PCOT design notes

James Finnis (jcf12@aber.ac.uk)

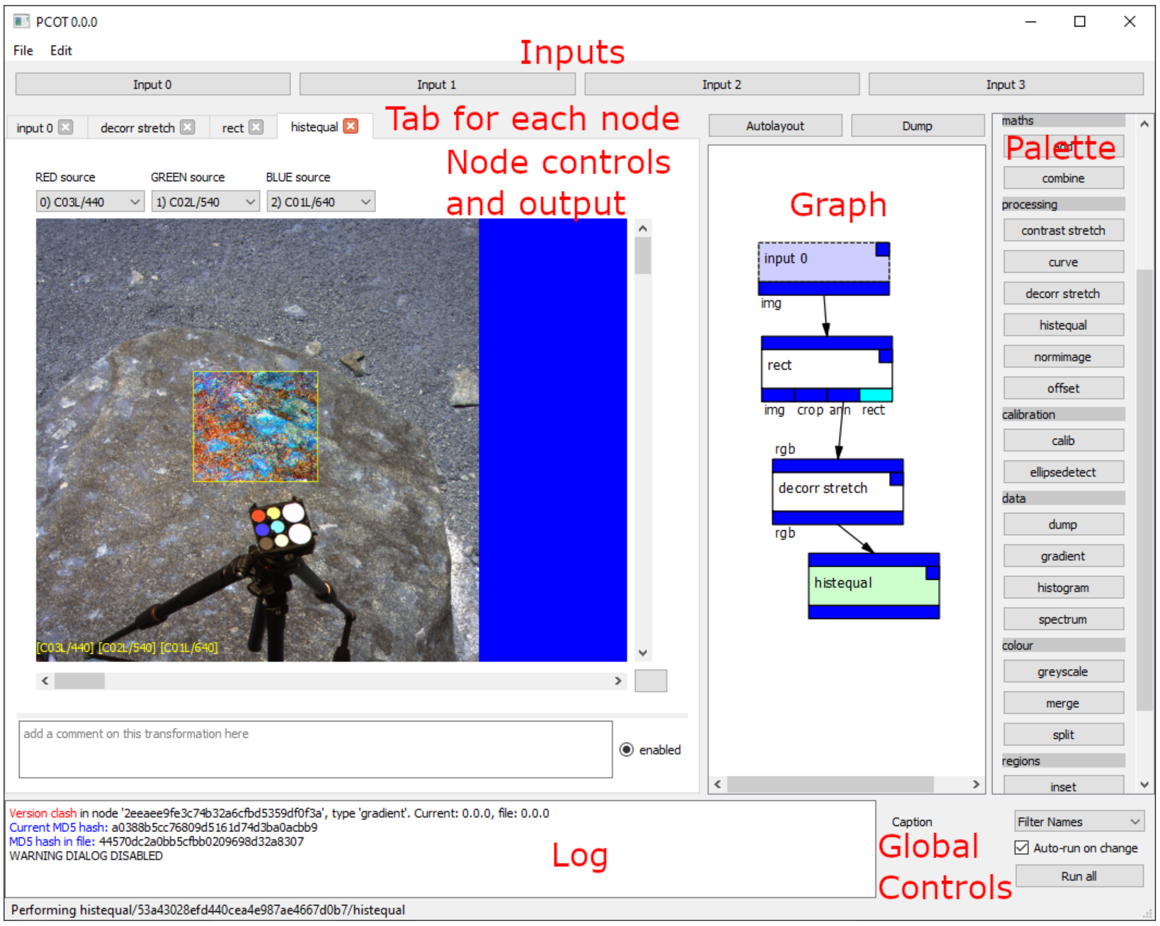
December 20, 2021

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.



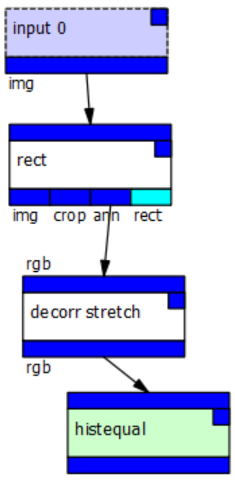
```
Version clash in node '2eeaee9fe3c74b32a6cfbd5359df0f3a', type 'gradient'. Current: 0.0.0, file: 0.0.0
Current MD5 hash: a0388b5cc76809d5161d74d3ba0acbb9
MD5 hash in file: 44570dc2a0bb5cfbb0209698d32a8307
WARNING DIALOG DISABLED
```

```
Performing histequal/53a43028efd440cea4e987ae4667d0b7/histequal
```

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. **??** — 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:
    ## @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:
    from ui 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.

1.2 Structure

The code is structured thus:

```
PCOT                    { main directory }
\-src                   { source code }
  \-pcot                { top-level package }
    \-assets            { data, typically .ui files }
     -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 contain `setup.py` and `setup.cfg` for wheel creation and installation. The `setup.py` is kept so that users can do `python setup.py develop` to create an editable install.

ToDo

- Change getInput and getOutput docs
- update conntypes doc
- add Datum details

2 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. **??**);
- the input manager and the state of the various input methods contained in each input (see Sec. **??**);
- 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.

2.1 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.

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.

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. **??**] 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.

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.

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.

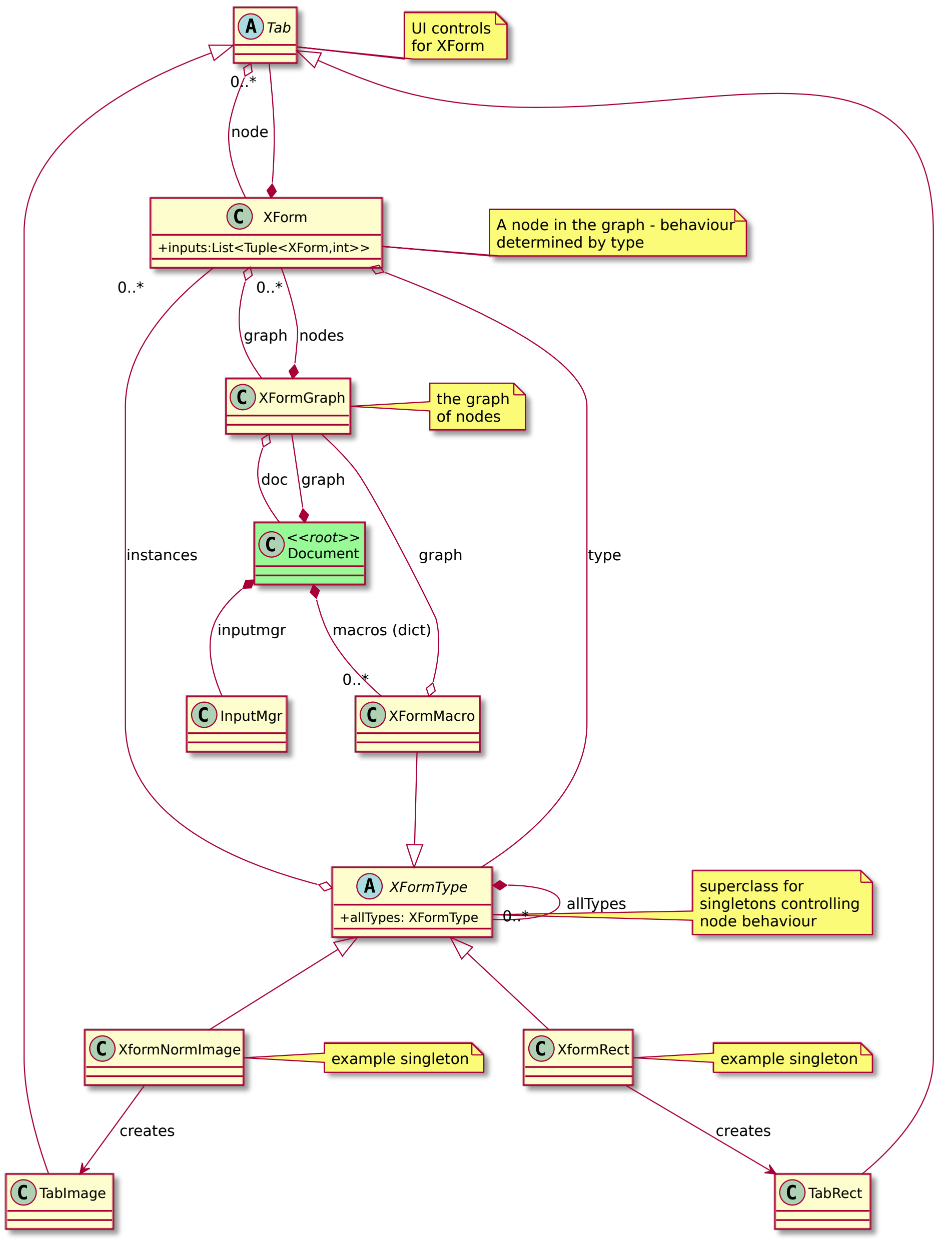
2.1.4 Overriding types and the variant type

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.

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, a node control changed in a tab, or an input is changed (see Sec. **??** 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. **??** 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).



xform.pdf

Figure 3: `XForm` and graph model

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
    $t$.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 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. **??** 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.