PCOT design notes

James Finnis (jcf1@aber.ac.uk)

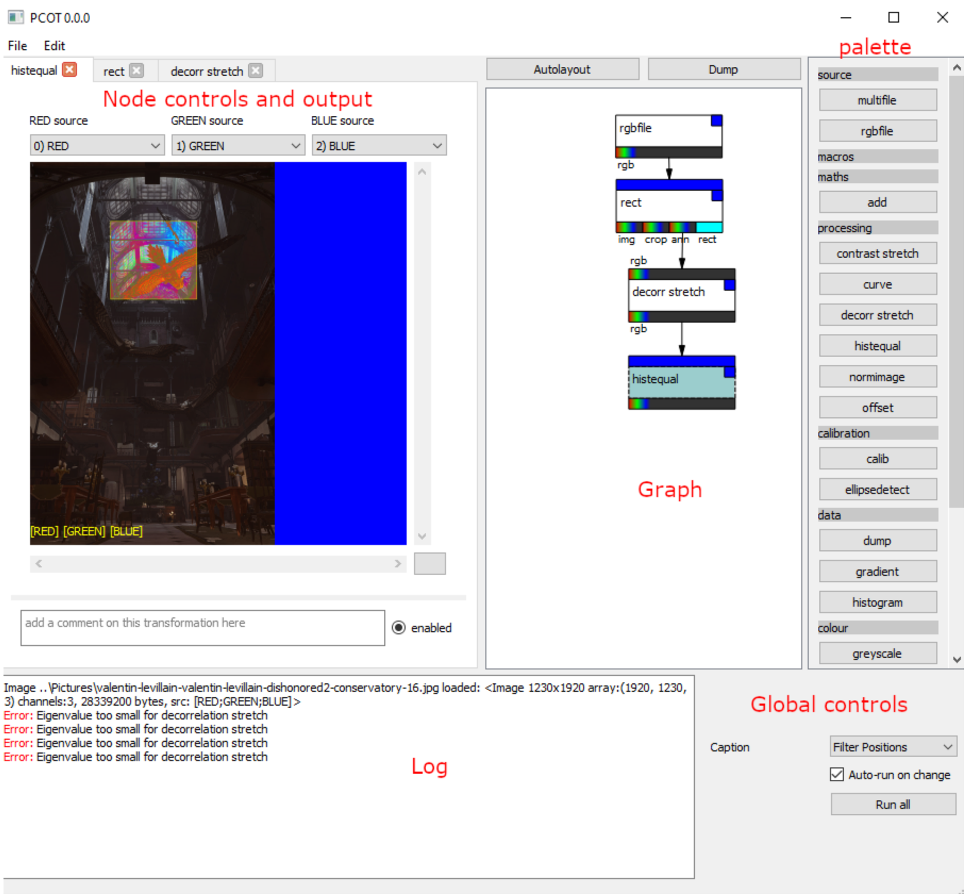
March 5, 2021

Contents

1 Introduction

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

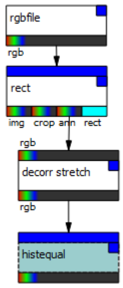
PCOT is based around a directed graph of nodes which perform transformations of data. For this reason, the nodes are sometimes called "transforms" and are represented by the `XForm` class in the code. Usually the data in question is an image, or rather an "image cube": these have an arbitrary number 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 RGB image from a file, perform a decorrelation stretch, and then a histogram equalisation on the three channels. 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 the `rect` 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:
    ## @var name
    # name of the type
    name: str
    ## @var group
    # the palette group to which it belongs
    group: str
    ## @var ver
    # version number
    ver: str
    ## @var hasEnable
    # does it have an enable button?
```

are straightforward (and note the Doxygen annotations): 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:

```
    ## @var instances
    # 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 graphscene
    import PyQt5.QtWidgets
```

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

```
## @var rect
# the main rectangle for the node in the scene
rect: ['graphscene.GMainRect']
## @var inrects
# input connector rectangles
inrects: List[Optional['graphscene.GConnectRect']]
## @var outrects
# output connector rectangles
outrects: List[Optional['graphscene.GConnectRect']]
## @var helpwin
# 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 actually 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.

2 The data model

The data model consists of two parts — the data itself (largely image cube data) and the graph. By far the most important kind of data from the point of view of this document is the image data, which ties into the user interface in complex ways. Other forms of data do exist, but these are much simpler.

2.1 Graph and nodes

The graph is a directed graph 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.

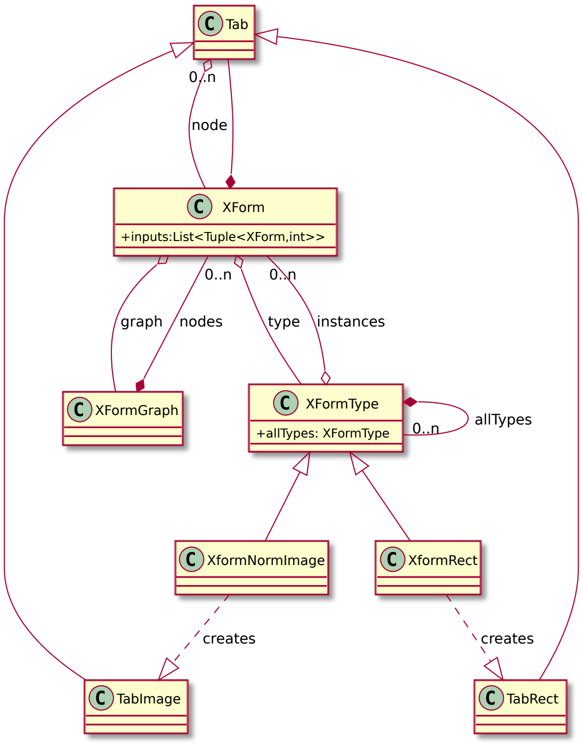
2.1.1 XFormType and type registration

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.

The singletons are automatically created and registered when the class is defined, through the `@xformtype` decorator. This does the following:

- 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 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.



xform.pdf

Figure 3: `XForm` and graph model

2.1.2 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.

2.1.3 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.

2.2 Performing the graph

The graph needs to be "performed" — its nodes executed — whenever the data changes, which is generally whenever the graph is edited or a node control changed in a tab. This is done by calling `changed()` on the node's tab, which calls `changed()` on the tab's node's graph, passing in the node. Here is the relevant code in `ui.tabs.Tab`:

```
def changed(self):
    self.node.graph.changed(self.node)
```

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. 2.4.2 for why this is a problem).

The `XFormGraph.performNodes()` 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.

2.2.1 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`.onNodeChanged()`
  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 only two 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. 2.4 for more details).

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.

2.2.2 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 connection type system in `conntypes.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).

2.3 Image data

Most classes making up the image data model are described in the `pancamimage.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. 4 although some links to channel sources and mapping from nodes are omitted; these will be explained later.

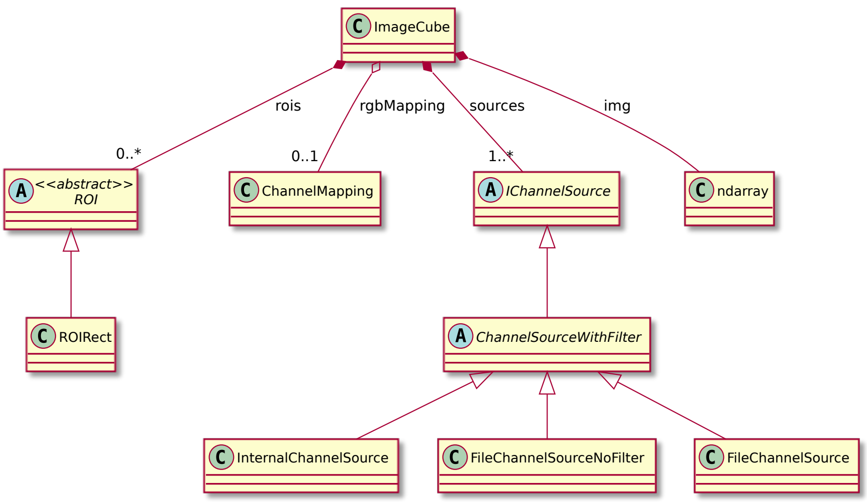


image.pdf

Figure 4: 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 of information. There is no upper limit on the number of channels in an image (or image cube) beyond system memory.

2.3.1 IChannelSource and its implementations

Each `ImageCube` has a number of channels, and for each channel there must be a corresponding entry in its `sources` list. This describes where that channel came from, so that (typically) filter information can be preserved, where appropriate, through the graph. The sources for each channel are a set of `IChannelSource` objects. For example, if an image was loaded through the RGB loader, it might have three “fake” channel sources for red, green and blue. Thus the sources will be

```
[ {RED}, {GREEN}, {BLUE} ]
```

i.e. a list of three sets, each with a single source. If the image is then converted to greyscale, this could become

```
[ {RED,GREEN,BLUE}, {RED,GREEN,BLUE}, {RED,GREEN,BLUE} ]
```

because each channel now contains information from the red, green and blue channels in the source file. The `RED`, `GREEN` and `BLUE` values refer in this case to `FileChannelSourceNoFilter` objects which contain “fake” filter information and a filename identifier. Each `IChannelSource` contains methods for accessing:

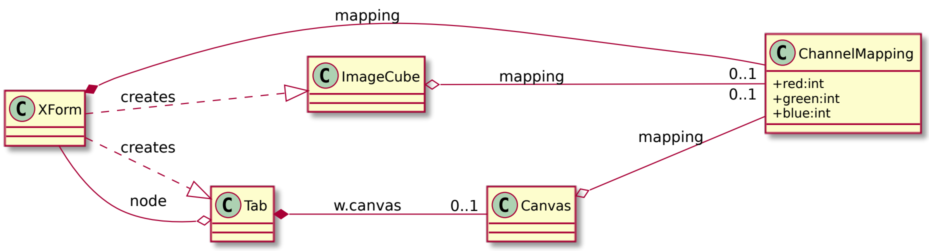
- an **identifier string** for the source from which the channel was acquired (typically a filename or data ID);
- a **filter** and methods for obtaining the filter name, filter position and an actual filter reference (for extra data such as centre wavelength) (note that much of this information will be “fake” for images loaded from plain RGB files);
- methods for getting string descriptors for this source.

Nodes generate and process this information in different ways. For example, a *gradient* node takes a single channel and converts it into an RGB image with a colour gradient: here, the output image’s sources are “internal RGB” sources with no identifier or sensible filter data because the output’s colour is entirely artificial. In contrast, a the *curve* for performing a sigmoid function on all channels of an image will give the output image the same sources as the input image.

Sources are used to keep track of each channel as it moves through the graph so they can be processed and displayed appropriately: Fig. 1 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.

2.3.2 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. 5.



rgb.pdf

Figure 5: RGB mapping classes

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.
- 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.

2.3.3 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. 4, 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)
    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, 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.

2.4 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.

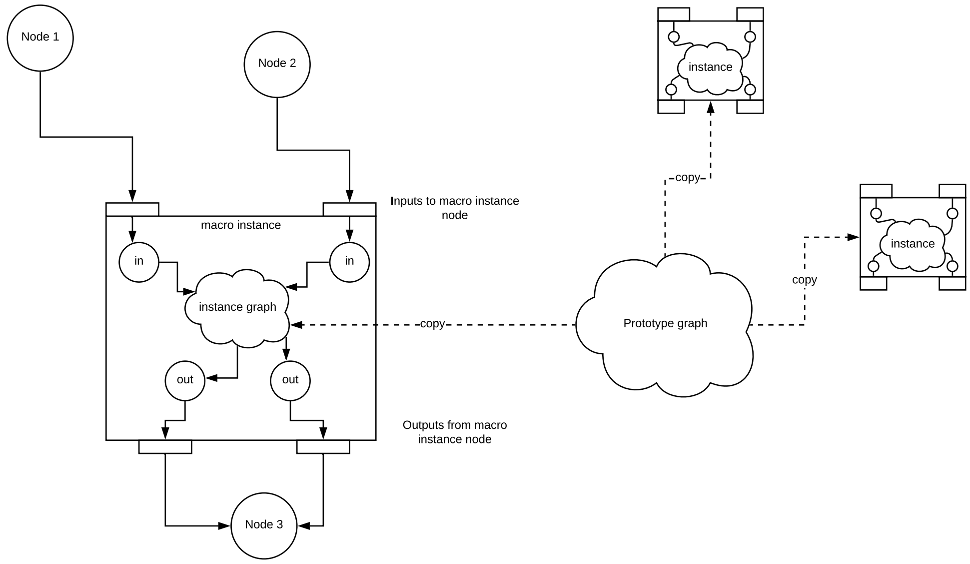
Users can create a new macro, which will open a window with a new graph in it. This is distinguished from the main window by its slightly different background colour. The user can create and manipulate transform nodes inside this graph, as usual. This is the **prototype graph** for the macro. The user 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.

2.4.1 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. 6. 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



macroobjs.pdf

Figure 6: An example of a graph containing macros

- 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)
  - 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.

2.4.2 Macro inefficiencies

As noted above in Sec. 2.2, 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)
```

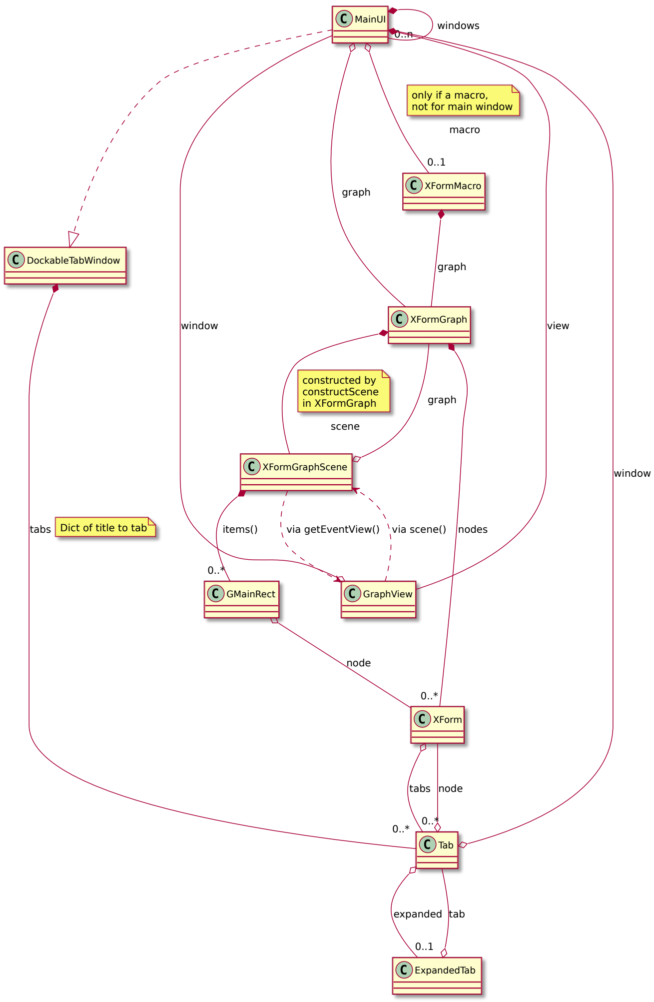
Here, `inst` is each `XFormMacro` node inside the main graph. Thus `inst.graph` will be the main graph (for an non-nested macro). The `self` value is the macro prototype graph, because this method was called on an object inside that graph. This code therefore calls `performNodes()` on the main graph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

2.5 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. 7. 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;
- `graphview.py`: contains `GraphView`, the `QGraphicsView` subclass which encapulates a view on the graph scene;
- `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).



ui.pdf

Figure 7: UI class diagram

2.5.1 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.

3 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[1]. 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` annotation to ensure it is registered and is a singleton, as described in Sec. 2.1.1.

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.

3.1 An example

The required methods are described in Sec. 2.1.1. 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.

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

3.1.1 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.

3.1.2 The `XFormType` subclass

As described above in Sec. 2.1.1, 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` annotation, 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("", "img")
        self.addOutputConnector("", "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 `conntypes.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[2]: 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
```

Finally we come to the `perform()` method, which describes how this type of node performs its operation. 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)
```

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. We then check to see if the input is None. If it 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:
```

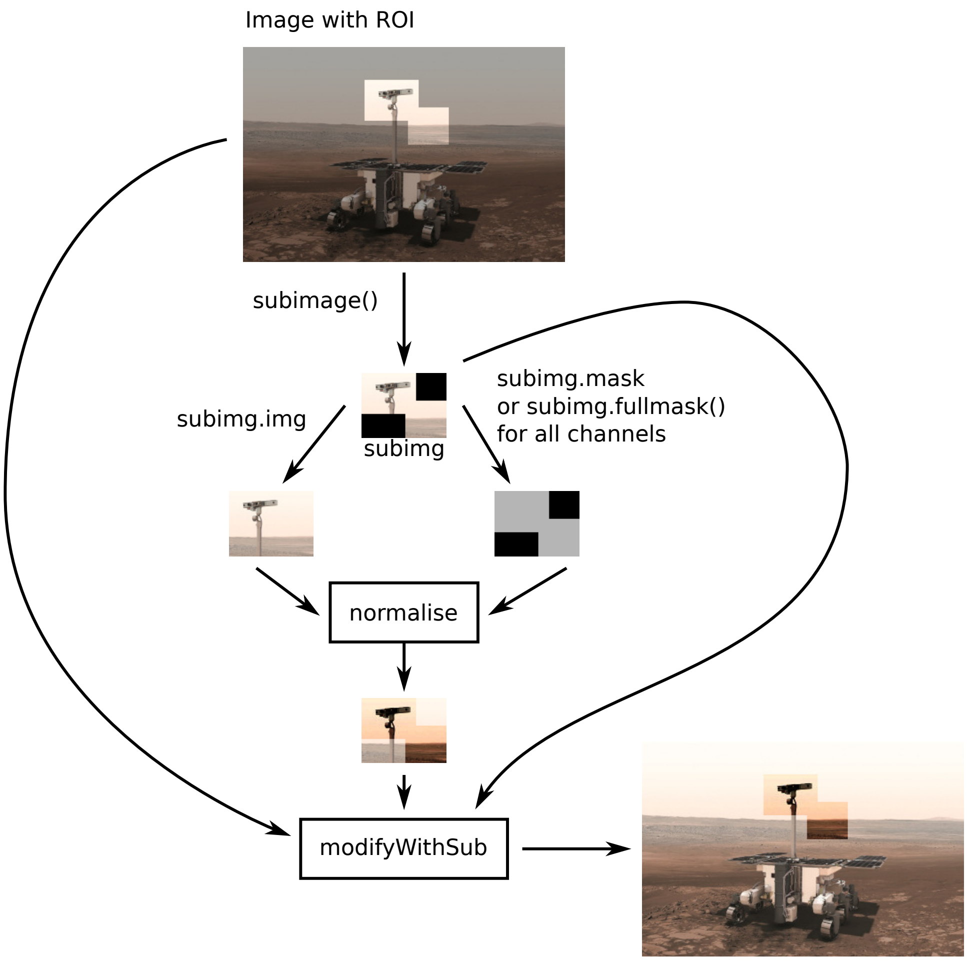
Assuming these are both true, we use the `ImageCube.subimage()` method to fetch a `SubImageCubeROI` object containing information about which parts of the image are in the current region of interest: the rectangle of pixels bounding the region and the mask describing which pixels within that bounding box are in the region. We can then pass these two numpy arrays into the normalization

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

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 2.2.2). It's still fine to patch in the subimage, though. This entire process is shown in Fig. 8.



norm.pdf

Figure 8: 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:

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