nip2 Manual

Version 7.18

John Cupitt, Rachel Billinge, Joseph Padfield, Clare Richardson, David Saunders

"It's quite simple really, and at the same time, rather complicated."
— A. Haddock, Sea captain (rtd.)

This document formatted April 16, 2009

Contents

List of Figures

List of Tables

Chapter 1

Getting started

nip2 is a user interface for the VIPS image processing library. It is designed to be fast, even when working with very large images, and to be easy to extend.

This guide is split into quite a few chapters:

- If you want to use nip2 to assemble infrared mosaics, you should read Chapter 3 on page 11. The middle section in the tutorial (see §2.2 on page 6) does IR mosaics very quickly.
- If you want to use nip2 for general image processing, work through Chapter 2 on page 3.
- If you have specific questions about some part of nip2's user-interface, look at Chapter 4 on page 17.
- If you're really hardcore, take a look at Chapter 6 on page 31, which covers programming.
- If you want to know more about VIPS, the image processing package underlying nip2, try the *VIPS Manual*.

If nip2 has installed correctly you should see something like Figure 1.1 when it starts up.

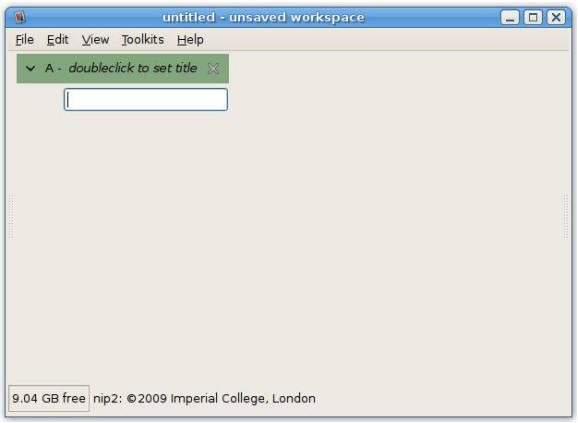

Figure 1.1: nip2 as it starts up

Chapter 2

Tutorial

This chapter runs very quickly through how to use nip2's user-interface. See Chapter 4 on page 17 if you want more details on how the different bits work.

2.1 Quick interface tour

Start up nip2. You should see something like Figure 1.1 on page 1 (the exact look might be different on your system).

The menus at the top of the window are very ordinary, except for the Toolkits menu which contains all of the image processing operations. The green thing is the current column (the thing that new objects will get added to). Click on the 25 GB free text and it toggles between showing space free on your disc and space free for calculations.

Click on File / Open to get a file dialog and load up an image. nip2 can load most image formats, try it and see. Check the Pin up box to have the dialog remain after you press OK. Click on Show thumbnails and nip2 will try to display thumbnail images for all the files in the directory. You can also drag files from the desktop or from your file manager.

After you've loaded an image, nip2 should look like Figure 2.1. Double click on the thumbnail to open an image view window. Alternatively, select Edit from the right-button menu on the thumbnail. See Figure 2.2.

As well as the standard keymappings, nip2 has extra shortcuts for navigating images, see Table 2.1 on page 4. Use the View / Toolbar menu to turn on other features. You can have a status bar (shows image properties, mouse position and pixel value), a display control bar (lets you change scale and offset for display pixels, click on the arrow on the left for a useful extra menu), a paint bar and some rulers.

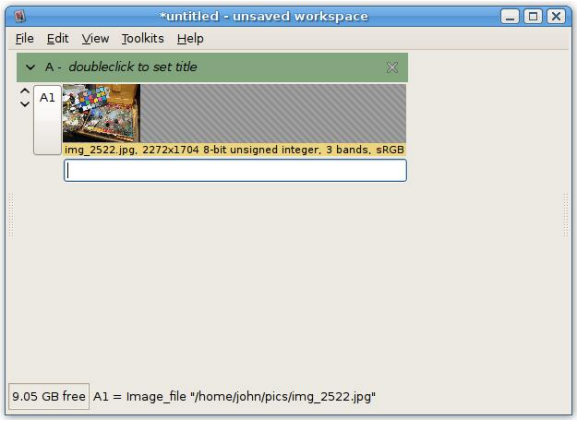

Figure 2.1: After loading an image

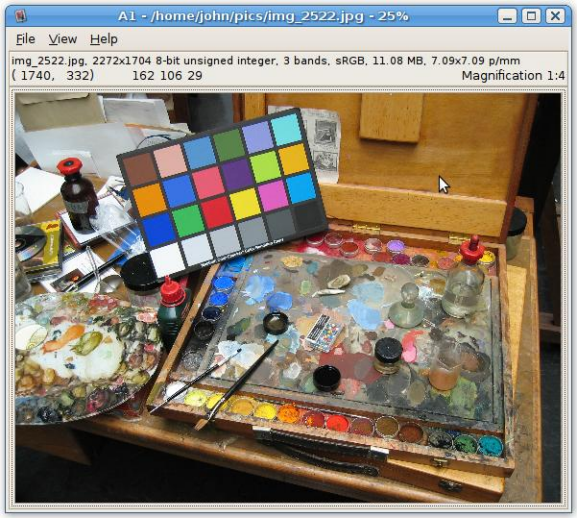

Figure 2.2: Image view window

Keys in image display widget	Action
i, +	Zoom in on mouse pointer
o, −	Zoom out
Cursor up/down/left/right	Scroll a small amount in direction
Shift and cursor up/down/left/right	Scroll a screenful in direction
Ctrl and cursor up/down/left/right	Scroll to edge of image
Middle mouse drag	Pan image
Mouse wheel	Scroll up/down
Shift and mouse wheel	Scroll left/right
Ctrl and mouse wheel	Zoom in/out
0 (zero key)	Zoom out to fit image to window
1, 2, 4, 8	Set magnification to 1, 2, 4 or 8
Ctrl and 2, 4, 8	Set zoom out factor to 2, 4 or 8

Table 2.1: nip2 shortcuts for the image view window

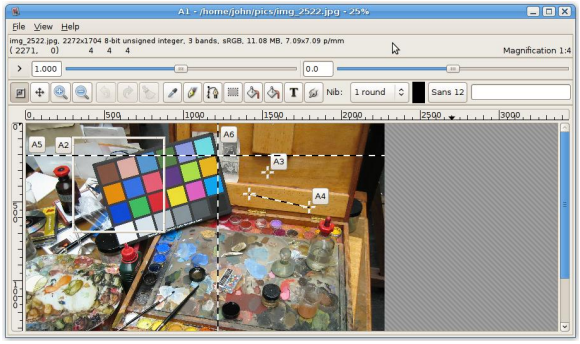

Figure 2.3: Image view window with marked regions

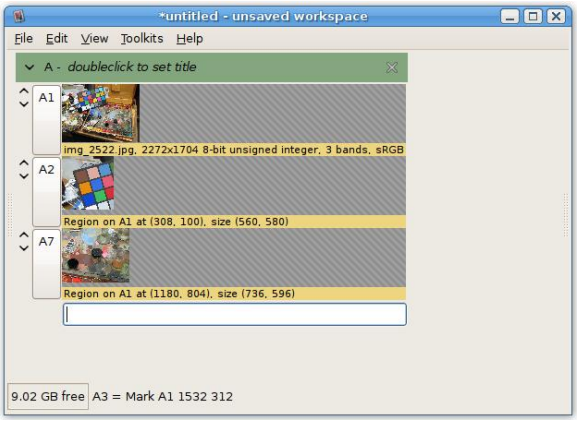

Figure 2.4: Main window, two regions marked

You can mark things on an image. Hold down Ctrl and drag down and right with the left mouse button to mark a region. Ctrl-left-click to mark a point. Drag up and left to mark an arrow (two points connected by a line). Drag from the rulers to mark guides. Right-click on a label to get a menu which you can use to remove or edit one of these things. Left drag on a label to move the things around, left-drag on the edges or corners to resize. Figure 2.3 shows the same image with stuff marked on it.

Clean up any messing about and leave two regions on your image. The main window should now look something like Figure 2.4.

There are three rows visible here, A1, A2 and A3. Each row has (from left to right) a pair of up/down arrows (these indicate that the row contains sub-rows: click on the down arrow several times to open the row up and see inside), the name button (left-click to select, Shift-left-click to extend-select, Ctrl-left-click to toggle select, click on the workspace background to unselect everything, left-drag on the name button to reorder items within the column) and the thumbnail image.

Left-click on the name button of one of these images to select it, and then click on Toolkits / Image / Transform / Rotate / Free (alternatively, if nothing is selected when you click on one of the toolkit menus, nip2 will apply the operation to the bottom object in the current column). A new row will appear representing the rotation operation. Drag the slider to rotate the image (or type an angle in degrees into the box to the left of the slider and press Return). Pick an

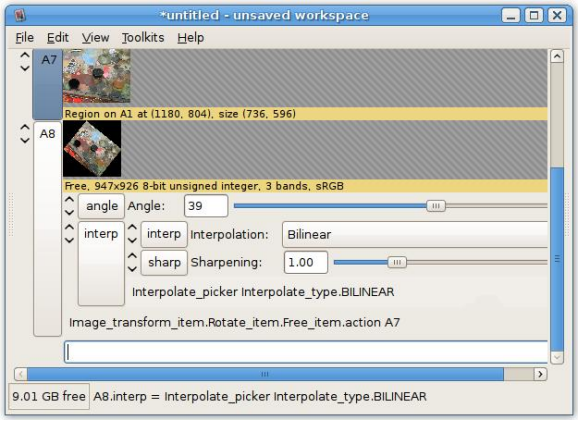

Figure 2.5: Using `Rotate` / `Free`

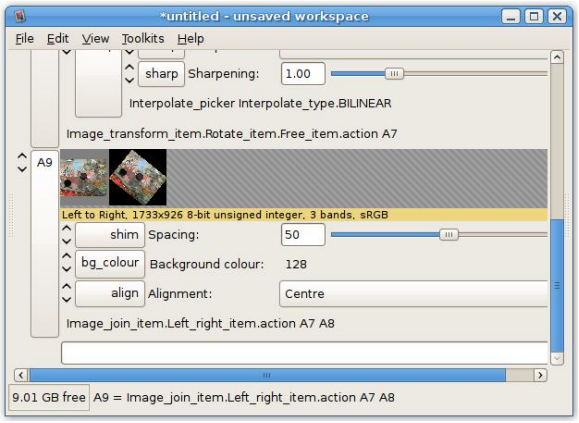

Figure 2.6: Using `Join` / `Left to Right`

interpolation type from the option menu and zoom in on some pixels to check the result. See Figure 2.5.

The same thing works for image processing operations that take two arguments. Left-click on one of your original regions, Ctrl-left-click on the rotated image (the box at the left of the main window status bar says what's selected and in what order), and click on `Toolkits` / `Image` / `Join` / `Left to Right`. A new row appears representing the join operation.

Click on `Background Colour`, type in 128 and press Return. Drag the `shim` slider to 50 (or type 50 into the box just to the left of the slider and press Return). See Figure 2.6.

The `Toolkits` menu is large and can be slow and

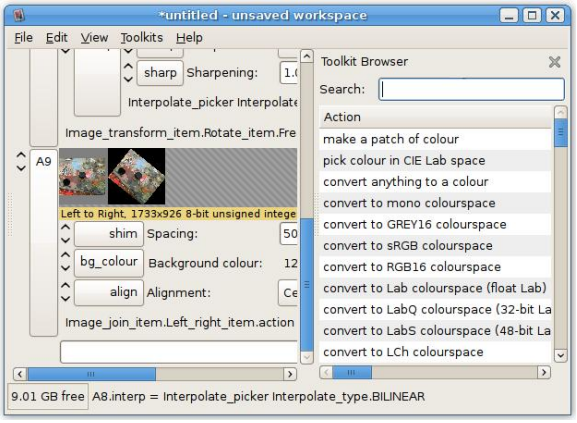

Figure 2.7: The toolkit browser

annoying to find things in, so `nip2` has several shortcuts. First, you can tear-off menus by clicking on the dotted line at the top. If you're going to be using one of the sub-menus repeatedly this can save a lot of clicking. Next, you can set keyboard shortcuts for menu items by moving the mouse pointer over the item and pressing the key combination you want to set.

Some systems won't let you edit menu shortcuts by default. For example, on GNOME, you need to enable this in `System` / `Preferences` / `Menus & Toolbars`.

Finally, there is a toolkit browser: this shows the same set of items, but in an easier-to-browse way. Click on `View` / `Toolkit Browser` and the browse side-panel will appear, see Figure 2.7. It shows all of the items as a single long list. Type into the search box at the top to only show items which match. Double-click (or press Return) on an item to activate it. Scroll to the right to see what arguments the item needs and what menu it appears in.

The box at the bottom of each column is for entering new expressions. You can type stuff here, and `nip2` will make a new row for each item you enter. Try typing `2 + 2` and pressing Return. The syntax is (almost, with a few small differences) the same as the C programming language. See §6.6 on page 34 for a list of the differences. Try multiplying the joined images by a small amount (eg. type something like `A5 * 1.2` and press Return). Normally `nip2` will pick names for new objects for you (like `A1`), but you can set a name yourself if you like. Try entering `fred = 12`.

Click the down button once on your brightened image and left-click on the area just below the thumbnail. You should see the stuff you typed to make that row. You can edit it to be anything else, press Return and nip2 will recalculate. Try going back to your original image (the one you loaded from a file), open an image view window, and try dragging one of the regions. You can change any of the sliders in the rotate or the join rows as well.

You can also edit the insides of objects. Click the down button next to one of your regions until the width and height rows appear, click on width and type height * 2. Now open the image window the region is defined on and try to resize it: you'll find that the width of the region is fixed, but that if you change the height, the width changes with it. This is a very general property of classes in nip2: you can use it to join objects together in complex ways, and to modify the behaviour of interactive objects.

Right click on a column title bar to get a useful menu. Click on File / New / Column make another column (handy for organising a workspace). If you drag from an image thumbnail on to the workspace background, nip2 will make a new column for you. You can drop thumbnails on to other thumbnails to make links. If nip2 falls over (I do hope it doesn't), you can usually get your work back by restarting nip2 and clicking on File / Search for Workspace Backups. There are a lot of preferences (perhaps too many), see Appendix A on page 49.

There is a lot of stuff in the Toolkits menus, but they do almost all have tooltips. If you let your mouse hover over a menu item for a moment you should get some helpful text. The toolkit menu is organised by object type. If you want to do something to a matrix, look in the Toolkits / Matrix menu. The exception is Toolkits / Tasks which repeats many of the regular toolkit items, but groups them by typical tasks instead.

Operations can work on groups as well as on single images, so you can batch things up. If you save a group of images, nip2 will number each image sequentially for you. You can use Edit / Duplicate to make copies of objects. If you select lots of objects and duplicate them, nip2 will (fairly intelligently) rename everything for you so it all still works.

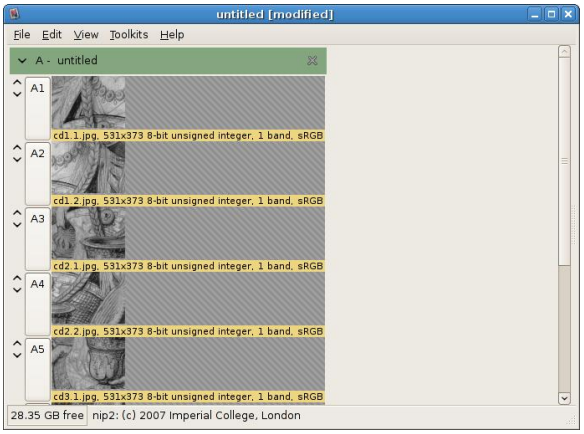

Figure 2.8: Loading the sample images

2.2 nip2 for reflectogram mosaics

This section quickly builds an infrared reflectogram mosaic using the sample images that come with nip2. See Chapter 3 on page 11 for detailed coverage.

Click on File / Open Examples. You should see a directory called 1_point_mosaic. Doubleclick and you'll see a file called 1pt_mosaic.ws. Doubleclick that and you'll load the workspace for this example.

If you'd rather make the workspace yourself, click on the file type filter and select All files. A set of 8 images should appear. Click on the first file, shift-click on the last, and click Open. See Figure 2.8.

The images have been named to match their positions in the mosaic, so for example cd2.1.jpg is the first image in row two. Open up viewing windows for the first two images by double clicking on the thumbnails. Move the two opened images viewers so that they are side by side. Adjust the zoom (using the i and o keys) and the pan (by dragging with the middle mouse button) so that the overlap area is visible in both images.

Mark a tie-point on each image by Ctrl-left-clicking on a feature you can see in both images, see Figure 2.9 on page 7. Move a point after you've marked it by dragging on the label. You don't need to be exact: nip2 just uses the point you select as the start point for a search. It can cope with misses of up to about 10 pixels. To mosaic the two images together, click on Toolkits / Tasks / Mosaic / One Point / Left to Right. See Figure 2.10 on page 7.

Picking items deep in the toolkit menu is fiddly, so

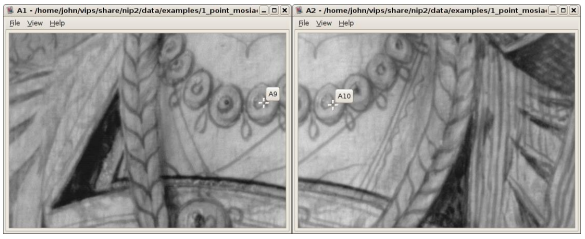

Figure 2.9: Ready to join

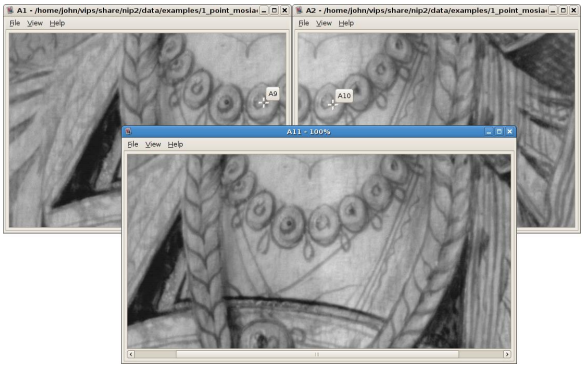

Figure 2.10: Joined images

nip2 has several shortcuts. First, you can tear off any toolkit menu by clicking on the dotted line at the top. Secondly, you can assign any keyboard accelerator to any menu item. Navigate to the menu item and while it is selected, press the key combination you want to use as a shortcut (for example, Ctrl-L might be good for Mosaic / One Point / Left to Right). Now whenever you press Ctrl-L with the keyboard focus in the main window, you will do a left-right mosaic join. Finally, you can use the toolkit browser to display a selection of the tools in a pane on the right-hand side of the main window. Click on View / Browse Toolkits, then type "mosaic" into the search box at the top. The toolkit browser will display all items related to mosaicing.

Some systems won't let you edit menu shortcuts by default. For example, on GNOME, you need to enable this in System / Preferences / Menus & Toolbars.

Join the rest of the pairs of sample images together left-right. Figure **??** on page **??** shows how they should fit together. Once you have made all the rows, join the rows together in turn to make the complete image using Mosaic / One Point / Top to Bottom.

When you've built the whole thing you'll see that there are differences in brightness between the tiles that make up your composite image. You can fix most problems like this automatically by selecting your final mosaiced image and clicking on Mosaic / Balance. This operation takes your mosaic apart, examines the overlap areas for differences in brightness, calculates a set of adjustment factors to minimise these differences, and then rebuilds the mosaic.

There can be some problems left even after mosaic balance. Use Mosaic / Tilt Brightness to remove any left-right or up-down graduations in brightness.

Save your mosaic workspace for future reference by clicking on File / Save Workspace. To save just the mosaiced image, right click on the thumbnail and select Save As.

2.3 nip2 for nerds

This section sprints through a bit of nip2 programming, see §6 on page 31 for full details and a more formal definition of the language.

The insides of nip2 are built with nip2's own pro-

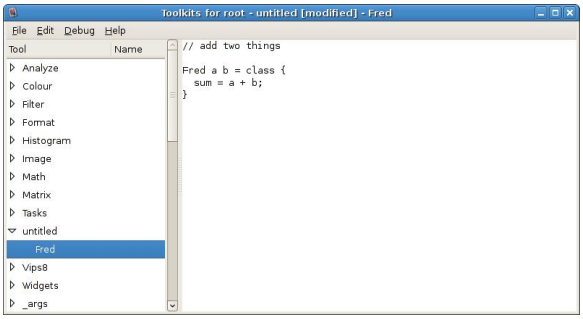

Figure 2.11: Programming `Fred`

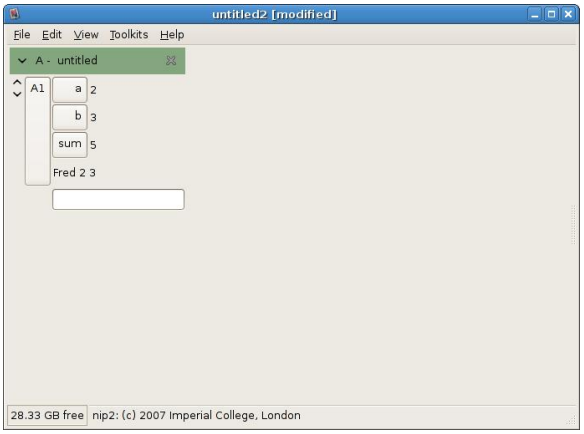

Figure 2.12: Main window `Fred`

gramming language. It's a pure lazy functional language with classes. It's C's expression syntax (more or less) plus approximately Miranda/Haskell function syntax, plus some basic class stuff. `nip2`'s main window is a class browser for this programming language.

Click on `Toolkits` / `Edit Toolkits` in `nip2`'s main window to pop up the programming window (see §4.4 on page 22 for details on all the bits in the window), then in the edit area there type:

```
// add two things

Fred a b = class {
  sum = a + b;
}
```

This defines a class called `Fred` whose constructor takes two arguments, `a` and `b`. There's one member, called `sum`, which is `a` and `b` added together.

In the program window, click `File` / `Process`. This makes `nip2` read what you typed, parse it, compile it and update itself. The program window should now look like Figure 2.11.

If you look back at the main `nip2` window, a new menu will have appeared under `Toolkits` called `untitled`. If you click on that, there will be a menu item called `Fred`. Let your mouse linger, and you'll see a tooltip too.

In the main window, type `Fred 2 3` into the box at the bottom of the current column. Press Return and `nip2` will make a `Fred` for you. Click on the down arrow to the left of your new `Fred` once to see the members of `Fred` (just `sum` in this case), click again to see the class parameters too. The main window should look like Figure 2.12.

Click to the right of `b`, type in a new value and press Return. The `sum` member should update. `nip2` keeps track of dependencies between rows, but it also tracks dependencies inside rows, both ones that come from the class, and ones created by any edits you do to the class instance after creating it. You won't see it in a simple example, but `nip2` also discovers and tracks dependencies which can arise at run time. Click on the text just to the right of the `b` button again, type `a` and press Return. Now edit `a`: press Return and both `b` and `sum` will update.

You can use `Fred` to add any two things together. Click on `Toolkits` / `Widgets` / `Scale` to make a scale widget, press Ctrl-U (the keyboard shortcut for `Edit` / `Duplicate`) to duplicate it, and finally click on `Toolkits` / `untitled` / `Fred`. Open up the new `Fred` and try dragging some of the scales around. The main window will look like Figure 2.13 on page 9.

The scales are classes too (instances of `Scale`). You can open them up and do strange things with them as well. Open up one of the scales you made (eg. `A2` in Figure 2.13 on page 9) and change the `from` parameter to be `A3.value`. Now try dragging the sliders again.

Try dragging the `sum` slider. Now go back and drag one of the original sliders. You'll see that `sum` no longer updates, it's stuck at the last position you dragged it to. This is because there are now two things affecting the value of `sum`: the underlying code (the `a + b` inside `Fred`), and the position you dragged the slider representing `sum` to. `nip2` has the rule that graphical edits (dragging the slider) override code. To make `sum`

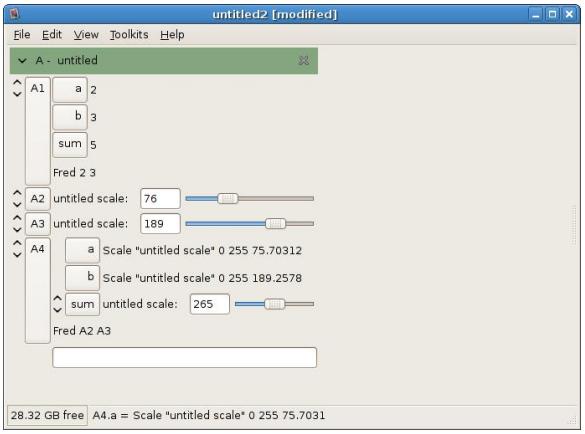

Figure 2.13: Scale `Fred`

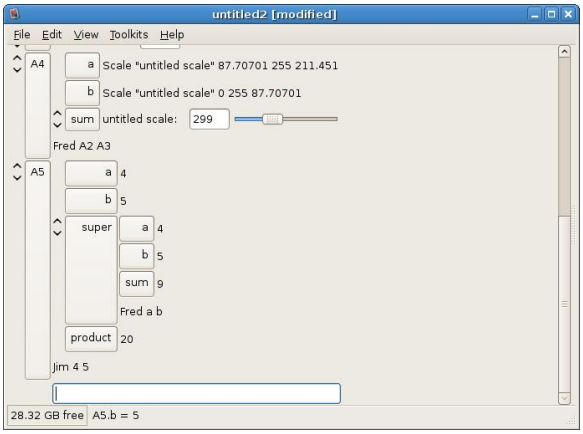

Figure 2.14: Browsing `Jim`

update again, right click on the `sum` button and select `Reset` from the pop up menu. Now drag one of the input sliders again, and `sum` will start updating once more.

Classes can inherit from other classes. Go back to the program window, click on `File` / `New` / `Tool` to clear the edit window, and type:

```
// multiply two things

Jim a b = class Fred a b {
  product = a * b;
}
```

This defines a class called `Jim` which inherits from `Fred`. Click `File` / `Process`, then back in the main window, type `Jim 4 5` into the bottom of the column. Click down once to expose the members (just `product`), click again to expose the parameters as well (`a` and `b`), and click a third time to expose the superclass member (which should be an instance of `Fred`). You can also open up the `super` member and see inside the `Fred` that this `Jim` is using as its superclass. See Figure 2.14.

`nip2` has about 20 different graphical classes like `Scale`. Whenever a row takes a new value, `nip2` checks to see if that value is an instance of one of these special classes, and if it is, it will add a graphical element to the row display which represents that class's value. It builds the graphical part by looking inside the class for certain members (for example, the scale graphic looks for members called `from`, `to` and

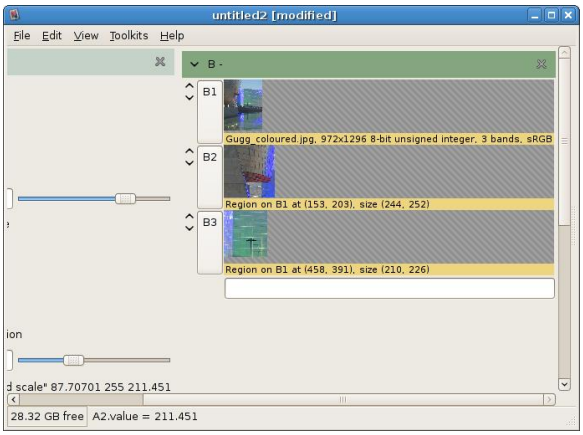

Figure 2.15: Two more regions

`value`). When you change the graphic (maybe by dragging the scale), `nip2` rebuilds the class by looking inside for a edit member (eg. `Scale_edit`) or if that's not defined, a constructor member (eg. `Scale`).

You can make your own graphic widgets by subclassing `nip2`'s built-in ones. By selectively overriding default constructors and adding edit members, you can control how your new widget will behave in expressions, and how it will behave if it's edited graphically.

Make a new column, load up an image (use `File` / `Open`), open an image viewer (double-click on the thumbnail), drag out two regions on it (hold down Ctrl and the left mouse button and drag down and right). Your main window should look like Figure 2.15.

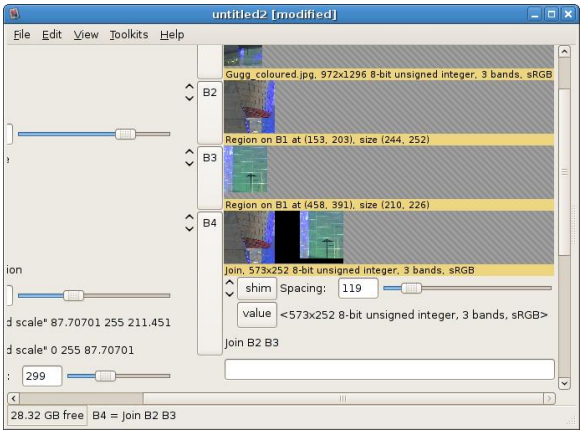

Figure 2.16: Joining two images with `Join`

`im_insert` is a VIPS operation that puts one image inside another at an (x, y) position. VIPS operations work on VIPS images. The `value` member of an `Image` or `Region` is the VIPS image that underlies the `nip2` row.

You can use `im_insert` to make a thing to join two images together. Back in the program window, click on `File` / `New` / `Tool` and enter:

```
// join two images left-right

Join a b = class Image value {
  shim = Scale "Spacing" 0 1000 0;
  value = im_insert a.value b.value
    (a.width + shim.value) 0;
}
```

Click `File` / `Process`. This defines a class `Join` which subclasses the `Image` graphic.

Now select your two regions (click on the first one, shift-click on the second) and click on `Toolkits` / `untitled` / `Join`. A new `Join` row will appear. Open it up and drag the slider to set the spacing between the two joined images. Go back to the image viewer for the image file you loaded and try dragging one of the regions. Figure 2.16 shows this class in action. The thing in `Toolkits` / `Image` / `Join` / `Left to Right` is just a fancier version of this.

You can change how the graphic widgets behave by subclassing them. Try:

```
Scale_int c f t v = class
  scope.Scale c f t ((int) v) {
  Scale = Scale_int;
}
```

This defines a new scale class called `Scale_int` which can only take integer values. The `Scale = Scale_int;` line is `Scale_int` overriding `Scale`'s constructor, so that a `Scale_int` stays a `Scale_int` when you drag. Because there's a local called `Scale`, `Scale_int` needs to use `scope.Scale` to refer to the superclass.

Here's a version of `Mark` which can only be dragged in a circle. You pass it an image to display on, an xy centre position, a radius and a start angle.

```
Mark_circle image x y r a = class
  scope.Mark image _x' _y' {
  // get rect cods for our point
  _pos = (x, y) + rectangular (r, a);
  _x' = re _pos;
  _y' = im _pos;

  Mark i l t
    = this.Mark_circle i x y r a'
  {
    // vector from centre of
    // circle to new position
    u = (l, t) - (x, y);

    // angle of vector
    a' = im (polar u);
  }
}
```

Chapter 3

Assembling infrared mosaics

VIPS has a package of functions designed to help join many small images together to make a single large image. They were originally designed to assemble infrared reflectograms but are general enough to be useful for other sorts of image as well, such as X-rays.

This chapter first introduces the mechanics of infrared imaging then explains how to use `nip2` to assemble the images you grab. Finally, it suggests some printing techniques.

3.1 Infrared imaging

Most museums use tube cameras (usually called Vidicons) for infrared imaging. Although they are relatively cheap they are not very stable and they suffer from (sometimes quite severe) geometric distortions. More modern solid-state cameras are still expensive but are becoming more widely used because of their greater stability. This guide assumes you are using a tube camera but almost all of it applies to solid-state cameras as well.

Whatever your camera there are three main sources of error which have to be addressed in order to be able to make successful mosaics:

1. Tube cameras suffer very badly from distortions in the image, usually either 'pin-cushioning' or 'barrelling'. These distortions result in alignment errors when sub-images are joined together.

2. The sensitivity of the tube varies across its surface, causing some parts of each sub-image to be brighter than others. This is made worse by unavoidable variations in illumination. When a lot of such images are joined together the result is a 'brick wall' effect.

3. The sensitivity of the tube also varies between sub-images, partly as the overall lightness in the field of view changes, and also because the electronics in the camera change as the camera heats up. This leads to a patchy, unbalanced mosaic.

The first two problems will be different in each Vidicon and will change each time a tube is replaced. All three problems need to be addressed to create successful infrared reflectogram mosaics.

3.1.1 Setting up your system

Mechanical set-up

It is vital that the optical axis of the Vidicon is at right-angles to the picture plane and that, whether it is the Vidicon or the painting that moves during image capture, it remains perpendicular. Obviously, with a seriously warped panel, this may not be possible. The lighting should be carefully adjusted so that the area of interest is lit as evenly as possible. If you can, arrange for the lights to remain stationary with respect to the camera.

An easy way to test camera alignment is to image a piece of graph paper, move the camera (either left-right or up-down) by 90% of the field of view, and see how features in the overlap area move. First rotate the centre around the optical axis to get the centre line of the images lined up. Next check the corners and adjust camera pitch and yaw.

Video set-up

On Linux, you can capture video directly into `nip2` provided that your capture card is compatible with v4l. You may need to adjust the `nip2` video settings. These settings can be found under the headings `Video`

for linux and General video capture in the Preferences window, which can be accessed by selecting Edit / Preferences from the main nip2 window. The default settings, see Figure 3.1 on page 13, are for the Hauppauge PCI capture card and should be changed as required.

Once you have set up your card, select Tasks / Capture / Capture Video Frame to create a new video object, which will appear as a still image in your current selected column. The captured image can be updated by opening the image in a viewing window and then pressing Ctrl-C (a shortcut for File / Recalculate Image).

You can create more than one video object: this can help you to get the overlaps right if you have two open at once (one showing the previous grab) when you are moving the camera around.

Setting the crop and aspect

While it is possible to correct geometric distortions after the image is captured, it is difficult to do the necessary modelling accurately and reliably. Instead, we suggest the grabbed images should simply be cropped, since the most severe distortion affects the perimeter of each video image.

To determine the area to crop, set up the Vidicon as it would be set to image a painting and capture an image of a rectangular grid — a piece of graph paper works well as a target. Before capturing the grid, check the current video crop settings in the Preferences window and ensure that the crop is set at maximum: left 0, top 0, width 768, height 576 (these are the dimensions for a PAL signal, they may be different on your system). Also set the Aspect ratio line to 1.

Create a new video object and look for the largest rectangle with little distortion (no more than a few pixels). If you create a region on the video image (by holding down the Ctrl key and dragging down and right with the left mouse button, see §**??** on page **??**) you can compare the straight edges of the region against the distorted lines of the grid. You can then expand and contract the region until you decide on the optimum area. The settings you need for the crop box can be taken from the values contained within the region object, which can be viewed by left-clicking several times on the down arrow to the left of the region object name.

Finally, you can set an aspect ratio: nip2 will automatically stretch video frames vertically by this factor. You can measure the aspect ratio of your capture card by taking a picture of something you know to be square and dividing the width in pixels by the height in pixels.

Move back to the Preferences window and enter the crop values in the General video capture section. Next time you create a new video object, you should find that it is cropped to the appropriate area. The settings you enter in the Preferences / window will be saved and automatically loaded again next time you start nip2. See appendix A.

Choosing the usable area of the image is a matter of compromise — the smaller the area, the more images are required to build a mosaic of a particular painting. If the area chosen is too large then the amount of distortion can cause serious errors in the final mosaic.

It's possible to use the VIPS rubber sheet plug in to detect and correct geometric distortion in your images automatically. This lets you use the full area of the sensor. It is a bit fiddly, but see the rubber sheet documentation if you are determined.

3.1.2 Capturing the data

Setting the gain and offset

Once everything is correctly set up, position the painting in front of the camera and experiment with the gain and offset settings on the Vidicon to achieve the optimum infrared image on the computer screen. Note that the appearance of the image on the computer will normally be different to its appearance on the video monitor connected directly to the camera.

Repeat this process at different points across the whole area to be captured. Although it is not always possible, the aim is to find a setting that does not need altering much during the capture of the data. This seems to lead to more successful mosaics.

Grey card correction

To counteract the problem of uneven sensitivity across the target area of the tube, it is necessary to capture an image of a piece of grey card. This grey card image is then used to correct all subsequent images. The card should be a flat, even grey, of around 50% reflectance. The Vidicon and lighting positions with respect to the painting should not be changed between the imaging of the grey card and the capturing of the mosaic. A grey card can be captured at any time, but it is good practice to start with one — you may forget later.

Figure 3.1: Recommended video preference settings

If your mosaic will take several hours to capture, you may wish to grab extra grey cards, since the sensitivity of the tube can change as it warms up. Be sure to note which data images correspond to which grey cards!

In association with the first grey card, it is a good idea to grab an image of your grid to record the scale at which the data images are being made.

Image capture

Open a video window, `Tasks` / `Capture` / `Capture Video Frame`, and when it shows the desired area, save the file. The choice of file name is a question of personal preference. We find it helpful to use a format that indicates where in the mosaic the data comes from — for example, `dat3.5.v` for the fifth image in row three.

It is helpful to be able to see the previous image to ensure that there is sufficient overlap. To achieve this, a second video window can be opened and placed alongside and the images grabbed into alternate windows.

Capture tips

- Make a new directory for each painting, and keep all of the image files for that painting (including a grey card and a grid) in that directory.
- VIPS can join up images in any layout, but you will get much less confused when you assemble your images if you stick to a regular grid. This can be difficult — a good compromise is to keep one axis fixed and grab in rows (or columns).
- You can help to reduce mosaicing errors later if you keep your rows (or columns) as short as possible. So if the painting is in landscape format, grab in columns (or turn the painting on its side and grab in rows); if the painting is portrait, grab in rows (or turn the painting on its side and grab in columns).
- The semi-automatic mosaic functions (see §3.2 on page 14) need a minimum overlap between the sub-images they join of around 20 pixels. For safety, you should aim for a larger overlap than this: we recommend an overlap of 60 pixels (around 20mm, usually).
- If the overlap area is featureless, it is worth identifying a good tie-point and ensuring it is visible in both images, even if this means increasing the overlap for one of the joins.

3.1.3 Correcting illumination

Before the mosaic can be assembled, the data images need to be corrected for non-uniformity of illumination using the grey card image. This function can be performed within `nip2` or directly using a predefined VIPS command-line tool.

3.1.4 Correcting with the command-line tool

First, close `nip2` and open a command line window, (xterm, mingw, etc). Move to the directory containing your image files with `cd`. For example, if you have made a directory called `raphael` inside your home directory, type:

```
prompt% cd raphael
```

The program you need to use is called `light_correct`. You need to give it the name of the grey-card image and the names of all of the image files you want it to correct with that gray card. Suppose you have saved your gray card image as `grey.v`, and your painting image files are called `dat1.1.v` and `dat1.2.v`. You would then enter:

```
prompt% light_correct grey.v dat1.1.v dat1.2.v
```

The program will run and print messages explaining its progress. It creates a new set of corrected image files, with the same names as before, but prefixed with ic_. In this example, it would create two new images files called ic_dat1.1.v and ic_dat1.2.v.

If there are a lot of image files to correct this could mean a lot of typing. Fortunately, you can use wildcard characters to abbreviate lists of file names. The example above can be abbreviated to:

```
prompt% light_correct grey.v dat*.v
```

The dat*.v means 'any filename which starts dat and ends with .v'.

You can use this technique to correct different parts of your mosaic with different grey cards. If you have a file called grey1.v for the first row in your mosaic, and a file called grey2.v for the second, you could do the correction in two parts:

```
prompt% light_correct grey1.v dat1.*.v \\
prompt% light_correct grey2.v dat2.*.v
```

3.1.5 Correcting within nip2

The function within nip2 used to preform this correction is Tasks / Capture / Flatfield. A set of images can be corrected at the same time by joining them together in a Group. A group can be produced by selecting all of the required images and then using the Edit / Group command.

Load all of your images into nip2, and group all the image except the grey image. Select your grey image and then your new group and then run the Tasks / Capture / Flatfield function. This will produce you a group of corrected images. Right-click on this new group and select Save As from the menu. In the save window type in a name, for example fred_01.v and then hit the save button. All of the images in your group will then be saved as fred_01.v, fred_02.v, fred_03.v ... fred_n.v.

If you want to keep row numbers in your file names, (see §3.1.2 on page 13), you will need to correct your images one row at a time, saving each row as fred01_01.v, fred02_01.v, etc.

3.2 Assembling the mosaic

The tutorial has a section on mosaic assembly with nip2: see §2.2 on page 6.

Mosaic assembly is normally painless. There are a few factors you should bear in mind when you are deciding how to assemble an image (particularly a large image):

- You can open up a mosaic join and change a few options, such as the blend width. If you want to change the defaults for a whole workspace, change the Mosaic defaults options in your Preferences.
- If two images just won't join correctly, try using Tasks / Mosaic / One Point / Manual Left to Right instead. These functions operate in the same way as the usual mosaic functions, but do not do a search. This is useful when the overlap is too small for the search to work correctly, or when the overlap area is very smooth and contains too few features for the search to find the exact overlap for you.
- nip2 does not do sub-pixel interpolation. As a result, each join will on average cause a positioning error of about 0.5 pixels, even if your input images contain no geometric distortion. If there are distortions in your input images (there usually are distortions in infrared images), then errors can be as much as 1–2 pixels per join. These errors accumulate as the number of images you join becomes larger.

  If you join a strip of 10 images together with Tasks / Mosaic / One Point / Left to Right, on average you can expect a total error of about 5 pixels. If you join two strips like this together top-bottom, you can therefore expect a mismatch of about 2.5 pixels at each end of the join.

  You can minimise the effect of these errors if you assemble your images differently. Suppose you have a 10 by 10 mosaic to build. Instead of making and joining 10 strips of 10 images each, make four 5 by 5 sub-mosaics (one for each quadrant) and then join these four quadrants together. The errors will now be more evenly spread over the image and therefore will be less visible.
- Some operating systems limit the number of files a program can have open at once: this in turn limits the size of the mosaics you can assemble in one go. If you are having problems putting together very large mosaics, try building your image in sections

(top, middle and bottom, perhaps), and later load up and join these larger pieces.

3.3 Balancing the mosaic

Like assembly, mosaic balancing is normally automatic and painless. You may sometimes have problems if the image is very large, or needs dramatic corrections:

- Each VIPS image file has an associated history, recording the operations on that image since it was loaded from a file. You can view an image's history by clicking on View / Image header in an image view window.

  The automatic balancer uses the history to work out how you built your mosaic. The balancer knows about left-right and top-bottom joins, but nothing else! If the history has other stuff recorded in there, you'll see unhelpful error messages like unable to open tmp/xxx.v, or more than one root.

  If you need to perform corrections to any of your sub-images, do them, save the image, load it again, and then build the mosaic. This will make sure the history of the image you are trying to balance only contains mosaic operations.

- On some systems the balancer can run out of memory or out of file descriptors on very large mosaics. If your mosaic is made up of more than a few hundred images, and you are having balancing problems you may have hit one of these limits.

  The solution (as with mosaic assembly) is to assemble and balance your mosaic in smaller pieces.

- If your grey-card correction is not accurate, you will find that the balancer will magnify any problems you have.

  Suppose your lighting and camera set-up always produces images which are brighter on the right than the left, and suppose, due to some problem with your grey-card correction, this effect is not completely removed. You will find that when you balance a mosaic, the small differences between left and right edges of your sub-images will have been smoothed out, but they will have caused a large difference in brightness between the extreme left edge of your final image and the extreme right.

  nip2 includes several functions which can help to fix this problem, the most commonly used being: Tasks / Mosaic / Tilt Brightness / Left to Right and Tasks / Mosaic / Tilt Brightness / Top to Bottom.

3.4 Other nip2 features useful for reflectograms

You can use nip2's general image processing facilities to play around with reflectogram mosaics. You can make false-colour images of your reflectograms, blend them with visible images or X-rays, search them for edges, and so on see the registering and overlays_and_blending examples for ideas.

There are also some first order mosaic functions: Tasks / Mosaic / Two Points / Left to Right and Tasks / Mosaic / Two Points / Top to Bottom. These functions automatically rotate and scale the right-hand image in a join. They are useful for assembling X-ray mosaics, and for fixing very difficult joins in reflectogram images. These functions work the same way as the One Point functions except that you will need to define two tie-points on each image.

You can mosaic images of any numeric type: 16-bit integer images are handy for mosaicing X-ray images, for example.

3.5 Printing

Once you have assembled a good reflectogram, you will want to print it, or to use it in other computer programs. The best way to do this is to save the final image in TIFF or JPEG format, and then load it into the new application — see §4.2 on page 18.

There are a couple of points to bear in mind: first, like any image, reflectograms look best on paper if you sharpen them up a little first. Click on Filter / Convolution / Custom Convolution, right click on the matrix button, select Replace from file. Double Click on the second or lower data directory listed in the left hand column to enter nip2's main data directory. Change the Image type select / option to All FIles (*) and then select and load rachel.con. This will usually produce an approprioatly sharpened reflectogram.

Secondly, you will need to try several prints with different contrasts and brightnesses to get a good match between the paper and the screen, try `Image` / `Levels` / `Linear`. You may even want to fiddle with the gamma, try `Image` / `Levels` / `Power`.

Finally, you may not need a full resolution image. For almost all printers there's no point going over about 300 dpi (dots per inch), or about 3000 by 2000 pixels for an A4 page. To reduce the size of an image, use one of the functions listed under `Resize` / `Transform` / `Resize`.

Chapter 4

Reference

This chapter is supposed to be a user-interface reference. Chapter 5 on page 25 describes the items in the `Toolkits` menu and Chapter 6 on page 31 is the programming language reference. Chapter 2 on page 3 has a tutorial-style introduction.

4.1 Image view window

Figure 2.2 on page 3 shows `nip2`'s image view window with all the toolbars turned on.

If you press `i` (or `+`) with the keyboard focus on the image you will zoom in on the pixel your mouse pointer is over. Press `o` (or `-`) to zoom out again, or press the number keys `1`, `2`, `4` and `8` to jump straight to a particular magnification. If you hold down the Ctrl key while pressing these numbers, `nip2` will zoom out by that amount. If you press `0` (the number zero), then `nip2` will pick a magnification or reduction which fits the image to the size of the window.

When the image is too large for the window, you can use the scroll bars to move about the image. With the keyboard focus on the image the cursor keys left, right, up and down move a few pixels in each direction; hold down Shift as well to move a screenful at a time; hold down Ctrl as well to jump to the extreme edges of the image.

You can also drag with the middle mouse button to pan around the image. Use the mousewheel to pan up and down, hold down Shift and the mousewheel to pan left and right. Use Ctrl and the mousewheel to zoom in and out.

Use the `View` menu to add extra elements to the window. You can turn the status bar on and off, and you can add a display control bar, a paintbox and a set of rulers to the window.

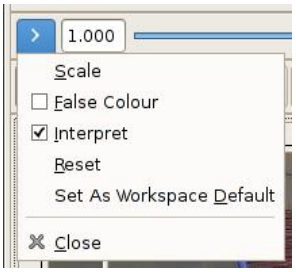

Figure 4.2: The display control bar menu

If you select `View` / `Toolbar` / `Display Control`, `nip2` will add a bar to the top of the window which you can use to change the contrast and brightness of the image you are viewing. The left-hand slider and text box set the gain for the image: each pixel is multiplied by this amount before display. The right-hand slider and text box set the offset: each pixel has this value added to it before display. This is useful for boosting the brightness in dark areas of images.

If you click the left mouse button on the arrow to the left of the display control bar, `nip2` pops up a menu of useful display functions — see Figure 4.2.

`Scale` searches the area of the image you are viewing for the darkest and brightest points and chooses settings for the gain and offset sliders which will stretch the image to use the full range of your screen. `False colour` tries to make small differences in brightness more visible by colour-coding them.

If `Interpret` is turned on (it is by default), then `nip2` will look at the `Type` field in the image header, and use that as a hint when transforming the image to a viewable form for you. This is usually the behaviour you want. `Reset` moves the sliders back to the de-

Figure 4.1: The display control bar

fault positions of 1.0 and 0.0. `Set As Workspace Default` makes the current display bar settings the default for all new image windows in this workspace. Finally, `Hide` removes this display control bar.

If you select `View` / `Toolbar` / `Rulers`, `nip2` will add rulers to the edges of the window which you can use to measure numbers of pixels. If you left-drag from the ruler, you can create a guide. Guides are useful for lining up other things in the view window, and also affect paint box actions. A right-button menu on the rulers lets you use a mm scale rather than a pixel scale, and controls whether the `Xoffset` and `Yoffset` header fields are used.

The `File` menu contains two useful items: select `Replace Image` to change the image which is being displayed in the window (you can also drag and drop new images in). Select `Save Image` to save the image you are viewing to a file. See §4.2 for details on `nip2`'s load and save dialogs. The `New` menu is a no-mouse route for creating regions, points, guides and arrows.

If you select `View` / `Toolbar` / `Paint`, `nip2` adds a paint bar to the top of the window. You can use the paint bar to do simple edits to the image being displayed. See Figure 4.3 on page 19.

While the paint bar is very limited, it does have two useful features. First, it can paint with any pixel value, even complex. For example you can take the fourier transform of an image and paint out the peaks. Secondly, it doesn't operate on a memory copy of an image, it operates directly on the file on disc. This means that you can paint on images of any size, but it does make the paint bar a bit dangerous.

Normally paint actions are live, that is, every time you paint something all the objects which depend on the thing you painted will recalculate. This can sometimes cause annoying delays: there's a preferences option to turn off automatic recalculations for the paint bar.

The `Undo` and `Redo` buttons move forward and back though paint actions. The `Clear` button wipes the undo/redo history (useful if memory is getting low). There's an option in the preferences workspace which controls the number of undo steps `nip2` tracks.

You can mark regions on images by holding down Ctrl and dragging down and right with the left mouse button. You can move the region about by dragging on the label with the left mouse button; you can resize it by dragging with the left mouse button in the border; you can get a useful context menu by right-clicking on the label; and you can pop up a box which will let you edit the region numerically by double-left-clicking on the label.

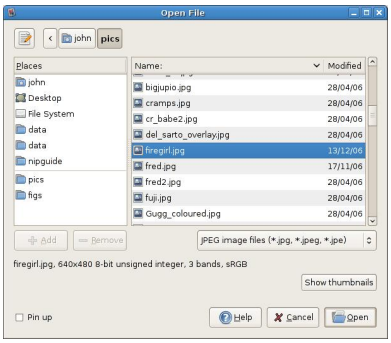

Figure 4.4: Open dialog

If you drag up and left, you will make an *arrow*. If you hold down Ctrl and just click the left mouse button, you will make a *point*. If you drag from a horizontal or vertical ruler, you'll make a *guide*. Guides are useful for lining up other things in the view window.

Use `File` / `New` to make regions, points, arrows and guides without the mouse.

4.2 File select dialogs

On most platforms you can drag files from your file manager directly to `nip2`'s main window. Alternatively, if you select `File` / `Open` in the main window, `nip2` will pop up a file dialog, see Figure 4.4. The open dialog has the following extra features:

Pin up button Normally the file dialog closes after you have opened something. If this item is checked, the dialog will stay up instead — this is useful if you want to load or save a series of objects.

Figure 4.3: The paint bar

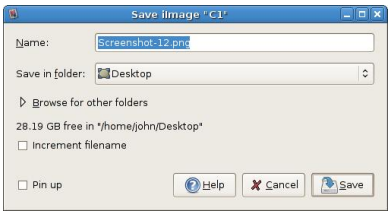

Figure 4.5: Save dialog

Image type select Use this menu to select the type of file you want `nip2` to display. There's a preference option to set the default image format. The VIPS file format is fast and accurate, but sadly not very widely supported (joke). You can also load and save images in TIFF, JPEG, PNG, HDR, CSV and PBM/PGM/PPM formats. You can usually load in many more formats, it depends how your `nip2` has been configured.

Show thumbnails Pressing this button pops up a window which shows thumbnail-sized images for all the matching files in the current directory.

The save dialog is a little different, see Figure 4.5.

If pin up and increment are both selected, then after a save `nip2` will attempt to add one to the selected file name. For example, if you save a file called `fred001.v`, after the save `nip2` will put the name `fred002.v` into the selected file name box. Again, this is useful if you want to save a series of images.

4.3 Image processing window

Figure 4.6 show's `nip2`'s main image processing window. The centre area is the workspace, the left-hand area is a pane you can reveal to write custom definitions for this workspace (see `View / Workspace Definitions`), and the right-hand pane is the toolkit browser (see `View / Toolkit Browser`).

Drag with the middle mouse button to scroll the workspace window. Drop a file on to the workspace

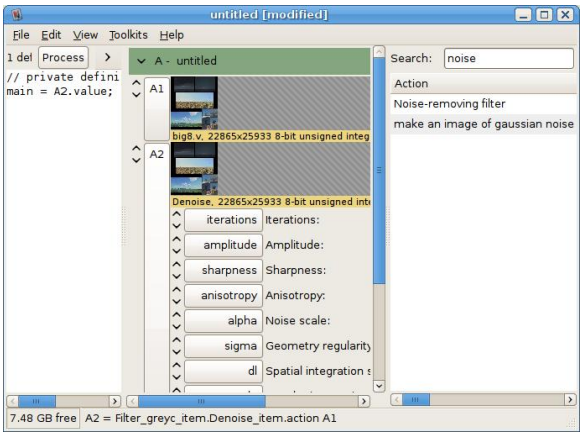

Figure 4.6: `nip2`'s main image processing window

background (from your file manager) to load that file. If you right-click on the workspace background, a useful menu will appear.

Workspace This area displays the objects (images, numbers and so on) which are currently loaded into `nip2`. The workspace is divided into columns of objects which each behave rather like windows: they can be moved around, folded away, loaded, saved and deleted.

Current column One column is the current column. This is the column to which all new objects are added. Single-left-clicking on the title bar of a column makes that the current column. See §4.3.1 on page 20.

File, Edit, View Use the `File` menu to create or save workspaces, to open workspaces or load other objects into this workspace, to merge workspaces and to search for workspace backups. Use the `Edit` menu to select, group, delete and duplicate sets of objects. Use `View` to show and hide elements of the main window, and to set the object view mode.

Toolkits This menu contains all of the image processing functions which are currently loaded into

nip2. They are generally grouped by object type: all of the operations on matricies are under Toolkits / Matrix, for example.

If you select one of these image processing operations, nip2 will apply that operation to the bottom few items in the current column (however many are necessary — two items for Math / Arithmetic / Add, for example), or alternatively, if you have selected some objects explicitly, it will try to apply the operation to the selected objects. See §4.3.3 on page 21. As you move the mouse pointer over menu items nip2 tries to display some helpful information about the operation, including the number and type of arguments the operation expects.

Toolkit Browser This side panel shows all the image processing operations again, but this time as a large flat list you can easily browse. Type into the search box at the top to filter operations by keyword. Doubleclick on an item to activate it.

Workspace Definitions This side pane shows private definitions for this workspace. Programs you write here are loaded and saved with this workspace. See the Programming chapter for details on nip2's programming language.

Free space This displays the amount of disc space you have left in your temporary file area. See §A on page 49 if you want to change the directory nip2 uses to store temporary files.

If you left-click on the label, it changes to display the space nip2 has free internally for performing calculations. You can change this limit in the Preferences workspace. Click again to switch back to disc free.

If you have objects selected, this area changes to show the names of the selected objects.

Status bar As you move the mouse pointer about the window, this bar tries to display useful information about the thing you are pointing at.

4.3.1 Columns

Columns are split into a number of areas:

Column name Each column has a name. You can pick any name you like when you make a new column with File / New / Column. There's no way to rename a column, unfortunately. Objects in the column are named using the column name, plus a number.

Column title bar Drag with the left mouse button held down on the column title bar to move the column around the workspace. Double-left-click on the title bar to change the comment attached to the column. Hold down the right mouse button on the column title bar to pop up a useful menu.

The items in the menu let you edit the caption, select all the objects in the column, make a new column which is a copy of this column, save the column to a file, convert the column into a menu item (see §4.3.6 on page 22) and remove the whole column.

Column fold button Left-clicking on the fold button folds the column away. Use this to hide columns which you still need, but which you are not interested in just now.

Expression entry You can perform calculations by typing expressions directly into this box. For example, try entering the following expressions, and pressing Return:

```
2 + 2
A1 + 120
"My cat likes\nlasagne"
fred = 12
```

The last example shows custom button name creation. Normaly nip2 will pick a name for you, but you can chose your own.

4.3.2 Rows

A column holds a number of rows. Each row comes in four main parts, not all of which are visible for all row values. Rows which represent classes have a pair or up/down arrows to the left of the row name button which you can use to control which parts of the row are visible.

Row name button Each row has a name. The name is normally formed from the name of the current column, plus a number.

If you double-left-click on the row name button, nip2 will pop up a viewer or dialog box for the

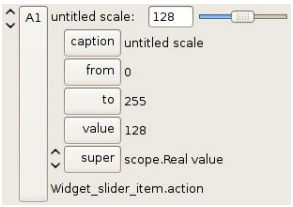

Figure 4.7: Components of a workspace row

value of the row. If you left-click, nip2 will select that row and deselect all other rows. If you click on an empty space in the workspace, it will deselect all rows. If you Ctrl-left-click, nip2 will toggle selection of that row. If you select one row and then Shift-left-click on another row in the same column it will select the second row and all the rows in between. If you drag with the left button, you can change the order of rows in a column. Hold down the right mouse button for a useful menu. If you let the mouse linger over a button, a useful tooltip will appear.

Graphic If the row's value is a class, and if the class is an instance of one of nip2's graphic classes, then nip2 will draw a graphic representation of the row's value. See §6.12.2 on page 44 for a more detailed explanation.

Members If the row has a class for a value, then nip2 will draw a sub-column listing the class members. Subcolumn members are in turn rows themselves.

Text Finally, the text part normally shows a text representation of the row's value. If you left-click on the value, it changes to show the formula which generated that value. You can edit the formula and press Return to change it.

Alternatively, selecting View / Show Formula toggles between displaying values for objects and displaying the formula.

Object name colours

nip2 changes the background colour of the row name button to show the state of the row. If background colours are not visible (perhaps your theme turns them off), try turning on the Display LEDs in workspace option in Preferences.

Green means the row is selected (click on the background to unselect), red indicates an error (right-click on the row button ans select Recalculate to see the full text of the error), brown indicates that the row value is out of date and needs recalculating and the various blues indicate parent and child relationships.

4.3.3 Applying operations to objects

There are three ways you can apply image processing operations to objects in your workspace:

1. Select the object you want to apply the operation to by single-left-clicking on the object name. When you single-click, the object name will change colour to show that it is selected, and nip2 will display the name of the selected object at the left end of the status bar (this is useful if the selected object is scrolled off the edge of the window).

   You can select additional objects with Ctrl-left-click and Shift-left-click. This is necessary if you want to use an image processing operation that takes more than one argument.

   Once you have selected the rows (sometimes you need to select them in a certain order), click on the processing operation you want from the Toolkits menu.

2. If there are no objects selected when you click on an image processing operation, nip2 uses the bottom few items (as many as are needed by the operation) in the current column.

3. You can also type your formula directly into the expresion entry line at the bottom of the selected column. Chapter 6 on page 31 describes the syntax in detail, but it's approximately C.

4.3.4 Batch processing

If you select a number of rows and then click Edit / Group, nip2 will group the rows together. Now if you select the group and click on an item in the Toolkits menu, nip2 will apply that operation to every item in the group. You can group groups, and you can mix grouped and non-grouped rows freely.

If you save a group, nip2 will write each item in the group to a separate file, incrementing the filename each time.

4.3.5 Error handling

If an object in your workspace has an error (for example, if you are trying to join two images of different types), then the object name button will turn red to show that this object contains an error and the tooltip for the button will show the error message.

4.3.6 Making menu items out of columns

If you make a column that does something useful, you can make it into a menu item by following these steps:

1. Make your column look nice. Drag with the left mouse button on the object name buttons to reorder items in the column, and add comments to explain what are the input fields and what are the output. Double-click on the column title bar to add a helpful title to the column.

   Add a comment by typing your text (enclosed in double quotes) into the line at the bottom of the column. Left-drag the row to the right place.

2. Select `Make Column Into Menu Item` from the column title-bar menu, see §4.3.1 on page 20.

   This will open up a new dialog box which you can use to set a name for your new menu item and the name of the top level menu the item should be added to.

3. That's it. You'll be prompted to save your new toolkit when you try to quit `nip2`. We recommend you just say `OK` to the suggested location for the file. Edit your menus with the programming window, see §4.4.

4.4 The programming window

To pop up the programming window, click on `Toolkits` / `Edit Toolkits` in `nip2`'s main image processing window. The window shown in Figure 2.11 on page 8 should appear.

Each of the things down the left of the program window is a toolkit. Each toolkit is a text file containing a set of definitions in `nip2`'s programming language. See Chapter 6 on page 31 for details on the language.

If you open a toolkit, `nip2` shows all of the definitions in that file. If you click on one of these `nip2` shows the source for that definition in the main part of the program window. After editing a definition, click on `File` / `Process` to make `nip2` read what you typed, compile it, and update itself.

Click on `File` / `New` / `Tool` to add a new definition to a toolkit, click on `File` / `New` / `Toolkit` to make a completely new toolkit. You can also right-click on tools and toolkits to get a context menu, and you can left-drag tools to move them around within a toolkit or between toolkits.

Some toolkits are loaded from files when `nip2` starts up, others are built from the VIPS operation database (for example, `_arithmetic`), and one (called `_builtin`) contains the functions that are built into `nip2`. If you select a tool and then click on `Help` / `Help on Tool`, `nip2` will try to display the relevant section from the VIPS manual in your web browser. Currently, this works only for things in the VIPS operation database: try `_arithmetic` / `im_add`, for example. There's a section in the `Preferences` workspace to control which web browser `nip2` uses and how it asks for a page.

Toolkits and tools whose names begin with an underscore character are not displayed in the main `Toolkits` menu. The idea is that they represent little utility functions, rather than stuff a user might be interested in. See §6.12.1 on page 43 for more information on how tools and toolkits are displayed.

You can have several programming windows open at the same time (often useful, if confusing). The `Edit` menu lets you search for patterns across all definitions. The `Jump To Definition` item jumps to the definition of a symbol.

The `Debug` menu has items which open a trace window (use this to track the actions taken by `nip2`'s reduction engine) and which report on unresolved symbols and list all current errors.

4.5 Command-line interface

You can use `nip2` from the command-line as well as from the GUI. This can be handy for automation: you can build a workspace and then run it over a whole set of images, or use `nip2` as part of a larger system. We've make websites which use `nip2` as the back-end.

In command-line mode `nip2` runs without a GUI of any sort, it doesn't even need a window system to be installed on the machine. This makes it possible to use

it in a server or batch context.

These notes are for the Unix command-line, but they should work for Windows as well.

nip2 has three main modes of operation from the command-line:

nip2 *filename1 ...* Start nip2 in GUI mode, loading the command-line arguments as files. Filenames can be images, workspaces, matricies, toolkits, and so on.

nip2 -e *expression arg1 ...* Start in no-GUI mode, print the value of *expression*. The list argv is set to be ["filename","arg1",..].

nip2 -s *filename arg1 ...* Start in no-GUI mode, read in *filename* as a set of definitions, print the value of symbol main. The list argv is set to be ["filename","arg1",..].

You can use the -o option to send output somewhere other than the screen. If these modes don't do quite what you need, you can get finer control of how nip2 behaves with a set of other options: see the man page for details.

4.5.1 Using expression mode

The -e option is very easy to use. For example:

```
nip2 -e "2 + 2"
```

Prints 4 to stdout.

```
nip2 -e "99 + Image_file argv?1" -o result.png fred.jpg
```

Loads argv1 (fred.jpg), adds 99, writes the result to result.png.

```
nip2 -e "Matrix [[1,2],[4,5]] ** -1" -o poop.mat
```

Invert the 2x2 matrix and write the result to poop.mat.

If the result of the expression is a list, each item is printed on a new line. For example:

```
nip2 -e "[1..5]"
```

Will print the numbers 1 to 5, each on a new line.

If you have a list result and you are using -o to direct the output to a file, the filename will be incremented each time you write. For example:

```
nip2 -e "map (add (Image_file argv?1)) [10, 20, .., 50]" -o result1.png fred.jpg
```

Will load fred.jpg, add 10, 20, 30, 40 and 50, then save those images to result1.png to result5.png.

4.5.2 Using script mode

With the -s option you can use nip2 as a Unix script interpreter.

Create a file in your favourite text editor called brighten containing:

```
#!/usr/bin/nip2 -s

main
  = clip2fmt infile.format (infile * scale), a
  = error "usage: infile scale -o outfile"
{
  infile = Image_file argv?1;
  scale = parse_float argv?2;
}
```

The first line needs to be the path to nip2 on your system. Use which nip2 to find the path if you don't know it. Mark the file as executable with chmod +x brighten, then use it on one of your image files with:

```
brighten fred.jpg 1.5 -o bright_fred.png
```

See Chapter 6 on page 31 for details on the programming language. This program multiplies each input pixel by the constant, producing a floating point image, then then clips the result back to the same format as the original image (usually 8-bit unsigned).

nip2 takes a while (a few seconds) to start up, so this isn't going to be appropriate for small images or simple calculations. But for complex operations, or operations on large images, this mode can be very useful.

4.5.3 Using --set

The --set option (which can be abbreviated to -=) lets you make changes to a workspace after loading it. Suppose the workspace test.ws has a row called A1 with the value 12. Then entering:

```
nip2 test.ws --set Workspaces.test.A1=45
```

Will, as normal, start nip2 and load test.ws. But before the first recalculation, nip2 will change the value of A1 to be 45. You can use --set to create new symbols as well.

4.5.4 Other modes

A set of sub-options let you mix up other modes yourself. For example, it's common to want to run a workspace on many files.

Suppose the workspace `process.ws` loads an image in `A1`, performs some processing and produces a result image `A10`. If you run `nip2` with:

```
nip2 -bp \
  -= 'Workspaces.process.A1=Image_file "fred.jpg"' \
  -= main=Workspaces.process.A10 \
  -o fred.jpg process.ws
```

This will start `nip2` in batch (ie. no GUI) mode (the `-b` switch), load `process.ws`, change `A1` to load another file, set `main` to be the value of `A10` and save the value of `A10` to `fred.jpg` (the `-p` switch).

Chapter 5

Image processing menus

This chapter is runs quickly through the `Toolkits` menu. See Chapter 6 on page 31 if you want to understand how the menus are written (or want to add more of your own). Use the Toolkit Browser to find stuff.

Some things are common to almost all menu items:

Tooltips If you rest your mouse pointer over an item, you'll see a quick description of what the item does.

Grouping You can select several objects, click `Edit` / `Group`, and then when you click the item, it will operate on all the objects in the group.

Any type Almost all items will work on any object. You can add an image and a number, for example, find the colour difference between a number and an image, or transform a matrix from LAB to XYZ.

5.1 Colour

This menu groups operations on colorimetric images and patches of colour. A colour patch is three float numbers plus a tag saying how those number should be interpreted as colour (for example, as a colour in CIE LAB colourspace). You can drag and drop between colour patches, and into and from the inkwell in an image paint window. Double-left-click on a colour patch to open a colour select dialog.

`nip2` has 9 main types of colorimetric image, see Table 5.1 on page 26. All these types are D65 (that is, daylight) absolute colorimetric. When it displays an image, `nip2` uses the `Type` field in the image header as a hint on how to transform the numbers in the image into RGB for the display. The current `Type` is displayed at the end of the caption line below an image thumbnail.

The `Mono`, `GREY16` and `RGB16` types are not really calibrated themselves: they are usually whatever you get by loading an image from a file. You'll usually need an extra step, such as applying an embedded ICC profile, before you get accurate colour.

`New` Make a patch of colour, or pick a colour from a slice through CIELAB colourspace.

`Convert To Colour` Convert anything into a Colour object.

`Colourspace` Change the colourspace. The stored numbers change, but the visual appearance should stay the same.

`Tag As` Change the colourspace tag (the `Type` field in the image header). The stored numbers stay the same, but the visual appearance should change.

`Colour Temperature` Change the colour temperature. `Move Whitepoint` just adjusts the ratios of X and Z using the CIE standard illuminants.

`D65 to D50` and `D50 to D65` transform using either a 3x3 matrix which is numerically minimal in XYZ space with respect to the colours on a Macbeth Color Checker, or via Bradford cone space. The Bradford transform omits the power term.

The final two items go from XYZ to LAB and back, but with D50 normalisation rather than the default D65.

`ICC` Transform images (not patches of colour) device space to profile connection space (LAB float) and back.

Name	Format	Notes
Mono	One band 8 bit	Not calibrated
sRGB	Three band 8 bit	Screen device space for the sRGB standard
GREY16	One band 16 bit	Not calibrated
RGB16	Three band 16 bit	Not calibrated
Lab	Three band float	The 1976 version of the CIE perceptual colourspace
LabQ	Four band 8 bit	Like Lab, but represented as 10:11:11 bits
LabS	Three band 16 bit	Like Lab, but represented as 15:16:16 bits
LCh	Three band float	Lab, but with polar coordinates
XYZ	Three band float	The base CIE colourspace
Yxy	Three band float	Sometimes useful for colour meters
UCS	Three band float	Highly uniform space from the CMC(l:c) standard

Table 5.1: nip2 colourspaces

You need to be careful about colour temperature issues: all printers work with D50, and nip2 is all D65. Use the D65 to D50 interchange items in the Colour Temperature menu to swap back and forth.

All printers also work with relative colorimetry, and nip2 is generally absolute. Use Absolute to Relative to scale an absolute colorimetric image by a media white point.

Radiance nip2 can read and write images written by the Radiance family of programs (usually with the suffix .hdr), commonly used in HDR photrography.

Images in this format used a packed floating point layout for their pixels. Items in this menu pack and unpack pixels for you.

Difference Calculate various colour difference metrics. You can mix patches of colour and colour images.

Adjust Change colour in a colorimetric way. Recombination multiplies each pixel in an image through a matrix. Cast displaces the neutral axis in LAB space. HSB lets you adjust an image in LCh colourspace.

Similar Colour find pixels in an image with a similar colour to a patch of colour.

Measure Colour Chart This takes a trimmed image of a colour chart (a rectangular grid of coloured squares), measures the average pixel value in the centre 50% of each square, and returns a matrix of the measured values.

Use Make Synthetic Colour Chart to make a colour chart image from a matrix of measurements.

Plot ab Scatter draws a 2 dimensional histogram of the distribution of pixel colours in LAB colourspace.

5.2 Filter

This menu groups operations which filter images, or which are filters in the photoshop sense.

Convolution This menu has several standard convolution operations (blur, sharpen, edge detect, etc.), plus the option to convolve with a custom kernel.

Two menu items are slightly more complicated. Unsharp Mask transforms to CIE LAB colour space, then sharpens just the L band with a cored unsharp filter. The Tasks / Print menu has a version of this filter tuned for typical inkjet printers.

Custom Blur builds and applies a square or gaussian convolution kernel for you based on a radius setting.

Rank A preset median filter, and a custom rank filter that lets you specify window size and rank.

The Image Rank item does pixel-wise ranking of a set of images.

Morphology These menu items implement basic morphological operations. Images are zero for background and non-zero (usually 255) for object. Matricies are shown as 0, 1 and * for background, object and don't-care.

The `Threshold` item does a simple level threshold. Use the `Math` / `Relational` menu to construct more complex image binarisations. Use `Math` / `Boolean` to combine morphologies.

The first half of the menu lists simple erode and dilate operations, 4- and 8-way connected. The second half contains several useful compound filters.

See also `Histogram` / `Find Profile` for something that can search an image for object edges. And `MathStatistics` / `Edges` can count the number of edges across and down an image.

Fourier A selection of ideal, Gaussian and Butterworth Fourier space filters.

You can make other mask shapes yourself using the `Image` / `Make Patterns` menus, then apply them using `Math` / `Fourier`. You can also use the image paintbox to directly paint out peaks in a fourier-space image before transforming back to real space.

Enhance A selection of simple image enhancement filters. `Statistical Difference` passes a window over an image and tries to match the region statistics at each point to a target mean and deviation.

Spatial Correlation Place a small image at every possible position in a big image and calculate the correlation at each position. `Simple Difference` is the much faster unnormalised version.

GREYCstoration VIPS includes a copy of the CImg library and you can use two useful CImg operations from this menu: denoising and enlarging.

Tilt Brightness A selection of tools for adjusting the brightness of an image across it's surface. Useful for correcting lighting problems.

Blend Blend two objects together using either a third object to control the blend at each point, or a slider to set all points together. You can blend almost anything with anything.

One useful version is to use a text image (see `Image` / `Make Patterns` / `Text`) to blend between two colours (see `Colour` / `New`).

`Along Line` does a left/right or top/bottom fade between two images.

Overlay Make a colour overlay of two monochrome images. Useful with `Image` / `Transform` / for testing image superposition.

Colourize Use a colour image to tint a monochrome image. Useful in conjunction with `Image` / `Transform` / .

Browse Look at either the bits or the bands of an image.

Photographic Negative and friends A small selection of simple, faintly photoshop-style filters.

5.3 Histogram

This menu groups operations for finding and transforming image histograms. `nip2` represents histograms and lookup tables as images with `Type` set to `Histogram`. Histograms may have pixels in any format and any number of bands. You can only find histograms of unsigned 8- and 16-bit images.

New This makes a new ramp histogram. A set of sliders let you adjust the shape. Use `Map Histogram` to apply your ramp to an image.

`Build LUT from Scatter` makes a histogram from a matrix of (x, y) values.

`Tag Image as Histogram` marks an image as actually being a histogram after all.

`Tone Curve` builds a tone curve which you can later apply to an image.

Find A one dimensional histogram treats each band as an independent variable. An n-dimensional histogram treats each pixel as a vector of n elements, where n is the number of bands in the image.

Map Looks up each pixel in the input in the histogram and sends the found value to the output.

Equalise Find the global or locally histogram equalised image.

Cumulative Use this and friends to calculate a cumulative histogram (integrate), normalise a histogram and match two histograms.

Find Profile Searches from the edges of an image for the first non-zero pixel and returns a profile histogram.

Find Projections Sum columns and rows in an image.

Plot Slice Mark a guide on an image (drag from the image rulers, or click File / New / Guide) and click Plot Slice to make a histogram which is a horizontal or vertical slice through an image. Use Extract Arrow to extract the area around an arrow or guide. Use Plot Object to make a plot of any object.

5.4 Image

This menu groups operations which apply only to images.

New Makes a new image. Region on Image makes a new region, arrow, guide or mark on an image. It's usually easier to open a viewer on an image and Ctrl-drag.

Convert to Image Try to make an image out of anything.

Format Switch between the various precisions.

Header Try to change or examine the image header in various ways.

Cache This caches an image in RAM. Use this to save the results of a long computation.

Levels Various tools that change the levels in an image. Tone Curve is the only complex one: it lets you adjust the image levels with a set of sliders.

Transform Various tools that change the geometry of an image.

To use Rotate / Straighten, mark an arrow on an image (Ctrl-drag up and left in an image view window) along a near-horizontal or near-vertical edge. When you click on Rotate / Straighten, nip2 will rotate the image by the smallest amount that makes that edge exactly horizontal or vertical.

Linear Match takes two images and rotates and scales the second so that the images can be superimposed. Drag the tie-=points to mark common features. Use Filter / Overlay or Filter / Colourize to actually superimpose them.

Rubber Sheet is useful for fixing things like lens distortion. You give Find two images, a reference and a distorted version of that reference, and it automatically finds a transform which will map the distorted image back on to the reference image. Use Apply to apply the discovered transform to another image.

Band Extract/insert/delete image bands. Use To Dimension to change image bands into a horizontal or vertical dimension. Use To Bands to compress the horizontal or vertical dimension into bands (small images only!).

Crop Crops an image. It's often easier to drag out a region. This menu item is only really useful for cropping large groups of images.

Insert This takes two images and pastes the smaller into the centre of the larger. The two images have to have the same number of bands. If you open an image viewer on the large image, you'll see an area which you can drag around to set the exact insert point.

Select Draw elipses and polygons on an image. Useful for selecting defined areas.

Join Use to join two images together bandwise, left/right or up/down. Array joins a list of lists of images together into a single large image.

Tile Repeat an image horizontally and vertically to make a larger image, or chop an image into a set of tiles.

Patterns These items all make useful images for you, from checkerboards to gaussian masks. XY Image is the most useful: you can use it to build other patterns.

Test Images These items make a variety of useful testcharts for evaluating spatial response and colour.

5.5 Math

Basic maths operations on any combination of any objects. You can add a slider to a matrix, for example, then divide by an image. Hopefully most of these are obvious.

Arithmetic/Absolute Value Vector The absolute value item normally calculates mod of each band of an image separately. By contrast, Absolute Value Vector treats each pixel as a vector and calculates the modulus of that.

List These aren't really maths operations, but they're in here too.

5.6 Matrix

This menu groups operations which operate on matricies. nip2 has four ways of displaying a matrix, but they all behave in the same way under the skin. Almost all the items in the Math menu will work on matricies. Most of the matrix operations will also work on images.

New The first four items make matricies which display and edit in various ways useful for different applications. The final two make matricies which are pre-filled with useful numbers.

Convert to Matrix Try to make anything into a matrix.

Extract This group of items extracts various submatricies. You can also do this graphically: just drag-select an area in matrix.

Insert, Delete, ... Also work on images, which can be handy. A 45 degree rotate will only work for square matricies with odd-length sides.

Invert Simple matrix-only maths operations.

Plot Scatter This takes a two-column matrix where the columns are the X and Y positions of points and draws a scatter graph.

5.7 Object

This groups a few items which had no obvious home and which change the format of objects.

Duplicate Copy an object, stripping off any derived classes. For images, this really takes a copy of the underlying object (using im_copy()).

List to Group Changes lists (see Math / List) into Groups (see Edit / Group) and back. A list os an ordered collection of objects. A group is a list that nip2 will automatically iterate over.

Break Up Object This tries to take an object apart. So a multi-band image becomes a list of 1-band images. A matrix becomes a list of vectors, and so on. Assemble Object is the inverse.

5.8 Tasks

This menu repeats many items from other menus, but tries to group them by tasks they are useful for, rather than by function.

5.8.1 Capture

This menu groups operations which are useful in capturing images, or for the initial processing you might want to do to an image captured from another program.

CSV Import Import an image from a CSV file, with a few controls.

Interpret Analyze 7 Header Read the meta fields for volume layout and calibration from the Analyze header and reformat the image appropriately.

Capture Video Frame This menu item will currently only work on Linux machines with a compatible video4linux capture card. See §3.1.1 on page 11 for notes on how it works.

Smooth Use this to remove texture from images. It's handy in conjunction with Flatfield.

Flatfield Use this to correct homogeneity. Select an image of a piece of white (or mid-grey) card, then select the image to correct, then click Flatfield. Use Smooth to renmove texture from the white card if necessary.

You can select a single white and a group of images to correct a large set in one step

White Balance Use this to move the white point to make an area of the image you know to be white, white. Mark a region on an image, enclosing a patch you know to be white. Select the region and the image and click on `White Balance`.

Find Colour Calibration Use this to colour calibrate an image. Drag a region enclosing an image of a Macbeth Color Checker Chart and click `Find Colour Calibration`.

Apply Colour Calibration Use this to apply the transform calculated by the previous item to another image. Select the calibration object, select the RGB image you want calibrated, and click `Apply Colour Calibration`.

5.8.2 Mosaic

The items in this menu are discussed in appalling detail in Chapter 3 on page 11.

One Point Join two images left-right or top-bottom with a simple translation. Mark a point on each image to be joined (open image view window, Ctrl-left-click, drag to position), then click on the mosaic button. The operation performs elaborate tie-point adjustment, so your selection of a common feature does not have to be exact.

The `Manual` versions do not perform automatic tie-point correction and are useful when joing very difficult images.

Two Point Do a join, but allow the right-hand (or bottom) image to rotate and scale if it will improve the match. You need to pick two points on each image.

Balance Break a mosaic apart, examine average pixel value in the overlap regions, adjust brightness to match, and reassemble. This only works for images which have been produced just by mosaic joins! If you've done anything else to the image since loading it, the balance will fail with a mysterious message.

Manual Balance Adjust the brightness in a set of masked areas to match. Useful for removing shadows.

Rebuild Use this to mosaic up one set of files based on joins you made in another. Breaks a mosaic part to component files, performs a string substitution on the file names, and reassembles.

Clone Area Select over- or under-exposed pixels in one image and replace them with the corresponding pixels from another image. Useful for removing lead numbers used to identify X-ray plates.

The function operates on two 8-bit mono images. Move and resize the region on the first image to define the area around the white number. Move the region on the second to overlapping area. A section of the area on the second image is cloned and blended into the first image. The amount of the defined area to be cloned in defined by a slider within the output image.

5.8.3 Picture Frame

Items useful for mocking up painting frames.

5.8.4 Print

Items useful while preparing an image for printing.

Sharpen Sharpen an image for printing. This is a version of `Filter` / `Convolution` / `Unsharp Mask` tuned for typical inkjet printers.

Adjust Tone Curve Adjust the reproduction tone curve in LAB. Most useful for offset work, especially from transparencies.

Chapter 6

Programming

nip2 includes a tiny lazy functional programming language. You can use it to glue VIPS image processing functions together to perform more complicated tasks. All of the nip2 toolkit menus are written in this language.

These first sections just describe the programming language. See §4.4 on page 22 for a description of the programming window. You use nip2's programming language to control the user interface: the link between what happens inside a nip2 function and what you see on the screen in covered in §6.12 on page 43.

6.1 Load and save

When nip2 starts up it loads all of the definition files (files with a .def extension) it can find in the directories listed in your start path. You can change the start path in Preferences. By default, the start path lists just two areas: a personal start directory that nip2 makes in your home area, and the main system nip2 start directory containing all the standard toolkits.

If there are two files with the same name on the start path, then nip2 will only load the first one. This means that if you modify one of nip2's built-in menus and save it to your personal start directory, in future you'll just see your personalised version.

You can load or reload a toolkit at any time with the File / Open Toolkit menu item in the program window. If you open a toolkit with the same name as an existing toolkit, nip2 will remove the old toolkit before it loads the new one.

6.2 Using an external editor

If you're going to be doing any more than a little programming in nip2 you probably won't want to use the built-in editor. I suggest you start your favorite editor in one window on the screen and then in the nip2 program window click File / Open Toolkit and check the Pin-up box in the file selector.

Now every time you want to try out your definition, save the file from your external editor and click OK in nip2's file selector.

nip2's editor automatically adds some semicolon characters to separate definitions in a file. If you're using an external editor, you'll need to put these in yourself. Also check the syntax for adding separators and column items to menus.

6.3 Syntax

The most basic sort of definition looks like this:

```
// very simple!
fred = 12
```

This defines a function called fred whose value is the number 12. The // marks a comment: everything to the end of the line is skipped. Case is distinguished, so Fred and fred are two different functions. You can use letters, numbers, underscores and single quotes in function names.

You can have patterns on the left of the equals sign. For example:

```
[fred, petra] = [12, 13]
```

defines fred to have the value 12 and petra to have the value 13. See §6.9 on page 38 for details.

Functions may take parameters:

```
/* A function with parameters.
 */
jim a b = a + b + 12
```

This defines a function called `jim` which takes two parameters and whose value is the sum of the two parameters, plus 12. The `/*` and `*/` enclose a multi-line comment.

Functions may have several right-hand-sides, each right-hand-side qualified by a guard expression. Guards are tested from top to bottom and the first guard which has the value `true` causes the function to have the value of that right-hand-side. If no guard evaluates to `true`, then the last right-hand-side is used.

```
jenny a b
  = 42, a + b >= 100
  = 43, a + b >= 50
  = 44
```

This defines a function called `jenny` which takes two parameters and whose value is 42 if the sum of the parameters is 100 or greater; 43 if the sum is greater than or equal to 50 but less than 100; and 44 if the sum is less than 50.

Any function may be followed by any number of local functions, enclosed in curly braces. So `jenny` could be written as:

```
jenny a b
  = 42, sum >= 100
  = 43, sum >= 50
  = 44
{
  sum = a + b;
}
```

Note that you need a semi-colon after each local function. A local function may refer to anything in an enclosing scope, including itself.

You can write `if-then-else` expressions:

```
david a = if a < 12 then "my cat"
  else "likes lasagne"
```

This is exactly equivalent to:

```
david a
  = "my cat", a < 12
  = "likes lasagne"
```

`if-then-else` expressions are sometimes easier to read than guards.

Functions application is with spaces (juxtaposition). For example:

```
harry = jim 2 3
```

defines `harry` to have the value 17.

All functions are curried, that is, they can accept their arguments in stages. For example:

```
sandro = jim 1
```

defines `sandro`, a function which takes one parameter and will add 13 to it. This trick becomes very useful with list processing, see §6.7 on page 38.

`nip2` has some built-in functions, see Table 6.1 on page 33. They mostly take a single argument. All other functions are defined in the various standard toolkits and can be edited in the program window.

6.4 Naming conventions

You can name things in any way you like, but we've used the following conventions.

- Classes start with a capital letter, words are separated with underscores, subsequent words are not capitalised (eg. `Image_file`)
- Private names are prefixed with underscores (and are hidden by most of the user interface)
- Functions from the VIPS library are prefixed with `im_`
- Global utility functions (eg. `map`), public members (eg. `Colour.colour_space`) are all lower case, words are separated with underscores, subsequent words are not capitalised
- Constants are capitalised (eg. `Operator_type.COMPOUND_REWRAP`)

6.5 Evaluation

`nip2` calculates the value of an expression by using the definitions you entered to successively reduce the expression until it becomes one of the base types. Sometimes there is a choice as to which part of the expression

Function	Description
`dir` *any*	List names in scope
`has_member` *[char] any*	Does class have member
`name2gtype` *[char]*	Search for a GType by name
`gtype2name` *real*	Return the name of a GType
`error` *[char]*	Stop with error message
`print` *any*	Convert to string
`expand` *[char]*	Expand environment variables in string
`search` *[char]*	Search for a file
`_` *[char]*	Translate string
`is_image` *any*	Test for image
`is_bool` *any*	Test for boolean
`is_real` *any*	Test for real
`is_class` *any*	Test for class
`is_char` *any*	Test for char
`is_list` *any*	Test for list
`is_complex` *any*	Test for complex
`is_instanceof` *[char] any*	Test for instance of class
`re` *image/complex/class*	Extract real part of complex
`im` *image/complex/class*	Extract imaginary part of complex
`hd` *list*	Extract head of list
`tl` *list*	Extract tail of list
`sin` *image/number/class*	Sine
`cos` *image/number/class*	Cosine
`tan` *image/number/class*	Tangent
`asin` *image/number/class*	Arc sine
`acos` *image/number/class*	Arc cosine
`atan` *image/number/class*	Arc tangent
`log` *image/number/class*	Natural log
`log10` *image/number/class*	Base 10 log
`exp` *image/number/class*	e to the power
`exp10` *image/number/class*	10 to the power
`ceil` *image/number/class*	Round up
`floor` *image/number/class*	Round down
`gammq` *real real*	Normalised incomplete Gamma function
`vips_image` *[char]*	Load image from file
`read` *[char]*	Load file as a string

Table 6.1: `nip2` built in functions

will be reduced next — nip2 will always choose to reduce the leftmost, outermost part of the expression first.

For example, consider this definition:

```
factorial n
  = n * factorial (n - 1), n > 1
  = 1
```

And here's how nip2 will evaluate the expression factorial 3:

```
factorial 3 -->
  3 > 1 -->
  true
3 * factorial (3 - 1) -->
  (3 - 1) > 1 -->
  2 > 1 -->
  true
3 * (2 * factorial (2 - 1)) -->
  (2 - 1) > 1 -->
  1 > 1 -->
  false
3 * (2 * 1) -->
3 * 2 -->
6
```

Note how nip2 delays evaluating parameters to functions until they are needed, but still shares the result. 3 - 1 is only evaluated once, for example, even though the result is used three times. nip2 has a trace window: click on Debug / Trace in the program window and check the View / Operators menu item.

The advantage of this style of computation over conventional imperative programming languages is that you can reason about your program mathematically[1].

This isn't the best way to write a factorial function. A function with lots of recursive calls can be hard to understand — it's much better to use one of the higher order functions from the standard environment to encapsulate the type of recursion you want to use.

The clearest definition for factorial is probably:

```
factorial n = product [1..n]
```

See §6.6.5 on page 36 for an explanation of the list syntax.

[1] Since programs are referentially transparent (that is, the value of an expression depends only upon its syntactic context, not upon computation history), you can easily do equational reasoning, proof by induction, and so on.

Expressions are like theorems, definitions are like axioms, computation is like proof.

6.6 Operators

nip2's expression syntax is almost exactly the same as C, with a few small changes. Table 6.2 on page 35 lists all of nip2's operators in order of increasing precedence. If you've used C, the differences are:

- C's ?: operator becomes if-then-else, see above
- Like almost every functional language, nip2 uses square brackets for list constants (see §6.6.5 on page 36), so to index a list, nip2 uses ?
- nip2 adds @ for function composition, see §6.6.6 on page 37
- The : operator is infix list cons, see §6.7 on page 38
- The ++ operator becomes an infix concatenation operator, -- becomes list difference. Again, see §6.6.5 on page 36

The only slightly tricky point is that function application binds very tightly (only list index and class project bind more tightly). So the expression:

```
jim = fred 2 + 3
```

binds as:

```
jim = (fred 2) + 3
```

This is almost always the behaviour you want.

There are two special equality tests: === and !==. These test for pointer equality, that is, they return true if their arguments refer to the same object. These are occasionally useful for writing interactive functions.

6.6.1 The real type

nip2 has a single number type for integers and real numbers. All are represented internally as 64-bit floating point values. You can use the four standard arithmetic operators (+, −, *, /), remainder after integer division (%), raise-to-power (**), the relational operators (<, <=, >, >=, ==), the bitwise logical operators (&, |, ^, ~), integer shift operators (<<, >>) and unary negation and positive (−, +).

Other mathematical functions are pre-defined for you: sin, cos, tan, asin, acos, atan, log,

Operator	Associativity	Description
`if then else`	Right	If-then-else construct
`=>`	Left	Form name/value pair
`||`	Left	Logical or
`&&`	Left	Logical and
`@`		Function composition (see §6.6.6 on page 37)
`|`	Left	Bitwise or
`^`	Left	Bitwise exclusive or
`&`	Left	Bitwise and
`==`	Left	Equal to
`!=`		Not equal to
`===`		Pointer equal to
`!==`		Pointer not equal to
`<`	Left	Less than
`<=`		Less than or equal to
`>`		Greater than
`>=`		Greater than or equal to
`<<`	Left	Left shift
`>>`		Right shift
`+`	Left	Addition
`-`		Subtraction
`*`	Left	Multiplication
`/`		Division
`%`		Remainder after division
`!`	Left	Logical negation
`~`		One's complement
`++`		Join (see §6.6.5 on page 36)
`--`		Difference (see §6.6.5 on page 36)
`-`		Unary minus
`+`		Unary plus
`(`*type*`)`		Type cast expression
`**`	Right	Raise to power
`:`		List CONS (see §6.6.5 on page 36)
space	Left	Function application
`?`	Left	List index (see §6.6.5 on page 36)
`.`	Left	Class project (see §6.11 on page 40)

Table 6.2: `nip2` operators in order of increasing precedence

log10, exp, exp10, ceil, floor. Each has the standard behaviour.

You can use type-casts on reals. However, they remain 64-bit floating point, the range is simply clipped. Casting to unsigned short produces a 64-bit float whose fractional part has been set to zero, and which has been clipped to the range 0 to 65535. This may or may not cause rounding problems.

You can write hexadecimal number constants as 0xff.

6.6.2 The complex type

Complex numbers are rather sketchily implemented. They are generally handy for representing vectors and coordinates rather than for doing arithmetic, so the range of operations is limited.

Complex constants are written as two numbers enclosed in round brackets and separated by a comma. You can use the four standard arithmetic operators (+, −, *, /), raise-to-power (**), and unary negation and positive (−, +). You can use == only of the relational operators. You can mix complex and real numbers in expressions. You can cast reals to complex and back. Use the functions re and im to extract the real and imaginary parts.

```
(12, 13) + 4 == (16, 13)
(12, 2 + 2) ==  (12, 4)
re (12, 13) == 12
im (12, 13) == 13
```

6.6.3 The character type

Character constants are written as single characters enclosed in single quotes. You can use the relational operators (<, <=, >, >=, ==) to sort characters by ASCII order. You can cast a character to a real to get its ASCII value. You can cast a real ASCII value to a character. You can use the standard C escapes to represent non-ASCII characters.

```
(int) 'A' == 65
(char) 65 == 'A'
is_digit x = '0' <= x && x <= '9'
newline == '\n'
```

6.6.4 The boolean type

The two boolean constants are written as true and false. Boolean values are generated by the relational operators. You can use the standard logical operators (&&, ||, !). You can use a boolean type as an argument in an if-then-else expression.

As with C, the logical operators do not evaluate their right-hand sides if their value can be determined just from evaluating their left-hand sides.

```
true && false == false
true || error "boink!" == true
if true then 12 else 13 == 12
```

6.6.5 The list type

Lists are created from two constructors. [] denotes the empty list. The list construction operator (:, pronounced CONS by LISP programmers) takes an item and a list, and returns a new list with the item added to the front. As a convenience, nip2 has a syntax for list constants. A list constant is a list of items, separated by commas, and enclosed in square brackets:

```
12:[] == [12]
12:13:14:[] == 12:(13:(14:[])) ==
  [12,13,14]
[a+2,3,4] == (a+2):3:4:[]
[2]:[3,4] == [[2],3,4]
```

Use the functions hd and tl to take the head and the tail of a list:

```
hd [12,13,14] == 12
tl [12,13,14] == [13,14]
```

Use .. in a list constant to define a list generator. List generators build lists of numbers for you:

```
[1..10] == [1,2,3,4,5,6,7,8,9,10]
[1,3..10] == [1,3,5,7,9]
[10,9..1] == [10,9,8,7,6,5,4,3,2,1]
```

List generators are useful for expressing iteration.

Lists may be infinite:

```
[1..] == [1,2,3,4,5,6,7,8,9 ..]
[5,4..] == [5,4,3,2,1,0,-1,-2,-3 ..]
```

Infinite lists are useful for expressing unbounded iteration. See §6.8 on page 38.

You can write list comprehensions like this:

```
[x :: x <- [1..]; x % 2 == 0]
```

This could be read as *All x such that x is in* `[1..]` *and x is even*, that is, the list of even numbers.

You can have any number of semicolon-separated qualifiers and each one can be either a generator (like `x <- [1..]`) introducing a new variable or pattern (see §6.9 on page 38), or a predicate (like `x % 2 == 0`) which filters the generators to the left of it.

Later generators change more rapidly, so for example:

```
[(x, y) ::
  x <- [1..3]; y <- [x..3]] ==
    [(1, 1), (1, 2), (1, 3),
      (2, 2), (2, 3), (3, 3)]
```

You can nest list comprehensions to generate more complex data structures. For example:

```
[[x * y :: x <- [1..10]] ::
  y <- [1..10]]
```

will generate a times-table.

You can use pattern-matching (see §6.9 on page 38) to loop over several generators at the same time. For example:

```
[(x, y) :: [x, y] <-
  zip2 [1..3] [1..3]] ==
    [(1, 1), (2, 2), (3, 3)]
```

As a convenience, lists of characters may be written enclosed in double quotes:

```
"abc" == ['a','b','c']
```

You can define a string constant which has the same form as a variable name (that is, letters, numbers, underscore and apostrophy only) with a `$` prefix. For example:

```
$form7 == "form7"
```

`nip2` often uses these in option lists.

You can define a name, value pair with the => operator.

```
$fred => 12 == ["fred", 12]
```

Again, these pairs are frequently used to pass options to objects.

A list may contain any object:

```
[1,'a',true,[1,2,3]]
```

Mixing types in a list tends to be confusing and should be avoided. If you want to group a set of diverse objects, define a class instead, see §6.11 on page 40.

Lists of lists of reals are useful for representing arrays.

You can use the list index operator (`?`) to extract an element from a position in a list:

```
[1,2,3] ? 0 == 1
"abc" ? 1 == 'b'
```

You can use the list join operator (`++`) to join two lists together end-to-end.

```
[1,2,3] ++ [4,5,6] == [1,2,3,4,5,6]
```

You can use the list difference operator (`--`) to remove elements of one list from another.

```
[1..10] -- [4,5,6] == [1,2,3,7,8,9,10]
```

6.6.6 The function type

Functions are objects just like any other. You can pass functions to other functions as parameters, store functions in lists, and so on.

You can create anonymous functions with `\` (lambda). For example:

```
map (\x x + 2) [1..3] == [3, 4, 5]
```

You can nest lambdas to make multi-argument anonymous functions, for example:

```
map2 (\x\y x + y) [1..3] [2..5] ==
  [3, 5, 7]
```

You can compose functions with the `@` operator. For example, for two functions of one argument `f` and `g`:

```
f (g 2) == (f @ g) 2
```

6.6.7 The image type

These represent a low-level handle to a VIPS image structure. You can make them with the `vips_image` builtin, and you can pass them as parameters to VIPS functions. The `Image` class is built on top of them, see §6.12.3 on page 45.

As an accident of history, `nip2` also lets you do arithmetic with them. This will probably be removed in the next version or two, so it's best to go through the higher-level `Image` class.

6.7 Lists and recursion

Functional programming languages do not have variables, assignment or iteration. You can achieve the same effects using just lists and recursion.

There are two main sorts of recursion over lists. The first is called *mapping*: a function is applied to each element of a list, producing a new list in which each element has been transformed.

```
map fn [a,b,c] == [fn a, fn b, fn c]
```

The second main sort of recursion is called *folding*: a list is turned into a single value by joining pairs of elements together with a function and a start value.

```
foldr fn start [a,b .. c] ==
  (fn a (fn b (.. (fn c start))))
```

(The function is called `foldr` as it folds the list up right-to-left. There is an analogous function called `foldl` which folds a list up left-to-right, but because of the way lists work, it is much slower and should be avoided if possible.)

`map` is defined in the standard list library for you:

```
/* map fn l: map function fn over list
 */

map fn l
  = [], l == []
  = fn (hd l) : map fn (tl l)
```

So, for example, you could use `map` like this:

```
map (add 2) [1..5] == [3,4,5,6,7,8]
```

`foldr` is defined in the standard list library for you:

```
/* foldr fn st l: fold up list l,
 * right to left with function fn and
 * start value st
 */

foldr fn st l
  = st, l == []
  = fn (hd l) (foldr fn st (tl l))
```

So, for example, you could use `foldr` like this:

```
foldr add 0 [1..5] == 15
```

(Mathematically, `foldr` is the more basic operation. You can write `map` in terms of `foldr`, but you can't write `foldr` in terms of `map`.)

Unconstrained recursion over lists can be very hard to understand, rather like `goto` in an imperative language. It's much better to use a combination of `map` and `foldr` if you possibly can.

The toolkit `_list` contains definitions of most of the standard list-processing functions. These are listed in Table 6.3 on page 39. Check the source for detailed comments.

6.8 Lazy programming

`nip2`'s programming language is *lazy*, that is, it delays evaluation as long as it possibly can. For example, `error` is a function which immediately halts execution of your function and pops up an alert window. So:

```
12 + error "wombat!"
```

Has no value: this expression will halt with an error message. However:

```
false && error "lasagne!"
l
```

Will evaluate to `false`, since `nip2` knows after looking at the left-hand-side of `&&` that the result must be `false`, and so does not evaluate the right-hand-side.

```
[12, error "hot chilli!"] ? 0 == 12
```

This also evaluates completely, since the second element of the list is never used, and therefore never evaluates.

Things become more confusing when you start calling functions, since the arguments to a function call are also not evaluated until the function needs that value. For example:

```
foldr (error "boink!") 2 [] == 2
```

Again, this evaluates successfully, since the function is never used by `foldr`.

6.9 Pattern matching

Any time you define a name, you can use a pattern instead. For example:

Name	Description
`all l`	and all the elements of list `l` together
`any l`	or all the elements of list `l` together
`concat l`	join a list of lists together
`drop n l`	drop the first `n` elements from list `l`
`dropwhile fn l`	drop while `fn` is true
`extract n l`	extract element `n` from list `l`
`filter fn l`	all elements of `l` for which `fn` holds
`foldl fn st l`	fold list `l` left-to-right with `fn` and `st`
`foldl1 fn l`	like `foldl`, but use the first element of the list as the start value
`foldr fn st l`	fold list `l` right-to-left with `fn` and `st`
`foldr1 fn l`	like `foldr`, but use the first element of the list as the start value
`index fn l`	search list `l` for index of first element matching predicate `fn`
`init l`	remove last element of list `l`
`iterate f x`	repeatedly apply `f` to `x`
`last l`	return the last element of list `l`
`len l`	find length of list `l`
`limit l`	find the first element of list `l` equal to its predecessor
`map fn l`	map function `fn` over list `l`
`map2 fn l1 l2`	map 2-ary function `fn` over lists `l1` and `l2`
`map3 fn l1 l2 l3`	map 3-ary function `fn` over lists `l1`, `l2` and `l3`
`member l x`	true if `x` is a member of list `l`
`mkset l`	remove duplicates from list `l`
`postfix l r`	add element `r` to the end of list `l`
`product l`	product of list l
`repeat x`	make an infinite list of `x`es
`replicate n x`	make `n` copies of `x` in a list
`reverse l`	reverse list `l`
`scan fn st l`	apply `(foldr fn r)` to every initial segment of list `l`
`sort l`	sort list `l` into ascending order
`sortc fn l`	sort list `l` into order by using a comparison function
`sortpl pl l`	sort list `l` by predicate list `pl`
`sortr l`	sort list `l` into descending order
`split fn l`	break list `l` into sections separated by predicate `fn`
`splits fn l`	break list `l` into single sections separated by predicate `fn`
`splitpl pl l`	break list `l` up by predicate list `pl`
`split_lines n l`	break list `l` into lines of length `n`
`sum l`	sum list l
`take n l`	take the first `n` elements from list `l`
`takewhile fn l`	take from the front of `l` while `fn` holds
`zip2 l1 l2`	zip two lists together
`zip3 l1 l2 l3`	zip three lists together

Table 6.3: Functions in the standard list-processing toolkit

```
[fred, petra] = [12, 13]
```

defines `fred` to have the value 12 and `petra` to have the value 13.

A pattern describes the structure you are expecting for the value. When the value is computed it is matched against the pattern and, if the match is successful, the names in the pattern are bound to those parts of the value. Our example is exactly equivalent to:

```
temp = [12, 13];
fred
  = temp?0, is_list temp &&
      is_list_len 2 temp
  = error "pattern match failed";
petra
  = temp?1, is_list temp &&
    is_list_len 2 temp
  = error "pattern match failed";
```

where `temp` is an invisible, anonymous symbol.

You can pattern match on any of `nip2`'s data structures and types. You can use:

`a:b` Tests for the value being a non-empty list and then assigns `a` to the head and `b` to the tail.

`(a,b)` Tests for the value being a complex and then assigns `a` to the real part and `b` to the imaginary.

`[a,b,c]` Tests for the value being a list of length three and then assigns `a`, `b` and `c` to the three elements.

(*class − name* **`b)`** Tests for the value being an instance of the named class, then assigns `b` to that class instance.

constant Tests for the value being equal to that constant. Constants are things like `"hello world"` or `12`.

You can nest patterns in any way you like. Patterns are useful in conjunction with list comprehensions, see §6.6.5 on page 36.

You can't use patterns in function arguments in the current version, hopefully this will added shortly.

6.10 The standard libraries

`nip2` comes with a lot of little utility functions. The functions for list processing are listed in Table 6.3 on page 39. There are a huge number more, too many to really list here. Table 6.4 on page 41 lists all the utility toolkits with some hints about the kinds of function they contain. Read the (heavily commented) toolkits for details.

6.11 Classes

You can define new types using `class`. For example:

```
Pasta_plain = class {
  lasagne = "large sheets";
  fusilli = "sort of twisty";
  radiatori = "lots of ridges";
}
```

This defines a new class called `Pasta_plain`. The class has three members (`lasagne`, `fusilli` and `radiatori`), each of which has a list of `char` as its value. By convention, we've named classes with an initial capital letter, but of course you can do what you like.

You can refer to the members of a class using the class project (`.`) operator. For example:

```
Pasta_plain.lasagne == "large sheets"
```

You can use an expression to the right of `.` if you enclose it in brackets. For example:

```
Pasta_plain.("las" ++ "agne") ==
  "large sheets"
```

Classes can contain any objects as members, including functions and sub-classes. Functions may define local classes, classes may define local functions, and all may refer to each other using the usual scope rules. For example:

```
Pasta_all = class {
  filled = class {
    tortelloni = "venus' navel";
    ravioli = "square guys";
  }
  plain = Pasta_plain;
}
```

When you define a class, `nip2` adds a few extra members for you. `name` is a list of `char` giving the name of the class. `this` and `super` are the most-enclosing class instance and the class instance this class

Toolkit	Contains	Description
_convert	parse_int l,...	convert ascii text to numbers
	to_matrix x,...	convert anything into a matrix
	colour_transform_to to x,...	convert between colour spaces
_generate	image_new w h ...	make a blank image
	image_white i	look at image i, try to guess what white is
	make_xy w h	make an image of size w by h whose pixel value are their coordinates
_types	Image i	all the standard classes and support functions, see §6.13 on page 45
_predicate	is_colour_space i	test for objects are in various categories or have various properties
_stdenv	logical_and x,...	function versions of all the operators
	bandsplit i,...	break up and recombine images by band
	mean x,...	statistical ops on objects
	transpose x, flipud x, rot90 x,...	flips, rotates, etc. on objects
	rad x, pi,...	trigonometry stuff
	sign x, conj x, polar x,...	complex stuff
	rint x, ceil x,...	various rounding things
	fwfft x,...	fourier stuff
	dilate m x, rank w h n i,...	morphology stuff
	conv m x,...	convolution stuff
	image_set_type t i,...	set various image header field
	resize x y i,...	resampling images
	recomb m i,...	recombinations
	clip2fmt f i,...	format conversions
	hist_find m x,...	histogram stuff
	id x, const x y,...	various useful operations on functions
	map_binary fn x y,...	mapping over groups

Table 6.4: Useful utility functions — see the source for details

is derived from (see §6.11.2 on page 42). nip2 also adds a default constructor: a member with the same name as the class, pointing back to the class constructor.

For efficiency reasons nip2 does not allow mutual recursion at the top level. If two functions depend on each other, neither will ever be calculated. For example:

```
a = 1 : b;
b = 2 : a;
```

Neither a nor b will have a value.

You can have mutual recursion between class members. For example:

```
Fred = class {
  a = 1 : b;
  b = 2 : a;
}
```

Now Fred.a will have the value [1, 2, 1, 2, 1, ...].

6.11.1 Parameterised classes

Classes can have parameters. Parameters behave like class members initialised from arguments to the class constructor. For example:

```
My_pasta pasta_name cooked = class {
  is_ready t = "your " ++
    pasta_name ++ " is " ++ state
  {
    state
      = "underdone!", t < cooked
      = "perfect", t == cooked
      = "yuk!";
  }
}
```

This defines a class called My_pasta which takes a pasta name and a cooking time as parameters. Once you have made an instance of My_pasta, you can test if it's been cooked at a certain time with the is_ready member. For example:

```
tele = My_pasta "telephoni" 10;
tele.is_ready 5 ==
  "your telephoni is underdone!"
```

6.11.2 Inheritance

Classes can inherit from a super-class. For example:

```
Pasta_more = class Pasta_plain {
  macaroni = "tubes";
  spaghetti = "long and thin";
  lasagne = "fairly large sheets";
}
```

Here the new class Pasta_more inherits members from the previous class Pasta_plain. It also overrides the definition of lasagne from Pasta_plain with a new value. For example:

```
Pasta_more.macaroni == "tubes"
Pasta_more.fusilli == "sort of twisty"
Pasta_more.lasagne == "fairly large sheets"
```

You can use this and super to refer to other members up and down the class hierarchy. super is the class instance that the current class inherits from (if there's no super-class, super has the value []), and this is the most-enclosing class instance.

```
Pasta_more.super == Pasta_plain
Pasta_more.this == Pasta_more
Pasta_more.super.this == Pasta_more
```

therefore:

```
Pasta_more.lasagne == "fairly large sheets"
Pasta_more.super.lasagne == "large sheets"
Pasta_more.super.this.lasagne ==
  "fairly large sheets"
```

There's a special symbol root which encloses all symbols. For example:

```
fred = 12;

Freddage = class {
  fred = 42;
  mystery = root.fred;
}
```

Now Fred.mystery will have the value 12.

There's another special symbol called scope which encloses all symbols in the file this definition was loaded from. If you want to refer to another definition in the same file which is being masked somehow, use scope.

You can use the built in function is_instanceof to test whether an instance is or inherits from a class. For example:

```
is_instanceof "Pasta_more" Pasta_more = true
is_instanceof "Pasta_plain" Pasta_more = true
is_instanceof "Pasta_more" Pasta_plain
  false
```

The super-class constructor can take arguments, and these arguments can refer to class members. For example:

```
Fresh_pasta pasta_name = class
  My_pasta pasta_name cooked {
  cooked = 2;
}
```

Defines a class for fresh pasta, which always cooks in 2 minutes. You need to be careful not to make loops: if `cooked` did tried to refer to something in the superclass, this class would never construct properly. `nip2` unfortunately does not check for this error.

Finally, the superclass can be a fully constructed class. In this case, the superclass is cloned and the new class members wrapped around it. You can use this to write a class which can wrap any other class and add members to it. Many of the toolkit menu items use this trick to enable them to work for any object type.

6.11.3 Minor class features

There are a couple of other things you can do with classes. You can define a special member called `_check`. If this member is defined, then when a class instance is created, the check member is returned instead of the class itself. You can use this to implement class argument type checks, for example:

```
Fred a b = class {
  _check
    = this, is_real a && is_real b
    = error "args to Fred must " ++
      "both be real"
}
```

Defines a class called `Fred` which has to have two real numbers as arguments.

You can define members called `oo_binary`, `oo_binary'` and `oo_unary` and do operator overloading. When `nip2` sees one of the standard operators being used on an instance of your class, it will look up one of these members and pass in the name of the operator and the argument. The two forms of the binary operator member are called for the class-on-left and the class-on-rights cases. So:

```
x = Fred 1 2
x + 12 == x.oo_binary "add" 12
12 + x == x.oo_binary' "add" 12
!x == x.oo_unary "negate"
```

These two features are very primitive. The `_Object` class in the `_types` toolkit builds on these to provide a fairly high-level system for checking class arguments and defining the meaning of operators. See §6.13 on page 45.

6.12 Controlling the interface

`nip2` looks at the scraps of program you type in and execute and tries to show them on the screen in a graphical way. The sorts of display you get depend on where in `nip2` you define the expression, and what sort of value it has.

6.12.1 Tools and toolkits

Definitions in toolkits are turned into menus off the `Toolkits` menu in the main window, and added to the toolkit browser. Toolkits are loaded from files at startup or can be made in the program window. Toolkit or a definition names which start with an underscore character are hidden and not displayed. The toolkits are always displayed in alphabetical order, but you can order the items within a toolkit in any way you like.

There are two ways to write toolkit definitions. Function definitions and zero-argument classes simply appear as menu items, built from static analysis of their source code. However, if a definition evaluates to an instance of the class `Menu`, a menu item is built from dynamic analysis of the value of the definition.

Static menu items

Zero-argument classes within toolkits are displayed as pull-right menus. You can nest classes to any depth.

`nip2` uses the first line of the comment before a definition as help text for that function, so it's a good idea to put a simple one-line description of the function at the start of a comment.

For example, if the following text is placed in a file called `Fred.def` on `nip2`'s start path, you'll get a menu in the tookits called `Fred` with a pull-right and a tooltip. See Figure 6.1 on page 44.

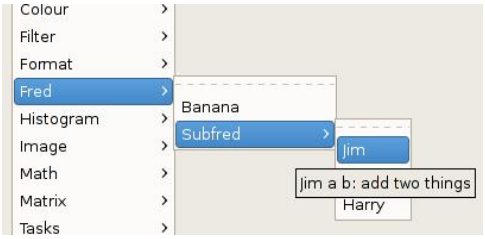

Figure 6.1: How `Fred.def` will look

```
Banana a = a * 3;

Subfred = class {
  // add two things
  Jim a b = a + b;
  Apple e = e * 12;
  Harry z = 12 + z;
}
```

Dynamic menu items

Dynamic menus give you much more control over the way menus are drawn and make it easy to reuse menus. A dynamic menu item is a class instance that is a subclass of `Menuitem`. It needs to have three members: `label`, the text that should appear in the menu (with an underscore character to indicate the mnenonic); `tooltip`, a short hint that appears as a tooltip or in the toolkit browser; `icon`, an optional image file to be displayed in the menu next to the text; and `action`, the function that is called when the menu item is activated. `label` and `tooltip` are constructor arguments for `Menu`.

So for example:

```
Wombat_find_item = class Menuitem
  "_Find Wombat"
  "analyse image and locate wombat" {
  icon = "nip-slider-16.png";
  action x = im_wombat_locate x;
}
```

will appear as shown in Figure 6.2.

A dynamic pullright menu is a subclass of `Menupullright`. It's just like `Menuitem`, but without the need for an `action` member. Any members which are subclasses of `Menu` are displayed as items in the submenu. So again:

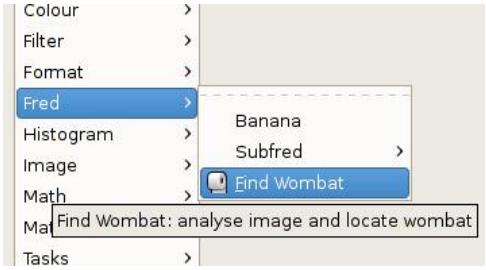

Figure 6.2: How `Wombat_find_item` will look

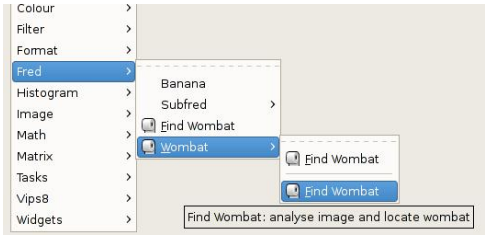

Figure 6.3: How `Wombat_item` will look

```
Wombat_item = class Menupullright
  "_Wombat"
  "wombat-related operations" {
  icon = "nip-slider-16.png";
  item1 = Wombat_find_item;
  sep = Menuseparator;
  boink = Wombat_find_item;
}
```

will appear as shown in Figure 6.3.

6.12.2 Workspaces

Definitions in workspaces are displayed with `nip2`'s class browser. Each row is displayed in four main parts: a button for the row name, a line of text, a set of sub-rows for the members of the row's class, and a graphic display representing the row's value. See Figure 6.4.

The text part of the right-hand-side of each row is always displayed, but the sub-rows are only displayed if the row represents a class, and the graphic is only displayed if the class is an instance of one of the classes in Table 6.5 on page 46. You can subclass these if you want to use the graphic display in your own widgets.

There are three separate ways to set the value for a row. You can edit the line of program text, you can

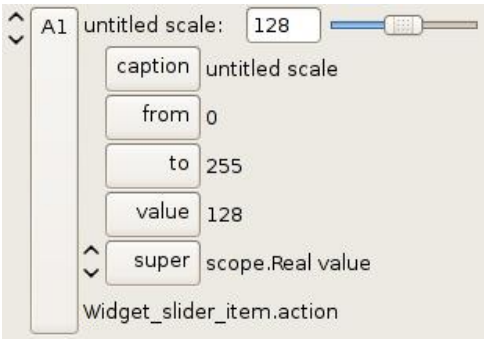

Figure 6.4: Components of a workspace row

edit one of the members, or you can manipulate the graphic representation (dragging a slider, or moving a region). These can be contradictory, so nip2 resolves conflicts by always applying changes in the order text, then graphic, then member.

When it applies a graphic change, nip2 rebuilds the class using a class member called *class-name*_edit, or if that is not defined, the class's constructor member. For example, the Colour class can be defined as:

```
Colour colour_space value = class {}
A1 = Colour "sRGB" [255,0,0];
```

There are two ways to change A1. You can open A1 and change colour_space to "Lab", or you can double-click on the swatch and drag the disc. When you click OK on the colour edit dialog, nip2 searches for a member called Colour_edit, fails to find it, and so picks the Colour member instead (the default constructor generated by nip2). It then replaces the value of A1 with

[needs finishing]

6.12.3 The Image class

.

say supports mioxed ops with real, vector and complex constants

6.12.4 The Colour class

This class displays a swatch of colour. If you double-click on the

```
Pathname caption value = class {}
```

6.13 The _Object class

6.14 Optimisation

nip2 performs three useful optimisations on expressions. First, it finds and removes common subexpressions in functions. So for example:

```
if a + b < 12 then a + b else b
```

will only evaluate a + b once. This can save a lot of time if a or b is a large image.

Second, nip2 detects arithmetic operations on unsigned char images, and replaces them with look-up tables. For example:

```
a = vips_image "campin.v"
b = a * (a - 1) ** 0.5
```

Provided campin.v is an 8 bit image image, this expression will evaluate with a single call to im_maplut().

Finally, nip2 has a VIPS operation cache. It memorises the arguments to the last few hundred calls to VIPS, and the result each call gave. Before calling VIPS again, it checks to see if there is a previous call with the same arguments and if there is, uses the result it obtained last time.

6.15 Calling VIPS functions

You can call any VIPS operation which has the following properties:

- There must be at least 1 output argument. If there's a single output argument, that becomes the value of the function. If there is more than one output, then the function returns a list with the outputs as members.
- The output arguments must all be one of:
  - IM_TYPE_DOUBLE,
  - IM_TYPE_INT,
  - IM_TYPE_COMPLEX,
  - IM_TYPE_STRING,
  - IM_TYPE_IMAGE,
  - IM_TYPE_DOUBLEVEC,

Class	Description
`Clock` *interval value*	A clock widget, handy for animations
`Expression` *caption expr*	Displays an editable expression
`Group` *value*	A group of objects for iteration
`List` *value*	A list of related objects
`Pathname` *caption value*	Displays a file browser
`Fontname` *caption value*	Displays a font browser
`Toggle` *caption value*	A toggle switch
`Scale` *caption from to value*	A slider
`Option` *caption labels value*	Select one item from a list
`Colour` *colour_space value*	A patch of colour
`Matrix_vips` *value scale offset filename display*	A matrix
`Arrow` *image left top width height*	Two points joined by a line on an image
`Region` *image left top width height*	A sub-area of an image
`Plot` *options value*	Displays a plot widget
`Image` *value*	An image
`Number` *caption value*	Displays an editable number
`Real` *value*	Displays a real number
`Vector` *value*	Displays a list of reals
`String` *caption value*	Displays an editable string
`Mark` *image left top*	A point on an image
`HGuide` *image top*	A horizontal line on an image
`VGuide` *image left*	A vertical line on an image
`Area` *image left top width height*	A sub-area of an image, fixed in size

Table 6.5: `nip2` built in graphic classes

  - IM_TYPE_DMASK,
  - IM_TYPE_IMASK

- The input arguments must all be one of the types above, or IM_TYPE_DISPLAY. If an argument is an input display, nip2 passes in its current display structure, it does not take a display from your program.

When nip2 starts up, it loads any VIPS plug ins it can find on its data search path. You can call functions from plug ins in just the same way. For information on writing plug ins, see the *VIPS Manual*.

Appendix A

Configuration

Click on the Edit / Preferences to see all the preference options. There are a lot of things you can change (probably too many). This section will list the most important.

A.0.1 Calculation

This column has the options which control how nip2 starts and how and when it calculates.

Data path This is a list of directories where nip2 searches for data files. These are any files that nip2 can use but which aren't loaded at startup. I usually append the main areas on my machine where I store image files, for example.

The default value is ["$HOME/.$PACKAGE-$VERSION/data", "$VIPSHOME/share/nip/data", "."].

Temporary files This is where any intermediate files will be stored. It defaults to a directory called tmp under your home area's .nip2-xx directory. If nip crashes, it may leave old files here.

Start path This lists directories which are searched when nip2 starts for any loadable files. Anything that nip2 comes across will be loaded up.

The default value is ["$HOME/.$PACKAGE-$VERSION/start", "$VIPSHOME/share/nip/start"].

Auto-recalc With this on (the default) nip2 will recalculate whenever anything changes. Turn this off if recalculations are taking a long time and you want to make a series of small changes.

Update sliders during drag This sets whether recalculation happens as sliders are dragged, or whether the recalculation waits until the drag finishes. There's a similar setting for regions.

Auto workspace save With this tured on (the default) nip2 will save the current workspace to the temporary files area a second after the last recalculation. If nip2 crashes, you can restart it and click File / Search for Workspace Backups and nip2 will reload the last workspace where you made a change.

Auto-reload on file change With this turned on nip2 will automatically reload any image files that change while it has them open. Handy if you're using nip2 to watch a file that another program is updating.

Maximum text display This sets the number of characters nip2 shows for string values. Turn it up if you want to see inside long strings.

Maximum heap This sets the limit on the heap size. Turn it up if you start getting Heap full error messages. If you left-click on the space free label in the bottom right of the main window, it will change to display the current heap statistics. There's a useful tooltip as well.

Number of CPUs to use If you have a machine with more than one CPU, you can make nip2 faster by upping this number.

A.0.2 Image display

This set of options control the default image display window settings. Useful if you're always having to turn the status bar on (for example). The maximum size option is handy if you're using nip2 on a machine with a small display.

The Auto popup option makes nip2 pop up an image display window automatically whenever you make a new image object.

A.0.3 Other options

Other areas of preferences are less useful.

Display LEDs If you're using a theme which uses bitmaps for widgets, you won't be able to see the button colour changes nip2 usually uses to indicate state. This option adds three small LEDs to each row which indicate select, busy and error.

Default image format By default nip2 file browsers show only VIPS images. If you find you mostly use (for example) JPEG images, you'll save yourself a few clicks on every file operation by switching this option to JPEG format.

Image format options You can set the save options for the various image formats.

Video for linux If you running Linux and have a capture card that supports the V4L interface, you can capture straight from the card into nip2. Set the capture options here.

Paintbox The paintbox normally tracks all undo operations. This can chew up a lot of memory, especially for flood fills. Reduce the number of undo steps to free up some RAM.