The specifics can be found in the `imwrite` help file. The `imwrite` routine can be used to store any of the data formats or data classes mentioned above; however, if the data array, `I`, is an indexed array, then it must be followed by the colormap variable, `map`. Most image formats actually store uint8 formatted data, but the necessary conversions are done by the `imwrite`.

The `imread` function is used to retrieve images from disk. It has the calling structure:

```
[I map] = imread('filename.ext',fmt or frame);
```

where `filename` is the name of the image file and `.ext` is any of the extensions listed above. The optional second argument, `fmt`, only needs to be specified if the file format is not evident from the filename. The alternative optional argument `frame` is used to specify which frame of a multiframe image is to be read in `I`. An example that reads multiframe data is found in Example 10.4. As most file formats store images in uint8 format, `I` will often be in that format. File formats `.tif` and `.png` support uint16 format, so `imread` may generate data arrays in uint16 format for these file types. The output class depends on the manner in which the data is stored in the file. If the file contains a grayscale image data, then the output is encoded as an intensity image, if truecolor, then as RGB. For both these cases the variable `map` will be empty, which can be checked with the `isempty(map)` command (see Example 10.4). If the file contains indexed data, then both output, `I` and `map` will contain data.

The type of data format used by a file can also be obtained by *querying* a graphics file using the function `infinfo`.

```
information = infinfo('filename.ext')
```

where `information` will contain text providing the essential information about the file including the ColorType, FileSize, and BitDepth. Alternatively, the image data and map can be loaded using `imread` and the format image data determined from the MATLAB `whos` command. The `whos` command will also give the structure of the data variable (uint8, uint16, or double).

Basic Arithmetic Operations

If the image data are stored in the double format, then all MATLAB standard mathematical and operational procedures can be applied directly to the image variables. However, the double format requires 4 times as much memory as the uint16 format and 8 times as much memory as the uint8 format. To reduce the reliance on the double format, MATLAB has supplied functions to carry out some basic mathematics on uint8- and uint16-format arrays. These routines will work on either format; they actually carry out the operations in double precision

on an element by element basis then convert back to the input format. This reduces roundoff and overflow errors. The basic arithmetic commands are:

```
I_diff = imabsdiff(I, J);        % Subtracts J from I on a pixel
                                 %  by pixel basis and returns
                                 %  the absolute difference
I_comp = imcomplement(I)         % Compliments image I
I_add = imadd(I, J);             % Adds image I and J (images and/
                                 %  or constants) to form image
                                 %  I_add
I_sub = imsubtract(I, J);        % Subtracts J from image I
I_divide = imdivide(I, J)        % Divides image I by J
I_multiply = immultiply(I, J)    % Multiply image I by J
```

For the last four routines, `J` can be either another image variable, or a constant. Several arithmetical operations can be combined using the `imlincomb` function. The function essentially calculates a weighted sum of images. For example to add 0.5 of image $I1$ to 0.3 of image $I2$, to 0.75 of Image $I3$, use:

```
% Linear combination of images
I_combined = imlincomb (.5, I1, .3, I2, .75, I3);
```

The arithmetic operations of multiplication and addition by constants are easy methods for increasing the contrast or brightness or an image. Some of these arithmetic operations are illustrated in Example 10.4.

Example 10.4 This example uses a number of the functions described previously. The program first loads a set of MRI (magnetic resonance imaging) images of the brain from the MATLAB Image Processing Toolbox's set of stock images. This image is actually a multiframe image consisting of 27 frames as can be determined from the command `imifinfo`. One of these frames is selected by the operator and this image is then manipulated in several ways: the contrast is increased; it is inverted; it is sliced into 5 levels (`N_slice`); it is modified horizontally and vertically by a Hanning window function, and it is thresholded and converted to a binary image.

```
% Example 10.4 and Figures 10.5 and 10.6
% Demonstration of various image functions.
% Load all frames of the MRI image in mri.tif from the the MATLAB
%  Image Processing Toolbox (in subdirectory imdemos).
% Select one frame based on a user input.
% Process that frame by: contrast enhancement of the image,
%  inverting the image, slicing the image, windowing, and
%  thresholding the image
```

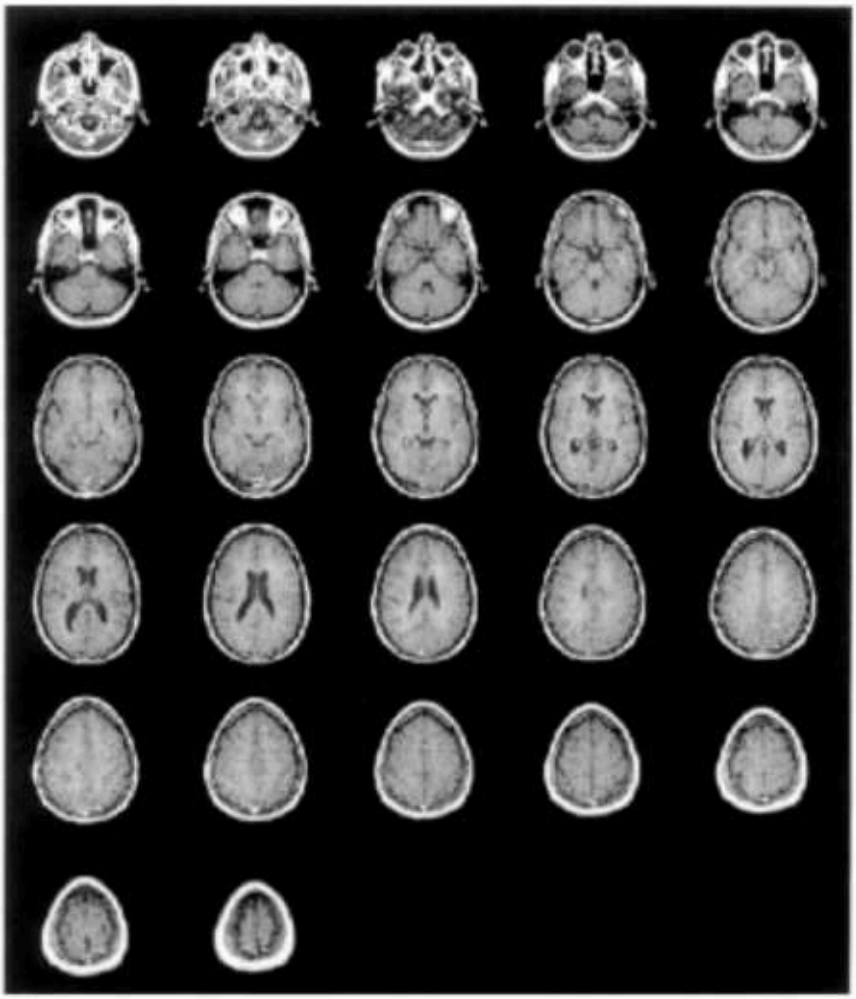

FIGURE 10.5 *Montage* display of 27 frames of magnetic resonance images of the brain plotted in Example 10.4. These multiframe images were obtained from MATLAB's `mri.tif` file in the images section of the Image Processing Toolbox. Used with permission from MATLAB, Inc. Copyright 1993–2003, The Math Works, Inc. Reprinted with permission.

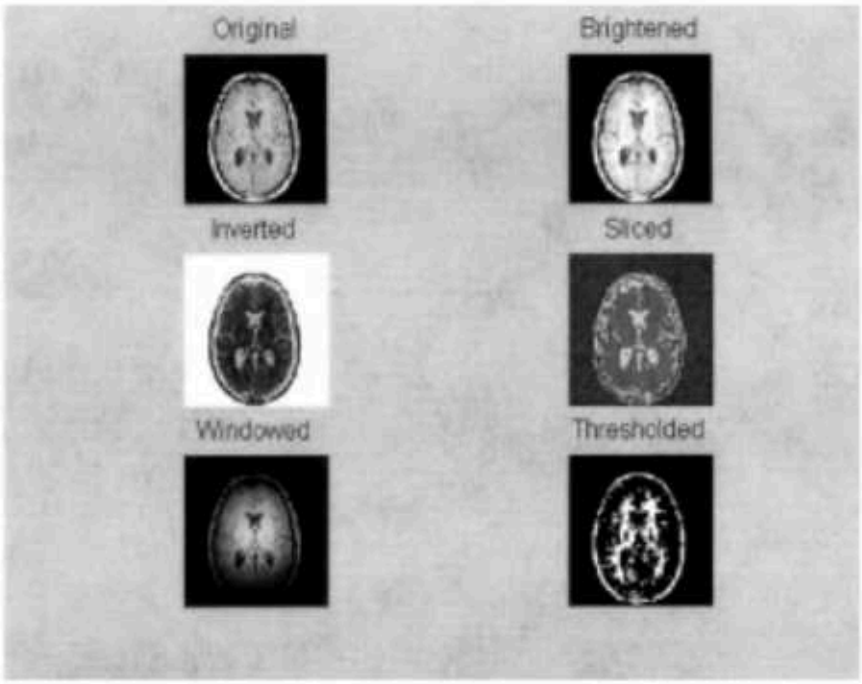

FIGURE 10.6 Figure showing various signal processing operations on frame 17 of the MRI images shown in Figure 10.5. Original from the MATLAB Image Processing Toolbox. Copyright 1993–2003, The Math Works, Inc. Reprinted with permission.

```
% Display original and all modifications on the same figure
%
clear all; close all;
N_slice = 5;                                % Number of sliced for
                                            %  sliced image
Level = .75;                                % Threshold for binary
                                            %  image
%
% Initialize an array to hold 27 frames of mri.tif
% Since this image is stored in tif format, it could be in either
%  unit8 or uint16.
% In fact, the specific input format will not matter, since it
%  will be converted to double format in this program.
mri = uint8(zeros(128,128,1,27));           % Initialize the image
                                            %  array for 27 frames
for frame = 1:27                            % Read all frames into
                                            %  variable mri
```

```
  [mri(:,:,:,frame), map ] = imread('mri.tif', frame);
end
montage(mri, map);                          % Display images as a
                                            %  montage
                                            % Include map in case
                                            %  Indexed
%
frame_select = input('Select frame for processing: ');
I = mri(:,:,:,frame_select);                % Select frame for
                                            %  processing
%
% Now check to see if image is Indexed (in fact 'whos' shows it
%  is).
if isempty(map) == 0                        % Check to see if
                                            %  indexed data
  I = ind2gray(I,map);                      % If so, convert to
                                            %  intensity image
end
I1 = im2double(I);                          % Convert to double
                                            %  format
%
I_bright = immultiply(I1,1.2);              % Increase the contrast
I_invert = imcomplement(I1);                % Compliment image
x_slice = grayslice(I1,N_slice);            % Slice image in 5 equal
                                            %  levels
%
[r c] = size(I1);                           % Multiple
for i = 1:r                                 %  horizontally by a
                                            %  Hamming window
  I_window(i,:) = I1(i,:) .* hamming(c)';
end
for i = 1:c                                 % Multiply vertically
                                            %  by same window
  I_window(:,i) = I_window(:,i) .* hamming(r);
end
I_window = mat2gray(I_window);              % Scale windowed image
BW = im2bw(I1,Level);                       % Convert to binary
%
figure;
subplot(3,2,1);                             % Display all images in
                                            %  a single plot
  imshow(I1); title('Original');
subplot(3,2,2);
  imshow(I_bright), title('Brightened');
subplot(3,2,3);
```

```
  imshow(I_invert); title('Inverted');
subplot(3,2,4);
  I_slice = ind2rgb(x_slice, jet        % Convert to RGB (see
  (N_slice));                           %  text)
  imshow(I_slice); title('Sliced');     % Display color slices
subplot(3,2,5);
  imshow(I_window); title('Windowed');
subplot(3,2,6);
  imshow(BW); title('Thresholded');
```

Since the image file might be indexed (in fact it is), the `imread` function includes map as an output. If the image is not indexed, then map will be empty. Note that `imread` reads only one frame at a time, the frame specified as the second argument of `imread`. To read in all 27 frames, it is necessary to use a loop. All frames are then displayed in one figure (Figure 10.5) using the `montage` function. The user is asked to select one frame for further processing. Since montage can display any input class and format, it is not necessary to determine these data characteristics at this time.

After a particular frame is selected, the program checks if the map variable is empty (function `isempty`). If it is not (as is the case for these data), then the image data is converted to grayscale using function `ind2gray` which produces an intensity image in double format. If the image is not Indexed, the image variable is converted to double format. The program then performs the various signal processing operations. Brightening is done by multiplying the image by a constant greater that 1.0, in this case 1.2, Figure 10.6. Inversion is done using `imcomplement`, and the image is sliced into `N_slice` (5) levels using `grayslice`. Since `grayslice` produces an indexed image, it also generates a map variable. However, this `grayscale` map is not used, rather an alternative map is substituted to produce a color image, with the color being used to enhance certain features of the image.* The Hanning window is applied to the image in both the horizontal and vertical direction Figure 10.6. Since the image, `I1`, is in double format, the multiplication can be carried out directly on the image array; however, the resultant array, `I_window`, has to be rescaled using `mat2gray` to insure it has the correct range for `imshow`. Recall that if called without any arguments; `mat2gray` scales the array to take up the full intensity range (i.e., 0 to 1). To place all the images in the same figure, `subplot` is used just as with other graphs, Figure 10.6. One potential problem with this approach is that Indexed data may plot incorrectly due to limited display memory allocated to

*More accurately, the image should be termed a pseudocolor image since the original data was grayscale. Unfortunately the image printed in this text is in grayscale; however the example can be rerun by the reader to obtain the actual color image.

the map variables. (This problem actually occurred in this example when the sliced array was displayed as an Indexed variable.) The easiest solution to this potential problem is to convert the image to RGB before calling `imshow` as was done in this example.

Many images that are grayscale can benefit from some form of color coding. With the RGB format, it is easy to highlight specific features of a grayscale image by placing them in a specific color plane. The next example illustrates the use of color planes to enhance features of a grayscale image.

Example 10.5 In this example, brightness levels of a grayscale image that are 50% or less are coded into shades of blue, and those above are coded into shades of red. The grayscale image is first put in double format so that the maximum range is 0 to 1. Then each pixel is tested to be greater than 0.5. Pixel values less that 0.5 are placed into the blue image plane of an RGB image (i.e., the third plane). These pixel values are multiplied by two so they take up the full range of the blue plane. Pixel values above 0.5 are placed in the red plane (plane 1) after scaling to take up the full range of the red plane. This image is displayed in the usual way. While it is not reproduced in color here, a homework problem based on these same concepts will demonstrate pseudocolor.

```
% Example 10.5 and Figure 10.7 Example of the use of pseudocolor
% Load frame 17 of the MRI image (mri.tif)
%  from the Image Processing Toolbox in subdirectory 'imdemos'.
```

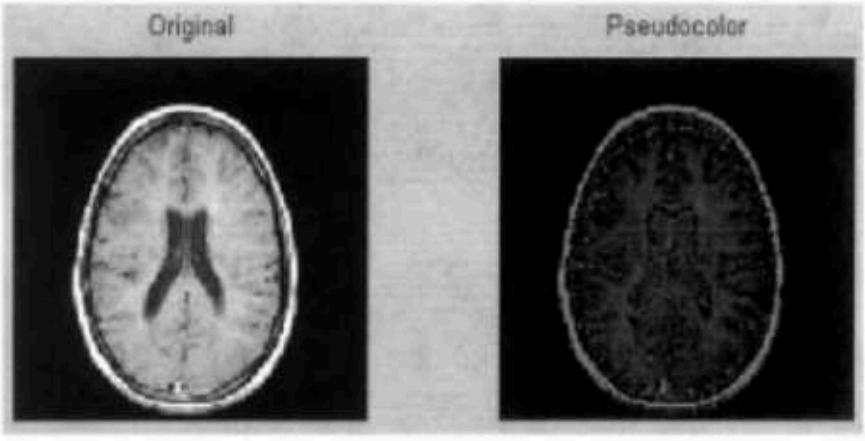

FIGURE 10.7 Frame 17 of the MRI image given in Figure 10.5 plotted directly and in pseudocolor using the code in Example 10.5. (Original image from MATLAB). Copyright 1993–2003, The Math Works, Inc. Reprinted with permission.