

IMPROC: AN INTERACTIVE IMAGE PROCESSING SOFTWARE PACKAGE

George Wolberg

Department of Computer Science
Columbia University
New York, NY 10027
wolberg@cs.columbia.edu

April 14, 1988
Technical Report CUCS-330-88

ABSTRACT

IMPROC is a general-purpose interactive image processing software package. The system includes a collection of library routines and a menu-driven environment in which to invoke all supported image operators. A wide range of image operations are available, including point, neighborhood, arithmetic, logic, and geometric processes. In addition, there are utilities for image transforms, compositing, colorization, look-up tables, and graphic display of images.

Key features of IMPROC include its emphasis on simplicity, generality, and device independence. Flexibility and software utility is augmented by the simple, and general, canonical representation of internal images. Unlike most systems that place strict restrictions on the pixel data type, IMPROC accommodates images having pixels of variable precision and arbitrary dimensions. A consequence of this feature is that all supported image operations are equally useful for general-format data. This uniform treatment of data that vary in size and type is critical to advanced processing techniques. Finally, device independence protects IMPROC from the inevitable obsolescence of the supporting hardware.

This paper is a complete guide to IMPROC. It includes a user's guide, programmer's manual, numerous examples, and an extensive bibliography. In addition, a discussion of the design philosophy is given to supply insight that maximizes user productivity and promotes uniform practices for code integration.

TABLE OF CONTENTS

SECTION 1	INTRODUCTION	2
SECTION 2	IMAGE OPERATIONS	
	2.1 Point Processes / Single Image	3
	2.2 Point Processes / Dual Images	3
	2.3 Neighborhood Processes	4
	2.4 Geometric Processes	5
	2.5 Transforms	6
SECTION 3	DESIGN PHILOSOPHY	
	3.1 General Format Image Representation	7
	3.2 Uniform Treatment of Arbitrary Data Formats	8
	3.3 Device Independence	8
	3.4 Easy Code Integration	8
	3.5 Keep It Simple	9
SECTION 4	IMAGE DATA	
	4.1 Interleaved vs. Non-Interleaved Format	10
	4.2 Channels	10
	4.3 Image Types	10
	4.4 Image File Formats	12
SECTION 5	INTERACTIVE USE OF IMPROC	
	5.1 Screen Layout	13
	5.2 Parameter Collection	14
	5.3 Image Display	14
	5.3.1 Clipping	14
	5.3.2 Embedding Intensities Into the Valid Range	14
	5.4 Command Line Arguments	16
	5.5 Getting Started	16
	5.6 Menu Hierarchy	17
SECTION 6	DESCRIPTION OF MENU OPTIONS	
	6.1 Main Menu	19
	6.2 Point Operation Submenus	21
	6.2.1 The Treatment of Color Images	21
	6.2.2 Point Ops Submenu	22
	6.2.3 Arithmetic Submenu	28
	6.2.4 Logic Ops Submenu	30
	6.3 Neighborhood Ops Submenu	31
	6.4 Geometric Ops Submenu	35
	6.5 Matte Ops Submenu	37
	6.6 Image Conversion	41
	6.7 Image Transforms Submenu	44

	6.8 Image Intensity Plot Submenu	45
SECTION 7	COLORIZATION	
	7.1 Overview	46
	7.2 Color Information	46
	7.2.1 Color Data Structure	47
	7.2.2 Color Database	48
	7.2.3 The C Buffer	49
	7.3 Colorization Submenu	50
	7.4 Display Images Sub-submenu	52
	7.5 Color Palette Sub-submenu	52
SECTION 8	LIBRARY FUNCTIONS	
	8.1 Data Structures	55
	8.2 Quad Manipulation Functions	57
	8.3 Channel Manipulation Functions	59
	8.4 Type Conversion	61
	8.5 Image I/O Functions	62
	8.6 Display Operations	63
	8.7 Point Operations: Single Image	64
	8.8 Point Operations: Arithmetic	66
	8.9 Point Operations: Logic	67
	8.10 Neighborhood Operations	68
	8.11 Geometric Operations	70
	8.12 Image Compositing Operations	71
	8.13 Image Transforms	73
	8.14 Image Utility Routines	75
	8.15 Query Routines	77
	8.16 Frame Buffer Functions	78
SECTION 9	PUTTING IT ALL TOGETHER	
	9.1 Basic Usage	81
	9.2 An Example	81
	9.3 Another Example: Bandpass Filters	82
	9.4 Channel Manipulation	84
	9.5 Yet Another Example: Image Halftones	85
	9.6 Adding Source Code to IMPROC	86
SECTION 10	REFERENCES AND SUGGESTED READING	87

1. INTRODUCTION

IMPROC is a general-purpose interactive image processing software package. It consists of a collection of library routines which facilitate operations on images. The routines, written in C, may be called from within a user's program or may be invoked within the interactive menu-driven environment provided by the system. The software runs on the EDGE 1200, HP 9000, and DEC VAX-11/750 machines under the UNIX[†] operating system. It currently uses a Raster Technology, HP, or Grinnell frame buffer. This can be easily extended to other hardware by modifying a configuration definition file which specifies all hardware specific features.

IMPROC was originally motivated by the desire to minimize the time and effort between considering an image processing routine, and viewing its effects. This conforms with the experimentalist spirit of asking "what if I were to ..." and then shortly seeing the resulting image. Of course the intervening time is spent either invoking already available routines upon the input image, or creating one yourself. By establishing standards on the treatment of image data, providing core routines and an image I/O facility, that interval is greatly reduced. Consequently, IMPROC provides a comprehensive and conducive environment in which to experiment with image processing routines and extend them. Its function as a testbed has obvious educational value.

This paper is a complete guide to IMPROC. It is divided into three parts: design philosophy, user's guide, and programmer's manual. The design philosophy is given to supply insight into the directions and consequences of design decisions. This is intended to be interpreted by two sets of readers: users and programmers. To the user, it is hoped that a greater awareness of the toolkit environment will enhance productivity and results. To the programmer, it is given to promote uniform practices in the design and integration of additional image processing routines.

The rudimentary classes of image operations that are supported by the system are explained in section 2. Since it is a brief and incomplete description of image processing, the user is encouraged to refer to the books and articles listed at the end of the paper. Section 3 describes the design philosophy. A description of the image data representation is given in section 4.

The user's guide begins in section 5 which describes the interactive menu-driven system. The supported functions are explained in section 6. Section 7 is devoted to a discussion of the colorization subsystem.

The programmer's manual begins in section 8 with a discussion of system internals. It is intended for those who wish to add new code to the system as well as maintain existing software. Section 9 presents useful examples. Finally, section 10 lists references and suggested readings in image processing.

[†] UNIX is a trademark of AT&T Bell Laboratories.

2. IMAGE OPERATIONS

Image processing can be categorized as transforming an input image to yield an output image or an alternate image representation. The objective is to visually enhance or statistically evaluate some aspect of an image not readily apparent in its original form. The transformation can make use of a variety of operations that may be broadly classified as point, neighborhood, and geometric processes.

2.1. Point Processes / Single Image

The most basic transformation is one in which the new pixel value (in the output image) is simply a function of the corresponding pixel value in the input image. This simple class of operations is known as *point processes* over single images. An example of this is thresholding, where the new value a pixel takes on is dependent on whether its value lies above or below a specified threshold value. Operations of this kind may be conveniently precomputed to yield a *look-up table* (LUT) whose contents at the location specified by the pixel value is the new value (Fig. 1). In general, all functions that can be implemented through a LUT are considered point processes. Other functions in this class include pseudo coloring, quantization, contrast enhancement, histogram sliding/stretching, and histogram equalization.

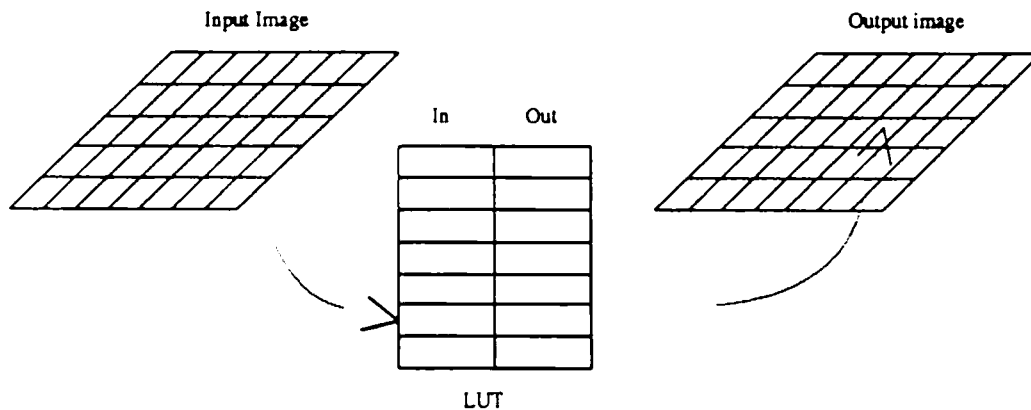


Figure 1: Point process / single image

2.2. Point Processes / Dual Images

Point processing on image pairs is a simple extension of the single image case. Instead of mapping a pixel value from a single input image onto an output image, we now map two pixel values, one from each input image, onto an output image. Arithmetic and logic combination of images are typical of this class of processes. Arithmetic operations are those producing an output image which is the pixel-by-pixel sum, difference, product, or quotient of two input images. Note that when a constant value is to be used in place of a second image (histogram sliding), the operation can be treated as a point process on a single image. Point processing on dual images is particularly useful for functions which make composite images, detect motion, and alter

brightness (histogram stretching).

Logic operations consist of performing pixel-by-pixel bitwise AND, OR, XOR, and NOT operations among two input images. The AND function can be used to mask off portions of an image. The mask image, or matte, is composed of pixels having a binary value equal to all 0s (black), where the original image should be masked, or 1s (white) where it should be allowed to appear in the output image. Combining the two images with the AND operation produces the final masked image. The OR operation allows the compositing of two images, given that they do not spatially overlap. The XOR (exclusive-OR) combination can be useful for identifying identical pixels in two images.

Figure 2 illustrates this class, where $F(A,B)$ represents an arithmetic or logic function.

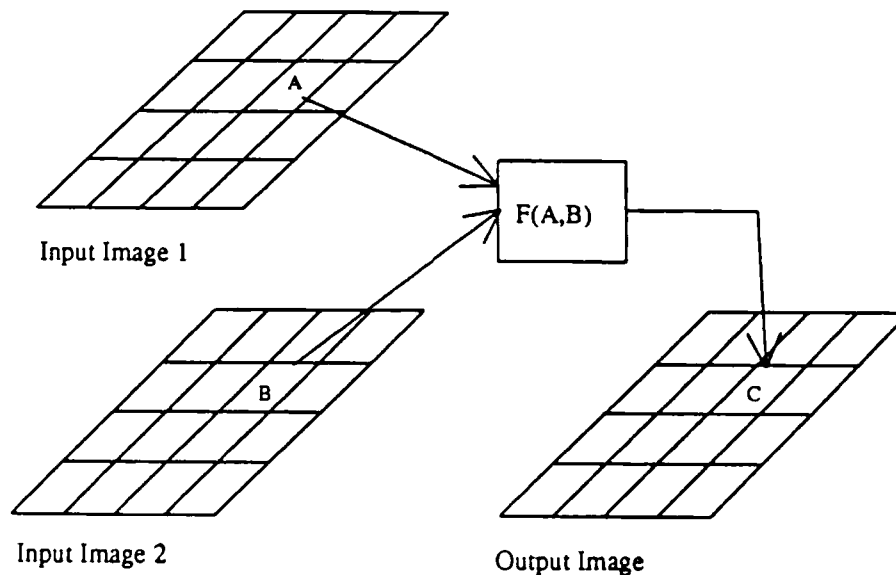


Figure 2: Point process / dual image

2.3. Neighborhood Processes

Neighborhood processes generate new values for a pixel based on the values of pixels in some neighborhood surrounding it. The size and shape of the neighborhood is defined by the user. A typical example is image smoothing (blurring, low-pass filtering) which involves replacing the luminance value of each pixel with a weighted average of luminance values of its neighboring pixels. Averaging over larger neighborhoods results in more smoothing, although 3×3 neighborhoods are typically used. The matrix of numbers that comprise the weights is known as the *convolution mask* or *kernel*. The spatial extent of the mask is known as the *window*. Spatial convolution is the process of shifting the mask across the image to different offsets, multiplying the superimposed mask values with the corresponding pixel values, and replacing the appropriate pixel in the output image with the sum of the product terms. Figure 3 illustrates the determination of an output pixel from the convolution mask and the underlying neighborhood. Convolution is a typical neighborhood operation. Other functions of this class include image

sharpening, edge detection (highpass filtering), and other forms of linear and nonlinear filtering.

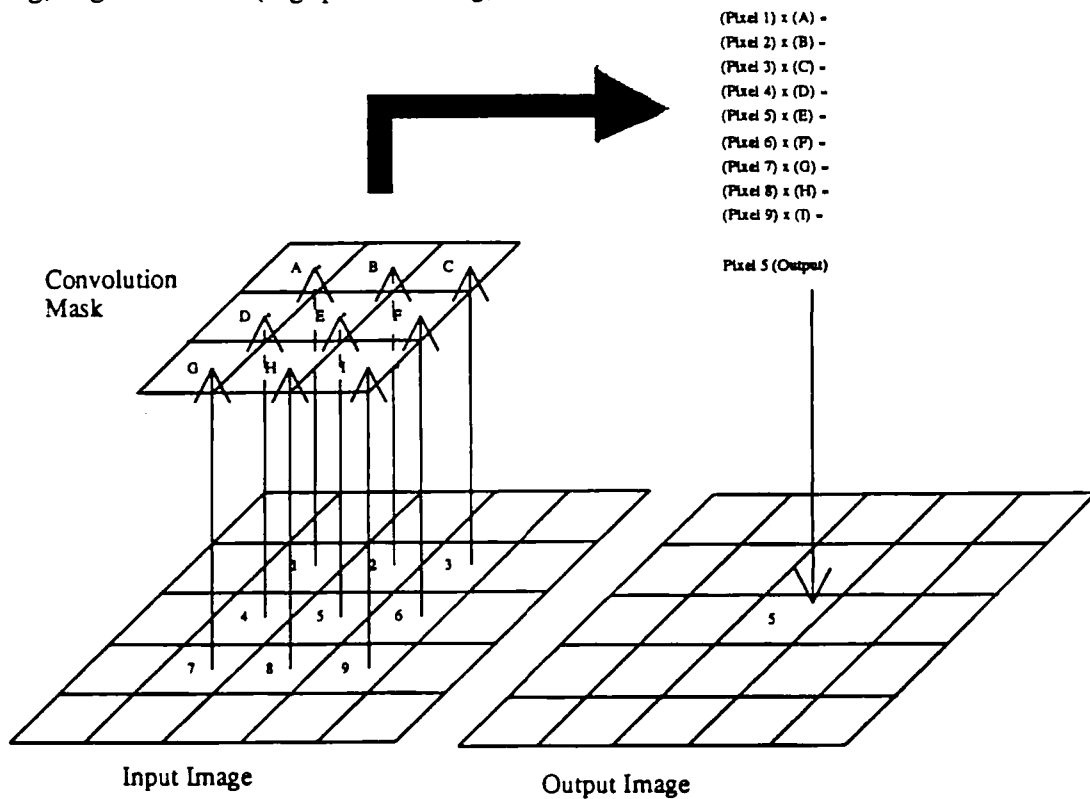


Figure 3: Neighborhood process

2.4. Geometric Processes

Geometric operations provide for the spatial reorientation of pixel data within an image. Typical operations include image scaling, rotation, and translation. In scaling and rotation, special care must be taken to avoid the artifacts that arise as a result of pixels not falling evenly on the sampling grid. This is characteristic of a problem in which the correspondence between the input and output images is not one-to-one and not oriented properly along the scanline direction. Technically, these artifacts arise from the undersampling of the image's higher spatial frequencies which give rise to a fictitious low frequency appearance[†]. The filtering used to combat this problem is called *antialiasing*. It consists of smoothing the image *prior* to the initial processing. This attenuates the high frequencies which are folding over to the low frequency components. Notice that geometric processes must therefore make use of neighborhood operations as a preprocessing antialiasing stage.

[†] Spatial frequency is the rate at which image brightness changes from dark to light. It is used in quantifying visual detail.

2.5. Transforms

Additional to the processes listed above are transforms. Frequency transforms (Fourier) are powerful operations that can be used to give a pictorial view of the spatial frequency component breakdown of an image. In addition, it can be used to aid in the specific filtering of undesired components. Image filtering may be carried out in the frequency domain in a more straightforward manner than convolution in the spatial domain. Although Fourier Transforms are computationally expensive, for large kernel sizes they are more practical, particularly using a Fast Fourier Transform (FFT) algorithm. Furthermore, a variant of Fourier Transform, called Discrete Cosine Transform (DCT), is useful for image compression. Since DCT decorrelates data (and consists exclusively of real numbers), an efficient quantization may be achieved over the spectrum of numbers to achieve an approximation of the image data with fewer bits. There exist other transforms, such as thinning and morphological operations, which yield alternate representations of the image. These will be discussed in more detail in the following sections.

The above summary of image operations is brief and incomplete. For more background into the subject refer to the books listed in section 10.

3. DESIGN PHILOSOPHY

This section reviews the system design issues addressed in IMPROC. The purpose of this discussion is to highlight system goals, features, and constraints. Among users, a greater awareness of these considerations is likely to enhance productive use of the toolkit environment. Among programmers, a fuller understanding of the design framework is critical to promoting uniform practices in software design and integration.

IMPROC was designed with the following issues in mind.

- 1) General format image representation
- 2) Uniform treatment of arbitrary data formats
- 3) Device independence
- 4) Easy code integration
- 5) "Keep it simple" philosophy

3.1. General Format Image Representation

Image data can be represented in many different formats. Underlying these formats are implicit assumptions about the data usage and its domain. For instance, images intended to be displayed on conventional frame buffers are usually represented in files as streams of pixels having three interleaved eight-bit color components. Images intended for use in image compositing operations will usually require an additional opacity component to each pixel. In other applications, images may contain floating point pixels as obtained from range data, surface normal estimations, and a variety of other input sources.

Despite the diversity of image formats, all images share many common filtering requirements. Consequently, image operators must treat general-format data uniformly. As a result, images must be represented in a single internal format that accommodates pixels with varying fields and precision. Establishing such a standard within IMPROC serves to maximize the utility of the library routines by allowing them to operate over a broad range of data. Furthermore, it separates IMPROC from the diversity of image file formats that exist.

Accommodating images of various data types has important consequences. Image processing routines that have traditionally been suitable for low precision images are now available for high precision data as well. Since one-dimensional lists of numbers often appear in high precision, they now become candidates for processing by the same image processing software. This opens the door to more innovative processing techniques since all data is subject to manipulation. Clever algorithms, for instance, can make use of multiple input sources to control the processing of images. Examples of this kind will appear throughout the paper.

Section 4 describes the internal image representation used in IMPROC. A review of the issues that were addressed in designing the format is given for clarity and insight. A more complete presentation of the actual data structure used is given in the programmer's manual in section 8.

3.2. Uniform Treatment of Arbitrary Data Formats

The general format is convenient for imposing uniform treatment upon image data that vary in size and type. However, in order to accomodate images with variable data types, each image processing routine must be prepared to work with all permutations of data types. This is achieved by having each routine convert the incoming image to the highest precision data type, compute the result, and convert the output image to the appropriate type (usually the same type as the input). The conversion procedure is known as *type casting*. Since images are often represented with low precision (one byte per pixel component), the software is made to test for this image type. If the image is found to satisfy this (common) special case, the computations are directly implemented with this data type, performing all necessary clipping operations. However, upon failure to satisfy this special case, the image is converted to an appropriate higher precision form upon which the image calculations take place. Since the necessary type casting consumes memory and time, the user is always advised to operate in low precision mode whenever possible. In other words, even if the ability to operate in any precision exists, there is no need to be wasteful.

3.3. Device Independence

IMPROC derives all of its flexibility from its collection of software tools. The library containing these tools is written in C and is portable across many machines. All frame buffer device-dependent details are isolated into a configuration definition file which specifies hardware specific features and software. The device independence offered by this software package has the obvious benefit of providing library routines that are accessible to a wide range of users and immune to the obsolescence of the supporting hardware. Consequently, the reliance on hardware functions are reduced to a minimum.

There have been no attempts at providing graphic interfaces for interactive use since they are outside the scope of the toolkit development. Instead, the image processing options are presented in a menu hierarchy that is invoked by entering corresponding option numbers. This method is direct, runs on all standard terminals, and can be easily upgraded to more sophisticated interfaces according to user needs. The menu interface is described in section 5.

It is, of course, productive to integrate special-purpose hardware if it is available. However, to keep in line with the emphasis towards portability, it is advised that a software-based version of the same operation be available as well.

3.4. Easy Code Integration

Serving its role as a software testbed for image processing algorithms, IMPROC has provisions for integrating new code easily. Only a simple three step procedure is necessary. Briefly, the programmer must install a menu entry, a function to collect user parameters, and a declaration of that function in the IMPROC header file. The details, and guidelines for new code, are given in section 9.

3.5. Keep It Simple

IMPROC is based on a philosophy which stresses the use of existing software tools to build larger, more sophisticated, routines. In this manner, it shares the same spirit as the UNIX programming environment [Kernighan 84].

For users, this synergism offers quick and powerful solutions to many image processing problems. Often, clever solutions will require the tools to be used in ways that were not originally anticipated. The ability to use the same set of library routines to process arbitrary data types and dimensions is an outgrowth of this goal towards conceptual simplicity.

For programmers, this approach maximizes the utility of the software. When new code can not make use of existing library routines, it should remain modular and simple so that it may serve as useful tools to others. Underlying much of the software design philosophy is that people's time is more valuable than computer resources. This is justified by the trend toward lower cost and higher performance hardware. Therefore, whenever this tradeoff emerged the decision has always favored the path most amenable to simplicity of design, readability, and maintenance.

4. IMAGE DATA

This section describes the internal representation of images in IMPROC. Knowledge of the internal image format is necessary to maximize full utility of the image operators available. In addition, descriptions of library routines will assume familiarity with the contents of this section. Motivation for the image format is included in the discussion for a fuller understanding of the design considerations.

4.1. Interleaved vs. Non-interleaved Format

Images generally exist as a two-dimensional array of data structures. Each data structure is a picture element, or *pixel*, and consists of several *fields*. Typically, monochrome images have only one field for luminance, while color images have three for the red, green, and blue (*RGB*) primary colors.

Pixel fields may be stored in interleaved or non-interleaved formats. Interleaved format is convenient when the entire pixel structure must be present together. Applications include pipeline processing, concatenation of image data, and compatibility with many file and display formats. Non-interleaved formats, however, are more convenient when it is desirable to modify the number and data type of pixel fields. In this form, no reshuffling of data is required upon modification of the pixel fields. Applications include pyramid construction, image composition, and arbitrary concatenation and type conversion of pixel fields.

Given the above tradeoffs, the non-interleaved format has been chosen due to its flexibility. In addition, programming issues are simplified by avoiding cumbersome indexing through the data, memory management becomes more efficient, and the conceptual view of the image becomes simple and concise. The only drawback of converting to/from interleaved format is relatively small when compared with the actual processing that takes place.

4.2. Channels

The non-interleaved form unscrambles the alternating pixel fields into *channels*, two-dimensional arrays each containing a field partition, e.g. a color component. Channels are commonly viewed as planes slicing through the pixels. There are typically three channels per image (*RGB*) and usually no more than four (*RGB* and opacity). IMPROC allows up to 16 channels per image, with more possible by resetting a system variable. Each channel has a distinct data type, allowing variable precision among the individual pixel fields. In this paper, we consider only the fundamental data types found in C: *unsigned char*, *short*, *int*, *long*, *float*, and *double*. Their sizes are 1, 2, 4, 4, 4, and 8 bytes, respectively, on most machines.

4.3. Image Types

Image processing operators often require semantic rules to define meaningful processing actions. These rules, for example, must know whether the image determined by the ensemble of channels is black-and-white, color, etc. Consequently, an *imgtype* attribute, defining the image

type (not data type), is associated with each image. Its purpose is to label the image with a code that routines may use for determining appropriate processing actions. Currently available *imgtype* values are given below.

Image Types			
<i>imgtype</i>	Channels	Data Type	Description
NULL_IMG	0	—	null image with no memory
DAT_IMG	0	—	frame buffer plotted data
BW_IMG	1	<i>unsigned char</i>	black-and-white image
MAT_IMG	1	<i>unsigned char</i>	matte image (for compositing operations)
BWA_IMG	2	<i>unsigned char</i>	black-and-white image with a matte
RGB_IMG	3	<i>unsigned char</i>	<i>RGB</i> color image
RGBA_IMG	4	<i>unsigned char</i>	<i>RGB</i> color image with a matte
YIQ_IMG	3	<i>short</i>	<i>YIQ</i> color image
VHS_IMG	3	<i>short</i>	(Value,Hue,Saturation) color image
LUT_IMG	1	<i>double</i>	one-dimensional look-up table
BIT_IMG	1	<i>unsigned char</i>	bitmap image
FFT_IMG	2	<i>float</i>	(Real,Imaginary) <i>FFT</i> image

Table 1: Image types

The numeric values assigned to these global variables are found in the IMPROC *ip.h* header file. Of course, user-defined images can be defined as well. By default, their *imgtype* attributes are initialized to the number of channels present.

Notice that without the *imgtype* attribute, there would be no way of distinguishing between, say, *RGB* and *YIQ* images. Although conceptually, both of these images can be treated identically, they have different domains and require different color conversion routines. Upon display, for example, a *YIQ* image must be converted to *RGB*. This information is crucial.

A second example can be given for the image compositing application. If an image is “held out” (like a cookie-cutter) by an image of type *BW_IMG*, the latter image must be converted to *BWA_IMG* before the operation can be performed. The conversion consists of appending a white channel to indicate constant opacity everywhere. However, had the latter image been of type *MAT_IMG*, no conversion would have been necessary. The *imgtype* attribute therefore provides the only semantic information about the image necessary for proper handling.

Each image type listed above has associated with it a default set of channel data types, shown in the “Data type” column. By default, all channels initially assume these types. At the user’s option, they may later be cast into other types to meet the needs of the application at hand.

4.4. Image File Formats

When an image is first read into the system, it must be converted into the canonical image format used by IMPROC. Image I/O routines are responsible for selecting and applying the appropriate conversion function. The selection is based on the filename suffix, or *tagname*, which identifies the image file format. Each image format must therefore have a unique tagname to identify itself from other formats.

Filenames are composed of two components, the *basename* and the *tagname*. The *basename* is a descriptive image name, and the *tagname* is a suffix preceded by a dot. A typical filename might be *sphere.r*. In this case, *sphere* is the *basename* and *r* is the *tagname*. The tagnames are known to the IMPROC I/O routines for proper image input and output among files. The currently supported file formats are listed below.

Image File Formats		
Tagname	Header	Data
rgb	$width_{32} height_{32}$	$red_8 green_8 blue_8$
bw	$width_{32} height_{32}$	$luminance_8$
U,S,I,L,F,D	$width_{32} height_{32}$	General format
p	$width_{16} height_{16}$	$red_8 green_8 blue_8 \alpha_8$
r	$0_8 0_8 40_{16} height_{16} width_{16}$	$red_8 green_8 blue_8$
lut	$length_{16} datatype_{16}$	List of LUT entries
cm	$length_{16} datatype_{16}$	List of colormap entries
sun1	SUN raster header	SUN bitmap

Table 2: Image file formats

The subscripts denoted above specify the length (in bits) of the associated number. Note that in all cases, the data is interleaved. In .p images, an additional α component is added to specify opacity information useful in compositing operations. The *datatype* entry refers to the values assigned to the channel data types in the IMPROC *ip.h* header file. The data types supported are: *unsigned char*, *short*, *int*, *long*, *float*, and *double*. Their respective *datatype* values are 0, 1, 2, 3, 4, and 5, and their tagname symbols are U, S, I, L, F, and D.

The most general format may be specified by concatenating the tagname symbols U, S, I, L, F, or D, to represent the channel data types. In this scheme, the number of tagname symbols denote the number of channels present, and the actual symbols determine their respective types. For example, the file *sphere.UUU* denotes an image with three interleaved channels, each of type *unsigned char* (identical to the *rgb* tagname). Types may be mixed, as in *cube.FLU* which contains three interleaved channels of type *float*, *long*, and *unsigned char*, respectively. Note that since a 16-channel limit is currently imposed, general format tagnames must not exceed 16 symbols.

5. INTERACTIVE USE OF IMPROC

IMPROC is a menu-driven image processing system. Before getting started with a description of the system, a brief explanation of the user interface is in order. There are three points which must be addressed before plunging ahead: the issues of screen layout, parameter collection, and image display.

5.1. Screen Layout

In doing image processing work it is critical to see more than one image on the monitor at once. Namely, it is desirable to see the input and output images simultaneously to quickly ascertain the result of the processing. As a result, IMPROC partitions the screen into square blocks, known as *quads*. Quads are identified by a numbering system that increases from left to right, and top to bottom. By default, the screen is split into four quads† as shown in Fig. 4, although any number may be specified by the user. All examples that follow will assume this default configuration.

Although the images may have to be scaled down in order to fit into the quads, this reduction in resolution is a small sacrifice for the benefit of comparing images and testing image processing algorithms over less data. Once an algorithm is fully tested and optimized in this testbed provided by IMPROC, it may then be put into a larger production system where interaction and visual feedback are no longer a priority.

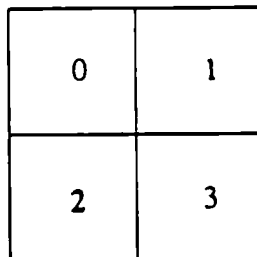


Figure 4: The four display quads

The quad containing the input image is referred to as *activeqd*, the active quad. It is clearly marked with a blue border. The output image is displayed in the next available quad, *nextqd*. *Nextqd* cycles around the screen, its value being incremented by one (modulo 4) after each image display. It, however, can never overwrite the *activeqd*, thus leaving the input image intact. Therefore, if the input image is currently in quad 1, subsequent image operations will display its output in quads 2, 3, 0, 2, 3, 0, etc. Options of course exist for allowing the user to change the *activeqd* and *nextqd* values.

† The term quad is derived from the default quadrant partition.

5.2. Parameter Collection

Now we discuss the method by which parameters are passed along to the system. At various times during the execution of a function, the user may be requested to supply parameters. The message prompts come in two formats:

Request for character string: <default>

Request for integer/real/hex: (min, max) <default>

In the first instance, a character string prompt is specified by two fields: the request and the default response. Defaults are always surrounded by '<' and '>', and are used whenever a carriage return is entered in response to a prompt.

In the second instance, a request is made for an integer, real, or hex number. Notice that this prompt is specified by three fields: the request, a range for the response, and the default value. The *min* and *max* values are always surrounded by '(' and ')'. This is not to be confused with the possible parentheses that may be contained in the request field. As long as the response does not lie within the range given (inclusive), the request is continually re-issued.

The distinction between an integer, real, or hex request is made as follows. If the number has any alphabetic characters between A and F (in upper or lowercase), it is in hex and a hex response is valid (i.e. 3F). If a decimal point appears in the number, then a real number is a valid response (an integer suffices as well). Any other number is an integer. If a real is given in response to an integer request, the fractional part is lost.

5.3. Image Display

Images must be properly converted to the appropriate data type and range required by the frame buffer before they can be displayed. Almost always, frame buffers require pixel intensities to lie between 0 (black) and 255 (white). IMPROC allows for two methods of constraining the range of displayed images: clipping and embedding.

5.3.1. Clipping

Clipping simply replaces all values below black and above white to black and white, respectively. It retains the dynamic range in the displayed interval while eliminating all information in the invalid range.

5.3.2. Embedding Intensities Into the Valid Range

Embedding refers to a combination of histogram sliding and histogram stretching performed on each channel independently. This means that additive and multiplicative constants are applied to the image to respectively slide and stretch the dynamic range. The result must, of course, lie within the valid range. In contrast to clipping, embedding intensities does not discard information from view. However, the contrast of the image may diminish due to excessive

values that force the dynamic range to undergo extreme compression.

These operations makes use of I_{min} and I_{max} , the minimum and maximum image channel values. By comparing them against black and white, we can determine the sequence of actions necessary to embed the image in the valid range. The following four cases are considered.

Embedding Intensities Into the Valid Range		
Test Condition	Comment	Action
$I_{min} > \text{black}, I_{max} < \text{white}$	Range is already valid.	No action.
$(I_{max} - I_{min}) > (\text{white} - \text{black})$	Range exceeds black/white difference. Must anchor minimum to black, and maximum to white.	Slide range by adding $-I_{min}$ to all values. Then scale resulting values so that $I_{max} - I_{min}$ is fixed at white.
$I_{min} < \text{black}$	Range must be slid up to anchor I_{min} to black.	Slide range by subtracting I_{min} from all values.
$I_{max} > \text{white}$	Range must be slid down to anchor I_{max} at white.	Slide range by adding $(\text{white} - I_{min})$ to all values.

Table 3: Embedding the intensity range

Note that the above set of conditions are tested in the order given. Therefore, we first check whether the range is already valid. If this test fails, then we test to see whether the range exceeds the 8-bit display range. If so, histogram sliding and stretching is necessary. Otherwise, we continue to test for the direction in which histogram sliding must take place.

The result of clipping or embedding is an image which satisfies the frame buffer constraints for display purposes. In addition to displaying this image, the user has the option of storing the result back into the originating quad.

Clip(-/+1) or Embed(-/+2) to 0-255 range? (-2,2) <1>

A reply of 1 or 2 will display a clipped or embedded image, respectively. This permits the internal image to retain its precision while its lower precision counterpart is displayed. In this mode, “what you see is *not* what you get.” However, responding with a negative value will cause the contents of the quad to be overwritten with the displayed image. This latter option enforces “what you see is what you get.”

5.4. Command Line Arguments

IMPROC can be invoked with several options:

`improc [—b] [—gnum] [—qnum] [—snum] [—xnum] [—ynum]`

- b selects the secondary frame buffer as the raster device (/dev/raster1).
- gnum sets the maximum valid gray level to be *num*. All intensities computed to be greater than *num* are clipped to that value.
- qnum partitions the screen into *num* quads.
- snum sets the quad size to *num*.
- xnum sets the leftmost quad column to *num*.
- ynum sets the topmost quad row to *num*.

If any of the above options are not specified, default values are used. By default, the maximum gray value is 255, and the screen is partitioned into four quads, centered on the screen, each having dimensions of 256×256 . The examples throughout this paper will assume this default configuration.

Input to IMPROC may be redirected from a file instead of the terminal. The file contains the same menu options and parameters that would have been entered directly. The only format provisions is that each number must be on a separate line. Furthermore, lines beginning with a '#' symbol serve as comments and are ignored. This redirection is useful if the user has a particular sequence of commands that is used repeatedly. Although the file is usually generated from the **Show history** option in the main menu, it may be edited at will by the user. When the file is consumed, IMPROC will reopen the terminal as the source of its input. Therefore, the input file serves to place the user in a particular state from which direct user interaction will proceed.

5.5. Getting Started

The program is invoked by typing "improc" at the command level. A list of menu options will flash by, and the user will then be prompted for a filename containing the input image:

Filename: <junk.r>

The name *junk.r* is a default filename. The user-specified file is then opened and read. If all is in order, a message is printed on the terminal indicating the height, width, and additional positioning information for the image. Note that if the specified file could not be opened, the user is continually prompted for a filename until one is found. At this point, IMPROC must be told whether to interpret the contents of the file as a black-and-white image or as a color image.

Color image? (0=no; 1=yes) (0, 1) <1>

Although most applications would normally retain the color information, this request is inserted to reduce processing time on images that are known in advance to be black-and-white. This is the case since treating color images involves three times the effort and space (*RGB* channels).

The system will now display the image in quad 0 (default), surrounded by a blue border indicating it as the active quad. The user can now select an operation by entering the desired option number.

Since it is not uncommon to forget the option numbers or have the list scroll off the terminal screen, the default option was chosen to be "Menu display." The user may thus enter a carriage return to see the list again. This practice is done uniformly in all the submenus of the system as well.

5.6. Menu Hierarchy

The image processing operations are organized in a hierarchy. The main menu offers options for image I/O operations and entry into submenus. Each of the basic classes discussed in section 2 are contained in separate submenus. Since the software continues to evolve (as it should since it is intended to encourage users to add their code), the description of the supported functions and the menu entries may be subject to change. Figure 5 illustrates the current menu hierarchy.

Improc	Point Ops	Neighborhood Ops	Geometric Ops
Menu display	Menu display	Menu display	Menu display
Set next quad	Set next quad	Set next quad	Set next quad
Set active quad	Set active quad	Set active quad	Set active quad
Read image from file	Histogram	Blur	Scale
Write image to file	Threshold	Blur with mask	Rotate
Point Ops	Threshold (color)	Edge preserving blur	Translate
Neighborhood Ops	Clip	Matte-normalized blur	Warp
Geometric Ops	Quantization	Image sharpening	Exit submenu
Arithmetic Ops	Histogram flattening	Convolution	
Logic Ops	Histogram equalization	Median filter	
Matte Ops	Intensity scaling	Laplacian image	
Image Conversion	Dither (Unordered)	Thresholded Laplacian	
Image Transforms	Dither (Ordered)	Relief map effect	
Image Intensity Plot	Dither (diffuse error)	Exit submenu	
Colorize	Halftone		
Shell Command	Make LUT		
Run script	Edit LUT		
Show history	Apply LUT		
Quad status	Exit submenu		
Exit			

Arithmetic Ops

Menu display
Set next quad
Set active quad
Image1 - Image2
Image1 - const
Image1 + Image2
Image1 + const
Image1 * Image2
Image1 * const
Image1 / Image2
Image1 / const
A + (B-C)*const
Histogram
Exit submenu

Logic Ops

Menu display
Set next quad
Set active quad
Image1 & Image2
Image1 & const
Image1 | Image2
Image1 | const
Image1 xor Image2
Image1 xor const
Image1 bic Image2
not Image1
Exit submenu

Image Intensity Plot

Menu display
Set next quad
Set active quad
Plot
Set gray level for traces
Set plot scale
Set z scale
Exit submenu

Matte Ops

Menu display
Set next quad
Set active quad
Cut out subimage
A cuts B when $A > 0$
A over B when $A > 0$
Matte cut $(A)(aA)$
A over B $(A)+(B)(1-aA)$
A in B $(A)(aB)$
A out B $(A)(1-aB)$
A atop B $(A)(aB)+(B)(1-aA)$
A xor B $(A)(1-aB)+(B)(1-aA)$
Hicon A (aA)
Darken A $(f*nonalphaA)$
Opaque A $(f*aA)$
Dissolve A $(f*A)$
Read pixel value
Read pixel location
Exit submenu

Image Conversion

Menu display
Set next quad
Set active quad
Convert channel type
Convert image type
Extract channels
Append channels
Copy channels
Color shift
Set global args
Exit submenu

Image Transforms

Menu display
Set next quad
Set active quad
Fourier transform (1D)
Fourier transform (2D)
Thin
Exit submenu

Colorize

Menu display
Set next quad
Set active quad
Display Images
Color Palette
Cut out subimage
Color entire region
Color gray levels
Fast region coloring
Thresholding
Read pixel value
Read color masks
Save color masks
Save color image
Change picture
Exit submenu

Display Images

Menu display
Set next quad
Set active quad
Display picture
Display mask
Display colored image
Highlight (un)colored regions
Exit sub-submenu

Color Palette

Menu display
Set next quad
Set active quad
List colors
Add colors
Modify color
Display all colors
Display group
Select and modify
Init and modify
Regenerate color maps
Exit sub-submenu

Figure 5: Menu hierarchy

6. DESCRIPTION OF MENU OPTIONS

This section describes the entries in the various IMPROC submenus. Each subsection is devoted to a different submenu. Examples of message prompts, parameters, and applications are given for clarification.

6.1. Main Menu

- **Menu display:** List the menu options and their option numbers on the terminal screen. Since this command is always the default option (for every submenu), the user must only enter a carriage return to list the menu choices.
- **Set next quad:** Set the next quad variable, *nextqd*, to the specified value. The default value is the current *nextqd* value. Changing this value is useful to prevent overwriting desirable quads. By default, the *nextqd* variable increments (modulo 4) after each image display. This results in a cyclic display order, with the exception of automatically skipping over the active quad.
- **Set active quad:** Set the active quad variable, *activeqd*, to the specified value. The default provided in the message prompt is the current *activeqd* value. This option is used to specify the quad containing the input image. It is necessary only when selecting a new input image.
- **Read image from file:** Read an image from a file. The user is prompted for the filename, as well as information on whether it should be treated as a color or black-and-white image. This information is identical to that requested when first entering IMPROC. An example is given in section 5.
- **Write image to file:** Save an image to a file. The following sequence of message prompts and responses illustrates this command.

```
Quad (-1=all):    (-1,3) <activeqd> 1
Filename:         <junk.r>          file.r
Provide scale(0) or dimensions(1) ? (0,1) <0> 0
Xscale:          (0,max_xscale) <1.> 1.
Yscale:          (0,max_yscale) <1.> 1.
```

The user first enters the quad number containing the image to be saved. A -1 response is entered to save the whole screen. The filename is then specified by the user. Here, as in other requests, a response that cannot be satisfied causes the request to be issued again. Causes for failure include files that fail to open or parameters that do not fall within the specified range (the numbers between the parentheses). Finally, the user may select to save the image in any arbitrary size. The parameters to rescale the image can be relayed as scale factors or dimensions. In the example above the user entered 0, thereby desiring to input scale factors. The factor in the *x* direction (along scanline) can take on any value between 0 and *max_xscale*, where *max_xscale* is the maximum allowed resolution, *XMRES*, (in *x* direction) divided by the current width. Selecting it will scale the *x* direction so that the resulting width will be *XMRES*. Currently, this

value is 1280 so that it may be displayed on High Definition Television (HDTV) monitors. The default value of 1. will leave the image size intact. An identical request occurs for the scale factor in the y direction (perpendicular to scanline), where *YMXRES* is 1028. Had the user desired to enter dimensions, a 1 would have been entered:

```
Provide scale(0) or dimensions(1) ? (0,1) <0> 1
Height: (1,YMXRES) <height> 256
Width: (1,XMXRES) <width> 256
```

Here the user selects a 256×256 final image size. The default values of *height* and *width* are the current dimensions of the image. Entering two carriage returns would have therefore left the image size intact. Note that image rescaling here only specifies the size of the image to be saved in the file. It does not alter the size of the image resident in the quad.

- **Point Ops:** Select the submenu containing point operations.
- **Neighborhood Ops:** Select the submenu containing neighborhood operations.
- **Geometric Ops:** Select the submenu containing geometric operations.
- **Arithmetic Ops:** Select the submenu containing arithmetic operations.
- **Logic Ops:** Select the submenu containing logic bitwise operations.
- **Matte Ops:** Select the submenu containing image compositing operations.
- **Image Conversion:** Select the submenu containing image conversion operations. This includes routines for channel type casting, color space conversion, and image reconfiguration.
- **Image Transforms:** Select the submenu containing transform operations.
- **Image Intensity Plot:** Select the submenu containing 3-D plotting routines.
- **Colorize:** Select the colorization submenu to apply color to monochrome images.
- **Shell command:** Enter interactive ksh. This permits the user to temporarily leave the IMPROC environment and return to the shell. Note that the frame buffer is made available to other users during that time. To re-enter IMPROC, simply type exit, *<ctrl> C*, or *<ctrl> D*.
- **Run script:** Run a user-specified program given in file *ipdo.c*. This option allows users to conveniently test new algorithms without having to make any serious modifications to the IMPROC source code. By replacing the default dummy function in *ipdo.c* with new code, the user must only recompile the package (explained in section 9) in order to integrate the test routine into the system. This proves workable when users are allowed to have direct access to the source code, or maintain private copies.
- **Show history:** Print the history of the IMPROC session.
- **Quad status:** List the entries to the quad and channel data structures.
- **Exit:** Exit program; terminate session.

6.2. Point Operation Submenus

This section describes the point operations that are supported. They are divided into three submenus: Point Ops, Arithmetic Ops, and Logic Ops. Operations that exclusively deal with the arithmetic and logic (bitwise) combination of images are found in the Arithmetic Ops and Logic Ops submenus, respectively. The remainder are contained in the Point Ops submenu.

6.2.1. The Treatment of Color Images

Before considering the operations available, it is necessary to address the issue related to the treatment of color images. Since color images have three channels (*RGB*), the question naturally arises as to whether the channels should be treated independently or coupled.

If we choose to couple the channels, we must operate on the luminance component since it implicitly couples the *RGB* information. The results of processing the luminance image then dictate the new values for the color channels. In particular, the luminance channel, C_1 , undergoes processing, yielding a monochrome output image, C_2 . For each pixel p_{C_2} in C_2 , the following computations are applied.

```
IF( $p_{C_2} > p_{C_1}$ ) {  
     $f = \frac{(p_{C_2} - p_{C_1})}{(255 - p_{C_1})}$   
     $p_R = p_R + (255 - p_R) \times f$   
} Else {  
     $f = \frac{p_{C_2}}{p_{C_1}}$   
     $p_R = p_R \times f$   
}
```

The above rule first checks whether p_{C_2} is closer to white than p_{C_1} . If this condition is found to be true, f is initialized with the normalized distance of p_{C_2} from p_{C_1} . It measures the proximity of p_{C_2} from white. This factor is then applied to p_R , a corresponding color component (i.e. red) to yield a value that is proportionately pulled to white. However, if p_{C_2} is found to be less than p_{C_1} , f reflects the normalized distance from black. Applied to p_R , the resulting value is similarly pulled towards black. This approach generates intuitive results since all three channels undergo the same transformation.

Decoupling the *RGB* channels often yields unpredictable and visually interesting images. It is useful in neighborhood operations where the output value is a weighted average of its neighbors. It is not recommended for operations that make abrupt value reassignments. Examples of the latter include thresholding, quantization, and histogram equalization.

The option of decoupling color images is common to a variety of point and neighborhood processes. The following message prompt appears when necessary:

Decouple *RGB*? (0=no; 1=yes) (0,1) <0>

The default is 0 (for *no* decoupling) since it yields more predictable results. Selecting to decouple the *RGB* channels increases the entropy of the resulting image. This has the visual effect of increasing the range of colors in the output image.

6.2.2. Point Ops Submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Histogram:** Display histogram of the image channels. A histogram is the graphical representation of the intensity distribution of an image. The horizontal axis denotes intensity, and the vertical axis represents the number of pixels at the given intensity. A histogram presents a clear indication of image contrast and brightness dynamic range. The user must specify which histogram to plot:

Channels (in hex): <ffff>

The supplied hexadecimal number indicates the channels which will be used for the histogram display. The default option of ffff specifies all channels. Note that the channels are encoded as bits in the format (blue,green,red). That is, the least significant bit represents the red channel. Therefore, to specify the red and green channels we want bits 00...011, or hex number 0003. The green and blue channels are denoted by bits 00...0110, or hex number 0006, etc. Any request that refers to a non-existing channel is ignored.

The histograms are superimposed and plotted in a cycle of red, green, blue, and white colors, with each cycle using darker intensity levels. Since the range of each channel may be arbitrary, there are tick marks displayed along the horizontal axis, with their corresponding values specified on the terminal. In addition, the minimum and maximum values for each range is provided as well.

- **Thresholding:** Threshold the input image.

Lower threshold level: (0,256)	<128>	95
Higher threshold level: (95,256)	<95>	150

The user selects two threshold levels, $T1$ and $T2$, with $T1 \leq T2$. All pixels with gray values below $T1$ are displayed as black. Those greater than or equal to $T1$ and less than $T2$ are displayed with a single gray value. Pixels having gray values greater than or equal to $T2$ become white. Displaying the thresholded image with these three levels offers additional visual information than the standard bilevel format. The standard black-and-white thresholded image

can be obtained by setting $T1$ equal to $T2$. Note that $T1$ is the default value for $T2$.

- **Threshold (color):** Threshold the input image about a user-specified color.

Read value (0=no; 1=yes)

Confirm: (0=no; 1=yes) (0,1) <1>

Color space: (0=RGB; 1=YIQ; 2=VHS) (0,1) <0> 0

Tolerance in R: (0,r) <0>

Tolerance in G: (0,g) <0>

Tolerance in B: (0,b) <0>

The user positions the activated mouse upon a desired color pixel. The color value of that pixel is printed on the terminal by entering a 1 to the first message prompt. As long as the user fails to enter 1 to the confirmation prompt, the user may continuously sample points. Once the loop is broken with an affirmative confirmation, the last sampled point is used to threshold the color image. The user must first specify the color space in which to perform the operation. Either the *RGB*, *YIQ*, or *VHS* color spaces may be selected. In the example above, the *RGB* space is used. The *r*, *g*, and *b*, values represent the maximal allowable deviations from the red, green, and blue color components, respectively. Their values are limited by their distance to either range limits.

- **Clip:** Clip the input image to a specified range.

Lower clip level: <0.>

Higher clip level: <255.>

The user selects two clip levels, $C1$ and $C2$, with $C1 \leq C2$. All image values below $C1$ or greater than $C2$ are set equal to $C1$ or $C2$, respectively.

- **Quantization:** Perform gray scale quantization on the input image.

Number of levels: (2,256) <8>

Method: (0=scale; 1=plain) (0,1) <0>

The number of gray levels, L , used to display the image is first specified. Next, the user must select one of two modes in which to perform quantization. In quantization, the full gray scale range is divided into L equal intervals. The *plain* mode simply reassigns gray values based on which interval they lie. The *scale* alternative performs intensity scaling on the input image before reassigning gray levels. This makes the input image occupy the full dynamic range prior to quantization. The user must select whether to decouple a color input image.

- **Histogram Flattening:** Reassign gray levels in order to achieve a flat histogram over the entire intensity range for the input image. This corresponds to an image with maximum entropy — exhibiting the widest possible dynamic range. Visually, this enhances contrast and accentuates details that were not readily apparent originally. This operation is useful for normalizing images

before further processing. It has been used extensively for enhancing medical X-rays and satellite imagery. Again, the user must select whether to decouple a color input image.

- **Histogram Equalization:** Reassign gray levels in order to achieve a user-specified histogram shape for the input image. For each image channel, the user must specify a look-up table (LUT) which is used to drive the equalization procedure. The LUT defines the *shape* of the resulting histogram. As a result, the LUT values can take on any range, and are not required to specify absolute histogram values.

```
LUT 0: (-1=file; o/w quad)    3
LUT 1: (-1=file; o/w quad)    -1
LUT file:    <>                hist.lut
LUT 2: (-1=file; o/w quad)    2
```

The LUT may come from a file or a quad. In the example above, the image has three channels, and the LUTs are found in quad 3, file *hist.lut*, and quad 2, respectively.

- **Intensity scaling:** Scales the intensity range of the input image to fit user-specified limits.

```
Lower limit:    (0.,255.)    <0.>
Higher limit:   (10.,255.)   <255.>
```

The user selects two scaling limits S_1 and S_2 , with $S_1 \leq S_2$. The minimum and maximum image values are mapped to S_1 and S_2 , respectively. All intermediate values are linearly scaled appropriately. Note that S_1 is the default value for S_2 . Again, the user must select whether to decouple a color input image.

- **Dither (Unordered):** Apply random noise to the input image prior to performing quantization. This technique attempts to diffuse the false contour artifacts present in quantization by sprinkling random noise on the image as a preprocessing step. In doing so, harsh edges in the quantized image become less noticeable since the added randomness push some pixels to neighboring intensity intervals.

```
Number of levels: (2,255)    <8>
Gamma (>1 darkens result):  (0.,5.)    <1.>
```

As in the quantization option, the user must enter the number of gray levels in which the image is to be displayed. Prior to dithering, the image is passed through a look-up table to alter the intensities for gamma correction. This serves to modify the perceived brightness of the image.

It replaces all values v by $v \times (\frac{v}{255})^\gamma$. This relation yields the family of curves shown in Fig. 7.

Note that $\gamma > 1$ corresponds to the curves that lie below the ramp and therefore darkens the result. In contrast, the family of curves having $\gamma < 1$ lie above the ramp and thus brighten the result. The ramp coincides with $\gamma = 1$, denoting that the output image is a copy of the input.

Picking excessive values will cause saturation. Also, notice that the black and white extremes stay intact.

- **Dither (Ordered):** Perform ordered dither on the input image to yield a bi-level output. In ordered dither, the output value of each pixel is dependent on its intensity and location, and on an $n \times n$ dither matrix D . Each pixel is thresholded on the basis of the value in the corresponding position of D . Larger dither matrices yield better results due to the fewer constraints they impose on the periodicity of the texture patterns. Furthermore, larger matrices accommodate more gray levels in the output image. A complete description can be found in [Foley 82].

Dither matrix size 4 or 8 (0=4; 1=8) (0,1) <1>
Gamma (>1 darkens result): (0.,5.) <1.>

The user must select between the use of a 4×4 or 8×8 dither matrix. The elements in these matrices are those given in the above reference. The user must also select a gamma value for altering the image brightness prior to dithering. This operation is necessary to compensate for the display device resolution and the visual perception of brightness driven by the discrete placement of dots. See the first dither entry for more detail.

- **Dither (error diffusion):** Dither the input image using error diffusion algorithms. Error diffusion dithering techniques are generally considered to be the best methods for converting continuous-tone images into bi-level output. The central idea here is that errors generated between the true value and its thresholded output should be spread to neighboring points for compensation. It is therefore a sequential adaptive thresholding technique where the threshold values are biased based on errors accrued from neighboring points. This approach serves to overcome the sharply noticeable textured characteristics of the other dithering approaches.

Method: (0=Floyd-Steinberg; 1=Jarvis) (0,1) <1> 0
Coefficients (0=default; 1=enter) (0,1) <0> 1
f0: <.4375>
f1: <.1875>
f2: <.3125>
f3: <.0625>
Gamma (>1 darkens result): (0.,5.) <1.>

Two error diffusion methods are offered: Floyd-Steinberg [Floyd 75] and Jarvis-Judice-Ninke [Jarvis 76]. The only difference between the Floyd-Steinberg and Jarvis approaches is the area in which the error is spread and the applied weights. Figure 6 shows the neighborhoods in which the error is diffused about pixel X for both cases.

The four default weights, (f_0, f_1, f_2, f_3) , used in the Floyd-Steinberg method are $(\frac{7}{16}, \frac{3}{16}, \frac{5}{16}, \frac{1}{16})$. Those used in the Jarvis method are $(\frac{7}{48}, \frac{5}{48}, \frac{3}{48}, \frac{1}{48})$. The user is free to enter arbitrary weights by replying appropriately to the request for coefficients, as shown above.

Notice that the defaults supplied above while entering coefficients are those values recommended by the algorithm. Again, the user must select a gamma value for altering the image brightness prior to dithering. See the first dither entry for more detail.

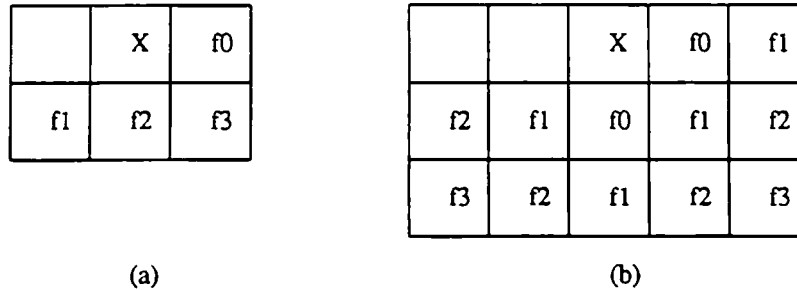


Figure 6: Error diffusion neighborhoods: (a) Floyd-Steinberg and (b) Jarvis methods.

- **Halftone:** Perform digital halftoning on the input image to generate a bi-level output. The halftone procedure replaces each pixel p with an $n \times n$ set of pixels s , called a superpixel. The number of white pixels in s is proportional to the intensity of p . These pixels are organized to take on a spiral pattern. This attempts to simulate the analog halftone process in which the area around each superpixel center is inversely proportional to the intensity.

Superpixel sz: (1,10) <4>
 Gamma (>1 darkens result): (0.,5.) <1.>

The user is requested for the dimension of the superpixel block. Note that the maximum block size is 10×10 . Again, the user must select a gamma value for altering the image brightness prior to dithering. See the first dither entry for more detail.

- **Make LUT:** Generate a look-up table (LUT) and apply it to an input image. The user is prompted to specify intensity ranges, and corresponding constants for the ranges which are then applied to the input image.

LUT dimension: (0,1000) <256> 256
 0) UCHAR
 1) SHORT
 2) INT
 3) LONG
 4) FLOAT
 5) DOUBLE
 LUT type: (0,5) <0> 5

```

Xfrom:  (0,255)      <0>      30
Xto:    (30,255)     <255>    80
Yfrom:  (-1000.,1000.) <0>      20
Yto:    (-1000.,1000.) <len>   120
Op: (0=exp; 1=ramp)  (0,1) <1>  1

```

```

Xfrom:  (80,255)     <80>      80
Xto:    (80,255)     <255>    255
Yfrom:  (-1000.,1000.) <120>   120
Yto:    (-1000.,1000.) <255>   200
Op: (0=exp; 1=ramp)  (0,1) <1>  0
Exp: (0.,100.)      <1.> .5

```

In the example above, a LUT is created with 256 entries of type *double*. By default, the LUT is initialized with a ramp. That is, all inputs pass untouched onto the output. Intervals of input values that are to be reassigned are specified by responding to the Xfrom and Xto prompts. The reassignment values are determined by responding to the Yfrom and Yto prompts.

The shape of the curves in the intervals may be given as a ramp or an exponential function. By default, the ramp is selected to yield linear interpolation between the user-specified interval endpoints. If the alternate option is chosen, an exponent must be supplied. The family of exponential curves is defined by the function

$$y = (y_2 - y_1) \times \left(\frac{x - x_1}{x_2 - x_1} \right)^{\text{exponent}}$$

This class of functions is given in Fig. 7. The $(y_2 - y_1)$ term guarantees that the interval endpoints equate to the user's entries. The second term normalizes the distance of the current point between the interval endpoints. Since its value is between 0 and 1, we can take advantage of the properties of f^{exponent} . Note that $\text{exponent} = 1$ yields a ramp. As exponent increases, curves are generated that lie farther below the ramp. When exponent decreases above 1, curves lie farther below the ramp.

The above sequence iterates until a range specifying the LUT entry for the maximum entry (255) has been defined. Even if no assignment of 255 is desired, it must be specified since it is recognized as the exit condition. In that case, the user may simply specify 255 for Xfrom, Xto, Yfrom, and Yto. This will avoid redefining the entry for that interval while, at the same time, signaling the termination of this operation.

- **Edit LUT:** Edit LUT. The procedure is analogous to **Make LUT**.
- **Exit submenu:** Return to the main menu.

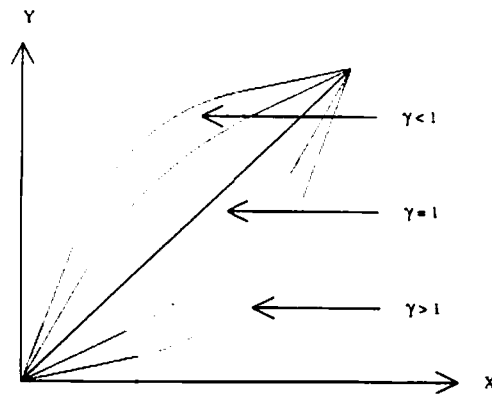


Figure 7: The family of exponential LUT curves

6.2.3. Arithmetic Submenu

All arithmetic operations are contained in this submenu. When combining images of different sizes or positions, the system automatically pads the two images with 0 (black) so that they are mutually superimposed. Note that the original dimensions of a padded image are not restored after the operation is executed (i.e. images retain their padding). Furthermore, when combining a color image with a monochrome image, the system will correctly duplicate the operations between each of the 3 color channels and the single monochrome channel.

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Image1 - Image2:** Subtract *image 2* from *image 1*.

Image1: (0,3) <activeqd>
Image2: (0,3) <activeqd>

The user is prompted for the quads containing these images. If the data type of the image is *unsigned char*, all negative values are clipped at 0. A message of the form *bctr=number* is printed, stating that *number* pixels were computed to be negative and clipped. Alternatively, *unsigned char* subtraction computations may be evaluated with the absolute value taken instead. This is achieved by setting the *ABSVAL* global variable to 1 in the *Set args* option. Two predominant uses for image subtraction are motion detection and background subtraction. Medical blood flow analysis and X-ray imagery often use this type of operation.

- **Image1 - const:** Subtract a constant value from all pixels of the specified image. This operation has the visible effect of darkening the image. Since the histogram of the processed image is only shifted downwards towards the lower intensities, the contrast of the output image will be identical to that of the input image. Again, all negative values in *unsigned char* computations are clipped at 0 unless *ABSVAL* is 1. Note that there is no clipping and *ABSVAL* is ignored if the computations are performed on elements with higher precision. Clipped pixels contribute to gray level saturation which results in a visual loss of intensity resolution. To preclude saturation

effects, it is recommended to select a constant that is less than the minimum gray level present in the image.

- **Image1 + Image2:** Add two images together. All results are clipped at white, the maximum gray level. The user is made aware of pixels that are clipped by a message of the form *wctr=number*. This states that *number* pixels were computed to be above the maximum gray level and clipped.
- **Image1 + const:** Add a positive constant value to an image. This operation shifts the histogram towards the higher intensity range, thereby brightening the image. Again, the results are clipped at white. Earlier remarks about saturation effects apply here as well.
- **Image1 * Image2:** Multiply two images together.
- **Image1 * const:** Multiply an image by a constant. The constant may be a real number. If the number is negative and the channel data type is *unsigned char* the magnitude of the result is subtracted from white. Therefore, multiplying an *unsigned char* image by -1 yields a negative image having the same appearance as a film negative of the original. That is, black pixels become white, white pixels become black, and the intermediate gray levels take on their respective reverse brightnesses. Saturation and quantization become more pronounced with larger constants.
- **Image1 / Image2:** Divide *image 1* by *image 2*. In order to avoid divide-by-zero errors, the output image takes on the value of *image 1* at all positions where *image 2* is 0.
- **Image1 / const:** Divide all pixels of the specified image by an integer constant.
- **A + (B - C) * const:** Add image *A* to the scaled difference of *B* and *C*. This operation is used to implement bandpass filtering, where *A* is the original image, *B* is a blurred copy, *C* is more blurred, and *const* is a brightness factor. This is introduced as a convenient shorthand for the necessary steps. Note that the $(B - C)$ term above is correctly left unclipped in the calculation.
- **Exit submenu:** Return to the main menu.

6.2.4. Logic Ops Submenu

- **Menu display:** List menu options and their option numbers.
- **Image1 & Image2:** Perform the logic pixel-by-pixel AND operation among the two specified images. This operation is usually used to mask off portions of an image. One of the images serves as a mask, containing black wherever the second image is to be masked, and white wherever the second image is to be allowed to appear.
- **Image1 & const:** AND each pixel of *image 1* with the specified constant.
- **Image1 | Image2:** Perform the logic OR operation among the two specified images. This operation is used to add together subimages into a composite output image. Given that two subimages do not spatially overlap and are masked, they may be ORed together combining both into a single output image.
- **Image1 | const:** OR each pixel of *image 1* with the specified constant.
- **Image1 XOR Image2:** Perform the EXclusive-OR operation among the two specified images. This operation may be useful for identifying those pixels which are identical in both images. Identical pixels will be displayed as black, otherwise they will be something other than black, depending on the actual bit-for-bit comparison.
- **Image1 XOR const:** Perform an XOR operation on *image 1* with a specified constant.
- **Image1 BIC Image2:** Non-black pixels in *image 2* mask out pixels in *image 1*. Black pixels in *image 2* allow the corresponding pixels in *image 1* to be displayed. This is an alternate instance of the AND operation in which the mask values must be black or white to achieve equivalent results.
- **NOT Image1:** Complement the pixels in the specified image. This achieves a negative image in which black pixels become white, white pixels become black, and intermediate gray levels take on their corresponding reverse brightness. This operation is useful in analyzing details characterized by small brightness deviations in the white regions. Due to the logarithmic response of the eye to intensity, equal brightness deviations in the black regions are more visible than that in the white regions. Therefore, complementing an image may be helpful in detecting these changes by bringing them into the darker intensity range.
- **Exit submenu:** Return to the main menu.

6.3. Neighborhood Ops Submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Blur:** Blur the input image with a box filter. Also known as low-pass filtering and image smoothing, this operation is useful in removing visual noise, albeit it sacrifices edge clarity. Each output pixel value is simply the average of the neighboring pixels. Although a weighted average is generally more desirable than an unweighted average, the latter is used here since it affords high computational savings. (It is based on an optimal implementation of summed-area tables based on [Crow 84]).

Convolution with a box filter is synonymous with unweighted averaging. For improved results, the image may undergo several recursive blurring iterations. This filter, known as repeated box filter, is discussed in [Heckbert 86]. Due to the Central Limit Theorem, repeated convolution with any single filter ultimately approximates a Gaussian (bell-shaped) filter. Gaussian filters are noted for their smooth frequency attenuation properties[†].

The user is requested for a window size of the box filter. The window dimensions are taken as floating point numbers. This allows blurring with subpixel accuracy thereby permitting smoothly varying blur operations. Increasingly blurred images are generated by using larger window sizes.

Directional blur may be achieved by selecting different dimensions for the window height and width. This, however, only yields directional blur along the horizontal or vertical directions. We may achieve this effect for any orientation by first rotating the image so that the desired orientation now lies horizontally or vertically, and then apply the appropriate window. For example, a significant 30° blur is realized by a rotation of 330° (-30°), a blur window of, say, height 3 and width 9, followed by a rotation of 30°.

Due to the manner of implementation, there are no run-time differences between filtering with large or small window sizes. However, filtering with odd window dimensions run slightly faster than using even window dimensions. This is due to the fact that odd windows are symmetric, positioning their centers directly on a whole pixel. Windows with even dimensions, on the other hand, must position their centers on a fractional pixel thereby requiring additional computation.

- **Blur with mask:** Blur the input image, with the extent of blurring specified by a second image serving as a mask. The luminance values of the mask image are used to interpolate between the input image and a maximal blurred version of it. Higher pixel values in the mask cause more blurring for the corresponding pixel in the output image. The following parameters are requested:

[†] This is due to the fact that the Fourier Transform of a Gaussian is a Gaussian itself.

Mask quad: (0,3) <nextqd>
Window width (x): (1.,256.) <3.>
Window height (y): (1.,256.) <3.>
Order of blur fall-off: (.01,10.) <1.>

The user first specifies the quad containing the mask image. The mask should be monochromatic. In the event that it is a color image, the luminance component is used. If necessary, the mask is padded with 0 (black) in order to take on the same dimensions and position as the input image. The largest mask values yield the greatest blur which is specified by the given window size of the box filter. All other mask values index between the original input image, and its maximal blurred copy. Note that the dynamic range of the masked image is first normalized.

Ideally, a stack of successively blurred copies of the input image would be precomputed. Then, the mask value would index into this stack and read the output value for its corresponding pixel. Lower values would index into the low end of the stack that is closer to the original (unblurred) image. Higher values would index into the higher, more blurred, portion of the stack. However, to approximate this procedure, we simply use the mask value to interpolate between the corresponding pixels in the blurred and unblurred copies of the input image. Implicit here is the critical assumption that the intensity variance is monotonically decreasing as the image is increasingly blurred. This approximation is adequate in most cases.

The user is given partial control in the method of interpolation by means of specifying the order of fall-off for the blur function. Let *min* and *max* be the minimum and maximum values in the mask image, respectively. They are used to normalize the mask value, *mask*, that is used to interpolate between *original* and *blur*, the unblurred and blurred pixels, respectively. The output pixel is *new*.

$$f = \frac{mask - min}{max - min}$$

$$new = original + (blur - original) \times f^{order}$$

For each pixel in the input image, the corresponding pixel in the mask and blurred image are used to compute *f*, the blurring index. The higher its value, the more blurred will be the output pixel. Since direct use of *f* may not provide the desired visual effect (a subjective matter), the user may skew the effect of the blur index by specifying *order*, the rate of fall-off. Since *f* is always in the range between 0 and 1, higher values of *order* will cause a rapid fall-off of added blur, yielding mildly blurred images. Lower values of *order* will exaggerate the blur index and generate more greatly blurred images.

- **Edge preserving blur:** Blur the input image, with the extent of blur determined by the spatial frequency content at each pixel. This makes use of the **Blur with mask** option, with the mask specified by the output of a highpass filter. The highpass image is simply the difference between the original and a blurred copy of it. Such an operation yields high values where the edge content is strong (high frequency details), and low values where there is little gray level activity.

Since decreased blurring is to correspond to high values in the mask image (stronger edges), a negative of the mask is computed and used in the calculations.

Window width (x): (1.,256.) <3.>
Window height (y): (1.,256.) <3.>
Order of blur fall-off: (.01,10.) <.5>

The window size indicates the width of the highpass filter. Larger window sizes yield stronger highpass filter output, consequently increasing the dynamic range (resolution) of the blur indices. This, in turn, can lead to smoother images. The user can control the effect of the computed blur indices by specifying the order of blur fall-off.

• **Image sharpening:** Enhances the visual details in the input image. Also known as unsharp masking enhancement, this operation produces an output image in which high frequency details are improved. It does so by passing the image through a highpass filter, and adding that output back onto itself. Since the highpass filter has high values in areas of high visual detail, adding its output onto the original image puts emphasis on the high frequency range of the image. The variables in this process are the extent of the low pass filtering to be applied upon the original image before subtraction, and the amount of brightness scaling to be used on the difference image before adding it to the original.

Window width (x): (1.,256.) <3.>
Window height (y): (1.,256.) <3.>
Emphasis factor: (1.,10.) <1.>

Smaller windows yield more granular images. This is due to the poorer noise discrimination that accompanies filters of small extent. The granularity disappears with larger windows since the noise power diminishes relative to the admitted signal. This translates into an increased signal-to-noise ratio.

When applied over an already blurred version of an image, this operation can be used to boost a frequency band of that image. This generates interesting visual results.

• **Median filter:** Apply a median filter upon the input image. A median filter selects the median value of the neighborhood as the output pixel. It is a nonlinear filter that attempts to remove image noise (i.e. speckles) without smearing the edges. Although it is a neighborhood operation, it does not fall within the category of spatial convolution since its result is not based on a weighted average of the neighborhood pixels. An extension of the median filter is to take the average of the L nearest pixels to the median. This process is known as an L-smooth filter. It yields better smoothing properties than straightforward median filtering alone.

Number of neighbors about median: (0,4) <0>

The default is 0 for a median filter without the additional smoothing option.

- **Thresholded Laplacian:** Restore a bilevel image that has undergone smoothing degradation. This algorithm was originally designed to recover black-and-white images of text that had been corrupted by the digitizer point spread error function. It is based on the observation that the convolution of a step function (bilevel image), $h(t)$, with a bell shaped function (such as the point spread function of digitizers) is convex where $h(t)$ is high (white) and concave where $h(t)$ is low (black).

It operates by first approximating the Laplacian, L , of the image at each position†. If the absolute value of L is below a user-specified threshold, the input image pixel is copied to the output image. Otherwise, if L is positive, $h(t)$ is high and the output pixel is made white. If L is negative, $h(t)$ is low and the output pixel is made black. Details of the algorithm can be found in [Pavlidis and Wolberg 86].

Although this operation is meaningful only over images that were originally bilevel, it yields interesting visual results over continuous tone images as well. Selecting a low threshold value causes increased noise sensitivity. This generates images that have a highly granular appearance — the granularity due to highlighting the noise present in the image. Interesting results can also be produced by applying this operation over images that have undergone severe blurring. The white streaks can be extracted (by thresholding) and reapplied onto the original image (by image summation). In general, an increase in threshold value corresponds to a decrease in noise sensitivity, yielding a lower density of edge elements.

- **Laplace image:** Display the edge contribution of each pixel. This operation is actually the application of a Laplacian operator on the input image. All negative values computed on *unsigned char* images are clipped at 0. Like the highpass filter, the output values are proportional to the strength of the edge passing through that point. Whereas, the highpass filter uses an adjustable window size, this is strictly a local operation using a 3×3 window. The user must specify a brightness factor to scale the computed values.

- **Relief map effect:** Compute the Laplacian of the input image, scale the results, and display it with an added offset. This is similar to the previous operation except that negative values are not clipped and the dynamic range is compressed.

- **Exit submenu:** Return to the main menu.

† The Laplacian in a 3×3 window is taken to be the maximum difference of slopes of two col-linear diagonals passing through the center pixel, computed along the horizontal, vertical, and two diagonal orientations.

6.4. Geometric Ops Submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Scale:** Scale the input image to arbitrary user-specified dimensions. The parameters collected for this operation are identical to those specified in the **Write image to file** option in the main menu. The user is referred to the description of that option for an example. In addition, the user is requested to input the *degree*, a parameter which determines the method of scaling. Current allowable values for *degree* are 0, 1, and 3. When enlarging the image, they invoke pixel replication, linear interpolation, and cubic interpolation, respectively.

Pixel replication, also known as the nearest neighbor technique, simply duplicates pixels. This is identical to hardware zooms available on frame buffers and yields a blocky appearance. Linear interpolation and cubic interpolation apply 1st and 3rd degree splines to the image points before resampling. This yields smoother results. Typically, linear interpolation is adequate unless excessive enlargements are specified in which case artifacts due to first derivative discontinuities appear. In these cases, the more expensive cubic interpolation method is in order.

When shrinking, the scaling function automatically blurs the image to prevent aliasing problems. This antialiasing is bypassed when *degree* is 0, thereby allowing faster, but poorer, renditions of minified images.

- **Rotate:** Rotate the input image. The user supplies θ , the angle of rotation, and (x,y,z) , a point in 3-space. The rotation algorithm produces an output image geometrically rotated counter-clockwise, about the rotation axis, through the angle θ . The rotation axis is taken to go through the origin and (x,y,z) . A right-hand coordinate system is used in which x increases from left to right, y increases from bottom to top, and positive z comes out of the screen towards the viewer. There is no foreshortening due to perspective — it is an affine transformation. The implementation is based entirely on skew transformations as documented in [Tanaka 1986] and [Paeth 1986].
- **Translate:** Shift the input image in the horizontal and vertical directions. The message prompt indicates the range of values that may be supplied in the x and y directions to avoid clipping. Beyond that range, the image will be clipped. Note that the input values may be floating point. This allows for shifts involving fractional pixel sizes.
- **Warp:** Warp a source image to occupy the space defined by a mask image. Unlike other rubber sheet transformations, the user is not required to supply a deformation mapping function. The user must only enter, via a mouse, the locations of corresponding points on the source and destination (mask) images. This operation can be used to create interesting visual effects.

```
Image 1:  (0,3) <activeqd>
Read value? (0=no; 1=yes)  (0,1) <1>
```

Image 2: (0,3) <activeqd>

Read value? (0=no; 1=yes) (0,1) <1>

The first message requests the user for the quad containing the source image. The user must then specify correspondence points on the contour of the source image. The mouse is activated and the user must position the mouse on the desired point(s), responding to the second message with a 1 as long as more points are desired. Entering a 0 will break this loop. The procedure is repeated again for the destination image. Notice that after each point is entered, a number is displayed at the position. The numbered points of the source image is made to correspond to those in the destination image. An error will occur if the number of points marked on each image are not equal.

- **Exit submenu:** Return to the main menu.

6.5. Matte Ops Submenu

The image compositing operations described in this section are taken from [Porter 84]. These operations facilitate the aggregation of foreground and background elements to generate more sophisticated images. Underlying this process is the ability to control opacity information for each element to retain proper balance and soft edges. Consequently, an additional component is added to each pixel to determine opacity of that point. This number specifies the mixing factor required to control the linear interpolation of foreground and background colors. These opacity components comprise a *matte* or *alpha* channel. For anti-aliasing purposes, their range is of comparable resolution to the color channels — the 0 to 255 range. A value of 255 is taken to mean total opacity, 0 means total transparency.

Some of the supported compositing operations are illustrated below in Fig. 8. The examples make use of images *A* and *B*. Note that image *A* is depicted with the coarse texture, and image *B* is shown with the fine texture. The F_A and F_B entries represent the multiplicative factors applied to *A* and *B*, respectively.

IMPORTANT: All images involved in these compositing operations are assumed to have already been premultiplied with the *alpha* channel. If this is not the case, then the user must apply the *Matte Cut* option first before proceeding.












operation	diagram	F_A	F_B
<i>A</i>		1	0
<i>B</i>		0	1
<i>A over B</i>		1	$1 - \alpha_A$
<i>B over A</i>		$1 - \alpha_B$	1
<i>A in B</i>		α_B	0
<i>B in A</i>		0	α_A
<i>A out B</i>		$1 - \alpha_B$	0
<i>B out A</i>		0	$1 - \alpha_A$
<i>A atop B</i>		α_B	$1 - \alpha_A$
<i>B atop A</i>		$1 - \alpha_B$	α_A
<i>A xor B</i>		$1 - \alpha_B$	$1 - \alpha_A$

Figure 8: Image compositing operations

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Cut out subimage:** Cut out a subimage from the input image.

Active quad: (0,3) <activeqd>
Enter 0 (quit) or 1 (abort): (0,1) <0>

The first message requests the user for the quad containing the input image. A crosshair then appears and the mouse is activated, allowing the user to draw an outline around the area to be extracted. Drawing is initiated by clicking the mouse button. The mouse is then tracked, leaving a trail of blue pixels in its path. Note that image memory is not overwritten by this process. Successive clicking on the mouse button repeatedly activates and deactivates drawing. This allows the user to move to a new position without drawing a line. Note that the drawn contour *must be closed*. When the contour is completed, the user responds to the second message with a 0. The contour is then flooded with a blue fill, the input image is erased, and the extracted subimage is displayed. The user may decide to abort by entering a 1 to the second message. In that case, the input image is left intact.

- **A cuts B when A>0:** Display image B where the corresponding pixels in image A are non-zero. This is useful for having a cutting template defined directly by the non-zero intensities in an image. Since A effectively defines a bilevel matte, the edges are likely to appear jagged.
- **A over B when A>0:** Overlay image A upon image B for non-zero pixels in A. This allows images, without matte information, to be used directly in overlay operations by considering their intensities alone. Note that this is equivalent to a bilevel matte and thereby yields hard (jagged) edges.
- **Matte cut:** Multiply the color components of the input image by the corresponding value in the *alpha* channel. Recall that the compositing operations require the input image to be pre-multiplied with the *alpha* channel. Therefore, this function should be used to make images suitable for subsequent compositing operators.
- **A over B (A)+(B)(1-aA):** Foreground A is placed in front of background B. The expression in the option name defines the actual computation. It says that A is added to the product of B and (1-aA), where aA is the *alpha* channel of A. In this notation, these values are taken to lie between 0 and 1. Therefore, the amount in which B participates in each pixel is determined by the opacity of A's pixel. Note that because A has been pre-multiplied with its *alpha* channel, it can be added directly. If it had not been pre-multiplied, then it would be possible to have foreground elements with *alpha*=0, which when added to $B \times (1-aA)$ would yield values outside the valid range. Unlike the previous **A over B when A>0** option, this function uses a matte and yields soft edges.

A: (0,3) <activeqd>

B: (0,3) <activeqd>

Alpha mtd: (0=add; 1-max) (0,1) <0>

The last prompt refers to the manner in which to compute the *alpha* component of the output image. Depending on the model of the pixel coverage, the output *alpha* value can either be the sum or maximum of the input *alpha* values. The sum is appropriate when it is assumed that the pixels at the border of *A* and *B* do not have overlapping contributions. Otherwise, the maximum is in order. If the addition mode is wrongly chosen, then successive overlays will generate an output image which has high opacity, possibly yielding hard edges. On the other hand, this method is cheaper to compute and is not normally subjected to the unfortunate scenario above.

- **A in B (A)(aB):** Take only that part of *A* which lies inside *B*. As indicated in the expression, the pixels are weighted by *B*'s *alpha* channel.
- **A out B (A)(1-aB):** Take only that part of *A* that lies outside *B*.
- **A atop B (A)(aB)+(B)(1-aA):** Take the union of *A* in *B* and *B* out *A*. Thus, *A* atop *B* includes *A* where it is on top of *B*, and *B* otherwise.
- **A xor B (A)(1-aB)+(B)(1-aA):** Take the union of *A* out *B* and *B* out *A*. This includes those areas which are mutually exclusive among *A* and *B*.
- **Hicon A (aA):** This refers to the *alpha* channel of *A*.
- **Darken A (f*nonalphaA):** This function requests the user for a darkening factor between 0 and 1. It applies that factor to the color components of *A*. Note that it is not applied to the *alpha* channel, and so *A* is simply darkened but not faded.
- **Opaque A (f*aA):** This function requests the user for an opacity factor. It applies that factor to the *alpha* channel of *A*. Note that it is not applied to the color components, and so only the pixel coverage information changes.
- **Dissolve A (f*A):** Requests the user for a dissolve factor between 0 and 1. It applies that factor to the color and *alpha* components. This results is a darkening and fading of the image. The operation is useful in cross-dissolves, or linear interpolation, between images *A* and *B*. This takes the form $\text{dissolve}(A,\alpha) + \text{dissolve}(B,1-\alpha)$.
- **Read pixel value:** Print the color value of a screen pixel. The user must position the mouse on the desired pixel and respond to the following message.

Read value? (0=no; 1=yes) (0,1) <1>

The user may repeatedly read color values from the screen by responding to the message with a 1. The loop is broken by entering a 0. The color values are given in the *RGB*, *YIQ*, and *HVS* color spaces.

- **Read pixel location:** Print the (x,y) position of a screen pixel. The user must position the mouse on the desired pixel and respond to the following message.

Read value? (0=no; 1=yes) (0,1) <1>

The user may repeatedly read (x,y) positions from the screen by responding to the message with a 1. The loop is broken by entering a 0.

- **Exit submenu:** Return to the main menu.

6.6. Image Conversion

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Convert channel type:** Convert the data type of the input image channels. The available data types are: *unsigned char*, *short*, *int*, *long*, *float*, and *double*. These types are listed with numbers and the user selects the desired number for each of the input image channels.
- **Convert image type:** Convert the input image to a new image type. The valid image types are listed with corresponding numbers. These types are identical to those listed in section 4.

0) MAT

1) BW

2) BWA

3) RGB

4) RGBA

5) VHS

6) YIQ

Image type: (0,6) *<imgtype>*

Note that the default value denotes the current *imgtype* of the input image.

- **Extract channels:** Extract user-specified channels from the input image. Copy them into *nextqd*.

Extract Channels: (hex) *<ffff>*

The channels to be extracted are specified by a hex number. Each channel corresponds to successive bits in the hex number, with the first channel given by the least significant bit. For example, extracting the first and third channels is indicated by bits 00...0101, or hex number 0005. This operation is useful for viewing individual channels.

- **Append channels:** Append channels onto the end of *nextqd*.

Append channels (0) or quad (1) to target? (0,1) *<1>* 0

Target quad: (0,3) *<activeqd>*

Enter quad -1 to exit loop

Source quad: (-1,3) *<activeqd>*

Channel: (0,*maxch*-1) *<0>*

The user first selects between two modes: appending channels, or appending an entire quad. In the above example, a 0 response indicates that individual channels will be appended onto the

specified target quad. These channels are denoted by their source quads and channel numbers. The user thus enters a cycle requesting this information. Note that the source quad refers to that quad containing the desired channel, and the channel number specifies the actual channel in the quad. The valid channel range is from 0 to *maxch*-1, where *maxch* is the number of channels present in the quad. The loop is broken by entering a -1 to the prompt requesting the source quad.

Had the user wanted to append a quad to the input image, a 1 would be entered to the initial prompt. The user would then specify the source quad, containing the appending channels, and the target quad onto which the channels will be appended. Typical applications for this operation is to convert a single-channel black-and-white image into a three-channel color image, or adding an *alpha* channel to an image for compositing operations.

- **Copy channels:** Copy quad channels from input image(s) into *nextqd*.

```
Target quad:      (0,3) <activeqd>
Enter quad -1 to exit loop
Source quad:      (-1,3) <activeqd>
Source channel:   (0,maxch) <0>
Target channel:   (0,maxch) <0>
```

The target quad refers to that quad into which the channels will be copied. These channels are denoted by their source quads and channel numbers. The user thus enters a cycle requesting this information. Note that the source quad refers to that quad containing the desired channel, and the channel number specifies the actual channel in the quad. The valid channel range is from 0 to *maxch*-1, where *maxch* is the number of channels present in the quad. The loop is broken by entering a -1 to the prompt requesting the source quad. Note that the channels copied into the target quad retain the same data types they had in their source quads. This operation is used to overwrite channels.

- **Color shift:** Apply a color shift operation on the input image.

```
VHS op; (0=add; 1=mult)    (0,1) <0>  0
Add to V:  (0,255)        <0>
Add to H:  (0,359)        <0>
Add to S:  (0,255)        <0>
```

The color shift is realized by either adding or multiplying constants to the (Value, Hue, Saturation) color components of the input image. In the above example, the user selected to add constants to the VHS components. This operation is particularly useful for shifting the hue between 0 and 360°. The VHS hexcone is shown below in Fig. 9. Notice, for instance, that by adding a constant of 60°, we can shift the colors such that red becomes yellow, yellow becomes green, etc.

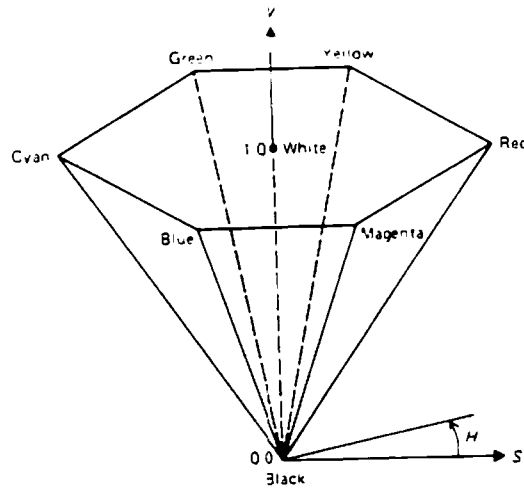


Figure 9: Single hexcone VHS color model.

- **Set global args:** Initialize several global variables useful for determining processing modes of operation. These variables include *ABSVAL*, *ROTCLIP*, and *READ_FIT*. If *ABSVAL*=1, the absolute value is taken for low-precision (*unsigned char*) subtracted images. Otherwise, they are clipped. If *ROTCLIP*=1, the rotated images are clipped to fit in the quad. Otherwise, they are initially scaled down so that the rotated result will fit without any clipping required.

READ_FIT determines the sampling technique used to fit images that are read into quads from files. A value of 0 performs uniform scaling with no anti-aliasing filtering. This point samples the image at appropriate positions so that the image takes on its specified dimensions exactly. A value of 1 performs integral point sampling. Again, there is no anti-aliasing filtering here. In addition, the dimensions of the resulting image are not guaranteed to match the requested dimensions since the sampling is constrained to be taken at regular integral intervals. A value of 3 forces the whole image to be read into the quad. If the quad is too small, it is replaced by a sufficiently larger one. A value of 4 performs uniform scaling *with* anti-aliasing filtering. Although this method is the most expensive of those listed above, it yields the best rendition of minified images. Note that the *READ_FIT* values are irrelevant if the image already fits into a quad without any down-sampling.

The values assigned to these global variables stay intact until another call to *Set args* resets them.

- **Exit submenu:** Return to the main menu.

6.7. Image Transforms Submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Fourier transform (1D):** Take the Fourier transform of a specified cross-section of the input image. The input, as well as the computed amplitude response, are plotted. The user first enters the orientation of the image cross-section, followed by its position.
- **Fourier transform (2D):** Take the Fourier transform of the input image, and display the power spectrum. The spectrum is displayed with the zero frequency located in the center. This tool is insightful into the role of frequency analysis in image filtering.
- **Thinning:** Generate the skeleton of the input image, assumed to be black and white (foreground and background). The thinning process successively removes the outer perimeter of foreground pixels in a manner similar to peeling an onion. If, however, the deletion of a point is found to change the connectivity of the local pixel neighborhood, then that pixel is considered to lie on the skeleton. The resulting skeleton permits us to more readily determine its essential topology and measure its components. Skeletons are guaranteed to be fully connected and retain the equivalent topology of the input image. This process is most meaningful when applied upon shapes with lineal properties. Implementation details and examples can be found in [Wolberg 1985].

The algorithm assumes that the input image has already been thresholded. The user must respond to the following message sequence:

Thinning method: (0=normal; 1=enhanced) (0,1) <0>
Foreground: (0=blk; 1=white) (0,1) <1>

There are two thinning modes: *normal* and *enhanced*. In the *normal* mode, skeletal pixels take on gray values proportional to their distance from the contour, with the furthest distance displayed as white. All deleted pixels are displayed as black.

The *enhanced* mode extends the thinning process by further classifying skeletal pixels as either *SKL* or *BRIDGE*. *SKL* pixels, like the skeletal pixels of the *normal* mode, have the property that they are necessary to maintain the connectivity of their local neighborhoods. *BRIDGE* pixels identify those skeletal pixels which exist only to maintain the connectivity of the skeleton itself. This additional label is useful for the rubber sheet image transformation (warping) discussed later. The *SKL* pixels are all displayed at the same bright gray level. *BRIDGE* pixels are displayed at a lower gray level.

- **Exit submenu:** Return to the main menu.

6.8. Image Intensity Plot Submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Plot:** Plot a 3D relief of an input image channel.

Angle between x-axis and horizontal: (-89,89) <30>
Channel: (0,maxch) <0>
Sample: (0,width) <8>
Blur? (0=no; 1=yes) (0,1) <1>

The orientation of the displayed axes is given in Figure 10. Angle θ is supplied by the user. If the image has more than one channel, the user must indicate the channel to be plotted. In addition, the user must specify the desired sampling rate for generating the resulting grid. Selecting a high sampling rate yields a more accurate plot. However, the drawback is that the points are plotted more densely, making it difficult to observe. The user has the option to blur the image data prior to subsampling it. This is the appropriate filter useful for preventing aliasing. The relief surface is plotted with all hidden lines removed.

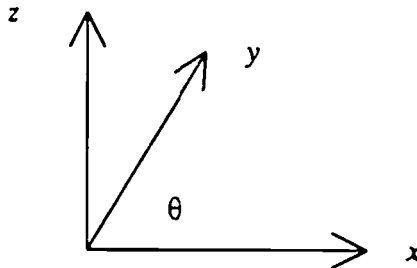


Figure 10: Orientation of displayed axes

- **Set gray for y-traces:** Set the gray level for lines drawn in the y-direction. This is useful in improving the display of the plotted surface.
- **Set z-scale:** Scale the relief map in the z-direction. This is useful to normalize and amplify features in the plot.
- **Exit submenu:** Return to the main menu.

7. COLORIZATION

This section deals with the operations available to apply color onto black-and-white images. The implemented approach differs from that of commercial paint-box systems. The process of assigning color information using paint-box systems is a tedious procedure of actually passing an electronic paintbrush over the image and an equally painful process of arriving at the desired color.

There were several major considerations in the design of the colorization system described below.

- 1) Minimize the effort in accurately extracting the regions to be colored by using histogram analysis.
- 2) Ease in assigning color to the extracted regions.
- 3) A facility for the quick editing of color in already colored regions.
- 4) An intuitive method of generating the color database.
- 5) A fast mode in which to fine tune the color prior to actually writing the information to the pixels.

7.1. Overview

The application of color consists of a cycle of three events: roughly outlining the desired region, applying a threshold operation to extract the region of interest, and specifying the color of the extracted region. Given an image of a face, for example, we might begin by colorizing the lips. The user would begin by using the mouse to draw a rough outline around the lips. Examining the histogram for that subimage, a range may be specified in which to threshold the subimage into three intensities: black, constant gray, and white. The objective in selecting the threshold is to capture all the pixels of interest between the two supplied threshold levels, thereby displayed with a constant gray in the thresholded image. In this case, only pixels lying on the lips should appear as gray. The user may then specify that all pixels at that particular gray level be assigned a certain color.

In cases where the dynamic range of the extracted subimage is too wide to discriminate between the foreground and background a simple flood fill must be used. For instance, care must be taken to cut out the hair region more carefully when it cannot be accurately discriminated from the neighboring facial pixels of similar intensities. Options also exist for adding color within a region without overwriting already colored pixels.

7.2. Color Information

There are various color spaces in which to define the applied color. We have chosen to use the YIQ color space. The Y channel is the luminance information, already provided by the original black-and-white image. The user must therefore assign the I and Q values. Although it is possible to allow the user to create a color palette by trying combinations of I and Q , the more

straightforward approach consists of reading a digitized image or video frame containing the desired color. For example, suppose that a particular blonde hair is sought. The user may then find a picture or video containing that blonde color, digitize it, and read pixel values off of that region. Taking the average of the recorded values, the user has the proper I and Q values. This approach is often more reliable and faster than starting from scratch.

7.2.1. Color Data Structure

Each color in the system is specified internally with the data structure given below.

```
typedef struct {
    char colname[MXSTRLEN]; /* color name */
    unsigned char Yref;      /* reference Y */
    int I, Q;                /* I,Q define color */
    int ofst;                /* bias for Yref */
    unsigned char red[MXGRYNUM]; /* color maps */
    unsigned char green[MXGRYNUM];
    unsigned char blue[MXGRYNUM];
} colorS, *colorP;
```

The values for $MXSTRLEN$ and $MXGRYNUM$ are defined in the IMPROC *ip.h* header file as 80 and 256, respectively.

Each color must have a name, a unique (I, Q) pair, and additional information needed to create its color maps. The color maps are used to provide the RGB values needed to display a colored image on a monitor, or store it in a frame buffer. Therefore, assigning color c to pixel p , simply causes p to index into c 's color map.

Given the (I, Q) pair, together with the input Y , RGB color maps may be generated using the following relation.

$$R = Y + .9483I + .624Q$$

$$G = Y - .2760I - .640Q$$

$$B = Y - 1.110I + 1.73Q$$

Recall, however, that it is possible to derive the I and Q values by reading them off of a digitized color image. In doing so, that (I, Q) pair is actually coupled to a particular Y , call it Y_{ref} , to yield a reference color. Blindly applying the above transformation to the full range of luminance values may yield artifacts, such as colored shadows and highlights. This problem is due to the saturation of one or more RGB color channels that, in turn, is due to the generation of color values outside the allowable range. This problem is corrected in this system by performing color interpolation between the reference color and the black and white extremes of the spectrum. Therefore, pixels below Y_{ref} are mapped onto a color ramp between black and Y_{ref} . In practice,

(Y_{ref}, I, Q) is first converted to $(\hat{R}, \hat{G}, \hat{B})$. Pixels above Y_{ref} are mapped onto a color ramp between $(\hat{R}, \hat{G}, \hat{B})$ and white. Those pixels below Y_{ref} are mapped between $(\hat{R}, \hat{G}, \hat{B})$ and black. This is illustrated for the R channel in Figure 11. Higher order interpolation may be used in place of linear interpolation for different effects. For instance, colors to be applied over specular surfaces may use higher order interpolation than colors to be applied over matte surfaces.

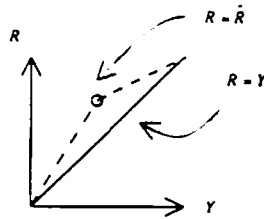


Figure 11: Color interpolation

The *offst* member in the color data structure is used in cases where the reference color is to appear on a particular Y value. If Y is not equal to Y_{ref} a bias must be added to Y in order to achieve the desired effect. The bias value, *offst*, must be added to the input image luminance values prior to computing the output color. Nonzero *offst* values alter the luminance of the output image. It may be used to advantage to correct for over/underexposure in the original image, and thereby generate superior color images. However, if the color is suppressed (using the monitor's color knob), blotches would appear in the resulting black-and-white image indicating the presence of various nonzero *offst* values applied to the original intensities. Therefore, *offst must be zero* when used to colorize images in which the viewer has the means of suppressing the color information. This includes the colorization of movies that are shown on television.

It is noteworthy here to draw a comparison to an alternate coloring scheme devised by Colorization Inc. of Toronto, a major colorization company. They use the *HVS* color space, where H is hue, V is value (identical to Y in YIQ), and S is saturation. In their system, the original black-and-white image serves as the V and S channels. The user must only provide H , the hue. While initializing H is less work than having to initialize both I and Q , their system is incapable of refining the color maps due to their restricted input. The system implemented here allows for that extra degree of freedom in selecting arbitrary interpolation methods between the reference color and the spectrum extremes.

7.2.2. Color Database

Each color is identified by its *group* name and *color* name. The *color* name is simply the *colorname* member of the color data structure. The *group* name is used to add meaningfulness to the large collection of colors that may accrue. An example of typical (*group, name*) pairs are (hair, brown), (hair, blond1), (hair, blond2), (eyes, blue), etc. This information is defined to the system by the user adding it to the session color list. It may be saved and used again in another session through system-maintained color files.

At the start of every colorization session, the user is prompted for a color file to initialize the color database for that session. Color files should be named with a .colors extension. Each

line in the file specifies a color and has the format shown below.

group name, color name, Yref, I, Q, offset

Each color list must contain one color indicating an uninitialized state which can be used to clear (uncolor) *C* buffer entries:

UNINIT, UNINIT, 0, 0, 0, 0

At the end of each colorization session, the color database is written back into the same file.

7.2.3. The *C* Buffer

The color information is overlaid upon the luminance channel provided by the black-and-white input image. Consequently, a *C* buffer is defined consisting of pointers to the color data structures. The combination of the *C* buffer with the underlying luminance values of the original image yields a colored image. The color is derived from the luminance indexing into the color maps of the corresponding color data structure.

In order to conserve memory and reduce execution time, the colorization is performed at half the resolution of the original image. This is adequate since the human visual system is not acutely sensitive to color discrimination in small areas. Luminance information provides the major cue in these instances. Therefore, the active quad providing the black-and-white input image must be point sampled (not uniformly sampled) by a factor of 2. In this instance, point sampling is necessary since the eventual enlargement of the *C* buffer must be done by replication since it is meaningless to speak of interpolating color pointers.

As stated earlier, the area of interest must first be extracted by drawing a rough outline around the desired region and then performing thresholding to display all desired pixels at a constant gray value. The gray pixels in the thresholded image serve as a mask indicating that the overlaying positions in the *C* buffer must be assigned the specified color (actually a pointer to the color data structure). This organization also allows easy editing of color by simply substituting one color (pointer) with another.

The *C* buffer may be saved into a file for use in a subsequent session. The only information actually saved are the (I, Q) pairs for each pixel. This data is expanded into a color data structure when read back into the system. This is done by matching each input (I, Q) pair with that of each color in the database. For this reason, no two colors in the database are allowed to share the same (I, Q) pair. Conversely, two colors may share the same *colorname* and be considered different colors as long as they do not both belong to the same *group*.

7.3. Colorization Submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Display Images:** Select the sub-submenu containing operations in which to display and highlight (un)colored images.
- **Color Palette:** Select the sub-submenu operating on the color database.
- **Cut out subimage:** See identical entry in Matte Ops submenu.
- **Color entire region:** Apply a color to the entire subimage contained in the active quad. Since the subimage may already be partially colored, the user must indicate whether only the uncolored regions are to be colored, or whether all pixels are to be colored, irregardless if they are already initialized.

Overwrite? (0=no; 1=yes) (0,1) <1>
Group name: <>
Color name: <>

Responding to the first message with a 1 causes all subimage pixels to be colored. A 0 response causes only uncolored pixels to be colored. The applied color is specified by the *group* and *color* names. If the user enters a name which is not currently in the color database, the option aborts and no color is applied. In general, this option is used to color regions which cannot be extracted more precisely (and easily) by means of thresholding. This includes regions containing a wide range of intensities.

- **Color gray levels:** Apply a color to those pixels displayed at specified gray levels. This option follows a thresholding operation in which pixels lying in the region of interest are now displayed at a constant intensity.

Active quad: (0,3) <activeqd>
Inclusive? (0=no; 1=yes) (0,1) <1>
Read value? (0=no; 1=yes) (0,1) <1>
Overwrite? (0=no; 1=yes) (0,1) <1>
Group name: <>
Color name: <>

The first message requests the user for the quad containing the thresholded image. In the second message, responding with a 1 (inclusive) indicates that the set of intensities to be entered should be regarded as those pixels which are to be colored. A response of 0 indicates that all other intensities are to be colored instead. The user must now specify the intensities. This is done by positioning the mouse upon any pixel of the appropriate intensity and responding to the "read

value?" prompt with a 1. That message prompt is continually re-issued as long as a 1 is entered. The loop is broken by entering a 0. As in the **color entire region** menu option, the user must specify whether already colored pixels may be overwritten, as well as the *group* and *color* names of the applied color.

- **Fast region coloring:** Colors a specified region interactively using a mouse to vary the color parameters.

Point to region and enter: (0=clr; 1=nonclr; 2=abort) (0,2) <0>
Color map (0=linear; 1=tint) (0,1) <0>
Fix white at *MXGRAY*? (0=no; 1=yes) (0,1) <0>
YIQ or VHS (0=YIQ; 1=VHS): (0,1) <1>

The mouse is activated and the user may position it over a desired pixel. Entering a 0 to the above message will read the *C* buffer entry for that pixel and perform fast recoloring on those pixels with identical entries. See the **Regenerate color maps** option in the Color Palette submenu for an explanation of the next two messages. See the **Select and modify** option in the Color Palette submenu for a description of the selection of a color space and the layout of the interactive parameter settings. Like that option, movement of the mouse causes color parameters to change and an immediate update of color over the specified pixels. This is due to the continuous update of the frame buffer LUTs and no update to the *C* buffer. It is recommended as a tool in which to quickly run through a range of colors for specified regions. Once the user is satisfied with a given color, the update session may be terminated by moving the mouse to the *exit* box and confirming the desire to exit. The color values are printed on the terminal. From there, the user may add them to the color database and apply them to the region.

- **Thresholding:** See identical entry in the Point Ops submenu.
- **Histogram:** See identical entry in the Point Ops submenu.
- **Read pixel value:** Print the color value of a screen pixel. The user must position the mouse on the desired pixel and respond to the following message.

Read value? (0=no; 1=yes) (0,1) <1>

The user may repeatedly read color values from the screen by responding to the message with a 1. The loop is broken by entering a 0. The color values are given in the *RGB*, *YIQ*, and *HVS* color spaces.

- **Read color masks:** Read color information from a file. This serves to initialize *C* with pointers associated to the newly read color values.
- **Save color masks:** Save color information into a user-specified file.
- **Save color image:** Combine the color information with the original black-and-white image. Save the resulting color image in a user-specified file. The *C* buffer is scaled to take on the

same dimensions as the original black-and-white image. Scaling is done by means of replication. An error message will appear if the dimensions of the original image are not integer multiples of the *C* buffer.

- **Change picture:** Assign image of specified quad as the new image to be colorized.
- **Exit submenu:** Return to the main menu.

7.4. Display Images Sub-submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **Display picture:** Display the black-and-white image being colorized.
- **Display mask:** Display the image from which the current subimage is derived.
- **Display colored image:** Display the colorized image.
- **Highlight (un)colored regions:** All regions that have been assigned a specified color are displayed in white. The remaining regions are displayed with their original intensities. This is useful for clearly verifying the status of the colorized image.

Point to region and enter: (0=clr; 1=nonclr; 2=abort) (0,2) <0>

The mouse is activated and the user may position it over a desired pixel. Entering a 0 to the above message will read the *C* buffer entry for that pixel and display in white all pixels with identical entries. Also, the position, quad number, and color data is printed on the terminal. If all uncolored pixels are to be highlighted, the user must enter 1 to the above message. The mouse will be ignored and all uninitialized entries are displayed in white. Finally, a 2 entry aborts the operation.

- **Exit sub-submenu:** Return to the main menu.

7.5. Color Palette Sub-submenu

- **Menu display:** List menu options and their option numbers.
- **Set next quad:** Specify the quad in which to display the next image.
- **Set active quad:** Specify quad containing the input image.
- **List colors:** Lists all groups, colors, and color data defined in the session's color database.
- **Add color:** Add color to the database.

Group name: <>
Color name:<test>
Y: (0,255) <128>

I: (-.6Y,.6Y) <0>
Q: (-.52Q,.52Q) <0>
Yref: (0,255) <Y>

The user enters the *group* name, *color* name, and the color data.

- **Delete color:** Delete color from the database.

Group name: <>
Delete entire group? (0=no; 1=yes) <0>
Color name:<>

The user first indicates the *group* name containing the color to be deleted. If the user chooses to delete all colors in the group, a 1 is entered in response to the second message. Otherwise, a 0 is entered and the third prompt appears requesting the user for the *color* name.

- **Modify color:** Modify a color in the database.
- **Display all colors:** Displays the colors of each entry in the database in small squares on the monitor.
- **Display group:** Display the colors of a specified group in small squares on the monitor.
- **Select and modify:** Point to a color on the monitor and interactively edit its values, seeing the resulting color on the monitor.

Read value? (0=no; 1=yes) (0,1) <1>
YIQ or VHS (0=YIQ; 1=VHS): (0,1) <1>

The first message requests the user to position the mouse above a desired pixel. Entering a 1 will then cause that pixel's color to be read (in *RGB*). A 0 entry will abort the operation. The second message requests the user for the color space in which to edit. Five rectangular boxes are then displayed in the lower half of a quad. They represent the ranges for the 3 color values (*YIQ* or *VHS*), the *offset* entry, and an *exit* box. Position the mouse on the any of the first four boxes causes a horizontal crossbar to track the mouse. Moving the crossbar up (down) raises (lowers) the value of that parameter. Notice that the values are printed under the boxes. The upper half is used to display the color resulting from the current settings. As the parameter is changed, the color is continually updated. To exit the color editing session, position the mouse over the *exit* box and reply to the confirmation message. After exiting, the final color parameters are printed on the terminal in the *RGB*, *YIQ*, and *HVS* color spaces.

- **Init and modify:** This is identical to the **select and modify** option except that the user is expected to provide the color data numerically, rather than pointing to a colored pixel.
- **Regenerate color maps:** Regenerate all the color maps using the specified form of color interpolation.

Color map (0=linear; 1=tint) (0,1) <0>

Fix white at *MXGRAY*? (0=no; 1=yes) (0,1) <0>

The user first specifies whether a linear interpolation, or “tint” is to be used. The linear interpolation makes use of the *Yref* member of the color data structure and is described in section 7. The “tint” method ignores the *Yref* value and simply uses the straightforward conversion from *YIQ* to *RGB*. If linear interpolation is used, the second message appears. If white is not fixed at *MXGRAY*, the maximum valid intensity, then it is allowed to be offset by the *offst* parameter. In standard colorization, this has no effect since *offst* must be 0 (see section 7). Subsequent displaying of the colorized image make use of the updated color maps.

- **Exit sub-submenu:** Return to the main menu.

8. LIBRARY FUNCTIONS

All menu options invoke functions contained in the IMPROC library. These functions serve as useful building blocks for customized software. This section lists the available functions and their arguments. Since there is generally a correspondence between the function and a menu option, the user is directed to section 6 for a description of the function. As before, the list is partitioned into classes of image operations.

8.1. Data Structures

The most fundamental data structures in IMPROC, given below, specify all relevant information about quads and channels.

```
typedef struct qdformat {
    unsigned char *buf[MXCHANNEL]; /* ptrs to channels */
    int chtype[MXCHANNEL];         /* list of channel element types */
    int x, y;                       /* location of upper left corner */
    int height, width;             /* image dimensions */
    int xoffst, yoffst;            /* offsets from upper left corner of qd */
    int imgtype;                   /* label of image type */
    char status;                   /* FREE or NOTFREE (for memory management) */
} qdS, *qdP;

typedef struct chformat {
    unsigned char *buf;            /* ptr to channel memory */
    int links;                     /* number of links to quads */
    long chsize;                   /* size of channel (in bytes) */
    char status;                   /* FREE or NOTFREE (for memory management) */
} chS, *chP;
```

Each quad holds exactly one image. It can be considered as an image control block, containing information about the image and quad display data. The quad data includes the screen coordinate of the upper left corner of the screen partition, a flag indicating whether the quad is currently linked to an image (for memory allocation purposes), and x and y offsets for the contained image. These offsets are useful for performing image translation without actually modifying the image memory.

The image data includes a list of pointers to image channels, a list of codes denoting the corresponding channel data types, the image type, and the image dimensions. Up to *MXCHANNEL* channels may be attributed to an image. By default, *MXCHANNEL* is set to 16.

Any number of quads may be displayed during an IMPROC session. The user may specify this number by invoking IMPROC with a *-q* flag. By default, only four quads are displayed at

any time. Beyond the visible quads, an arbitrary number of additional off-screen quads may be requested by the user. Off-screen quads may be brought into view by simply copying them into any of the visible quads. Note that the x and y members in the quad structure are only meaningful for the visible quads.

The library functions described below are those routines which are invoked by the menu system in the interactive mode. They facilitate the non-interactive processing necessary in many image processing applications. All functions are fully general in the sense that they properly deal with all image types with channels of arbitrary data types. In addition, the output image can be designated to be the same as the input. All the necessary buffering is handled automatically. Finally, any operation which requires a channel to be replaced takes care of memory reallocation and its effects on any other quads which may be linked to it.

8.2. Quad Manipulation Functions

qdP getqd(height,width,ctype) int height, width, *ctype;

Return a pointer to a quad structure. Sufficient memory is allocated to accomodate an image with dimensions $height \times width$. Furthermore, the data types of the channels are specified in the *ctype* list. The values of the channel types are defined in the *ip.h* header file and are listed in section 4. Since these values are positive, a -1 is used to terminate the list. When the quad is no longer needed, the user should use **freeqd** to release the quad. In this manner, a subsequent request for an image of compatible size may avoid having to allocate more memory and thereby optimize system performance.

qdP getqd_qdp(q) qdP q;

Identical to **getqd** except that the argument values are provided by those corresponding to quad *q*. That is, the *height*, *width*, and *ctype* values are taken from *q*.

freeqd(q) qdP q;

Free quad *q*. This places *q* on a free-list so that it may be reused upon a subsequent call to **getqd**.

cpqd(q1,q2) qdP q1, q2;

Copy quad *q 1* to quad *q 2*. This includes the members of the quad structure as well as image memory.

cpqdinfo(q1,q2) qdP q1, q2;

Copy the members of quad structure *q 1* to *q 2*. It does not copy the image pointers or image memory. This is useful for initializing the size and type of an output image.

cpqdinfo3(q1,q2,q3) qdP q1, q2, q3;

Similar to **cpqdinfo** except that the information is determined from both *q 1* and *q 2*. That is, the dimensions and data types of the output image, *q 3*, are taken to be the maximum (most precise) of those offered by *q 1* and *q 2*, on a channel-by-channel basis. This is useful to assure that the output of a binary operation will yield an image that properly passes the precision and member information of the two input quads prior to the operation.

clearqd(q) qdP q;

This clears all the pixels in quad *q* to 0.

setaquad()

Print a message requesting the new active quad. It initializes the variable *activeqd* and draws a blue border around the image. **setaquad** is useful for software that will be installed into IMPROC and that must request a new active quad.

setnquad()

Request the user for the next quad in which to display an image. The request is re-issued if the active quad is specified (since it may not be overwritten).

incnextqd()

Initialize the *nextqd* variable to the next valid value. That is, *nextqd* is incremented and compared to *activeqd*. If they are found to be equal, it is incremented again (modulo the number of visible quads). This is useful following line plotting routines that write to a quad. Since only **dsply** (see later) increments *nextqd*, any other writing to a quad must use **incnextqd** to avoid overwriting that quad when displaying the next image.

qdstatus()

Prints the quad status information. This is the function used by the **Quad Status** option. It is useful for debugging purposes.

8.3. Channel Manipulation Functions

initchinfo(q, chtype) qdP q; int *chtype;

Assign suitably sized memory channels to *q* based on the image dimensions and the corresponding entry in *chtype*, the list of channel data types. If the current memory channel is larger than the required size, then it is retained. Otherwise, it is replaced by a channel of the determined size. This function serves to initialize the channel pointers stored in the *q* data structure. Note that there is no initialization of the channel elements.

nextch(q, num, ptr, type) qdP q; int num, *type; unsigned char *ptr;

Pass the memory pointer and data type corresponding to channel *num* in quad *q* through parameters *ptr* and *type*, respectively. If *num* refers to an existing channel, then the function returns a 1 in addition to the passed parameters. However, if *num* exceeds the number of channels actually contained in *q*, the function returns a 0 signaling this condition, in addition to the data pertaining to the last channel in *q*. This feature allows two images to be treated channel-by-channel even though they don't have the same number of channels. This is intuitive in cases where, say, a single channel BW image must be combined in some way with a 3 channel RGB image. By repeatedly passing the last available channel, the BW image is implicitly treated as a compatible RGB image, allowing the operation to proceed normally. When the **nextch** function for *both* images return 0s, then we know that neither image has any more channels to offer, and the operation may terminate. This routine is used in all functions that access successive channels.

cpch(q1, c1, q2, c2) qdP q1, q2; int c1, c2;

Copy channel *c 1* of quad *q 1* into channel *c 2* of quad *q 2*. Channel *c 2* inherits the data type of channel *c 1*. If the channel numbers do not refer to valid channels, then an error message is issued.

cpchend(q1, c1, q2) qdP q1, q2; int c1;

Copy channel *c 1* of quad *q 1* onto the end of quad *q 2*. An additional channel is allotted to *q 2* prior to appending channel *c 1*. An error message is issued if the dimensions of *q 1* and *q 2* do not agree. It normally follows a call to **NEWQD**, a macro which returns a new quad with null dimensions and type.

linkch(q1, c1, q2, c2) qdP q1, q2; int c1, c2;

Link channel *c 1* of quad *q 1* onto channel *c 2* of quad *q 2*. This function effectively aliases channel *c 1* in *q 1* as a channel in *q 2*. This permits a filter operating upon *q 2* to access a channel actually contained in *q 1*. **linkch** is useful for passing a specified set of channels to an image operation. For example, if we wish to process only channels 0 and 2, we may link them to a new quad and subject that quad to an image operation. The output of this operation will contain the processed channels 0 and 2. Note that no copying of memory was involved; only a link, or renaming procedure, was needed. It is inherent that the dimensions of all linked channel agree.

linkchend(q1,c1,q2) qdP q1, q2; int c1;

Link channel $c1$ of quad $q1$ onto the end of quad $q2$. This is similar to appending a channel to $q2$ except that the channel memory is only renamed, not copied. **linkchend** is commonly used to collect, without the expense of copying, those channels which are to undergo the same filtering. It normally follows a call to **NEWQD**, a macro which returns a new quad with null dimensions and type.

tfrch(q,ch,h,w,type) qdP q; int ch, h, w, type;

Transfers a channel pointer to channel ch in quad q accommodating memory with dimension $h \times w$ and $type$ data type.

tfrchbf(q,ch,h,w,type,bf) qdP q; int ch, h, w, type; unsigned char *bf;

Replace channel ch in quad q with the channel pointed to by bf . The new channel has dimensions $h \times w$ and data type $type$. Note that all quad channels that are linked to the replaced channel are updated.

maxch(q) qdP q;

Return the number of channels present in quad q .

maxtype(q) qdP q;

Return the maximum channel data type in quad q .

8.4. Type Conversion

ch_conv(q1,types,q2) qdP q1, q2; int *types;

Convert the channel data types in *q1* to those specified by the *types* list, with the result going in quad *q2*. The *types* list contains the channel data type codes given in section 4. A -1 terminator is used to signal the end of the list.

Let *t* be the number of elements in *types*, and *c* be the number of channels in *q1*. Typically, *t* equals *c* to denote a straightforward conversion of each channel in *q1*. However, if *t* < *c*, then the *t* channels are converted and the remaining channels in *q1* are *not* passed onto *q2*. If *t* > *c*, then the last channel in *q1* is replicated into *q2* having the type specified by the *types* entries.

ch_conveq(q1,type,q2) qdP q1, q2; int type;

Converts all *q1* channels into datatype *type*. The result goes into *q2*. Note that by forcing all channels to have the same data type, this is more limited than the **ch_conv** function which permits arbitrary types for each channel. However, this function provides a shorthand means of specifying the new channel data type without having to initialize a channel type list containing identical entries.

ch_convmin(q1,type,q2) qdP q1, q2; int type;

Convert all channels having data types less than *type* into *type*. The result goes into *q2*. Here, the order of the data types refers to their numerical codes. This is used to assure that all channels have at least as much precision as that associated with *type*.

ch_convmax(q1,type,q2) qdP q1, q2; int type;

Convert all channels having data types greater than *type* into *type*. The result goes into *q2*. This is used to assure than no channel has precision greater than *type*.

ch_conv1(q1,c1,q2,c2,type) qdP q1, q2; int c1, c2, type;

Convert channel *c1* of quad *q1* into channel *c2* of quad *q2*, Channel *c2* is cast to the data type specified by *type*.

img_conv(q1,imgtype,q2) qdP q1, q2; int imgtype;

Convert quad *q1* into an image having the *imgtype* specified. This involves performing the type casting and color space conversion necessary to make *q1* conform to the default data types characterized by that *imgtype* value. These default types are found in file *ipdecl.h* and are described in section 4. The result is stored in quad *q2*.

8.5. Image I/O Functions

qdP read_image(filename) char *filename;

Return a quad containing the image in file *filename*. Note that the format of the image is determined from the tagname of *filename*.

save_image(q,filename) qdP q; char *filename;

Save quad *q* in file *filename*. The tagname of *filename* must be compatible with the type of image being saved.

rd_image(fd,q) qdP q; int fd;

Read an image into quad *q* from a file specified by file descriptor *fd*.

sv_image(q,fd) qdP q; int fd;

Save quad *q* into the file specified by the file descriptor *fd*.

rd_imghdr(fd,tag,q) qdP q; int fd; char *tag;

Read the image header from the file specified by file descriptor *fd*. The tagname *tag* is used to indicate the format in which the image header is stored. The resulting height and width data are initialized in quad *q*.

sv_imghdr(q,fd,tag) qdP q; int fd; char *tag;

Save the image header of quad *q* into file *filename*. The tagname *tag* specifies which format to write the header.

sv_screen(fd,tag) int fd; char *tag;

Dump the screen contents into an RGB file with format specified by the tagname *tag*.

8.6. Display Operations

dsply()

Display the contents of quad *nextqd*. When bringing an off-screen quad to view, it must be copied to *nextqd* (using **cpqd**) and then displayed.

plot(q1,ch,smpl,xtheta,q2) qdP q1, q2; int ch, smpl, xtheta;

Plot a relief surface of *q1*'s channel *ch* into *q2*. The *y*-axis is displayed at an angle of *xtheta* counter-clockwise to the *x*-axis. One in every *smpl* pixels is sampled and used to plot the surface. For best results, be sure to blur the channel before subsampling to prevent aliasing artifacts.

dsplyhist(q1,q2,channels) qdP q1, q2; int channels;

Display histogram of *q1* channels in quad *q2*. The channels displayed are specified by the hex number *channels*. The least significant bit refers to channel 0 with each successive bit corresponding to increasing channel numbers. The displayed histograms are color-coded in cycles of red, green, blue, and white colors. The first four channels are displayed in bright red, green, blue and white, respectively. Subsequent channels are displayed with darker values of the same cycle. *q1* and *q2* must be on-screen (visible) quads.

drawborder(qd) int qd;

Draw a blue border around quad *qd*, where *qd* is a visible quad.

eraseqd(qd) int qd;

Erase quad *qd* on monitor. Fill area with black.

8.7. Point Operations: Single Image

histeval(q,ch,histo,len,hmin,hmax)

qdP q; int ch, len; long *histo; double *hmin, *hmax;

Evaluate the histogram of channel *ch* in quad *q*. The histogram is stored in *histo* which has *len* entries. If the range of the channel does not lie properly between 0 and *len*, then the histogram is evaluated on an embedded version of the channel. See section 5 for a definition of embedding. The minimum and maximum values in the channel are returned in *hmin* and *hmax*, respectively.

thr(q1,t1,t2,g1,g2,g3,q2)

qdP q1, q2; double t1, t2, g1, g2, g3;

Threshold the image in quad *q 1* and store the output image in quad *q 2*. *t 1* and *t 2* are the threshold levels. *g 1*, *g 2*, and *g 3* are the intensity levels displayed in the thresholded image. They correspond to the three ranges delimited by *t 1* and *t 2*.

qntize(q1,levels,q2)

qdP q1, q2; int levels;

Quantize the image in *q 1* into *levels* intensities per channel, storing the result in *q 2*.

histflat(q1,q2)

qdP q1, q2;

Perform histogram flattening on the image in quad *q 1*, storing the result in quad *q 2*.

histeq(q1,qlut,q2)

qdP q1, q2, qlut;

Perform histogram equalization on the image in quad *q 1*, storing the result in quad *q 2*. The histograms to which *q 1* must conform are given in quad *qlut*. Note that the histograms of *q 2*'s channels will thereby resemble those given in *qlut*. As usual, if *qlut* has fewer channels than *q 1*, its last channel supplies the histograms for the remainder of *q 1*'s channels.

clip(q1,t1,t2,q2)

qdP q1, q2; double t1, t2;

Clip the values in *q 1* to lie between *t 1* and *t 2*, where $t 1 \leq t 2$. The result goes in *q 2*.

scale_range(q1,t1,t2,q2)

qdP q1, q2; double t1, t2;

Scale the values in *q 1* to lie between *t 1* and *t 2*, where $t 1 \leq t 2$. The output goes in *q 2*.

embed_range(q1,t1,t2,q2)

qdP q1, q2; double t1, t2;

Embed range of *q 1* to lie between values *t 1* and *t 2*, where $t 1 \leq t 2$. The result goes in *q 2*.

gamma_correct(q1,expo,q2)

qdP q1, q2; double expo;

Perform gamma correction on quad *q 1*. The gamma curve is defined as f^{expo} , where *f* represents the intensity range normalized to lie between 0 and 1. Note that the image need not lie in this range. An *expo* value of 1 represents a ramp in which the intensities are unaltered. Larger values yield a family of curves that lie below the ramp and thereby darken the output image. Smaller values brighten the image.

applylut(q1,qlut,q2) qdP q1, qlut, q2;

Apply the LUT in quad *qlut* upon quad *q 1*. The result is stored in quad *q 2*.

applylut_intrp(q1,qlut,q2) qdP q1, qlut, q2;

Identical to **applylut** except that floating point data in *q 1* serves to interpolate into the LUT of quad *qlut*. That is, instead of truncating fractional parts when indexing the LUT, that fraction is used to interpolate between adjacent table entries.

dither_unorder(q1,levels,expo,q2) qdP q1, q2; int levels; double expo;

Perform unordered (random) dithering on the image in quad *q 1*. The output image, with *levels* intensities per channel, is stored in quad *q 2*. Prior to dithering the image is gamma corrected with the *expo* parameter (see **gamma_correct**).

dither_order(q1,dim,expo,q2) qdP q1, q2; int dim; double expo;

Perform ordered dithering on the image in quad *q 1*, storing the output in quad *q 2*. The dimensions of the square dither matrix is *dim*. This value is restricted to be either 4 or 8. Again, the image is first gamma corrected with the *expo* parameter used to specify the correction curve.

dither_floyd(q1,coeff,expo,q2) qdP q1, q2; double *coeff, expo;

Perform error diffusion dithering on quad *q 1* based on the Floyd-Steinberg 4 neighborhood algorithm. The output goes in quad *q 2*. The coefficients of the 4 neighborhood weights are given in the *coeff* list. If *coeff* is 0 (null pointer), then default coefficient values are used. Gamma correction is specified by the *expo* value.

dither_jarvis(q1,coeff,expo,q2) qdP q1, q2; double *coeff, expo;

Perform error diffusion dithering on quad *q 1* based on the Jarvis-Judice-Ninke 12 neighborhood algorithm. The output goes in quad *q 2*. The coefficients of the 12 neighborhood weights are given in the *coeff* list. If *coeff* is 0 (null pointer), then default coefficient values are used. Gamma correction is specified by the *expo* value.

halftone(q1,sz,expo,q2) qdP q1, q2; int sz; double expo;

Halftone the image in quad *q 1* using *sz* × *sz* superpixels. These superpixels contain spiral patterns whose density reflects the intensity it is intended to represent. The maximum value of *sz* is 10. Again, the image first undergoes gamma correction with the value *expo*. The output image is stored in quad *q 2*. Note that the dimensions of *q 2* will be *sz* times larger than those of *q 1*.

8.8. Point Operations: Arithmetic

subtrct(q1,q2,q3) qdP q1, q2, q3;
 $q3 = q1 - q2.$

subconst(q1,const,q2) qdP q1, q2; double *const;
 $q2 = q1 - const.$ The constant values applied to each channel are given in the *const* array.

add(q1,q2,q3) qdP q1, q2, q3;
 $q3 = q1 + q2.$

addconst(q1,const,q2) qdP q1, q2; double *const;
 $q2 = q1 + const.$ The constant values applied to each channel are given in the *const* array.

mult(q1,q2,q3) qdP q1, q2, q3;
 $q3 = q1 \times q2.$

multconst(q1,const,q2) qdP q1, q2; double *const;
 $q2 = q1 \times const.$ The constant values applied to each channel are given in the *const* array.

divide(q1,q2,q3) qdP q1, q2, q3;
 $q3 = q1 / q2.$

divconst(q1,const,q2) qdP q1, q2; double *const;
 $q2 = q1 / const.$ The constant values applied to each channel are given in the *const* array.

add_dolp(q1,q2,q3,const,q4) qdP q1, q2, q3, q4; double const;
 $q4 = q1 + (q2 - q3) * const.$ The name is derived from *adding* the difference of lowpass (*dolp*) between two blurred versions of the original, back onto the original.

8.9. Point Operations: Logic

and(q1,q2,q3) **qdP q1, q2, q3;**
 $q3 = q1 \& q2.$

andconst(q1,const,q2) **qdP q1, q2; unsigned char *const;**
 $q2 = q1 \& \text{const}.$ The constant values applied to each channel are given in the *const* array.

or(q1,q2,q3) **qdP q1, q2, q3;**
 $q3 = q1 | q2.$

orconst(q1,const,q2) **qdP q1, q2; unsigned char *const;**
 $q2 = q1 | \text{const}.$ The constant values applied to each channel are given in the *const* array.

xor(q1,q2,q3) **qdP q1, q2, q3;**
 $q3 = q1 \text{ XOR } q2.$

xorconst(q1,const,q2) **qdP q1, q2; unsigned char *const;**
 $q2 = q1 \text{ XOR } \text{const}.$ The constant values applied to each channel are given in the *const* array.

bic(q1,q2,q3) **qdP q1, q2, q3;**
 $q3 = q1 \text{ bic } q2.$

not(q1,q2) **qdP q1, q2;**
 $q2 = \text{COMPLEMENT OF } q1.$

8.10. Neighborhood Operations

blur(q1,xsz,ysz,q2) qdP q1, q2; double xsz, ysz;

Blur the image in quad *q 1* with a box filter of dimensions $xsz \times ysz$. The output image is stored in quad *q 2*.

blur1D(src,len,offst,winsz,dst)

unsigned char *src, *dst;

int len, offst;

double winsz;

Blur the 1D list of values in *src* using a box filter with window size *winsz*. The increment between successive elements in *src* is *offst*. This permits scanline processing along rows (*offst*=1) or columns (*offst*=width of row). The result is stored in *dst*.

fblur1D(src,len,offst,winsz,dst)

float *src, *dst;

int len, offst;

double winsz;

Identical to **blur1D** except that the blurring is applied over floating point data.

blur_mask(q1,qmsk,qblur,order,q2) qdP q1, qmsk, qblur, q2; double order;

Blur *q 1*, using *qmsk* as the mask image specifying values for which to interpolate between *q 1* and its maximally blurred version, *qblur*. *order* specifies the blur fall-off. The output image is stored in quad *q 2*.

sharpen(q1,xsz,ysz,fctr,q2) qdP q1, q2; double xsz, ysz, fctr;

Sharpen *q 1* using a box filter of dimensions $xsz \times ysz$ and a multiplicative factor of *fctr*. The result is put in quad *q 2*.

median(q1,k,q2) qdP q1, q2; int k;

Apply a median filter on *q 1*, averaging the *k* nearest pixels to the median. The result is stored in *q 2*.

thr_laplace(q1,thresh,q2) qdP q1, q2; double *thresh;

Threshold the Laplacian of the image in quad *q 1* using threshold levels given in list *thresh*. Successive *thresh* entries correspond to successive quad channels. The output image is stored in quad *q 2*.

laplace(q1,mid,fctr,relief,q2) qdP q1, q2; double *mid, *fctr; int relief;

Apply a Laplacian operator on *q 1*, storing the results in *q 2*. The *mid* list supplies the bias values for each channel. Laplacian results are scaled by the appropriate *fctr* entry before being

added to the bias. If *relief* is 0, the absolute value of the Laplacian are used in the computations. If *relief* is 1, the Laplacian results are used directly.

8.11. Geometric Operations

scale(q1,newh,neww,degree,q2) qdP q1, q2; int newh, neww, degree;

Scale the image in quad *q 1* so that it takes on the dimensions of height *newh* and width *neww*. The method in which to scale is specified by *degree*. Current allowable values are 0, 1, and 3 to denote pixel replication, linear, and cubic interpolations, respectively. The result is stored in quad *q 2*.

scale1D(src,len,nlen,offst,degree,dst)

unsigned char *src, *dst;

int len, nlen, offst, degree;

Scale the 1D list of values in *src* so that its length is resized from *len* to *nlen*. The increment between successive elements in *src* is *offst*. This permits scanline processing along rows (*offst*=1) or columns (*offst*=width of row). As in *scale*, the method in which to scale is specified by *degree*. The result is stored in *dst*.

fscale1D(src,len,nlen,offst,degree,dst)

float *src, *dst;

int len, nlen, offst, degree;

Identical to *scale1D* except that the scaling is applied over floating point data.

rotate(q1,n1,n2,ang,q2) qdP q1, q2; double n1, n2, ang;

Rotate the image in quad *q 1* by angle *ang* about the rotation axis, specified by direction cosines *n 1* and *n 2* and the image center. *n 1* is equal to the *x*–projection of the rotation axis divided by the axis' length. *n 2* is computed using the *y*–projection. The *z*–projection is not needed since it is determined by *n 1* and *n 2*. The rotated image is stored in quad *q 2*.

translate(q1,dx,dy,q2) qdP q1, q2; double dx, dy;

Translate the image in quad *q 1* by *dx* and *dy* in the *x* and *y* directions, respectively. The result goes into quad *q 2*.

8.12. Image Compositing Operations

fg_extract(q1,q2,q3) qdP q1, q2, q3;

q2 cuts *q1* where *q2* pixels are non-zero. The result goes into *q3*. This version yields hard edges since *q2* effectively defines a bilevel matte.

fg_overlay(q1,q2,q3) qdP q1, q2, q3;

Overlay *q2* upon *q1* where *q2* pixels are non-zero. The result is found in *q3*.

mat_cut(q1,flg,q2) qdP q1, q2; int flg;

Compute soft matte cut of *q1* into *q2*. If *flg*=0, then *a1*, *q1*'s *alpha* channel, is used directly in the operation. Otherwise, $(1-a1)$ is used to compute the soft matte cut.

mat_over(q1,q2,flg,q3) qdP q1, q2, q3; int flg;

q3 = *q1* over *q2*. If *flg*=0, the resulting *alpha* channel is the sum of the input *alpha* channels. Otherwise, the maximum of the *alpha* channels is taken at each pixel.

mat_in(q1,q2,q3) qdP q1, q2, q3;

q3 = *q1* in *q2*.

mat_out(q1,q2,q3) qdP q1, q2, q3;

q3 = *q1* out *q2*.

mat_atop(q1,q2,flg,q3) qdP q1, q2, q3; int flg;

q3 = *q1* atop *q2*. If *flg*=0, the resulting *alpha* channel is the sum of the input *alpha* channels. Otherwise, the maximum of the *alpha* channels is taken at each pixel.

mat_xor(q1,q2,flg,q3) qdP q1, q2, q3; int flg;

q3 = *q1* xor *q2*. If *flg*=0, the resulting *alpha* channel is the sum of the input *alpha* channels. Otherwise, the maximum of the *alpha* channels is taken at each pixel.

mat_hicon(q1,q2) qdP q1, q2;

The alpha channel of *q1* is stored in *q2*.

mat_darken(q1,num,q2) qdP q1, q2; double num;

Applies the multiplicative factor *num* to the non-alpha channels of *q1*. The result is stored in *q2*. This darkens/brightens the image without introducing fading. Despite the function's name, *num* is not restricted to be less than 1.

mat_opaque(q1,num,q2) qdP q1, q2; double num;

Applies the multiplicative factor *num* to the alpha channel of *q1*, storing the result in *q2*. This

causes the image to become more faded/opaque. There are no restrictions on the value of *num*.

mat_dissolve(q1,num,q2) qdP q1, q2; double num;

Applies the multiplicative factor *num* to all the channels of *q* 1, storing the result in *q* 2. This causes the image to both darken/brighten and become more opaque/faded. Note that there are no restrictions on the value of *num*.

8.13. Image Transforms

fft1D(q1,dir,q2) qdP q1, q2; char dir;

Perform the Fast Fourier Transform on N complex numbers (N pairs of real and imaginary numbers) contained in quad $q1$. The real and imaginary numbers are found in channels 0 and 1, respectively. N is taken as the *height* \times *width* product of $q1$. If *dir* is 'f', the forward FFT is taken. Otherwise, if *dir* is 'i', the inverse FFT is taken. The real and imaginary results are stored in $q2$ in channels 0 and 1, respectively, and are taken as type *float*.

fft2d(q1,dir,q2) qdP q1, q2; char dir;

Perform a 2D FFT on $q1$, storing the results in $q2$. Again, *dir* specifies whether a forward or inverse FFT is computed. The real and imaginary numbers are stored in channels 0 and 1 as type *float*.

ld_fftnum(q1,q2) qdP q1, q2;

Load channel *ch* in quad $q1$ into quad $q2$, so that it is compatible with the input expected by *fft1D* or *fft2D*. This involves converting the channel to be of type *FLOAT_TYPE* (a *float* channel), and appending a second channel of 0s, representing the null imaginary terms. Therefore, $q2$ is made to have two channels, one for the real components and one for the imaginary terms.

ldchar(q1,dir,q2) qdP q1, q2; char dir;

If *dir* is 'f', store the amplitude spectrum, contained in $q1$, into $q2$ with the zero frequency in the center. Otherwise, if *dir* is 'i', load the image, stored in $q1$ as complex numbers, into $q2$.

thin(q,fg,iterations,acted) qdP q; int fg, *iterations, *acted;

Thin image in quad q , putting the skeleton back in q . *fg* is the foreground pixel value. The background is taken as $1 - fg$. Thinning iterates for **iterations* times unless **iterations* is less than 0, in which case thinning proceeds until completion. The number of iterations completed is returned in *iterations*. The number of pixels changed from foreground to skeleton is returned in *acted*.

warpthin(q,fg,DELPXL,SKL,BRIDGE,iterations,acted)

qdP q;

int fg, DELPXL, SKL, BRIDGE, *iterations, *acted;

The enhanced thinning procedure described in section 6. *DELPXL* is the value given to pixels which are deleted and labeled as background. The *DELPXL* may be used to traverse the contour. As each of these pixels is visited, the *DELPXL* label may be reset to the background value, $1 - fg$. The remaining parameters are described in section 6 and *thin*.

shrink(q1,q2,cnct) qdP q1, q2; int cnct;

Shrink binary image $q1$ by peeling away outermost layer. The background is taken to be *cnct*.

connected, where *cnct* is either 4 or 8. The result of a one iteration shrink is stored in *q 2*.

dilate(q1,q2,cnct) **qdP q1, q2; int cnct;**

Dilate binary image *q 1* by coating outermost layer. The background is taken to be *cnct*-connected, where *cnct* is either 4 or 8. The result of a one iteration dilation is stored in *q 2*.

8.14. Image Utility Routines

interlv(q1,q2) qdP q1, q2;

Interleave the channels of quad *q 1*. The data is stored in the first channel of *q 2*.

uninterlv(q1,q2) qdP q1, q2;

Uninterleave the interleaved data stored in *q 1*'s first channel. Store the result in the channels of quad *q 2*.

normalsz(q1,q2) qdP q1, q2;

Normalize *q 1* and *q 2* to take on identical dimensions. The normalization takes the form of added padding such that they mutually superimpose.

fixborder(q1,xmin,ymin,xmax,ymax,q2) qdP q1, q2; int xmin, ymin, xmax, ymax;

Resize the image to fit in the window given by diagonal endpoints (*xmin,ymin*) and (*xmax,ymax*). The size adjustment is made by either clipping or padding the border with a null (0) border. Note that the minimum *x* and *y* values of the image are denoted by the *xoffst* and *yoffst* entries in the quad header. These values must be greater than 0. The maximum coordinate values are determined by the addition of the width and height to the *xoffst* and *yoffst* values. As an example, if *xoffst* = 0 and *xmin* = -3, then the left side of the image will be padded with a 3-column null border. On the other hand, if *xmin* was given as 3, then the first 3 columns on the left side of the image would be clipped off. This operation is useful for normalizing the dimensions of an image which must be superimposed upon a second image.

rgbdcpl(q1,q2) qdP q1, *q2;

If quad *q 1* is a black-and-white image, return it in *q 2*. Otherwise, prompt the user to determine whether the RGB image should be decoupled for subsequent processing. If the user wishes to decouple the channels, *q 1* is returned through *q 2* since all routines treat the channels independently anyway. Otherwise, the luminance of the color image is computed and returned in *q 2*. An eventual invocation of *rgbcouple* will adjust the color components based on this luminance image its computed result.

rgbcouple(q1,q2,q3) qdP q1, q2, q3;

Update the RGB values of *q 1* to follow the result of operating on a (coupled) luminance image. If *q 1* is the same value as *q 3*, no previous coupling was performed, and there is therefore no need to update any values. However, if *q 1* is distinct from *q 3*, then a luminance image had been previously computed (in *q 3*) and together with its result (in *q 2*) are used to determined the new color values to be stored in the output image *q 2*. The method used for this computation is described in section 6.

minmax(q,minval,maxval) qdP q; double *minval, *maxval;

Return the minimum and maximum values of quad q in $minval$ and $maxval$, respectively.

`minmax1(q,ch,minval,maxval)` `qdP q; double *minval, *maxval;`

Return the minimum and maximum values of quad q 's channel ch in $minval$ and $maxval$, respectively.

8.15. Query Routines

askint(str,min,max,default) char *str; int min, max, default;

Print message *str*, range *min* and *max*, and *default*. Re-issue the message as long as the response is not within the range specified. If a newline is entered, return *default*. Otherwise, return valid response.

askstring(str1,str2,str3) char *str1, *str2, *str3;

Print message *str* 1, default response *str* 2, and return response in *str* 3.

askdouble(str,min,max,default) char *str; double min, max, default;

Identical to **askint** except that the parameters now have data type *double* instead of *int*.

askhex(str,default) char *str, *default;

Print message *str* 1 and *default*. Return hex response.

8.16. Frame Buffer Functions

In an effort to make IMPROC portable across different frame buffers, all device-dependent routines are contained in file *ipdev*.c*. In addition, some definitions are found in the header file *ip.h*.

init_fb()

Initialize the frame buffer device, making it available to the user.

flush_fb()

Flush buffers to make the frame buffer picture current.

close_fb()

Close the frame buffer device, making it available for other users.

setrgb(r,g,b) **int r, g, b;**

Set the current color to be (r, g, b) . Subsequent plotting uses this color.

readrgb(r,g,b) **int *r, *g, *b;**

Return the *RGB* value of the current position in (r, g, b) .

readersr(x,y) **int *x, *y;**

Return the current cursor position in (x, y) .

moveto(x,y) **int x, y;**

Move to screen position (x, y) , making it the current position.

drawto(x,y) **int x, y;**

Draw a line from the current position to (x, y) .

drawrel(dx,dy) **int dx, dy;**

Draw a line relative from the current position by (dx, dy) .

line(x1,y1,x2,y2) **int x1, y1, x2, y2;**

Draw a line from position (x_1, y_1) to (x_2, y_2) .

rect(x1,y1,x2,y2) **int x1, y1, x2, y2;**

Draw a rectangle with diagonal vertices (x_1, y_1) and (x_2, y_2) .

rectrel(dx,dy) **int dx, dy;**

Draw a rectangle with one diagonal vertex at the current position and the second vertex at a relative distance of (dx, dy) .

text(x,y,str) int x, y; char *str;

Draw text in *str* at position (x,y).

writepix(height,width,q) int height, width; qdP q;

Write a $height \times width$ image into the frame buffer at the current position. The image is contained in quad *q*. Note that the format in which the image is stored is device-dependent. Thus, some devices will require the data to be interleaved or uninterleaved. Instead, for such low-level access, the user is advised to use **put_urows** or **put_irows** to make this distinction clear.

readpix(height,width,q) int height, width; qdP q;

Read an image from the frame buffer. Store the $height \times width$ image in quad *q*. Note that the format in which the image is stored in *q* is device-dependent. Thus, some devices will naturally leave the data interleaved or uninterleaved in the quad. Instead, for such low-level access, the user is advised to use **get_urows** or **get_irows** to make this distinction clear.

put_urows(x,y,q) int x, y; qdP q;

Display the uninterleaved image stored in quad *q*. Its upper left corner is positioned at location (x,y). Note that the buffer memory stores scanlines in top-to-bottom order. Also, device-dependent issues in display format are automatically handled by this function.

put_irows(x,y,q) int x, y; qdP q;

Identical to **put_urows** except that the image is stored in interleaved format in quad *q*.

get_urows(x,y,h,w,q) int x, y, h, w; qdP q;

Read an image from the frame buffer and stores it into quad *q* in uninterleaved format. The image has dimensions $h \times w$ beginning from the upper left corner at location (x,y). Note that the buffer memory stores scanlines in top-to-bottom order. Also, this function handles the device-dependent issues regarding frame buffer display format.

get_irows(x,y,h,w,q) int x, y, h, w; qdP q;

Identical to **get_urows** except that the image is stored into quad *q* in interleaved format.

erase(x1,y1,x2,y2) int x1, y1, x2, y2;

Erase area on screen bounded by (x 1,y 1) and (x 2,y 2).

prmfll(flag) int flag;

If *flag* is 1, fill subsequent drawn primitives (i.e. rectangle) with current color. Otherwise, leave the primitives unfilled.

clipwndw(x1,y1,x2,y2) int x1, y1, x2, y2;

Define a clipping window with diagonal vertices (x 1,y 1) and (x 2,y 2).

start()

Gain access to the Raster Technology frame buffer.

quit()

Quit access from the Raster Technology frame buffer.

mvcrsr()

Have the cursor follow the mouse movement.

pen()

Draw a streak of blue pixels following the mouse movement.

followersr()

Track the mouse position. This is similar to **mvcrsr** except that the screen position of the moving current point is put in a predefined frame buffer register.

clear()

Turn off the cursor tracking and crosshair.

rdpxlval()

Prompt user to read value from the screen.

rdpxlloc()

Prompt user to read the cursor location.

flood()

Flood the entire frame buffer screen, making it take on the current color. Useful for clearing the screen.

areafill()

Prompt the user to position the mouse in the interior of a region closed by a curve drawn with blue color (0,0,*MXGRAY*). When the user confirms the point, the area is filled with the same blue color.

9. EXAMPLES: PUTTING IT ALL TOGETHER

This section provides some examples that are intended to clarify the usage of library functions. In addition, it is shown how the combination of some basic functions can yield interesting and useful operations.

9.1. Basic Usage

All software invoking IMPROC functions must include the header file *ip.h*. This file contains macros, typedefs, and external declarations. The file containing the *main* function must also include *ipdecl.h* which contains the definitions for the global variables. These variables, some of which are listed below, may be redefined by the programmer. This should be done at the outset in *main*. See *ip.h* and *ipdecl.h* for the full list of global variables and macros. In particular, the data type definitions for image channels is found in *ipdecl.h*. To register some variables which may be dependent on these newly initialized data, the programmer must call *init_vars()*. This should be included even if no global variables are redefined. If the frame buffer is to be accessed by the program, make sure to insert the function *init_fb()* to initialize the frame buffer. If the program will be using *dsply()* to display images, the programmer must also include *init_visqd()* to initialize the visible quads. **IMPORTANT:** *init_vars()* must precede either *init_fb()* or *init_visqd()*.

Here are various global variables and macros that the programmer should be aware.

- | | |
|------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>activeqdp</i> | The active quad, <i>activeqd</i> , is actually a number between 0 and 3. It serves as an index into <i>qdptr</i> , a list of quads, to yield <i>activeqdp</i> , the pointer to the active quad. <i>init_vars()</i> sets <i>activeqd</i> to 0, and <i>activeqdp</i> to <i>qdptr</i> [0]. |
| <i>nextqdp</i> | The next quad, <i>nextqd</i> , is set to 0 in <i>init_vars</i> , and its corresponding quad pointer, <i>nextqdp</i> , is initialized to <i>qdptr</i> [0]. |
| <i>MXGRAY</i> | This is the maximum allowable gray level. By default, it is set to 255. However, this value is reserved by IMPROC for other use. The maximum intensity value for image data is set to <i>MXGRAY</i> - 1, which is <i>MXGRAY</i> - 1. |
| <i>YSZ, XSZ</i> | This specifies the maximum height and width, respectively, of each quad. It is used as the dimensions of the visible quads. For convenience, <i>YSZ</i> - 1 and <i>XSZ</i> - 1 are automatically defined as <i>YSZ</i> - 1 and <i>XSZ</i> - 1, respectively. |
| <i>NEWQD</i> | This returns a new quad with null dimensions and data type (uninitialized). It is used to obtain a quad which will be initialized in some processing operation. It is implied here that it must first be used as a destination quad, in which its attributes will be assigned by the processing function. |

9.2. An Example

The partial program shown below reads an image from *file.r*, displays it, performs histogram equalization, displays the result, and saves it in *output.r*.

```
#include "ip.h"
#include "ipdecl.h"

main() {
    qdP q;

    /* preliminary initialization */
    init_vars();           /* init global variables and pointers */
    init_visqd();          /* init visible quads */
    init_fb();             /* init frame buffer */

    /* read image file; display image in activeqd (=0) */
    q = read_image("file.r");
    cpqd(q,nextqdp);       /* copy into nextqd for dsply() */
    dsply();               /* display input image; nextqd is incremented to 1 */

    /* perform histogram flattening, save, and display result */
    histflat(activeqdp,nextqdp); /* perform hist flattening */
    save_image(nextqdp,"histflat.r");
    dsply();               /* display result */
    /* Note: had dsply() gone before save_image(), then nextqdp
       would have been pointing to another quad, resulting in the
       saving of an uninitialized quad.
    */

    freeqd(q);             /* release quad q */
    close_fb();            /* close access to frame buffer */
}
```

The IMPROC library must be linked at compile time with a *-limproc* appendage. For convenience, a command is available for compiling the user programs with all the necessary frame buffer, math, and IMPROC libraries. Just type *ipcc* with any number of source files.

ipcc file.c

9.3. Another Example: Bandpass Filters

Bandpass filters attenuate all frequency components outside their ranges. They are specified by their cut-in and cut-off frequencies, as well as their gain within the working range. For convenience, we will assume unity gain. The Fourier transform is obviously a straightforward tool to use in implementing a bandpass filter. Here we will show a far cheaper method of arriving at the same result: taking the difference of two lowpass filters. We can expect to make use of the *blur* function since it implements a lowpass filter.

```
main() {
    double const[3];
    qdP q1, q2;

    q1 = read_image("input.r");
    blur(q1,5.,5.,q1);           /* blur q1 by 5 x 5 window */
    blur(q1,5.,5.,q2);          /* q2 is q1 blurred by 5 x 5 window */
    subtrct(q1,q2,q3);          /* q3 = q1 - q2 */
    const[0] = const[1] = const[2] = 15.; /* RGB factors */
    multconst(q3,const,nextqdp); /* nextqdp = q3 * 15. */
    dsply();                    /* display result */
}
```

The segment of code given above implements a bandpass filter. It knocks out some high frequencies with the first blur. The result is stored in *q1*. Some more of the higher frequencies are attenuated with the second blur, where the results are stored in *q2*. Subtracting *q2* from *q1* leaves us with the frequencies that exist in *q1* but were attenuated in *q2*. Since this difference tends to be dim when displayed, each channel was multiplied by 15. In order to preclude saturation effects, it is advised to avoid such magic numbers and instead convert the image to higher precision (to avoid clipping) and then scale the intensity range to maximize the full dynamic range. The segment below converts the image to have data type *short* and then scales the intensity range to lie between 0 and *MXGRAY1*. Recall that *MXGRAY* is reserved by IMPROC, therefore the maximum displayable intensity is *MXGRAY1*, one value less than *MXGRAY*.

```
blur(q1,5.,5.,q1);
blur(q1,5.,5.,q2);
ch_convmin(q1,SHORT_TYPE,q1);
ch_convmin(q2,SHORT_TYPE,q2);
subtrct(q1,q2,q3);
/* scale intensity range of q3 to lie between 0 and MXGRAY1 */
scale_range(q3, 0., (double) MXGRAY1, nextqdp);
dsply();
```

Had we not converted *q1* and *q2* to at least have data type *short*, the above examples would have failed to generate exact bandpassed images because *subtrct* clips negative values to 0 when operating upon *unsigned char* (1 byte per component) images. This is reasonable since displayed pixel values must be positive. However, in order to compute an unclipped bandpassed image without explicit conversion we may use the *add_dolp* function. This function will add the unclipped difference of two blurred images back onto an original image.

An interesting variation can be generated by applying the *imgsharp* function to an already blurred image. Recall that *imgsharp* amplifies the high frequency components of an input image without clipping intermediate results. Since boosting the high frequencies of a blurred image is equivalent to boosting an intermediate frequency range of the original image while simultaneously attenuating the highest frequencies, *imgsharp* can be used to correctly add a bandpassed

image back onto the original with suppressed noise. This yields visually interesting results.

9.4. Channel Manipulation

There are instances when only a subset of the available channels are to undergo processing. Since IMPROC routines uniformly treat all channels of the input quad, we must isolate individual channels into a new quad. That quad is then sent to be processed. In order to avoid excessive copying, we use *linkchend* to link the desired channels to the end of the new quad. The following code segment quantizes only the red and blue channels.

```
qtmp = NEWQD;
linkchend(qinput, 0, qtmp);    /* link channel 0 (red) to qtmp */
linkchend(qinput, 2, qtmp);    /* link channel 2 (blue) to qtmp */
qntize(qtmp,8,qtmp);          /* quantize channels to 8 levels */
freeqd(qtmp);
```

Note that the linking process does not actually copy the channel memory (that would be *cpchend*). Instead, the pointer of the channel is copied and used to index into channel memory. As a result, it is necessary to pass the input quad through the filter and *back into itself*. Otherwise, a new two-channel quad would have been generated. Since this does not comply with the intent of the user, an additional (and wasteful) copying stage would have been necessary to copy the two channels back into their appropriate place in the input quad. As a result, the output quad should be the same as the input quad for all routines exploiting the linking capability.

In some instances, the user may wish to avoid decoupling the RGB channels. That is, instead of allowing the processing routines to handle each channel independently, it may be desired to couple the channels so that they follow the values of a single computed channel. This is particularly useful in routines that make discontinuous value reassignments, i.e. thresholding, quantization, etc. The following code segment illustrates the use of *rgbcouple* to perform this channel coupling.

```
img_conv(q1,BW_IMG,q2);        /* compute luminance of input q1 */
qntize(q2, 8, q3);              /* result of quantization is in q3 */
rgbcouple(q1,q3,q2);            /* restore color values with new result in q3 */
```

Note that the coupling of channel data was performed by computing the luminance image using *img_conv*. The *rgbcouple* routine then uses the input luminance and the quantized luminance to determine the necessary multiplicative factors to apply to the color channels in *q1*. The new result overwrites the previous contents of *q3*. Channel coupling is explained in more detail in section 6.

9.5. Yet Another Example: Image Halftones

The code given below invokes a halftone operation upon an input image with user parameters given on a command line. Due to the fact that this program is intended to generate bitmap images to be outputted by a printer, the pixel values must be flipped. This makes “on” pixels black, and “off” pixels white. An example of the usage is

```
halftone input.r 8 2.4 1024 1024 output.sun1
```

```
#include "ip.h"
#include "ipdecl.h"

main(argc,argv)
int argc;
char **argv;
{
    int i, pxlsz, rows, cols;
    unsigned char *p;
    double gamma;
    qdP q0, q1, q2, qflip;

    if(argc != 7) {
        fprintf(stderr,
            "Usage: halftone infile pxlsz gamma rows cols outfile\n");
        exit();
    }

    pxlsz = atoi(argv[2]);
    gamma = atof(argv[3]);
    rows = atoi(argv[4]);
    cols = atoi(argv[5]);

    XSZ = YSZ = MAX(rows, cols);
    if(XSZ > MXRES) {
        fprintf(stderr, "Error: a dimension exceeds %d\n", MXRES);
        exit();
    }
    init_vars();

    fprintf(stderr, "Reading image: ");
    q0 = read_image(argv[1]);
    img_conv(q0, BW_IMG, q0);
    q1 = NEWQD;
    q2 = NEWQD;
```

```
/* init LUT to make negative due to printing process */
/* black (0) becomes white; >1 becomes black */
qflip = getqd(1, 256, BW_TYPE);
p = qflip->buf[0];
*p++ = 255; /* black becomes white */
for(i=0; i<256; i++) p[i] = 0; /* non-black becomes black */

fprintf(stderr, "Scaling ... ");
scale(q0, rows/pxlsz, cols/pxlsz, 1, q1);

fprintf(stderr, "Halftoning ... ");
halftone(q1, pxlsz, gamma, q2);

fprintf(stderr, "Negative ... ");
applylut(q2, qflip, q2);

fprintf(stderr, "Saving bitmap\n");
save_image(q2, argv[6]);
}
```

9.6. Adding Source Code to IMPROC

Adding source code to IMPROC is performed by the system administrator to incorporate additional *debugged* image processing functions. It involves three steps. Firstly, an entry is made to the appropriate place in the menu system. This requires adding a line in *ipmenus.c* which defines the menu message and the function that is referenced upon an invocation. Secondly, this function must be declared with an *extern* statement in *ip.h*, the IMPROC header file. Thirdly, the calling function is included in *ipfctcall.c*, the file that contains all the functions invoked via the menu.

The calling function must collect all user parameters necessary for the filtering routine. Since parameters cannot be entered directly through the menu invocation, the installed code must prompt the user for this data, pass it onto the processing function, and display the result. When these three procedures are completed, the entire package is recompiled by typing *make* in the directory containing these files. Of course, the added processing function, invoked by the new calling function, must either be appended to one of the files already listed in the makefile, or its file must be added to the list. To reassemble the library archive, type *make iplib*. This generates an archive file called *libimproc.a* which must be moved into */usr/include* for public access.

1. REFERENCES AND SUGGESTED READING

- [Baxes 84] Baxes, G., *Digital Image Processing: A Practical Primer*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [Crow 84] Crow, F., "Summed-Area Tables for Texture Mapping," *Computer Graphics*, (SIGGRAPH '84 Proceedings), vol. 18, no. 3, July 1984, pp. 207-212.
- [Castleman 79] Castleman, K., *Digital Image Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1979.
- [Floyd 75] Floyd, R.W. and L. Steinberg, "Adaptive Algorithm for Spatial Grey Scale," *SID Intl. Sym. Dig. Tech. Papers*, pp. 36-37, 1975.
- [Gonzalez 77] Gonzalez, F. and P. Wintz, *Digital Image Processing*, Addison-Wesley, Reading, MA, 1977.
- [Green 83] Green, W.B., *Digital Image Processing: A Systems Approach*, Van Nostrand Reinhold, New York, NY, 1983.
- [Heckbert 86] Heckbert, P., "Filtering by Repeated Filtering," *Computer Graphics*, (SIGGRAPH '86 Proceedings), vol. 20, no. 4, July 1986, pp. 315-321.
- [Jarvis 76] Jarvis, J.F., C.N. Judice, and W.H. Ninke, "A Survey of Techniques for the Display of Continuous-Tone Pictures on Bilevel Displays," *Comp. Graph. Image Processing*, vol. 5, pp. 13-40, 1976.
- [Kernighan 84] Kernighan, B. and R. Pike, "The UNIX Programming Environment," Prentice-Hall, Englewood Cliffs, NJ, 1984.
- [Paeth 86] Paeth, A., "A Fast Algorithm for General Raster Rotation," *Graphics Interface* 1986, pp. 77-81.
- [Pavlidis 82] Pavlidis, T., *Algorithms for Graphics and Image Processing*, Computer Science Press, Rockville, MD, 1982.
- [Pavlidis 86] Pavlidis, T. and G. Wolberg, "An Algorithm for the Segmentation of Bilevel Images," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, Miami, Florida, June 1986, pp. 570-575.
- [Porter 84] Porter, T. and T. Duff, "Compositing Digital Images," *Computer Graphics*, (SIGGRAPH '84 Proceedings), vol. 18, no. 3, July 1984, pp. 253-259.
- [Pratt 78] Pratt, W., *Digital Image Processing*, John Wiley, NY, 1978.
- [Rosenfeld 82] Rosenfeld, A. and A. Kak, *Digital Image Processing*, Academic Press, NY, 1982.
- [Tanaka 86] Tanaka, A., M. Kameyama, S. Kazama, and O. Watanabe, "A Rotation Method for Raster Image Using Skew Transformation," *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, Miami, Florida, June 1986, pp. 272-277.

- [Wolberg 85] Wolberg, G., "An Omni-font Character Recognition System," M.E.E. thesis, Cooper Union School of Engineering, Oct. 1985. (Available from UMI, Ann Arbor, Michigan).
- [Wolberg 88] Wolberg, G., "Image Warping Among Arbitrary Planar Shapes," *Proceedings of Computer Graphics International*, Geneva, Switzerland, May 1988.