module)
        myworkflow.py    - defines MyWorkflow.py
```

Here's what our `entry_points` might look like in `setup.py`:

```
entry_points = {
    "xicam.plugins.GUIPlugin": ["myguiplugin = xicam.myplugin:MyGUIPlugin"],
    "xicam.plugins.OperationPlugin": [
        "laplace_operation = xicam.myplugin.operations.edge_detection:laplace",
        "sobel_operation = xicam.myplugin.operations.edge_detection:sobel"
    ],
}
```

As seen above, `entry_points` is a dictionary, where each key is an entry point and each value is a list of objects / types being registered to that entry point.

The syntax is: `"entry point name":  ["some_identifier = package.subpackage.module:ClassName"]`.

In this case, we are registering `MyGUIPlugin` to the `xicam.plugins.GUIPlugin` entry point. Similarly, we are registering the `laplace` and `sobel` operations to the `xicam.plugins.OperationPlugin` entry point.

Note that `Workflows` are not registered in this way; they are not Xi-CAM plugins.

Whenever you modify entry points, you must reinstall your package. You can do this by running `pip install -e .` in your package directory.

When Xi-CAM loads, it will see the `xicam.plugins.GUIPlugin` entry point key and load in `MyGUIPlugin` defined (in the value). Similarly, Xi-CAM will see the `xicam.plugins.OperationPlugin` entry point key and load in the `laplace` and `sobel` operations.

2.2.2 More Information

For more information about entry points, see the following:

- https://entrypoints.readthedocs.io/en/latest/
- https://packaging.python.org/specifications/entry-points/
- https://amir.rachum.com/blog/2017/07/28/python-entry-points/

2.3 Data ingestion in Xi-CAM

2.3.1 What is an ingestor?

The `ingestor` design is specified by the Databroker team to provide an entrypoint for data generated external from the Bluesky environment. An ingestor is a `Callable` that accepts a URI (often a local file path) and yields `(name, doc)` pairs. The yielded data follows the Bluesky `event-model` structure (see event-model documentation). Synthesizing these event-model documents is made easier with the `RunBuilder` (see Bluesky-Live documentation).

2.4 Resources

2.4.1 Example Xi-CAM Plugins

- Xi-CAM CatalogViewer Plugin - Example of a simple single-stage GUIPlugin.
- Xi-CAM Log Plugin - Example of another simple single-stage GUIPlugin.
- Xi-CAM BSISB Plugin - Example of a multi-stage GUIPlugin with more functionality.
- Xi-CAM NCEM Plugin - Another example of a multi-stage GUIPlugin with more functionality.

2.4.2 Git

- Try GitHub - Landing page for some introductions and resources about git and GitHub.
- Git Handbook - An introduction to git and GitHub.

2.4.3 NSLS-II

Useful resources about NSLS-II software that Xi-CAM uses.

- Databroker Catalog - Describes how to configure and use a databroker catalog.
- Event Model - Describes an event-based data model.
- Bluesky Documents - Describes what a bluesky document is.

2.4.4 Python

Here are a few resources regarding object-oriented programming with Python3. Feel free to look through these or even through resources you find on your own if you are interested.

- Python OOP Introduction and Tutorial -
- Presentation on OOP in Python -
- Python OOP

2.4.5 Qt

Qt is a framework written in C++ for developing graphical user interfaces. PySide2 and PyQt5 are two different python bindings to the Qt C++ API. QtPy is a wrapper that allows for writing python Qt code with either PyQt5 or PySide2 installed.

Xi-CAM uses QtPy to interact with different Python bindings to Qt. QtPy allows you *"to write your code as if you were using PySide2 but import Qt modules from qtpy instead of PySide2 (or PyQt5)"*. The references below show PySide2 examples and documentation; when writing a Xi-CAM plugin, make sure to use the `qtpy` modules when importing.

- PySide2 Documentation - Documentation for PySide2. Since the QtPy API resembles PySide2, this documentation is helpful for looking up python Qt modules and classes that you may use.
- PyQt5 GUI Tutorial - Introductory tutorial for learning the basic concepts of Qt. *Note: this tutorial is written for PyQt5, remember to import from `qtpy` instead of `PyQt5` or `PySide2` when writing code for Xi-CAM.*
- PySide2 Simple Clickable Button - Short tutorial that describes the concept of signals and slots in Qt and shows how to create a button that responds to clicking.

- PyQtGraph - Documentation for the pyqtgraph package, which relies on Qt and provides basic data visualization (plotting) and various widgets (helpful for writing Xi-CAM GUIPlugins).

THREE

LINKS

- Xi-CAM GitHub Organization

INDICES AND TABLES

- genindex
- modindex
- search