VIPS Manual

Version 7.14

John Cupitt, Kirk Martinez

This manual formatted September 15, 2008

Contents

List of Figures

List of Tables

Chapter 1

VIPS from C++ and Python

1.1 Introduction

This chapter describes the C++ API for the VIPS image processing library. The C++ API is as efficient as the C interface to VIPS, but is far easier to use: almost all creation, destruction and error handling issues are handled for you automatically.

The Python interface is a very simple wrapping of this C++ API generated automatically with SWIG. The two interfaces are identical, except for language syntax.

1.1.1 If you've used the C API

To show how much easier the VIPS C++ API is to use, compare Figure 2.2.5 on page 18 to Figure 1.1 on page 2. Figure 1.2 on page 2 is the same thing in Python.

A typical build line for the C++ program might be:

```
g++ invert.cc \
  `pkg-config vipsCC-7.14 \
    --cflags --libs`
```

The key points are:

- You just include <vips/vips> — this then gets all of the other includes you need. Everything is in the vips namespace.
- The C++ API replaces all of the VIPS C types — IMAGE becomes VImage and so on. The C++ API also includes VDisplay, VMask and VError.
- Image processing operations are member functions of the VImage class — here, VImage(argv[1]) creates a new VImage object using the first argument to initialise it (the input filename). It then calls the member function invert(), which inverts the VImage and returns a new VImage. Finally it calls the member function write(), which writes the result image to the named file.
- The VIPS C++ API uses exceptions — the VError class is covered later. If you run this program with a bad input file, for example, you get the following output:

```
example% invert jim fred
invert: VIPS error: im_open:
  "jim" is not a supported
  format
```

1.2 The VIPS file format

VIPS has its own very simple file format. It is used inside VIPS to hold images during computation. You can save images in VIPS format if you want, but the VIPS format is not widely used and you may have problems reading your images into other packages.

If you intend to keep an image, it's much better to save it as TIFF, JPEG, PNG or PBM/PGM/PPM. VIPS can transparently read and write all these formats.

1.2.1 VIPS file header

All VIPS image files start with a 64-byte header giving basic information about the image dimensions, see Table 1.1 on page 4. This is followed by the image data. This is usually just the pixel values in native format (ie. the byte order used by the machine that wrote the file) laid out left-to-right and top-to-bottom. After the image data comes a block of optional XML which holds extra

```
#include <iostream>
#include <vips/vips>

int
main (int argc, char **argv)
{
  if (argc != 3)
    {
      std::cerr << "usage: " << argv[0] << " infile outfile\n";
      exit (1);
    }

  try
  {
    vips::VImage fred (argv[1]);

    fred.invert ().write (argv[2]);
  }
  catch (vips::VError e)
  {
    e.perror (argv[0]);
  }

  return (0);
}
```

Figure 1.1: `invert` program in C++

```
#!/usr/bin/python

import sys
from vipsCC import *

try:
  a = VImage.VImage (sys.argv[1])
  a.invert ().write (sys.argv[2])
except VError.VError, e:
  e.perror (sys.argv[0])
```

Figure 1.2: `invert` program in Python

image metadata, such as ICC profiles and image history. You can use the command-line program `header` to extract the XML from an image and `edvips` to replace it, see the man pages.

The `Type` field, the `Xres`/`Yres` fields, and the `Xoffset`/`Yoffset` fields are advisory. VIPS maintains their value (if you convert an image to *CIE* $L^*a^*b^*$ colour space with `im_XYZ2Lab()`, for example, VIPS will set `Type` to be `IM_TYPE_LAB`), but never uses these values itself in determining the action of an image processing function. These fields are to help the user, and to help application programs built on VIPS which are trying to present image data to the user in a meaningful way.

The `BandFmt`, `Coding` and `Type` fields can take the values shown in tables 1.2, 1.3 and 1.4. The C++ and Python names for these values are slightly different, for historical reasons.

1.2.2 Computation formats

This type of image has `Coding` set to `IM_CODING_NONE`. The header is then followed by a large array of pixels, laid out left-to-right, top-to-bottom. Each pixel contains the specified number of bands. Each band has the specified band format, which may be an 8-, 16- or 32-bit integer (either signed or unsigned), a single or double precision IEEE floating point number, or a pair of single or double precision floats forming a complex number.

All values are stored in the host-machine's native number representation (that is, either most-significant first, as in SPARC and 680x0 machines, or least-significant first, for Intel and DEC machines). The VIPS library will automatically byte-swap for you if necessary, but this can be slow.

1.2.3 Storage formats

All storage formats have other values for the `Coding` field. This release supports only `IM_CODING_LABQ` format.

`IM_CODING_LABQ` stores L^*, a^* and b^* for each pixel, with 10 bits for L^* and 11 bits for each of a^* and b^*. These 32 bits are packed into 4 bytes, with the most significant 8 bits of each value in the first 3 bytes, and the left-over bits packed into the final byte as 2:3:3.

This format is a little awkward to process. Some VIPS functions can work directly on `IM_CODING_LABQ` images (`im_extract()`, for example), but most will require you to unpack the image to one of the computation formats (for example with `im_LabQ2Lab()`) first.

1.3 The `VImage` class

The `VImage` class is a layer over the VIPS `IMAGE` type. It automates almost all of the image creation and destruction issues that complicate the C API, it automates error handling, and it provides a convenient system for composing operations.

1.3.1 Constructors

There are two principal constructors for `VImage`:

```
VImage::VImage( const char *name,
  const char *mode = "r" );
VImage::VImage();
```

The first form creates a new `VImage`, linking it to the named file. `mode` sets the mode for the file: it can take the following values:

"r" The named image file is opened read-only. This is the default mode.

"w" A `VImage` is created which, when written to, will write pixels to disc in the specified file.

"t" As the `"w"` mode, but pixels written to the `VImage` will be saved in a temporary memory buffer.

"p" This creates a special 'partial' image. Partial images represent intermediate results, and are used to join VIPS operations together, see §1.3.6 on page 7.

"rw" As the `"r"` mode, but the image is mapped into your address space read-write. This mode is only provided for the use of paintbox-style applications, which need to directly modify an image. See §4.2.8 on page 63.

The second form of constructor is shorthand for:

```
VImage( "VImage:1", "p" )
```

Bytes	Represent	VIPS name
0–3	VIPS magic number (in hex, 08 f2 f6 b6)	
4–7	Number of pels per horizontal line (integer)	`Xsize`
8–11	Number of horizontal lines (integer)	`Ysize`
12–15	Number of bands (integer)	`Bands`
16–19	Unused (legacy)	`Bbits`
20–23	Band format (eg. `IM_BANDFMT_USHORT`)	`BandFmt`
24–27	Coding type (eg. `IM_CODING_NONE`)	`Coding`
28–31	Type (eg. `IM_TYPE_LAB`)	`Type`
32–35	Horizontal resolution (float, pixels mm^{-1})	`Xres`
36–39	Vertical resolution (float, pixels mm^{-1})	`Yres`
40–43	Unused (legacy)	`Length`
44–45	Unused (legacy)	`Compression`
46–47	Unused (legacy)	`Level`
48–51	Horizontal offset of origin	`Xoffset`
52–55	Vertical offset of origin	`Yoffset`
56–63	For future expansion (all zeros for now)	

Table 1.1: VIPS header

`BandFmt`	C++ and Python name	Value	Meaning
`IM_BANDFMT_NOTSET`	`FMTNOTSET`	-1	
`IM_BANDFMT_UCHAR`	`FMTUCHAR`	0	Unsigned 8-bit int
`IM_BANDFMT_CHAR`	`FMTCHAR`	1	Signed 8-bit int
`IM_BANDFMT_USHORT`	`FMTUSHORT`	2	Unsigned 16-bit int
`IM_BANDFMT_SHORT`	`FMTSHORT`	3	Signed 16-bit int
`IM_BANDFMT_UINT`	`FMTUINT`	4	Unsigned 32-bit int
`IM_BANDFMT_INT`	`FMTINT`	5	Signed 32-bit int
`IM_BANDFMT_FLOAT`	`FMTFLOAT`	6	32-bit IEEE float
`IM_BANDFMT_COMPLEX`	`FMTCOMPLEX`	7	Complex (2 floats)
`IM_BANDFMT_DOUBLE`	`FMTDOUBLE`	8	64-bit IEEE double
`IM_BANDFMT_DPCOMPLEX`	`FMTDPCOMPLEX`	9	Complex (2 doubles)

Table 1.2: Possible values for `BandFmt`

`Coding`	C++ and Python name	Value	Meaning
`IM_CODING_NONE`	`NOCODING`	0	VIPS computation format
`IM_CODING_LABQ`	`LABQ`	2	LABQ storage format

Table 1.3: Possible values for `Coding`

Type	C++ and Python name	Value	Meaning
IM_TYPE_MULTIBAND	MULTIBAND	0	Some multiband image
IM_TYPE_B_W	B_W	1	Some single band image
IM_TYPE_HISTOGRAM	HISTOGRAM	10	Histogram or LUT
IM_TYPE_FOURIER	FOURIER	24	Image in Fourier space
IM_TYPE_XYZ	XYZ	12	*CIE XYZ* colour space
IM_TYPE_LAB	LAB	13	*CIE* $L^*a^*b^*$ colour space
IM_TYPE_CMYK	CMYK	15	im_icc_export()
IM_TYPE_LABQ	LABQ	16	32-bit *CIE* $L^*a^*b^*$
IM_TYPE_RGB	RGB	17	Some RGB
IM_TYPE_UCS	UCS	18	*UCS(1:1)* colour space
IM_TYPE_LCH	LCH	19	*CIE LCh* colour space
IM_TYPE_LABS	LABS	21	48-bit *CIE* $L^*a^*b^*$
IM_TYPE_sRGB	sRGB	22	sRGB colour space
IM_TYPE_YXY	YXY	23	*CIE Yxy* colour space
IM_TYPE_RGB16	RGB16	25	16-bit RGB
IM_TYPE_GREY16	GREY16	26	16-bit monochrome

Table 1.4: Possible values for Type

It is used for representing intermediate results of computations.

Two further constructors are handy for wrapping VImage around existing images.

```
VImage( void *buffer,
  int width, int height, int bands,
  TBandFmt format );
VImage( void *image );
```

The first constructor makes a VImage from an area of memory (perhaps from another image processing system), and the second makes a VImage from an IMAGE.

In both these two cases, the VIPS C++ API does not assume responsibility for the resouces: it's up to you to make sure the buffer is freed.

The Python interface adds the usual frombuffer and fromstring methods.

```
VImage.fromstring (string,
  width, height, bands, format) ->
  VImage

VImage.frombuffer (buffer,
  width, height, bands, format) ->
  VImage
```

Use fromstring to avoid worries about object lifetime, but you'll see a lot of copies and high memory use. Use frombuffer for speed, but you have to manage object lifetime yourself.

They are useful for moving images into VIPS from other image processing libraries. There's also a utility function, vips_from_PIL_mode, which turns a PIL mode into a VIPS band, format, type triple.

```
VImage.vips_from_PIL_mode (mode) ->
  (bands, format, type)
```

See also tobuffer and tostring below.

1.3.2 File conversion

VIPS can read and write a number of different file formats. Information about file format conversion is taken from the filename. For example:

```
VImage jim( "fred.jpg" );
```

This will decompress the file fred.jpg to a memory buffer, wrap a VIPS image around the buffer and build a reference to it called jim.

Options are passed to the file format converters embedded in the filename. For example:

```
VImage out( "jen.tif:deflate", "w" );
```

Writing to the descriptor `out` will cause a TIFF image to be written to disc with deflate compression.

See the manual page for `im_open(3)` for details of all the file formats and conversions available.

1.3.3 Large files

Opening large files in formats like JPEG which do not support random access can use large amounts of memory, since the file has to be decompressed before it can be used.

The `convert2disc()` member lets you decompress to a disc file rather than to memory. For example:

```
VImage fred = VImage::convert2disc( "im_jpeg2vips",
  "file.jpg", "temp.v" );
```

Will decompress to the file `temp.v`, then open that and return a reference to it. You will need to delete the file once you are finished with it.

1.3.4 Projection functions

A set of member functions of `VImage` provide access to the fields in the header:

```
int Xsize();
int Ysize();
int Bands();
TBandFmt BandFmt();
TCoding Coding();
TType Type();
float Xres();
float Yres();
int Length();
TCompression Compression();
short Level();
int Xoffset();
int Yoffset();
```

Where `TBandFmt`, `TCoding`, `TType` and `TCompression` are `enums` for the types in the VIPS file header. See section §1.2.1 on page 1 for an explanation of all of these fields.

Two functions give access to the filename and history fields maintained by the VIPS IO system.

```
char *filename();
char *Hist();
```

You can get and set extra metadata fields with `meta_get()` and `meta_set()`. They read and write `GValue` objects, see §2.2.6 on page 16.

```
void meta_set( const char *field, GValue *value );
void meta_get( const char *field, GValue *value_copy
GType meta_get_type( const char *field );
```

A set of convenience functions build on these two to provide accessors for common types.

```
int meta_get_int( const char *field )
double meta_get_double( const char *field )
const char *meta_get_string( const char *field )
void *meta_get_area( const char *field )
void *meta_get_blob( const char *field, size_t *leng

void meta_set( const char *field, int value )
void meta_set( const char *field, double value )
void meta_set( const char *field, const char *value
void meta_set( const char *field,
VCallback free_fn, void *value )
void meta_set( const char *field,
VCallback free_fn, void *value, size_t length )
```

The `image()` member function provides access to the `IMAGE` descriptor underlying the C++ API. See the §2.1 on page 11 for details.

```
void *image();
```

The `data()` member function returns a pointer to an array of pixel data for the image.

```
void *data() const;
```

This can be very slow and use huge amounts of RAM.

The Python interface adds `tobuffer` and `tostring`. These operations call `data()` to generate the image pixels and then either copy it and return the copy as a string, or wrap the pixels up as a Python buffer object.

Use `tostring` to avoid worries about object lifetime, but you'll see a lot of copies and high memory use. Use `tobuffer` for speed, but you have to manage object lifetime yourself.

They are useful for moving images from VIPS into other image processing libraries. There's also a utility function, `PIL_mode_from_vips`, which makes a PIL mode from a VIPS image.

```
VImage.PIL_mode_from_vips (vips-image) ->
  mode
```

See also `frombuffer` and `fromstring` above.

1.3.5 Assignment

VImage defines copy and assignment, with reference-counted pointer-style semantics. For example, if you write:

```
VImage fred( "fred.v" );
VImage jim( "jim.v" );

fred = jim;
```

This will automatically close the file fred.v, and make the variable fred point to the image jim.v instead. Both jim and fred now point to the same underlying image object.

Internally, a VImage object is just a pointer to a reference-counting block, which in turn holds a pointer to the underlying VIPS IMAGE type. You can therefore efficiently pass VImage objects to functions by value, and return VImage objects as function results.

1.3.6 Computing with VImages

All VIPS image processing operations are member functions of the VImage class. For example:

```
VImage fred( "fred.v" );
VImage jim( "jim.v" );

VImage result = fred.cos() + jim;
```

Will apply im_costra() to fred.v, making an image where each pixel is the cosine of the corresponding pixel in fred.v; then add that image to jim.v. Finally, the result will be held in result.

VIPS is a demand-driven image processing system: when it computes expressions like this, no actual pixels are calculated (although you can use the projection functions on images — result.BandFmt() for example). When you finally write the result to a file (or use some operation that needs pixel values, such as min(), find minimum value), VIPS evaluates all of the operations you have called to that point in parallel. If you have more than one CPU in your machine, the load is spread over the available processors. This means that there is no limit to the size of the images you can process.

§4.2 on page 55 lists all of the VIPS packages. These general rules apply:

- VIPS operation names become C++ member function names by dropping the im_ prefix, and if present, the tra postfix, the const postfix and the _vec postfix. For example, the VIPS operation im_extract() becomes extract(), and im_costra() becomes cos().
- The VImage object to which you apply the member function is the first input image, the member function returns the first output. If there is no image input, the member is declared static.
- INTMASK and DOUBLEMASK types become VMask objects, im_col_display types become VDisplay objects.
- Several C API functions can map to the same C++ API member. For example, im_andimage, im_andimage_vec and im_andimageconst all map to the member andimage. The API relies on overloading to discriminate between these functions.

This part of the C++ API is generated automatically from the VIPS function database, so it should all be up-to-date.

There are a set of arithmetic operators defined for your convenience. You can generally write any arithmetic expression and include VImage in there.

```
VImage fred( "fred.v" );
VImage jim( "jim.v" );

Vimage v = int((fred + jim) / 2);
```

1.3.7 Writing results

Once you have computed some result, you can write it to a file with the member write(). It takes the following forms:

```
VImage write( const char *name );
VImage write( VImage out );
VImage write();
```

The first form simply writes the image to the named file. The second form writes the image to the specified VImage object, for example:

```
VImage fred( "fred.v" );
VImage jim( "jim buffer", "t" );

Vimage v = (fred + 42).write( jim );
```

This creates a temporary memory buffer called `jim`, and fills it with the result of adding 42 to every pixel in `fred.v`.

The final form of `write()` writes the image to a memory buffer, and returns that.

1.3.8 Type conversions

Two type conversions are defined: you can cast `VImage` to a `VDMask` and to a `VIMask`.

```
operator VDMask();
operator VIMask();
```

These operations are slow and need a lot of memory! Emergencies only.

1.4 The `VMask` class

The `VMask` class is an abstraction over the VIPS `DOUBLEMASK` and `INTMASK` types which gives convenient and safe representation of matricies.

`VMask` has two sub-classes, `VIMask` and `VDMask`. These represent matricies of integers and doubles respectively.

1.4.1 Constructors

There are four constructors for `VIMask` and `VDMask`:

```
VIMask( int xsize, int ysize );
VIMask( int xsize, int ysize,
  int scale, int offset, ... );
VIMask( int xsize, int ysize,
  int scale, int offset,
  std::vector<int> coeff );
VIMask( const char *name );
VIMask();
VDMask( int xsize, int ysize );
VDMask( int xsize, int ysize,
  double scale, double offset, ... );
VDMask( int xsize, int ysize,
  double scale, double offset,
  std::vector<double> coeff );
VDMask( const char *name );
VDMask();
```

The first form creates an empty matrix, with the specified dimensions; the second form initialises a matrix from a varargs list; the third form sets the matrix from a vector of coefficients; the fourth from the named file. The final form makes a mask object with no contents yet.

The varargs constructors are not wrapped in Python — use the vector constructor instead. For example:

```
m = VMask.VIMask (3, 3, 1, 0,
  [-1, -1, -1,
   -1,  8, -1,
   -1, -1, -1])
```

1.4.2 Projection functions

A set of member functions of `VIMask` provide access to the fields in the matrix:

```
int xsize() const;
int ysize() const;
int scale() const;
int offset() const;
const char *filename() const;
```

`VDMask` is the same, except that the `scale()` and `offset()` members return `double`. `VMask` allows all operations that are common to `VIMask` and `VDMask`.

1.4.3 Assignment

`VMask` defines copy and assignment with pointer-style semantics. You can write stuff like:

```
VIMask fred( "mask" );
VMask jim;

jim = fred;
```

This reads the file `mask`, noting a pointer to the mask in `fred`. It then makes `jim` also point to it, so `jim` and `fred` are sharing the same underlying matrix values.

Internally, a `VMask` object is just a pointer to a reference-counting block, which in turn holds a pointer to the underlying VIPS `MASK` type. You can therefore efficiently pass `VMask` objects to functions by value, and return `VMask` objects as function results.

1.4.4 Computing with VMask

You can use [] to get at matrix elements, numbered left-to-right, top-to-bottom. Alternatively, use () to address elements by x, y position. For example:

```
VIMask fred( "mask" );

for( int i = 0; i < fred.xsize(); i++ )
    fred[i] = 12;
```

will set the first line of the matrix to 12, and:

```
VDMask fred( "mask" );

for( int x = 0; x < fred.xsize(); x++ )
    fred(x, x) = 12.0;
```

will set the leading diagonal to 12.

See the member functions below for other operations on VMask.

1.4.5 VIMask operations

The following operations are defined for VIMask:

```
// Cast to VDMask and VImage
operator VDMask();
operator VImage();

// Build gaussian and log masks
static VIMask gauss( double, double );
static VIMask log( double, double );

// Rotate
VIMask rotate45();
VIMask rotate90();

// Transpose, invert, join and multiply
VDMask trn() ;
VDMask inv();
VDMask cat( VDMask );
VDMask mul( VDMask );
```

1.4.6 VDMask operations

The following operations are defined for VDMask:

```
// Cast to VIMask and VImage
operator VIMask();
operator VImage();

// Build gauss and log masks
static VDMask gauss( double, double );
static VDMask log( double, double );

// Rotate
VDMask rotate45();
VDMask rotate90();

// Scale to intmask
VIMask scalei();

// Transpose, invert, join and multiply
VDMask trn();
VDMask inv();
VDMask cat( VDMask );
VDMask mul( VDMask );
```

1.4.7 Output of masks

You can output masks with the usual << operator.

1.5 The VDisplay class

The VDisplay class is an abstraction over the VIPS im_col_display type which gives convenient and safe representation of VIPS display profiles.

VIPS display profiles are now obsolete. You're better off using the ICC colour management VImage member functions ICC_export() and ICC_import().

1.5.1 Constructors

There are two constructors for VDisplay:

```
VDisplay( const char *name );
VDisplay();
```

The first form initialises the display from one of the standard VIPS display types. For example:

```
VDisplay fred( "sRGB" );
VDisplay jim( "ultra2-20/2/98" );
```

Makes fred a profile for making images in sRGB format, and jim a profile representing my workstation display, as of 20/2/98. The second form of constructor makes an uninitialised display.

1.5.2 Projection functions

A set of member functions of `VDisplay` provide read and write access to the fields in the display.

```
char *name();
VDisplayType &type();
matrix &mat();
float &YCW();
float &xCW();
float &yCW();
float &YCR();
float &YCG();
float &YCB();
int &Vrwr();
int &Vrwg();
int &Vrwb();
float &Y0R();
float &Y0G();
float &Y0B();
float &gammaR();
float &gammaG();
float &gammaB();
float &B();
float &P();
```

Where `VDisplayType` is defined as:

```
enum VDisplayType {
    BARCO,
    DUMB
};
```

And `matrix` is defined as:

```
typedef float matrix[3][3];
```

For a description of all the fields in a VIPS display profile, see the manual page for `im_XYZ2RGB()`.

1.6 The `VError` class

The `VError` class is the class thrown by the VIPS C++ API when an error is detected. It is derived from `std::exception` in the usual way.

1.6.1 Constructors

There are two constructors for `VError`:

```
VError( std::string str );
VError();
```

The first form creates an error object initialised with the specified string, the last form creates an empty error object.

1.6.2 Projection functions

A function gives access to the string held by `VError`:

```
const char *what();
```

You can also send to an `ostream`.

```
std::ostream& operator<<(
    std::ostream&, const error& );
```

1.6.3 Computing with `VError`

Two member functions let you append elements to an error:

```
VError &app( std::string txt );
VError &app( const int i );
```

For example:

```
VError wombat;
int n = 12;

wombat.app( "possum: no more than " ).
app( n ).app( " elements\n" );
throw( wombat );
```

will throw a `VError` with a diagnostic of:

```
possum: no more than 12 elements
```

The member function `perror()` prints the error message to `stdout` and exits with a code of 1.

```
void perror( const char * );
void perror();
```

1.6.4 Convenience function

The convenience function `verror` creates an `VError` with the specified error string, and throws it. If you pass `""` for the string, verror uses the contents of the VIPS error buffer instead.

```
extern void verror( std::string str = "" );
```

Chapter 2

VIPS for C programmers

2.1 Introduction

This chapter explains how to call VIPS functions from C programs. It does not explain how to write new image processing operations (See §3.1 on page 33), only how to call the ones that VIPS provides. If you want to call VIPS functions from C++ programs, you can either use the interface described here or you can try out the much nicer C++ interface described in §1.1 on page 1.

See §4.1 on page 55 for an introduction to the image processing operations available in the library. Figure 2.1 on page 12 tries to show an overview of this structure.

VIPS includes a set of UNIX manual pages. Enter (for example):

```
example% man im_extract
```

to get an explanation of the `im_extract()` function.

All the comand-line vips operations will print help text too. For example:

```
example% vips im_extract
usage: vips im_extract input output
  left top width height band
where:
        input is of type "image"
        output is of type "image"
        left is of type "integer"
        top is of type "integer"
        width is of type "integer"
        height is of type "integer"
        band is of type "integer"
extract area/band, from package
  "conversion"
flags: (PIO function)
  (coordinate transformer)
  (area operation)
  (result can be cached)
vips: error calling function
im_run_command: too few arguments
```

2.2 Core C API

VIPS is built on top of several other libraries, two of which, glib and gobject, are exposed at various points in the C API.

You can read up on glib at the GTK+ website:

```
http://www.gtk.org/api
```

There's also an excellent book by Matthias Warkus, *The Official GNOME 2 Developer's Guide*, which covers the same material in a tutorial manner.

2.2.1 Startup

Before calling any VIPS function, you need to start VIPS up:

```
int im_init_world( const char *argv0 );
```

The `argv0` argument is the value of `argv[0]` your program was passed by the host operating system. VIPS uses this with `im_guess_prefix()` and `im_guess_libdir()` to try to find various VIPS data files.

If you don't call this function, VIPS will call it for you the first time you use a VIPS function. But it won't be able to get the `argv0` value for you, so it may not be able to find it's data files.

VIPS also offers the optional:

```
GOptionGroup *im_get_option_group( void );
```

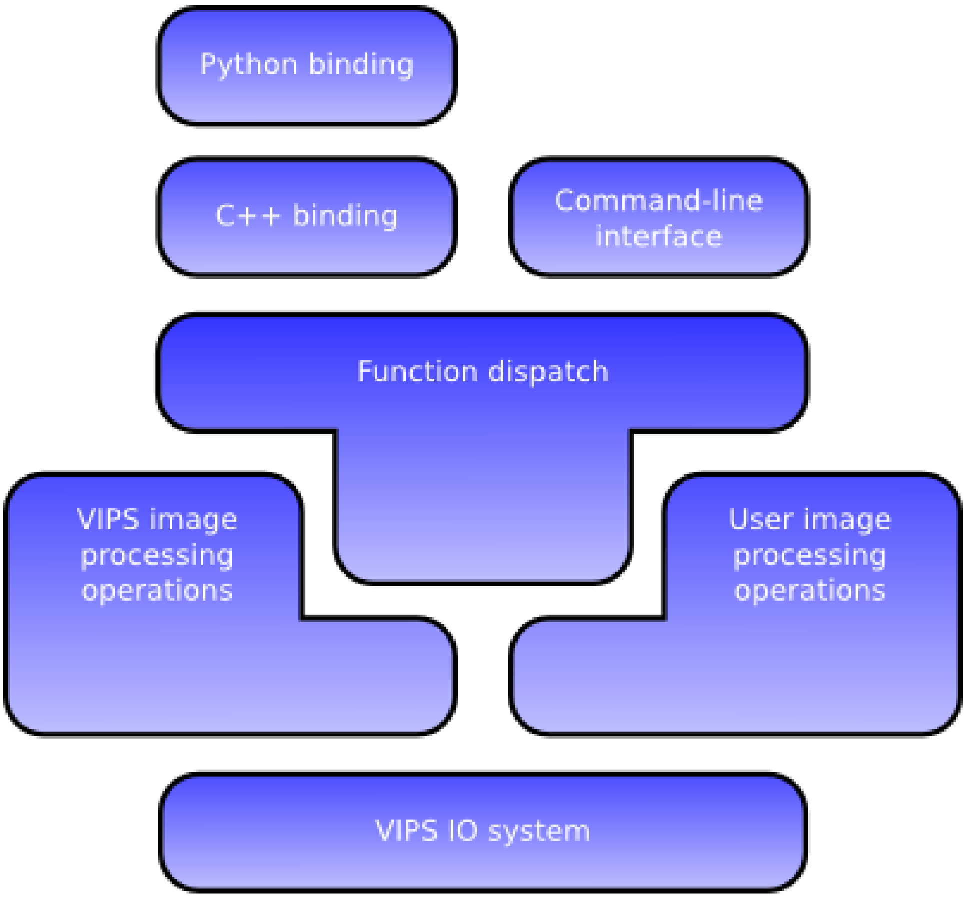

Figure 2.1: VIPS software architecture

You can use this with GOption to parse your program's command-line arguments. It adds several useful VIPS flags, including `--vips-concurrency`.

Figure 2.2 on page 14 shows both these functions in use. Again, the GOption stuff is optional and just lets VIPS add some flags to your program. You do need the `im_init_world()` though.

2.2.2 Image descriptors

The base level of the VIPS I/O system provides `IMAGE` descriptors. An image represented by a descriptor may be an image file on disc, an area of memory that has been allocated for the image, an output file, a delayed computation, and so on. Programs need (usually) only know that they have a descriptor, they do not see many of the details. Figure 2.3 on page 15 shows the definition of the `IMAGE` descriptor.

The first set of fields simply come from the image file header: see §1.2.1 on page 1 for a full description of all the fields. The next set are maintained for you by the VIPS I/O system. `filename` is the name of the file that this image came from. If you have attached an eval callback to this image, `time` points to a set of timing statistics which can be used by user-interfaces built on VIPS to provide feedback about the progress of evaluation — see §2.2.8 on page 19. Finally, if you set `kill` to non-zero, VIPS will block any pipelines which use this descriptor as an intermediate. See §2.2.12 on page 23.

The remaining fields are private and are used by VIPS for housekeeping.

2.2.3 Header fields

You can access header fields either directly (as `im->Xsize`, for example) or programmatically with `im_header_int()` and friends. For example:

```
int i;

im_header_int( im, "Xsize", &i );
```

There's also `im_header_map()` to loop over header fields, and `im_header_get_type` to test the type of fields. These functions work for image meta fields as well, see §2.2.6 on page 16.

2.2.4 Opening and closing

Descriptors are created with `im_open()`. You can also read images with the format system: see §2.4 on page 30. The two APIs are complimentary, though `im_open()` is more useful.

`im_open()` takes a file name and a string representing the mode with which the descriptor is to be opened:

```
IMAGE *im_open( const char *filename,
  const char *mode )
```

The possible values for mode are:

"r" The file is opened read-only. If you open a non-VIPS image, or a VIPS image written on a machine with a different byte ordering, `im_open()` will automatically convert it to native VIPS format. If the underlying file does not support random access (JPEG, for example), the entire file will be converted in memory.

VIPS can read images in TIFF, JPEG, PPM/PGM/-PBM, PNG and VIPS format, all in both big- and little-endian varieties. You can control the details of the conversion with extra characters embedded in the filename. For example:

```
fred = im_open( "fred.tif:2",
  "r" );
```

will read page 2 of a multi-page TIFF. See the man pages for details.

If VIPS has been built with libMagick, it can also read any of the 80 or so libMagick-supported image file formats.

"w" An `IMAGE` descriptor is created which, when written to, will write pixels to disc in the specified file.

VIPS looks at the filename suffix to determine the save format. If there is no suffix, or the filename ends in `".v"`, the image is written in VIPS native format.

If the filename ends in `".tif"` or `".tiff"`, the image is written with `im_vips2tiff()`. If the filename ends in `".jpg"`, `".jpeg"` or `".jpe"`, the image is written with `im_vips2jpg()`. If the filename ends with `".pbm"`, `".pgm"` or `".ppm"`, the image is written using

```
#include <stdio.h>
#include <vips/vips.h>

static gboolean print_stuff;

static GOptionEntry options[] = {
  { "print", 'p', 0, G_OPTION_ARG_NONE, &print_stuff,
    "print \"hello world!\"", NULL },
  { NULL }
};

int
main( int argc, char **argv )
{
  GOptionContext *context;
  GError *error = NULL;

  if( im_init_world( argv[0] ) )
    error_exit( "unable to start VIPS" );

  context = g_option_context_new( "- my program" );
  g_option_context_add_main_entries( context,
    options, "main" );
  g_option_context_add_group( context, im_get_option_group() );
  if( !g_option_context_parse( context, &argc, &argv, &error ) ) {
    if( error ) {
      fprintf( stderr, "%s\n", error->message );
      g_error_free( error );
    }
    error_exit( "try \"%s --help\"", g_get_prgname() );
  }
  g_option_context_free( context );

  if( print_stuff )
    printf( "hello, world!\n" );

  return( 0 );
}
```

Figure 2.2: Hello World for VIPS

```
typedef struct {
    /* Fields from image header.
     */
    int Xsize;              /* Pels per line */
    int Ysize;              /* Lines */
    int Bands;              /* Number of bands */
    int Bbits;              /* Bits per band */
    int BandFmt;            /* Band format */
    int Coding;             /* Coding type */
    int Type;               /* Type of file */
    float XRes;             /* Horizontal res in pels/mm */
    float YRes;             /* Vertical res in pels/mm  */
    int Length;             /* Obsolete (unused) */
    short Compression;      /* Obsolete (unused) */
    short Level;            /* Obsolete (unused) */
    int Xoffset;            /* Position of origin */
    int Yoffset;

    /* Derived fields that may be read by the user.
     */
    char *filename;         /* File name */
    im_time_t *time;        /* Timing for eval callback */
    int kill;               /* Set to non-zero to block eval */

    ... and lots of other private fields used by VIPS for
    ... housekeeping.
} IMAGE;
```

Figure 2.3: The `IMAGE` descriptor

im_vips2ppm(). If the filename ends with ".png", the image is written using im_vips2png(). Case is not considered when testing the suffix.

If you want to control the details of the conversion to the disc format (such as setting the Q factor for a JPEG, for example), you embed extra control characters in the filename. For example:

```
fred = im_open( "fred.jpg:95",
  "w" );
```

writes to fred will write a JPEG with Q 95. Again, see the man pages for the conversion functions for details.

"t" As the "w" mode, but pels written to the descriptor will be saved in a temporary memory buffer.

"p" This creates a special partial image. Partial images are used to join VIPS operations together, see §2.2.12 on page 21.

"rw" As the "r" mode, but the image is mapped into the caller's address space read-write. This mode is only provided for the use of paintbox-style applications which need to directly modify an image. Most programs should use the "w" mode for image output.

If an error occurs opening the image, im_open() calls im_error() with a string describing the cause of the error and returns NULL. im_error() has type

```
void im_error( const char *domain,
  const char *format, ... )
```

The first argument is a string giving the name of the thing that raised the error (just "im_open", for example). The format and subsequent arguments work exactly as printf(). It formats the message and appends the string formed to the error log. You can get a pointer to the error text with im_error_buffer().

```
const char *im_error_buffer()
```

Applications may display this string to give users feedback about the cause of the error. The VIPS exit function, error_exit(), prints im_error_buffer() to stderr and terminates the program with an error code of 1.

```
void error_exit( const char *format,
  ... )
```

There are other functions for handling errors: see the man page for im_error().

If the file name given to im_open() ends with ".v", VIPS looks in the same directory as the image for a file with the same name but with the extension ".desc". If present, this file is read in and a pointer to the data put in the Hist field of the descriptor. See the notes on im_updatehist() in §2.2.5.

Descriptors are closed with im_close(). It has type:

```
int im_close( IMAGE *im )
```

im_close() returns 0 on success and non-zero on error. If the descriptor represents a disc file which has been written to and whose name ends in ".v", VIPS writes the Hist field of the image descriptor to a file in the same directory whose name ends in ".desc".

2.2.5 Examples

As an example, Figure 2.2.5 on page 17 will print the width and height of an image stored on disc.

To compile this example, use:

```
cc `pkg-config vips-7.14 \
  --cflags --libs` myfunc.c
```

As a slightly more complicated example, Figure 2.2.5 on page 18 will calculate the photographic negative of an image.

2.2.6 Metadata

VIPS lets you attach arbitrary metadata to an IMAGE. For example, ICC profiles, EXIF tags, whatever you like. VIPS will efficiently propogate metadata as images are processed (usually just by copying pointers) and will automatically save and load metadata from VIPS files (see §1.2.1 on page 1).

A piece of metadata is a value and an identifying name. A set of convenience functions let you set and get int, double, string and blob. For example:

```
int im_meta_set_int( IMAGE *,
  const char *field, int );
int im_meta_get_int( IMAGE *,
  const char *field, int * );
```

```
#include <stdio.h>
#include <vips/vips.h>

int
main( int argc, char **argv )
{
  IMAGE *im;

  /* Check arguments.
   */
  if( im_init_world( argv[0] ) )
    error_exit( "unable to start VIPS" );
  if( argc != 2 )
    error_exit( "usage: %s filename", argv[0] );

  /* Open file.
   */
  if( !(im = im_open( argv[1], "r" )) )
    error_exit( "unable to open %s for input", argv[1] );

  /* Process.
   */
  printf( "width = %d, height = %d\n", im->Xsize, im->Ysize );

  /* Close.
   */
  if( im_close( im ) )
      error_exit( "unable to close %s", argv[1] );

  return( 0 );
}
```

Figure 2.4: Print width and height of an image

```
#include <stdio.h>
#include <vips/vips.h>

int
main( int argc, char **argv )
{
  IMAGE *in, *out;

  /* Check arguments.
   */
  if( im_init_world( argv[0] ) )
    error_exit( "unable to start VIPS" );
  if( argc != 3 )
    error_exit( "usage: %s infile outfile", argv[0] );

  /* Open images for read and write, invert, update the history with our
   * args, and close.
   */
  if( !(in = im_open( argv[1], "r" )) ||
    !(out = im_open( argv[2], "w" )) ||
    im_invert( in, out ) ||
    im_updatehist( out, argc, argv ) ||
    im_close( in ) ||
    im_close( out ) )
    error_exit( argv[0] );

  return( 0 );
}
```

Figure 2.5: Find photographic negative

So you can do:

```
if( im_meta_set_int( im, "poop", 42 ) )
  return( -1 );
```

to create an int called "poop", then at some later point (possibly much, much later), in an image distantly derived from im, you can use:

```
int i;

if( im_meta_get_int( im, "poop", &i ) )
  return( -1 );
```

And get the value 42 back.

You can use im_meta_set() and im_meta_get() to attach arbitrary GValue to images. See the man page for im_meta_set() for full details.

You can test for a field being present with im_meta_get_type() (you'll get G_TYPE_INT back for "poop", for example, or 0 if it is not defined for this image).

2.2.7 History

VIPS tracks the history of an image, that is, the sequence of operations which have led to the creation of an image. You can view a VIPS image's history with the header command, or with nip2's View Header menu. Whenever an application performs an action, it should append a line of shell script to the history which would perform the same action.

The call to im_updatehist() in Figure 2.2.5 on page 18 adds a line to the image history noting the invocation of this program, its arguments, and the time and date at which it was run. You may also find im_histlin() helpful. It has type:

```
void im_histlin( IMAGE *im,
  const char *fmt, ... )
```

It formats its arguments as printf() and appends the string formed to the image history.

You read an image's history with im_history_get(). It returns the entire history of an image, one action per line. No need to free the result.

```
const char *
  im_history_get( IMAGE *im );
```

2.2.8 Eval callbacks

VIPS lets you attach callbacks to image descriptors. These are functions you provide which VIPS will call when certain events occur. See §3.3.6 on page 46 for a full list.

Eval callbacks are called repeatedly during evaluation and can be used by user-interface programs to give feedback about the progress of evaluation.

2.2.9 Detailed rules for descriptors

These rules are intended to answer awkward questions.

1. You can output to a descriptor only once.

2. You can use a descriptor as an input many times.

3. You can only output to a descriptor that was opened with modes "w", "t" and "p".

4. You can only use a descriptor as input if it was opened with modes "r" or "rw".

5. If you have output to a descriptor, you may subsequently use it as an input. "w" descriptors are automatically changed to "r" descriptors.

   If the function you are passing the descriptor to uses WIO (see §2.2.12 on page 23), then "p" descriptors become "t". If the function you are passing the descriptor to uses PIO, then "p" descriptors are unchanged.

2.2.10 Automatic resource deallocation

VIPS lets you allocate resources local to an image descriptor, that is, when the descriptor is closed, all resources which were allocated local to that descriptor are automatically released for you.

Local image descriptors

VIPS provides a function which will open a new image local to an existing image. im_open_local() has type:

```
IMAGE *im_open_local( IMAGE *im,
  const char *filename,
  const char *mode )
```

```
/* Add another image to the accumulated total.
 */
static int
sum1( IMAGE *acc, IMAGE **in, int nin, IMAGE *out )
{
    IMAGE *t;

    if( nin == 0 )
        /* All done ... copy to out.
         */
        return( im_copy( acc, out ) );

    /* Make a new intermediate, and add to it..
     */
    return( !(t = im_open_local( out, "sum1:1", "p" )) ||
        im_add( acc, in[0], t ) ||
        sum1( t, in + 1, nin - 1, out ) );
}

/* Sum the array of images in[]. nin is the number of images in
 * in[], out is the descriptor we write the final image to.
 */
int
total( IMAGE **in, int nin, IMAGE *out )
{
    /* Check that we have at least one image.
     */
    if( nin <= 0 ) {
        im_error( "total", "nin should be > 0" );
        return( -1 );
    }

    /* More than 1, sum recursively.
     */
    return( sum1( in[0], in + 1, nin - 1, out ) );
}
```

Figure 2.6: Sum an array of images

It behaves exactly as im_open(), except that you do not need to close the descriptor it returns. It will be closed automatically when its parent descriptor im is closed.

Figure 2.6 on page 20 is a function which will sum an array of images. We need never close any of the (unknown) number of intermediate images which we open. They will all be closed for us by our caller, when our caller finally closes out. VIPS lets local images themselves have local images and automatically makes sure that all are closed in the correct order.

It is very important that these intermediate images are made local to out rather than in, for reasons which should become apparent in the section on combining operations below.

There's also im_open_local_array() for when you need a lot of local descriptors, see the man page.

Local memory allocation

VIPS includes a set of functions for memory allocation local to an image descriptor. The base memory allocation function is im_malloc(). It has type:

```
void *im_malloc( IMAGE *, size_t )
```

It operates exactly as the standard malloc() C library function, except that the area of memory it allocates is local to an image. If im_malloc() is unable to allocate memory, it returns NULL. If you pass NULL instead of a valid image descriptor, then im_malloc() allocates memory globally and you must free it yourself at some stage.

To free memory explicitly, use im_free():

```
int im_free( void * )
```

im_free() always returns 0, so you can use it as an argument to a callback.

Three macros make memory allocation even easier. IM_NEW() allocates a new object. You give it a descriptor and a type, and it returns a pointer to enough space to hold an object of that type. It has type:

```
type-name *IM_NEW( IMAGE *, type-name )
```

The second macro, IM_ARRAY(), is very similar, but allocates space for an array of objects. Note that, unlike the usual calloc() C library function, it does not initialise the array to zero. It has type:

```
type-name *IM_ARRAY( IMAGE *, int, type-name )
```

Finally, IM_NUMBER() returns the number of elements in an array of defined size. See the man pages for a series of examples, or see §2.3.1 on page 26.

Other local operations

The above facilities are implemented with the VIPS core function im_add_close_callback(). You can use this facility to make your own local resource allocators for other types of object — see the manual page for more help.

2.2.11 Error handling

All VIPS operations return 0 on success and non-zero on error, setting im_error(). As a consequence, when a VIPS function fails, you do not need to generate an error message — you can simply propogate the error back up to your caller. If however you detect some error yourself (for example, the bad parameter in the example above), you must call im_error() to let your caller know what the problem was.

VIPS provides two more functions for error message handling: im_warn() and im_diag(). These are intended to be used for less serious messages, as their names suggest. Currently, they simply format and print their arguments to stderr, optionally supressed by the setting of an environment variable. Future releases of VIPS may allow more sophisticated trapping of these functions to allow their text to be easily presented to the user by VIPS applications. See the manual pages.

2.2.12 Joining operations together

VIPS lets you join image processing operations together so that they behave as a single unit. Figure 2.7 on page 22 shows the definition of the function im_Lab2disp() from the VIPS library. This function converts an image in *CIE* $L^*a^*b^*$ colour space to an RGB image for a monitor. The monitor characteristics (gamma, phosphor type, etc.) are described by the im_col_display structure, see the man page for im_col_XYZ2rgb().

The special "p" mode (for partial) used to open the image descriptor used as the intermediate image in this function 'glues' the two operations together. When you use im_Lab2disp(), the two operations inside it will execute together and no extra storage is necessary for

```
int
im_Lab2disp( IMAGE *in, IMAGE *out, struct im_col_display *disp )
{
    IMAGE *t1;

    if( !(t1 = im_open_local( out, "im_Lab2disp:1", "p" )) ||
        im_Lab2XYZ( in, t1 ) ||
        im_XYZ2disp( t1, out, disp ) )
        return( -1 );

    return( 0 );
}
```

Figure 2.7: Two image-processing operations joined together

the intermediate image (`t1` in this example). This is important if you want to process images larger than the amount of RAM you have on your machine.

As an added bonus, if you have more than one CPU in your computer, the work will be automatically spread across the processors for you. You can control this parallelisation with the `IM_CONCURRENCY` environment variable and with `im_concurrency_set()`. See the man page for `im_generate()`.

How it works

When a VIPS function is asked to output to a `"p"` image descriptor, all the fields in the descriptor are set (the output image size and type are set, for example), but no image data is actually generated. Instead, the function attaches callbacks to the image descriptor which VIPS can use later to generate any piece of the output image that might be needed.

When a VIPS function is asked to output to a `"w"` or a `"t"` descriptor (a real disc file or a real memory buffer), it evaluates immediately and its evaluation in turn forces the evaluation of any earlier `"p"` images.

In the example in Figure 2.7, whether or not any pixels are really processed when `im_Lab2disp()` is called depends upon the mode in which `out` was opened. If `out` is also a partial image, then no pixels will be calculated — instead, a pipeline of VIPS operations will be constructed behind the scenes and attached to `out`.

Conversely, if `out` is a real image (that is, either `"w"` or `"t"`), then the final VIPS operation in the function (`im_XYZ2disp()`) will output the entire image to `out`, causing the earlier parts of `im_Lab2disp()` (and indeed possibly some earlier pieces of program, if `in` was also a `"p"` image) to run.

When a VIPS pipeline does finally evaluate, all of the functions in the pipeline execute together, sucking image data through the system in small pieces. As a consequence, no intermediate images are generated, large amounts of RAM are not needed, and no slow disc I/O needs to be performed.

Since VIPS partial I/O is demand-driven rather than data-driven this works even if some of the operations perform coordinate transformations. We could, for example, include a call to `im_affine()`, which performs arbitrary rotation and scaling, and everything would still work correctly.

Pitfalls with partials

To go with all of the benefits that partial image I/O brings, there are also some problems. The most serious is that you are often not quite certain when computation will happen. This can cause problems if you close an input file, thinking that it is finished with, when in fact that file has not been processed yet. Doing this results in dangling pointers and an almost certain core-dump.

You can prevent this from happening with careful use of `im_open_local()`. If you always open local to your output image, you can be sure that the input will not be closed before the output has been generated to a file or memory buffer. You do not need to be so careful with non-image arguments. VIPS functions which take extra non-image arguments (a matrix, perhaps) are careful to make their own copy of the object before returning.

Non-image output

Some VIPS functions consume images, but make no image output. `im_stats()` for example, scans an image calculating various statistical values. When you use `im_stats()`, it behaves as a data sink, sucking image data through any earlier pipeline stages.

Calculating twice

In some circumstances, the same image data can be generated twice. Figure 2.8 on page 24 is a function which finds the mean value of an image, and writes a new image in which pixels less than the mean are set to 0 and images greater than the mean are set to 255.

This seems straightforward — but consider if image `in` were a `"p"`, and represented the output of a large pipeline of operations. The call to `im_avg()` would force the evaluation of the entire pipeline, and throw it all away, keeping only the average value. The subsequent call to `im_moreconst()` will cause the pipeline to be evaluated a second time.

When designing a program, it is sensible to pay attention to these issues. It might be faster, in some cases, to output to a file before calling `im_avg()`, find the average of the disc file, and then run `im_moreconst()` from that.

Blocking computation

`IMAGE` descriptors have a flag called `kill` which can be used to block computation. If `im->kill` is set to a non-zero value, then any VIPS pipelines which use `im` as an intermediate will fail with an error message. This is useful for user-interface writers — suppose your interface is forced to close an image which many other images are using as a source of data. You can just set the `kill` flag in all of the deleted image's immediate children and prevent any dangling pointers from being followed.

Limitations

Not all VIPS operations are partial-aware. These non-partial operations use a pre-VIPS7.0 I/O scheme in which the whole of the input image has to be present at the same time. In some cases, this is because partial I/O simply makes no sense — for example, a Fourier Transform can produce no output until it has seen all of the input. `im_fwfft()` is therefore not a partial operation. In other cases, we have simply not got around to rewriting the old non-partial operation in the newer partial style.

You can mix partial and non-partial VIPS operations freely, without worrying about which type they are. The only effect will be on the time your pipeline takes to execute, and the memory requirements of the intermediate images. VIPS uses the following rules when you mix the two styles of operation:

1. When a non-partial operation is asked to output to a partial image descriptor, the `"p"` descriptor is magically transformed into a `"t"` descriptor.

2. When a non-partial operation is asked to read from a `"p"` descriptor, the `"p"` descriptor is turned into a `"t"` type, and any earlier stages in the pipeline forced to evaluate into that memory buffer.

   The non-partial operation then processes from the memory buffer.

These rules have the consequence that you may only process very large images if you only use partial operations. If you use any non-partial operations, then parts of your pipelines will fall back to old whole-image I/O and you will need to think carefully about where your intermediates should be stored.

2.3 Function dispatch and plug-ins

As image processing libraries increase in size it becomes progressively more difficult to build applications which present the operations the libbrary offers to the user. Every time a new operation is added, every user interface needs to be adapted — a job which can rapidly become unmanageable.

To address this problem VIPS includes a simple database which stores an abstract description of every image processing operation. User interfaces, rather than having special code wired into them for each operation, can simply interrogate the database and present what they find to the user.

The operation database is extensible. You can define new operations, and even new types, and add them to VIPS. These new operations will then automatically appear in all VIPS user interfaces with no extra programming effort. Plugins can extend the database at runtime:

```
int
threshold_at_mean( IMAGE *in, IMAGE *out )
{
    double mean;

    if( im_avg( in, &mean ) ||
        im_moreconst( in, out, mean ) )
        return( -1 );

    return( 0 );
}
```

Figure 2.8: Threshold an image at the mean value

when VIPS starts, it loads all the plugin in the VIPS library area.

2.3.1 Simple plugin example

As an example, consider this function:

```
#include <stdio.h>

#include <vips/vips.h>

/* The function we define. Call this
 * from other parts of your C
 * application.
 */
int
double_integer( int in )
{
  return( in * 2 );
}
```

The source for all the example code in this section is in the vips-examples package.

The first step is to make a layer over this function which will make it look like a standard VIPS function. VIPS insists on the following pattern:

- The function should be int-valued, and return 0 for success and non-zero for error. It should set `im_error()`.
- The function should take a single argument: a pointer to a `NULL`-terminated array of `im_objects`.
- Each `im_object` represents one argument to the function (either output or input) in the form specified by the corresponding entry in the function's argument descriptor.

The argument descriptor is an array of structures, each describing one argument. For this example, it is:

```
/* Describe the type of our function.
 * One input int, and one output int.
 */
static im_arg_desc arg_types[] = {
  IM_INPUT_INT( "in" ),
  IM_OUTPUT_INT( "out" )
};
```

`IM_INPUT_INT()` and `IM_OUTPUT_INT()` are macros defined in `<vips/dispatch.h>` which make argument types easy to define. Other macros available are listed in table 2.1.

The argument to the type macro is the name of the argument. These names are used by user-interface programs to provide feedback, and sometimes as variable names. The order in which you list the arguments is the order in which user-interfaces will present them to the user. You should use the following conventions when selecting names and an order for your arguments:

- Names should be entirely in lower-case and contain no special characters, apart from the digits 0-9 and the underscore character '_'.
- Names should indicate the function of the argument. For example, `im_add()` has the following argument names:

Macro	Meaning	`im_object` has type
`IM_INPUT_IMAGEVEC`	Vector of input images	`IMAGE **`
`IM_INPUT_IMAGE`	Input image	`IMAGE *`
`IM_OUTPUT_IMAGE`	Output image	`IMAGE *`
`IM_RW_IMAGE`	Read-write image	`IMAGE *`
`IM_INPUT_DOUBLE`	Input double	`double *`
`IM_INPUT_DOUBLEVEC`	Input vector of double	`im_realvec_object *`
`IM_INPUT_INTVEC`	Input vector of int	`im_intvec_object *`
`IM_OUTPUT_DOUBLE`	Output double	`double *`
`IM_INPUT_INT`	Input int	`int *`
`IM_OUTPUT_INT`	Output int	`int *`
`IM_INPUT_STRING`	Input string	`char *`
`IM_OUTPUT_STRING`	Output string	`char *`
`IM_INPUT_DISPLAY`	Input display	`im_col_display *`
`IM_OUTPUT_DISPLAY`	Output display	`im_col_display *`
`IM_OUTPUT_COMPLEX`	Output complex	`double *`
`IM_INPUT_DMASK`	Input double array	`im_mask_object *`
`IM_OUTPUT_DMASK`	Output double array to file	`im_mask_object *`
`IM_OUTPUT_DMASK_STATS`	Output double array to screen	
`IM_INPUT_IMASK`	Input int array	`im_mask_object *`
`IM_OUTPUT_IMASK`	Output int array to file	`im_mask_object *`
`IM_INPUT_GVALUE`	Input GValue	`GValue *`
`IM_OUTPUT_GVALUE`	Output GValue	`GValue *`

Table 2.1: Argument type macros

```
example% vips -help im_add
vips: args: in1 in2 out
where:
  in1 is of type "image"
  in2 is of type "image"
  out is of type "image"
add two images, from package
  "arithmetic"
flags:
  (PIO function)
  (no coordinate transformation)
  (point-to-point operation)
```

- You should order arguments with large input objects first, then output objects, then any extra arguments or options. For example, `im_extract()` has the following sequence of arguments:

```
example% vips -help im_extract
vips: args: input output left top
  width height channel
where:
  input is of type "image"
  output is of type "image"
  left is of type "integer"
  top is of type "integer"
  width is of type "integer"
  height is of type "integer"
  channel is of type "integer"
extract area/band, from package
  "conversion"
flags:
  (PIO function)
  (no coordinate transformation)
  (point-to-point operation)
```

This function sits over `double_integer()`, providing VIPS with an interface which it can call:

```
/* Call our function via a VIPS
 * im_object vector.
 */
static int
double_vec( im_object *argv )
{
  int *in = (int *) argv[0];
  int *out = (int *) argv[1];

  *out = double_integer( *in );

  /* Always succeed.
   */
  return( 0 );
}
```

Finally, these two pieces of information (the argument description and the VIPS-style function wrapper) can be gathered together into a function description.

```
/* Description of double_integer.
 */
static im_function double_desc = {
  "double_integer",
  "double an integer",
  0,
  double_vec,
  IM_NUMBER( arg_types ),
  arg_types
};
```

`IM_NUMBER()` is a macro which returns the number of elements in a static array. The `flags` field contains hints which user-interfaces can use for various optimisations. At present, the possible values are:

IM_FN_PIO This function uses the VIPS PIO system (see §3.3 on page 38).

IM_FN_TRANSFORM This the function transforms coordinates.

IM_FN_PTOP This is a point-to-point operation, that is, it can be replaced with a look-up table.

IM_FN_NOCACHE This operation has side effects and should not be cached. Useful for video grabbers, for example.

This function description now needs to be added to the VIPS function database. VIPS groups sets of related functions together in packages. There is only a single function in this example, so we can just write:

```
/* Group up all the functions in this
 * file.
 */
static im_function
  *function_list[] = {
  &double_desc
};
```

```
/* Define the package_table symbol.
 * This is what VIPS looks for when
 * loading the plugin.
 */
im_package package_table = {
  "example",
  IM_NUMBER( function_list ),
  function_list
};
```

The package has to be named `package_table`, and has to be exported from the file (that is, not a static). VIPS looks for a symbol of this name when it opens your object file.

This file needs to be made into a dynamically loadable object. On my machine, I can do this with:

```
example% gcc -fPIC -DPIC -c
  `pkg-config vips-7.12 --cflags`
  plug.c -o plug.o
example% gcc -shared plug.o
  -o double.plg
```

You can now use `double.plg` with any of the VIPS applications which support function dispatch. For example:

```
example% vips -plugin double.plg \
  double_integer 12
24
example%
```

When VIPS starts up, it looks for a directory in the library directory called `vips-`, with the vips major and minor versions numbers as extensions, and loads all files in there with the suffix `.plg`. So for example, on my machine, the plugin directory is `/usr/lib/vips-7.16` and any plugins in that directory are automatically loaded into any VIPS programs on startup.

2.3.2 A more complicated example

This section lists the source for `im_extract()`'s function description. Almost all functions in the VIPS library have descriptors — if you are not sure how to write a description, it's usually easiest to copy one from a similar function in the library.

```
/* Args to im_extract.
 */
static im_arg_desc
  extract_args[] = {
  IM_INPUT_IMAGE( "input" ),
  IM_OUTPUT_IMAGE( "output" ),
  IM_INPUT_INT( "left" ),
  IM_INPUT_INT( "top" ),
  IM_INPUT_INT( "width" ),
  IM_INPUT_INT( "height" ),
  IM_INPUT_INT( "channel" )
};

/* Call im_extract via arg vector.
 */
static int
extract_vec( im_object *argv )
{
  IMAGE_BOX box;

  box.xstart = *((int *) argv[2]);
  box.ystart = *((int *) argv[3]);
  box.xsize = *((int *) argv[4]);
  box.ysize = *((int *) argv[5]);
  box.chsel = *((int *) argv[6]);

  return( im_extract(
    argv[0], argv[1], &box ) );
}

/* Description of im_extract.
 */
static im_function
  extract_desc = {
  "im_extract",
  "extract area/band",
  IM_FN_PIO | IM_FN_TRANSFORM,
  extract_vec,
  NUMBER( extract_args ),
  extract_args
};
```

2.3.3 Adding new types

The VIPS type mechanism is extensible. User plug-ins can add new types and user-interfaces can (to a certain extent) provide interfaces to these user-defined types.

Here is the definition of `im_arg_desc`:

```
/* Describe a VIPS command argument.
```

```
 */
typedef struct {
  char *name;
  im_type_desc *desc;
  im_print_obj_fn print;
} im_arg_desc;
```

The `name` field is the argument name above. The `desc` field points to a structure defining the argument type, and the `print` field is an (optionally `NULL`) pointer to a function which VIPS will call for output arguments after your function successfully completes and before the object is destroyed. It can be used to print results to the terminal, or to copy results into a user-interface layer.

```
/* Success on an argument. This is
 * called if the image processing
 * function succeeds and should be
 * used to (for example) print
 * output.
 */
typedef int (*im_print_obj_fn)
  ( im_object obj );
```

`im_type_desc` is defined as:

```
/* Describe a VIPS type.
 */
typedef struct {
  im_arg_type type;
  int size;
  im_type_flags flags;
  im_init_obj_fn init;
  im_dest_obj_fn dest;
} im_type_desc;
```

Where `im_arg_type` is defined as

```
/* Type names. You may define your
 * own, but if you use one of these,
 * then you should use the built-in
 * VIPS type converters.
 */
#define IM_TYPE_IMAGEVEC "imagevec"
#define IM_TYPE_DOUBLEVEC "doublevec"
#define IM_TYPE_INTVEC "intvec"
#define IM_TYPE_DOUBLE "double"
#define IM_TYPE_INT "integer"
#define IM_TYPE_COMPLEX "complex"
#define IM_TYPE_STRING "string"
#define IM_TYPE_IMASK "intmask"
#define IM_TYPE_DMASK "doublemask"
#define IM_TYPE_IMAGE "image"
#define IM_TYPE_DISPLAY "display"
#define IM_TYPE_GVALUE "gvalue"
typedef char *im_arg_type;
```

In other words, it's just a string. When you add a new type, you just need to choose a new unique string to name it. Be aware that the string is printed to the user by various parts of VIPS, and so needs to be "human-readable". The flags are:

```
/* These bits are ored together to
 * make the flags in a type
 * descriptor.
 *
 * IM_TYPE_OUTPUT: set to indicate
 * output, otherwise input.
 *
 * IM_TYPE_ARG: Two ways of making
 * an im_object --- with and without
 * a command-line string to help you
 * along. Arguments with a string
 * are thing like IMAGE descriptors,
 * which require a filename to
 * initialise. Arguments without are
 * things like output numbers, where
 * making the object simply involves
 * allocating storage.
 */

typedef enum {
  IM_TYPE_OUTPUT = 0x1,
  IM_TYPE_ARG = 0x2
} im_type_flags;
```

And the `init` and `destroy` functions are:

```
/* Initialise and destroy objects.
 * The "str" argument to the init
 * function will not be supplied
 * if this is not an ARG type.
 */
typedef int (*im_init_obj_fn)
  ( im_object *obj, char *str );
typedef int (*im_dest_obj_fn)
  ( im_object obj );
```

As an example, here is the definition for a new type of unsigned integers. First, we need to define the `init` and `print` functions. These transform objects of the type to and from string representation.

```
/* Init function for unsigned int
 * input.
 */
static int
uint_init( im_object *obj, char *str )
{
  unsigned int *i = (int *) *obj;

  if( sscanf( str, "%d", i ) != 1 ||
    *i < 0 ) {
    im_error( "uint_init",
      "bad format" );
    return( -1 );
  }

  return( 0 );
}

/* Print function for unsigned int
 * output.
 */
static int
uint_print( im_object obj )
{
  unsigned int *i =
    (unsigned int *) obj;

  printf( "%d\n", (int) *i );

  return( 0 );
}
```

Now we can define the type itself. We make two of these — one for unsigned int used as input, and one for output.

```
/* Name our type.
 */
#define TYPE_UINT "uint"

/* Input unsigned int type.
 */
static im_type_desc input_uint = {
  TYPE_UINT,          /* Its an int */
  sizeof( unsigned int ),/* Memory */
  IM_TYPE_ARG,        /* Needs arg */
  uint_init,          /* Init */
  NULL                /* Destroy */
};

/* Output unsigned int type.
 */
static im_type_desc output_uint = {
  TYPE_UINT,          /* It's an int */
  sizeof( unsigned int ),/* Memory */
  IM_TYPE_OUTPUT,     /* It's output */
  NULL,               /* Init */
  NULL                /* Destroy */
};
```

Finally, we can define two macros to make structures of type `im_arg_desc` for us.

```
#define INPUT_UINT( S ) \
  { S, &input_uint, NULL }
#define OUTPUT_UINT( S ) \
  { S, &output_uint, uint_print }
```

For more examples, see the definitions for the built-in VIPS types.

2.3.4 Using function dispatch in your application

VIPS provides a set of functions for adding new image processing functions to the VIPS function database, finding functions by name, and calling functions. See the manual pages for full details.

Adding and removing functions

```
im_package *im_load_plugin(
  const char *name );
```

This function opens the named file, searches it for a symbol named `package_table`, and adds any functions it finds to the VIPS function database. When you search for a function, any plug-ins are searched first, so you can override standard VIPS function with your own code.

The function returns a pointer to the package it added, or `NULL` on error.

```
int im_close_plugins( void )
```

This function closes all plug-ins, removing then from the VIPS function database. It returns non-zero on error.

Searching the function database

```
void *im_map_packages(
  im_list_map_fn fn, void *a )
```

This function applies the argument function `fn` to every package in the database, starting with the most recently added package. As with `im_list_map()`, the argument function should return `NULL` to continue searching, or non-`NULL` to terminate the search early. `im_map_packages()` returns `NULL` if `fn` returned `NULL` for all arguments. The extra argument `a` is carried around by VIPS for your use.

For example, this fragment of code prints the names of all loaded packages to `fd`:

```
static void *
print_package_name( im_package *pack,
  FILE *fp )
{
  (void) fprintf( fp,
    "package: \"%s\"\n",
    pack->name );

  /* Continue search.
   */
  return( NULL );
}

static void
print_packages( FILE *fp )
{
  (void) im_map_packages(
    (im_list_map_fn)
    print_package_name, fp );
}
```

VIPS defines three convenience functions based on `im_map_packages()` which simplify searching for specific functions:

```
im_function *
  im_find_function( char *name )
im_package *
  im_find_package( char *name )
im_package *
  im_package_of_function( char *name )
```

Building argument structures and running commands

```
int im_free_vargv( im_function *fn,
  im_object *vargv )
int im_allocate_vargv(
  im_function *fn,
  im_object *vargv )
```

These two functions allocate space for and free VIPS argument lists. The allocate function simply calls `im_malloc()` to allocate any store that the types require (and also initializes it to zero). The free function just calls `im_free()` for any storage that was allocated.

Note that neither of these functions calls the `init`, `dest` or `print` functions for the types — that's up to you.

```
int im_run_command( char *name,
  int argc, char **argv )
```

This function does everything. In effect,

```
im_run_command( "im_invert", 2,
  { "fred.v", "fred2.v", NULL } )
```

is exactly equivalent to

```
system( "vips im_invert fred.v "
  "fred2.v" )
```

but no process is forked.

2.4 Image formats

VIPS has a simple system for adding support for new image file formats. You can register a new format and it will automatically be supported by all the VIPS interfaces. You can ask VIPS to find a format to load a file with, or to select a image file writer based on a filename. Convenience functions copy a file to an `IMAGE`, or an `IMAGE` to a file.

This is a parallel API to `im_open()`, see §2.2.4 on page 13. The format system is useful for images which are large or slow to open, because you pass a descriptor to write to and so control how and where the decompressed image is held. `im_open()` is useful for images in formats which can be directly read from disc, since you will avoid a copy operation and can directly control the disc file. The inplace operations (see §4.2.8 on page 63), for example, will only work directly on disc images if you use `im_open()`.

2.4.1 How a format is represented

See the man page for `im_format` for full details, but briefly, an image format consists of the following items:

- A name, a name that can be shows to the user, and a list of possible filename suffixes (`.tif`, for example)
- A function which tests for a file being in that format, a function which loads just the header of the file (that is, it reads properties like width and height and does not read any pixel data) and a function which loads the pixel data
- A function which will write an IMAGE to a file in the format
- And finally a function which examines a file in the format and returns flags indicating how VIPS should deal with the file. The only flag in the current version is one indicating that the file can be opened lazily

2.4.2 The format table

VIPS keeps a table of known formats, sorted by insert order and priority. You register new formats with `im_format_register()` and, optionally, unregister with `im_format_unregister()`. You can call these operations from a plugin's init function.

Any of the functions may be left NULL and VIPS will try to make do with what you do supply. Of course a format with all functions as NULL will not be very useful.

The priority system is useful if a file can be read by several possible format loaders. For example, the libMagick loader can read TIFF files, but not as well as VIPS' native TIFF reader. To make sure the VIPS TIFF reader is tried first, the libMagick format is given a low priority. Most of the time, you won't need this.

A switch to the `vips` command-line program is handy for listing the supported formats. Try:

```
vips --list formats
```

As an example, Figure 2.9 on page 32 shows how to register a new format in a plugin.

2.4.3 Finding a format

You can loop over the format table in order with `im_format_map()`. Like all the map functions in VIPS, this take a function and applies it to every element in the table until it returns non-zero, or until the table has been all covered.

You find an `im_format_t` to use to open a file with `im_format_for_file()`. This searches the VIPS format table and returns the first format whose test function returns true, setting an error message and returning NULL if no format is found.

You find a format to write a file with `im_format_for_name()`. This returns the first format with a save function whose suffix list matches the suffix of the supplied filename.

2.4.4 Convenience functions

A pair of convenience functions, `im_format_write()` and `im_format_read()`, will copy an image to and from disc using the appropriate format.

```
static const char *my_suffs[] = { ".me", NULL };

static int
is_myformat( const char *filename )
{
  unsigned char buf[2];

  if( im__get_bytes( filename, buf, 2 ) &&
    (int) buf[0] == 0xff &&
    (int) buf[1] == 0xd8 )
    return( 1 );

  return( 0 );
}

char *
g_module_check_init( GModule *self )
{
  im_format_t *format;

  format = im_format_register( "myformat",
    _( "My image format" ),
    my_suffs,
    is_myformat,
    read_myformat_header,
    read_myformat_image,
    write_myformat,
    NULL
  );
  im_format_set_priority( format, 100 );
}
```

Figure 2.9: Registering a format in a plugin

Chapter 3

Writing VIPS operations

3.1 Introduction

This chapter explains how to write image processing operations using the VIPS image I/O (input-output) system. For background, you should probably take a look at §2.1 on page 11. This is supposed to be a tutorial, if you need detailed information on any particular function, use the on-line UNIX manual pages.

3.1.1 Why use VIPS?

If you use the VIPS image I/O system, you get a number of benefits:

Threading If your computer has more than one CPU, the VIPS I/O system will automatically split your image processing operation into separate threads (provided you use PIO, see below). You should get an approximately linear speed-up as you add more CPUs.

Pipelining Provided you use PIO (again, see below), VIPS can automatically join operations together. A sequence of image processing operations will all execute together, with image data flowing through the processing pipeline in small pieces. This makes it possible to perform complex processing on very large images with no need to worry about storage management.

Composition Because VIPS can efficiently compose image processing operations, you can implement your new operation in small, reusable, easy-to-understand pieces. VIPS already has a lot of these: many new operations can be implemented by simply composing existing operations.

Large files Provided you use PIO and as long as the underlying OS supports large files (that is, files larger than 2GB), VIPS operations can work on files larger than can be addressed with 32 bits on a plain 32-bit machine. VIPS operations only see 32 bit addresses; the VIPS I/O system transparently maps these to 64 bit operations for I/O. Large file support is included on most unixes after about 1998.

Abstraction VIPS operations see only arrays of numbers in native format. Details of representation (big/little endian, VIPS/TIFF/JPEG file format, etc.) are hidden from you.

Interfaces Once you have your image processing operation implemented, it automatically appears in all of the VIPS interfaces. VIPS comes with a GUI (`nip2`), a UNIX command-line interface (`vips`) and a C++ and Python API.

Portability VIPS operations can be compiled on most unixes, Mac OS X and Windows NT, 2000 and XP without modification. Mostly.

3.1.2 I/O styles

The I/O system supports three styles of input-output.

Whole-image I/O (WIO) This style is a largely a left-over from VIPS 6.x. WIO image-processing operations have all of the input image given to them in a large memory array. They can read any of the input pels at will with simple pointer arithmetic.

Partial-image I/O (PIO) In this style operations only have a small part of the input image available to them at any time. When PIO operations are joined together into a pipeline, images flow through them in small pieces, with all the operations in a pipeline executing at the same time.

In-place The third style allows pels to be read and written anywhere in the image at any time, and is used by the VIPS in-place operations, such as `im_fastline()`. You should only use it for operations which would just be impossibly inefficient to write with either of the other two styles.

WIO operations are easy to program, but slow and inflexible when images become large. PIO operations are harder to program, but scale well as images become larger, and are automatically parallelized by the VIPS I/O system.

If you can face it, and if your algorithm can be expressed in this way, you should write your operations using PIO. Whichever you choose, applications which call your operation will see no difference, except in execution speed.

If your image processing operation performs no coordinate transformations, that is, if your output image is the same size as your input image or images, and if each output pixel depends only upon the pixel at the corresponding position in the input images, then you can use the `im_wrapone()` and `im_wrapmany()` operations. These take a simple buffer-processing operation supplied by you and wrap it up as a full-blown PIO operation. See §3.3.1 on page 38.

3.1.3 What's new in this version

The VIPS API is mostly unaltered since 7.3, so there are not many major changes. I've just reworked the text, reformatted, fixed a few typos, and changed the dates.

VIPS has acquired some crud over the years. We are planning to clean all this stuff up at some stage (and break backwards-compatibility). Maybe for VIPS 8 :-(

3.2 Programming WIO operations

WIO is the style for you if you want ease of programming, or if your algorithm must have the whole of the input image available at the same time. For example, a Fourier transform operation is unable to produce any output until it has seen the whole of the input image.

3.2.1 Input from an image

In WIO input, the whole of the image data is made available to the program via the `data` field of the descriptor. To make an image ready for reading in this style, programs should call `im_incheck()`:

```
int im_incheck( IMAGE *im )
```

If it succeeds, it returns 0, if it fails, it returns non-zero and sets `im_error()`. On success, VIPS guarantees that all of the user-accessible fields in the descriptor contain valid data, and that all of the image data may be read by simply reading from the `data` field (see below for an example). This will only work for images less than about 2GB in size.

VIPS has some simple macros to help address calculations on images:

```
int IM_IMAGE_SIZEOF_ELEMENT( IMAGE * )
int IM_IMAGE_SIZEOF_PEL( IMAGE * )
int IM_IMAGE_SIZEOF_LINE( IMAGE * )
int IM_IMAGE_N_ELEMENTS( IMAGE * )
char *IM_IMAGE_ADDR( IMAGE *,
  int x, int y )
```

These macros calculate `sizeof()` a band element, a pel and a horizontal line of pels. `IM_IMAGE_N_ELEMENTS` returns the number of band elements across an image. `IM_IMAGE_ADDR` calculates the address of a pixel in an image. If `DEBUG` is defined, it does bounds checking too.

Figure 3.1 on page 35 is a simple WIO operation which calculates the average of an unsigned char image. It will work for any size image, with any number of bands. See §3.2.3 on page 36 for techniques for making operations which will work for any image type. This operation might be called from an application with:

```
#include <stdio.h>
#include <stdlib.h>

#include <vips/vips.h>

void
find_average( char *name )
{
  IMAGE *im;
  double avg;

  if( !(im = im_open( name, "r" )) ||
```

```
#include <stdio.h>
#include <stdlib.h>

#include <vips/vips.h>

int
average( IMAGE *im, double *out )
{
  int x, y;
  long total;

  /* Prepare for reading.
   */
  if( im_incheck( im ) )
    return( -1 );

  /* Check that this is the kind of image we can process.
   */
  if( im->BandFmt != IM_BANDFMT_UCHAR ||
    im->Coding != IM_CODING_NONE ) {
    im_error( "average", "uncoded uchar images only" );
    return( -1 );
  }

  /* Loop over the image, summing pixels.
   */
  total = 0;
  for( y = 0; y < im->Ysize; y++ ) {
    unsigned char *p = (unsigned char *) IM_IMAGE_ADDR( im, 0, y );

    for( x = 0; x < IM_IMAGE_N_ELEMENTS( im ); x++ )
      total += p[x];
  }

  /* Calculate average.
   */
  *out = (double) total /
    (IM_IMAGE_N_ELEMENTS( im ) * im->Ysize));

  /* Success!
   */
  return( 0 );
}
```

Figure 3.1: Find average of image

```
    average( im, &avg ) ||
    im_close( im ) )
    error_exit( "failure!" );

  printf( "Average of \"%s\" is %G\n",
    name, avg );
}
```

When you write an image processing operation, you can test it by writing a VIPS function descriptor and calling it from the `vips` universal main program, or from the `nip2` interface. See §2.1 on page 11.

3.2.2 Output to an image

Before attempting WIO output, programs should call `im_outcheck()`. It has type:

```
int im_outcheck( IMAGE *im )
```

If `im_outcheck()` succeeds, VIPS guarantees that WIO output is sensible.

Programs should then set fields in the output descriptor to describe the sort of image they wish to write (size, type, and so on) and call `im_setupout()`. It has type:

```
int im_setupout( IMAGE *im )
```

`im_setupout()` creates the output file or memory buffer, using the size and type fields that were filled in by the program between the calls to `im_outcheck()` and `im_setupout()`, and gets it ready for writing.

Pels are written with `im_writeline()`. This takes a y position (pel (0,0) is in the top-left-hand corner of the image), a descriptor and a pointer to a line of pels. It has type:

```
int im_writeline( int y,
  IMAGE *im, unsigned char *pels )
```

Two convenience functions are available to make this process slightly easier. `im_iocheck()` is useful for programs which take one input image and produce one image output. It simply calls `im_incheck()` and `im_outcheck()`. It has type:

```
int im_iocheck( IMAGE *in, IMAGE *out )
```

The second convenience function copies the fields describing size, type, metadata and history from one image descriptor to another. It is useful when the output image will be similar in size and type to the input image. It has type:

```
int im_cp_desc( IMAGE *out, IMAGE *in )
```

There's also `im_cp_descv()`, see the man page.

Figure 3.2 on page 37 is a WIO VIPS operation which finds the photographic negative of an unsigned char image. See §2.2.10 on page 21 for an explanation of `IM_ARRAY`. This operation might be called from an application with:

```
#include <stdio.h>
#include <stdlib.h>

#include <vips/vips.h>

void
find_negative( char *inn, char *outn )
{
  IMAGE *in, *out;

  if( !(in = im_open( inn, "r" )) ||
    !(out = im_open( outn, "w" )) ||
    invert( in, out ) ||
    im_updatehist( out, "invert" ) ||
    im_close( in ) ||
    im_close( out ) )
    error_exit( "failure!" );
}
```

See §2.2.7 on page 19 for an explanation of the call to `im_updatehist()`.

3.2.3 Polymorphism

Most image processing operations in the VIPS library can operate on images of any type (`IM_BANDFMT_UCHAR`, as in our examples above, also `IM_BANDFMT_UINT` etc.). This is usually implemented with code replication: the operation contains loops for processing every kind of image, and when called, invokes the appropriate loop for the image it is given.

As an example, figure 3.3 calculates `exp()` for every pel in an image. If the input image is `double`, we write `double` output. If it is any other non-complex type, we

```
#include <stdio.h>
#include <stdlib.h>

#include <vips/vips.h>
#include <vips/util.h>

int
invert( IMAGE *in, IMAGE *out )
{
  int x, y;
  unsigned char *buffer;

  /* Check images.
   */
  if( im_iocheck( in, out ) )
    return( -1 );
  if( in->BandFmt != IM_BANDFMT_UCHAR || in->Coding != IM_CODING_NONE ) {
    im_error( "invert", "uncoded uchar images only" );
    return( -1 );
  }

  /* Make output image.
   */
  if( im_cp_desc( out, in ) )
    return( -1 );
  if( im_setupout( out ) )
    return( -1 );

  /* Allocate a line buffer and make sure it will be freed correctly.
   */
  if( !(buffer = IM_ARRAY( out,
    IM_IMAGE_SIZEOF_LINE( in ), unsigned char )) )
    return( -1 );

  /* Loop over the image!
   */
  for( y = 0; y < in->Ysize; y++ ) {
    unsigned char *p = (unsigned char *) IM_IMAGE_ADDR( in, 0, y );

    for( x = 0; x < IM_IMAGE_N_ELEMENTS( in ); x++ )
      buffer[x] = 255 - p[x];
    if( im_writeline( y, out, buffer ) )
      return( -1 );
  }

  return( 0 );
}
```

Figure 3.2: Invert an image

write `float`. If it is complex, we flag an error (`exp()` of a complex number is fiddly). The example uses an image type predicate, `im_iscomplex()`. There are a number of these predicate functions, see the manual page.

3.3 Programming PIO functions

The VIPS PIO system has a number of advantages over WIO, as summarised in the introduction. On the other hand, they are a bit more complicated.

3.3.1 Easy PIO with `im_wrapone()` and `im_wrapmany()`

PIO is a very general image IO system, and because of this flexibility, can be complicated to program. As a convenience, VIPS offers an easy-to-use layer over PIO with the funtions `im_wrapone()` and `im_wrapmany()`.

If your image processing function is uninterested in coordinates, that is, if your input and output images are the same size, and each output pixel depends only upon the value of the corresponding pixel in the input image or images, then these functions are for you.

Consider the `invert()` function of figure 3.2. First, we have to write the core of this as a buffer-processing function:

```
#include <stdio.h>
#include <stdlib.h>

#include <vips/vips.h>

/* p points to a buffer of pixels which
 * need inverting, q points to the buffer
 * we should write the result to, and n
 * is the number of pels present.
 */
static void
invert_buffer( unsigned char *p,
   unsigned char *q, int n )
{
   int i;

   for( i = 0; i < n; i++ )
      q[i] = 255 - p[i];
}
```

Now we have to wrap up this very primitive expression of the invert operation as a PIO function. We use `im_wrapone()` to do this. It has type:

```
int
im_wrapone( IMAGE *in, IMAGE *out,
   im_wrapone_fn fn, void *a, void *b )
```

where:

```
void
(*im_wrapone_fn)(void *in, void *out,
   int n, void *a, void *b )
```

almost the same type as our buffer-processing function above. The values `a` and `b` are carried around by VIPS for whatever use you fancy. `invert()` can now be written as:

```
int
invert( IMAGE *in, IMAGE *out )
{
   /* Check parameters.
    */
   if( in->BandFmt != IM_BANDFMT_UCHAR ||
      in->Bands != 1 ||
      in->Coding != IM_CODING_NONE ) {
      im_error( "invert", "bad image" );
      return( -1 );
   }

   /* Set fields in output image.
    */
   if( im_cp_desc( out, in ) )
      return( -1 );

   /* Process! We don't use either of the
    * user parameters in this function,
    * so leave them as NULL.
    */
   if( im_wrapone( in, out,
      (im_wrapone_fn) invert_buffer,
      NULL, NULL ) )
      return( -1 );

   return( 0 );
}
```

And that's all there is to it. This function will have all of the desirable properties of PIO functions, while being

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <vips/vips.h>
#include <vips/util.h>

/* Exponential transform.
 */
int
exptra( IMAGE *in, IMAGE *out )
{
  int x, y;
  unsigned char *buffer;

  /* Check descriptors.
   */
  if( im_iocheck( in, out ) )
    return( -1 );
  if( in->Coding != IM_CODING_NONE || im_iscomplex( in ) ) {
    im_error( "exptra", "uncoded non-complex only" );
    return( -1 );
  }

  /* Make output image.
   */
  if( im_cp_desc( out, in ) )
    return( -1 );
  if( in->BandFmt != IM_BANDFMT_DOUBLE )
    out->BandFmt = IM_BANDFMT_FLOAT;
  if( im_setupout( out ) )
    return( -1 );
```

Figure 3.3: Calculate `exp()` for an image

```
  /* Allocate a line buffer.
   */
  if( !(buffer = IM_ARRAY( out, IM_IMAGE_SIZEOF_LINE( in ), unsigned char )) )
    return( -1 );

/* Our inner loop, parameterised for both the input and output
 * types. Note the use of `\', since macros have to be all on
 * one line.
 */
#define loop(IN, OUT) { \
  for( y = 0; y < in->Ysize; y++ ) { \
    IN *p = (IN *) IM_IMAGE_ADDR( in, 0, y ); \
    OUT *q = (OUT *) buffer; \
    \
    for( x = 0; x < IM_IMAGE_N_ELEMENTS( in ); x++ ) \
      q[x] = exp( p[x] ); \
    if( im_writeline( y, out, buffer ) ) \
      return( -1 ); \
   } \
}

  /* Switch for all the types we can handle.
   */
  switch( in->BandFmt ) {
    case IM_BANDFMT_UCHAR: loop( unsigned char, float ); break;
    case IM_BANDFMT_CHAR:  loop( char, float ); break;
    case IM_BANDFMT_USHORT:loop( unsigned short, float ); break;
    case IM_BANDFMT_SHORT: loop( short, float ); break;
    case IM_BANDFMT_UINT:  loop( unsigned int, float ); break;
    case IM_BANDFMT_INT:   loop( int, float ); break;
    case IM_BANDFMT_FLOAT: loop( float, float ); break;
    case IM_BANDFMT_DOUBLE:loop( double, double ); break;
    default:
      im_error( "exptra", "internal error" );
      return( -1 );
  }

  /* Success.
   */
  return( 0 );
}
```

Figure 3.4: Calculate `exp()` for an image (cont)

as easy to program as the WIO `invert()` earlier in this chapter.

This version of `invert()` is not very general: it will only accept one-band unsigned char images. It is easy to modify for n-band images:

```
/* As before, but use one of the user
 * parameters to pass in the number of
 * bands in the image.
 */
static void
invert_buffer( unsigned char *p,
   unsigned char *q, int n,
   IMAGE *in )
{
   int i;
   int sz = n * in->Bands;

   for( i = 0; i < sz; i++ )
      q[i] = 255 - p[i];
}
```

We must also modify `invert()`:

```
int
invert( IMAGE *in, IMAGE *out )
{
   /* Check parameters.
    */
   if( in->BandFmt != IM_BANDFMT_UCHAR ||
      in->Coding != IM_CODING_NONE ) {
      im_error( "invert", "bad image" );
      return( -1 );
   }

   /* Set fields in output image.
    */
   if( im_cp_desc( out, in ) )
      return( -1 );

   /* Process! The first user-parameter
    * is the number of bands involved.
    */
   if( im_wrapone( in, out,
      (im_wrapone_fn)invert_buffer,
      in, NULL ) )
      return( -1 );

   return( 0 );
}
```

There are two significant hidden traps here. First, inside the buffer processing functions, you may only read the contents of the user parameters `a` and `b`, you may not write to them. This is because on a multi-CPU machine, several copies of your buffer-processing functions will be run in parallel — if they all write to the same place, there will be complete confusion. If you need writeable parameters (for example, to count and report overflows), you can't use `im_wrapone()`, you'll have to use the PIO system in all its gory detail, see below.

Secondly, your buffer processing function may not be called immediately. VIPS may decide to delay evaluation of your operation until long after the call to `invert()` has returned. As a result, care is needed to ensure that you never read anything in your buffer-processing function that may have been freed. The best way to ensure this is to use the local resource allocators, such as `im_open_local()` and `im_malloc()`. This issue is discussed at length in the sections below, and in §2.1 on page 11.

`im_wrapone()` is for operations which take exactly one input image. VIPS provides a second function, `im_wrapmany()`, which works for any number of input images. The type of `im_wrapmany()` is slightly different:

```
int
im_wrapmany( IMAGE **in, IMAGE *out,
   im_wrapmany_fn fn, void *a, void *b )

void
(*im_wrapmany_fn)( void **in, void *out,
   int n, void *a, void *b )
```

`im_wrapmany()` takes a `NULL`-terminated array of input images, and creates a `NULL`-terminated array of buffers for the use of your buffer processing function. A function to add two `IM_BANDFMT_UCHAR` images to make a `IM_BANDFMT_UCHAR` image might be written as:

```
static void
add_buffer( unsigned char **in,
   unsigned short *out, int n,
   IMAGE *in )
{
   int i;
   int sz = n * in->Bands;
   unsigned char *p1 = in[0];
```

```
   unsigned char *p2 = in[1];

   for( i = 0; i < sz; i++ )
      out[i] = p1[i] + p2[i];
}
```

This can be made into a PIO function with:

```
int
add_uchar( IMAGE *i1, IMAGE *i2,
   IMAGE *out )
{
   IMAGE *invec[3];

   /* Check parameters. We don't need to
    * check that i1 and i2 are the same
    * size, im_wrapmany() does that for
    * us.
    */
   if( i1->BandFmt != IM_BANDFMT_UCHAR ||
      i1->Coding != IM_CODING_NONE ||
      i2->BandFmt != IM_BANDFMT_UCHAR ||
      i2->Coding != IM_CODING_NONE ||
      i1->Bands != i2->Bands ) {
      im_error( "add_uchar", "bad in" );
      return( -1 );
   }

   /* Set fields in output image. As
    * input image, but we want a USHORT.
    */
   if( im_cp_desc( out, i1 ) )
      return( -1 );
   out->BandFmt = IM_BANDFMT_USHORT;
   out->Bbits = IM_BBITS_SHORT;

   /* Process! The first user-parameter
    * is the number of bands involved.
    * invec is a NULL-terminated array of
    * input images.
    */
   invec[0] = i1; invec[1] = i2;
   invec[2] = NULL;
   if( im_wrapmany( invec, out,
      (im_wrapone_fn)add_buffer,
      i1, NULL ) )
      return( -1 );

   return( 0 );
}
```

3.3.2 Region descriptors

Regions are the next layer of abstraction above image descriptors. A region is a small part of an image, held in memory ready for processing. A region is defined as:

```
typedef struct {
   Rect valid;
   IMAGE *im;

   ... and some other private fields,
   ... used by VIPS for housekeeping
} REGION;
```

where `valid` holds the sub-area of image `im` that this region represents, and `Rect` is defined as:

```
typedef struct {
   int left, top;
   int width, height;
} Rect;
```

two macros are available for `Rect` calculations:

```
int IM_RECT_RIGHT( Rect *r )
int IM_RECT_BOTTOM( Rect *r )
```

where `IM_RECT_RIGHT()` returns `left` + `width`, and `IM_RECT_BOTTOM()` returns `top` + `height`. A small library of C functions are also available for `Rect` algebra, see the manual pages for `im_rect_intersectrect()`.

Regions are created with `im_region_create()`. This has type:

```
REGION *im_region_create( IMAGE *im )
```

`im_region_create()` returns a pointer to a new region structure, or `NULL` on error. Regions returned by `im_region_create()` are blank — they contain no image data and cannot be read from or written to. See the next couple of sections for calls to fill regions with data.

Regions are destroyed with `im_region_free()`. It has type:

```
int im_region_free( REGION *reg )
```

And, as usual, returns 0 on success and non-zero on error, setting `im_error()`. You must free all regions you create. If you close an image without freeing all the regions defined on that image, the image is just marked for future closure — it is not actually closed until the final region is freed. This behaviour helps to prevent dangling pointers, and it is not difficult to make sure you free all regions — see the examples below.

3.3.3 Image input with regions

Before you can read from a region, you need to call `im_prepare()` to fill the region with image data. It has type:

```
int im_prepare( REGION *reg, Rect *r )
```

Area `r` of the image on which `reg` has been created is prepared and attached to the region.

Exactly what this preparation involves depends upon the image — it can vary from simply adjusting some pointers, to triggering the evaluation of a series of other functions. If it returns successfully, `im_prepare()` guarantees that all pixels within `reg->valid` may be accessed. Note that this may be smaller or larger than `r`, since `im_prepare()` clips `r` against the size of the image.

Programs can access image data in the region by calling the macro `IM_REGION_ADDR()`. It has type

```
char *IM_REGION_ADDR( REGION *reg,
  int x, int y )
```

Provided that point (x,y) lies inside `reg->valid`, `IM_REGION_ADDR()` returns a pointer to pel (x, y). Adding to the result of `IM_REGION_ADDR()` moves to the right along the line of pels, provided you stay strictly within `reg->valid`. Add `IM_REGION_LSKIP()` to move down a line, see below. `IM_REGION_ADDR()` has some other useful features — see the manual page.

Other macros are available to ease address calculation:

```
int IM_REGION_LSKIP( REGION *reg )
int IM_REGION_N_ELEMENTS( REGION *reg )
int IM_REGION_SIZEOF_LINE( REGION *reg )
```

These find the number of bytes to add to the result of `IM_REGION_ADDR()` to move down a line, the number of band elements across the region and the number of bytes across the region.

Figure 3.5 on page 44 is a version of `average()` which uses regions rather than WIO input. Two things: first, we should really be using `im_iterate()`, see §3.3.4, to do the rectangle algebra for us. Secondly, note that we call `im_pincheck()` rather than `im_incheck()`. `im_pincheck()` signals to the IO system that you are a PIO-aware function, giving `im_prepare()` much more flexibility in the sorts of preparation it can do. Also see the manual pages for `im_poutcheck()` and `im_piocheck()`.

This version of `average()` can be called in exactly the same way as the previous one, but this version has the great advantage of not needing to have the whole of the input image available at once.

We can do one better than this — if the image is being split into small pieces, we can assign each piece to a separate thread of execution and get parallelism. To support this splitting of tasks, VIPS has the notion of a sequence.

3.3.4 Splitting into sequences

A sequence comes in three parts: a start function, a processing function, and a stop function. When VIPS starts up a new sequence, it runs the start function. Start functions return sequence values: a void pointer representing data local to this sequence. VIPS then repeatedly calls the processing function, passing in the sequence value and a new piece of image data for processing. Finally, when processing is complete, VIPS cleans up by calling the stop function, passing in the sequence value as an argument. The types look like this:

```
void *
(*start_fn)( IMAGE *out,
   void *a, void *b )
int
(*process_fn)( REGION *reg,
   void *seq, void *a, void *b )
int
(*stop_fn)( void *seq, void *a, void *b )
```

The values `a` and `b` are carried around by VIPS for your use.

For functions like `average()` which consume images but produce no image output, VIPS provides `im_iterate()`. This has type:

```
int im_iterate( IMAGE *in,
   void *(*start_fn)(),
   int (*process_fn)(),
   int (*stop_fn)(),
   void *a, void *b )
```

```
#include <stdio.h>
#include <stdlib.h>
#include <vips/vips.h>
#include <vips/region.h>

int
average( IMAGE *im, double *out )
{
   int total, i, y;
   REGION *reg;
   Rect area, *r;

   /* Check im.
    */
   if( im_pincheck( im ) )
      return( -1 );
   if( im->BandFmt != IM_BANDFMT_UCHAR || im->Coding != IM_CODING_NONE ) {
      im_error( "average", "uncoded uchar images only" );
      return( -1 );
   }

   /* Make a region on im which we can use for reading.
    */
   if( !(reg = im_region_create( im )) )
      return( -1 );
```

Figure 3.5: First PIO average of image

```
   /* Move area over the image in 100x100 pel chunks.
    * im_prepare() will clip against the edges of the image
    * for us.
    */
   total = 0;
   r = &reg->valid;
   area.width = 100; area.height = 100;
   for( area.top = 0; area.top < im->Ysize; area.top += 100 )
      for( area.left = 0; area.left < im->Xsize;
          area.left += 100 ) {
          /* Fill reg with pels.
           */
          if( im_prepare( reg, &area ) ) {
             /* We must free the region!
              */
             im_region_free( reg );
             return( -1 );
          }

          /* Loop over reg, adding to our total.
           */
          for( y = r->top; y < IM_RECT_BOTTOM( r ); y++ ) {
             unsigned char *p = IM_REGION_ADDR( reg, r->left, y );

             for( i = 0; i < IM_REGION_N_ELEMENTS( reg ); i++ )
                 total += p[i];
          }
      }

   /* Make sure we free the region.
    */
   im_region_free( reg );

   /* Find average.
    */
   *out = (double) total / (IM_IMAGE_N_ELEMENTS( im ) * im->Ysize);

   return( 0 );
}
```

Figure 3.6: First PIO average of image (cont.)

VIPS starts one or more sequences, runs one or more processing functions over image in until all of in has been consumed, and then closes all of the sequences down and returns. VIPS guarantees that the regions the process_fn() is given will be complete and disjoint, that is, every pixel in the image will be passed through exactly one sequence. To make it possible for the sequences to each contribute to the result of the function in an orderly manner, VIPS also guarantees that all start and stop functions are mutually exclusive.

A note on types: <vips/region.h> declares prototypes for im_iterate() and im_generate() (see §3.3.5), but does not give prototypes for the function arguments. This loses a little type-safety, but gains some convenience.

An example should make this clearer. This version of average() is very similar to the average function in the VIPS library — it is only missing polymorphism.

There are a couple of variations on im_prepare(): you can use im_prepare_to() to force writing to a particular place, and im_prepare_thread() to use threaded evaluation. See the man pages.

3.3.5 Output to regions

Regions are written to in just the same way they are read from — by writing to a pointer found with the IM_REGION_ADDR() macro.

im_iterate() does input — im_generate() does output. It has the same type as im_iterate():

```
int
im_generate( IMAGE *out,
   void *(*start_fn)(),
   int (*process_fn)(),
   int (*stop_fn)(),
   void *a, void *b )
```

The region given to the process function is ready for output. Each time the process function is called, it should fill in the pels in the region it was given. Note that, unlike im_iterate(), the areas the process function is asked to produce are not guaranteed to be either disjoint or complete. Again, VIPS may start up many process functions if it sees fit.

Here is invert(), rewritten to use PIO. This piece of code makes use of a pair of standard start and stop functions provided by the VIPS library: im_start_one() and im_stop_one(). They assume that the first of the two user arguments to im_generate() is the input image. They are defined as:

```
REGION *
im_start_one( IMAGE *out, IMAGE *in )
{
   return( im_region_create( in ) );
}
```

and:

```
int
im_stop_one( REGION *seq )
{
   return( im_region_free( seq ) );
}
```

They are useful for simple functions which expect only one input image. See the manual page for im_start_many() for many-input functions.

Functions have some choice about the way they write their output. Usually, they should just write to the region they were given by im_generate(). They can, if they wish, set up the region for output to some other place. See the manual page for im_region_region(). See also the source for im_copy() and im_extract() for examples of these tricks.

Note also the call to im_demand_hint(). This function hints to the IO system, suggesting the sorts of shapes of region this function is happiest with. VIPS supports four basic shapes — choosing the correct shape can have a dramatic effect on the speed of your function. See the man page for full details.

3.3.6 Callbacks

VIPS lets you attach callbacks to image descriptors. These are functions you provide that VIPS will call when certain events occur. There are more callbacks than are listed here: see the man page for full details.

Close callbacks

These callbacks are invoked just before an image is closed. They are useful for freeing objects which are associated with the image. All callbacks are triggered

```
#include <stdio.h>
#include <stdlib.h>
#include <vips/vips.h>
#include <vips/region.h>

/* Start function for average(). We allocate a small piece of
 * storage which this sequence will accumulate its total in. Our
 * sequence value is just a pointer to this storage area.
 *
 * The first of the two pointers VIPS carries around for us is a
 * pointer to the space where we store the grand total.
 */
static int *
average_start( IMAGE *out )
{
   int *seq = IM_NEW( out, int );

   if( !seq )
      return( NULL );
   *seq = 0;

   return( seq );
}

/* Stop function for average(). Add the total which has
 * accumulated in our sequence value to the grand total for
 * the program.
 */
static int
average_stop( int *seq, int *gtotal )
{
   /* Stop functions are mutually exclusive, so we can write
    * to gtotal without clashing with any other stop functions.
    */
   *gtotal += *seq;

   return( 0 );
}
```

Figure 3.7: Final PIO average of image

```
/* Process function for average(). Total this region, and
 * add that total to the sequence value.
 */
static int
average_process( REGION *reg, int *seq )
{
   int total, i, y;
   Rect *r = &reg->valid;

   /* Get the appropriate part of the input image ready.
    */
   if( im_prepare( reg, r ) )
      return( -1 );

   /* Loop over the region.
    */
   total = 0;
   for( y = r->top; y < IM_RECT_BOTTOM( r ); y++ ) {
      unsigned char *p = IM_REGION_ADDR( reg, r->left, y );

      for( i = 0; i < IM_REGION_N_ELEMENTS( reg ); i++ )
          total += p[i];
   }

   /* Add to the total for this sequence.
    */
   *seq += total;

   return( 0 );
}
```

Figure 3.8: Final PIO average of image (cont.)

```
/* Find average of image.
 */
int
average( IMAGE *im, double *out )
{
   /* Accumulate grand total here.
    */
   int gtotal = 0;

   /* Prepare im for PIO reading.
    */
   if( im_pincheck( im ) )
      return( -1 );

   /* Check it is the sort of thing we can process.
    */
   if( im->BandFmt != IM_BANDFMT_UCHAR ||
      im->Coding != IM_CODING_NONE ) {
      im_error( "average", "uncoded uchar images only" );
      return( -1 );
   }

   /* Loop over the image in pieces, and possibly in parallel.
    */
   if( im_iterate( im,
      average_start, average_process, average_stop,
      &gtotal, NULL ) )
      return( -1 );

   /* Calculate average.
    */
   *out = (double) gtotal / (IM_IMAGE_N_ELEMENTS( im ) * im->Ysize);

   return( 0 );
}
```

Figure 3.9: Final PIO average of image (cont.)

```
#include <stdio.h>
#include <stdlib.h>
#include <vips/vips.h>
#include <vips/region.h>

/* Process function for invert(). Build the pixels in or
 * from the appropriate pixels in ir.
 */
static int
invert_process( REGION *or, REGION *ir )
{
   Rect *r = &or->valid;
   int i, y;

   /* Ask for the part of ir we need to make or. In this
    * case, the two areas will be the same.
    */
   if( im_prepare( ir, r ) )
      return( -1 );

   /* Loop over or writing pels calculated from ir.
    */
   for( y = r->top; y < IM_RECT_BOTTOM( r ); y++ ) {
      unsigned char *p = IM_REGION_ADDR( ir, r->left, y );
      unsigned char *q = IM_REGION_ADDR( or, r->left, y );

      for( i = 0; i < IM_REGION_N_ELEMENTS( or ); i++ )
          q[i] = 255 - p[i];
   }

   /* Success!
    */
   return( 0 );
}
```

Figure 3.10: PIO invert

```
/* Invert an image.
 */
int
invert( IMAGE *in, IMAGE *out )
{
   /* Check descriptors for PIO compatibility.
    */
   if( im_piocheck( in, out ) )
      return( -1 );

   /* Check input image for compatibility with us.
    */
   if( in->BandFmt != IM_BANDFMT_UCHAR || in->Coding != IM_CODING_NONE ) {
      im_error( "invert", "uncoded uchar images only" );
      return( -1 );
   }

   /* out inherits from in, as before.
    */
   if( im_cp_desc( out, in ) )
      return( -1 );

   /* Set demand hints for out.
    */
   if( im_demand_hint( out, IM_THINSTRIP, in, NULL ) )
      return( -1 );

   /* Build out in pieces, and possibly in parallel!
    */
   if( im_generate( out,
      im_start_one, invert_process, im_stop_one,
      in, NULL ) )
      return( -1 );

   return( 0 );
}
```

Figure 3.11: PIO invert (cont.)

in the reverse order to the order in which they were attached. This is sometimes important when freeing objects which contain pointers to other objects. Close callbacks are guaranteed to be called, and to be called exactly once.

Use `im_add_close_callback()` to add a close callback:

```
typedef int (*im_callback)( void *, void * )
int im_add_close_callback( IMAGE *,
    im_callback_fn,
    void *, void * )
```

As with `im_generate()`, the two `void *` pointers are carried around for you by VIPS and may be used as your function sees fit.

Preclose callbacks

Preclose callbacks are called before any shutdown has occured. Everything is still alive and your callback can do anything to the image. Preclose callbacks are guaranteed to be called, and to be called exactly once. See the manual page for `im_add_preclose_callback()` for full details.

Eval callbacks

These are callbacks which are invoked periodically by VIPS during evaluation. The callback has access to a struct containing information about the progress of evaluation, useful for user-interfaces built on top of VIPS. See the manual page for `im_add_eval_callback()` for full details.

3.3.7 Memory allocation revisited

When you are using PIO, memory allocation becomes rather more complicated than it was before. There are essentially two types of memory which your function might want to use for working space: memory which is associated with each instance of your function (remember that two copies of you function may be joined together in a pipeline and be running at the same time — you can't just use global variables), and memory which is local to each sequence which VIPS starts on your argument image.

The first type, memory local to this function instance, typically holds copies of any parameters passed to your image processing function, and links to any read-only tables used by sequences which you run over the image. This should be allocated in your main function.

The second type of memory, memory local to a sequence, should be allocated in a start function. Because this space is private to a sequence, it may be written to. Start and stop functions are guaranteed to be single-threaded, so you may write to the function-local memory within them.

3.4 Programming in-place functions

VIPS includes a little support for in-place functions — functions which operate directly on an image, both reading and writing from the same descriptor via the data pointer. This is an extremely dangerous way to handle IO, since any bugs in your program will trash your input image.

Operations of this type should call `im_rwcheck()` instead of `im_incheck()`. `im_rwcheck()` tries to get a descriptor ready for in-place writing. For example, a function which cleared an image to black might be written as:

```
#include <stdio.h>
#include <memory.h>

#include <vips/vips.h>

int
black_inplace( IMAGE *im )
{
    /* Check that we can RW to im.
     */
    if( im_rwcheck( im ) )
        return( -1 );

    /* Zap the image!
     */
    memset( im->data, 0,
        IM_IMAGE_SIZEOF_LINE( im ) *
        im->Ysize );

    return( 0 );
}
```

This function might be called from an application as:

```
#include <stdio.h>
```

```
#include <stdlib.h>

#include <vips/vips.h>

void
zap( char *name )
{
   IMAGE *im;

   if( !(im = im_open( name, "rw" )) ||
      black_inplace( im ) ||
      im_updatehist( im, "zap image" ) ||
      im_close( im ) )
      error_exit( "failure!" );
}
```

Chapter 4

VIPS reference

4.1 Introduction

This document introduces the functions available in the VIPS image processing library. For detailed information on particular functions, refer to the UNIX on-line manual pages. Enter (for example):

```
example% man im_abs
```

for information on the function `im_abs()`.

All the comand-line vips operations will print help text too. For example:

```
example% vips im_extract
usage: vips im_extract input output
  left top width height band
where:
        input is of type "image"
        output is of type "image"
        left is of type "integer"
        top is of type "integer"
        width is of type "integer"
        height is of type "integer"
        band is of type "integer"
extract area/band, from package
  "conversion"
flags: (PIO function)
  (coordinate transformer)
  (area operation)
  (result can be cached)
vips: error calling function
im_run_command: too few arguments
```

Once you have found a function you need to use, you can call it from a C program (see §2.1 on page 11), you can call it from C++ or Python (see §1.1 on page 1), you can call it from the `nip2` ((see the *nip Manual*), or SIAM graphical user-interfaces, or you can run it from the UNIX command line with the `vips` program. For example:

```
john% vips im_vips2tiff cam.v t1.tif none
john% vips im_tiff2vips t1.tif t2.v.v 0
john% vips im_equal cam.v t2.v t3.v
john% vips im_min t3.v
255
```

VIPS may have been set up at your site with a set of links which call the vips program for you. You may also be able to type:

```
john% im_vips2tiff cam.v t1.tif none
john% im_tiff2vips t1.tif t2.v.v 0
john% im_equal cam.v t2.v t3.v
john% im_min t3.v
```

There are a few VIPS programs which you cannot run with `vips`, either because their arguments are a very strange, or because they are complete mini-applications (like `vips2dj`). These programs are listed in table 4.1, see the man pages for full details.

4.2 VIPS packages

4.2.1 Arithmetic

See Figure 4.1 on page 58.

Arithmetic functions work on images as if each band element were a separate number. All operations are point-to-point — each output element depends exactly upon the corresponding input element. All (except in a few cases noted in the manual pages) will work with images of any type (or any mixture of types), of any size and of any number of bands.

Name	Description
binfile	Read RAW image
debugim	Print an image pixel by pixel
edvips	Change fields in a VIPS header
header	Print fields from a VIPS header
printlines	Print an image a line at a time
vips	VIPS universal main program
vips-7.14	VIPS wrapper script
find_mosaic	Analyse a set of images for overlaps
mergeup	Join a set of images together
cooc_features	Calculate features of a co-occurence matrix
cooc	Calculate a co-occurence matrix
glds_features	Calculate features of a grey-level distribution matrix
glds	Calculate a grey-level distribution matrix
simcontr	Demonstrate simultaneous contrast
sines	Generate a sinusoidal test pattern
spatres	Generate a spatial resolution test pattern
squares	Generate some squares
batch_crop	Crop a lot of images
batch_image_convert	File format convert a lot of images
batch_rubber_sheet	Warp a lot of images
light_correct	Correct a set of images for shading errors
mitsub	Format a VIPS image for output to a Mitsubishi 3600
shrink_width	Shrink to a specific width
vdump	VIPS to mono Postscript
vips2dj	VIPS to high-quality colour Postscript

Table 4.1: Miscellaneous programs

Arithmetic operations try to preserve precision by increasing the number of bits in the output image when necessary. Generally, this follows the ANSI C conventions for type promotion — so multiplying two `IM_BANDFMT_UCHAR` images together, for example, produces a `IM_BANDFMT_USHORT` image, and taking the `im_costra()` of a `IM_BANDFMT_USHORT` image produces a `IM_BANDFMT_FLOAT` image. The details of the type conversions are in the manual pages.

4.2.2 Relational

See Figure 4.2 on page 59.

Relational functions compare images to other images or to constants. They accept any image or pair of images (provided they are the same size and have the same number of bands — their types may differ) and produce a `IM_BANDFMT_UCHAR` image with the same number of bands as the input image, with 255 in every band element for which the condition is true and 0 elsewhere.

They may be combined with the boolean functions to form complex relational conditions. Use `im_max()` (or `im_min()`) to find out if a condition is true (or false) for a whole image.

4.2.3 Boolean

See Figure 4.3 on page 59.

The boolean functions perform boolean arithmetic on pairs of `IM_BANDFMT_UCHAR` images. They are useful for combining the results of the relational and morphological functions. You can use `im_eorconst()` with 255 as `im_not()`.

4.2.4 Colour

See Figure 4.5 on page 61.

The colour functions can be divided into two main types. First, functions to transform images between the different colour spaces supported by VIPS: `RGB` (also referred to as `disp`), `sRGB`, `XYZ`, `Yxy`, `Lab`, `LabQ`, `LabS`, `LCh` and `UCS`), and second, functions for calculating colour difference metrics. Figure 4.4 shows how the VIPS colour spaces interconvert.

The colour spaces supported by VIPS are:

LabQ This is the principal VIPS colorimetric storage format. See the man page for `im_LabQ2Lab()` for an explanation. You cannot perform calculations on `LabQ` images. They are for storage only. Also refered to as `LABPACK`.

LabS This format represents coordinates in *CIE* $L^*a^*b^*$ space as a three- band `IM_BANDFMT_SHORT` image, scaled to fit the full range of bits. It is the best format for computation, being relatively compact, quick, and accurate. Colour values expressed in this way are hard to visualise.

Lab `Lab` colourspace represents *CIE* $L^*a^*b^*$ colour values with a three-band `IM_BANDFMT_FLOAT` image. This is the simplest format for general work: adding the constant 50 to the L channel, for example, has the expected result.

XYZ *CIE XYZ* colour space represented as a three-band `IM_BANDFMT_FLOAT` image.

XYZ *CIE Yxy* colour space represented as a three-band `IM_BANDFMT_FLOAT` image.

RGB (also refered to as `disp`) This format is similar to the RGB colour systems used in other packages. If you want to export your image to a PC, for example, convert your colorimetric image to `RGB`, then turn it to TIFF with `im_vips2tiff()`. You need to supply a structure which characterises your display. See the manual page for `im_col_XYZ2rgb()` for hints on these guys.

VIPS also supports `sRGB`. This is a version of RGB with a carefully defined and standard conversion from XYZ. See:

`http://www.color.org/`

LCh Like `Lab`, but rectangular ab coordinates are replaced with polar Ch (Chroma and hue) coordinates. Hue angles are expressed in degrees.

UCS A colour space based on the CMC(1:1) colour difference measurement. This is a highly uniform colour space, much better than *CIE* $L^*a^*b^*$ for expressing small differences. Conversions to and from `UCS` are extremely slow.

All VIPS colourspaces assume a D65 illuminant.

The colour-difference functions calculate either ΔE *CIE* $L^*a^*b^*$ (1976 or 2000) or ΔE CMC(1:1) on two images in `Lab`, `XYZ` or `disp` colour space.

```
john% vips --list arithmetic
im_abs               - absolute value
im_acostra           - acos of image (result in degrees)
im_add               - add two images
im_asintra           - asin of image (result in degrees)
im_atantra           - atan of image (result in degrees)
im_avg               - average value of image
im_point_bilinear    - interpolate value at single point, linearly
im_bandmean          - average image bands
im_ceil              - round to smallest integal value not less than
im_cmulnorm          - multiply two complex images, normalising output
im_costra            - cos of image (angles in degrees)
im_cross_phase       - phase of cross power spectrum of two complex images
im_deviate           - standard deviation of image
im_divide            - divide two images
im_exp10tra          - 10^pel of image
im_expntra           - x^pel of image
im_expntra_vec       - [x,y,z]^pel of image
im_exptra            - e^pel of image
im_fav4              - average of 4 images
im_floor             - round to largest integal value not greater than
im_gadd              - calculate a*in1 + b*in2 + c = outfile
im_invert            - photographic negative
im_lintra            - calculate a*in + b = outfile
im_linreg            - pixelwise linear regression
im_lintra_vec        - calculate a*in + b -> out, a and b vectors
im_litecor           - calculate max(white)*factor*(in/white), if clip == 1
im_log10tra          - log10 of image
im_logtra            - ln of image
im_max               - maximum value of image
im_maxpos            - position of maximum value of image
im_maxpos_avg        - position of maximum value of image, averaging in case of draw
im_maxpos_vec        - position and value of n maxima of image
im_measure           - measure averages of a grid of patches
im_min               - minimum value of image
im_minpos            - position of minimum value of image
im_minpos_vec        - position and value of n minima of image
im_multiply          - multiply two images
im_powtra            - pel^x ofbuildimage
im_powtra_vec        - pel^[x,y,z] of image
im_remainder         - remainder after integer division
im_remainderconst    - remainder after integer division by a constant
im_remainderconst_vec - remainder after int division by vector of constants
im_rint              - round to nearest integal value
im_sign              - unit vector in direction of value
im_sintra            - sin of image (angles in degrees)
im_stats             - many image statistics in one pass
im_subtract          - subtract two images
im_tantra            - tan of image (angles in degrees)
```

Figure 4.1: Arithmetic functions

```
john% vips --list relational
im_blend            - use cond image to blend between images in1 and in2
im_equal            - two images equal in value
im_equal_vec        - image equals doublevec
im_equalconst       - image equals const
im_ifthenelse       - use cond image to choose pels from image in1 or in2
im_less             - in1 less than in2 in value
im_less_vec         - in less than doublevec
im_lessconst        - in less than const
im_lesseq           - in1 less than or equal to in2 in value
im_lesseq_vec       - in less than or equal to doublevec
im_lesseqconst      - in less than or equal to const
im_more             - in1 more than in2 in value
im_more_vec         - in more than doublevec
im_moreconst        - in more than const
im_moreeq           - in1 more than or equal to in2 in value
im_moreeq_vec       - in more than or equal to doublevec
im_moreeqconst      - in more than or equal to const
im_notequal         - two images not equal in value
im_notequal_vec     - image does not equal doublevec
im_notequalconst    - image does not equal const
```

Figure 4.2: Relational functions

```
john% vips --list boolean
im_andimage         - bitwise and of two images
im_andimageconst    - bitwise and of an image with a constant
im_andimage_vec     - bitwise and of an image with a vector constant
im_orimage          - bitwise or of two images
im_orimageconst     - bitwise or of an image with a constant
im_orimage_vec      - bitwise or of an image with a vector constant
im_eorimage         - bitwise eor of two images
im_eorimageconst    - bitwise eor of an image with a constant
im_eorimage_vec     - bitwise eor of an image with a vector constant
im_shiftleft        - shift integer image n bits to left
im_shiftright       - shift integer image n bits to right
```

Figure 4.3: Boolean functions

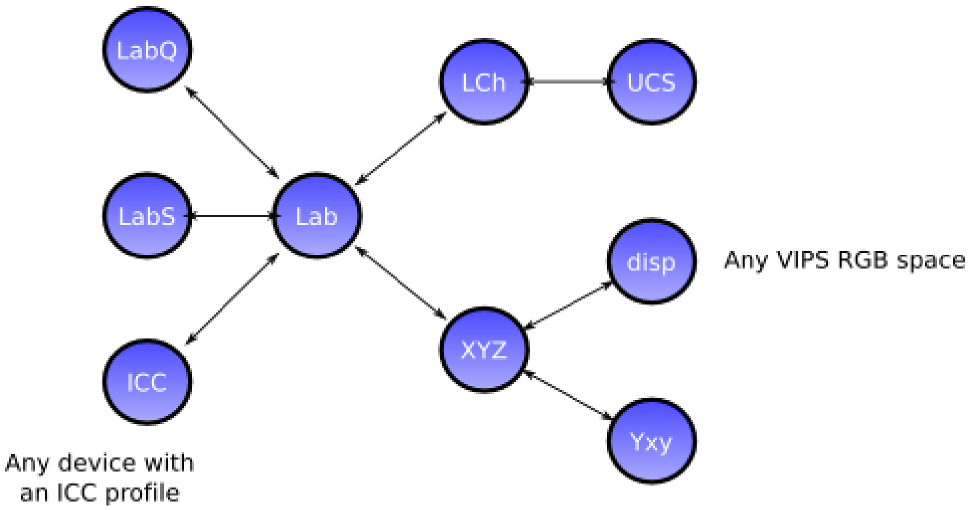

Figure 4.4: VIPS colour space conversion

4.2.5 Conversion

See Figure 4.6 on page 62.

These functions may be split into three broad groups: functions which convert between the VIPS numeric formats (`im_clip2fmt()`, for example, converts an image of any type to the specified `IM_BANDFMT`), functions supporting complex arithmetic (`im_c2amph()`, for example, converts a complex image from rectangular to polar co ordinates) and functions which perform some simple geometric conversion (`im_extract()` forms a sub-image).

`gbandjoin` and the C function `im_gbandjoin()` will do a bandwise join of many images at the same time. See the manual pages.

4.2.6 Matricies

See Figure 4.8 on page 64.

VIPS uses matricies for morphological operations, for convolutions, and for some colour-space conversions. There are two types of matrix: integer (`INTMASK`) and double precision floating point (`DOUBLEMASK`).

For convenience, both types are stored in files as ASCII. The first line of the file should start with the matrix dimensions, width first, then on the same line an optional scale and offset. The two size fields should be integers; the scale and offset may be floats. Subsequent lines should contain the matrix elements, one row per line. The scale and offset are the conventional ones used to represent non-integer values in convolution masks — in other words:

$$result = \frac{value}{scale} + offset$$

If the scale and offset are missing, they default to 1.0 and 0.0. See the sections on convolution for more on the use of these fields. So as an example, a 4 by 4 identity matrix would be stored as:

```
4 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

And a 3 by 3 mask for block averaging with convolution might be stored as:

```
3 3 9 0
1 1 1
1 1 1
1 1 1
```

(in other words, sum all the pels in every 3 by 3 area, and divide by 9).

This matrix contains only integer elements and so could be used as an argument to functions expecting

```
example% vips --list colour
im_LCh2Lab           - convert LCh to Lab
im_LCh2UCS           - convert LCh to UCS
im_Lab2LCh           - convert Lab to LCh
im_Lab2LabQ          - convert Lab to LabQ
im_Lab2LabS          - convert Lab to LabS
im_Lab2UCS           - convert Lab to UCS
im_Lab2XYZ           - convert D65 Lab to XYZ
im_Lab2XYZ_temp      - convert Lab to XYZ, with a specified colour temperature
im_Lab2disp          - convert Lab to displayable
im_LabQ2LabS         - convert LabQ to LabS
im_LabQ2Lab          - convert LabQ to Lab
im_LabQ2XYZ          - convert LabQ to XYZ
im_LabQ2disp         - convert LabQ to displayable
im_LabS2LabQ         - convert LabS to LabQ
im_LabS2Lab          - convert LabS to Lab
im_UCS2LCh           - convert UCS to LCh
im_UCS2Lab           - convert UCS to Lab
im_UCS2XYZ           - convert UCS to XYZ
im_XYZ2Lab           - convert D65 XYZ to Lab
im_XYZ2Lab_temp      - convert XYZ to Lab, with a specified colour temperature
im_XYZ2UCS           - convert XYZ to UCS
im_XYZ2Yxy           - convert XYZ to Yxy
im_XYZ2disp          - convert XYZ to displayble
im_XYZ2sRGB          - convert XYZ to sRGB
im_Yxy2XYZ           - convert Yxy to XYZ
im_dE00_fromLab      - calculate delta-E CIE2000 for two Lab images
im_dECMC_fromLab     - calculate delta-E CMC(1:1) for two Lab images
im_dECMC_fromdisp    - calculate delta-E CMC(1:1) for two displayable images
im_dE_fromLab        - calculate delta-E for two Lab images
im_dE_fromXYZ        - calculate delta-E for two XYZ images
im_dE_fromdisp       - calculate delta-E for two displayable images
im_disp2Lab          - convert displayable to Lab
im_disp2XYZ          - convert displayable to XYZ
im_icc_ac2rc         - convert LAB from AC to RC using an ICC profile
im_icc_export        - convert a float LAB to an 8-bit device image with an ICC profi
im_icc_export_depth  - convert a float LAB to device space with an ICC profile
im_icc_import        - convert a device image to float LAB with an ICC profile
im_icc_import_embedded - convert a device image to float LAB using the embedded profi
im_icc_present       - test for presence of ICC library
im_icc_transform     - convert between two device images with a pair of ICC profiles
im_lab_morph         - morph colourspace of a LAB image
im_sRGB2XYZ          - convert sRGB to XYZ
```

Figure 4.5: Colour functions

```
example% vips --list conversion
im_bandjoin          - bandwise join of two images
im_bernd             - extract from pyramid as jpeg
im_black             - generate black image
im_c2amph            - convert real and imaginary to phase and amplitude
im_c2imag            - extract imaginary part of complex image
im_c2ps              - find power spectrum of complex image
im_c2real            - extract real part of complex image
im_c2rect            - convert phase and amplitude to real and imaginary
im_clip2c            - convert to signed 8-bit integer
im_clip2cm           - convert to complex
im_clip2d            - convert to double-precision float
im_clip2dcm          - convert to double complex
im_clip2f            - convert to single-precision float
im_clip2fmt          - convert image format to ofmt
im_clip2i            - convert to signed 32-bit integer
im_clip2s            - convert to signed 16-bit integer
im_clip2ui           - convert to unsigned 32-bit integer
im_clip2us           - convert to unsigned 16-bit integer
im_clip              - convert to unsigned 8-bit integer
im_copy              - copy image
im_copy_morph        - copy image, setting pixel layout
im_copy_swap         - copy image, swapping byte order
im_copy_set          - copy image, setting informational fields
im_copy_set_meta     - copy image, setting a meta field
im_extract_area      - extract area
im_extract_areabands - extract area and bands
im_extract_band      - extract band
im_extract_bands     - extract several bands
im_extract           - extract area/band
im_falsecolour       - turn luminance changes into chrominance changes
im_fliphor           - flip image left-right
im_flipver           - flip image top-bottom
im_gbandjoin         - bandwise join of many images
im_grid              - chop a tall thin image into a grid of images
im_insert            - insert sub-image into main image at position
im_insert_noexpand   - insert sub-image into main image at position, no expansion
im_lrjoin            - join two images left-right
im_mask2vips         - convert DOUBLEMASK to VIPS image
im_msb               - convert to uchar by discarding bits
im_msb_band          - convert to single band uchar by discarding bits
im_print             - print string to stdout
im_recomb            - linear recombination with mask
im_replicate         - replicate an image horizontally and vertically
im_ri2c              - join two non-complex images to form complex
```

Figure 4.6: Conversion functions

```
im_rot180            - rotate image 180 degrees
im_rot270            - rotate image 270 degrees clockwise
im_rot90             - rotate image 90 degrees clockwise
im_scale             - scale image linearly to fit range 0-255
im_scaleps           - logarithmic scale of image to fit range 0-255
im_rightshift_size   - decrease size by a power-of-two factor
im_slice             - slice an image using two thresholds
im_subsample         - subsample image by integer factors
im_system            - run command on image
im_tbjoin            - join two images top-bottom
im_text              - generate text image
im_thresh            - slice an image at a threshold
im_vips2mask         - convert VIPS image to DOUBLEMASK
im_wrap              - shift image origin, wrapping at sides
im_zoom              - simple zoom of an image by integer factors
```

Figure 4.7: Conversion functions (cont.)

both INTMASK and DOUBLEMASK matricies. However, masks containing floating-point values (such as the output of im_matinv()) can only be used as arguments to functions expecting DOUBLEMASKs.

A set of functions for mask input and output are also available for C-programmers — see the manual pages for im_read_dmask(). For other matrix functions, see also the convolution sections and the arithmetic sections.

4.2.7 Convolution

See Figure 4.9 on page 65.

The functions available in the convolution package can be split into five main groups.

First, are the convolution functions. The most useful function is im_conv() which will convolve any non-complex type with an INTMASK matrix. The output image will have the same size, type, and number of bands as the input image. Of the other im_conv() functions, functions whose name ends in _raw do not add a black border around the output image, functions ending in f use a DOUBLEMASK matrix and write float (or double) output, and functions containing sep are for seperable convolutions. im_compass(), im_lindetect() and im_gradient() convolve with rotating masks. im_embed() is used by the convolution functions to add the border to the output.

Next, are the build functions. im_gauss_*mask() and its ilk generate gaussian masks, im_log_*mask() generate logs of Laplacians. im_addgnoise() and im_gaussnoise() create or add gaussian noise to an image.

Two functions do correlation: im_fastcor() does a quick and dirty correlation, im_spcor() calculates true spatial correlation, and is rather slow.

Some functions are provided for analysing images: im_zerox() counts zero-crossing points in an image, im_mpercent() finds a threshold that will isolate a percentage of points in an image.

Finally, im_resize_linear() and im_shrink() do as you would expect.

4.2.8 In-place operations

See Figure 4.10 on page 66.

A few of the in-place operations are available from the command-line. Most are not.

4.2.9 Frequency filtering

See Figure 4.11 on page 66.

The basic Fourier functions are im_fwfft() and im_invfft(), which calculate the fast-fourier transform and inverse transform of an image. Also im_invfftr(), which just returns the real part of the inverse transform. The Fourier image has its origin at pel (0,0) — for viewing, use im_rotquad() to move the origin to the centre of the image.

```
example% vips --list matrix
im_matcat          - append matrix in2 to the end of matrix in1
im_matinv          - invert matrix
im_matmul          - multiply matrix in1 by matrix in2
im_mattrn          - transpose matrix
```

Figure 4.8: Matrix functions

Once an image is in the frequency domain, it can be filtered by multiplying it with a mask image. The VIPS mask generator is `im_create_fmask()` see the manual page for details of the arguments, but it will create low pass, high pass, ring pass and band pass filters, which may each be ideal, Gaussian or Butterworth. There is also a fractal mask option.

The other functions in the package build on these base facilities. `im_freqflt()` transforms an input image to Fourier space, multiplies it by a mask image, and transforms it back again. `im_flt_image_freq()` will create a mask image of the correct size for you, and call `im_freqflt()`. `im_disp_ps()` will call the right combinations of functions to make a displayable power spectrum for an image.

4.2.10 Histograms and LUTs

See Figure 4.12 on page 67.

VIPS represents histograms and look-up tables in the same way — as images.

They should have either `Xsize` or `Ysize` set to 1, and the other dimension set to the number of elements in the table. The table can be of any size, have any band format, and have any number of bands.

Use `im_histgr()` to find the histogram of an image. Use `im_histnD()` to find the n-dimensional histogram of an n-band image. Perform operations on histograms with `im_histcum()`, `im_histnorm()`, `im_histspec()`, `im_invertlut()`. Visualise histograms with `im_histplot()`. Use a histogram (or LUT) to transform an image with `im_maplut()`. Build a histogram from scratch with `im_identity()` or `im_identity_ushort()`.

Use `im_lhist*()` for local histogram equalisation, and `im_stdif*()` for statisticaol differencing. The `im_tone_*()` functions are for operations on the L channel of a LAB image. Other functions are useful combinations of these basic operations.

4.2.11 Morphology

See Figure 4.13 on page 67.

The morphological functions are used on one-band `IM_BANDFMT_UCHAR` binary images (images containing only zero and not-zero). They search images for particular patterns of pixels (specified with the mask argument), either adding or removing pixels when they find a match. They are useful for cleaning up images — for example, you might threshold an image, and then use one of the morphological functions to remove all single isolated pixels from the result.

If you combine the morphological operators with the mask rotators (`im_rotate_imask45()`, for example) and apply them repeatedly, you can achieve very complicated effects: you can thin, prune, fill, open edges, close gaps, and many others. For example, see 'Fundamentals of Digital Image Processing' by A. Jain, pp 384-388, Prentice-Hall, 1989 for more ideas.

Beware that VIPS reverses the usual image processing convention, by assuming white objects on a black background.

The mask you give to the morphological functions should contain only the values 0 (for background), 128 (for don't care) and 255 (for object). The mask must have odd length sides — the origin of the mask is taken to be the centre value. For example, the mask:

```
3 3
128 255 128
255 0   255
128 255 128
```

applied to an image with `im_erode()`, will find all black pixels 4-way connected with white pixels. Essentially, `im_dilate()` sets pixels in the output if any part of the mask matches, whereas `im_erode()` sets pixels only if all of the mask matches.

The `_raw()` version of the functions do not add a black border to the output. `im_cntlines()` and

```
example% vips --list convolution
im_addgnoise         - add gaussian noise with mean 0 and std. dev. sigma
im_compass           - convolve with 8-way rotating integer mask
im_contrast_surface  - find high-contrast points in an image
im_contrast_surface_raw - find high-contrast points in an image
im_conv              - convolve
im_conv_raw          - convolve, no border
im_convf             - convolve, with DOUBLEMASK
im_convf_raw         - convolve, with DOUBLEMASK, no border
im_convsep           - seperable convolution
im_convsep_raw       - seperable convolution, no border
im_convsepf          - seperable convolution, with DOUBLEMASK
im_convsepf_raw      - seperable convolution, with DOUBLEMASK, no border
im_convsub           - convolve uchar to uchar, sub-sampling by xskip, yskip
im_dmask_xsize       - horizontal size of a doublemask
im_dmask_ysize       - vertical size of a doublemask
im_embed             - embed in within a set of borders
im_fastcor           - fast correlate in2 within in1
im_fastcor_raw       - fast correlate in2 within in1, no border
im_gauss_dmask       - generate gaussian DOUBLEMASK
im_gauss_imask       - generate gaussian INTMASK
im_gaussnoise        - generate image of gaussian noise with specified statistics
im_grad_x            - horizontal difference image
im_grad_y            - vertical difference image
im_gradcor           - non-normalised correlation of gradient of in2 within in1
im_gradcor_raw       - non-normalised correlation of gradient of in2 within in1, no p
im_gradient          - convolve with 2-way rotating mask
im_imask_xsize       - horizontal size of an intmask
im_imask_ysize       - vertical size of an intmask
im_rank_image        - point-wise pixel rank
im_lindetect         - convolve with 4-way rotating mask
im_log_dmask         - generate laplacian of gaussian DOUBLEMASK
im_log_imask         - generate laplacian of gaussian INTMASK
im_maxvalue          - point-wise maximum value
im_mpercent          - find threshold above which there are percent values
im_phasecor_fft      - non-normalised correlation of gradient of in2 within in1
im_rank              - rank filter nth element of xsize/ysize window
im_rank_raw          - rank filter nth element of xsize/ysize window, no border
im_read_dmask        - read matrix of double from file
im_resize_linear     - resize to X by Y pixels with linear interpolation
im_rotate_dmask45    - rotate DOUBLEMASK clockwise by 45 degrees
im_rotate_dmask90    - rotate DOUBLEMASK clockwise by 90 degrees
im_rotate_imask45    - rotate INTMASK clockwise by 45 degrees
im_rotate_imask90    - rotate INTMASK clockwise by 90 degrees
im_sharpen           - sharpen high frequencies of L channel of LabQ
im_shrink            - shrink image by xfac, yfac times
im_spcor             - normalised correlation of in2 within in1
im_spcor_raw         - normalised correlation of in2 within in1, no black padding
im_stretch3          - stretch 3%, sub-pixel displace by xdisp/ydisp
im_zerox             - find +ve or -ve zero crossings in image
```

Figure 4.9: Convolution functions

```
example% vips --list inplace
im_circle          - plot circle on image
im_flood_blob_copy - flood while pixel == start pixel
im_insertplace     - draw image sub inside image main at position (x,y)
im_line            - draw line between points (x1,y1) and (x2,y2)
im_lineset         - draw line between points (x1,y1) and (x2,y2)
```

Figure 4.10: In-place operations

```
example% vips --list freq_filt
im_create_fmask    - create frequency domain filter mask
im_disp_ps         - make displayable power spectrum
im_flt_image_freq  - frequency domain filter image
im_fractsurf       - generate a fractal surface of given dimension
im_freqflt         - frequency-domain filter of in with mask
im_fwfft           - forward fast-fourier transform
im_rotquad         - rotate image quadrants to move origin to centre
im_invfft          - inverse fast-fourier transform
im_invfftr         - real part of inverse fast-fourier transform
```

Figure 4.11: Fourier functions

im_profile are occasionally useful for analysing results.

See the boolean operations im_and(), im_or() and im_eor() for analogues of the usual set difference and set union operations.

4.2.12 Mosaicing

See Figure 4.14 on page 68.

These functions are useful for joining many small images together to make one large image. They can cope with unstable contrast, and arbitary sub-image layout, but will not do any geometric correction. The mosaicing functions can be grouped into layers:

The lowest level functions are im_correl(). and im_affine(). im_correl() searches a large image for a small sub-image, returning the position of the best sub-image match. im_affine() performs a general affine transform on an image: that is, any transform in which parallel lines remain parallel.

Next, im_lrmerge() and im_tbmerge() blend two images together left-right or up-down.

Next up are im_lrmosaic() and im_tbmosaic(). These use the two low-level merge operations to join two images given just an approximate overlap as a start point. Optional extra parameters let you do 'balancing' too: if your images have come from a source where there is no precise control over the exposure (for example, images from a tube camera, or a set of images scanned from photographic sources), im_lrmosaic() and im_tbmosaic() will adjust the contrast of the left image to match the right, the right to the left, or both to some middle value.

The functions im_lrmosaic1() and im_tbmosaic1() are first-order analogues of the basic mosaic functions: they take two tie-points and use them to rotate and scale the right-hand or bottom image before starting to join.

Finally, im_global_balance() can be used to re-balance a mosaic which has been assembled with these functions. It will generally do a better job than the low-level balancer built into im_lrmosaic() and im_tbmosaic(). See the man page. im_remosaic() uses the same techniques, but will reassemble the image from a different set of source images.

4.2.13 CImg functions

See Figure 4.15 on page 68.

These operations wrap the anisotropic blur function from the CImg library. They are useful for removing

```
example% vips --list histograms_lut
im_gammacorrect      - gamma-correct image
im_heq               - histogram-equalise image
im_hist              - find and graph histogram of image
im_histcum           - turn histogram to cumulative histogram
im_histeq            - form histogram equalistion LUT
im_histgr            - find histogram of image
im_histnD            - find 1D, 2D or 3D histogram of image
im_histnorm          - form normalised histogram
im_histplot          - plot graph of histogram
im_histspec          - find histogram which will make pdf of in match ref
im_hsp               - match stats of in to stats of ref
im_identity          - generate identity histogram
im_identity_ushort   - generate ushort identity histogram
im_ismonotonic       - test LUT for monotonicity
im_lhisteq           - local histogram equalisation
im_lhisteq_raw       - local histogram equalisation, no border
im_invertlut         - generate correction table from set of measures
im_buildlut          - generate LUT table from set of x/y positions
im_maplut            - map image through LUT
im_project           - find horizontal and vertical projections of an image
im_stdif             - statistical differencing
im_stdif_raw         - statistical differencing, no border
im_tone_analyse      - analyse in and create LUT for tone adjustment
im_tone_build        - create LUT for tone adjustment of LabS images
im_tone_build_range  - create LUT for tone adjustment
im_tone_map          - map L channel of LabS or LabQ image through LUT
```

Figure 4.12: Histogram/LUT functions

```
example% vips --list morphology
im_cntlines        - count horizontal or vertical lines
im_dilate          - dilate image with mask, adding a black border
im_dilate_raw      - dilate image with mask
im_erode           - erode image with mask, adding a black border
im_erode_raw       - erode image with mask
im_profile         - find first horizontal/vertical edge
```

Figure 4.13: Morphological functions

```
example% vips --list mosaicing
im_affine             - affine transform
im_align_bands        - align the bands of an image
im_correl             - search area around sec for match for area around ref
im__find_lroverlap    - search for left-right overlap of ref and sec
im__find_tboverlap    - search for top-bottom overlap of ref and sec
im_global_balance     - automatically rebuild mosaic with balancing
im_global_balancef    - automatically rebuild mosaic with balancing, float output
im_lrmerge            - left-right merge of in1 and in2
im_lrmerge1           - first-order left-right merge of ref and sec
im_lrmosaic           - left-right mosaic of ref and sec
im_lrmosaic1          - first-order left-right mosaic of ref and sec
im_match_linear       - resample ref so that tie-points match
im_match_linear_search - search sec, then resample so that tie-points match
im_maxpos_subpel      - subpixel position of maximum of (phase correlation) image
im_remosaic           - automatically rebuild mosaic with new files
im_similarity_area    - output area xywh of similarity transformation
im_similarity         - similarity transformation
im_tbmerge            - top-bottom merge of in1 and in2
im_tbmerge1           - first-order top-bottom merge of in1 and in2
im_tbmosaic           - top-bottom mosaic of in1 and in2
im_tbmosaic1          - first-order top-bottom mosaic of ref and sec
```

Figure 4.14: Mosaic functions

```
example% vips --list cimg
im_greyc            - noise-removing filter
im_greyc_mask       - noise-removing filter, with a mask
```

Figure 4.15: CImg functions

noise from images.

4.2.14 Other

See Figure 4.16 on page 70.

These functions generate various test images. You can combine them with the arithmetic and rotate functions to build more complicated images.

The `im_benchmark*()` operations are for testing the VIPS SMP system.

4.2.15 IO functions

See Figure 4.17 on page 70.

These functions are related to the image IO system.

4.2.16 Format functions

See Figure 4.18 on page 70.

These functions convert to and from various image formats. See §2.4 on page 30 for a nice API over these functions.

```
example% vips --list other
im_benchmark       - do something complicated for testing
im_benchmark2      - do something complicated for testing
im_benchmarkn      - do something complicated for testing
im_eye             - generate IM_BANDFMT_UCHAR [0,255] frequency/amplitude image
im_grey            - generate IM_BANDFMT_UCHAR [0,255] grey scale image
im_feye            - generate IM_BANDFMT_FLOAT [-1,1] frequency/amplitude image
im_fgrey           - generate IM_BANDFMT_FLOAT [0,1] grey scale image
im_fzone           - generate IM_BANDFMT_FLOAT [-1,1] zone plate image
im_make_xy         - generate image with pixel value equal to coordinate
im_zone            - generate IM_BANDFMT_UCHAR [0,255] zone plate image
```

Figure 4.16: Other functions

```
example% vips --list iofuncs
im_binfile         - open a headerless binary file
im_cache           - cache results of an operation
im_guess_prefix    - guess install area
im_guess_libdir    - guess library area
im_header_get_type - return field type
im_header_int      - extract int fields from header
im_header_double   - extract double fields from header
im_header_string   - extract string fields from header
im_version         - VIPS version number
im_version_string  - VIPS version string
```

Figure 4.17: IO functions

```
example% vips --list format
im_csv2vips          - read a file in csv format
im_jpeg2vips         - convert from jpeg
im_magick2vips       - load file with libMagick
im_png2vips          - convert PNG file to VIPS image
im_exr2vips          - convert an OpenEXR file to VIPS
im_ppm2vips          - read a file in pbm/pgm/ppm format
im_analyze2vips      - read a file in analyze format
im_tiff2vips         - convert TIFF file to VIPS image
im_vips2csv          - write an image in csv format
im_vips2jpeg         - convert to jpeg
im_vips2mimejpeg     - convert to jpeg as mime type on stdout
im_vips2png          - convert VIPS image to PNG file
im_vips2ppm          - write a file in pbm/pgm/ppm format
im_vips2tiff         - convert VIPS image to TIFF file
```

Figure 4.18: Format functions