NAVAL POSTGRADUATE SCHOOL
Monterey, California

DTIC
S ELECTE
OCT 1 9 1992
C
D

THESIS

DEVELOPMENT OF A MATLAB TOOLBOX
FOR IMAGE PROCESSING

by

Dorothy J. Freer

June 1992

Thesis Advisor: Charles W. Therrien

REPORT DOCUMENTATION PAGE		Form Approved OMB No 0704-0188

1a REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b RESTRICTIVE MARKINGS

2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	Approved for public release; distribution is unlimited

4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)

6a NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b OFFICE SYMBOL (If applicable) EC	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School

6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000

8a. NAME OF FUNDING/SPONSORING ORGANIZATION	8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (City, State, and ZIP Code)

10 SOURCE OF FUNDING NUMBERS			
PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)

DEVELOPMENT OF A MATLAB TOOLBOX FOR IMAGE PROCESSING

12 PERSONAL AUTHOR(S)
FREER, Dorothy J.

13a TYPE OF REPORT Master's Thesis	13b TIME COVERED FROM _____ TO _____	14 DATE OF REPORT (Year, Month Day) 1992 June	15 PAGE COUNT 49

16 SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the US government.

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	image processing; MATLAB; Toolbox; display; MEX-files

19 ABSTRACT (Continue on reverse if necessary and identify by block number)
This thesis provides an Image Processing Toolbox for use with MATLAB which contains ready-made tools for students and faculty who wish to continue research in image processing and related areas. The Toolbox is available for several computer environments. The documentation provided with the distribution diskette contains both a tutorial and reference section in the MATLAB style. This thesis report provides information needed to write and compile C language programs for use as MEX-files, an overview of the Toolbox, and a case study which illustrates the use of some of the functions in the Toolbox.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a NAME OF RESPONSIBLE INDIVIDUAL THERRIEN, Charles W.	22b TELEPHONE (Include Area Code) 408-646-3347	22c OFFICE SYMBOL EC/Ti

i

DEVEVLOPMENT OF A MATLAB TOOLBOX FOR IMAGE PROCESSING

by

Dorothy J. Freer
Lieutenant, United States Navy
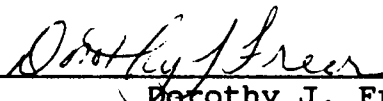B.S., U.S. Naval Academy, 1982

Submitted in partial fulfillment of the
requirements for the degree of
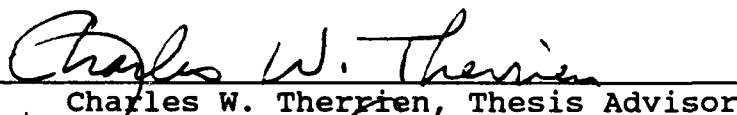
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1992

Author: _____
Dorothy J. Freer

Approved by: _____
Charles W. Therrien, Thesis Advisor

_____
Roberto Cristi, Second Reader

_____
Michael A. Morgan, Chairman
Department of Electrical and Computer Engineering

ii

ABSTRACT

This thesis provides an Image Processing Toolbox for use with MATLAB which contains ready-made tools for students and faculty who wish to continue research in image processing and related areas. The Toolbox is available for several computer environments. The documentation provided with the distribution diskette contains both a tutorial and reference section in the MATLAB style. This thesis report provides information needed to write and compile C language programs for use as MEX-files, an overview of the Toolbox, and a case study which illustrates the use of some of the functions in the Toolbox.

iii

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor Therrien for giving me the support and encouragement to complete this thesis and the accompanying book.

My sister Carol who was always available for keeping me going through the rough spots.

My dear friends and confidantes, Karen Callaghan and Karen Hagerman, whose constant humor kept me laughing, and whose technical expertise made the completion of this thesis a breeze.

My husband Bob and daughter Kerri, whose encouragement, support, and love have meant so very, very much.

I. INTRODUCTION

A. AN OVERVIEW AND PURPOSE

The MATLAB programming environment is a user-friendly workspace oriented system based on a very high level language that has vectors and matrices as its main variables. There are several toolboxes currently available for use with MATLAB that provide added benefits. These include the Signal Processing, Controls, and Systems Identification toolboxes, to name a few. This thesis develops a new toolbox for MATLAB called the Image Processing Toolbox. Many of the basic functions in the toolbox were provided by Erkan Aykaç in his thesis, *"Enhancement of Image Processing Capabilities for Different Environments"*.[Ref. 5] This thesis adds to and improves the efficiency of many of the functions written by Aykaç and provides several other new functions. The result is a set of image processing functions contained in an Image Processing Toolbox, complete with a Tutorial and Reference Guide, as well as distribution diskette. The complete software includes input/output and display routines, morphological operations, filters, histogram manipulation functions, edge detection and enhancement operations, and size manipulation functions. The functions are contained in M-files and MEX-files, and are available for use in PRO-MATLAB, 386-MATLAB, and PC-MATLAB. The MEX-files are source coded in the C language. For the 386-MATLAB version the software was written and compiled using Metaware High C with the Phar Lap 386|Dos-Extender. For the PC-MATLAB version, the source code was written and compiled using Borland Turbo C. The source code for PRO-MATLAB was compiled using the compiler on the SUN Microsystems Operating System version 4.1.1.

1

This thesis describes the new toolbox and provides documentation for adding to the system. It gives detailed descriptions for how to write and compile C source code for all three systems, as well as instructions for accomplishing the task for the new MATLAB version 4.0, that will soon become available. In order to upgrade the toolbox to version 4.0 new source code need not be written; there is a switch that can be used to compile the version 3.x code for use in version 4.0. The procedures are described in Chapter II. In addition to the above, this thesis provides descriptions for use of the software, and a case study that illustrates application of the toolbox functions to an image processing problem.

II. CREATING MEX-FILES

A. INTRODUCTION

Most functions written by users in MATLAB are written as so-called M-files. These are ASCII files written in the (MATLAB) language and are interpreted by the MATLAB interpreter. MATLAB also provides the ability to write compiled functions in FORTRAN or C and incorporate these into the language. MATLAB refers to these as "MEX" files for "MATLAB executable". For development of this Toolbox we found it highly desirable, and sometimes absolutely essential to implement some of the functions as MEX-files.

B. WHY MEX-FILES INSTEAD OF M-FILES

The objective of writing a function in MATLAB should be to improve the ease with which recurring calculations are performed. To that end one must consider the advantages and disadvantages of both the M-file option and the MEX-file option. The M-file option may appear to be the simplest route, however since M-files are interpreted, not compiled, the speed of calculations may be hampered by MATLAB's methods of computation. If one can take advantage of the efficiency with which MATLAB performs array-based calculations, then the M-file approach may be the best solution. In other cases involving large amounts of computations, M-files may be too slow. "For loops" are especially slow since all of the variable indexing and testing is done by the interpreter. On the other hand MEX files, written in C, can directly take advantage of the machine index registers and test and branch instructions and so can perform computations that require loops extremely fast. Another advantage to the MEX-file solution is the user's choice of

3

what type of numerical calculations are performed. In C, the programmer has a choice of using several variable types, such as the unsigned or signed char, unsigned int, short int, and double. MATLAB automatically stores all variables as double precision floating point numbers and thus performs calculations on double precision floating point numbers This involves sixteen bytes for each number stored and computation may be relatively slow even with the math coprocessor. When C programming is used, the programmer can choose short int, which involves two bytes, or even unsigned char, which involves only one byte per number! The savings in computation time can be tremendous when one considers that a typical size for an image matrix is 512 by 512 elements or larger. The following descriptions for creating MEX-files assumes the reader has a working knowledge of the C programming language.

C. COMPILATION OF C PROGRAMS

MEX files are compiled using the CMEX utility contained in the \MATLAB\MEX directory of each version of MATLAB. In order to use C programs with MATLAB they have to have special characteristics that are unique to MEX programs. Most of these characteristics are explained in the *Calling C and FORTRAN Subroutines* section of the PC-MATLAB, 386-MATLAB, and PRO-MATLAB manuals.[Ref. 7, Ref. 8, Ref. 9] Some points to remember are listed below:

- At the very least the file "cmex.h" and the library <math.h> must be **#included** in the function to be compiled. Several *MEX* routines are internal to MATLAB and, when used in the MEX file in place of the standard C routines, eliminate the requirement to include and thus link the entire C libraries.

- **main( )** is not used, **user_fcn( )** is used instead. The arguments of **user_fcn( )** are nlhs,plhs,nrhs,prhs. The arguments nlhs and nrhs represent the number of left-hand

4

side (output) arguments and the number of right-hand side (input) arguments, while plhs and prhs are pointers to arrays of length nlhs and nrhs, respectively, that are pointers to the output and input matrices.

- A Matrix struct pointer must be defined for each variable, both input and output.

- The number of left-hand side arguments and right-hand side arguments must be correct. This is a difference encountered when invoking a MEX function as opposed to a function written as an M-file. An error occurs if there is no output argument when the C program is expecting one. (This can be avoided if an additional subroutine is included to create "ans" as the output argument, however that is a somewhat cumbersome process.)

- Matrices are stored in MATLAB columnwise, and are stored in C row-wise.

- The initial coordinates of a matrix stored in C are [0][0], the initial coordinates of a matrix stored in MATLAB are (1,1).

The same section of the PC-MATLAB manual explains the contents of the \MATLAB\MEX directory and gives an example of C source code. The example, YPRIME.C, is very useful for illustrating the organization of MEX files. Two other example functions written by Erkan Aykaç are **mexampl1** and **mexampl2**. The descriptions for these are in his thesis, [Ref. 5: Appendix A], and the source code and compiled MEX-files are provided on the MATLAB Image Processing Toolbox distribution diskette.

D.   USEFUL SUBROUTINES FOR IMAGE PROCESSING FUNCTIONS

The special characteristic of images that makes MEX files much faster for computations is that they are exclusively stored as integer valued matrices. That is, all data manipulation done on image matrices an some computations involve only integer

5

arithmetic (computations, such as filtering, however, may need to be done in floating point). When writing a MEX file, the variables are automatically cast in the *Matrix struct* as defined by the CMEX utility. The *Matrix struct* casts all variables as double precision floating point one-dimensional arrays. Thus for every pixel there is an index calculation to reach the pixel, and then the calculations performed are on doubles. In order to avoid these superfluous calculations each time, we have provided three functions that can be added to the C program that:

1.   Recast the real pointer variable as a one-dimensional integer array.

2.   Put the integer array into a doubly-dimensioned array.

3.   Once calculations are complete, recast the output variables as one-dimensional arrays of doubles.

These functions are called *recalloc, redimension,* and *rematlab*, respectively. The source code for them is shown in Figure 3-1 and an example of their use, the function **equal**, is printed in Appendix A.

All of the previous points regarding writing MEX files hold for all three versions of MATLAB (PC-MATLAB, 386-MATLAB, and PRO-MATLAB). The size of the image data file is not a concern when using 386- or PRO-MATLAB since the operating system that is being used will dictate how large the data files can be. In PC-MATLAB, however, there is a restriction on the size of the variables that MATLAB can use. The maximum number of elements that a single matrix can have in PC-MATLAB is 8188 elements. Since the input/output routines contained in this thesis's Toolbox are all written in C, the standard MATLAB check of the size of the variable before loading it into the environment is not performed. Therefore, when writing MEX files for use in the PC-MATLAB environment, an additional check must be added to ensure that there is no attempt to import a variable too large into the environment. This is a short *if loop* that checks the

6

```
short int** recalloc(m,n)
int  m,n;
{
short int **array;
int  i;
   array=(short int**) mex_calloc(m,sizeof(short int*));
   for (i=0;i<m;++i)
     array[i] = (short int*) mex_calloc(n,sizeof(short int));
   return array;
}
```

(a)

```
void  redimension(array_out,array_in,m,n)
short int **array_out;
int       m,n;
double    *array_in;
{
int  i,j;
   for (i=0;i<n-1;++i)
     for (j=0;j<m-1;++j)
       array_out[j][i]=array_in[i*m+j];
}
```

(b)

```
void  rematlab(array_out,array_in,m,n)
short int **array_in;
int       m,n;
double    *array_out;
{
int  i,j;
   for (i=0;i<n-1;++i)  {
     for (j=0;j<m-1;++j)
       array_out[i*m+j]=(double)array_in[j][i];
   }
}
```

(c)

Figure 3-1. C-code for (a) Recalloc, (b) Redimension, (c) Rematlab.

7

total number of elements in the data file. If the image is too large only the upper left corner, 64 by 64 pixels, is loaded into MATLAB. All of the MEX-files for PC-MATLAB perform this check. The code for the *if loop* is shown in Figure 3-2.

```
unsigned long  total;    /*total is the total number of 8-bit elements in the image file*/
unsigned int   m,n;            /*m,n are the image dimensions*/
 if (total>8188L) {
    for (j=0;j<16;j++) {         /*gets the pixel values in a row by row fashion*/
      for (i=0;i<16;i++)
        b[j+16*i]=(double)fgetc(in);  /*b is the output variable, in is the in_file pointer*/
      for (k=0;k<(n-16);k++)
        fgetc(in);                /*dump the rest of the line beyond 16*/
    }
 }
```

Figure 3-2. If-loop for truncating size of matrix.

E.   CONVERSION OF MEX FILES TO PRO-MATLAB VERSION 4

The newest version of MATLAB, version 4.0, is not yet available for public release. The beta copy has been released to a limited number of users including the Naval Postgraduate School. The information provided in this section was obtained from the beta documentation [Ref. 4] that was supplied with the Beta 3 PRO-MATLAB version 4.0. The PRO-MATLAB version is the only MATLAB 4.0 available at this time, and no specific information for the 386-MATLAB or PC-MATLAB is available yet.   The flag to use to convert code written for MATLAB version 3.5 to compile and use in MATLAB version 4.0 is **-v3.5** and is added at the end of the cmex command. A sample command is shown below:

cmex foo.c -v3.5

All of the C source code provided on the distribution diskette with the Image Processing Toolbox can be compiled for version 4.0 using the flag.

8

There are some significant changes from the version 3.5 MEX-file structure. The program must now contain two distinct sections, a "gateway routine" and a "computational routine." [Ref. 4] The gateway routine serves as the interface between MATLAB and the computational routine. The computational routine is the portion of the code that actually performs the numerical computations. Reference 4 contains an example MEX-file program, YPRIME.C. Significant points to note from that example are:

- The entry point to the computational routine must be called **mexFunction** instead of user_fcn while the arguments remain the same.

- The computational routine may be placed before the gateway routine.

Several routines that were contained in *cmex.h* have been renamed in *mex.h*. Table 2-1 contains a partial list. (See Ref. 4 for a complete list.)

One additional feature of MATLAB version 4.0 that is of some interest to this author is the addition of a switch for the cmex command that creates a stand-alone program from the MEX-file code for use with a standard C debugger! This feature will undoubtedly save many hours of frustration! The reader is cautioned that the beta version may not be exactly the same as the publicly released version 4.0. Therefore some information in this section may not be correct when the final product is released from The MathWorks.

TABLE 2-1

CONVERSIONS FROM MATLAB VERSION 3.5 TO VERSION 4.0

MATLAB 3.5	MATLAB 4.0
user_fcn	mexFunction
mex_error	mexErrMsg
matlab_fcn	MexCallMATLAB
mex_calloc	mxCalloc
p ->m	mxGetM
=p ->pr	mxGetPr
create_matrix	mxCreateFull

III. MATLAB TOOLBOX

A. IMAGES AS MATRICES

MATLAB uses only one type of object, a rectangular numerical matrix that can have real or complex elements. Vectors, i.e. variables indexed by a single integer rather than a pair of integers, are also recognized by the language, but in reality these variables are the same as a matrix with either a single row or a single column. Scalars are also recognized as matrices with one row and one column. In the Image Processing Toolbox the matrix is further considered to be in one of three possible categories:

Numerical— a numerical variable (matrix, vector or scalar). The values can be either real or complex.

Image— a rectangular or square array of real integers valued from 0 to 255. These values represent eight-bit gray levels or intensity values.

Graphic— an array of binary values (0 or 1).

Some functions work only with graphics, some only with images, and some with all three types. The **help** command tells you which data type is used for each function.

B. INPUT/OUTPUT AND DISPLAY FUNCTIONS

Images can be stored in several formats. The two specific types recognized by this toolbox conform to the USC/SIPI and ITEX PCplus data structures. The USC/SIPI format does not store a header in the image; the ITEX PCplus may or may not store a header. Both store the image in row order and store each pixel as a single byte. The image input/output routines of this toolbox check for image type and strip out the header information when retrieving the data.

The function **readhead** returns the header information from the image file. This function is useful when the size or other information about the image is desired without loading the entire image into the MATLAB environment.

The functions **readim** and **rim** both strip the header information from the image and load the image into a variable in the MATLAB environment. **readim** returns the entire image while **rim** returns only a portion of the image specified when you invoke it.

When an image is obtained by using the frame grabber in Spanagel 315 with the *prt_menu* software on the PC, the image is stored as a 511 by 511 matrix. Most of the applications in this Image Processing Toolbox function best with images that are square and have dimensions that are powers of 2. Therefore the function **ITEX2PCP** has been included in the Toolbox. It simply takes a 511 by 511 matrix and appends a row and column of zeros to create a 512 by 512 matrix.

The functions **putim** and **saveim** save images to the disk. The **putim** function saves the image in the USC/SIPI format with no header. The **saveim** command saves the image in the ITEX/PCplus format with a header and the option to include additional comments. The header and comments are returned by the **readhead** functions mentioned above.

The two remaining input/output functions, **putdata** and **getdata,** are used with matrices instead of images and are not specific to image processing. The data for these functions may be generated by any programming language, and can be used in a variety of applications within MATLAB. **putdata** creates an ASCII data file containing a single matrix variable and **getdata** reads this type of file and creates a variable in the workspace. Since the data file is in ASCII format, it may be created, viewed, or modified with any standard text editor. The **getdata** and **putdata** function pair is a convenient way to transfer data between MATLAB and any other language such as C, FORTRAN, or APL. These two functions are similar to the *load* and *save* commands in MATLAB with the

added benefit that you can edit the data file. Currently, however the format does not handle complex numbers. Complex matrices must therefore be stored as a pair of real matrices.

The data file for these two functions has a relatively simple format. The first line, indicated by an end of line character or carriage return, contains the matrix row and column dimensions, in order. Generally there are one or two integer numbers in the first line. If there are more than two integers in the first line, **getdata** counts them, multiplies them all together, and returns the results of the count, and the data as a column vector with a length equal to the product of the numbers in the first line. If there is one integer in the first line, **getdata** returns a column vector of the length specified by that number. If there is a non-integer in the first line **getdata** truncates the number at the decimal point and interprets it as an integer.

The following examples illustrate the use of **getdata** and **putdata**:

Consider the file:

```
4   3

1.0       3       .5E01     7

9       11.0       2        4

60E-02  8         10        12        .
```

This would be read by **getdata** to the matrix:

$$
\begin{bmatrix}
1.0000 & 3.0000 & 5.0000 \\
7:0000 & 9.0000 & 11.0000 \\
2.0000 & 4.0000 & 0.6000 \\
8.0000 & 10.0000 & 12.0000
\end{bmatrix}
$$

On the other hand the file:

```
3    4
1    3    5    7    9
11   2    4
6    8
10
12
```

would be read by **getdata** as the matrix:

$$\begin{bmatrix} 1 & 3 & 5 & 7 \\ 9 & 11 & 2 & 4 \\ 6 & 8 & 10 & 12 \end{bmatrix}$$

Note that it does not matter how many numbers are contained on each line, just the first line numbers determine the matrix dimensions. If the number of data points in the file is not equal to the product of the numbers in the first line, **getdata** appends zeros or truncates the input, whichever is necessary. Also, if there is data that **getdata** does not recognize as real numbers, it skips that data.

One additional function available on the UNIX workstations at the Naval Postgraduate School and with 386-MATLAB is the **show** command. The function **show**, when used with *Sunview* windows or in 386-MATLAB, allows the user to display images while inside the MATLAB environment. Invoking the **show** command in 386-MATLAB calls the **DISPLAY** menu described below. In the UNIX environment, **show** calls the program *looksun* to open a window and display the image on the screen.

Displaying images while inside PC-MATLAB is not possible at this time. However the separate stand-alone program called **DISPLAY** is available. To use this program you must save your image to a file and exit MATLAB. (Under the Windows 3.1 environment

you can display in a separate DOS window without exiting MATLAB in its own window.) The command **display** *filename* will call up **DISPLAY** menu. From this menu you can display your entire image, part of your image, or reduce the size of your image and then display it. ESC returns you to the prompt from which you called **display**, either the C:> prompt or the MATLAB>> prompt in 386-MATLAB (if **show** was used).

C. EDGE DETECTION OPERATIONS

A very useful set of tools for the image processor is the set of edge detectors. The Image Processing Toolbox includes the following edge detectors: **gradv**, **gradh**, **laplac**, **roberts**, and **sobel**.

Edges in images are caused by spatially abrupt changes in intensity from one region in an image to the next. The method of determining edges in all of the edge detectors in this Toolbox is based on computing the local derivative, or gradient operator. By analyzing the value of the gradient the program can determine whether an edge is present or not. A high (positive or negative) value of the gradient indicates presence of an edge while a low or zero value for the gradient indicates no edge.

The **gradv** and **gradh** functions determine vertical and horizontal edges, respectively. They are also the basis for the **sobel** function. The gradient functions use a digital approximation for determining the gradients and employ a mask that is convolved with the image.

The vertical mask is for **gradv** is
$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

The horizontal mask is for **gradh** is
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

15

These masks are the vertical and horizontal gradient components of the Sobel operator. They weight the pixels closest to the center by 2 to produce additional smoothing and use a 3 × 3 operator instead of a 2 × 2 operator to make the derivative less sensitive to noise. The Sobel operator, $G(m,n)$, is then defined as:

$$G(m,n) = \sqrt{\nabla_h^2(m,n) + \nabla_v^2(m,n)}$$
$$\text{where } \nabla_h = gradh$$
$$\text{and } \nabla_v = gradv$$

These functions, ∇_h and ∇_v, are simple two dimensional convolutions, and can be implemented using the *conv2* function in the SIGNAL PROCESSING TOOLBOX; however, they are provided as MEX-files in the Image Processing Toolbox for speedier execution. The implementation in C takes advantage of the integer-valued nature of images and the fixed masks, it is therefore much faster than the *conv2* function in MATLAB.

The **roberts** operator, or *Roberts gradient*, is another method of approximating the gradient. Roberts used the cross-differences of the pixels to determine the gradient. Using the pixel references in Figure 3-1, Roberts gradient, $Gr[f(m,n)]$, is defined for an image as [Ref. 1:p. 177]:

$$Gr[f(m,n)] = |f(m,n) - f(m+1,n+1)|$$
$$+ |f(m+1,n) - f(m,n+1)|$$

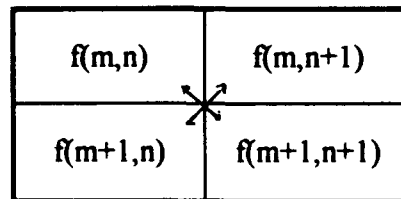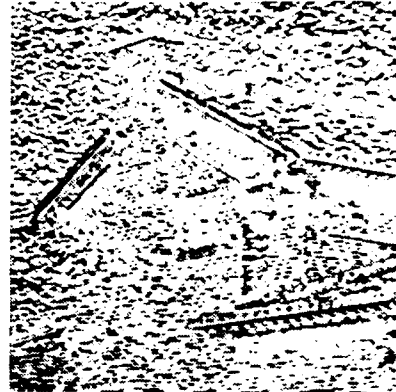| f(m,n)	f(m,n+1)
f(m+1,n)	f(m+1,n+1)

Figure 3-1. Roberts Operator pixel representation.
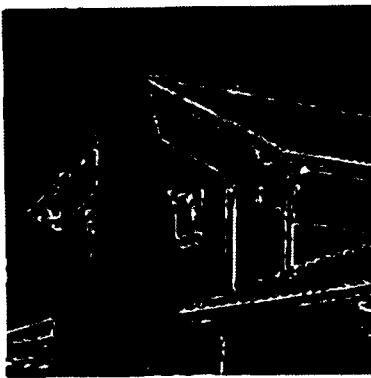
Figure 3-2 shows four images of a house: the original, and three edge images produced using gradh, sobel, and roberts.
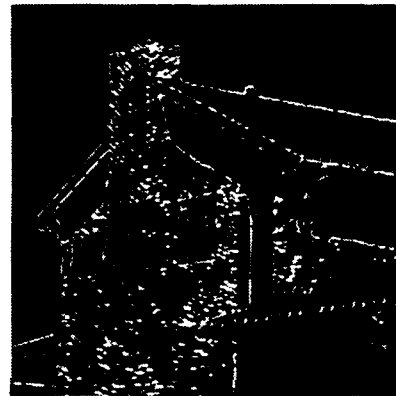
Figure 3-2. Edge Detectors (a) Original image, (b) Edges produced by gradh operator, (c) Edges produced by Sobel operator, (d) Edges produced by Roberts operator.

D. FILTERS

Filters are used for a variety of applications in image processing. For example, lowpass filtering is used to smooth edges and eliminate noise spikes in images. Lowpass filtering blurs the image by taking out the high frequency components of the image's Fourier transform. Highpass filtering is sometimes used for just the opposite effect, image sharpening. By suppressing the low frequency components of the image's Fourier transform, edges are enhanced.

Two specific filter functions, **lowpass** and **highpass,** are provided in the Image Processing Toolbox. These filters are circular Butterworth filters with a cutoff frequency and filter order specified by the user. A third filter is the **blur** function which simply averages the pixels over a mask that is convolved with the entire image.

The median filter function, **med,** is a type of nonlinear image processing filter. It is generally used to suppress spurious noise. The median filter takes a small section surrounding a point (size specified by the user) in the image and replaces that point by the median value of the points in the section. Small isolated noise spikes smaller in area than the size of the filter, are suppressed, but edges tend to be preserved. Figure 3-3 below shows the USC image, Lenna, with random white noise added, and the effect of median filtering.

(a)   (b)

Figure 3-3. Lenna (a) with noise, (b) image in *(a)* **med** filtered.

Another use for the **med** function is to smooth an image that has been enlarged using the **reduce** command. The **reduce** function is used to either reduce or enlarge an image by sub-sampling or replicating the pixels. The function is written as an M-file and takes advantage of the array processing capabilities of MATLAB. For this reason the reduction or enlargement factor must be an integer. The size of the output image is

18

determined by raising 2 to the power of the reduction factor. For enlargement the number is a negative integer, while for reduction the number is a positive integer.

When enlarging an image the result is a somewhat crude image, resembling that of an image displayed in four-bit binary instead of eight-bit binary. Below is an algorithm to enlarge an image by 4, (the image is stored in the variable X). The images in Figure 3-4 show the reduced image and the enlarged images. Figure 3-4(b) is simply an enlargement without any additional filtering and Figure 3-4(c) uses the algorithm below.

Y = **reduce**(X,1);
tempy = **med**(Y,3);
Y = **reduce**(*tempy*,1);
Yout = **med**(Y,3);

(a)

(b)   (c)

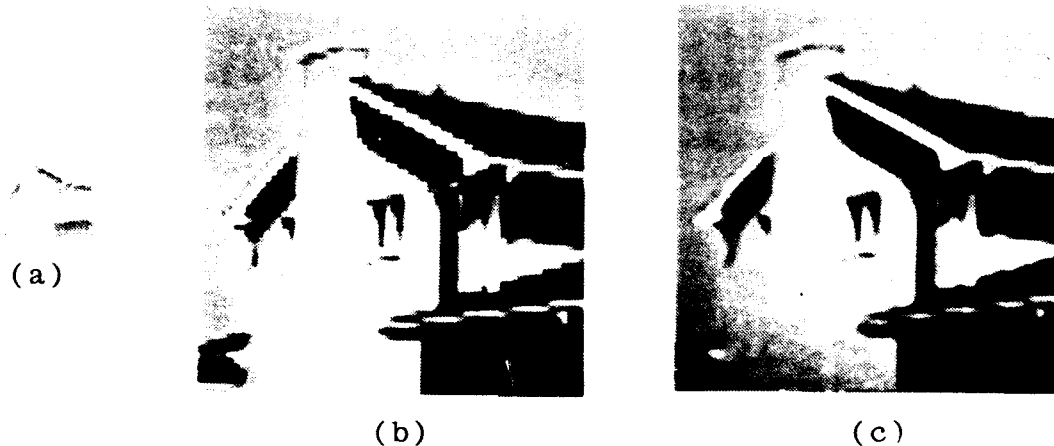Figure 3-4. Enlargement using *reduce*. (a) Original House, (b) image in *(a)* quadrupled once, (c) image in *(b)* doubled two times and **med** filtered after each doubling.

E.   MORPHOLOGICAL OPERATIONS

The term *morphological* means "having to do with structure or form." Morphological operations in the Image Processing Toolbox generally operate on graphics, i.e., matrices consisting of ones and zeros.

The basic functions in the morphological category include Minkowski addition and subtraction. Combinations of these two functions, along with the set union and set difference operations create many other interesting morphological operations. Included in the Image Processing Toolbox are the functions **bound, close, minkadd, minksub, open,** and **graphic**.

Minkowski addition (also called dilation [Ref. 2:p. 384, Ref. 3:p. 473]) is defined as:

$$X \oplus B \equiv \{x : B_x \cap X \neq 0\}$$

where B_x is B translated to have its origin at x.

$X \oplus B$ is therefore the set of points, determined by the position of the origin of B, where the intersection of B and X is not null. The Toolbox function **minkadd** performs this function.

The following example illustrates Minkowski addition:

Let X be represented by the following object:

```
O O O O O O O O O O O
O O O O O O ● ● ● ● O
O O O O O O ● ● ● ● O
O O O O O O ● ● ● ● O
O O O O ● ● ● ● O O O
O O O O ● ● ● ● O O O
O ● ● ● ● ● ● ● O O O
O ● ● ● ● O O O O O O
O ● ● ● ● O O O O O O
O O O O O O O O O O O
```
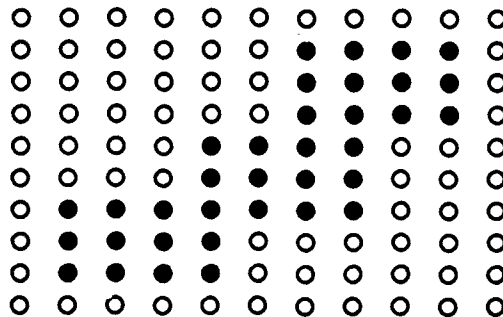
Let B, which is referred to as the structure element, be represented by this object:

```
O O ●   origin
O O ●
● ● ●
```

20

Then $X \oplus B$ is:

```
O  O  O  O  O  ●  ●  ●  ●  ●  ●
O  O  O  O  O  ●  ●  ●  ●  ●  ●
O  O  O  O  O  ●  ●  ●  ●  ●  ●
O  O  O  ●  ●  ●  ●  ●  ●  O  O
O  O  O  ●  ●  ●  ●  ●  ●  O  O
●  ●  ●  ●  ●  ●  ●  ●  ●  O  O
●  ●  ●  ●  ●  ●  ●  O  O  O  O
●  ●  ●  ●  ●  ●  ●  O  O  O  O
●  ●  ●  ●  O  O  O  O  O  O  O
●  ●  ●  ●  O  O  O  O  O  O  O
```
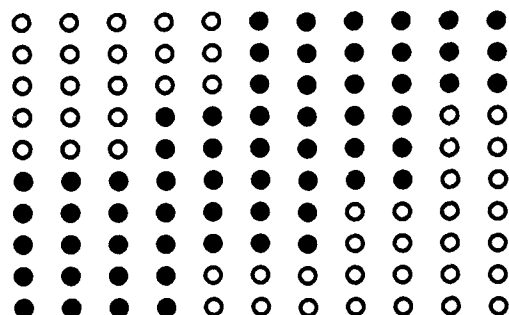
Minkowski subtraction (also called erosion [Ref. 2:p. 384, Ref. 3:p. 476]) is defined as:

$$X \ominus B \equiv \{ x : B_x \subset X \}$$
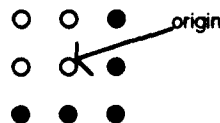
where B_x is B translated to have its origin at x.

The Toolbox function **minksub** performs this function.

Again let X be represented by the following object:

```
O  O  O  O  O  O  O  O  O  O  O
O  O  O  O  O  O  ●  ●  ●  ●  O
O  O  O  O  O  O  ●  ●  ●  ●  O
O  O  O  O  O  O  ●  ●  ●  ●  O
O  O  O  O  ●  ●  ●  ●  O  O  O
O  O  O  O  ●  ●  ●  ●  O  O  O
O  ●  ●  ●  ●  ●  ●  ●  O  O  O
O  ●  ●  ●  ●  O  O  O  O  O  O
O  ●  ●  ●  ●  O  O  O  O  O  O
O  O  O  O  O  O  O  O  O  O  O
```

Let B be represented by this object:

```
O  ●  ○   origin
●  ●  O
O  O  O
```

Then $X \ominus B$ is:

```
O  O  O  O  O  O  O  O  O  O  O
O  O  O  O  O  O  O  O  O  O  O
O  O  O  O  O  O  O  ●  ●  ●  O
O  O  O  O  O  O  O  ●  ●  ●  O
O  O  O  O  O  O  ●  ●  O  O  O
O  O  O  O  O  ●  ●  ●  O  O  O
O  O  O  O  ●  ●  ●  ●  O  O  O
O  ○  ●  ●  ●  O  O  O  O  O  O
O  O  ●  ●  ●  O  O  O  O  O  O
O  O  O  O  O  O  O  O  O  O  O
```
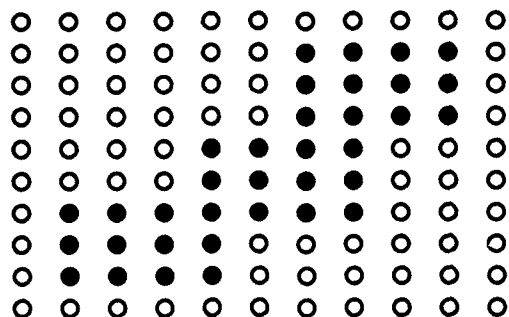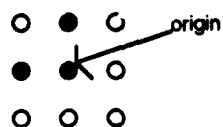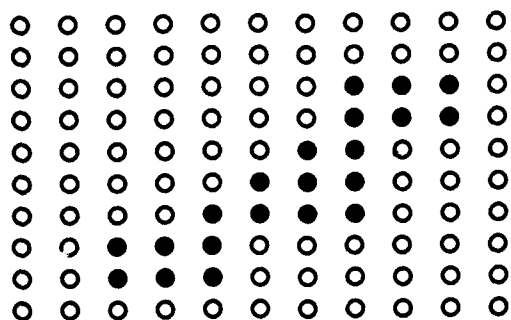
Bound is another useful morphological operation. It returns the outline or boundary of an object. It uses **minksub** and the set difference operation (& ~):

$$\text{bound}(X) = X \ \&\sim \text{minksub}(X, \text{ones}(3)).$$

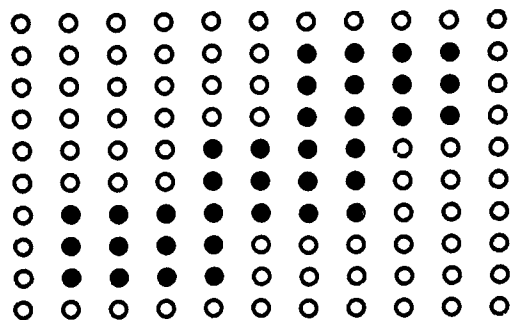The following is an example of **bound**:

Let X be the same as in the previous example:

```
O  O  O  O  O  O  O  O  O  O  O
O  O  O  O  O  O  ●  ●  ●  ●  O
O  O  O  O  O  O  ●  ●  ●  ●  O
O  O  O  O  O  O  ●  ●  ●  ●  O
O  O  O  O  ●  ●  ●  ●  O  O  O
O  O  O  O  ●  ●  ●  ●  O  O  O
O  ●  ●  ●  ●  ●  ●  ●  O  O  O
O  ●  ●  ●  ●  O  O  O  O  O  O
O  ●  ●  ●  ●  O  O  O  O  O  O
O  O  O  O  O  O  O  O  O  O  O
```

The structure element used in **bound** is:

bound(X) is:

```
O O O O O O O O O O O
O O O O O O ● ● ● ● O
O O O O O O ● O O ● O
O O O O O O ● ● ● ● O
O O O O ● ● ● ● O O O
O O O O ● O O ● O O O
O ● ● ● ● ● ● ● O O O
O ● O O ● O O O O O O
O ● ● ● ● O O O O O O
O O O O O O O O O O O
```
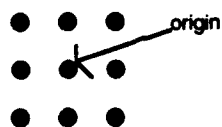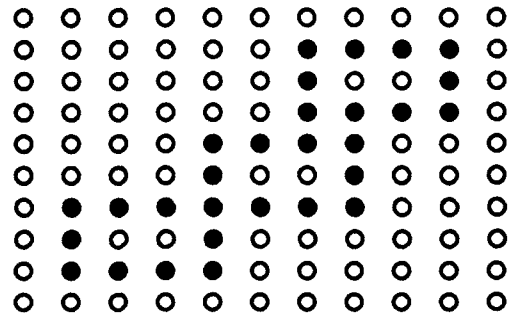
These examples can be reproduced in MATLAB using 0's for the open dots and 1's for the solid dots. Use of the *format* + command will produce the following result on the screen:

```
            + + + +
            +     +
            + + + +
        + + + +
        +       +
    + + + + + + + +
    +       +
    + + + +
```
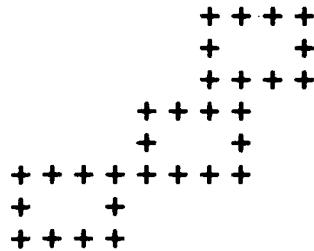
F.   HISTOGRAMS

Histograms and histogram equalization are very useful tools for image processing. They are used to enhance entire images, and to enhance specific features of images. The functions available in the Image Processing Toolbox for histogram related operations are: **equalize, histogra**, and **hisplot**.

Histogra is the most basic of the functions. It computes the histogram of an image and, if there is no output argument, it calls **hisplot** to plot the histogram. If there is an output argument, a vector of length 256 is returned containing the number of times each intensity level from intensity 0 to intensity 255 occurred in the image. By using an output

23

argument, you can reproduce the plot any number of times for comparison without recalculating the histogram.

The function **equalize,** can be used in two ways. The first use is with no lookup table specified. **equalize** calls **histogra** to compute the densities of the pixel intensities, and then uses the MATLAB function *cumsum* to equalize the densities of the pixels. It then uses these new density values to map the output pixel values. This results in a new image where all intensity values occur in approximately equal proportions. The second utilization of **equalize** is with a lookup table specified in the function call. This can be used to apply a grey scale transformation to an image.

The examples in Figure 3-5 show the effects of direct histogram equalization. All three images have a border and a label added to them using the **insert** and **label** commands, respectively. The two equalized images show the effect of the different border shade on the overall equalization process. The darker the border, the lighter the equalized image. Very dark images, with very little resolution, can often be equalized and the resultant image shows features not discernible in the original image.

24

(a)

(b)

(c)

Figure 3-5. Kathy (a) Framed, labeled, (b) Framed, labeled, histogram equalized, (c) Framed in white, labeled, histogram equalized.

G.   SUMMARY

The MATLAB Toolbox created with this thesis has a broad range of applications for image processing. The input/output functions provide flexibility in the choice of format for the image files, as well as speed. The morphological operations, when combined with the edge detectors and histogram equalization operations, enhance the user's ability to detect unseen objects in images, as well as enhance the overall picture quality. The filters provide further means for image enhancement. The next chapter is a case study involving an image that is of poor quality. The functions of the MATLAB

25

toolbox are applied to accomplish varying degrees of image enhancement, and to illuminate parts of the image that are invisible to the human eye in the original image. The case study illustrates one of many different sequences that will distort or enhance an image. Many times the sequence of steps is determined by trial and error. A basic knowledge of how the functions work, and exactly how that translates to the displayed image is the key to selecting appropriate functions. Sometimes the best method is to try several approaches and determine which results in the best image, either qualitatively or subjectively. Appendix B provides suggestions for further reading. This same list is reproduced in the MATLAB Image Processing Toolbox Documentation.

IV. A CASE STUDY

A. DISCUSSION

This chapter describes a case study that illustrates application of the toolbox functions to an image processing problem. The study begins with an image that has had its histogram altered, its edges blurred, and deteriorated by added white noise. The image, pictured below in Figure 4-1, resembles a house on a dark and snowy night.
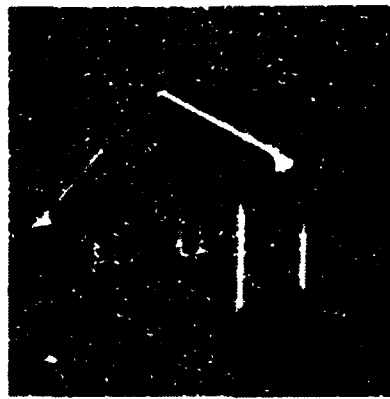

Figure 4-1. Distorted Image.

The image is obviously quite dark and has a large number of missing pixels. Some possibilities for improving it are lowpass filtering, blurring, and median filtering. Each are forms of filtering and serve to eliminage high (spatial) frequency noise and "fill in" the missing pixels. However, both the blur function and lowpass Butterworth filter function developed in this thesis smooth sharp edges (also characterized by high spatial frequencies). The blur function averages the pixel values within the mask and the result replaces the pixel value in the image. The lowpass filter multiplies the frequency response of a lowpass Butterworth filter (with cutoff frequency and filter order specified by the user) with the two-dimensional Discrete Fourier Transform of the image. The resultant

27

image is obtained by taking the inverse two-dimensional Discrete Fourier Transform. While this image will have the pixel dropouts filled in, the sharp edges will not be preserved.

One of the median filter's specific advantages is that it tends to preserve edges. The median filter takes a user-specified size mask and slides it over the entire image. Each pixel is replaced by the median value inside the mask. Thus edges tend to stay edges, and a spurious noise spike or a pixel dropout is replaced by an intensity exactly equal to at least one of the surrounding intensities. The median filter was chosen to fill in the pixel dropouts for this case study. The result is shown in Figure 4-2. Worth noting is that virtually all of the pixel dropouts have been eliminated while the edges have been preserved. When the unsharp masking is performed in the next step the latter point will be verified.
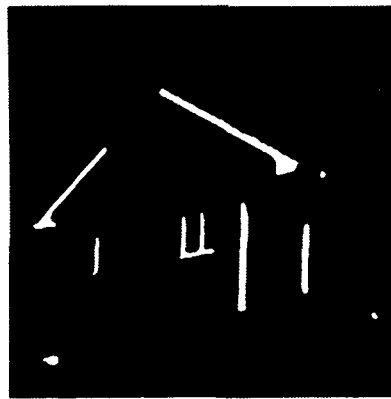

Figure 4-2. Median filtering of Figure 4-1.

This image is still very dark. Details other than the house's trim are virtually impossible to discern. The obvious first choices for the next step is to perform a histogram modification. This may not be the best choice. A not so obvious choice is to perform unsharp masking. Since the image appears so dark the possibility exists that no

more details can be shown in the image. By performing unsharp masking before histogram modification, edges that are not visible to the eye at this point can be brought out. If the edges are really not present than the process must take a different route, perhaps performing other functions such as edge detection or histogram modification before the median filtering step.

The unsharp masking subtracts a weighted low-resolution copy of the image from the image. The low-resolution copy is constructed by using the blur function with a mask size chosen by the user. The pixel intensities in the image are averaged with the surrounding intensities as discussed in the previous section. The larger the number the more blurred the resultant image appears. The unsharp masking function takes the blur function mask dimension and the weighting factor as inputs. The image in Figure 4-3 shows the result of unsharp masking the image of Figure 4-2 with the mask dimension of 5 and the weighting factor of 0.833. This weighting factor makes the starting image-to-blurred image ratio 5:1. The original pixel intensity values are all within a narrow range close to zero and the subtraction of the blurred image makes the range even smaller. The higher the intensity value in the blurred image, the more its weighted value is subtracted from to the starting image's intensity value. This results in the intensity values being grouped in a very narrow range. The last step in the unsharp masking algorithm is to scale the intensity values between 0 and 255. Since the largest intensity value before scaling is only 33 to 80 percent of its original value, the scaled image's resulting histogram is spread over a larger range of values this produces higher contrast and makes more details visible. Note that the compression of intensities must first occur before the scaling can have any effect on the contrast.

Figure 4-3. Unsharp masking of dark image.

The image above shows quite a bit of detail, however it can still be improved. By looking at the histogram of Figure 4-3 the next step in the enhancement process becomes clear. In Figure 4-4 the histogram of Figure 4-3 reveals that the intensities are all below 100 and most are between 60 and 80.
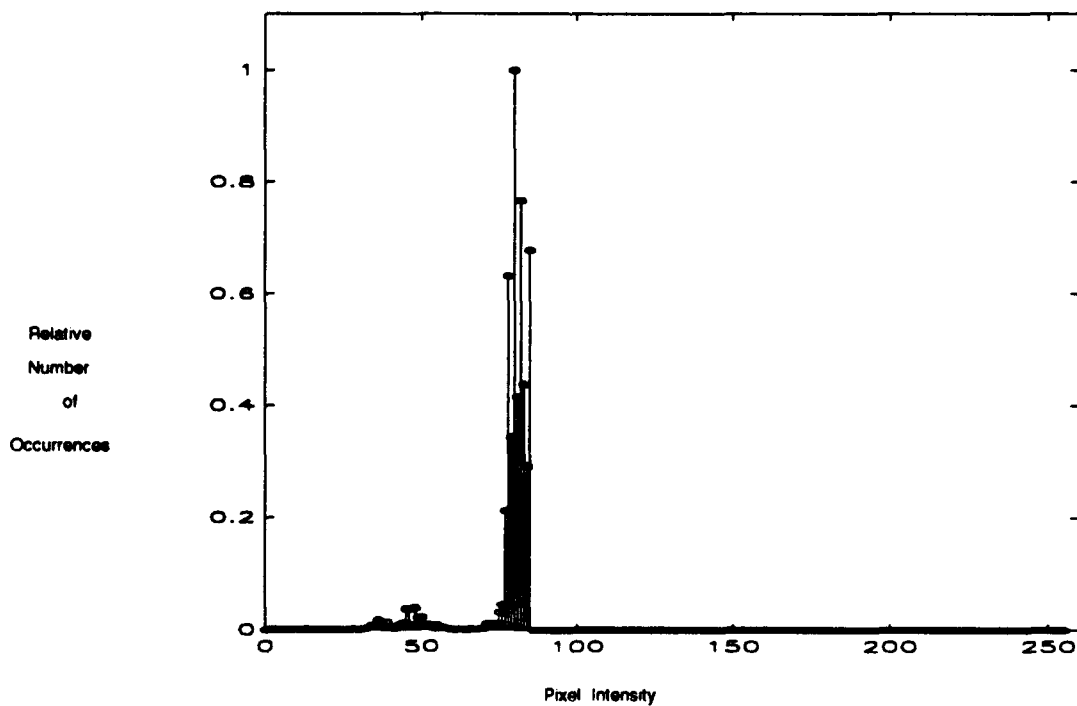

Figure 4-4. Histogram of Figure 4-3.

A modification of the histogram is necessary to improve the image. The intensities must be spread over a greater range of values to improve contrast in the image. The transformation that is necessary to achieve this effect should have a gradual slope in the lower range, with a slope increasing in the middle and gradually decreasing to zero in the upper third of the range. A piecewixe linear transformation that has these characteristics is shown in Figure 4-5.
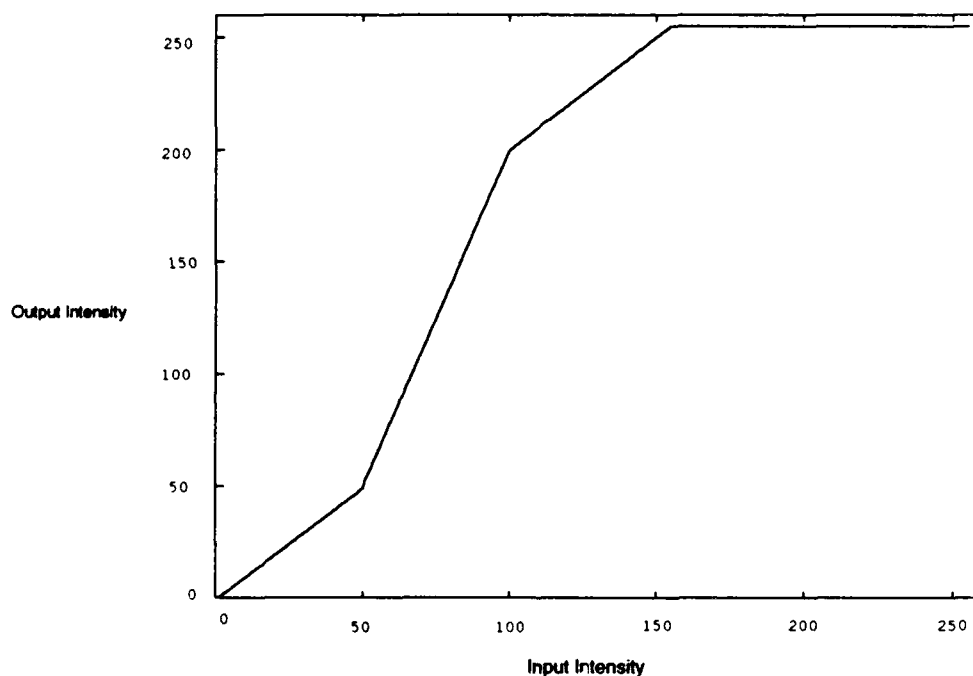

Figure 4-5. Histogram transformation function.

The image of Figure 4-3 now needs to be histogram modified by the transformation in Figure 4-5 using the equalize function. The result is shown in Figure 4-6.
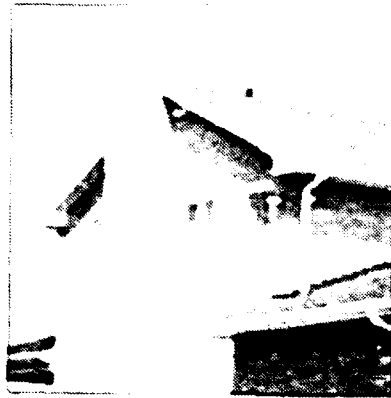
Figure 4-6. Image of Figure 4-4 after histogram transformation.

The last step in the process is to unsharp mask the image once more. This sharpens the edges and improves the contrast. One significant feature to note between the previous use of unsharp masking this case study and the use here is the difference in the contrast changes for each step. The first use of unsharp masking produced a much greater difference in the contrast because the range of intensity values prior to the unsharp masking was much narrower than in this step. (Note the non-zero values from 75 to 255. Figure 4-7 shows the histogram for the image in Figure 4-6.

Figure 4-7. Histogram of Figure 4-6.

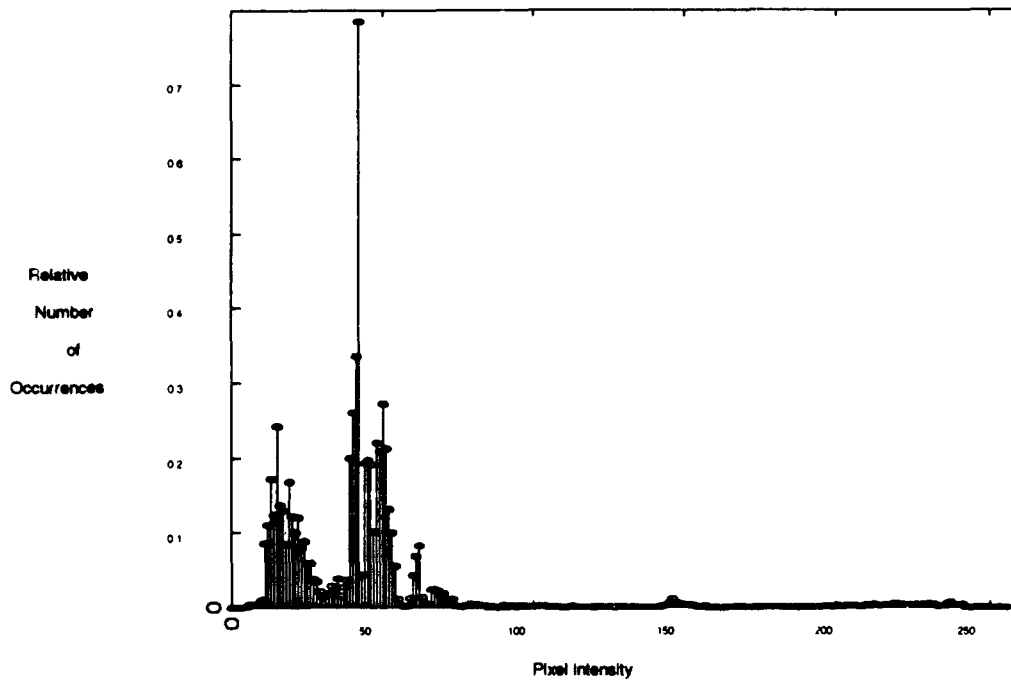In this step the parameters for the unsharp masking happen to be the same as in the previous use of the unsharp masking procedure, blur function dimension equal to 5 and weighting factor equal to 0.833. The finished product in this image enhancement process is shown in Figure 4-8.

Figure 4-8. Result of Unsharp Masking.

33

B. CONCLUSIONS

The final product is a significant improvement to the image in Figure 4-1. The type of functions and the order in which they were used in this process are by no means the only method to accomplish image enhancement. The type of image must be considered before the method of enhancement can be chosen. Some aspects to consider are whether the image has sharp edges, high or low contrast, or possibly large areas of constant intensity. The steps to consider in the enhancement process can then be addressed. For instance, if the image does not have sharp edges, unsharp masking will only serve to spread out the intensity values. If there are large areas of constant intensity then unsharp masking will have little, if any, effect on the image.

Other functions in the Image Processing Toolbox can be used to produce different effects in images. The edge detection operations can be used with the morphological functions to outline certain objects or areas in an image. The graphic and label operations can be used to put a label in an image for viewing. (See Figure 3-5) The Morphological operations can also be used to find and isolate objects within an image.

V. SUMMARY AND CONCLUSIONS

A.   REVIEW OF THESIS

This thesis provides an Image Processing Toolbox for use in MATLAB. Several functions were provided by Erkan Aykaç in his thesis, *"Enhancement of Image Processing Capabilities for Different Environments"*. [Ref. 5] The Toolbox contains input/output functions from Aykaç's thesis, improvements to several of his functions, and many new functions. It consists of 38 functions to read in, enhance, restore, transform, filter, detect edges, re-size, display, and save images. It also provides complete documentation under a separate cover which includes a tutorial section, and a distribution diskette. The documentation is in the style of that of other toolboxes available from MATLAB. The functions are user-friendly, generally fast, and encompass a wide range of possibilities within each of the categories listed above. All of the functions on the distribution diskette provide on-line information when the MATLAB *help* command is invoked. Many of the functions, which have algorithms that require the use of one or more *for-loops*, are written as MEX-files, with an M-file included for the help facility.

The Toolbox has a separate distribution for SUN SPARCstations and PC compatibles. The diskette for the PC compatibles can be used for both PC-MATLAB and 386-MATLAB. All functions that involve input/output are MEX-files which check for the array size, a requirement for PC-MATLAB. All other functions are internal to the MATLAB environment and thus need not include this feature. The MEX-files for PC-MATLAB are compiled with the *.mex* extension while the MEX-files for 386-MATLAB are compiled with the *.mx3* extension. PC-MATLAB only recognizes *.mex* when looking for MEX files and 386-MATLAB only recognizes *.mx3*, so there is no need for a separate diskette for the two PC compatible Toolboxes.

35

A detailed description for creating new MEX-files is presented in Chapter II, including information on compilation in MATLAB version 4.0 when it becomes available. Appendix A continues the description with a full program included. A tutorial is provided in Chapter III that includes examples for many of the functions. Chapter IV contains a case study in which a distorted image is restored using the Toolbox functions.

B. AREAS FOR FUTURE WORK

1. Expanding the Toolbox

The Toolbox can be expanded in several areas. One such area is in the histogram modification set of functions. Presently the transformation function must be formulated through educated guessing and trial and error. A more effective and efficient method would be to use the input histogram and a desired histogram to build a transformation function that modifies the input histogram automatically. A technique for doing this is described in Reference 6. Other areas for possible future expansion are extension of the morphological operations for use on images (vice just graphics), and addition of more types of filters, pseudo-color, and functions for image encoding. Image processing is an interesting area of study. The Toolbox provided from this thesis also makes it fun to do in MATLAB!

2. Creating an Interface Between MATLAB and the SPIDER Library

While this Toolbox contains a large and diverse set of image processing functions, the SPIDER Library in use at the Naval Postgraduate School has a far greater expanse of functions. SPIDER is not as user-friendly as MATLAB, but it may be possible to access the SPIDER Library functions from MATLAB via an interface program using the FMEX utility provided with MATLAB.

APPENDIX A

EXAMPLE MEX-FILE

The source code for the *for-loop* in the **equalize** function is provided in this appendix to illustrate the use of the three functions *recalloc, redimension,* and *rematlab* in a MEX-file, and show a short but complete program. This program replaces a double *for-loop* in the M-file function equalize. The M-file with *for-loops* implemented in MATLAB on the SUN SPARC workstation takes approximately four minutes for an image of 256×256 pixels. Implementing it with this MEX-file the M-file takes less than 3 seconds! The program was compiled with the Metaware High C compiler for the 386-MATLAB Toolbox, and with Borland Turbo C++ for the PC-MATLAB Toolbox. For the UNIX version, the function prototypes cannot be included because the compiler for the UNIX system at Naval Postgraduate School uses traditional C and will not accept function prototypes during compilation. The function parameter declarations must be the classic style for the Metaware High C compiler; for the UNIX they can be either classic or modern form (ANSI C standard).

equal.c Program

```
/*This routine is for MATLAB version 3.5
   equal routine called by the equalize.m routine to execute the loop */


#include  <math.h>
#include  "cmex.h"
#define  A_IN  prhs[0]      /* Gives a label to the first input argument */
#define  S_OUT  plhs[0]     /* Gives a label to the output argument */
#define  B_IN  prhs[1]/* Gives a label to the second input argument */


user_fcn(nlhs,plhs,nrhs,prhs)  /* Used in place of main() */


int  nlhs,nrhs;
Matrix  *prhs[],*plhs[];        /* Sets type Matrix for the variable pointers */
{
  int  i,j,am,an;
  short int  **immage,**filt_im;
  double  *filtered_im,*image,*lut;
              /* Check for correct number  of input,output arguments */
  if(nrhs !=2) mex_error("must be two input arguments");
  if(nlhs !=1) mex_error("must be one output argument");
  lut = B_IN -> pr;     /* Sets pointer to real part of the array and calls it lut */
  image = A_IN -> pr;  /* Sets pointer to real part of the input image and calls it image */
  an = A_IN -> n;      /* Gets the column dimension of the input array */
  am = A_IN -> m;      /* Gets the row dimension of the input array */
  S_OUT = create_matrix(am,an,REAL); /* Creates space, sets pointer for output */
  filtered_im = S_OUT -> pr;
  filt_im=(short int**)recalloc(am,an);  /* creates 1-D arrays of short ints */
  immage=(short int**)recalloc(am,an);
  redimension(filt_im,image,am,an);   /* Recasts 1-D array to 2-D arrays, (faster access)
*/
  for (i=1;i<am-1;++i)
    {
    for (j=1;j<an-1;++j)
        filt_im[i][j]=lut[filt_im[i][j]];  /* performs function--histogram transformation */
    }                                       /* (the whole purpose for this program) */
  rematlab(filtered_im,filt_im,am,an);     /* recasts result to 1-D array to return to */
}                                           /* MATLAB */
```

```c
/****************************************************************/
recalloc(m,n)
int   m,n;
{
short int  **array;
int   i;
  array=(short int**) mex_calloc(m,sizeof(short int*));
  for (i=0;i<m;++i)
    array[i] = (short int*) mex_calloc(n,sizeof(short int));
  return array;
}
/****************************************************************/
redimension(array_out,array_in,m,n)
short int   **array_out;
int           m,n;
double        *array_in;
{
int   i,j;
  for (i=0;i<n;++i)
    for (j=0;j<m;++j)
      array_out[j][i]=array_in[i*m+j];
}

/****************************************************************/
rematlab(array_out,array_in,m,n)
short int   **array_in;
int           m,n;
double        *array_out;
{
int   i,j;
  for (i=0;i<n;++i)
  {
    for (j=0;j<m;++j)
      array_out[i*m+j]=(double)array_in[j][i];
  }
}
```

APPENDIX B

SUGGESTED READING

Charles R. Giardina and Edward R. Dougherty, *Morphological Methods in Image and Signal Processing*, Prentice-Hall, 1987.

Rafael C. Gonzalez and Paul Wintz, *Digital Image Processing*, Addison Wesley, 1987.

Anil K. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall, 1986.

Jae S. Lim, *Two-Dimensional Signal and Image Processing*, Prentice-Hall, 1990.

W.K. Pratt, *Digital Image Processing, Second Edition* John Wiley and Sons, Inc., 1991.

MATLAB User's Guide, The MathWorks, Inc., 1985-1991.

REFERENCES

1. Gonzalez, R.C. and Wintz, P., *Digital Image Processing*, Addison Wesley, 1987, Chapter 4.

2. Jain, A.K., *Fundamentals of Digital Image Processing*, Prentice-Hall, 1989.

3. Pratt, W.K., *Digital Image Processing, Second Edition*, John Wiley and Sons, Inc., 1991.

4. *MATLAB User's Guide, Beta 3*, The MathWorks, Inc., 1992.

5. Aykaç, E., *"Enhancement of Image Processing Capabilities for Different Environments"*, Master's Thesis, Naval Postgraduate School, Monterey, California, June, 1991.

6. Lim, J.S., *Two-Dimensional Signal and Image Processing*, Prentice-Hall, 1990, pp. 455-459.

7. *MATLAB User's Guide for MS-DOS Personal Computers*, The MathWorks, Inc., 1990.

8. *386-MATLAB for 80386-based MS-DOS Personal Computers User's Guide*, The MathWorks, Inc., October 15, 1990.

9. *PRO-MATLAB for Sun Workstations User's Guide*, The MathWorks, Inc., January 31, 1990.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center   2
    Cameron Station
    Alexandria, Virginia 22304-6145

2. Library, Code 52   2
    Naval Postgraduate School
    Monterey, California 93943-5002

3. Chairman, Code EC   1
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California 93943-5000

4. Prof. C. Therrien, Code EC/Ti   5
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California 93943-5000

5. Prof. R. Cristi, Code EC/Cx   1
    Department of Electrical and Computer Engineering
    Naval Postgraduate School
    Monterey, California 93943-5000

6. LCDR Dorothy J. Freer   3
    NAPRA
    PSC 477 Box 35
    FPO AP 96306

7. Dr. R. Madan   1
    ONR Code 1114
    Office of Naval Research
    Arlington, Virginia 22217-5000