PCOT design notes

James Finnis (jcf12@aber.ac.uk)

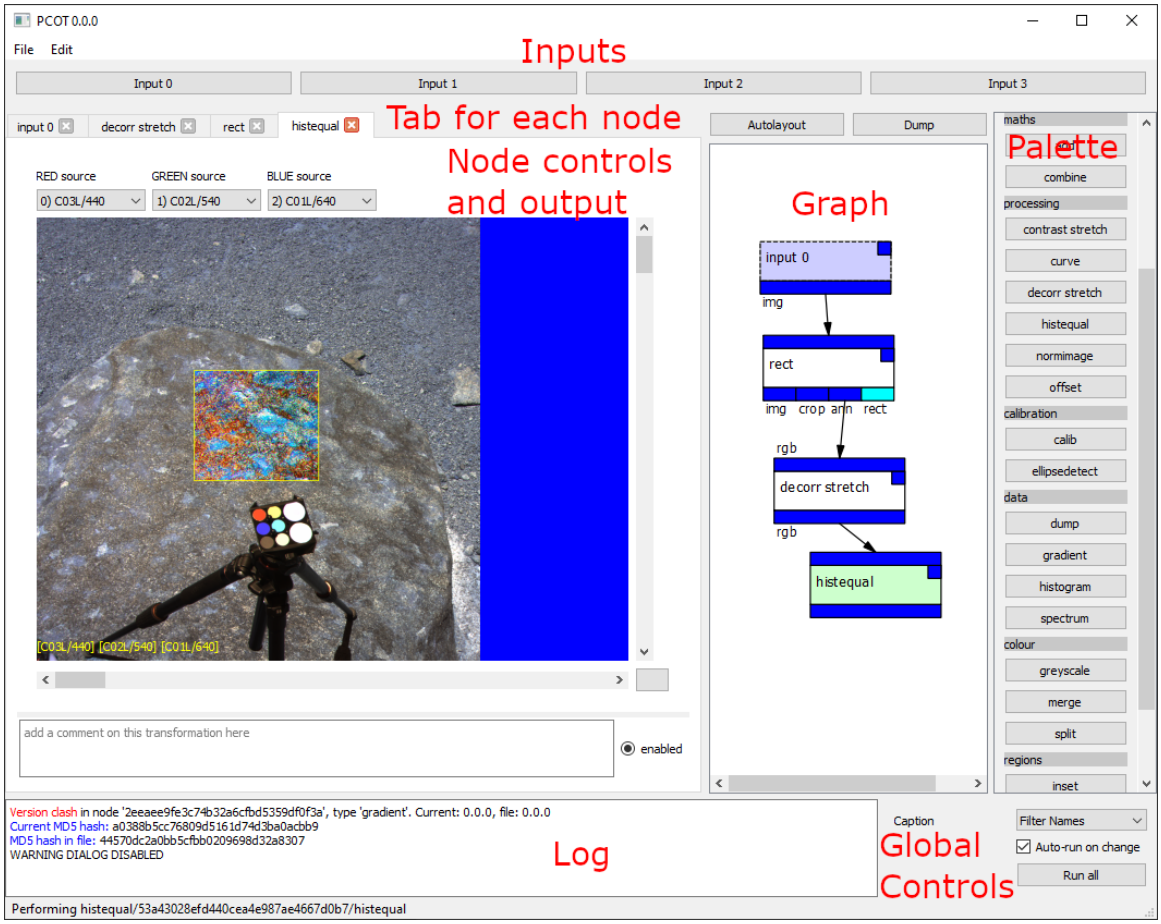
December 23, 2021

Contents

1 Introduction

These notes provide some architectural details for PCOT to help maintainers. I'll try to keep them up to date.

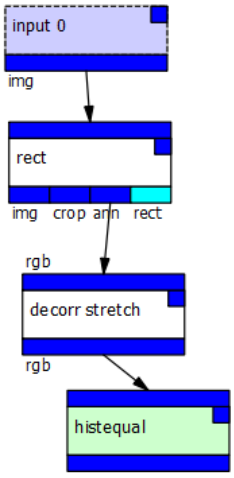
PCOT is based around a directed graph of nodes which perform transformations of data. For this reason, the nodes are sometimes called "transforms" and are represented by the `XForm` class in the code. Usually the data in question is an image, or rather an "image cube": these have an arbitrary number of channels, not just the typical RGB or greyscale. However, the data can be anything at all — it depends on the node. There is some typechecking when constructing the graph: for exampel, you can't connect a "rectangle" output to an "image" input. The entire application is shown in Fig. 1. On the right is a "palette" from which nodes can be selected to add to the graph, while on the left is an area which can show controls for each node in the graph, while in the centre-right is the graph itself. This is shown in more detail in Fig. 2.



app.png

Figure 1: The PCOT application

This will take an image from one of the data inputs — these exist independently of the graph proper and are configured with the four buttons at the top of the window, see Sec. 3.4 — and perform a decorrelation stretch followed by a histogram equalisation on the three channels selected in the `rect` node. It will only do this to a rectangular portion of the image (defined by the `rect` node), annotating the region with some text defined in that node's controls. The control region is currently showing the output of the histogram equalisation.



graph.png

Figure 2: An example graph

1.1 Notes on type checking

Python is dynamically typed, but there are a lot of "type annotations" in the code. Unfortunately, Python's import rules mean there's also some odd stuff going on. Annotations like

```
class XFormType:
    # name of the type
    name: str
    # the palette group to which it belongs
    group: str
    # version number
    ver: str
    # does it have an enable button?
    hasEnable: bool
```

are straightforward: we have three string fields (`name`, `group` and `ver`) and a boolean field (`hasEnable`). The next annotation, however, is a link to a class defined further down the file, which in turn has a field which is `XFormType`: a cyclic dependency. In such cases, the standard PEP 0484 tactic is to use a string literal — the type checker will resolve this successfully and give appropriate warnings:

```
    # all instances of this type in all graphs
    instances: List['XForm']
```

defines `instances` as a list of `XForm`.

Another oddity you may see in some of the files is this (from the top of `xform.py` like the previous example):

```
if TYPE_CHECKING:
    import PyQt5.QtWidgets
    from macros import XFormMacro, MacroInstance
```

These lines are only run when type checking, and are used to ensure that appropriate classes are imported for type hints like this:

```
    # an open help window, or None
    helpwin: Optional['PyQt5.QtWidgets.QMainWindow']
```

Without the `TYPE_CHECKING` guard the program will not run, because these imports are potentially cyclic. However, they are only needed at compile time, so the `if`-statement is added to stop the import at run time. Note the quotes: they are there to stop Python trying to resolve the symbols at run time.

1.2 Structure

The code is structured thus:

```
PCOT                    { main directory }
\-src                   { source code }
  \-pcot                { top-level package }
    \-assets            { data, typically .ui files }
     -calib             { code for handling colour calibration with PCT }
     -expressions       { expression parser package }
     -inputs            { data input method package }
     -operations        { "operations" package for node/expressions }
     -ui                { core user interface code and widget code }
     -utils             { utilities, e.g. archive managed, colour functions }
     -xforms            { the auto-registered XFormType node types }
```

Notes:

- **operations** is a package for things which are both nodes and expression functions, such as *curve* and *norm*: the operation is encapsulated in a single function which is used by both a node and a lambda for registering with the expression parser.
- As well as containing the `src` directory, the top level also contains files for installation and building single-file executables with Poetry and PyInstaller.

2 Installation

The following instructions are intended for developers or users who wish to use PCOT as a library, or to develop their own plugins (typically XForm node types or *expr* node functions). Other users receive a single-file executable constructed by PyInstaller using the instructions below. This does not require installation — just running the file will run PCOT. Different executables are required for each platform.

Firstly, you will need to install Anaconda (https://docs.anaconda.com/anaconda/install/index.html). You may already have it installed. Then you'll need to open an command line into which you can enter Anaconda commands — this is platform dependent; on Linux it's just a normal shell (and I think this is also true on MacOS), while on Windows it will be a program in the Anaconda package. The following commands will create a new Anaconda environment called `pcot` and install the required packages into it:

`conda create -n pcot python=3.8 poetry`	Create a minimal conda environment containing just a known version of Python and the Poetry dependency management system.
`conda activate pcot`	Activate the environment.
`poetry install`	Use Poetry to install all required packages as listed in the `pyproject.toml` and `poetry.lock` files.

Now you should be able to run PCOT using the `pcot` command.

2.1 Running PCOT inside PyCharm for development

If you wish to run PCOT from inside IntelliJ's PyCharm IDE, follow these instructions after building an Anaconda environment as shown above. First, tell PyCharm about the environment:

- Open PyCharm and open the PCOT directory as an existing project.
- Open **Settings..** (Ctrl+Alt+S)
- Select **Project:PCOT / Python Interpreter**
- Select the cogwheel to the right of the Python Interpreter dropdown and then select **Add**.
- Select **Conda Environment**.
- Select **Existing Environment**.
- Select the environment: it should be something like `anaconda3/envs/pcot/bin/python`.
- Select **OK**.

Now set up the run configuration:

- Select **Edit Configurations** from the configurations drop down in the menu bar
- Add a new configuration (the + symbol)

- Set **Script Path** to `PCOT/src/pcot/main.py`
- Make sure the interpreter is something like `Project Default (Python 3.8 (pcot))`, i.e. the Python interpreter of the pcot environment.

You should now be able to run and debug PCOT from inside PyCharm.

2.2 Building a single file executable

These brief instructions are for core PCOT developers only. They deal with creating single file executables for redistribution to users.

The following commands from inside an Anaconda shell should create a single file executable, assuming you have successfully created a PCOT Anaconda install as given in Sec. 2 above.

`conda activate pcot`	Ensure the PCOT environment is active.
`pip3 install pyinstaller`	Install PyInstaller into the environment (we don't do this by default because not every user requires it).
`cd pyInstaller`	Go into the PyInstaller directory inside the main PCOT directory — this contains the necessary specification files.
`pyinstaller file.spec`	Run PyInstaller: this will build a single, very large file called `dist/pcot`. It will take a very long time.

This process is prone to fail: small changes in PCOT's structure can make it difficult for PyInstaller to work. One particular problem is "hidden imports" – packages imported through non-standard means. For example, XForms cannot be automatically detected when inside a single executable, so under a PyInstaller build they must be explicitly listed in a file called `xformlist.txt` inside the main PCOT package. Luckily, the PyInstaller specification file is just a Python program, and can automatically create this file. Read `file.spec` for the gory details. Also, any non-standard widgets used in UI files need to be added to hidden imports, and any data files need to be added to a list of such files.

3 The data model

The "root" of the data model is the `Document` object. This contains:

- the "main graph" — that is, the graph of nodes not contained inside macros;
- a dictionary of macro type objects (`XFormMacro`) by name, which are local to this document — each macro also contains a "prototype graph" (see Sec. 3.6);
- the input manager and the state of the various input methods contained in each input (see Sec. 3.4);
- document-wide settings.

The most important parts of the model are the graph and the data which it manipulates — this is largely image cube data passed between the graph nodes. Other forms of data do exist, but these are much simpler.

3.1 The Datum type

All data passed between nodes is of the `Datum` type. This is also the type used by all values in the *expr* expression evaluator node. A `Datum` consists of:

- **tp**: a type, e.g. `Datum.IMG` for an image or `Datum.NUMBER` for a number. This is an enum defined in `Datum` itself, some values of which are never held by actual datum objects (e.g. `Datum.ANY`, which is used to indicate that a connection can be of any type).
- **val**: a value of the appropriate type to match **tp**.
- **sources**: an object matching the `SourcesObtainable` interface. This can be queried to get a set of source objects describing which inputs (and filter bands within multichannel inputs) a datum is derived from. Nodes and functions which manipulate data are required to handle sources correctly, and this can be difficult.

3.2 Graph and nodes

Graphs are directed acyclic graphs of nodes represented by the `XFormGraph` class. Each node is an instance of `XForm` (short for "transform node"). The function of each node is determined by its `type` field, which references an `XFormType` singleton. See Fig. 3 for an overview, which also shows an outline of how the main graph and macro prototype graphs fit into the document.

3.2.1 XFormType and type registration

Each node type is represented by a subclass of `XFormType`, and each subclass has a singleton to which nodes of that type link. For example, the *rect* node's behaviour is specified by the `XformRect` class, which has a singleton instance. All `XForm` nodes which are *rect* nodes have a `type` field pointing to this singleton.

Most singletons are automatically created and registered when the class is defined, through the `@xformtype` decorator and `createXFormTypeInstances()`. The decorator stores the required information —- name, class, constructor arguments etc. — into a list, while `createXFormTypeInstances()`, which runs just before the app starts (as late as possible) does the following for each entry:

- Creates an instance of the class;
- Creates an MD5 hash of the class' source code which is stored inside the type singleton for version control;
- Changes the semantics of the class constructor so that it always returns the instance we just created (thus making the class a singleton).

The reason for the deferred instantiation of the type objects is to ensure that all user hooks are run before the instantiation (so that *expr* function hooks [Sec. 5] work).

The base constructor for `XFormType` adds the singleton to a dictionary class variable `allTypes`, so we can always obtain the singleton object and create new nodes which perform that node type. In addition, each type singleton has a "group" field which determines where in the palette to create a construction button for the type.

Some other `XFormType` objects are dealt with differently:

- Macros are represented by `XFormMacro` objects — a subclass of `XFormType`. However, these are stored inside the document, not in the global type dictionary — this ensures that macros are document-local[1]
- Some nodes are registered normally in `allTypes` but are given a "non-existent" group name to stop them appearing in the palette. These include the `XFormDummy` type, assigned to nodes whose type cannot be found; and macro connectors which can only be created inside macros.

3.2.2 XFormType methods

In order to perform a node's action, the type classes must contain the following methods:

- `init(node:XForm)` : initialise any extra data inside the node required to perform this type's behaviour
- `perform(node:XForm)` : perform this node's behaviour — read any inputs, manipulate the data, set the outputs.
- `createTab(node:XForm, window:MainWindow)` : create a UI tab to edit/view this node.

Several other methods may optionally be overridden.

3.2.3 Linkage

`XForm` node objects are linked together by their inputs. Each `XForm` contains an `inputs` list indexed by input number. The length of this list is determined by the number of inputs the type object specifies. Each entry is a $(node, output)$ tuple where *node* is a reference to another `XForm` and *output* is the index of an output on that `XForm`.

Methods are provided in `XForm` for connecting and disconnecting nodes (also checking for cycles and providing basic type checking), and getting inputs and setting outputs inside the type's `perform()` method.

The number and types of inputs and outputs is set inside the constructor for the `XFormType` singleton, by using methods to add these connectors. They are then indexed in order of creation. For example, the *inset* `XFormType` has the constructor:

[1] The ramifications of macros being global to the application are problematic: consider opening a file with a macro "foo" defined one way, and then opening another file with a different definition of the same macro!

```
    def __init__(self):
        super().__init__("inset", "regions", "0.0.0")
        self.addInputConnector("img", Datum.IMG)     # input 0
        self.addInputConnector("inset", Datum.IMG)   # input 1
        self.addInputConnector("rect", Datum.RECT)   # input 2
        self.addOutputConnector("", "img")
```

The arguments for these methods are (*name*, *type*, *description*) where the name is often empty (it just shows a hint) and the description is optional (it appears in help text). The type is a `datum.Type` value — these are stored by name as static members of `Datum`.

Within an executing graph, the data between nodes is stored as an `xform.Datum` object containing the type and value. Inside the node type's `perform()` method, we typically do something like:

```
    node.setOutput(0, Datum(Datum.IMG, img))
```

which sets output 0 to hold the image (i.e. `ImageCube`) stored in `img`. Fetching data from an input is also done in `perform()`, and takes two forms. The first returns a `Datum`:

```
    datum = node.getInput(0)
```

while the second examines the datum internally, and returns the value field if the type matches or None if it doesn't:

```
    image = node.getInput(0, Datum.IMG)
```

Note that None is a permissible value in `setOutput()`, which `getInput()` will always return as None.

3.2.4 Overriding types and the variant type

It is possible for an individual `XForm` node to override the input and output types set in its `XFormType`, by setting values in the `inputTypes` and `outputTypes` array. This is used in macros and mathematical operations. Often the "default" type is `Datum.VARIANT`, which is a special type which must be overriden by some user action before the node becomes useful. A variant connection shows as crosshatched, and cannot be connected until its type is changed.

3.3 Performing the graph

The graph needs to be "performed" — its nodes executed — whenever the data changes, which is generally whenever the graph is edited, a node control changed in a tab, or an input is changed (see Sec. 3.4 below). This is done by calling `changed()` on the node's graph (or on the tab itself). The `XFormGraph.changed()` method takes a node, and either calls `performNodes()` on the main graph passing in that node, or, if the node is inside a macro prototype graph (q.v.), copies the prototype graph to all its instances and runs `performNodes()` for all those instances (see Sec. 3.6.2 for why this is a problem).

The `XFormGraph.performNodes(node=None)` method itself takes a single argument and performs either the entire graph or a portion of the graph starting at a particular node. If the argument is None, the internal list of all nodes in the graph is traversed looking for nodes with no inputs and these are performed. If the argument is not None that node is performed. Nodes are performed by calling their `perform()` method. This will recursively run the child nodes, but only when they are "ready to run" (all their inputs have been set).

3.3.1 Performing a node

The `XForm.perform()` method will run a node, and recursively run all child nodes, although there are some complexities here (mainly to accommodate macros). It will not perform the node if it has already run in this call to `performNodes()` or if it is not ready to run (some inputs do not yet have values):

if node has not already run **and** node is ready to run **then**
  clear all outputs
  run the node type singleton's `perform()` method on the node
  mark node as having run
  for all t in open tabs for this node **do**
    t`.nodeChanged()`
  end for
  for all c in child nodes **do**
    perform node c
  end for
end if

A node is ready to run if it has no inputs which do not yet have values set. This is determined by checking the outputs of the nodes from which the inputs come to see if they have yet been set with values. The `performNodes()` method is called from these places:

- `XFormGraph.changed(node)`: when the graph or a node in a graph has been changed. This is the most frequent call, and is typically called with a node to avoid running the whole graph. The node is passed down into `performNodes()`).
- `XFormMacro.perform(node)`: when a node which contains a macro instance is being performed (see Sec. 3.6 for more details).
- `PaletteButton.click()`: when a new node is created from the palette, to initialise it.

A few other calls may occur, for example the *constant* node calls this method when its value is changed. The `XFormGraph.changed()` method is called from several places:

- `XFormGraph.runAll()` : called when a graph is explicitly run.
- `XForm.disconnect()` and `XForm.connect()`: when connections between nodes are made or broken.
- `XForm.setEnabled()`: when a node is enabled or disabled.
- `Tab.changed()`: when a tab signals that a node it controls has had a parameter change.
- `XFormGraph.deserialise()`: when a graph has been loaded.

3.3.2 Error handling

Error handling is generally required in three cases:

- connection type mismatch discovered at connect time,
- connection type mismatch discovered at perform time,

- other errors at perform time.

The first is dealt with using the data type system in `datum.py`: each connection (input and output) has a string type and the connections must match, or they will not be made. An input of type "any" can accept any type. Additional checks are made to avoid cycles.

The other types of error are both handled inside the node's `perform()`, and take two forms:

- either the type's `perform()` throws an `XFormException`, which is handled by `XForm.perform`,
- or `perform()` completes but perhaps shows an empty image, and calls `setError(exception)` to set the error state.

In both cases `setError()` will print a message to console and set an error state in the node. This error state will have been cleared in all nodes before the graph is performed. A redraw of the entire graph is done after the graph is performed to draw those nodes in an error state differently, showing the brief error code passed into `XFormException`. The error will also be shown in the tab for that node and in its "help box" (opened by clicking the box in the corner).

3.4 Inputs

Inputs are components which take external data from the DAR or from files and bring them into PCOT. They are considered part of the document, but not part of the graph of nodes which appears in the right-hand pane. This may sound like an odd choice, but there are good reasons for it:

- We can change the graph without changing the input files: this means we can set up an input from the DAR (which may be complex) and then load an entirely different processing graph to view it a different way. This also permits the use of "template" graphs, which can be loaded to perform a piece of standardised processing but do not change the inputs.
- Conversely, we can change the inputs without changing the graph. This means we don't have to play "hunt the input node" if we want to process another piece of data.

Clicking on an input at the top of the window will open the the editor for that input. This consists for some buttons, one for each input method, and a panel for editing the current method's data. By default the method is "null", which means "no input." Clicking on another button will change the active input method and allow it to be edited. To bring the data into the graph, an *input* node can be created from the palette. How inputs work is summarised in Fig. 4:

- Documents own an `InputManager`.
- The `InputManager` owns one `Input` object for each input.
- Each `Input` has a number of input method objects, each of which is of a different subclass of `InputMethod`. These handle different kinds of input: RGB file, multifile, null (does nothing) and eventually different kinds of DAR input. Although all these methods are always present (for persistence reasons), only one is active. An index to the active method is stored in `Input`.
- the `Input` may also have an `InputWindow`, if its UI window is open. This will contain a set of widgets, each of which is subclass of `MethodWidget`, one for each input method. Only the widget corresponding to the active method will be visible.

Code for the inputs is the `pcot.inputs` package, except for the UI code which is in the `pcot.ui.inputs` module.

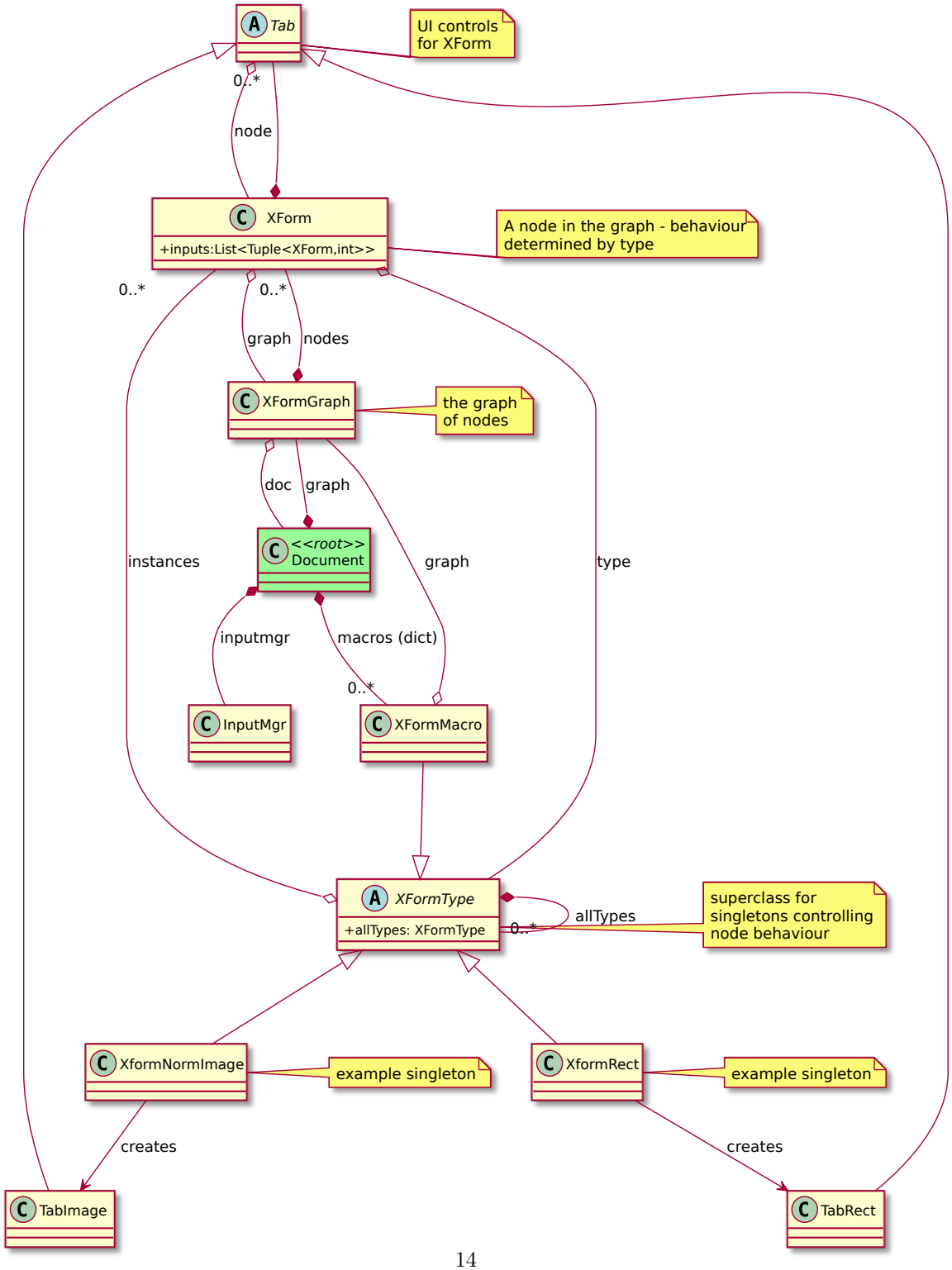
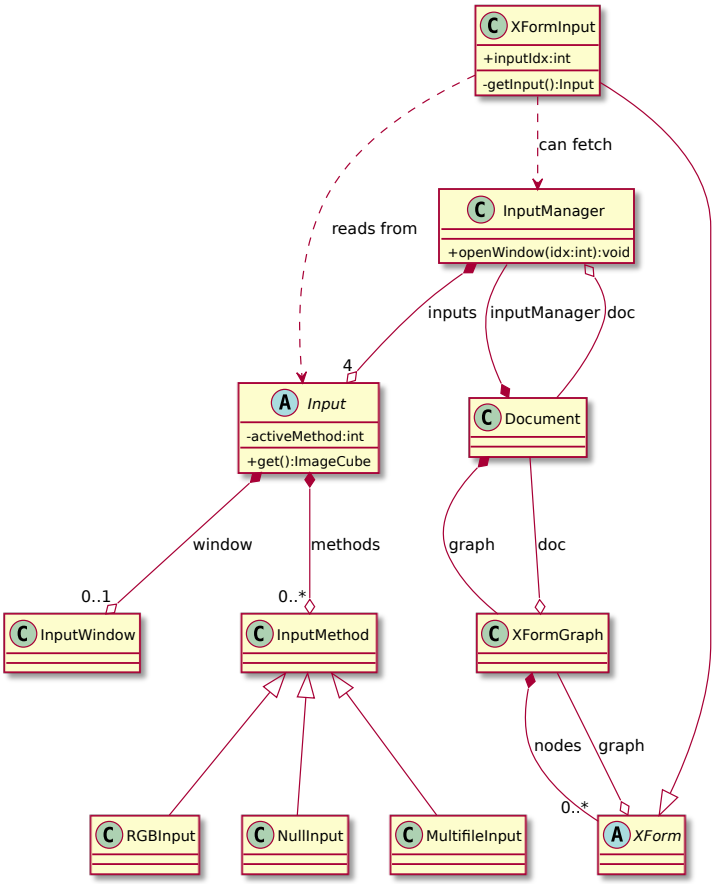


Figure 3: `XForm` and graph model



inputs.pdf

Figure 4: Input system

3.5 Image data and data sources

Most classes making up the image data model are declared in the `imagecube.py` file, including the main `ImageCube` class. Some additional classes describing where images can come from are in `channelsource.py`. The model is shown in outline in Fig. 5 although some links to channel sources and mapping from nodes are omitted; these will be explained later.

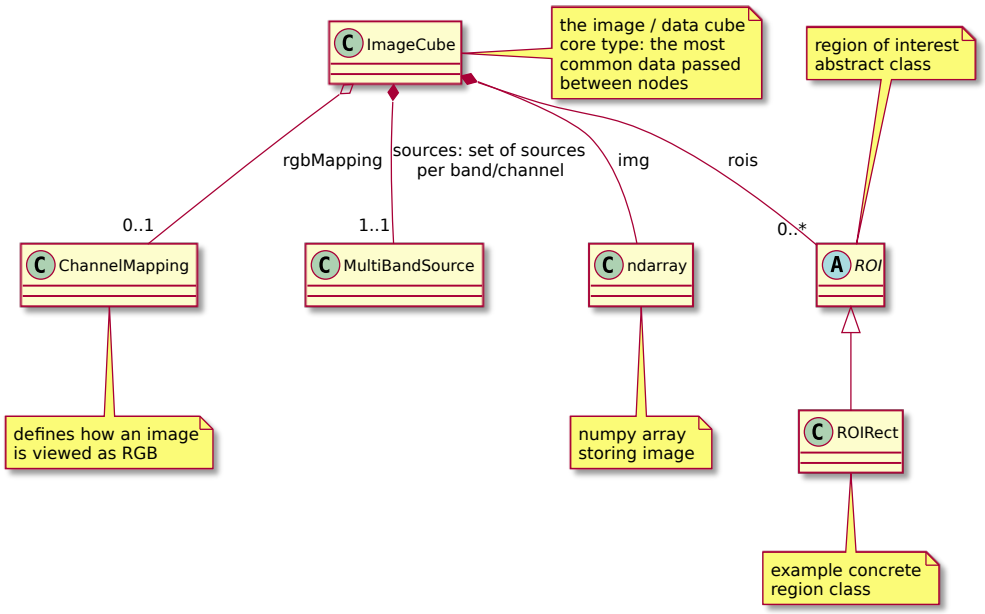


image.pdf

Figure 5: Outline UML class diagram of image model

The main class is `ImageCube`: this encapsulates a numpy array `img` which is the actual image data cube. This is either a $w \times h \times depth$ array for genuine cubes with multiple channels, or a $w \times h$ array for a single channel image. The data type is 32-bit floating point, and images are typically normalized to the range [0,1].

> In this document I have a tendency to refer to image data as both "image" and "image cube." Both terms refer to the same thing: an array of floating point image data, with 1 or more channels (slices) of information. There is no upper limit on the number of channels in an image (or image cube) beyond system memory.

3.5.1 Data sources

Each piece of data in PCOT passing through a node or used as part of an *expr* node's expression has metadata describing where it came from. This is ultimately always a PCOT input (i.e. from an *input* node); data from user input or elsewhere is disregarded. This is handled by the following entities:

- **SourcesObtainable** is the root interface of the system, consisting of all classes which have the `getSources()` method which can return a set of sources.
- **SourceSet** is a set of `Source` objects, indicating which sources are contributing to a given piece of data. For example, some operation may have combined several bands into one using a mathematical operation — the output band's SourceSet will consist of the all the contributing Sources. It is not a `Source` itself, but it is a `SourcesObtainable` and so a SourceSet which is the union of all its constituent SourceSets can be obtained.
- **Source** is the base class of all concrete source types, currently consisting of just `InputSource` and the dummy `_NullSource` "placeholder" class for sources which can be filtered out.
- **InputSource** represents a PCOT input source of data. It holds references to the Input, Document and may also hold information about which PanCam Filter (or other device) the information comes from.
- **MultiBandSource** is a collection of SourceSets, one for each band in a multiband image. It is not a `Source` itself, but it is a `SourcesObtainable` and so a SourceSet which is the union of all its constituent SourceSets can be obtained.

These classes are shown as a UML diagram in Fig. 6.

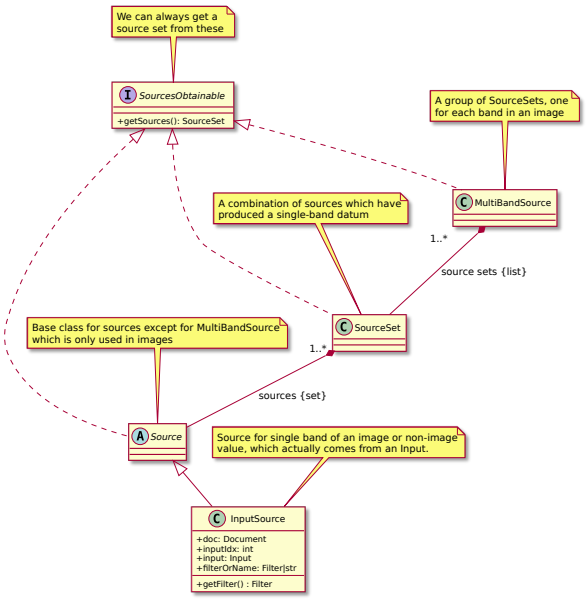
Nodes generate and process source information in different ways. For example, a *gradient* node takes a single band and converts it into an RGB image with a colour gradient: here, the output image has a MultiBandSource of three identical SourceSets, which are the SourceSet of the single band image input. In contrast, a the *curve* for performing a sigmoid function on all bands of an image will give the output image the same sources as the input image.

3.5.2 Important methods for sources

As well as `getSources()`, which return a SourceSet which is the union of all constituent sources, all `Source` subclasses have the following methods:

- **copy()** produces a "reasonably deep" copy of the source
- **matches()** returns true if a source matches some criteria — details are below
- **getFilter()** returns the Filter object if one is present (or None)
- **brief()** returns a brief description
- **long()** returns a long description

The `match()` method can match on the following criteria — all of which must be true, but criteria which are None are ignored:



source.pdf

Figure 6: Source management classes

- **hasFilter**: the Source has a filter
- **filterNameOrCWL**: the Source has a filter with the same name (if a string) or centre wavelength (if a float)
- **inp**: the Source has the same input index (i.e. number of the PCOT input)

SourceSet has the following additional functionality:

- The constructor can take a Source, a SourceSet, or a collection of either/both, and convert them into a single union SourceSet.
- **match()** returns true if any source in the set matches all criteria. If the `single` flag is true, sets are ignored if they contain more than one element.

MultiBandSource also has the following:

- The constructor takes a list of SourceSets and/or Sources
- **createEmptySourceSets(n)** class method constructs a MultiBandSource of n empty source sets
- **createBandwiseUnion(lst)** takes a list of MultiBandSources with the same number of bands, and constructs a new MultiBandSource where each band's source set is a union of the

corresponding bands' source sets in each input. For example, consider a MultiBandSource M_1 consisting of three SourceSets A, B, C, and two other MultiBandSources M_2, M_3 defined similarly:

$$M_1 = \{A, B, C\}$$
$$M_2 = \{D, E, F\}$$
$$M_3 = \{G, H, I\}$$
$$\text{createBandwiseUnion}([M_1, M_2, M_3]) = \{A \cup D \cup G, B \cup E \cup H, C \cup F \cup I\}$$

This is typically in *expr* binary operations such as image addition.

- **add(s)** adds a source set to the list
- **copy()** returns a "reasonably deep" copy
- **search()** returns a list of indices of bands whose source sets contain a source which match all of the criteria given (again, None criteria are ignored). Criteria are:
  - **hasFilter**: the Source has a filter
  - **filterNameOrCWL**: the Source has a filter with the same name (if a string) or centre wavelength (if a float)
  - **inp**: the Source has the same input index (i.e. number of the PCOT input)
  - **single**: the Source is the only member of the set
- **brief()** returns a brief description
- **long()** returns a long description

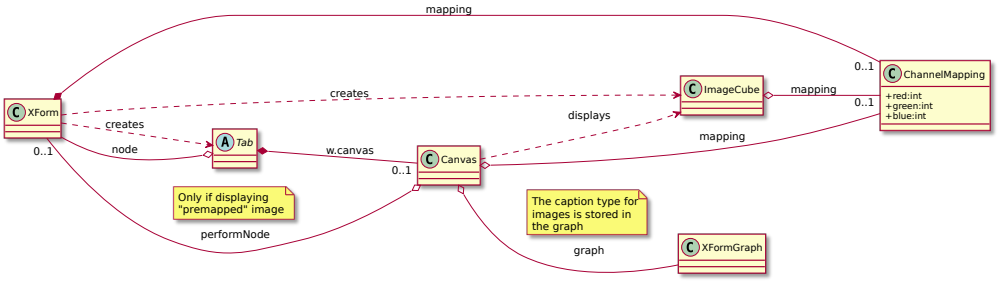
Sources are used to keep track of each datum as it moves through the graph so they can be processed and displayed appropriately, particularly for images: Fig. 1 (p. 3) shows a typical node in the "node controls and output" section. This section, as it does in many nodes, contains a canvas displaying an image. Above the canvas are three combo boxes which select the channels in the image cube to display on the canvas, and these are typically labelled by a string generated from the source data for each channel (along with the index). Sources are also used to select channels to combine and manipulate in those nodes which do so.

3.5.3 RGB channel mappings

The previous section briefly mentions the three RGB mapping combo boxes at the top of the canvas component in the node controls in Fig 1. In canvases — components which display images — a multi-channel image cube must be displayed as RGB data. The combo boxes control how this is done, and the data is made persistent in a slightly complicated way, as shown in Fig. 7.

The relationships will hopefully become clearer when I come to describe how nodes work in more detail, but for now:

- A `ChannelMapping` object consists of three integers giving the indices of the channels in an image cube to display in the red, green and blue channels on screen.



rgb.pdf

Figure 7: RGB mapping classes

- An `XForm` (i.e. a node) may need to show an image on a canvas. If so, it will use the `mapping` field in `XForm` to store a mapping. This is provided as a convenience, not all node classes will use it. Some may need to create more if they display more than one image.
- When an `XForm` creates an image for display or passes one through, it sets the mapping of the image to the mapping of the node. This is used in the `rgb()` method to generate the RGB representation.
- When an `XForm` is opened for modification, it creates a subclass of `Tab`. This will contain a `Canvas`, which is given a reference to the mapping inside the `XForm`. It needs this so that the mapping can be modified even when no image is present.

This may seem rather redundant, but

- The mapping must be be owned by the node, so it can be serialised and persists when there is no image and no open tab.
- We must have a reference to the mapping in the canvas, so it can be manipulated by the combo boxes even when no image is present.
- Finally, we need a reference to the mapping in the image so that `rgb()` can be called on the image when it is input into another node. This is used in nodes like *inset*, which operate entirely on RGB representations — it's much neater if these are the RGB representations output by the nodes which feed in.

A few nodes have an extra complication — they may output an RGB representation. In this case, the RGB conversion needs to be done inside the node itself for output, rather than inside the canvas solely for display. A good node to study for details on how to do this is *rect*. In essence, the Canvas is told to display the "premapped" RGB image generated inside the node's `perform()` method. It is also given a node reference so that the node can be performed again (thus regenerating the RGB image) whenever a mapping combo box is changed.

3.5.4 Regions of interest

Regions of interest belong to images, and modify how nodes process those images. They are added to images by region of interest nodes such as *rect*. Figs. 1 and 2 show this in action:

- A file is read in, producing an image cube
- A *rect* node adds a region of interest to this image. The outputs are:
  - the image with the rectangle added to its list of ROIs;
  - the image cropped to the bounding box of the list of ROIs (at the moment, just this rectangle)'
  - an RGB representation of the image (according to the previous node's canvas RGB mapping); annotated with the rectangle and some text, and also an ROI added describing the rectangle;
  - the rectangle datum itself.
- a *decorr stretch* takes the annotated RGB representation output and imposes a decorrelation stretch — but only on the regions of interest in the image (in this case, inside the rectangle)
- a *histequal* node performs a histogram equalisation, again honouring the regions of interest which have been passed through the previous node unchanged.

As shown in Fig. 5, each image contains a list of `ROI` objects, each of which is an instantiation of a subclass of `ROI`.

To honouring regions of interest inside a node's `perform()` method:

- `ImageCube.subimage()` will return a `SubImageCubeROI` object. This encapsulates a numpy array containing the image bounded to a rectangle around the regions of interest and a boolean mask (again as a numpy array) specifying which pixels in this rectangle are actually in regions of interest.
- The manipulation can now be performed on the `img` field of this "subimage," but only on those pixels whose values are true in the corresponding `mask` field.
- The modified pixels can be "spliced" into the original image cube, creating a new image cube, using the `modifyWithSub` method.

This example shows the operation of the decorrelation stretch:

```
    def perform(self,node):
        img = node.getInput(0, Datum.IMG)
        if img is None:
            node.img = None
        elif not node.enabled:
            node.img = img
        elif img.channels != 3:
            ui.error("Can't decorr stretch images with other than 3 channels")
        else:
            subimage = img.subimage()
            newimg = decorrstretch(subimage.img, subimage.mask)
            node.img = img.modifyWithSub(subimage, newimg)
        if node.img is not None:
            node.img.setMapping(node.mapping)
        node.setOutput(0, Datum(Datum.IMG, node.img))
```

There are several checks for whether the node is actually enabled, and whether there is an image present to stretch, but the core lines are these:

```
            subimage = img.subimage()
            newimg = decorrstretch(subimage.img, subimage.mask)
            node.img = img.modifyWithSub(subimage, newimg)
```

The `decorrstretch` takes two arguments: the numpy array containing the pixels which bound the ROIs, and the mask for those pixels in that array which are in the ROIs. It returns an image of the same size, which is then spliced back into the original image. The new image returned will have the same channel sources, the same ROIs and the same RGB mapping.

> Much of the ROI system is work in progress, particularly combining multiple ROIs. This documentation might change.

3.6 Macros

Macros are one of the more complex parts of the PCOT data model, so it's important that they are documented here. First, a quick description of how they work from a user standpoint.

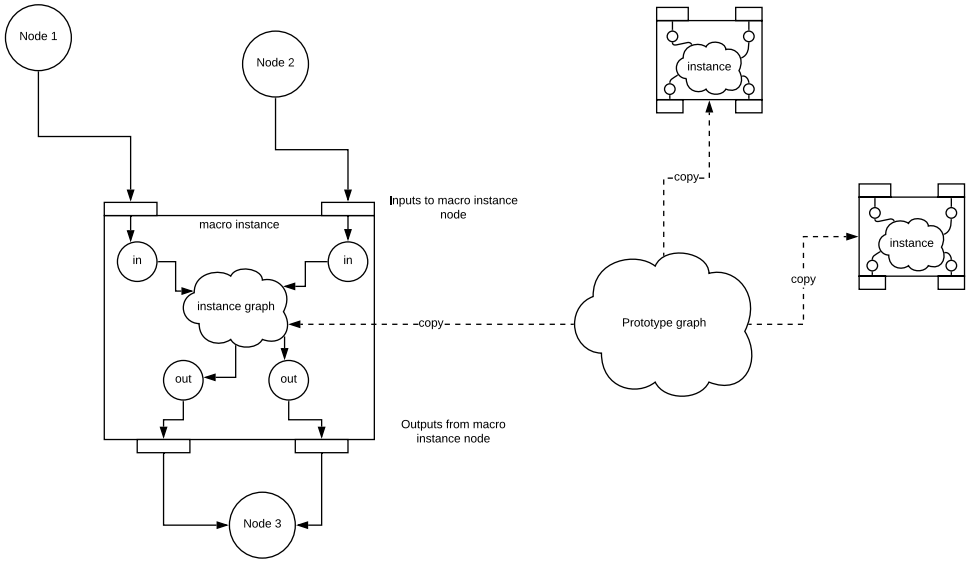
Users can create a new macro, which will open a window with a new graph in it. This is distinguished from the main window by its slightly different background colour. The user can create and manipulate transform nodes inside this graph, as usual. This is the **prototype graph** for the macro. The user should create special input and output nodes inside this graph to allow data to flow into and out of the macro.

When a new macro is created, a button for that macro appears in the "palette" on the right of the app window. Clicking on this button will create an **instance** of the macro inside the main graph. This is a node which contains a copy of the prototype graph — it must be a copy, because multiple instances of the macro with different data may exist. When the user edits the prototype graph (including the parameters of any nodes within it) the changes are copied to the instance graphs for that macro. Thus the user now has a single graph which can be changed, which represents a set of nodes inside a single node. Multiple instances of the macro can be made which will all share the same parameters.

3.6.1 Macro internals

Each macro node contains an instance graph copied from the prototype graph, which has a set of components interfacing between the instance graph and the graph in which the instance is embedded. Consider the situation in Fig. 8. This contains three instance nodes of a single macro. I have "zoomed in" on one of the instances, showing that it is connected to three other nodes: two on its inputs, one on its outputs. When this main graph runs, the following happens:

- Node 1 is able to run, does so, and sets its output
- The macro instance cannot yet run
- Node 2 is able to run, does so, and sets its output
- The macro instance can now run:
  - The macro instance node copies its input data into the input nodes contained within the instance graph
  - The input nodes in the instance graph are run
  - The nodes dependent on those input nodes are run (i.e. the instance graph proper)



macroobjs.pdf

Figure 8: An example of a graph containing macros

  - The output nodes in the instance graph are run, copying their inputs into the macro instance node's outputs
  - the instance now has now completed its run

- Node 3 can now run, reading its inputs from the macro instance node.

3.6.2 Macro inefficiencies

As noted above in Sec. 3.3, all the nodes in all instance graphs are run whenever the prototype graph is changed. This is very inefficient and may cause considerable delays. It's done by simply forcing all `XFormMacro` nodes which are instances of the macro to perform themselves. Here is the relevant part of `XFormGraph.changed()`:

```
    # distribute changes in macro prototype to instances.
    # what we do here is go through all instances of the macro.
    # We copy the changed prototype to the instances, then run
    # the instances in the graphs which contain them (usually the
    # main graph).
    # This could be optimised to run only the relevant (changed) component
    # within the macro, but that's very hairy.

    for inst in self.proto.instances:
        inst.instance.copyProto()
        inst.graph.performNodes(inst)
```

Here, `inst` is each `XFormMacro` node inside the main graph. Thus `inst.graph` will be the main graph (for an non-nested macro). The `self` value is the macro prototype graph, because this method was called on an object inside that graph. This code therefore calls `performNodes()` on the main graph to run the `XFormMacro` node inside that graph.

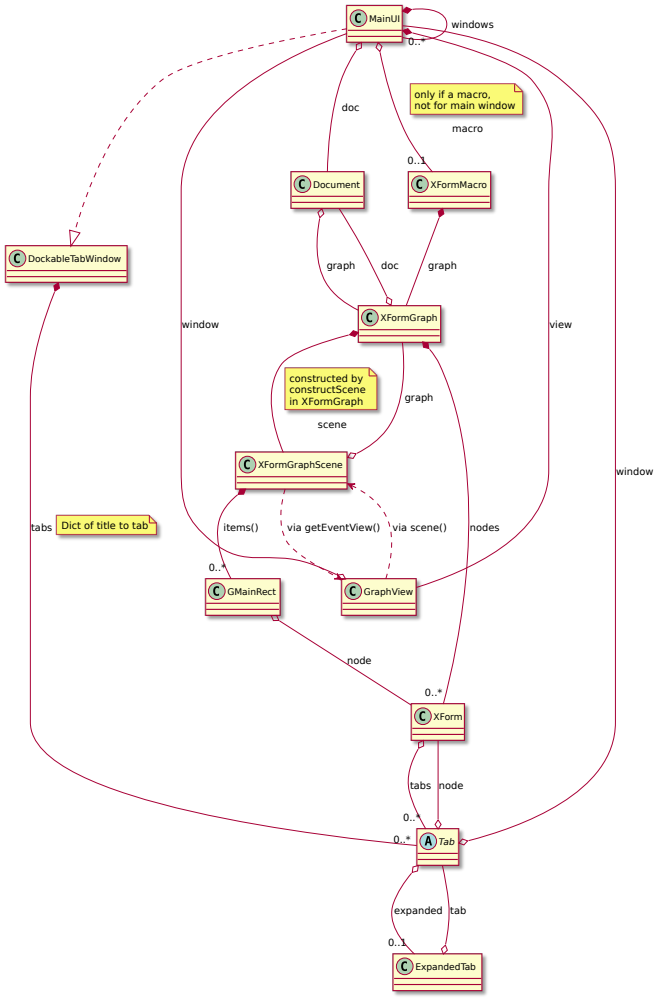
In an ideal world, this process — calling `changed()` — would identify the instance node which corresponds to the prototype node which was changed, and only run that inside the instance graph for the macro. The child nodes of the instance node would then need to be run.

3.7 User interface

The user interface is mainly in the `ui` package, although each node type's file contains its UI code (a subclass of `ui.tabs.Tab`. A reasonably full view of the system is shown as a class diagram in Fig. 9. The main files involved are:

- `ui/mainwindow.py`: the main window classes;
- `ui/tabs.py`: the `DocktableTabWindow` and `Tab` classes (also the `ExpandedTab` class for when a tab is undocked);
- `ui/canvas.py`: the Canvas widget for viewing image cube slices;
- `ui/graphview.py`: contains `GraphView`, the `QGraphicsView` subclass which encapulates a view on the graph scene;
- `ui/graphscene.py`: contains `XFormGraphScene`, the `QGraphicsScene` subclass which contains a set of 2D objects representing an `XFormGraph` and its nodes. It also contains the classes representing those 2D objects (subclasses of various Qt graphics item classes).

The input system's UI is not discussed here — it's a separate system in `ui/inputs.py`. See Sec. 3.4 for more details.



ui.pdf

Figure 9: UI class diagram

3.7.1 The main window

The main window class is `ui.mainwindow.MainUI`. They contain the following main widgets:

- a `QTabWidget` to hold the dockable tabs (tab docking is handled by the `DocktableTab` superclass);
- a `GraphView` widget to manage viewing and manipulating the graph;
- a `Palette` to contain the buttons to create new nodes

The bottom pane contains various widgets, such as the log console and caption control combo box. Main windows are created:

- when the application opens the first empty main graph in `main.py`
- when a new, empty main graph window is created
- when a window for a macro prototype graph is created via `createMacroWindow()`.

There are some ownership oddities here. In the first two cases the `MainUI` constructor is called with no arguments. This will cause it to create a new `XFormGraph` which the window will own. In the last case, the constructor is informed that the graph is a macro prototype. This causes the UI to be created slightly differently. The `createMacroWindow()` static method then sets the graph to the macro's prototype, which is owned by the `XFormMacro`.

In both cases, the `XFormGraphScene` contains a Qt Graphics Scene which is constructed and regularly updated from the graph (by calling its `rebuild()` method). This is viewed in the main window through the `GraphView` class. Both these classes accept various user actions and use them to modify the graph.

4 Customising PCOT

After running PCOT for the first time, each user will have a `.pcot.ini` file in the their home directory for settings and some persistent data. This section of the document describes this data.

4.1 Locations section

- **pcotfiles**: location of PCOT documents; the directory of the last PCOT file saved or loaded.
- **images**: the directory of the last image loaded.
- **mplplots**: the directory of the last MPL plot saved.
- **pluginpath**: a colon-separated list of directories scanned for plugins (see below).

4.2 Plugins

Each directory inside the plugin path (see above) is scanned at application startup, on import of the `pcot` module. All Python files found are imported. Various mechanisms are provided to let users create their own nodes, *expr* functions and menu items.

4.2.1 Creating new types of node

Simply adding a new subclass of `XFormType` preceded by the `@xformtype` decorator is sufficient to register a new node type. See Sec. 6 for more details, or read some of PCOT's source code — node types can be found in the `xforms` directory.

4.2.2 Creating new *expr* functions

This is done by adding an expression function hook to the list of hooks. These hooks are functions which take an `ExpressionEvaluator` class. Add something like the following to one of your Python files in your plugin directory:

```
import pcot

def testfunc(args, optargs):
    # just a numeric test function that calculates a+b*2 for two parameters a,b
    # get the two values as numbers
    a = args[0].get(Datum.NUMBER)
    b = args[1].get(Datum.NUMBER)
    # calculate the result (yes, I'm doing this the long-winded way)
    result = a+b*2
    # convert the result into a Datum and return
    return Datum(Datum.NUMBER, result)

def regfuncs(e):
    from pcot.expression.parse import Parameter
    # register a function "test" with a description
    p.registerFunc("test", "func description",
    # one entry for each mandatory argument/parameter
                   [Parameter("a", "description for A", Datum.NUMBER),
                    Parameter("b","description for B", Datum.NUMBER)
                   ],
                   [], # no optional arguments
                   testfunc # finally, the fnction
                   )

# register the function by adding it to the expression function hooks - this
# will run regfuncs() when the expression evaluator is created.

pcot.exprFuncHooks.append(regfuncs)
```

See 5 for more details.

5 Writing custom *expr* functions

This section covers writing custom functions for use in the expression evaluation node *expr*. All these functions have two arguments and return a single value. The arguments are:

- **args** is an array of mandatory positional arguments
- **optargs** is an array of optional positional arguments which may follow the mandatory arguments.

All arguments and the return value are `Datum` objects, and we obtain the contained data by calling the `get()` method with the appropriate `Type` object (type objects are static members of `Datum`). To register a function we use the `registerFunc()` method of `ExpressionEvaluator`, which has the following arguments:

- function name

- function description
- array of `Parameter` objects describing the mandatory arguments
- array of `Parameter` objects describing the optional arguments (which must be numeric)
- the function itself

For built-in functions this is done in `ExpressionEvaluator`'s constructor. For user functions, you must add an expression function hook from a plugin file and register the functions in the hook (which is passed an `ExpressionEvaluator`) — see Sec. 4 for how to do this.

The `Parameter` constructor takes the following arguments:

- parameter name
- parameter description
- tuple of permitted types for this parameter
- default value (which must be numeric!) for optional parameters

5.1 Example of a simple function

Here is a simple function which generates a greyscale from an incoming image. There is an optional numeric argument which is used as a flag to control how the function processes RGB images: if the flag is true, 3-channel images are converted using OpenCV's method[2], otherwise the mean of all channels is used. This latter method is always used when the image does not have 3 channels.

```
def funcGrey(args, optargs):

    # Note that documentation strings are important: they are used in the
    # help system!

    """Greyscale conversion. If the optional second argument
    is nonzero, and the image has 3 channels, we'll use CV's
    conversion equation rather than just the mean."""

    # get the first argument as an ImageCube

    img = args[0].get(Datum.IMG)

    # get the image's sources, combined into a single set

    sources = set.union(*img.sources)

    # get the optional argument (or the default if not
    # provided).

    if optargs[0].get(Datum.NUMBER) != 0:

        # We are using the OpenCV method for 3 channels

        if img.channels != 3:
            # but there aren't 3 channels, raise an exception!
            raise XFormException('DATA', "Image must be RGB for OpenCV greyscale conversion")

        # generate a new image cube from the greyscale data, but keep the same
        # image mapping. Use the combined sources for this single channel.
```

[2] $0.299r + 0.587g + 0.114b$

```
            img = ImageCube(cv.cvtColor(img.img, cv.COLOR_RGB2GRAY), img.mapping, [sources])
        else:
            # create a transformation matrix specifying that the output is a single channel which
            # is the mean of all the channels in the source

            mat = np.array([1 / img.channels] * img.channels).reshape((1, img.channels))

            # use it to generate the image
            out = cv.transform(img.img, mat)

            # and turn this into an image cube
            img = ImageCube(out, img.mapping, [sources])

        # return the image cube as a Datum
        return Datum(Datum.IMG, img)
```

We can then register this function. This is a built-in function, so it is registered inside the constructor for the `ExpressionEvaluator` owned by the `XFormExpr` type object:

```
            self.registerFunc("grey", # name
                "convert an image to greyscale", # description
                # a single mandatory parameter: an image
                [Parameter("image", "an image to process", Datum.IMG)],
                # a single optional parameter (which must be numeric):
                [Parameter("useCV",
                   "if non-zero, use openCV greyscale conversion (RGB input only): 0.299*R +
                        0.587*G + 0.114*B",
                    Datum.NUMBER, deflt=0)],
                funcGrey)
```

6 Anatomy of an `XForm`: how to write nodes

As noted above and shown in Fig. 3, all nodes are implemented as subclasses of `XFormType`. Each node is an `XForm` with a link to an `XFormType` object controlling its behaviour[3]. There is only one object of each `XFormType` class; they are singletons. The data differentiating each instance of a given node type is stored in the `XForm` itself.

To write a new node type we need to write a new `XFormType`, create a singleton object of that class, and register it with the system (so the user can see it). The last two steps are dealt with automatically; all we need do is write the class inside the `xforms` directory and make sure it has the `@xformtype` decorator to ensure it is registered and is a singleton, as described in Sec. 3.2.1.

In addition to an `XFormType`, it may be necessary to write a subclass of `ui.tabs.Tab` to display its controls and output. If the new node only displays an image and has no extra controls, the built-in `TabImage` can be used: I will discuss this case first.

6.1 An example

The required methods are described in Sec. 3.2.1. This section will give an example of how to build an image manipulation node — an image normalisation node, which will normalise all channels to the range [0,1]. It will also honour regions of interest: only pixels inside the currently active ROI will be processed. This makes image processing a little more complicated.

[3] an example of "favouring composition over inheritance."

6.1.1 Writing the operation

We will be working in a file inside the `xforms` directory, which is imported by `main.py`. We'll call this file `xformnorm.py`. First, we need to write a function to normalise the image as a 3D numpy array, taking into account a boolean mask of pixels to ignore (for the region of interest). The declaration is simple:

```
def norm(img, mask):
```

Now we need to generate a numpy masked array from the image and mask. Note that the mask passed in uses True to indicate array elements which should be used — this is intuitively more obvious, but the construction of a masked array uses True to indicate elements which are masked out. Thus we need to negate the mask:

```
    masked = np.ma.masked_array(img, mask=~mask)
```

Now we need to create a copy of the array to write the data to because we don't want to modify the original image:

```
    cp = img.copy()
```

Next we want to find the minimum and maximum of the pixels in the masked image (i.e. ignoring unmasked pixels):

```
    mn = masked.min()
    mx = masked.max()
```

If the range is zero, we generate an error — we'll return this and deal with it in `perform()`, our node's actual work function. We also generate a zero image as the result. If the range is OK, the exception is None and the result image is the input image normalised to the range of the masked pixels:

```
    if mn == mx:
        ex = XFormException("DATA", "cannot normalize, image is a single value")
        res = np.zeros(img.shape,np.float32)
    else:
        ex = None
        res = (masked - mn) / (mx - mn)
```

We now put the result image into the image copy we generated earlier, but this time we don't negate the mask (because `putmask` works the right way — pixels which are True in the mask are written). We then return the exception and the modified image copy.

```
    np.putmask(cp, mask, res)
    return ex, cp
```

As you can see, error handling and dealing with regions of interest is often the most complicated part of a node! Now we can start to write the actual node class.

6.1.2 The `XFormType` subclass

As described above in Sec. 3.2.1, the behaviour of an `XForm` node is determined by the `XFormType` singleton to which it is linked. The code for this class will start like this:

```
@xformtype
class XformNormImage(XFormType):
```

We're creating a new subclass of `XFormType` and giving it the `@xformtype` decorator, which will create an instance and register it automatically with the application. Because of this, the declaration is the only place where the name of the class is used. Now the constructor:

```
    def __init__(self):
        super().__init__("normimage", "processing", "0.0.0")
        self.addInputConnector("", Datum.IMG)
        self.addOutputConnector("", Datum.IMG)
        self.hasEnable = True
```

The superconstructor call sets the node name, the group under which it appears in the palette, and a version number. The next two lines define an input and output connector. These have optional names, which are both empty in this example. This is the usual approach: names are only used to annotate the nodes in the graph, and are left empty where they are obvious. In the code, connectors are referenced by index in order of creation. Both connectors here are images, with type `IMG`. See `datum.py` for all the types. The final line sets the node to have an `enabled` value and associated toggle button. This is used in nodes which may be computationally intensive, to temporarily "turn them off." This will usually be handled by passing through the data unchanged.

The next method is `createTab()`, which is used to create a tab for a particular node. Like most methods in this class, it takes a reference to the node. It also takes a reference to the main window in which the tab should be created:

```
    def createTab(self, n, w):
        return TabImage(n, w)
```

In this class, we are using the built-in `TabImage` tab which has a single canvas widget displaying an image. Many node types use this.

The `init()` method is used to initialise an `XForm` node to be a particular class (beyond setting the `type` field, which is done elsewhere). It therefore takes a node, and typically sets up private data this type requires inside the node object. This uses an advantage of Python[4]: we can add fields to objects after they have been instantiated. Here we just add a `img` field to store the normalised image, initialising it to None (the canvas widget will display a None image as a blue empty rectangle):

```
    def init(self, node):
        node.img = None
```

6.1.3 Writing the perform method in full

Finally we come to the `perform()` method, which describes how this type of node performs its operation. Here I'll describe it in full. Many nodes, however, perform a simple operation with a few numeric parameters on an image and output an image. These nodes can be written in a simpler way which also allows them to be used as functions in the *eval* node. This will be dealt with later.

Again, it requires a reference to the `XForm` node object which contains the node state, and its first action is to pre-set the output image to None and then fetch the input image:

```
    def perform(self, node):
        node.img = None
        img = node.getInput(0, Datum.IMG)
```

This will get a reference to the image stored in the output field of the node connected to this node's zeroth input (the first added in the type constructor). If there is no connection, None will be returned. None will also be returned if the type is not `IMG`: this version of `getInput()` will unwrap the `Datum` and type-check (see Sec. 3.2.3). We then check to see if the input is None. If it

[4] Although from a software engineering point of view it is also a weakness.

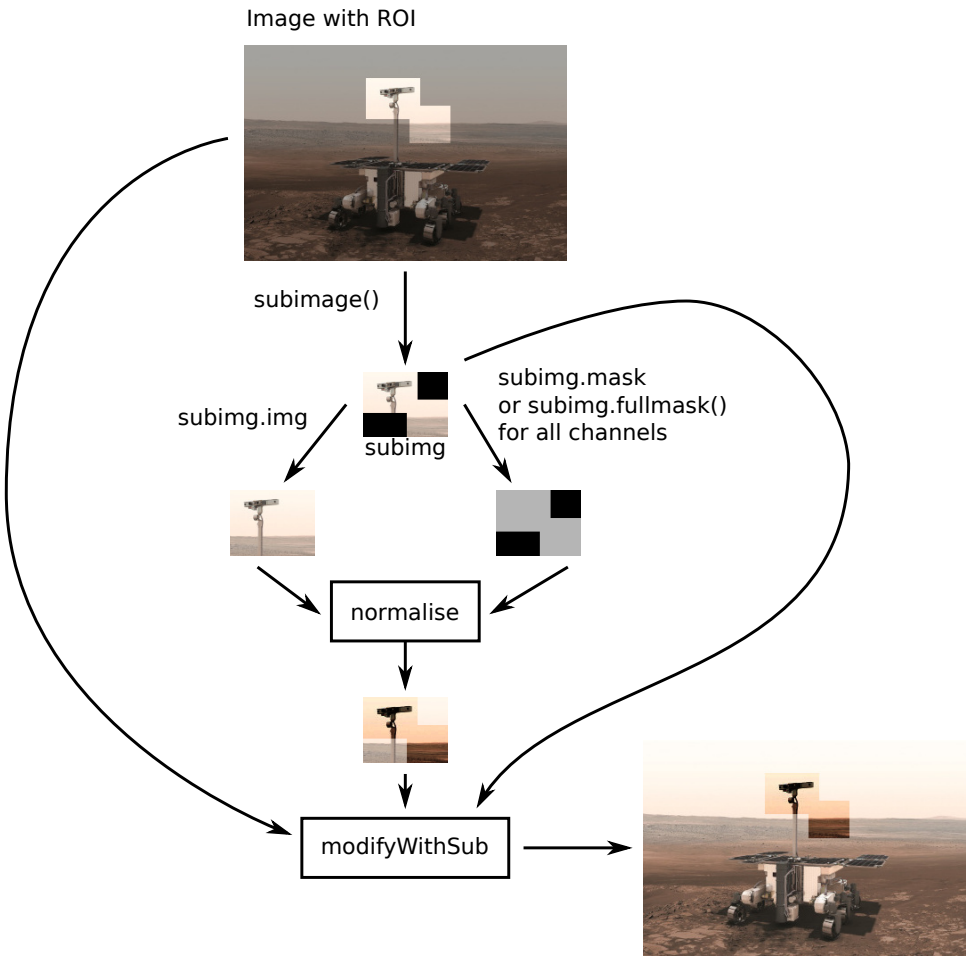
is, we do nothing (leaving `node.img` as None). We also check to see if the node has been disabled (this node has an Enabled button):

```
        if img is not None:
            if node.enabled:
```

Assuming these are both true, we use the `ImageCube.subimage()` method to fetch a `SubImageCubeROI` object containing information about which parts of the image are in the current region of interest: the rectangle of pixels bounding the region and the mask describing which pixels within that bounding box are in the region. We can then pass these two numpy arrays into the normalization function we wrote earlier, obtaining another numpy array: the bounded image normalized. We then call `modifyWithSub()`, which creates a new ImageCube in which the section described by the region of interest (taking into account the mask) has been replaced by the new, normalized image data:

```
                subimage = img.subimage()
                ex, newsubimg = norm(subimage.img, subimage.fullmask())
                if ex is not None:
                    node.setError(ex)
                node.img = img.modifyWithSub(subimage, newsubimg)
```

Note the error check: our normalization function returns an exception and an image. If the exception exists (is not None), we set the error state in the node (see Sec 3.3.2). It's still fine to patch in the subimage, though. This entire process is shown in Fig. 10.



norm.pdf

Figure 10: The image processing in the norm node

Finally, if the node was not enabled we set `node.img` (our output) to be the input image:

```
            else:
                node.img = img
```

and output the image to the node's zeroth (and only) output, wrapping it in a `Datum`:

```
        node.setOutput(0, Datum(Datum.IMG, node.img))
```

6.1.4 Writing the perform method using operation functions

> For now, see `xformcurve.py` along with `operations` and its files.