Sample Project Solutions

for

Digital Image Processing Using MATLAB®
2nd edition

Rafael C. Gonzalez
Richard E. Woods
Steven L. Eddins

©2009
Gatesmark Publishing
a division of Gatesmark, LLC
ISBN: 9780982085400

Book web site: www.imageprocessingplace.com

Version 1.0

May 1, 2009

Introduction

This document contains solutions to all the sample projects listed in the book web site. These projects are intended to serve as guidelines for formulating projects dealing with other topics in the book.

The projects range in complexity from straightforward extensions of the material in the book to more comprehensive undertakings that require several hours to solve. All the material required for the projects is contained in the book web site. Normally, the solutions are based on material that has been covered up to the chapter in question, but we often use material from later chapters in order to arrive at a well-formulated solution. In these instances, we indicate where the needed material is located in the book and, on rare occasions, we point to online resources.

One of the most interesting aspects of a course in digital image processing in an academic environment is the pictorial nature of the subject. It has been our experience that students truly enjoy and benefit from judicious use of computer projects to complement the material covered in class. Since computer projects are in addition to course work and homework assignments, we try to keep the formal project reporting as brief as possible. In order to facilitate grading, we try to achieve uniformity in the way project reports are prepared. A useful report format is as follows:

Page 1: Cover page.

- Project title
- Project number
- Course number
- Student's name
- Date due
- Date handed in

Abstract (not to exceed ½ page).

Page 2: One to two pages (max) of technical discussion.

Page 3 (or 4): Discussion of results. One to two pages (max).
Results: Image results (printed typically on a laser or inkjet printer). All images must contain a number and title referred to in the discussion of results.

Appendix: Program listings, focused on any original code prepared by the student. For brevity, functions and routines provided to the student are referred to by name, but the code is not included.

Layout: The entire report must be on a standard sheet size (e.g., 8.5 by 11 inches), stapled with three or more staples on the left margin to form a booklet, or bound using a clear cover, standard binding product.

Although formal project reporting of the nature just discussed is not typical in an independent study or industrial/research environment, the projects outline in this document offer valuable extensions to the material in the book, where lengthy examples are kept to a minimum for the sake of space and continuity in the discussion. Image processing is a highly experimental field and MATLAB, along with the Image Processing

Toolbox, offers an unparalleled software development environment. By taking advantage of MATLAB's vectorization capabilities solutions can be made to run fast and efficiently, a feature that is especially important when working with large image data bases. The use of DIPUM Toolbox functions is encouraged in order to save time in arriving at project solutions and also as exercises for becoming more familiar with these functions.

The reader will find the projects in this document to be useful for both learning purposes and as a reference source for ideas on how to attack problems of practical interest. As a rule, we have made an effort to match the complexity of projects to the material that has been covered up to the chapter in which the project is assigned. When one or more functions that have not been covered are important in arriving at a well-formulated solution, we generally suggest in the problem statement that the reader should become familiar with the needed function(s).

As will become evident in the solution material that follows, there usually isn't a single, "best" solution to a given project. The following criteria can be used as a guide when a project calls for new code to be developed:

How quickly can I implement this solution?
How robust is the solution for "rare" cases?
How fast does the solution run?
How much memory does the solution require?
How easy is the code to read, understand, and modify?

We offer multiple solutions to some projects to illustrate these points (e.g., see the solutions to Project 2.1). Also, it is important to keep in mind that solutions can change over time, as new MATLAB or Image Processing Toolbox functions become available.

Chapter 2
Sample Project Solutions

PROJECT 2.1

MATLAB does not have a function to determine which elements of an array are integers (i.e., . . ., 2, 1, 0, 1, 2 , . . .). Write a function for this purpose, with the following specifications:

```
function I = isinteger(A)
%ISINTEGER Determines which elements of an array are integers.
%   I = ISINTEGER(A) returns a logical array, I, of the same size
%   as A, with 1s (TRUE) in the locations corresponding to integers
%   (i.e., . . . -2 -1 0 1 2 . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.
```

Use of `while` or `for` loops is not allowed. Note: Integer and double-precision arrays with real or complex values are numeric, while strings, cell arrays, and structure arrays are not. *Hints:* Become familiar with function `floor`. If you include the capability to handle complex numbers, become familiar with functions `real` and `imag`.

SOLUTIONS

Three solutions are provided. Solution 1 is a quick, easy-to-understand implementation that is acceptable for "most" purposes. It isn't perfect, however. For example, it returns true (1) for `Inf`, `-Inf`, and for complex numbers whose real and imaginary parts are integers, such as `1 + i`. Solution 2 addresses these problems. It has the advantage of being more robust, but it has the disadvantage of taking more time to run because of the extra checks. Also, Solution 2 always converts `A` to double, which is unnecessary work for the integer-class arrays. Solution 3 adds additional logic to avoid the double conversion when it is not needed. It is therefore faster than the other solutions if the input has integer class. The disadvantage of Solution 3 is that the code is longer and more difficult to understand, especially compared with Solution 1.

SOLUTION 1

```
function I = isinteger1(A)
%ISINTEGER1 Determines which elements of an array are integers.
%   I = ISINTEGER1(A) returns a logical array, I, of the same size
%   as A, with 1s (TRUE) in the locations corresponding to integers
%   (i.e., . . . -2, -1, 0, 1, 2, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:19:03 $

% Check the validity of A.
if ~isnumeric(A)
    error('A must be a numeric array.');
end
```

```
A = double(A);
I = A == floor(A);
```

SOLUTION 2

```
function I = isinteger2(A)
%ISINTEGER2 Determines which elements of an array are integers.
%   I = ISINTEGER2(A) returns a logical array, I, of the same size
%   as A, with 1s (TRUE) in the locations corresponding to integers
%   (i.e., . . . -2, -1, 0, 1, 2, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:19:03 $

% Check the validity of A.
if ~isnumeric(A)
    error('A must be a numeric array.');
end

A = double(A);
I = isfinite(A) & (imag(A) == 0) & (A == floor(A));
```

SOLUTION 3

```
function I = isinteger3(A)
%ISINTEGER3 Determines which elements of an array are integers.
%   I = ISINTEGER3(A) returns a logical array, I, of the same size
%   as A, with 1s (TRUE) in the locations corresponding to integers
%   (i.e., . . . -2, -1, 0, 1, 2, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:19:04 $

% Check the validity of A.
if ~isnumeric(A)
    error('A must be a numeric array.');
end

if isa(A, 'double')
   I = isfinite(A) & (imag(A) == 0) & (A == floor(A));

elseif isa(A, 'single')
   A = double(A);
   I = isfinite(A) & (imag(A) == 0) & (A == floor(A));

else
   % A must be one of the integer types, so we don't have to convert
   % to double.
   if isreal(A)
      I = true(size(A));
   else
      I = imag(A) == 0;
   end
end
```

****

PROJECT 2.2

MATLAB does not have a function to determine which elements of an array are even numbers (i.e., . . . 4, 2, 0, 2, 4 , . . .). Write a function for this purpose, with the following specifications:

```
function E = iseven(A)
%ISEVEN Determines which elements of an array are even numbers.
%   E = ISEVEN(A) returns a logical array, E, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to even numbers
%   (i.e., . . . -3, -1, 0, 2, 4, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.
```

Use of `while` or `for` loops is not allowed. See Project 2.1 regarding numeric arrays. *Hint*: Become familiar with function `floor`.

SOLUTION

```
function E = iseven(A)
%ISEVEN Determines which elements of an array are even numbers.
%   E = ISEVEN(A) returns a logical array, E, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to even numbers
%   (i.e., . . . -4, -2, 0, 2, 4, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.4 $  $Date: 2009/03/15 00:28:22 $

% Check the validity of A.
if ~isnumeric(A)
    error('A must be a numeric array.');
end

A = double(A);
E = floor(A/2) == (A/2);
```

****

PROJECT 2.3

MATLAB does not have a function to determine which elements of an array are odd numbers (i.e., . . ., 3, 1, 1, 3 , . . .). Write a function for this purpose, with the following specifications:

```
function D = isodd(A) function D = isodd(A)
%ISODD Determines which elements of an array are odd numbers.
%   E = ISODD(A) returns a logical array, D, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to odd numbers
%   (i.e., . . . -3, -1, 1, 3, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.
```

Use of `while` or `for` loops is not allowed. See Project 2.1 regarding numeric arrays. *Hint*: Become familiar with function `floor`.

SOLUTION

```
function D = isodd(A)
%ISODD Determines which elements of an array are odd numbers.
%   E = ISODD(A) returns a logical array, D, of the same size as A,
%   with 1s (TRUE) in the locations corresponding to odd numbers
%   (i.e., . . . -3, -1, 1, 3, . . . )in A, and 0s (FALSE) elsewhere.
%   A must be a numeric array.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.3 $  $Date: 2009/03/15 00:28:22 $

% Check the validity of A.
if ~isnumeric(A)
```

```
    error('A must be a numeric array.');
end

A = double(A);
D = floor((A + 1)/2) == ((A + 1)/2);
```

****

PROJECT 2.4

Write an M-function with the following specifications:

```
function H = imcircle(R, M, N)
%IMCIRCLE Generates a circle inside a rectangle.
%   H = IMCIRCLE(R, M, N) generates a circle of radius R centered
%   on a rectangle of height M and width N. H is a binary image with
%   1s on the circle and 0s elsewhere. R must be an integer >= 1.
```

Your program must check the validity of `R` and also it should check to make sure that the specified circle fits in the given rectangle dimensions. Use of `for` or `while` loops is not permitted. *Hint:* Review function `meshgrid` and become familiar with function `floor`.

SOLUTION

```
function H = imcircle(R, M, N)
%IMCIRCLE Generates a circle inside a rectangle.
%   H = IMCIRCLE(R, M, N) generates a circle of radius R centered
%   on a rectangle of height M and width N. H is a binary image with
%   1s on the circle and 0s elsewhere. R must be an integer >= 1.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.

%   $Revision: 1.2 $  $Date: 2009/01/16 17:34:43 $

% The values of the coordinates of the rectangle are x = 0, 1 , 2,
% . . ., M - 1 and y = 0, 1, 2, . . ., N - 1, so the center is at
% coordinates (floor(M/2), floor(N/2)). Check to make sure that the
% circle will fit in the rectangle.
X0 = floor(M/2);
Y0 = floor(N/2);
if R > M - X0 - 1 | R > N - Y0 - 1
    error('Circle does not fit in rectangle.')
end

if (floor(R) ~= R) | (R < 1)
    error('The radius R must be an integer >= 1.')
end

% Compute a matrix, A, of distances from any point in the rectangle to its
% center.
x = 0:M - 1;
y = 0:N - 1;
[Y, X] = meshgrid(y, x);
A = sqrt((X - X0).^2 + (Y - Y0).^2);

% Find all the distances in the range (R - L/2) >= A(I,J) <= (R + L/2), with
% L = sqrt(2).  These are the points comprising the circumference of the
% circle within a tolerance of sqrt(2). This tolerance is chosen based on
% the fact that the coordinates are separated by 1 in the horizontal
% and vertical directions and by sqrt(2) in the diagonal direction.
H = ((R - sqrt(2)/2) <= A) & (A <= (R + sqrt(2)/2));
```

****

PROJECT 2.5

The main purposes of this project are to learn how to work with displaying and changing directories and how to use that information to read an image from a specified directory. Write an M-function with the following specifications:

```
function [I, map] = imagein(path)
%IMAGEIN Read image in from current-working or specified directory.
%   I = IMAGEIN displays a window containing all the files in the
%   current directory, and saves in I the image selected from the
%   current directory.
%   [I, MAP] = IMAGEIN variable MAP is required to be an output
%   argument when the image being read is an indexed image.
%   [ . . .] = IMAGEIN('PATH') is used when the image to be read
%   resides in a specified directory. For example, the input
%   argument 'C:\MY_WORK\MY_IMAGES' opens a window showing
%   the contents of directory MY_IMAGES. An image selected from
%   that directory is read in as image I.
```

Hint: Use online help to become familiar with functions `cd`, `pwd`, and `uigetfile`. For an alternative solution, consider using function `fullfile` instead of function `cd`.

SOLUTION 1

```
function [I, map] = imagein1(path)
%IMAGEIN1 Read image in from current-working or specified directory.
%   I = IMAGEIN1 displays a window containing all the files in the
%   current directory, and saves in I the image selected from the
%   current directory.
%   [I, MAP] = IMAGEIN1 variable MAP is required to be an output
%   argument when the image being read is an indexed image.
%   [ . . .] = IMAGEIN1('PATH') is used when the image to be read
%   resides in a specified directory. For example, the input
%   argument 'C:\MY_WORK\MY_IMAGES' opens a window showing
%   the contents of directory MY_IMAGES. An image selected from
%   that directory is read in as image I.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:20:44 $

if nargin < 1
   path = pwd;
end

original_directory = pwd; % Save the current working directory.
cd(path); % Change current directory to specified directory.

% Read image from current directory.
[file, pathname] = uigetfile('*.*', 'Image Open');
if isequal(file, 0) | isequal(pathname, 0)
   disp('Image input canceled.');
   I = [];
   map = [];
else
   [I, map] = imread(file);
end

% Change back to the original working directory.
cd(original_directory);
```

SOLUTION 2

```
function [I, map] = imagein2(path)
%IMAGEIN2 Read image in from current-working or specified directory.
%   I = IMAGEIN2 displays a window containing all the files in the
%   current directory, and saves in I the image selected from the
```

```
%   current directory.
%   [I, MAP] = IMAGEIN2 variable MAP is required to be an output
%   argument when the image being read is an indexed image.
%   [ . . .] = IMAGEIN2('PATH') is used when the image to be read
%   resides in a specified directory. For example, the input
%   argument 'C:\MY_WORK\MY_IMAGES' opens a window showing
%   the contents of directory MY_IMAGES. An image selected from
%   that directory is read in as image I.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:20:44 $

if nargin < 1
   path = pwd;
end

% Read image from specified path.
[file, pathname] = uigetfile(fullfile(path, '*.*'), 'Image Open');
if isequal(file, 0) | isequal(pathname, 0)
   disp('Image input canceled.');
   I = [];
   map = [];
else
   [I, map] = imread(fullfile(pathname, file));
end
```

****

Chapter 3
Sample Project Solutions

PROJECT 3.1 Intensity Transformation (Mapping) Function

Write an M-function for performing general intensity transformations of gray-scale images. The specifications for the function are as follows:

```
function z = intxform (s, map)
%INTXFORM Intensity transformation.
%   Z = INTXFORM(S, MAP) maps the intensities of input
%   image S using mapping function, MAP, whose values are assumed to
%   be in the range [0 1].  MAP specifies an intensity transformation
%   function as described in Section 3.2. For example, to create a map
%   function that squares the input intensities of an input image of
%   class uint8 and then use function INTXFORM to perform the mapping
%   we write:
%
%       t = linspace(0, 1, 256);
%       map = t.^2;
%       z = intxform(s, map);
%
%   The output image is of the same class as the input.
```

Hint: Your code will be simplified if you use functions `tofloat` and `interp1`.

SOLUTION

```
function z = intxform (s, map)
%INTXFORM Intensity transformation.
%   Z = INTXFORM(S, MAP) maps the intensities of input image S using
%   mapping function, MAP, whose values are assumed to be in the range
%   [0 1].  MAP specifies an intensity transformation function as
%   described in Section 3.2. For example, to create a map function that
%   squares the input intensities of an input image of class uint8 and
%   then use function INTXFORM to perform the mapping we write:
%
%       t = linspace(0, 1, 256);
%       map = t.^2;
%       z = intxform(s, map);
%
%   The output image is of the same class as the input.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.3 $  $Date: 2009/03/15 00:28:22 $

classin = class(s); % For use later.
[s, revertclass] = tofloat(s);

% Construct x and y vectors, representing the intensity
% transformation, to be used with INTERP1.  Make sure they are both
% column vectors.
x = linspace(0, 1, numel(map))';
y = map(:);
```

```
% Apply the intensity transformation using INTERP1.
z = interp1(x, y, s, 'linear');

% Change to class of the input.
z = revertclass(z);
```

****

PROJECT 3.2 Histogram Processing

Toolbox function `histeq` attempts to produce as flat a histogram. Write a histogram equalization function that implements the equations discussed in Section 3.3.2 directly:

$$\begin{aligned} s_k &= T(r_k) \\ &= \sum_{j=1}^{k} p_r(r_j) \\ &= \sum_{j=1}^{k} \frac{n_j}{n} \end{aligned}$$

The function for this project has the following specifications:

```
function h = histeq2(f)
%HISTEQ2 Histogram equalization.
%   H = HISTEQ2(F) histogram-equalizes F and outputs the result
%   in H. Unlike Toolbox function histeq, HISTEQ2 implements the direct,
%   "standard" histogram equalization approach explained in Section
%   3.3.2. F can be of class uint8, uint16, or double. If F is double,
%   then its values are assumed to be in the range [0 1]. The intensity
%   range of the input image (regardless of class) is divided into 256
%   increments for computing the histogram and the corresponding cumulative
%   distribution function (CDF). Recall that the CDF is actually the mapping
%   function used for histogram equalization.
%
%   The output is of the same class as the input.
```

Hint: Your code will be simplified if you use the function developed in Project 3.**1** and function `cumsum`.

SOLUTION

```
function h = histeq2(f)
%HISTEQ2 Histogram equalization.
%   H = HISTEQ2(F) histogram-equalizes F and outputs the result
%   in H. Unlike Toolbox function histeq, HISTEQ2 implements the direct,
%   "standard" histogram equalization approach explained in Section
%   3.3.2. F can be of class uint8, uint16, or double. If F is double,
%   then its values are assumed to be in the range [0 1]. The intensity
%   range of the input image (regardless of class) is divided into 256
%   increments for computing the histogram and the corresponding cumulative
%   distribution function (CDF). Recall that the CDF is actually the mapping
%   function used for histogram equalization.
%
%   The output is of the same class as the input.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.3 $  $Date: 2009/03/15 00:28:22 $

[f, revertclass] = im2uint8(tofloat(f)); %Work with 8-bit accuracy.

% Compute the histogram and the cumulative distribution function.
```

```
den = numel(f);
hst = imhist(f, 256) / den;
cdf = cumsum(hst);  % Values are in the range [0 1].

% Now map the values of the input image to obtain the histogram-
% equalized image, g.
h = intxform(f, cdf);

% Convert to the class of the input
h = revertclass(h);
```

****

PROJECT 3.3 Local Histogram Equalization

The global histogram equalization technique is easily adaptable to local histogram equalization. The procedure is to define a square or rectangular window (neighborhood) and move the center of the window from pixel to pixel. At each location, the histogram of the points inside the window is computed and a histogram equalization transformation function is obtained. This function is finally used to map the intensity level of the pixel centered in the neighborhood to create a corresponding (processed) pixel in the output image. The center of the neighborhood region is then moved to an adjacent pixel location and the procedure is repeated. Since only one new row or column of the neighborhood changes during a pixel-to-pixel translation of the region, updating the histogram obtained in the previous location with the new data introduced at each motion step is possible. This approach has obvious advantages over repeatedly computing the histogram over all pixels in the neighborhood region each time the region is moved one pixel location.

Write an M-function for performing local histogram equalization. Your function should have the following specifications.

```
function g = localhisteq(f, m, n)
%LOCALHISTEQ Local histogram equalization.
%   G = LOCALHISTEQ(F, M, N) performs local histogram equalization
%   on input image F using a window of (odd) size M-by-N to produce
%   the processed image, G. To handle border effects, image F is
%   extended by using the symmetric option in function padarray.
%   The amount of extension is determined by the dimensions of the
%   local window. If M and N are omitted, they default to
%   3. If N is omitted, it defaults to M. Both must be odd.
%
%   This function accepts input images of class uint8, uint16, or
%   double. However, all computations are done using 8-bit intensity
%   values to speed-up computations. If F is of class double its
%   values should be in the range [0 1]. The class of the output
%   image is the same as the class of the input.
```

Hint: One approach is to write a function that uses the results of projects 3.1 and 3.2. This will result in a simpler solution. However, the function will be on the order of five times slower than a function that computes the local histogram once and then updates it as the local window moves throughout the image, one pixel displacement at a time. The idea is that the new histogram, *h_new*, is equal to the to the old histogram, *h_old*, plus the histogram of data added as the window is moved, *h_datain*, minus the histogram of the data that moved out of the window as a result of moving the window, *h_dataout*. That is: *h_new = h_old + h_datain h_dataout*. In addition to being faster important advantage of this implementation is that it is self-contained. Note that the histograms *h_datain* and *h_dataout* are normalized by the factor equal to the total number of pixels, *mn*, in order to be consistent with the fact that the area of histograms *h_new* and *h_old* must be 1.0. Your code will be simplified if you use functions `cumsum` and `tofloat`. Keep in mind that only the intensity level of the center of the neighborhood needs to be mapped at each location of the window.

SOLUTION 1. Using a simpler (but slower running) approach.

```
function g = localhisteq1(f, m, n)
```

```
%LOCALHISTEQ1 Local histogram equalization.
%   G = LOCALHISTEQ1(F, M, N) performs local histogram equalization
%   on input image F using a window of (odd) size M-by-N to produce
%   the processed image, G. To handle border effects, image F is
%   extended by using the symmetric option in function padarray.
%   The amount of extension is determined by the dimensions of the
%   local window. If M and N are omitted, they default to
%   3. If N is omitted, it defaults to M. Both must be odd.
%
%   This function accepts input images of class uint8, uint16, or
%   double. However, all computations are done using 8-bit intensity
%   values to speed-up computations. If F is of class double its
%   values should be in the range [0 1]. The class of the output
%   image is the same as the class of the input.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:17:24 $

% This implementation is the most straightforward of the solutions,
% but it runs about 2 times slower than a similar implementation that
% maps only the center point of each neighborhood (localhisteq2), and
% about 6 times slower than the implementation in localhisteq3. An external,
% external, general-purpose histogram equalization function and mapping
% function are required.

% Preliminaries.
if nargin == 1
   m = 3; n = 3
elseif nargin == 2
   n = m;
end
if ~(round(m/2) ~= m/2) | ~(round(n/2) ~= n/2)
   error('The dimensions of the neighborhood must be odd')
end

[f, revertclass] = im2uint8(tofloat(f)); %Work with 8-bit accuracy.

% Extend the borders of the input image.
[R C] = size(f); % For use later.
f = padarray(f, [m n], 'symmetric', 'both');

% Move the window from pixel to pixel and histogram equalize at
% each location.
xg = 0;
den = m*n;
for x = (m + 1)/2 + 1 : R + (m + 1)/2
   xg = xg + 1;
   yg = 0;
   for y = (n + 1)/2 + 1 : C + (n + 1)/2
      yg = yg + 1;
      subimage = f(x:x + m - 1, y:y + n - 1);
      h = histeq2(subimage); % Can also use imhist, but watch the scaling.
      % Pick the value in the center of the equalized subimage. Function
      % histeq contains the necessary mapping function.
      g(xg, yg) = h((m + 1)/2, (n + 1)/2);
   end
end

% Convert back to the class of the input.
g = revertclass(g);
```

SOLUTION 2. An improvement over SOLUTION 1

```
function g = localhisteq2(f, m, n)
%LOCALHISTEQ2 Local histogram equalization.
%   G = LOCALHISTEQ2(F, M, N) performs local histogram equalization
%   on input image F using a window of (odd) size M-by-N to produce
%   the processed image, G. To handle border effects, image F is
```

```
%   extended by using the symmetric option in function padarray.
%   The amount of extension is determined by the dimensions of the
%   local window. If M and N are omitted, they default to
%   3. If N is omitted, it defaults to M. Both must be odd.
%
%   This function accepts input images of class uint8, uint16, or
%   double. However, all computations are done using 8-bit intensity
%   values to speed-up computations. If F is of class double its
%   values should be in the range [0 1]. The class of the output
%   image is the same as the class of the input.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:17:24 $

% This implementation is a variation of localhisteq1. However, here we use
% function intxform to apply the CDF transformation function to only the
% center pixel of the subimage, instead of to all the pixels in the subimage,
% as in localhisteq1. This runs about twice as fast.

% Preliminaries.
if nargin == 1
   m = 3; n = 3
elseif nargin == 2
   n = m;
end
if ~(round(m/2) ~= m/2) | ~(round(n/2) ~= n/2)
   error('The dimensions of the neighborhood must be odd')
end

[f, revertclass] = im2uint8(tofloat(f)); %Work with 8-bit accuracy.

% Extend the borders of the input image.
[R C] = size(f); % For use later.
f = padarray(f, [m n], 'symmetric', 'both');

% Move the window from pixel to pixel and histogram equalize at
% each location.
xg = 0;
den = m*n;
for x = (m + 1)/2 + 1 : R + (m + 1)/2
   xg = xg + 1;
   yg = 0;
   for y = (n + 1)/2 + 1 : C + (n + 1)/2
      yg = yg + 1;
      subimage = f(x:x + m - 1, y:y + n - 1);
      den = numel(subimage);
      hst = imhist(subimage, 256) / den;
      cdf = cumsum(hst);
      g(xg, yg) = intxform(subimage((m + 1)/2, (n + 1)/2), cdf);
   end
end

% Convert back to the class of the input.
g = revertclass(g);
```

SOLUTION 3 A faster running, but more complex implementation.

```
function g = localhisteq3(f, m, n)
%LOCALHISTEQ3 Local histogram equalization.
%   G = LOCALHISTEQ3(F, M, N) performs local histogram equalization
%   on input image F using a window of (odd) size M-by-N to produce
%   the processed image, G. To handle border effects, image F is
%   extended by using the symmetric option in function padarray.
%   The amount of extension is determined by the dimensions of the
%   local window. If M and N are omitted, they default to
%   3. If N is omitted, it defaults to M. Both must be odd.
%
```

```
%   This function accepts input images of class uint8, uint16, or
%   double. However, all computations are done using 8-bit intensity
%   values to speed-up computations. If F is of class double its
%   values should be in the range [0 1]. The class of the output
%   image is the same as the class of the input.

%   Copyright 2009 R. C. Gonzalez, R. E. Woods, & S. L. Eddins
%   For use with Digital Image Processing Using MATLAB, 2nd ed.
%   Gatesmark Publishing, 2009.
%   $Revision: 1.1 $  $Date: 2009/03/15 00:17:24 $

% This implementation is about 6 times faster than the implementation
% in localhisteq1. However, this implementation is more complex because
% it is based on computing the histogram of an m-by-n region only once.
% After that, the histogram is updated as the region is moved one pixel
% location down and then it is moved across the entire image, one pixel at
% a time. The idea is that the new histogram, hnew, is equal to the
% old histogram, hold, plus the histogram of data added as the
% window is moved, hdatain, minus the histogram of the data that
% moved out of the window as a result of moving the window, hdataout.
% That is: hnew = hold + hdatain - hdataout. Another important
% advantage of this implementation is that it is self-contained,
% while localhisteq1 and localhisteq2 are not.

% Preliminaries.
[f, revertclass] = im2uint8(tofloat(f)); %Work with 8-bit accuracy.
if nargin == 1
   m = 3; n = 3;
elseif nargin == 2
   n = m;
end
if ~(round(m/2) ~= m/2) | ~(round(n/2) ~= n/2)
   error('The dimensions of the neighborhood must be odd')
end

den = m*n;
v = 0:255;

% Extend the borders of the input image.
[R C] = size(f);
f = padarray(f, [m n], 'symmetric', 'both');

% Starting coordinates of top, left of local window.
x0 = (m + 1)/2 + 1;
y0 = (m + 1)/2 + 1;

% Initial subimage and histogram, displaced by one pixel
% vertically and horizontally. This is to obtain initial
% values of histogram.
subimage = f(x0 - 1:x0 + m - 2, y0 - 1: y0 + n - 2);
histim = hist(double(subimage(:)), v)/den;

xg = 0;
for x = x0 : R + (m + 1)/2
   %Update histogram of new subimage.
   histabove = hist(double(f(x - 1, y0 - 1: y0 + n - 2)), v)/den;
   histbelow = hist(double(f(x + m - 1, y0 - 1: y0 + n - 2)), v)/den;
   histim = histim - histabove + histbelow;
   histimhoriz = histim;
   xg = xg + 1;
   yg = 0;
   %Now update histograms of subimages as they move to the right.
   for y = y0 : C + (n + 1)/2
      yg = yg + 1;
      histleft = hist(double(f(x:x + m - 1, y - 1)), v)/den;
      histright = hist(double(f(x:x + m - 1, y + n - 1)), v)/den;
      histimhoriz = histimhoriz - histleft + histright;
      cdf=cumsum(histimhoriz);
      %Intensity value of center of subimage:
      value= f(x + (m + 1)/2 - 1, y + (n + 1)/2 - 1);
      %Use value as an index into the transformation function, cdf.
      %Have to add 1 to value because indexing is 1 to 256, but
```

```
      %intensity values are 0 to 255.
      g(xg, yg) = cdf(double(value) + 1); %The +1 needed for value = 0.
   end
end

g = revertclass(g);
```

* *

PROJECT 3.4 Comparing Global and Local Histogram Equalization.

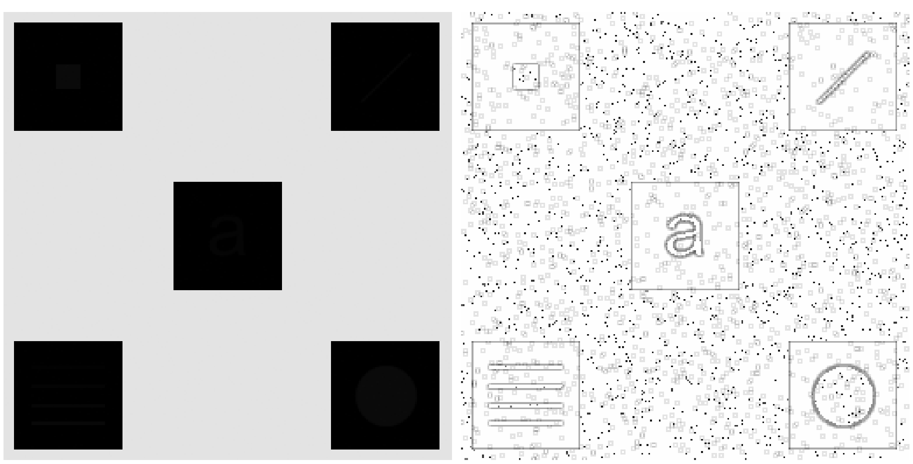
(a) Download image `FigP0304(embedded_objects_noisy).tif` and process it with function `localhisteq` using neighborhoods of sizes 3 x 3 and 7 x 7. Explain the differences in your results.

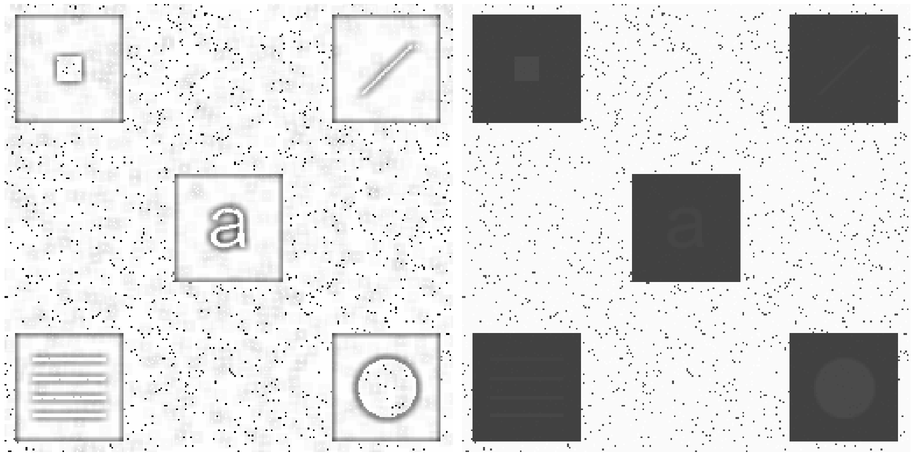
(b) Histogram-equalize it using the global function `histeq2` from Project 3.2. If you did not do that project, then use Toolbox function `histeq`. Although the background tonality will be different between the two functions, the net result is that global histogram equalization will be ineffective in bringing out the details embedded in the black squares within the image. Explain why. Note in both (a) and (b) how noise is enhanced by both methods of histogram equalization, with local enhancement producing noisier results than global enhancement.

SOLUTION

(a) The image in Fig. P3.4(a) is the original image. Figure P3.4(b) is the result of local histogram equalized using a 3 x 3 window. Figure P3.4(c) is the local histogram equalized result using a 7 x 7 window. The main difference between the two locally processed images is that, as the size of the local window increases, so does distortion (seen as a blurring or halo effect) caused by the intensity mappings being affected by the histogram of a larger area.

(b) The image in Fig. P3.4(d) is the result of global histogram equalized using function `histeq2`. Global equalization could not bring out the details because they had little effect on the shape of the global transformation function (i.e., the cumulative distribution function).





a b
c d

FIGURE P3.4 (a) Original image. (b) Image processed by local histogram equalization with a window of size 3   3. (c) Local histogram equalized image obtained using a window of size 7   7. (d) Result of performing global histogram equalization in the original image.

* *

PROJECT 3.5 Experimenting with Larger "Laplacian" Masks

It is shown in Example 3.10 that the Laplacian mask w8 = [1, 1, 1; 1 – 8 1; 1, 1, 1] yields a result sharper than the result with a similar mask with a 4 in the center.

(a) Write an M-function that generates a "Laplacian" mask of arbitrary odd size. For example, the mask of size 5   5 would consist of all 1s with a –24 in the center location.

(b) Download the image `FigP0305(blurry_moon).tif` and compare the results between Fig. 3.18(c) and the results obtained with masks of size *n* x *n* for *n* = 5, 9, 15, and 25. **(c)** Explain the differences in the resulting images.

SOLUTION

(a) The program is as follows:

```
function w = genlaplacian(n)
%GENLAPLACIAN Generates a Laplacina mask of (odd) size n-by-n.
%   W = GENLAPLACIAN(N) Generates a Laplacian mask, W, of size
%   N-by-N, where N is an odd, positive integer.

% Check validy of the input. (See Chapt 2 projects for functions
% isinteger and iseven
if ~isinteger(n) | n <= 0 | iseven(n)
   error('n must be a positive, odd integer')
end

% Generate the mask.
center = (n^2) - 1;
w = ones(n);
w((n + 1)/2, (n + 1)/2) = -center;
```

(b) Figure P3.5(a) shows the original image, and Fig. P3.5(b) shows the image obtained with the commands

```
>> fd = im2double(imread('FigP3.5(a)(blurry_moon).tif'));
>> g3 = fd - imfilter(fd, genlaplacian(3), 'replicate');
```

Similarly, the images in the middle and last rows were obtained using the commands

```
>> g5 = fd - imfilter(fd, genlaplacian(5), 'replicate');
>> g9 = fd - imfilter(fd, genlaplacian(9), 'replicate');
>> g15 = fd - imfilter(fd, genlaplacian(15), 'replicate');
>> g25 = fd - imfilter(fd, genlaplacian(25), 'replicate');
```

(c) The result in Fig. P3.5(b) is the same as Fig. 3.18(c). As the rest of the images show, increasing the size of the mask produced progressively thicker edge features. The reason for this can be explained easily by considering an image in which all pixels, except the pixel in the center, are black (0). Suppose that the center pixel is white (1). Thus, the image contains a single impulse in the center. We know that convolution of a function with an impulse copies the function at the location of the impulse. Thus, if we convolve the standard 3 x 3 Laplacian we would get an image with a 8 in the in the center, with all its 8-neighbors being 1. The rest of the pixels are 0. If, instead, we use a 5 x 5 mask, the result will be a 5 x 5 area of 1s, except the center point, which would have a value of 24. The point is that as the mask gets larger, larger an larger areas around the center point will be nonzero. If we now consider the image as a superposition of impulses, each impulse being multiplied by the intensity of the image at the location of the impulse, it becomes clear that larger masks will produce results that at each point will be influenced by size of the mask. The larger the mask, the “thicker” the features will be.

* *

PROJECT 3.6 Unsharp masking in the spatial domain

Briefly, unsharp masking consists of the following steps: (1) blur the original image; (2) subtract the blurred image from the original (the resulting difference image is called the "mask"); and (3) add the mask (or a fraction of the mask) to the original. In other words:

$$g(x,y) = f(x,y) - f_{\text{blurred}}(x,y)$$

This is the mask. Then,

$$f_{\text{unsharp}} = f + g(x,y)$$

It is not difficult to show that

$$f_{\text{blurred}}(x,y) = f(x,y) - f_{\text{sharp}}(x,y)$$

so that

$$f_{\text{unsharp}}(x,y) = f(x,y) + f_{\text{sharp}}(x,y)$$

If we use the Laplacian to produce the sharp image, then we can implement the preceding equation by convolving *f* with a 3   3 mask with a 1 in the center and 0s elsewhere; convolve *f* with a Laplacian mask; and add the results if the center coefficient of the Laplacian mask is positive, or subtract the results if we use a Laplacian mask whose center is negative. In either case, we can combine both operations by convolving *f* with the single mask:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Function `fspecial` generates the preceding unsharp spatial filter when you choose ALPHA = 0. (Note: The range of ALPHA mentioned in page 100 of the first edition of the book mentions that alpha has to be greater than 0. The latest release of the Image Processing Toolbox has eliminates that restriction.)

A slight generalization of the unsharp expression is as follows:

$$f_{\text{unsharp}}(x, y) = f(x, y) + k * f_{\text{sharp}}(x, y)$$

where k is a scaling factor.

(a) Use function `fspecial` to generate the unsharp spatial filter discussed above.

(b) Apply your unsharp filter to image `FigP0305(blurry_moon).tif`.

SOLUTION

(a)

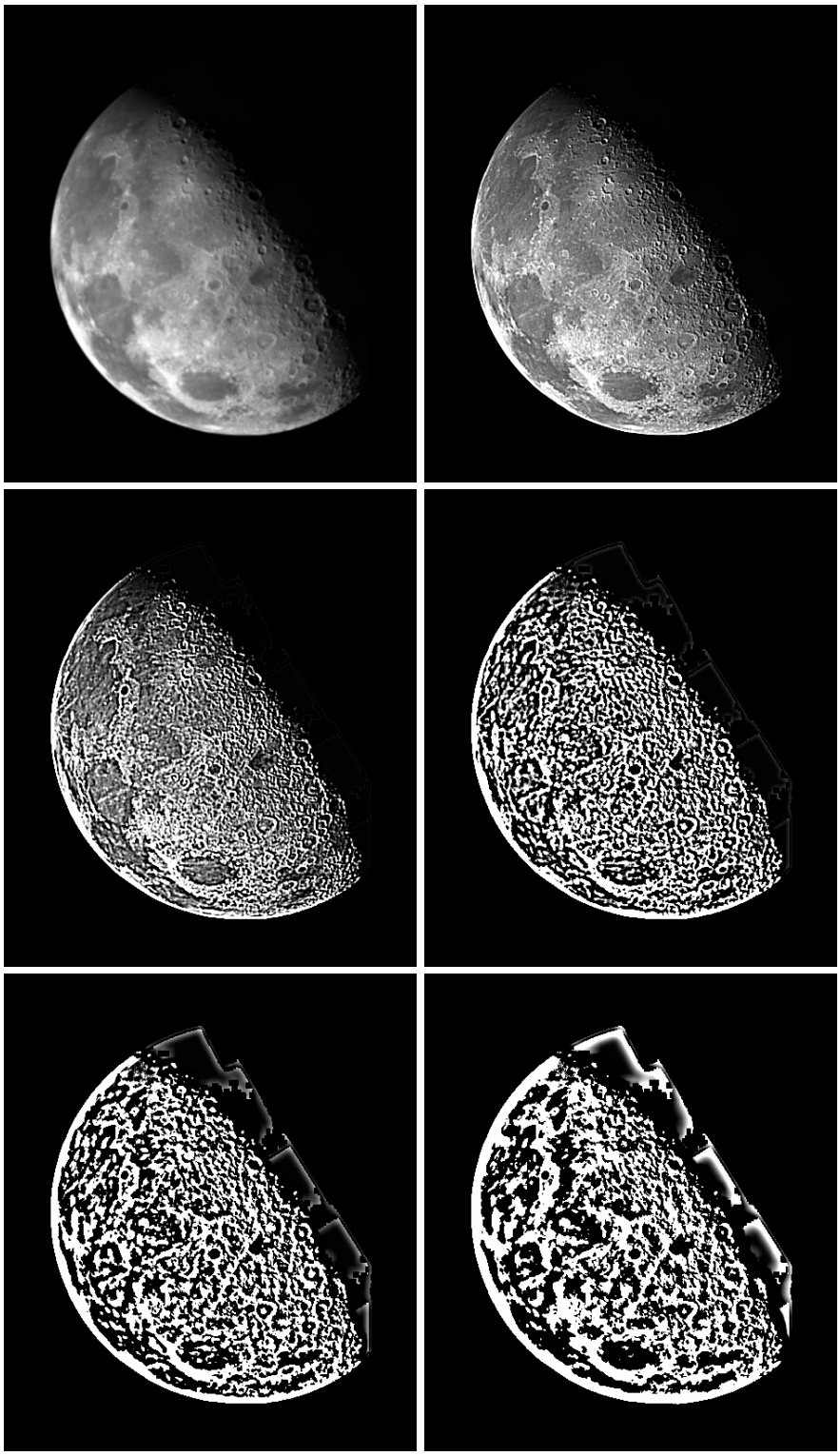
```
>> w = fspecial('unsharp', 0);
```

(b)

```
>> f = imread('FigP0305(blurry_moon).tif');
>> imshow(f) % Fig. P3.6(a)
>> g = imfilter(f, w, 'replicate');
>> figure, imshow(g) % Fig. P3.6(b).
```

As expected, the result of unsharp masking is sharper than the original image.

* *



a b
c d
e f

FIGURE P3.5. (a) Original. (b)-(f) Images processed with Laplacian masks of sizes 3 3, 5 5, 9 9, 15 15, and 25 25. (Original image courtesy of NASA.)

a b

FIGURE P3.6 (a) Original image. (b) Result of spatial unsharp masking.

Chapter 4
Sample Project Solutions

PROJECT 4.1 Working with the Power Spectrum and Using Special Variables `varargout` and `varargin`

The main objectives of this project are to investigate the effect that filtering has on average image power and to practice using functions `varargout` and `varargin`.
As discussed in Section 4.1, the power spectrum is defined as

$$P(u,v) = |F(u,v)|^2$$

where $F(u,v)$ is the Fourier transform of an image, $f(x,y)$. The *average image power* is defined as

$$f_A = \frac{1}{MN}\sum_u\sum_v |F(u,v)|^2$$

(a) Write an M-function with the following specifications:

```
function [varargout] = impower(f, varargin)
%IMPOWER Filters an image and computes its average power.
%   fA = impower(f) Computes the average image power, fA.  [fA, fR,
%   g] = impower(f, TYPE, SIG) computes the average image power, fA,
%   of input image f.  It also filters f using a Gaussian highpass
%   filter if TYPE = 'high' or a Gaussian lowpass filter if TYPE =
%   'low'.  Either filter has sigma = SIG. The resulting filtered
%   image and its power are output as g and fR.
```

As discussed in Section 3.2.3, `varargout` is one way to write a function whose number of outputs is variable. Although in this case we could just as easily have used the syntax `function [fA, fR, g]` to handle both instances of outputs, the purpose of using `varargout` here is simply as an exercise. In general, `varargout` is quite useful when the number of outputs is large and variable. Similar comments apply for `varargin` regarding input variables.

(b) Download `FigP4.1(a)(test_pattern).tif`. Obtain its average image power, `fA`, using function `impower`.

(c) Then, use `SIG = 16` and generate `fRL` and `gL` for a lowpass filter and `fRH` and `gH` for a highpass filter. Show the original image and the two filtered images.

(d) You will find that `fRL` and `fA` are close in value, but `fRH` is approximately one order of magnitude smaller than these two quantities. Explain the main reason for the difference.

SOLUTION

(a) The program is as follows. Keep in mind that the special variables `varargout` and `varargin` are cell arrays.

```
function [varargout] = impower(f, varargin)
%IMPOWER Filters an image and computes its average power.
%   fA = impower(f) Computes the average image power, fA.  [fA, fR,
%   g] = impower(f, TYPE, SIG) computes the average image power, fA,
%   of input image f.  It also filters f using a Gaussian highpass
%   filter if TYPE = 'high' or a Gaussian lowpass filter if TYPE =
%   'low'.  Either filter has sigma = SIG. The resulting filtered
%   image and its power are output as g and fR.

% Obtain average image power, fA.
[M N] = size(f);
F = fft2(f);
F = abs(F).^2;
varargout{1} = sum(F(:))/M*N; % This is fA.

if nargin == 1
   return
end

PQ = paddedsize(size(f));

if strcmp(lower(varargin{1}), 'low')
   %Note that varargin{1} is the SECOND input because f is
   %listed in the argument as being a separate, first input.
   w = lpfilter('gaussian', PQ(1), PQ(2), varargin{2});
elseif strcmp(lower(varargin{1}), 'high')
   w = hpfilter('gaussian', PQ(1), PQ(2), varargin{2});
else
   error('Unknown filter type.')
end

% Filter the image.
varargout{3} = dftfilt(f, w); % This is g.

% Obtain the filtered image power, fR:
G = fft2(varargout{3});
G = abs(G).^2;
varargout{2} = sum(G(:))/M*N; % This is fR.
```

(b) Read image and obtain `fA`:

```
>> f = imread('FigP0401(test_pattern).tif');
>> fA = impower(f)

fA =

  2.9995e+015
>> imshow(f) % Fig. P4.1(a).
```

(c) Compute `fRL`, `gL`, `fRH`, and `gH`:

```
>> [fA, fRL, gL] = impower(f, 'low', 16);
>> fRL

fRL =

  2.5959e+015

>> figure, imshow(gL, []) % Fig. P4.1(b).
>> [fA, fRH, gH] = impower(f, 'high', 16);
>> fRH

fRH =

  2.1202e+014

>> figure, imshow(gH, []) % Fig. P4.1(c).
```

(d) Typically, the largest term in $F(u,v)$ is the DC term [i.e., $F(0,0)$]. Highpass-filtering an image attenuates this term to 0, thus reducing the value of the average image power.

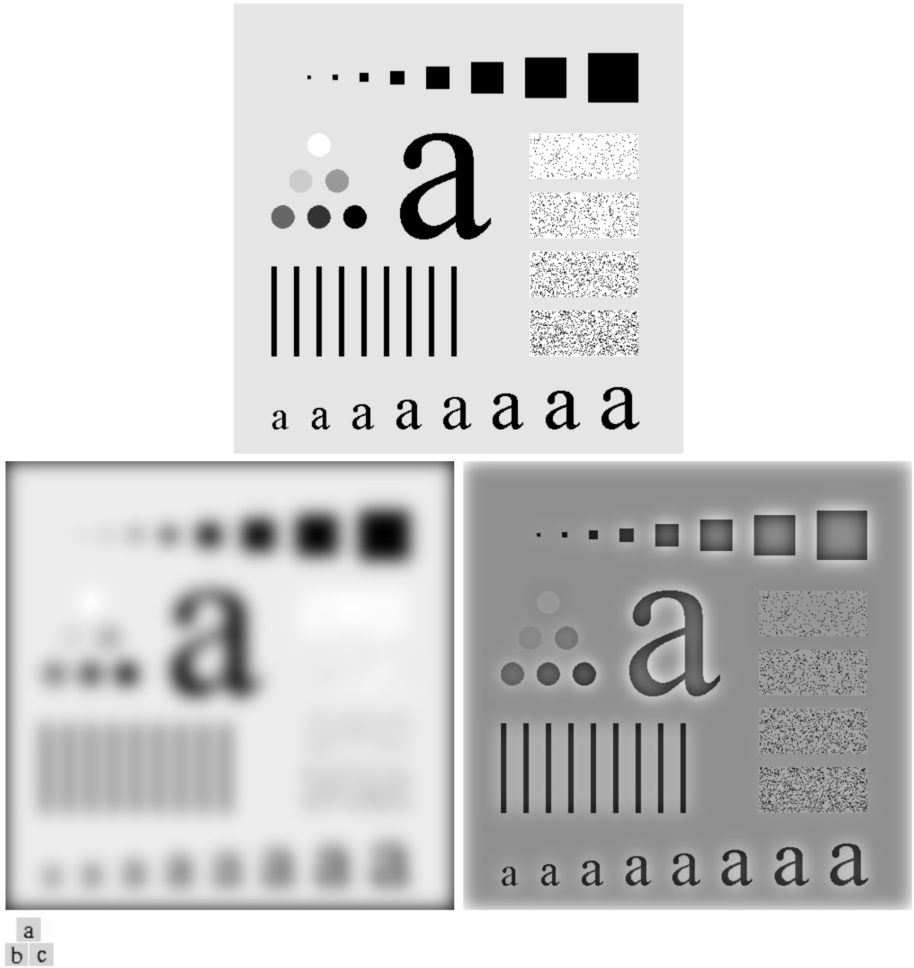


FIGURE P4.1. (a) Original. (b) Lowpass filtered image. (c) Highpass filtered image.

* *

PROJECT 4.2 Working with Phase Angles

As indicated in Section 4.1 of the book, the Fourier transform is complex, so it can be expressed in polar form as

$$F(u,v) = |F(u,v)| e^{j\phi(u,v)}$$

(Note that the exponent is positive, not negative as it is shown in the initial printing of the book.) Here,

$$|F(u,v)| = \left[R^2(u,v) + I^2(u,v) \right]^{1/2}$$

is the spectrum and

$$\phi(u,v) = \tan^{-1}\left[\frac{I(u,v)}{R(u,v)} \right]$$

is the phase angle. The purposes of this projects are (1) to show that the phase angle carries information about the location of image elements, (2) that the spectrum carries information regarding contrast and intensity

transitions, and (3) that phase information is dominant over the spectrum regarding visual appearance of an image. Download `FigP0402(a)(woman).tif` and `FigP0402(b)(test_pattern).tif` (call them `f` and `g`). It is recommended that you look up functions `complex` and `angle` before starting the project.

(a) Compute the spectrum and the phase angle of image `f` and show the phase image. Then compute the inverse transform using only the phase term (i.e., `ifft2` of $e^{j\phi(u,v)}$] and show the result. Note how the image is void of contrast, but clearly shows the woman's face and other features.

(b) Compute the inverse using only the magnitude term (i.e., `ifft2` of $|F(u,v)|$). Display the result and note how little structural information it contains.

(c) Compute the spectrum of `g`. Then compute the inverse FFT using the spectrum of `g` for the real part and the phase of `f` for the imaginary part. Display the result and note how the details from the phase component dominate the image.

SOLUTION

(a) Read image `f` and compute the spectrum and phase angle:

```
>> f = imread('FigP0402(a)(woman).tif');
>> imshow(f) % FigP4.2-1(a).
>> F = fft2(f);
>> Sf = abs(F);
>> Af = angle(F);
>> figure, imshow(Af, []) % FigP4.2-2(a).
```

Next, compute the inverse using only the phase term of the transform:

```
>> E = complex(0, Af);
>> fphase = real(ifft2(exp(E)));
>> figure, imshow(fphase, []); % FigP4.2-2(b).
```

Although the features of this image are unmistakable from the original, all gray tonality was lost. This information is carried by the spectrum, as shown next.

(b) Compute the inverse using only the spectrum:

```
>> fspec = real(ifft2(Sf));
>> figure, imshow(log(1 + abs(fspec)), []) % FigP4.2-2(c).
```

As discussed in Section 3.2.3, the range of values in the spectrum typically needs to be scaled to get a meaningful display. An image based on these values also has a large dynamic range, thus the use of the log in the preceding command. Note that mainly gray tonality is present in the image obtained from the inverse of the spectrum, thus confirming the comments made earlier. A few features (such as a faint representation of the bright earrings) are discernible in the image, but the predominant nature of the image is gray content. Based on the results in (a) and (b), we conclude that an image recovered using only the magnitude of the Fourier transform is unrecognizable and has severe dynamic range problems. The image recovered using only the phase faintly is severely degraded in quality, showing only faint outlines of the principal features in the original image.

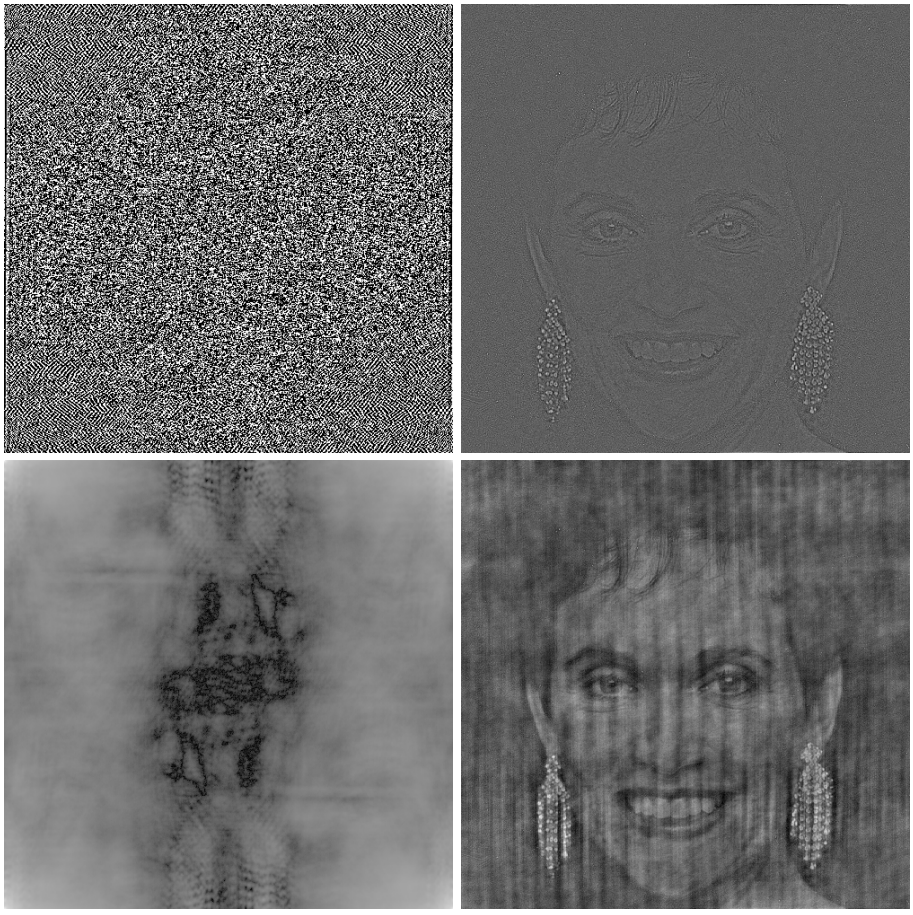
(c) Construct the Fourier transform using the spectrum of `g` and the phase from `f`, and compute the inverse:

```
>> g = imread('FigP0402(b)(test_pattern).tif');
>> figure, imshow(g) % FigP4.2-1(b).
>> Sg = abs(fft2(g));
>> FT = Sg.*exp(E);
>> fg = real(ifft2(FT));
>> figure, imshow(fg, []) % FigP4.2-2(d).
```

Note how the woman's face dominates the result, indicating that phase information is dominant in determining the visual appearance of an image.

a b

FIGURE P4.2-1. Original images.



a b
c d

FIGURE P4.2-2. (a) Phase image of Fig. P4.2-1(a). (b) Image reconstructed using only the phase angle. (c) Image reconstructed using only the spectrum. (d) Image obtained by computing the inverse FFT with the real part being the spectrum of Fig. P4.2-1(b) and the phase angle from Fig. P4.2-1(a).

* *

PROJECT 4.3 Fun Project: Distorting Signs and Symmetries in the Fourier Transform

Interesting and sometimes nonsensical results are obtained when the symmetry and signs in the Fourier transform are not preserved, as the following project shows.

Read image `FigP0402(a)(woman).tif`, and obtain its spectrum, `S`, and phase, `P`, as in Project 4.2. Then,

(a) Let `S1 = S` and `S1(1:M/2, 1:N/2) = 0` where `[M, N] = size(S)`. Recover the image using `S1` and `P`. Show your result.

(b) Let `P2 = conj(P)`, where `conj` is the complex conjugate. Recover the image using `S` and `P2`. Show your result.

(c) In Chapter 4, all filters used are zero-phase-shift filters, which have no effect on the phase because they multiply the real and imaginary components of the transform by the same quantity. Here, we want to show the effects of changing the real and imaginary components of the Fourier transform differently. Let `F2 = complex(0.25*real(F), imag(F))`. Note that this affects both the spectrum and the phase. However, based on the results from Project 4.2, we expect that the effect on the phase will dominate in the recovered image. Obtain the inverse transform and show the resulting image.

SOLUTION

```
>> f = imread('FigP0402(a)woman).tif');
>> imshow(f); % Fig. P4.3(a).
>> F = fft2(f);
>> S = abs(F);
>> P = angle(F);
>> [M N] = size(S);
```

(a)

```
>> S1 = S;
>> S1(1:M/2, 1:N/2) = 0;
>> C = complex(0, P);
>> g1 = real(ifft2(S1.*exp(C)));
>> figure, imshow(g1, []); % Fig. P4.3(b).
```

(b)

```
>> C2 = complex(0,-P);
>> g2 = real(ifft2(S.*exp(C2)));
>> figure, imshow(g2, []); % Fig. P4.3(c).
```

(c)

```
>> F2 = complex(0.25*real(F), imag(F));
>> g3 = real(ifft2(F2));
>> figure, imshow(g3, []); % Fig. 4.3(d).
```

a b
c d

FIGURE P4.3 (a) Original. (b) Recovered after one-quarter of the spectrum was set to 0. (c) Recovered after setting the phase to its complex conjugate. (d) Recovered after the real part of the Fourier transform was multiplied by 0.25. The imaginary part was not changed.

* *

PROJECT 4.4 Bandreject and Bandpass Filtering

In the book we discuss lowpass and highpass filtering in detail. Although used less often, bandreject and bandpass filters are an important category of frequency domain filters, especially because they are much more difficult to implement using spatial techniques. The transfer function of a Butterworth bandreject filter (BBRF) of order *n* is

$$H_{br}(u,v) = \frac{1}{1 + \left[\dfrac{D(u,v)W}{D^2(u,v) - D_0^2}\right]^{2n}}$$

where W is the width of the band, and D_0 is the radius to the center of the band. The corresponding bandpass filter is given by

$$H_{bp}(u,v) = 1 - H_{br}(u,v)\,.$$

(a) Write a function with the following specifications to implement these filters:

```
function H = bandfilter(type, M, N, RADII, WIDTH, ORDER)
```

```
%BANDFILTER Generate a bandreject or bandpass filter.
%   H = BANDFILTER(TYPE, M, N, RADII, WIDTH, ORDER) generates
%   a Butterworth band filter, H, of size M-by-N with a total of
%   K bands. H is a bandpass or bandreject filter, depending on
%   whether TYPE is specified as 'pass' or as 'reject'. RADDII
%   is a vector of length K containing the radius (to the center of
%   of the band) for each band filter specified. WIDTH is vector
%   of length K containing the corresponding width for each
%   specified band. If WIDTH is a scalar, then the same width
%   is used for each band. ORDER is a vector of length K containing
%   the order for the Butterworth filter corresponding to each
%   band. If ORDER is a scalar, the same order is applied to
%   all bands.
%
%   The equation implemented for the reject filter is the
%   Butterworth expression
%
%                            1
%           Hr = ----------------------------
%                            D(u,v)W
%                1 + [ --------------- ]^2n
%                       D(u,v)^2 - D0^2
%
%   The bandpass filter is simply Hbp = 1 - Hr.
```

(b) Generate and show 3-D perspective plots of the two filters for `M = N = 600`, `RADII = [50 200]`, `W = [5 5]`, and `n = 20`. Don't forget to use function `fftshift` to center the filter and thus produce nicer looking plots.

(c) Write the plots to disk in `tif` format using 300 DPI resolution.

SOLUTION

(a) The M-function follows.

```
function H = bandfilter(type, M, N, RADII, WIDTH, ORDER)
%BANDFILTER Generate a bandreject or bandpass filter.
%   H = BANDFILTER(TYPE, M, N, RADII, WIDTH, ORDER) generates
%   a Butterworth band filter, H, of size M-by-N with a total of
%   K bands. H is a bandpass or bandreject filter, depending on
%   whether TYPE is specified as 'pass' or as 'reject'. RADDII
%   is a vector of length K containing the radius (to the center of
%   of the band) for each band filter specified. WIDTH is vector
%   of length K containing the corresponding width for each
%   specified band. If WIDTH is a scalar, then the same width
%   is used for each band. ORDER is a vector of length K containing
%   the order for the Butterworth filter corresponding to each
%   band. If ORDER is a scalar, the same order is applied to
%   all bands.
%
%   The equation implemented for the reject filter is the
%   Butterworth expression
%
%                            1
%           Hr = ----------------------------
%                            D(u,v)W
%                1 + [ --------------- ]^2n
%                       D(u,v)^2 - D0^2
%
%   The bandpass filter is simply Hbp = 1 - Hr.

% Preliminaries
H = zeros(M, N);
K = length(RADII); % Number of bands.
if length(WIDTH) == 1
   WIDTH = WIDTH*ones(K, 1);
end
if length(ORDER) == 1
```

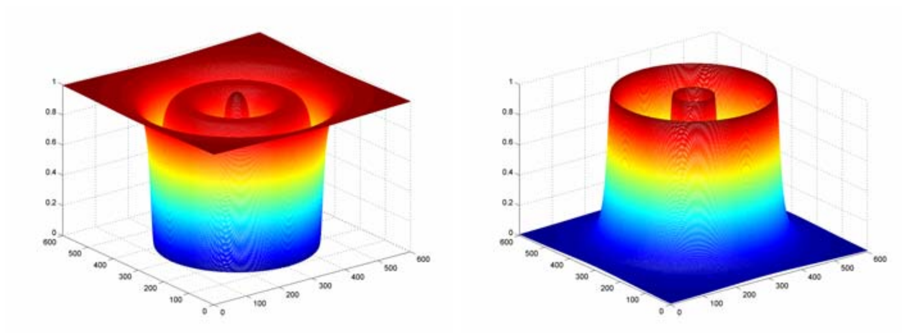
```
   ORDER = ORDER*ones(K, 1);
end

% Generate meshgrid array.
[U, V] = dftuv(M, N);

% Compute square distances array.
D = sqrt(U.^2 + V.^2);

% Generate filter.
K = length(RADII); % Number of bands.
H = zeros(M, N);
for I = 1:K
   W = WIDTH(I);
   D0 = RADII(I);
   n = ORDER(I);
   DEN = (W*D./(D.^2 - D0^2 + eps)).^2*n;
   % Generate as bandpass to avoid adding the limiting value of 1
   % in the bandreject filters.
   H = H + (1 - 1./(1 + DEN));
end

% Determine filter type. If type is not 'reject', assume that % filter is of the
pass type.
if strcmp(type, 'reject')
   H = 1 - H; % Bandreject filter.
end

% Scale filter amplitude to the range [0 1].
H = H - min(H(:));
H = H./max(H(:));
```

(b)

```
>> Hr = bandfilter('reject', 600, 600, [50 200], [5 5], 20);
>> Hp = bandfilter('pass', 600, 600, [50 200], [5 5], 20);
>> figure, mesh(fftshift(Hr)) % Fig. P4.4(a).
>> figure, mesh(fftshift(Hp)) % Fig. P4.4(b).
```

(c)

```
>> print -f1 -dtiff -r300 FigP4.4(a)(bandreject_filter).tif
>> print -f2 -dtiff -r300 FigP4.4(b)(bandpass_filter).tif
```



a b

FIGURE P4.4 (a) Bandreject filter. (b) Bandpass filter.

* *

PROJECT 4.5 Using Bandreject Filtering for Image Denoising

Image `FigP0405(HeadCT_corrupted.tif` is a tomography image of a human head, heavily corrupted by sinusoidal noise in at least two directions.

(a) Clean up the image using bandreject filtering. To simplify your project you may ignore padding in this case. *Hint:* Since the noise is sinusoidal, it will show in the spectrum as impulses. Display the spectrum as a guide to where to set up the band of your filter. If you use function `pixval` to determine the coordinates of the impulses interactively, keep in mind that this function lists the column coordinates (`v`) first and the row coordinates (`u`) second.

(b) Use bandpass filtering to extract the noise pattern.

NOTE: When attempting to remove sinusoidal spatial interference, it generally is advisable to do all DFT filtering without padding. Even if the pattern is a pure sine wave, padding introduces frequencies in the sine wave that often are quite visible and difficult to remove in the filtered image. Thus, in this case, the effect of wraparound error introduced by not using padding usually is negligible by comparison.

SOLUTION

(a) Read the image and compute its spectrum.

```
>> f = imread('FigP0405(HeadCT_corrupted).tif');
>> imshow(f)  % Fig4.5(a).
>> [M, N] = size(f);
>> F = fft2(f);
>> figure, imshow(fftshift(log(1 + abs(F))), []) % Fig. P4.5(b).
```

The impulses are evident in the image (they were enlarged to make them easier to see). To determine their distance from the origin we use function `impixelinfo`:

```
>> impixelinfo
```

The distances are approximately 10, 20, and 56.6 units. We use this number to set the radius of the band. To allow for some variation, we set the width to 5. The filter should be reasonably sharp—Here we used `n = 1`.

```
>> Hr = bandfilter('reject', M, N, [10 20 56.6], 3, 2);
>> g = dftfilt(f, Hr);
>> figure, imshow(g, []); % Fig. P4.5(c).
```

(b)

```
>> Hp = 1 - Hr;
>> gn = dftfilt(f, Hp);
>> figure, imshow(gn, []) % Fig. P4.5(d).
```

Note in Fig. P4.5(c) that the strong noise components were eliminated, but some distortion is evident as a result of having removed from the transform image data not associated with the noise impulses. We solve the same problem in Project 5.4 using notch filtering and show that better results can be obtained by removing only the data localized around the impulses. The noise pattern contains residual parts from the image itself because data not associated with the noise patterns were passed by the bandpass filter.

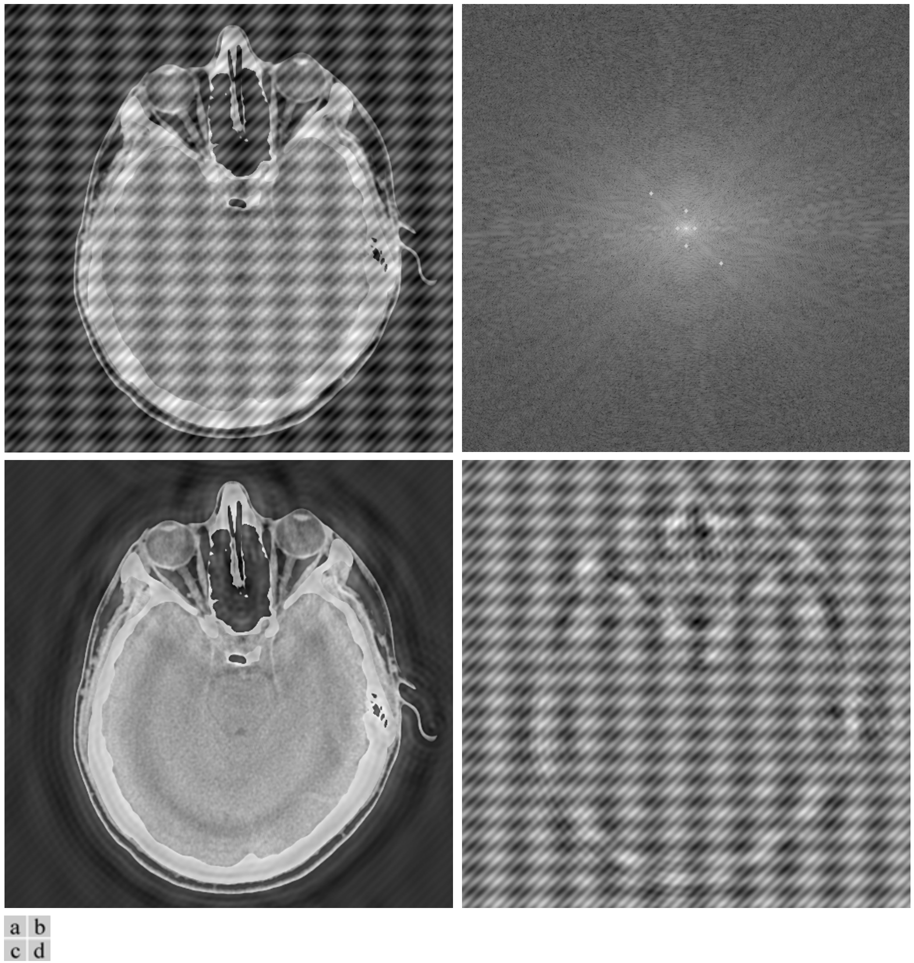


FIGURE P4.5 (a) Original image. (b) Spectrum, showing impulses corresponding to noise pattern (the impulses were enlarged to make them easier to see). (c) Denoised image obtained using bandreject filtering. (d) Noise pattern obtained using bandpass filtering. (Original image courtesy of Dr. David Pickens, Department of Radiology and Radiological Sciences, Vanderbilt University Medical Center.)

* *

PROJECT 4.6 Unsharp Masking in the Frequency Domain

With reference to the discussion in Project 3.6, a frequency domain implementation of unsharp masking consists of adding to an image a fraction of an sharpened version of the image.

(a) Download image `FigP0406(blurry_moon).tif` and enhance it using unsharp masking. Your final result should be of class `uint8`, which is the same class as the result obtained in Project 3.6. Use a Butterworth highpass filter of order 2 with D0 = 256 to generate the sharp filter that is added to the original (see Problem 3.6). Vary *k* in increments of 1 until the absolute difference between your result and the result from Project 3.6 is as small as you can make it. Display both results.

(b) Repeat (a) but this time using a frequency domain filter generated from the spatial mask `w = [0 1 0; 1 -4 1; 0 1 0]` (see Section 4.4). Convert your final result to class `uint8`. The absolute difference should be zero in this case.

SOLUTION

(a) In the following solution we wrote a small program to mechanize the procedure. The inputs to the program were the original image, the image from Project 3.6, and *k*. The program outputs the result of unsharp masking in the frequency domain and the absolute difference between the two images. The value $k = 4$ yielded the smallest absolute difference ($k = 5$ yielded the same result).

```
>> f = imread('FigP0406(blurry_moon).tif');
>> imshow(f) % Fig. P4.6(a).
%  Redo the image from Project 3.6
>> w = fspecial('unsharp', 0);
>> g_unsharpspatial = imfilter(f, w);
>> figure, imshow(g_unsharpspatial); % Fig. P4.6(b)

%  Now do unsharp masking in the freq. domain.
>> fd = im2double(f);
>> PQ = paddedsize(size(f));
>> H = hpfilter('btw', PQ(1), PQ(2), 256, 2);
>> fsharp = dftfilt(fd, H);
>> k = 4;
>> g_unsharpfd = fd + k*fsharp;
>> g_unsharpfd = im2uint8(g_unsharpfd);
>> figure, imshow(g_unsharpfd) % Fig. P4.6(c).
>> diff1 = imabsdiff(g_unsharpfd, g_unsharpspatial);
>> max(diff1(:))

ans =

     28
```

The results in Figs 4.6(b) and (c) are quite close visually.

(b) Next, we generate the exact frequency domain equivalent of the spatial unsharp filter and use the approach discussed in Section 4.4.

```
>> PQ = paddedsize(size(f));
>> h = fspecial('unsharp', 0);
>> Hfd=freqz2(h, PQ(1), PQ(2));
>> Hfd = ifftshift(Hfd);
>> gfd = dftfilt(fd, Hfd);
>> gfd8 = im2uint8(gfd);
>> diff2 = imabsdiff(gfd8, g_unsharpspatial);
>> max(diff2(:))

ans =

     0

>> figure, imshow(gfd8) % Fig. P4.6(d).
```

Although the difference is numerically lower, the visual appearance of this result is identical to Fig. 4.6(c) for all practical purposes.

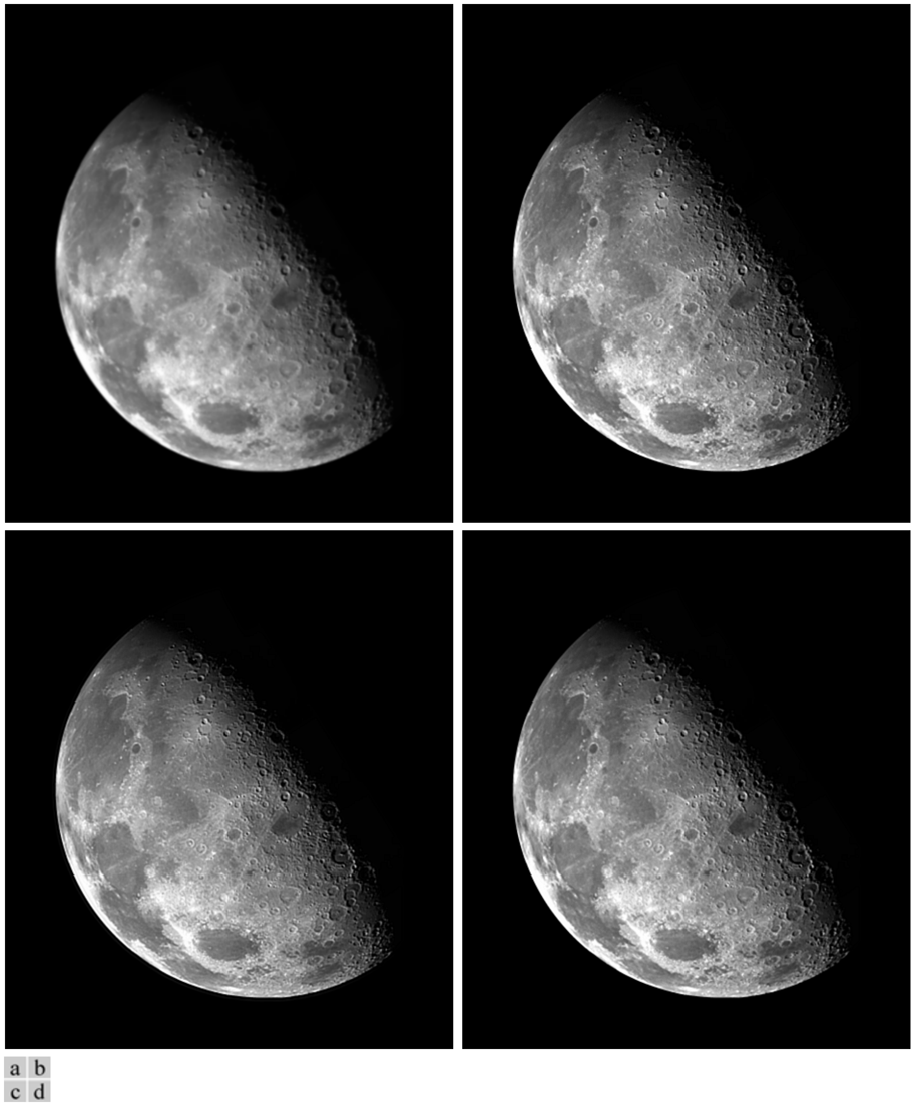


FIGURE P4.6 (a) Original image. (b) Image sharpened using unsharp masking in the spatial domain. (c) Image sharpened with unsharp masking in the frequency domain techniques based on a Butterworth filter and a scaling factor of 5. (d) Unsharp masking result obtained using a frequency domain filter generated from the spatial filter used in (a). (Original image courtesy of NASA.)

* *

PROJECT 4.7 Laplacian in the Frequency Domain

It is not difficult to show that the Laplacian in the frequency domain can be implemented using the filter $H(u,v) = -4\pi^2(u^2 + v^2)$; that is,

$$\Im\left[\nabla^2 f(x,y)\right] = -4\pi^2(u^2 + v^2)F(u,v)$$

(a) Download image `FigP4.6(a)(blurry_moon).tif` and implement in the frequency domain the steps leading to the images in Fig. 3.17(d) in the book. You will note that your image looks more like Fig. 3.18(c), indicating that results obtained in the frequency domain more closely resemble the spatial results obtained using a Laplacian mask with a −8, rather than a −4, in the center.

(b) Generate a frequency domain filter from `h = [1 1 1; 1 -8 1; 1 1 1]` using the approach discussed in Section 4.4. Use this filter to process the image in the frequency domain and compare your result with (a).

SOLUTION

```
>> f = imread('FigP0406(blurry_moon).tif');
>> imshow(f); % Fig. P4.7(a)
>> fd = im2double(f);
```

(a) Construct the filter and then obtain the Laplacian.

```
>> PQ = paddedsize(size(f));
>> [U, V] = dftuv(PQ(1), PQ(2));
>> H = -4*(pi)^2*(U.^2 + V.^2);
>> g1 = dftfilt(fd, H); %This is del^2[f(x, y)] in double format.
```

Due to filter values and FFT scaling, the range of values of `g1` typically is far greater than the range of values in the original image. One way to bring the values within a comparable range is to use divide `g1` by its maximum value so that its maximum value will be 1 (it is important that its negative values be kept) and to convert `f` to double by using function `im2double`, which guarantees that its maximum value will be 1 also.

```
>> g2 = fd - g1./max(g1(:));
>> figure, imshow(g2) % Fig. P4.7(b).
```

(b)

```
>> h = [1 1 1; 1 -8 1; 1 1 1];
>> H2 = ifftshift(freqz2(h, PQ(1), PQ(2)));
>> g3 = dftfilt(f, H2);
>> g4 = fd - g3./max(g3(:));
>> figure, imshow(g4) % Fig. P4.7(c).
```

Obtain and show the difference image:

```
>> gd = g2 - g3;
>> figure, imshow(gd) % Fig. P4.7(d)
```

The difference image shows relatively small values. In fact, the maximum value of `gd` is less than 25% of the maximum of either `g2` or `g3`:

```
>> max(gd(:))

ans =

    0.4250

>> max(g2(:))

ans =
    1.8898

>> max(g4(:))

ans =

    1.8716
```

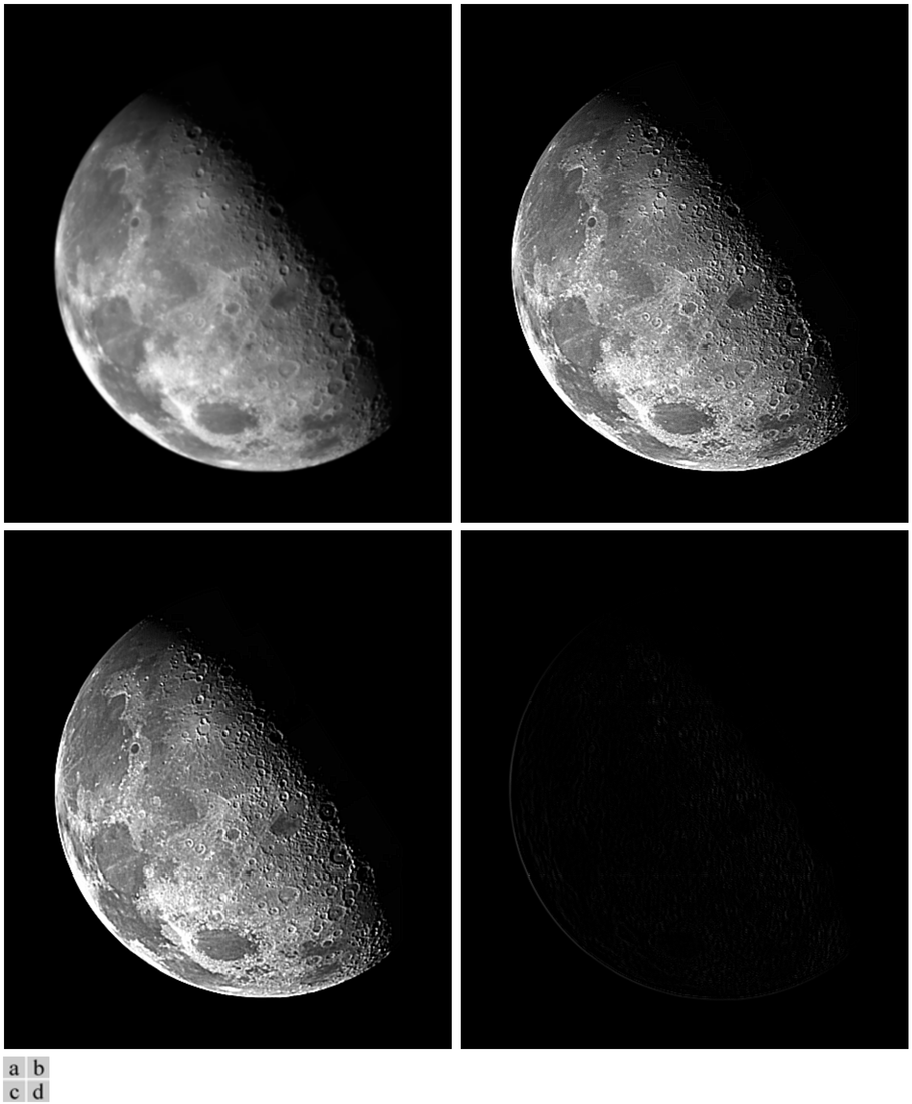


FIGURE P4.7 (a) Original image. (b) Image enhanced using the Laplacian filter in the frequency domain. (c) Image enhanced using a frequency domain filter derived from the spatial Laplacian. (d) Difference image. (Original image courtesy of NASA.)

* *

PROJECT 4.8 Homomorphic Filtering

(a) Homomorphic filtering attempts to achieve simultaneous reduction of the dynamic range of an image and coupled with an increase in sharpness (see Gonzalez and Woods [2002] for details of this technique). A frequency domain filter used frequently for homomorphic filtering has the form

$$H(u,v) = A + \frac{C}{1 + \left[D0 / D(u,v)\right]^B}$$

where $D(u,v)$ is the distance from the origin of the (centered) Fourier transform, and $A < B$. Write a function for performing homomorphic using this filter as the default:

```
function g = homofilt(f, varargin)
%HOMOFILT Homomorphic filtering.
```

```
%   G = HOMOFILT(F, H) performs homomorphic filtering on the input
%   image, F, using the specified filter, H.  G = HOMOFILT(F, A, B,
%   C, D0) performs homomorphic filtering on F using the default
%   filter:
%
%        H = A + C/{1 + [D0/D(U, V)]^B}
%
%   For this filter to make sense, it is required that A < B.  A good
%   set of starting values: A = 0.25, B = 2, C = 2, and D0 = min(M, N)/8.
```

(b) Read image `FigP0408(PET_image).tif` and perform homomorphic filtering on it using the default filter.

SOLUTION

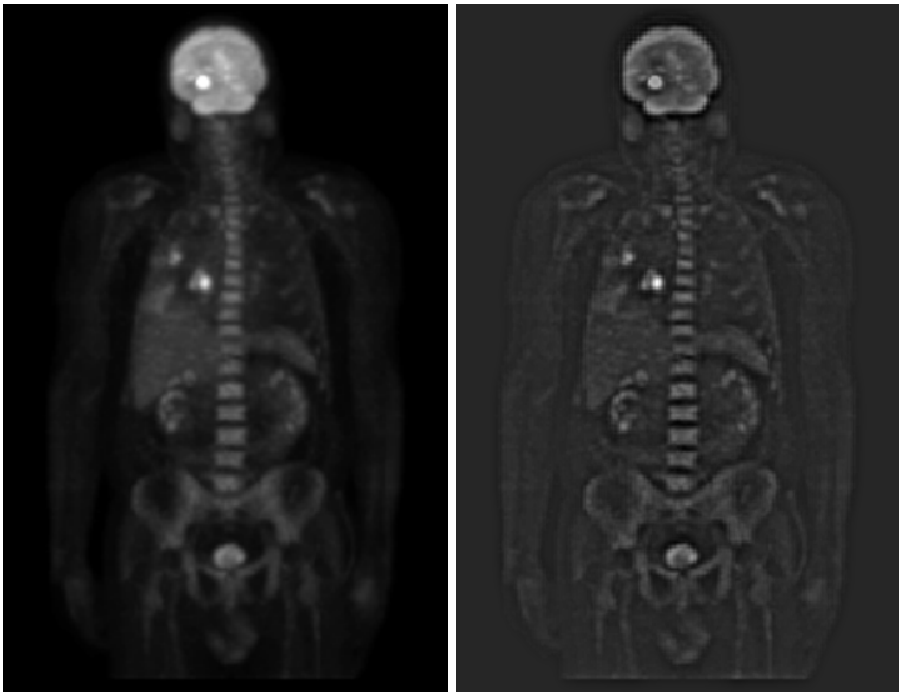
(a)

```
function g = homofilt(f, varargin)
%HOMOFILT Homomorphic filtering.
%   G = HOMOFILT(F, H) performs homomorphic filtering on the input
%   image, F, using the specified filter, H.  G = HOMOFILT(F, A, B,
%   C, D0) performs homomorphic filtering on F using the default
%   filter:
%
%        H = A + C/{1 + [D0/D(U, V)]^B}
%
%   For this filter to make sense, it is required that A < B.  A good
%   set of starting values: A = 0.25, B = 2, C = 2, and D0 = min(M, N)/8.

% Process inputs.
if length(varargin) == 4
  A = varargin{1};
  B = varargin{2};
  C = varargin{3};
  D0 = varargin{4};
  if A >= B
     disp('Warning: A should be less than B.')
  end
  % Set up meshgrid array and compute distances.
  PQ = paddedsize(size(f));
  [U, V] = dftuv(PQ(1), PQ(2));
  % Compute the distances, D(U, V).
  D = sqrt(U.^2 + V.^2);
  % Generate the filter.
  H = A + C./(1 + (D0./(D + eps)).^B);
elseif length(varargin) == 1
   H = varargin{1};
else
   error('Incorrect number of inputs.')
end

% Compute the natural log of the input image.
f = log(im2double(f) + 1);

% Now do the filtering and compute exp of the result.
g = dftfilt(f, H);
g = exp(g) - 1;
```

(b) We started with the default values and experimented with D0 to obtain the results in Fig. P4.8(b).

```
>> f = imread('FigP0408(PET_image).tif');
>> imshow(f); % Fig. P4.8(a).
>> g = homofilt(f, .25, 2, 2, 40);
>> figure, imshow(g, []) % Fig. P4.8(b).
```

Note how much sharper the "hot" spots are in the processed image, and how much more detail is visible in this image.



a b

FIGURE P4.8 (a) Original image. (b) Image enhanced using homomorphic filtering. (Original image courtesy of Computer Technology and Imaging.)

Chapter 5
Sample Project Solutions

PROJECT 5.1 Estimating Noise PDFs and Their Parameters

The image in `FigP0501(noisy_superconductor_image).tif` is corrupted by noise.

(a) Download this image and extract its *noise* histogram. Display the histogram (using function `bar`) and indicate (by name) what you think the noise PDF is. Determine the relevant noise parameter(s) using the histogram you extracted. (*Hint*: Use function `roipoly` to extract the data you think will help you identify the noise.)

(b) Use function `imnoise` or `imnoise2`, as appropriate, to generate `X` samples of the noise type and parameter(s) you determined in (a). Generate the histogram of the samples using function `hist`, and display the histogram. Here, `X` is the number of pixels in the ROI in (a). Compare with the corresponding histogram from (a).

SOLUTION

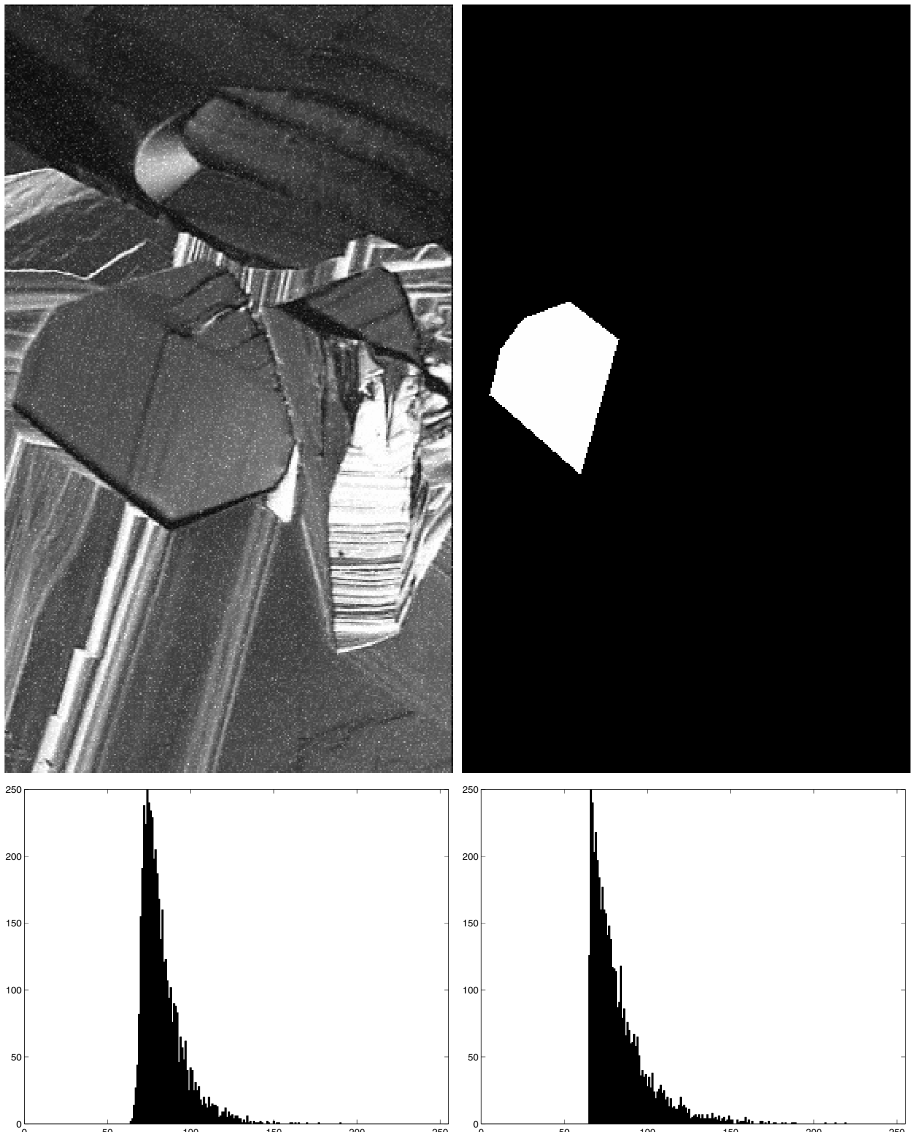
(a) To obtain the PDF of the noise we use function roipoly to extract data from a region with relatively constant background.

```
>> fn = imread('FigP0501(noisy_superconductor_image).tif');
>> imshow(fn) % Fig. P5.1(a).
>> [B, c, r] = roipoly(fn);
>> figure, imshow(B) % Fig. P5.1(b)(region of interest).
>> [p, npix] = histroi(fn, c, r); %Histogram of roi.
>> figure, bar(p, 1)
>> axis([0 256 0 250]) % Fig. 5.1(c).
>> [v, unv] = statmoments(p, 2) %Obtain mean and variance.

v =

    0.3223    0.0026

unv =

   82.1947  167.2227
```

By comparing the histogram in Fig. P5.1(c) and Fig. 5.2 in the book, we conclude that the closest match is an exponential PDF. As Fig. 5.2(e) shows, this PDF starts at 0, and the PDF in Fig. P5.1(c) starts at about 64. If we subtract 64 to bring the PDF to 0, the mean, `unv(1)`, becomes 18 $(82.2 - 64 \approx 18)$. We then generate `npix` exponential random variables using function `imnoise2` with a = 1/18 (see Table 5.1) and add 64 back so that the resulting histogram can be compared with `p` from (a):

```
>> exn = imnoise2('exponential', npix, 1, 1/18) + 64;
>> z = 0:255;
>> h = hist(exn, z);
>> figure, bar(h, 1)
>> axis([0 255 0 250]) % Fig. P5.1(d)
```

By comparing Figs. P5.1(c) and (d) we see a reasonable resemblance between the two histograms, with the exception that the rise at the onset of the PDF in Fig. 5.1(c) is not as sharp. However, an exponential PDF is the closest match from the possible PDFs that can be obtained with random data generated using function `imnoise2`. (Note: Student results will be slightly different, depending on the ROI they select, and on the particular random sequence generated.)



a b
c d

FIGURE P5.1 (a) Noisy, optical microscope image of a superconductor. (b) ROI determined using function `roipoly`. (c) Histogram of ROI. (d) Histogram of exponential random variables generated using the parameters obtained from the ROI data. (Original image courtesy of Dr. Michael W. Davidson, Florida State University.)

* *

PROJECT 5.2 Spatial Noise Reduction

(a) Use function `spfilt` to denoise image `FigP0502(a)(salt_only).tif`. This is an optical microscope image of a nickel-oxide thin film specimen magnified 600X. The image is heavily corrupted by salt noise.

(b) Use function `spfilt` to denoise image `FigP0502(b)(pepper_only).tif`. This is the same nickel oxide image, but corrupted this time by pepper noise only.

The tradeoff in your selection of a filter in (a) and (b) should be to produce the cleanest image possible with as little image distortion (e.g., blurring) as possible. (*Hint:* If you need to become more familiar with the details of the filters available in function `spfilt`, consult Chapter 5 of Gonzalez and Woods [2002].)

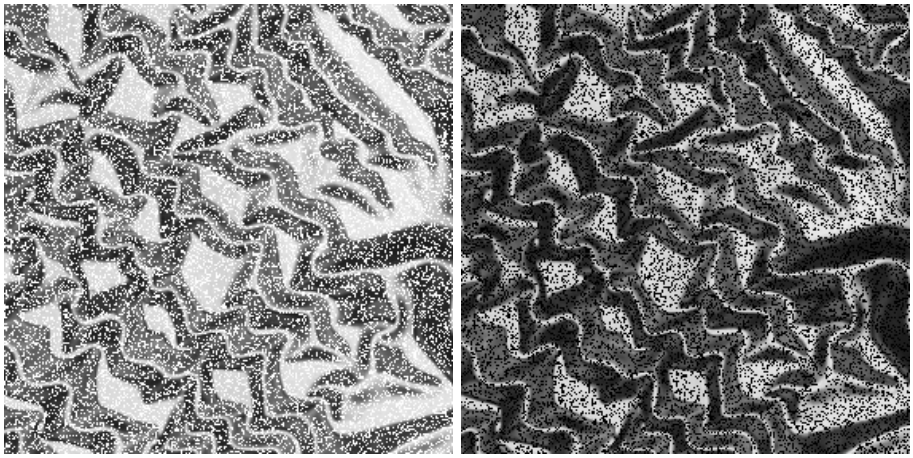
SOLUTION

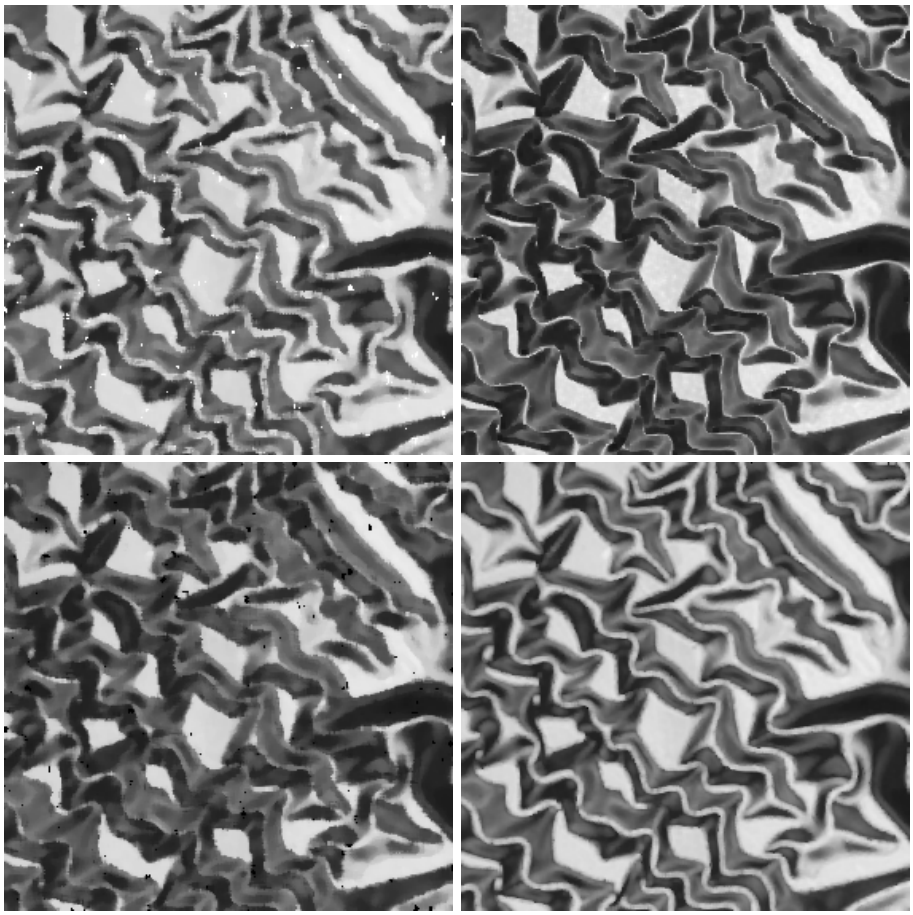
(a) The best choices in `spfilt` for the elimination of salt noise are median and contraharmonic mean filters. We give solutions for both. After a little experimentation and visual evaluation of the results we obtained the following results:

```
>> gs = imread('FigP0502(a)(salt_only).tif');
>> imshow(gs)  % Fig. P5.2(a).
>> fsmed = spfilt(g, 'median', 5, 5);
>> figure, imshow(fsmed)  % Fig. P5.2(c).
>> fschmean = spfilt(g,'chmean', 3, 3, -5);
>> figure, imshow(fschmean)  % Fig. P5.2(d).
```

(b) For the pepper image, the best median filter was the same as in (a), but the best contraharmonic mean filter result was simply the default, which is a 3 3 filter with Q = 1.5. Note that the contraharmonic mean filter yielded superior results for both salt and pepper noise corruption.

```
>> gp = imread('FigP0502(b)(pepper_only).tif');
>> figure, imshow(gp)  % Fig. P5.2(b).
>> fpmed = spfilt(gp, 'median', 5, 5);
>> figure, imshow(fpmed)  % FigP5.2(e).
>> fpchmean = spfilt(gp,'chmean');
>> figure, imshow(fpchmean)  % FigP5.2(f).
```





a b
c d
e f

FIGURE P5.2 (a) Image corrupted by salt noise. (b) Image corrupted by pepper noise. (c) Result of filtering (a) with a 5 5 median filter. (d) Result of filtering (a) with a contraharmonic mean filter of size 3 3 with Q = –5. (e) Result of filtering (b) with a median filter of size 5 5. (e) Result of filtering (b) with a contraharmonic mean filter of size 3 3 with Q = 1.5. (Original image courtesy of Dr. Michael W. Davidson, Florida State University.)

* *

PROJECT 5.3 Adaptive Median Filtering

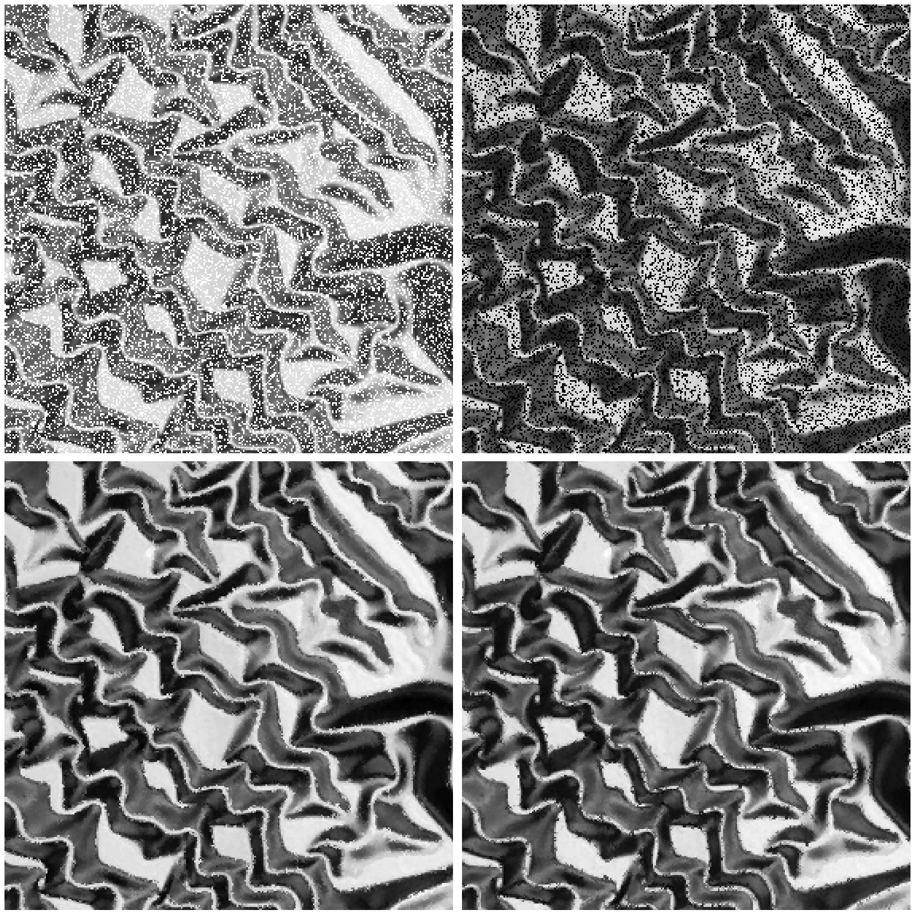
Repeat Project 5.2 using adaptive median filtering and compare the results by generating difference images of the results. Select the size of filter window to give the best (in your opinion) visual results.

SOLUTION

The best visual results for both images were obtained using a window of size 9 9. The results are slightly noisier than the best results in Project 5.2, but the images are sharper.

```
>> gs = imread('FigP0502(a)(salt_only).tif');
>> imshow(gs)  % Fig. P5.3(a).
>> gp = imread('FigP0502(b)(pepper_only).tif');
>> figure, imshow(gp)  % Fig. P5.3(b).
>> fsadpmed = adpmedian(gs, 9);
>> fsadpmed = gscale(fsadpmed);
>> figure, imshow(gsadpmed)  % Fig. P5.3(c).
```

```
>> fpadpmed = adpmedian(gp, 9);
>> fpadpmed = gscale(fpadpmed);
>> figure, imshow(fpadpmed)  % Fig. P5.3(d).
```



a b
c d

FIGURE P5.3 (a) Image corrupted by salt noise. (b) Image corrupted by pepper noise. (c) Image in (a) processed using function `adpmedian` with a window of size 9   9. (d) Image in (b) processed using function `adpmedian` with a window of size 9   9.

* *

PROJECT 5.4 Notch Filtering

Use function `notchfilter` to denoise `FigP4.5(a)(HeadCT_corrupted).tif`. Show the original image, an image of your specified filter, and the restored result. If you completed Project 4.5, compare your result with the image denoised using bandreject filtering in that project. If you use function `impixelinfo` to determine the coordinates of the impulses interactively, keep in mind that this function lists the column coordinates (`v`) first and the row coordinates (`u`) second.

NOTE: When attempting to remove sinusoidal spatial interference, it generally is advisable to do all DFT filtering without padding. Even if the pattern is a pure sine wave, padding introduces frequencies in the sine wave that often are quite visible and difficult to remove in the filtered image. Thus, in this case, the effect of wraparound error introduced by not using padding usually is negligible by comparison.

SOLUTION

```
>> f = imread('FigP0405(HeadCT_corrupted).tif');
>> imshow(f) % Fig. P5.5(a).
```

Use the spectrum from Project 4.5 to obtain the location of the impulses for use in specifying the centers of the notches. The locations thus determined were (256, 266), (276, 256), and (296, 296). We used the following notch filter to denoise the image:

```
>> Hnr = cnotch ('btw', 'reject', 512, 512, [256 266; 276 256; 296 296], 5, 10);
```

The filtering was done as follows:

```
>> g = dftfilt(f, Hnr);
>> figure, imshow(g, [ ])
% Convert to uint8 and scale.
>> g = gscale(g);
>> figure, imshow(g) % Fig. P5.5(b).
```

As Fig. P5.5(b) shows, the sinusoidal interference was removed for all practical purposes. To compare with the equivalent result from Project 4.5 (which also was scaled and converted to `uint8`), we compute the difference between Figs. P4.5(c) and Fig. P5.5(b):

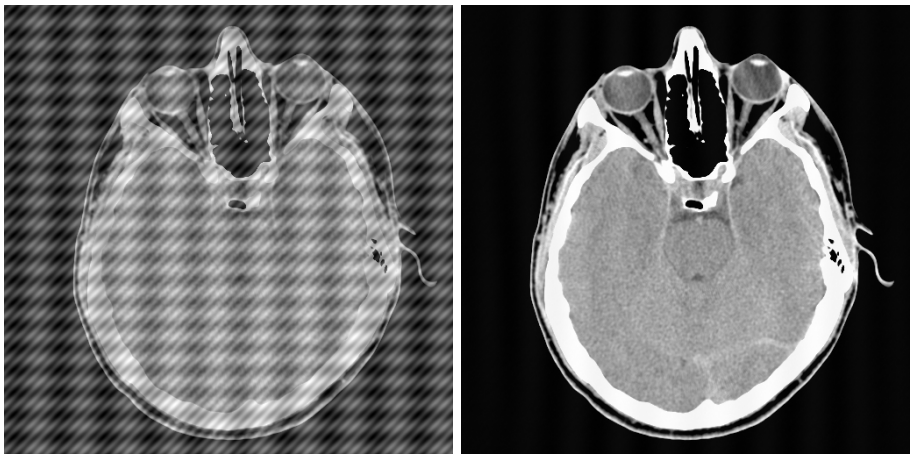
```
>> g4pt5c = imread('FigP0405(c)(denoised_image).tif');
>> d = imsubtract(g, g4pt5c);
>> figure, imshow(d,[]) % Fig. P5.5(c)
```

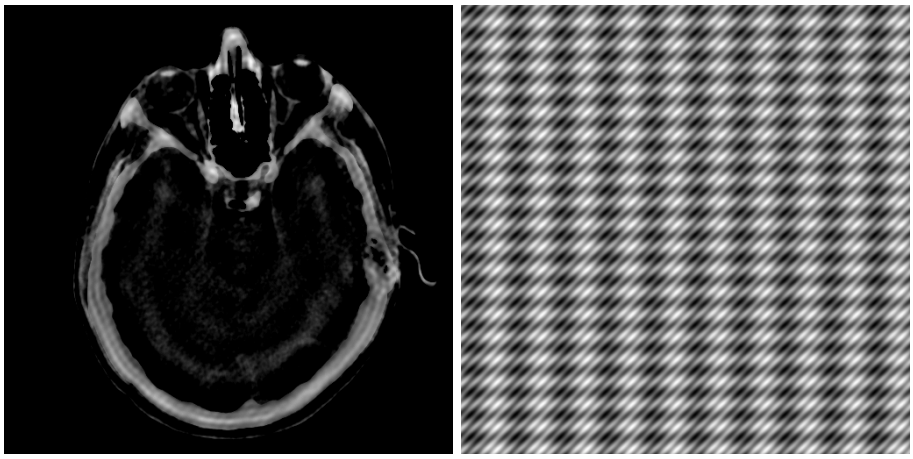
The differences between the two images is evident, and it consists mainly of the data that did not correspond to the impulses, but was passed by the bandpass filter.

Finally, we extract the noise pattern by filtering the image with a notchpass filter:

```
>> Hnp = 1 - Hnr;
>> gn = dftfilt(f, Hnp);
>> figure, imshow(gn, [ ]) % Fig. P5.5(d).
```

Comparing Fig. P5.5(b) with Fig. P4.5(c) we again see that notch filtering was superior.





a b
c d

FIGURE P5.5 (a) Original image. (b) Image denoised using notch filtering. (c) Noise pattern obtained using notchpass filtering. (d) Difference between Figs. P4.5(c) and P5.5(b). (Original image courtesy of Dr. David Pickens, Department of Radiology and Radiological Sciences, Vanderbilt University Medical Center.)