Introduction to Programming using the Monash Image Library for Linux

CSE3314 Image Processing
School of Computer Science and Software Engineering
Faculty of Information Technology
Monash University, Clayton, Victoria, Australia, 3800

Document written by Torsten Seemann

http://www.csse.monash.edu.au/software/mil/

Updated 26 July 2000

Contents

1 Introduction **3**

2 Review of Basic Concepts **3**
- 2.1 What is an Image? 3
- 2.2 What is a Pixel? 3
  - 2.2.1 Greyscale Images 3
  - 2.2.2 Colour Images 4
- 2.3 Summary 4

3 Using the *Monash Image Library* with Linux **4**
- 3.1 Installing 4
  - 3.1.1 For the bash shell 4
  - 3.1.2 For the tcsh shell 5
- 3.2 Compiling 5
- 3.3 Example Programs 5
- 3.4 Common Error Messages 5
- 3.5 Printing Images 6
- 3.6 Example Images 6
- 3.7 Disk Usage 6
- 3.8 Problems 6

4 Basic *Monash Image Library* Structure **7**
4.1 Data Types . . . 7
4.2 Image Level Functions . . . 7
4.3 Pixel Level Functions . . . 8

5 Example 1 **9**
5.1 Purpose . . . 9
5.2 Listing . . . 9
5.3 Discussion . . . 10

6 Example 2 **10**
6.1 Purpose . . . 10
6.2 Listing . . . 10
6.3 Discussion . . . 11

7 Example 3 **12**
7.1 Purpose . . . 12
7.2 Listing . . . 13
7.3 Discussion . . . 14

8 Example 4 **14**
8.1 Purpose . . . 14
8.2 Listing . . . 15
8.3 Discussion . . . 17

9 Command Line Programs **17**
9.1 Example . . . 17
9.2 Useful Command Line Programs . . . 18
9.3 Using Pipes . . . 19

10 Further help using the *Monash Image Library* **20**

1 Introduction

The *Monash Image Library* is a library of over 200 C functions for loading, saving and manipulating digital images. It was developed internally at the Department of Computer Science (now School of Computer Science and Software Engineering) at Monash University and is used by both staff and students.

This document is intended to provide an introduction to setting up and using the *Monash Image Library* under Linux. It is assumed the reader is familiar with C programming and has some basic understanding of Linux, the command line, and Makefiles. The aim has been to teach how to use the library functions by using programming examples as much as possible so that the reader can quickly commence writing their own programs.

Firstly, some simple "image" concepts will be reviewed. Then we show how to set up your Linux account for using the *Monash Image Library*. The essential data types and functions available to the programmer will then be described, followed by a series of explained C code examples. The use of command line versions of functions and Unix pipes will finish the discussion.

2 Review of Basic Concepts

2.1 What is an Image?

The word "image" is used in the computing world to describe the pictures and graphics that are stored digitally and can be displayed on an output device such as your computer screen or printer.

The *Monash Image Library* has a slightly more restrictive view of what an image is. It considers an image to be *a rectangular grid of pixels*. An image therefore has 3 main attributes: the height or number of rows in the image, the width or number of columns in the image, and the type of pixel at each grid position.

2.2 What is a Pixel?

The word "pixel" is short for "picture element". Each pixel occupies one position in the rectangular grid which makes up an image. We usually think of pixels as having a particular "colour". The *Monash Image Library* supports many different pixel types but for most purposes you will only need to use the following two types:

2.2.1 Greyscale Images

A greyscale image is one in which the pixels are all shades of grey, much like the picture that an old monochrome or "black and white" television set provides. The different shades correspond to different *intensities* of light, ranging from black to grey to white.

A typical greyscale image would have 256 different intensity levels. The *Monash Image Library* uses integers to describe these intensities, with 0 representing the darkest shade (ie. black) and 255 the brightest (ie. white). Shades in between black and white take on intermediate values. For example, a dark grey might have intensity value 37.

For the computer to store a pixel intensity which could take on 256 different values, it requires 8 *bits per pixel*, as $2^8 = 256$. An image whose pixels only ranged from 0 (black) to 15 (white) would only need 4 bits per pixel, as $2^4 = 16$, $\log_2 16 = 4$.

2.2.2 Colour Images

Colour images are slightly more complicated. The *Monash Image Library* supports many different colour image formats, but I will only describe the simplest and most common one which will be used — the RGB format.

In the RGB format, each pixel is represented by a three integers: a red component, a blue component and a green component. Usually we use 8 bits per component, which results in 24 bits per pixel overall.

The colour of the pixel is generated by *mixing* the three primary components. For example, the brightest purest green would have the RGB values $(0, 255, 0)$. That is, no red, full green and no blue. Mixing red and green gives yellow, so pure yellow would be $(255, 255, 0)$, and a medium purple (or indigo, violet) shade could be represented by the RGB triplet $(136, 0, 136)$. Black is $(0, 0, 0)$ and white is $(255, 255, 255)$.

2.3 Summary

From the previous section we have learnt that an image is just a two-dimensional rectangular grid of pixels. Each pixel has a coordinate or position in the grid (which row and column it is in). In the *Monash Image Library* pixels are represented by integers. Greyscale pixel intensities are represented by a single integer, usually ranging from 0 to 255 (black to white). Colour pixels require a vector of 3 integers containing the red/green/blue components.

3 Using the *Monash Image Library* with Linux

3.1 Installing

To use the various image library and utility programs you need to set some Unix environment variables. How this is done varies depending on which Unix shell you use. Typing `finger $USER` when you've logged in will tell you which shell you are using.

3.1.1 For the bash shell

Add the following lines to the *end* of `$HOME/.bashrc` file:

```
export MILHOME=/cs/cc/lib/mil
export PATH=${PATH}:${MILHOME}/bin
export IMAGES=${MILHOME}/images/grey:${MILHOME}/images/colour:.
```

You can now log out and log back in *or* just type:

```
source $HOME/.bashrc ; hash -r
```

3.1.2 For the tcsh shell

Add the following lines to the *end* of `$HOME/.tcshrc` file:

```
setenv MILHOME /cs/cc/lib/mil
setenv PATH ${PATH}:${MILHOME}/bin
setenv IMAGES ${MILHOME}/images/grey:${MILHOME}/images/colour:.
```

You can now log out and log back in *or* just type:

```
source $HOME/.tcshrc ; rehash
```

3.2 Compiling

User applications which call functions in the library must `#include <image.h>` and link with `-limage -lX11 -lm`. This header file and the library of functions must be specified when compiling.

The best way is to copy and adapt the `Makefiles` supplied with the example programs. However if you need to to compile `prog.c` to `prog` on the command line you would (all on the one line) type:

```
gcc -Wall -I$MILHOME/include -L$MILHOME/lib
-o prog prog.c -limage -lX11 -lm
```

3.3 Example Programs

The example C programs in Sections 5–8 document are located in `$MILHOME/examples`.

Make sure you examine the source code of the examples and the `Makefiles` carefully before beginning the assignments. To compile and test them, you will have to first *copy the directories* to your home directory structure somewhere. Then type `make`. Assuming you have set up your `PATH` variable properly, you will end up with an executable which you can try out.

Remember to type `make clean` when you are done. This will remove the `.o` files, any `core` files and the usually very large executable.

3.4 Common Error Messages

- *xv: Can't open display* means that your `$DISPLAY` variable is not set correctly or that you are not even running X-Windows.
- *You must set the `$MILHOME` environment variable to where you installed the Monash Image Library eg. `setenv MILHOME $HOME/mil`* means you didn't set the appropriate environment variables as described in Section 3.1
- *Segmentation fault dumped* or *Bus error* or *TraceBPT trap* and *Abort (core dumped)* means that you have probably incorrectly used a pointer variable in your program, such as going over the end of an array or using an uninitialized pointer.

- *sh: xv: command not found* means that you do not have the `xv` program in your `PATH` or even installed at all. This needs to be installed.
- *sh: less: command not found* means that you do not have the `less` file viewer program in your `PATH` or even installed at all. You need to either install it or make a symbolic link to another pager program such as `more` or `cat`.

3.5 Printing Images

To print images to the Computer Centre printers you must first convert the images to the Postscript format. This can be done either with `xv` (which also allows you to cut and paste images together) or with the `idump` command. For example, assuming the printer outside Room 107 is called `cl_ctl_107_ps`, you could type:

```
idump mark.gif mark ps     (Note the space before ps)
lpr -h -x -Pcl_ctl_107_ps mark.ps
```

Ideally, you should preview the Postscript file *before* printing so any errors can be caught. To do this you can use `ghostview` or `gs`. eg. `gs mark.ps`

3.6 Example Images

`$MILHOME/images/grey/` has greyscale images suitable for enhancement, segmentation and edge detection algorithms. `$MILHOME/images/colour/` has some colour (24 bit RGB) images.

3.7 Disk Usage

The image library programs you write, their object files, and the images and Postscript files you generate all consume large amounts of disk space. It is in your better interests to only maintain the source files and details of how the images were generated so that they can be reproduced only when required.

3.8 Problems

If you have any problems with the image library, please ensure that you have carefully read the handouts provided, the example programs, and the help pages provided by `ihelp`. Fellow students are also a good source of information, as they are probably having or have had similar difficulties.

If you have serious problems with the library, or you believe you have found a bug in any of the routines, please email Torsten Seemann at `torsten@csse.monash.edu.au` with the string “Monash Image Library” in the Subject.

4 Basic *Monash Image Library* Structure

4.1 Data Types

The *Monash Image Library* has one fundamental data type which is a C structure called `IMAGE`. It has many different fields but only the four which are relevant to most programmers are shown below:

```
typedef struct IMAGE_T
{
  long    rows;           /* number of scan lines */
  long    cols;           /* pixels per scan line */
  long    bitsperpixel;   /* pixel size */
  char    name[I_MAXNAM]; /* name of image */
  /* Have a look in image.h for the rest of the fields */
}
IMAGE;
```

When writing programs an `IMAGE` structure is *always* accessed via a pointer like the `picture` variable shown below. To access the fields the C `->` notation must be used.

```
IMAGE* picture = 0;
/* ... Open or create an image here with other functions ... */
printf("rows = %ld\n", picture->rows);
printf("cols = %ld\n", picture->cols);
printf("bpp  = %ld\n", picture->bpp );
printf("name = %s\n",  picture->name);
```

4.2 Image Level Functions

Image level functions are those which operate on or return whole images (via `IMAGE` pointers). The functions to create empty images, load, save and display images are examples of these. Here is a list of the most common image level functions:

`IMAGE *i_open(char *filename)`

Loads an image file from disk into memory. The function will search the current directory and directories listed in the `IMAGES` environmental variable. It returns a pointer to the new image.

`void i_close(IMAGE *image)`

Closes the image, releasing all memory allocated to the image.

`void i_dump(IMAGE *image, char *name, char *format)`

Saves (dumps) the image to disk with given name in given format. Useful formats for greyscale images are: `cif`, `gif`, `img`. For colour images `cif` is preferred.

`IMAGE *i_mktemp(int rows, int cols, int bpp)`

Returns a new empty image of height `rows`, width `cols` and bits per pixel `bpp`. Use 8 bpp for greyscale and 24 bpp for colour images. The pixels are not set to zero so you must use the `i_zero` function to do that.

`void i_zero(IMAGE *image)`

Zeroes all pixels in the image. This will result in an all-black image for greyscale and RGB image types.

`IMAGE *i_hist(IMAGE *image)`

Returns an intensity histogram (as an image) of the input image.

`void i_xv(IMAGE *image)`

Displays the image in a new window. Only works if you are running X-Windows and the `xv` image viewer program is installed and in your path (in the `$PATH` variable on your Unix shell).

`void i_error(int LEVEL, char* format, ...)`

Prints an error message and exits (depending on the error LEVEL). The LEVEL argument is a macro defined in `image.h` which must be one of I_INFORM, I_WARNING, I_FATAL, I_PANIC, I_FATALSYS or I_PANICSYS. The format and other arguments are as for `printf`.

4.3 Pixel Level Functions

Pixel level functions are those which manipulate individual pixels within an image. Here is a small list of the more commonly used pixel level functions:

`int i_getpix(IMAGE *image, int row, int col)`

Returns the value of the pixel at (row,col) in the given greyscale image.

`i_putpix(IMAGE *image, int row, int col, int pixel)`

Stores the pixel value at (row,col) in the given greyscale image. The programmer must ensure that it fits into the range allowed by the bits per pixel of the image eg. only 0–255 for 8 bpp images.

`void i_getrgb(IMAGE *image, int row, int col, int *red, int *green, int *blue)`

Gets the colour pixel at (row,col) in the image. The user must supply three non-null pointers to integers to store the (red,green,blue) components of the colour pixel.

```
void i_putrgb(IMAGE *image, int row, int col, int red, int green,
int blue)
```

Stores the colour pixel with components (red,green,blue) at (row,col) in the given image.

```
int i_getred(IMAGE *image, int row, int col)
int i_getgreen(IMAGE *image, int row, int col)
int i_getblue(IMAGE *image, int row, int col)
```

These functions can be used instead of the i_getrgb function if you only need to access one specific colour component.

```
int i_putred(IMAGE *image, int row, int col, int red)
int i_putgreen(IMAGE *image, int row, int col, int green)
int i_putblue(IMAGE *image, int row, int col, int blue)
```

These functions can be used instead of the i_putrgb function if you only need to access one colour component.

5 Example 1

5.1 Purpose

This example will take an image filename as a parameter and display both the image and its histogram on the screen.

5.2 Listing

```
1 #include <image.h>
2
3 int main(int argc, char *argv[])
4 {
5   IMAGE* in;
6   IMAGE* hist;
7
8   if (argc < 2)
9     i_error(I_FATAL, "Usage: %s <image>", argv[0]);
10
11   in = i_open(argv[1]);
12   i_xv(in);
13
14   hist = i_hist(in);
15   i_xv(hist);
16
17   i_close(in);
18   i_close(hist);
19
```

```
20   return 0;
21 }
```

5.3 Discussion

Line 1 includes the standard *Monash Image Library* header file. In lines 5 and 6 we create two local variables which are pointers to `IMAGE` types. These pointers do not point to anything yet — they will be used later in the program to point to `IMAGE`s returned by other functions.

Lines 8–9 just check if the user has supplied an image filename as a parameter to the program. If not, it calls the `i_error` function which will print out an error message explaining how to use the program, and then exit.

Line 11 uses the `i_open` function to open an existing image from disk; here we pass it the first command line parameter. The function returns a pointer to the image, which is stored in `in`. This `in` pointer variable can then be passed to other functions which operate on the image in some way.

For example, in line 12 we pass it to the `i_xv` function. This function will display the image on the screen in a new window.

Line 14 uses the `i_hist` function which takes our original input image (via the `in` pointer) and returns a pointer to a new image which provids a graphical histogram and some statistics about the original image. In line 15 it is also displayed on the screen.

In lines 17 and 18 we close the two images we have created (one was created using the `i_open` function, and one using the `i_hist` function). This will free up any memory that the images were using.

Running `ex1 parts.gif` results in Figures 1 and 2.

6 Example 2

6.1 Purpose

This example will show how to create a new blank image, modify the individual pixels in the image, display the image and save the image to disk.

6.2 Listing

```
1  #include <image.h>
2
3  int main(int argc, char *argv[])
4  {
5    IMAGE* out;
6    int row, col, value;
7
8    out = i_mktemp(256, 256, 8);
9    i_zero(out);
10
11   for (row=0; row < out->rows; row++)
12   {
```

Figure 1: Example 1 input image

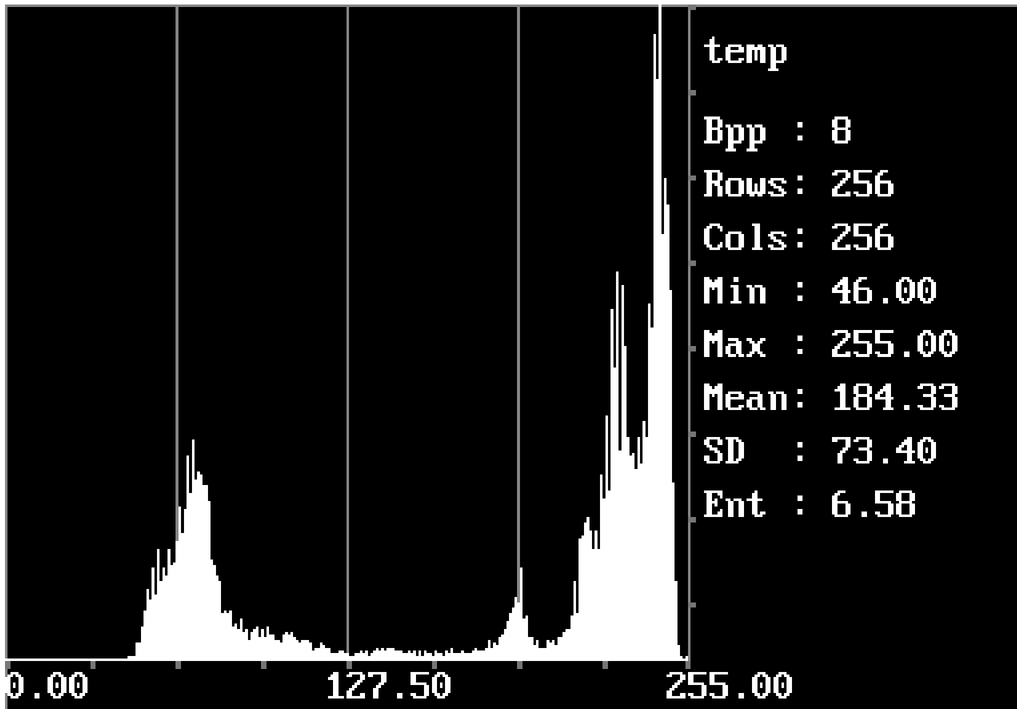
```
13      for (col=0; col < out->cols; col++)
14      {
15        value = col;
16        i_putpix(out, row, col, value);
17      }
18    }
19
20    i_xv(out);
21    i_dump(out, "pattern", "gif");
22    i_close(out);
23
24    return 0;
25  }
```

6.3 Discussion

The start of the program is similar to Example 1. In line 8 we use the `i_mktemp` function to create a new image which has 256 rows, 256 columns, and 8 bits per pixel. ie. a small square greyscale image which can hold pixel values from 0 to 255. Line 8 uses the `i_zero` function to set each pixel in the image to value 0, which corresponds to black.

For this example we will draw a smooth surface on the image going from black on the left to white on the right. That is, all the pixels in column 0 will have grey level 0, column 1 has value 1,

Figure 2: Example 1 input image histogram



until the final column 255 has pixels with value 255.

Lines 11–18 show two nested `for` loops. The outer loop goes through each row one by one and the inner loop goes across each column in each of those rows. Note that we are accessing the `rows` and `cols` members of the `out` `IMAGE` structure.

The body of the two loops is in lines 15–16. In line 16 the `i_putpix` function is used to set the pixel of the `out` image at the specified row and column to `value`.

Line 20 displays the image in a new window on the screen. Line 21 uses the `i_dump` function to save the `out` image to disk in the `gif` format resulting in the filename `pattern.gif`. The image can be viewed using the `ixv` *command* (not C function — see Section 9) on the Unix command line. eg. `ixv pattern.gif`

Running `ex2` results in the output of in Figure 3.

7 Example 3

7.1 Purpose

This program opens an existing image and creates a new image which is the "photographic negative" of it. The program works on greyscale images of any size and any bits per pixel.

Figure 3: Example 2 generated image

7.2 Listing

```c
#include <image.h>

int main(int argc, char *argv[])
{
  IMAGE* in;
  IMAGE* out;
  int row, col, value;
  int maxPixelValue;

  if (argc < 2)
    i_error(I_FATAL, "Usage: %s <image>", argv[0]);

  in = i_open(argv[1]);
  out = i_mktemp(in->rows, in->cols, in->bitsperpixel);

  maxPixelValue = (1 << in->bitsperpixel) - 1;

  for (row=0; row < out->rows; row++)
  {
```

```
20      for (col=0; col < out->cols; col++)
21      {
22        value = maxPixelValue - i_getpix(in, row, col);
23        i_putpix(out, row, col, value);
24      }
25    }
26
27    i_xv(in);
28    i_close(in);
29
30    i_xv(out);
31    i_close(out);
32
33    return 0;
34  }
```

7.3 Discussion

Line 13 opens the specified image. Line 14 creates a new empty image which has the the same dimensions and same pixel type. It does this by accessing the `rows`, `cols` and `bpp` members of the `out IMAGE` structure.

Line 16 computes the largest pixel value that is allowed. This is equal to $2^{bitsperpixel} - 1$. For example an 8 bpp image would come out to be 255, and we know from previous examples that these images have pixels which range from 0 (black) to 255 (white).

To "negate" an image means that our idea of black and white are reversed: black becomes white, white becomes black, dark grey become light grey etc. So for an 8 bpp image we can achieve this by subtracting the old pixel value from 255 to get the new pixel value.

Lines 18–25 loop through each pixel, using the `i_getpix` function to read the original pixel value from `in`, and then placing the reversed pixel value into the `out` image using the `i_putpix` function.

The original and "negative" images are then both displayed on the screen and then closed.

Figures 4 and 5 shows the result of running `ex3 mark.gif`.

8 Example 4

8.1 Purpose

This program will read in a 24 bit per pixel colour image and output an 8 bit per pixel greyscale version of the image. The formula

$$intensity = \sqrt{\frac{red^2 + green^2 + blue^2}{3}}$$

will be used to convert the RGB triplet into an intensity. ie. the normalized length of the colour vector in RGB space.

Figure 4: Example 3 input image

8.2 Listing

```
1  #include <image.h>
2
3  int main(int argc, char *argv[])
4  {
5    IMAGE* in;
6    IMAGE* out;
7    int row, col;
8    int r, g, b;
9    int maxPixelValue;
10   int intensity;
11
12   if (argc < 2)
13     i_error(I_FATAL, "Usage: %s <colour_image>", argv[0]);
14
15   in = i_open(argv[1]);
16
17   if (in->bitsperpixel != 24)
18   {
19     i_close(in);
```

Figure 5: Example 3 negated output image

```
20      i_error(I_FATAL, "This␣program␣only␣works␣on␣24␣bpp␣colour␣images.");
21    }
22
23    maxPixelValue = (1 << in->bitsperpixel) - 1;
24
25    out = i_mktemp(in->rows, in->cols, 8);
26
27    for (row=0; row < out->rows; row++)
28    {
29      for (col=0; col < out->cols; col++)
30      {
31        i_getrgb(in, row, col, &r, &g, &b);
32
33        /* Or we could have used the following three lines:
34           r = i_getred(in, row, col);
35           g = i_getgreen(in, row, col);
36           b = i_getblue(in, row, col); */
37
38        intensity  = (int) sqrt( (r*r + g*g + b*b) / 3.0 );
39
40        if (intensity < 0)
```

```
41          intensity = 0;
42        else if (intensity > maxPixelValue)
43          intensity = maxPixelValue;
44
45        i_putpix(out, row, col, intensity);
46      }
47    }
48
49    i_close(in);
50
51    i_xv(out);
52    i_close(out);
53
54    return 0;
55  }
```

8.3 Discussion

In lines 17–20 we check to see that the input image is a 24 bpp colour image. If not, we print out an error message and exit. In line 25 we create an empty greyscale output image of the same dimensions.

The body of the program is the familiar double loop through rows and columns. The `i_getred`, `i_getgreen`, `i_getblue` functions can be used to extract the three colour components from a colour image just as the `i_getpix` function does for a greyscale image. However, as it is common to require all three components at once, the more efficient `i_getrgb` function is used. Notice that we pass three *pointers* to integers to this function so that it can modify the r,g,b variables.

Line 38 applies the supplied formula to produce an intensity value. Lines 40–43 are used to ensure that the intensity value falls into the legal range for an 8 bpp image, that is 0 to 255. In line 45 we place the clipped pixel value into the output image. The images are then closed and displayed.

Running `ex4 lenna_512_512.rgb` results in our colour input image being converted into a greyscale version, as shown in Figures 6 and 7.

9 Command Line Programs

So far we have shown how to use the *Monash Image Library* library of C functions in your own C programs to open, save, manipulate and view images stored on disk. However most of the image level functions are also available as programs which can be directly run on the Unix command line.

9.1 Example

In some of the example programs we used the `i_xv` C function to display an image (stored in an `IMAGE` variable) in a new window on the screen. If you just need to temporarily view an image you can use the corresponding command line program `ixv`. Notice that is has the same

Figure 6: Example 4 colour input image

name as the C function, but without the underscore "_" character in it. To view an image called `landscape.cif` you would simply type the following:

```
ixv landscape.cif
```

By just typing `ixv` by itself on the command line you will be given some help describing the C function and the command line versions of the function. For `ixv` it would print something like this:

```
display image using xv as a forked child process
Usage: ixv [-b{123a}] in
Function prototype: no transformation or return value
void i_xv(IMAGE *image)
```

9.2 Useful Command Line Programs

Here are some of the more useful command line programs you may wish to use:

```
ixv in
```

Displays the `in` image in a new window on the screen. Like the `i_xv()` function except it loads an image file from the disk rather than via an `IMAGE` pointer in a C program.

Figure 7: Example 4 greyscaled output image

`ihist in out[.fmt]`

Creates an intensity histogram (as an image) of the input image `in` and saves to disk in file format `fmt` with file name `out.fmt`. The format can be one of these strings: `img smg lmg dmg cmg xbm rgb sgi gif ps cps bps cif pbm pgm ppm pnm`.

`idump in basename fmt`

Converts input image `in` to format `fmt` and saves it with file name `basename.fmt`. Note the the command line syntax is slightly different to the `ihist` program. The separation of basename and format arguments is non-standard — the rest of the *Monash Image Library* commands use a combined file name and format.

9.3 Using Pipes

These command line functions become more powerful when used in conjunction with Unix pipes. A pipe allows you do use the output of one program as the input to the next. It does this by using the special Unix files `stdin` and `stdout`. All the image level functions in the *Monash Image Library* either just take an image as input (and output non-image info), produce an image as output, or have an image as both input and output.

The special filename "`-`" (the minus or dash character) is used to tell the *Monash Image Library* to use `stdin` or `stdout` instead of reading/writing a filename to/from the disk. Here is an example of *displaying* the histogram of an image called `mandrill.cif`:

```
ihist mandrill.cif - | ixv -
```

Usually, instead of the first minus sign, you would put the name of the file you wanted to save the histogram image into, but here we want to send it to the next program in the pipe which is `ixv`. The pipe character "|" is used to separate the two programs in the pipe (this is standard Unix shell syntax). The second minus sign (which is in place of the name of a filename to view) says that the input file will be coming from the previous program in the pipe (assuming it used a minus sign on its output, which it has).

This is useful because it is quicker than writing and compiling a program, you don't need temporary files, and the pipes can consist of a series of programs, not just two.

```
isob mandrill.cif - | ibinary - - 40 | ixv -
```

For example, the above line will do Sobel edge detection on the `mandrill.cif` image, perform binary edge detection with threshold 40 and then display the result.

10 Further help using the *Monash Image Library*

This document

This document describes the most commonly used functions in the *Monash Image Library*. The example programs described in this document should be enough to get you started. The best way to learn is to try experimenting by modifying these existing programs rather than attempting to write something from scratch.

The `ihelp` program

If you have the *Monash Image Library* installed you can simply type `ihelp` on the command line. This will describe the command line arguments that can be supplied to get more detailed help on the functions and programs that make up the *Monash Image Library*.

The command line programs

If you know the name of a *Monash Image Library* function (eg. `i_xv`) then simply typing the name of the function without the underscore (eg. `ixv`) will print up the C prototype for the function and a brief description of what it does.

The Internet

This userguide and all the `ihelp` documentation is available on the web at `http://www.csse.monash.edu.au/software/mil/`.

The local network

This userguide and all the `ihelp` documentation is available where the *Monash Image Library* is installed, specifically `$MILHOME/doc`.