Analyzing and Enhancing Images

This section describes the Image Processing Toolbox functions that support a range of standard image processing operations for analyzing and enhancing images. Topics covered include

Terminology

An understanding of the following terms will help you to use this chapter.

Terms	Definitions
Adaptive filter	A filter whose properties vary across an image depending on the local characteristics of the image pixels.
Contour	A path in an image along which the image intensity values are equal to a constant.
Edge	A curve that follows a path of rapid change in image intensity. Edges are often associated with the boundaries of objects in a scene. Edge detection is used to identify the edges in an image.
Property	A quantitative measurement of an image or image region. Examples of image region properties include centroid, bounding box, and area.
Histogram	A graph used in image analysis that shows the distribution of intensities in an image. The information in a histogram can be used to choose an appropriate enhancement operation. For example, if an image histogram shows that the range of intensity values is small, you can use an intensity adjustment function to spread the values across a wider range.
Noise	Errors in the image acquisition process that result in pixel values that do not reflect the true intensities of the real scene.
Profile	A set of intensity values taken from regularly spaced points along a line segment or multiline path in an image. For points that do not fall on the center of a pixel, the intensity values are interpolated.
Quadtree decomposition	An image analysis technique that partitions an image into homogeneous blocks.

Pixel Values and Statistics

The Image Processing Toolbox provides several functions that return information about the data values that make up an image. These functions return information about image data in various forms, including:

- The data values for selected pixels (`pixval`, `impixel`)
- The data values along a path in an image (`improfile`)
- A contour plot of the image data (`imcontour`)
- A histogram of the image data (`imhist`)
- Summary statistics for the image data (`mean2`, `std2`, `corr2`)
- Feature measurements for image regions (`imfeature`)

Pixel Selection

The toolbox includes two functions that provide information about the color data values of image pixels you specify:

- The `pixval` function interactively displays the data values for pixels as you move the cursor over the image. `pixval` can also display the Euclidean distance between two pixels.
- The `impixel` function returns the data values for a selected pixel or set of pixels. You can supply the coordinates of the pixels as input arguments, or you can select pixels using a mouse.

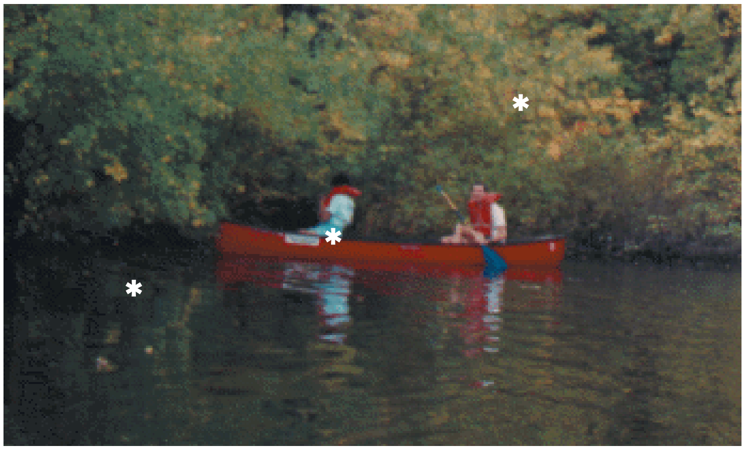
To use `pixval`, you first display an image and then enter the `pixval` command. `pixval` installs a black bar at the bottom of the figure, which displays the (x,y) coordinates for whatever pixel the cursor is currently over, and the color data for that pixel.

If you click on the image and hold down the mouse button while you move the cursor, `pixval` also displays the Euclidean distance between the point you clicked on and the current cursor location. `pixval` draws a line between these points to indicate the distance being measured. When you release the mouse button, the line and the distance display disappear.

`pixval` gives you more immediate results than `impixel`, but `impixel` has the advantage of returning its results in a variable, and it can be called either interactively or noninteractively. If you call `impixel` with no input arguments, the cursor changes to a crosshair when it is over the image. You can then click on the pixels of interest; `impixel` displays a small star over each pixel you select. When you are done selecting pixels, press **Return**. `impixel` returns the color values for the selected pixels, and the stars disappear.

In this example, you call `impixel` and click on three points in the displayed image, and then press **Return**.

```
imshow canoe.tif
vals = impixel
```



```
vals =

0.1294    0.1294    0.1294
0.5176         0         0
0.7765    0.6118    0.4196
```

Notice that the second pixel, which is part of the canoe, is pure red; its green and blue values are both 0.

For indexed images, `pixval` and `impixel` both show the RGB values stored in the colormap, not the index values.

Intensity Profile

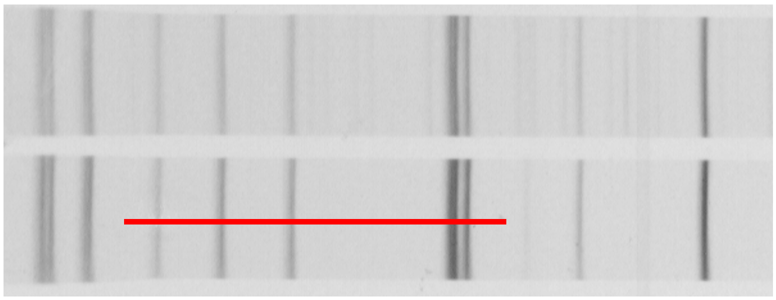
The `improfile` function calculates and plots the intensity values along a line segment or a multiline path in an image. You can supply the coordinates of the line segments as input arguments, or you can define the desired path using a mouse. In either case, `improfile` uses interpolation to determine the values of equally spaced points along the path. (By default, `improfile` uses nearest neighbor interpolation, but you can specify a different method. See Chapter 4, “Spatial Transformations”, for a discussion of interpolation.) `improfile` works best with intensity and RGB images.

For a single line segment, `improfile` plots the intensity values in a two-dimensional view. For a multiline path, `improfile` plots the intensity values in a three-dimensional view.

If you call `improfile` with no arguments, the cursor changes to a cross hair when it is over the image. You can then specify line segments by clicking on the endpoints; `improfile` draws a line between each two consecutive points you select. When you finish specifying the path, press **Return**. `improfile` displays the plot in a new figure.

In this example, you call `improfile` and specify a single line with the mouse. The line is shown in red, and is drawn from left to right.

```
imshow debye1.tif
improfile
```



`improfile` displays a plot of the data along the line.

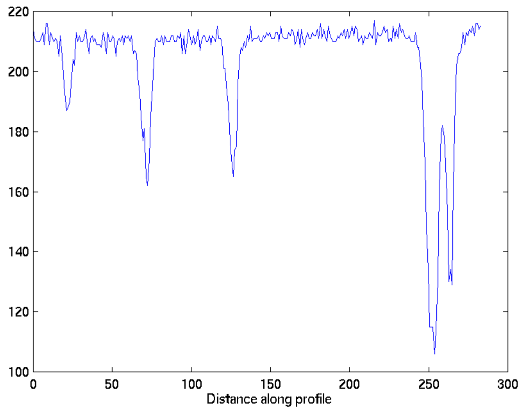
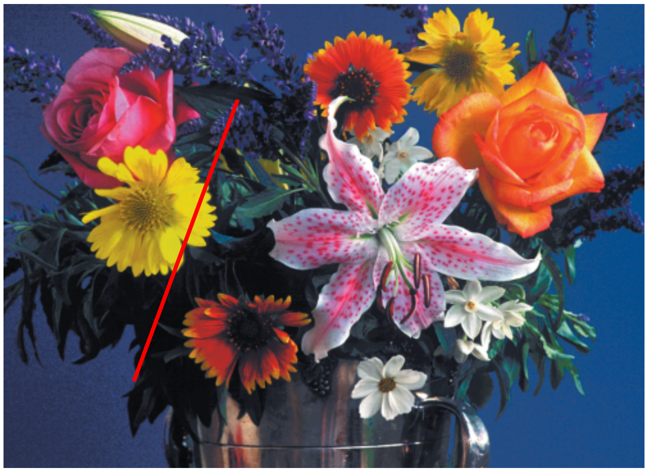


Figure 10-1: A Plot of Intensity Values Along a Line Segment in an Intensity Image

Notice the peaks and valleys and how they correspond to the light and dark bands in the image.

The example below shows how `improfile` works with an RGB image. The red line indicates where the line selection was made. Note that the line was drawn from top to bottom.

```
imshow flowers.tif
improfile
```



The `improfile` function displays a plot with separate lines for the red, green, and blue intensities.

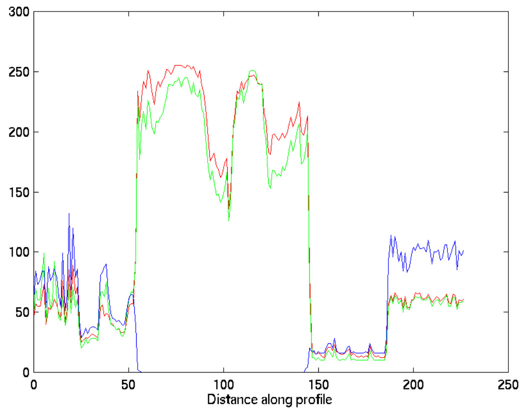


Figure 10-2: A Plot of Intensity Values Along a Line Segment in an RGB Image

Notice how the lines correspond to the colors in the image. For example, the central region of the plot shows high intensities of green and red, while the blue intensity is 0. These are the values for the yellow flower.

Image Contours

You can use the toolbox function `imcontour` to display a contour plot of the data in an intensity image. This function is similar to the `contour` function in MATLAB, but it automatically sets up the axes so their orientation and aspect ratio match the image.

This example displays an intensity image of grains of rice and a contour plot of the image data.

```
I = imread('rice.tif');
imshow(I)
figure, imcontour(I)
```

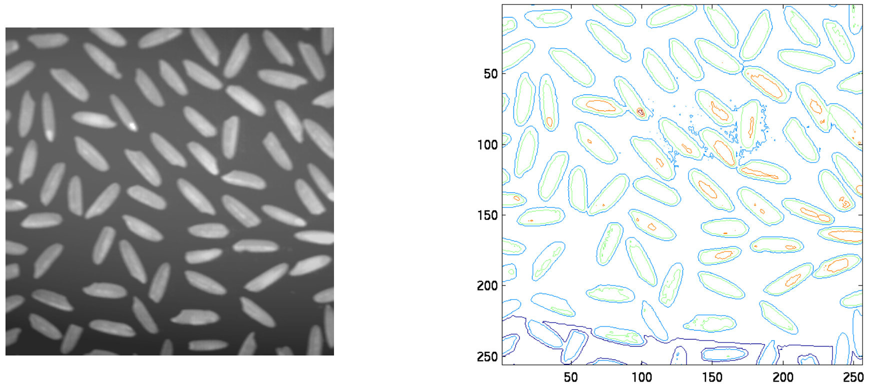


Figure 10-3: Rice.tif and Its Contour Plot

You can use the `clabel` function to label the levels of the contours. See the description of `clabel` in the MATLAB Function Reference for details.

Image Histogram

An *image histogram* is a chart that shows the distribution of intensities in an indexed or intensity image. The image histogram function `imhist` creates this plot by making `n` equally spaced bins, each representing a range of data values. It then calculates the number of pixels within each range. For example, the commands below display an image of grains of rice, and a histogram based on 64 bins.

```
I = imread('rice.tif');
imshow(I)
figure, imhist(I,64)
```

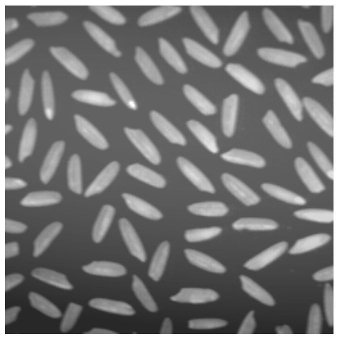
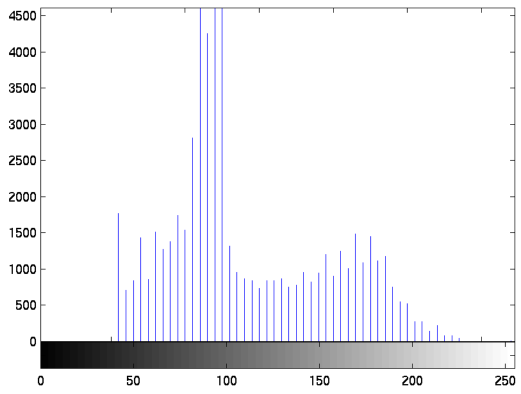



Figure 10-4: Rice.tif and Its Histogram

The histogram shows a peak at around 100, due to the dark gray background in the image. For information about how to modify an image by changing the distribution of its histogram, see “Intensity Adjustment” on page 10-14.

Summary Statistics

You can compute standard statistics of an image using the `mean2`, `std2`, and `corr2` functions. `mean2` and `std2` compute the mean and standard deviation of the elements of a matrix. `corr2` computes the correlation coefficient between two matrices of the same size.

These functions are two-dimensional versions of the `mean`, `std`, and `corrcoef` functions described in the MATLAB Function Reference.

Region Property Measurement

You can use the `regionprops` function to compute properties for image regions. For example, `regionprops` can measure such properties as the area, center of mass, and bounding box for a region you specify. See the reference page for `regionprops` for more information.

Image Analysis

Image analysis techniques return information about the structure of an image. This section describes toolbox functions that you can use for these image analysis techniques:

- Edge detection
- Quadtree decomposition

The functions described in this section work only with intensity images.

Edge Detection

You can use the `edge` function to detect edges, which are those places in an image that correspond to object boundaries. To find edges, this function looks for places in the image where the intensity changes rapidly, using one of these two criteria:

- Places where the first derivative of the intensity is larger in magnitude than some threshold
- Places where the second derivative of the intensity has a zero crossing

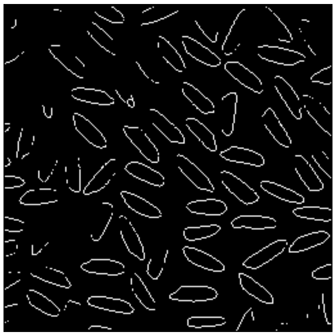
`edge` provides a number of derivative estimators, each of which implements one of the definitions above. For some of these estimators, you can specify whether the operation should be sensitive to horizontal or vertical edges, or both. `edge` returns a binary image containing 1's where edges are found and 0's elsewhere.

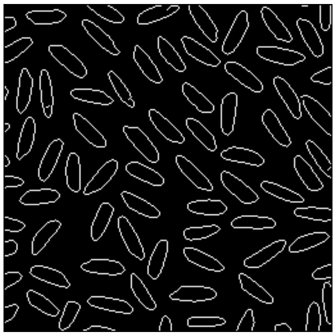
The most powerful edge-detection method that `edge` provides is the Canny method. The Canny method differs from the other edge-detection methods in that it uses two different thresholds (to detect strong and weak edges), and includes the weak edges in the output only if they are connected to strong edges. This method is therefore less likely than the others to be "fooled" by noise, and more likely to detect true weak edges.

The example below illustrates the power of the Canny edge detector. It shows the results of applying the Sobel and Canny edge detectors to the `rice.tif` image.

```
I = imread('rice.tif');
BW1 = edge(I,'sobel');
BW2 = edge(I,'canny');
imshow(BW1)
```

```
figure, imshow(BW2)
```


Sobel Filter


Canny Filter

For an interactive demonstration of edge detection, try running `edgedemo`.

Quadtree Decomposition

Quadtree decomposition is an analysis technique that involves subdividing an image into blocks that are more homogeneous than the image itself. This technique reveals information about the structure of the image. It is also useful as the first step in adaptive compression algorithms.

You can perform quadtree decomposition using the `qtdecomp` function. This function works by dividing a square image into four equal-sized square blocks, and then testing each block to see if it meets some criterion of homogeneity (e.g., if all of the pixels in the block are within a specific dynamic range). If a block meets the criterion, it is not divided any further. If it does not meet the criterion, it is subdivided again into four blocks, and the test criterion is applied to those blocks. This process is repeated iteratively until each block meets the criterion. The result may have blocks of several different sizes.

For example, suppose you want to perform quadtree decomposition on a 128-by-128 intensity image. The first step is to divide the image into four 64-by-64 blocks. You then apply the test criterion to each block; for example, the criterion might be

```
max(block(:))   min(block(:)) <= 0.2
```

If one of the blocks meets this criterion, it is not divided any further; it is 64-by-64 in the final decomposition. If a block does not meet the criterion, it is then divided into four 32-by-32 blocks, and the test is then applied to each of these blocks. The blocks that fail to meet the criterion are then divided into four 16-by-16 blocks, and so on, until all blocks "pass." Some of the blocks may be as small as 1-by-1, unless you specify otherwise.

The call to `qtdecomp` for this example would be

```
S = qtdecomp(I,0.2)
```

`S` is returned as a sparse matrix whose nonzero elements represent the upper-left corners of the blocks; the value of each nonzero element indicates the block size. `S` is the same size as `I`.

Note The threshold value is specified as a value between 0 and 1, regardless of the class of `I`. If `I` is `uint8`, the threshold value you supply is multiplied by 255 to determine the actual threshold to use; if `I` is `uint16`, the threshold value you supply is multiplied by 65535.

The example below shows an image and a representation of its quadtree decomposition. Each black square represents a homogeneous block, and the white lines represent the boundaries between blocks. Notice how the blocks are smaller in areas corresponding to large changes in intensity in the image.

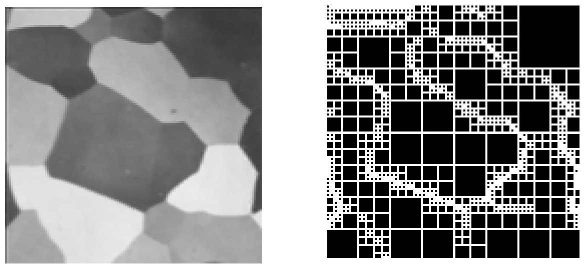


Figure 10-5: An Image (left) and a Representation of its Quadtree Decomposition

You can also supply `qtdecomp` with a function (rather than a threshold value) for deciding whether to split blocks; for example, you might base the decision on the variance of the block. See the reference page for `qtdecomp` for more information.

For an interactive demonstration of quadtree decomposition, try running `qtdemo`.

Image Enhancement

Image enhancement techniques are used to improve an image, where "improve" is sometimes defined objectively (e.g., increase the signal-to-noise ratio), and sometimes subjectively (e.g., make certain features easier to see by modifying the colors or intensities).

This section discusses these image enhancement techniques:

- "Intensity Adjustment"
- "Noise Removal"

The functions described in this section apply primarily to intensity images. However, some of these functions can be applied to color images as well. For information about how these functions work with color images, see the reference pages for the individual functions.

Intensity Adjustment

Intensity adjustment is a technique for mapping an image's intensity values to a new range. For example, `rice.tif.` is a low contrast image. The histogram of `rice.tif`, shown in Figure 10-4, indicates that there are no values below 40 or above 225. If you remap the data values to fill the entire intensity range [0, 255], you can increase the contrast of the image.

You can do this kind of adjustment with the `imadjust` function. The general syntax of `imadjust` is

```
J = imadjust(I,[low_in high_in],[low_out high_out])
```

where `low_in` and `high_in` are the intensities in the input image which are mapped to `low_out` and `high_out` in the output image. For example, this code performs the adjustment described above.

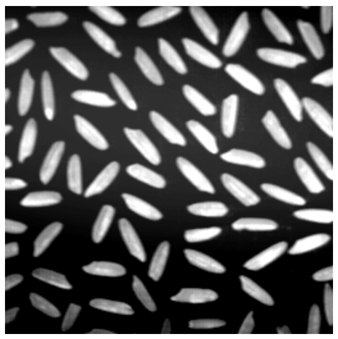
```
I = imread('rice.tif');
J = imadjust(I,[0.15 0.9],[0 1]);
```

The first vector passed to `imadjust`, `[0.15 0.9]`, specifies the low and high intensity values that you want to map. The second vector, `[0 1]`, specifies the scale over which you want to map them. Thus, the example maps the intensity value 0.15 in the input image to 0 in the output image, and 0.9 to 1.

Note that you must specify the intensities as values between 0 and 1 regardless of the class of `I`. If `I` is `uint8`, the values you supply are multiplied by 255 to determine the actual values to use; if `I` is `uint16`, the values are multiplied by 65535. To learn about an alternative way to set this limits automatically, see “Setting the Adjustment Limits Automatically” on page 10-16.

This figure displays the adjusted image and its histogram. Notice the increased contrast in the image, and that the histogram now fills the entire range.

```
imshow(J)
figure, imhist(J,64)
```



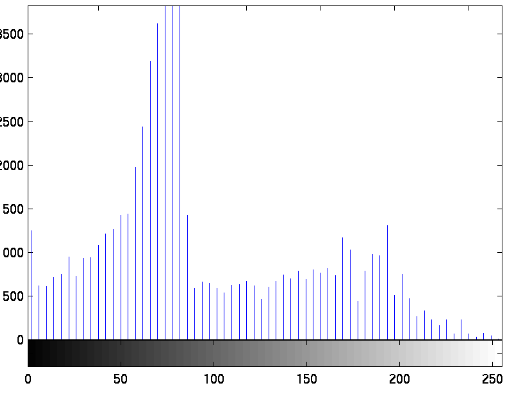


Figure 10-6: Rice.tif After an Intensity Adjustment and a Histogram of Its Adjusted Intensities

Similarly, you can decrease the contrast of an image by narrowing the range of the data, as in this call.

```
J = imadjust(I,[0 1],[0.3 0.8]);
```

In addition to increasing or decreasing contrast, you can perform a wide variety of other image enhancements with `imadjust`. In the example below, the man’s coat is too dark to reveal any detail. The call to `imadjust` maps the range [0,51] in the `uint8` input image to [128,255] in the output image. This brightens the image considerably, and also widens the dynamic range of the dark portions of the original image, making it much easier to see the details in the coat. Note, however, that because all values above 51 in the original image get mapped to 255 (white) in the adjusted image, the adjusted image appears “washed out.”

```
I = imread('cameraman.tif');
J = imadjust(I,[0 0.2],[0.5 1]);
imshow(I)
figure, imshow(J)
```

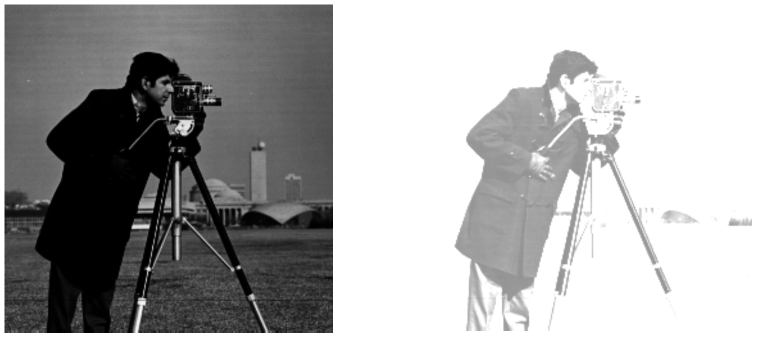


Figure 10-7: Remapping and Widening the Dynamic Range

Setting the Adjustment Limits Automatically

To use `imadjust`, you must typically perform two steps:

1. View the histogram of the image to determine the intensity value limits.
2. Specify these limits as a fraction between 0.0 and 1.0 so that you can pass them to `imadjust` in the `[low_in high_in]` vector.

For a more convenient way to specify these limits, use the `stretchlim` function. This function calculates the histogram of the image and determines the adjustment limits automatically. The `stretchlim` function returns these values as fractions in a vector that you can pass as the `[low_in high_in]` argument to `imadjust`; for example,

```
I = imread('rice.tif');
J = imadjust(I,stretchlim(I),[0 1]);
```

By default, `stretchlim` uses the intensity values that represent the bottom 1% (0.01) and the top 1% (0.99) of the range as the adjustment limits. By trimming the extremes at both ends of the intensity range, `stretchlim` makes more room in the adjusted dynamic range for the remaining intensities. But you can specify other range limits as an argument to `stretchlim`. See the `stretchlim` reference page for more information.

Gamma Correction

`imadjust` maps `low` to `bottom`, and `high` to `top`. By default, the values between `low` and `high` are mapped linearly to values between `bottom` and `top`. For example, the value halfway between `low` and `high` corresponds to the value halfway between `bottom` and `top`.

`imadjust` can accept an additional argument which specifies the *gamma correction* factor. Depending on the value of gamma, the mapping between values in the input and output images may be nonlinear. For example, the value halfway between `low` and `high` may map to a value either greater than or less than the value halfway between `bottom` and `top`.

Gamma can be any value between 0 and infinity. If gamma is 1 (the default), the mapping is linear. If gamma is less than 1, the mapping is weighted toward higher (brighter) output values. If gamma is greater than 1, the mapping is weighted toward lower (darker) output values.

The figure below illustrates this relationship. The three transformation curves show how values are mapped when gamma is less than, equal to, and greater than 1. (In each graph, the *x*-axis represents the intensity values in the input image, and the *y*-axis represents the intensity values in the output image.)

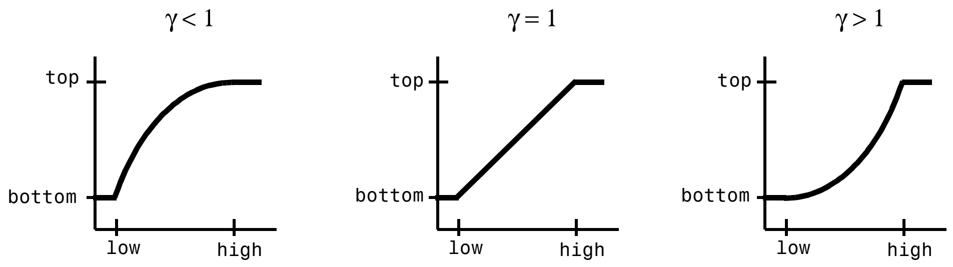


Figure 10-8: Plots Showing Three Different Gamma Correction Settings

The example below illustrates gamma correction. Notice that in the call to `imadjust`, the data ranges of the input and output images are specified as empty matrices. When you specify an empty matrix, `imadjust` uses the default range of [0,1]. In the example, both ranges are left empty; this means that gamma correction is applied without any other adjustment of the data.

```
[X,map] = imread('forest.tif')
I = ind2gray(X,map);
J = imadjust(I,[],[],0.5);
imshow(I)
figure, imshow(J)
```

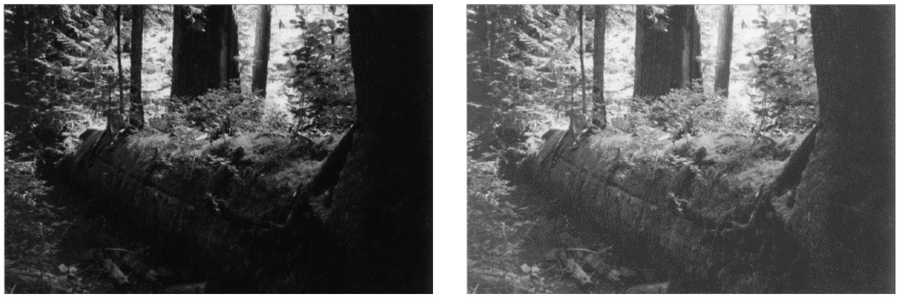


Figure 10-9: Forest.tif Before and After Applying Gamma Correction of 0.5

Histogram Equalization

The process of adjusting intensity values can be done automatically by the `histeq` function. `histeq` performs *histogram equalization,* which involves transforming the intensity values so that the histogram of the output image approximately matches a specified histogram. (By default, `histeq` tries to match a flat histogram with 64 bins, but you can specify a different histogram instead; see the reference page for `histeq`.)

This example illustrates using `histeq` to adjust an intensity image. The original image has low contrast, with most values in the middle of the intensity range. `histeq` produces an output image having values evenly distributed throughout the range.

```
I = imread('pout.tif');
J = histeq(I);
imshow(I)
figure, imshow(J)
```

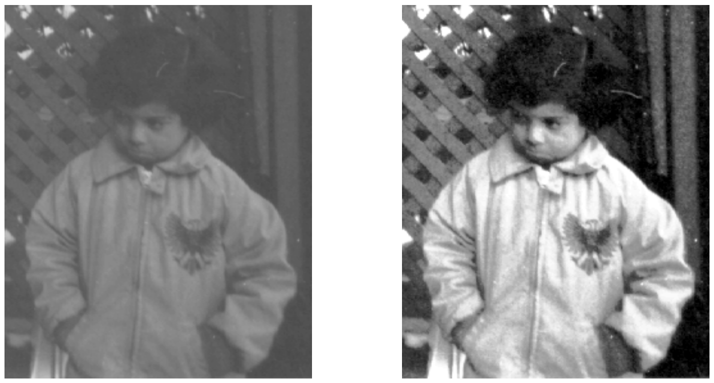


Figure 10-10: Pout.tif Before and After Histogram Equalization

The example below shows the histograms for the two images.

```
figure, imhist(I)
figure, imhist(J)
```

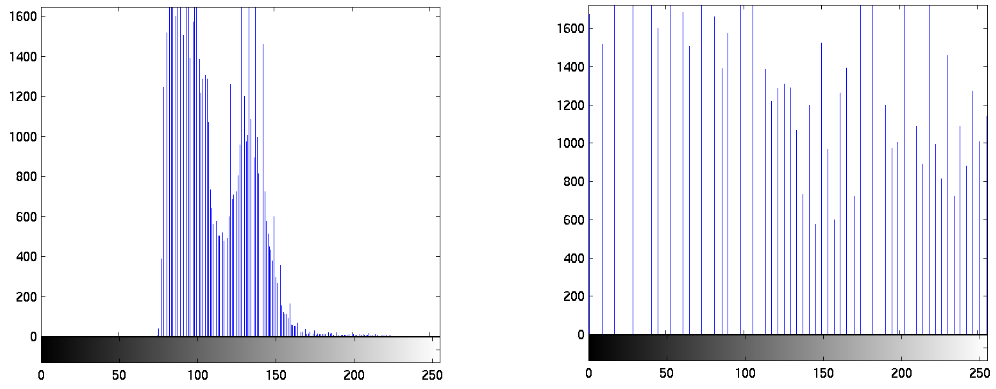
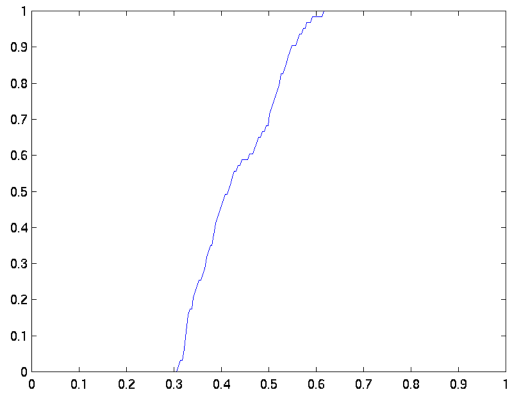


Figure 10-11: Histogram Before Equalization (left) and After Equalization (right)

`histeq` can return an additional 1-by-256 vector that shows, for each possible input value, the resulting output value. (The values in this vector are in the range [0,1], regardless of the class of the input image.) You can plot this data to get the transformation curve. For example,

```
I = imread('pout.tif');
[J,T] = histeq(I);
figure,plot((0:255)/255,T);
```



Notice how this curve reflects the histograms in the previous figure, with the input values being mostly between 0.3 and 0.6, while the output values are distributed evenly between 0 and 1.

For an interactive demonstration of intensity adjustment, try running `imadjdemo`.

Noise Removal

Digital images are prone to a variety of types of noise. There are several ways that noise can be introduced into an image, depending on how the image is created. For example:

- If the image is scanned from a photograph made on film, the film grain is a source of noise. Noise can also be the result of damage to the film, or be introduced by the scanner itself.
- If the image is acquired directly in a digital format, the mechanism for gathering the data (such as a CCD detector) can introduce noise.

- Electronic transmission of image data can introduce noise.

The toolbox provides a number of different ways to remove or reduce noise in an image. Different methods are better for different kinds of noise. The methods available include:

- Linear filtering
- Median filtering
- Adaptive filtering

Also, in order to simulate the effects of some of the problems listed above, the toolbox provides the `imnoise` function, which you can use to *add* various types of noise to an image. The examples in this section use this function.

Linear Filtering

You can use linear filtering to remove certain types of noise. Certain filters, such as averaging or Gaussian filters, are appropriate for this purpose. For example, an averaging filter is useful for removing grain noise from a photograph. Because each pixel gets set to the average of the pixels in its neighborhood, local variations caused by grain are reduced.

See “Linear Filtering” on page 7-4 for more information.

Median Filtering

Median filtering is similar to using an averaging filter, in that each output pixel is set to an “average” of the pixel values in the neighborhood of the corresponding input pixel. However, with median filtering, the value of an output pixel is determined by the *median* of the neighborhood pixels, rather than the mean. The median is much less sensitive than the mean to extreme values (called *outliers*). Median filtering is therefore better able to remove these outliers without reducing the sharpness of the image.

The `medfilt2` function implements median filtering. The example below compares using an averaging filter and `medfilt2` to remove *salt and pepper* noise. This type of noise consists of random pixels being set to black or white (the extremes of the data range). In both cases the size of the neighborhood used for filtering is 3-by-3.

First, read in the image and add noise to it.

```
I = imread('eight.tif');
```

```
J = imnoise(I,'salt & pepper',0.02);
imshow(I)
figure, imshow(J)
```

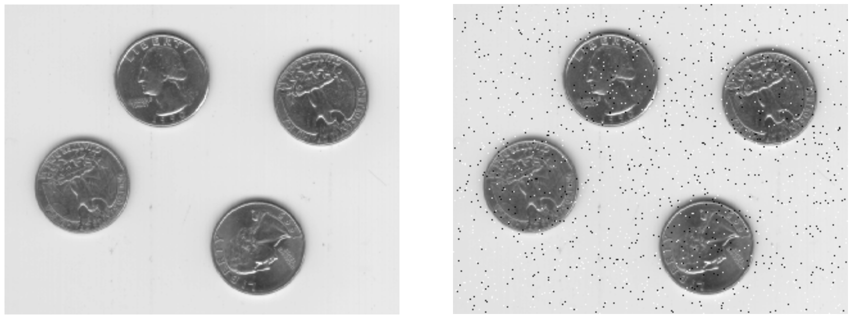


Figure 10-12: Eight.tif Before and After Adding Salt-and-Pepper Noise

Now filter the noisy image and display the results. Notice that `medfilt2` does a better job of removing noise, with less blurring of edges.

```
K = filter2(fspecial('average',3),J)/255;
L = medfilt2(J,[3 3]);
figure, imshow(K)
figure, imshow(L)
```

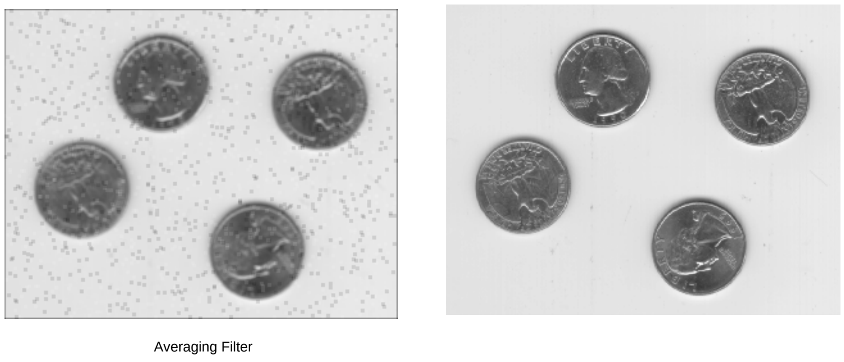


Figure 10-13: Noisy Version of Eight.tif Filtered with Averaging Filter (left) and Median Filter (right)

Median filtering is a specific case of *order-statistic filtering*, also known as *rank filtering*. For information about order-statistic filtering, see the reference page for the `ordfilt2` function.

Adaptive Filtering

The `wiener2` function applies a Wiener filter (a type of linear filter) to an image *adaptively,* tailoring itself to the local image variance. Where the variance is large, `wiener2` performs little smoothing. Where the variance is small, `wiener2` performs more smoothing.

This approach often produces better results than linear filtering. The adaptive filter is more selective than a comparable linear filter, preserving edges and other high frequency parts of an image. In addition, there are no design tasks; the `wiener2` function handles all preliminary computations, and implements the filter for an input image. `wiener2`, however, does require more computation time than linear filtering.

`wiener2` works best when the noise is constant-power ("white") additive noise, such as Gaussian noise. The example below applies `wiener2` to an image of Saturn that has had Gaussian noise added.

```
I = imread('saturn.tif');
J = imnoise(I,'gaussian',0,0.005);
K = wiener2(J,[5 5]);
imshow(J)
figure, imshow(K)
```

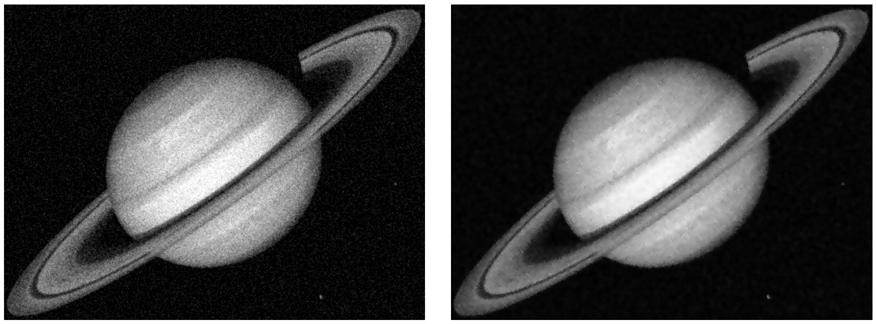


Figure 10-14: Noisy Version of Saturn.tif Before and After Adaptive Filtering

For an interactive demonstration of filtering to remove noise, try running `nrfiltdemo`.

Linear Filtering

Filtering is a technique for modifying or enhancing an image. For example, you can filter an image to emphasize certain features or remove other features.

Filtering is a *neighborhood operation,* in which the value of any given pixel in the output image is determined by applying some algorithm to the values of the pixels in the neighborhood of the corresponding input pixel. A pixel's neighborhood is some set of pixels, defined by their locations relative to that pixel. (See Chapter 6, "Neighborhood and Block Operations", for a general discussion of neighborhood operations.)

Linear filtering is filtering in which the value of an output pixel is a linear combination of the values of the pixels in the input pixel's neighborhood.

This section discusses linear filtering in MATLAB and the Image Processing Toolbox. It includes:

- A description of filtering, using convolution and correlation
- A description of how to use the `imfilter` function to perform filtering
- A discussion about using predefined filter types

See "Filter Design" on page 7-16 for information about how to design filters.

Convolution

Linear filtering of an image is accomplished through an operation called *convolution*. In convolution, the value of an output pixel is computed as a weighted sum of neighboring pixels. The matrix of weights is called the *convolution kernel*, also known as the *filter*.

For example, suppose the image is

```
A = [17  24   1   8  15
     23   5   7  14  16
      4   6  13  20  22
     10  12  19  21   3
     11  18  25   2   9]
```

and the convolution kernel is

```
h = [8    1    6
     3    5    7
     4    9    2]
```

Then Figure 7-1 shows how to compute the (2,4) output pixel using these steps:

1 Rotate the convolution kernel 180 degrees about its center element.

2 Slide the center element of the convolution kernel so that lies on top of the (2,4) element of A.

3 Multiply each weight in the rotated convolution kernel by the pixel of A underneath.

4 Sum up the individual products from step 3.

Hence the (2,4) output pixel is

$$1 \cdot 2 + 8 \cdot 9 + 15 \cdot 4 + 7 \cdot 7 + 14 \cdot 5 + 16 \cdot 3 + 13 \cdot 6 + 20 \cdot 1 + 22 \cdot 8 = 575$$

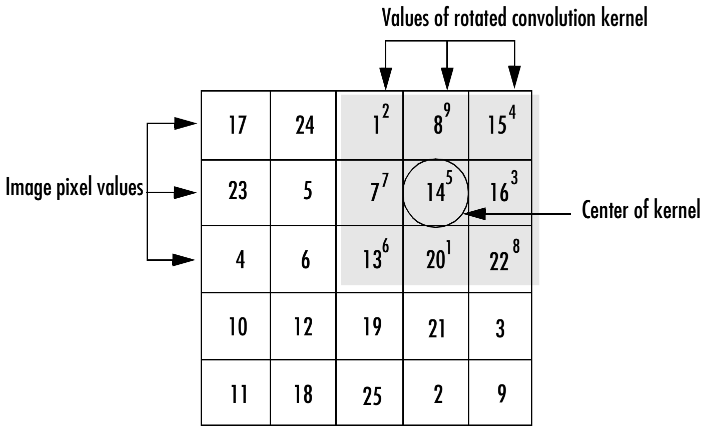


Figure 7-1: Computing the (2,4) Output of Convolution

Correlation

The operation called *correlation* is closely related to convolution. In correlation, the value of an output pixel is also computed as a weighted sum of neighboring pixels. The difference is that the matrix of weights, in this case called the *correlation kernel*, is not rotated during the computation. Figure 7-2 shows how to compute the (2,4) output pixel of the correlation of A, assuming h is correlation kernel instead of a convolution kernel, using these steps:

1. Slide the center element of the correlation kernel so that lies on top of the (2,4) element of A.
2. Multiply each weight in the correlation kernel by the pixel of A underneath.
3. Sum up the individual products from step 3.

The (2,4) output pixel from the correlation is

$$1 \cdot 8 + 8 \cdot 1 + 15 \cdot 6 + 7 \cdot 3 + 14 \cdot 5 + 16 \cdot 7 + 13 \cdot 4 + 20 \cdot 9 + 22 \cdot 2 = 585$$

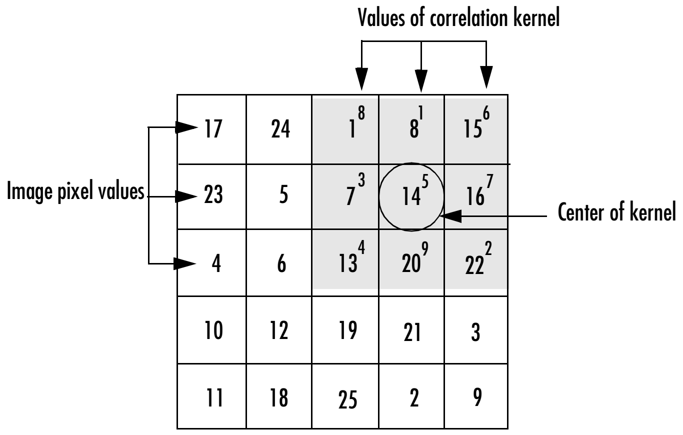
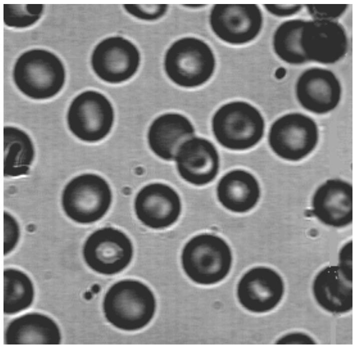


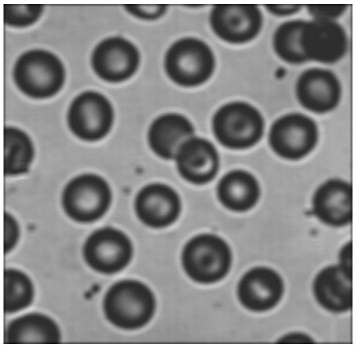
Figure 7-2: Computing the (2,4) Output of Correlation

Filtering Using imfilter

Filtering of images, either by correlation or convolution, can be performed using the toolbox function `imfilter`. This example filters the image in the file `blood1.tif` with a 5-by-5 filter containing equal weights. Such a filter is often called an *averaging filter*.

```
I = imread('blood1.tif');
h = ones(5,5) / 25;
I2 = imfilter(I,h);
imshow(I), title('Original image')
figure, imshow(I2), title('Filtered image')
```


Original Image


Filtered Image

Data Types

The `imfilter` function handles data types similar to the way the image arithmetic functions do, as described in "Image Arithmetic Truncation Rules" on page 2-22. The output image has the same data type, or numeric class, as the input image. The `imfilter` function computes the value of each output pixel using double-precision, floating-point arithmetic. If the result exceeds the range of the data type, the `imfilter` function truncates the result to that data type's allowed range. If it is an integer data type, `imfilter` rounds fractional values.

Because of the truncation behavior, you may sometimes want to consider converting your image to a different data type before calling `imfilter`. In this example, the output of `imfilter` has negative values when the input is of class `double`.

```
A = magic(5)

A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

h = [-1 0 1]

h =
    -1     0     1

imfilter(A,h)

ans =
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2
```

Notice that the result has negative values. Now suppose A was of class uint8, instead of double.

```
A = uint8(magic(5));
imfilter(A,h)

ans =

    24     0     0    14     0
     5     0     9     9     0
     6     9    14     9     0
    12     9     9     0     0
    18    14     0     0     0
```

Since the input to imfilter is of class uint8, the output also is of class uint8, and so the negative values are truncated to 0. In such cases, it may be appropriate to convert the image to another type, such as a signed integer type, single, or double, before calling imfilter.

Correlation and Convolution Options

The imfilter function can perform filtering using either correlation or convolution. It uses correlation by default, because the filter design functions, described in “Filter Design” on page 7-16, and the fspecial function, described in “Using Predefined Filter Types” on page 7-14, produce correlation kernels.

However, if you want to perform filtering using convolution instead, you can pass the string 'conv' as optional input argument to imfilter. For example,

```
A = magic(5);
h = [-1 0 1]
imfilter(A,h)   % filter using correlation

ans =
    24   -16   -16    14    -8
     5   -16     9     9   -14
     6     9    14     9   -20
    12     9     9   -16   -21
    18    14   -16   -16    -2

imfilter(A,h,'conv')   % filter using convolution

ans =

   -24    16    16   -14     8
    -5    16    -9    -9    14
    -6    -9   -14    -9    20
   -12    -9    -9    16    21
   -18   -14    16    16     2
```

Boundary Padding Options

When computing an output pixel at the boundary of an image, a portion of the convolution or correlation kernel is usually off the edge of the image, as illustrated in the figure below.

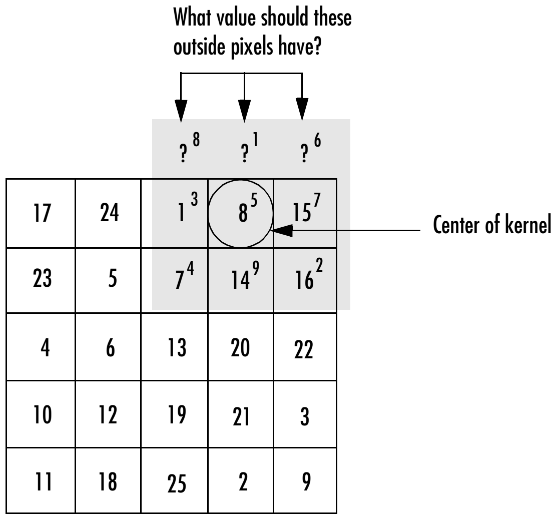


Figure 7-3: When the Values of the Kernel Fall Outside the Image

The `imfilter` function normally fills in these "off-the-edge" image pixels by assuming that they are `0`. This is called zero-padding and is illustrated in the figure below.

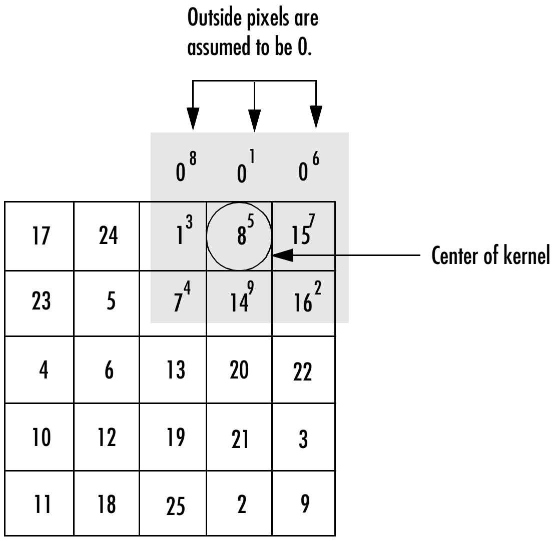
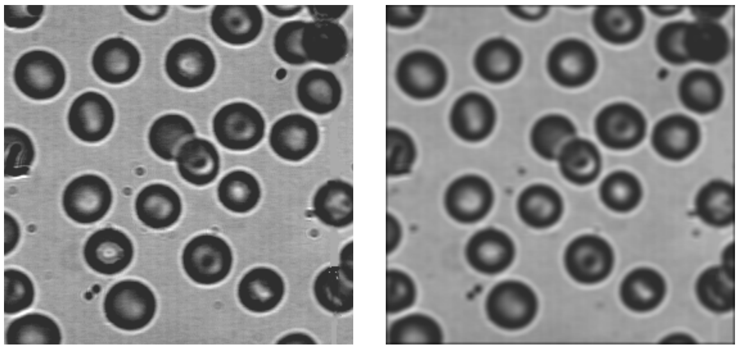


Figure 7-4: Zero-Padding of Outside Pixels

When filtering an image, zero-padding can result in a dark band around the edge of the image, as shown in this example.

```
I = imread('blood1.tif');
h = ones(5,5)/25;
I2 = imfilter(I,h);
imshow(I), title('Original image')
figure, imshow(I2), title('Filtered image')
```



To eliminate the zero-padding artifacts around the edge of the image, `imfilter` offers an alternative boundary padding method called *border replication*. In border replication, the value of any pixel outside the image is determined by replicating the value from the nearest border pixel. This is illustrated in the figure below.

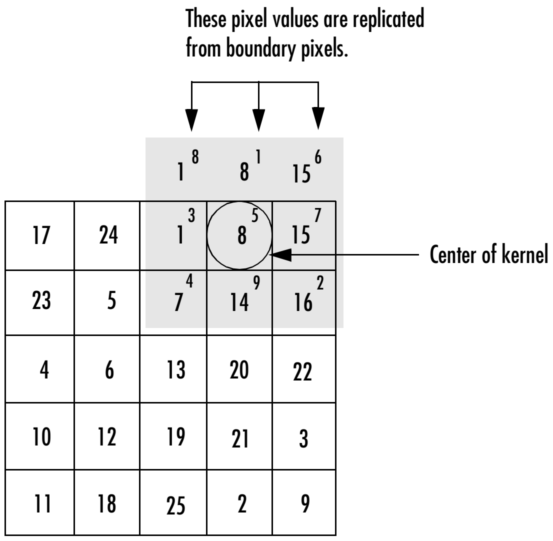


Figure 7-5: Replicated Boundary Pixels

To filter using border replication, pass the additional optional argument `'replicate'` to `imfilter`.

```
I3 = imfilter(I,h,'replicate');
figure, imshow(I3), title('Filtered with border replication')
```

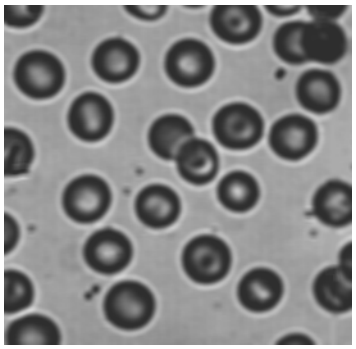


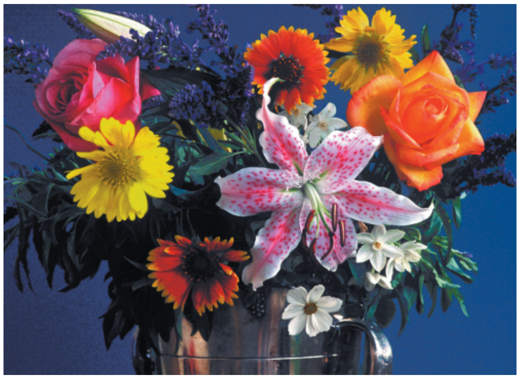
Image Border with Replication

The `imfilter` function supports other boundary padding options, such as `'circular'` and `'symmetric'`. See the reference page for `imfilter` for details.

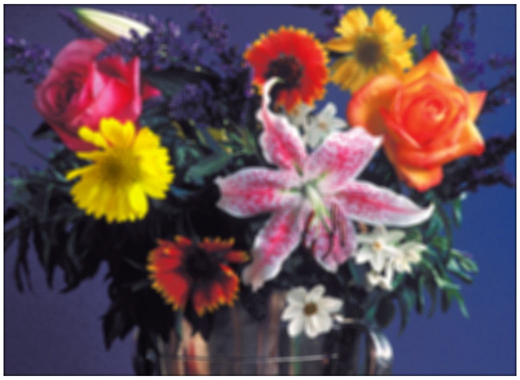
Multidimensional Filtering

The `imfilter` function can handle both multidimensional images and multidimensional filters. A convenient property of filtering is that filtering a three-dimensional image with a two-dimensional filter is equivalent to filtering each plane of the three-dimensional image individually with the same two-dimensional filter. This property makes it easy, for example, to filter each color plane of a truecolor image with the same filter.

```
rgb = imread('flowers.tif');
h = ones(5,5) / 25;
rgb2 = imfilter(rgb,h);
imshow(rgb), title('Original image')
figure, imshow(rgb2), title('Filtered image')
```



Original Image



Filtered Image

Relationship to Other Filtering Functions

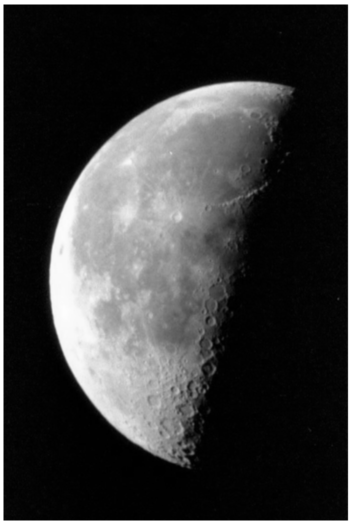
MATLAB has several two-dimensional and multidimensional filtering functions. The function `filter2` performs two-dimensional correlation; `conv2` performs two-dimensional convolution; and `convn` performs multidimensional convolution. Each of these other filtering functions always converts the input to `double`, and the output is always `double`. Also, each of these other filtering functions always assumes the input is zero-padded, and they do not support other padding options.

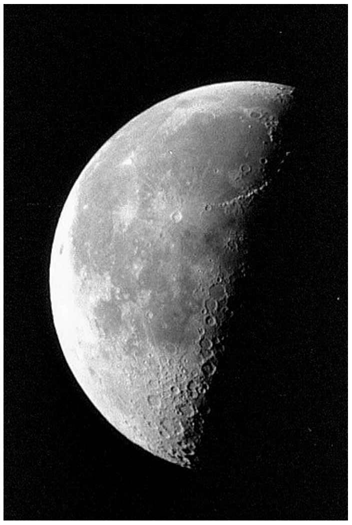
In contrast, the `imfilter` function does not convert input images to `double`. The `imfilter` function also offers a flexible set of boundary padding options, as described in “Boundary Padding Options” on page 7-10.

Using Predefined Filter Types

The `fspecial` function produces several kinds of predefined filters, in the form of correlation kernels. After creating a filter with `fspecial`, you can apply it directly to your image data using `imfilter`. This example illustrates applying an *unsharp masking* filter to an intensity image. The unsharp masking filter has the effect of making edges and fine detail in the image more crisp.

```
I = imread('moon.tif');
h = fspecial('unsharp');
I2 = imfilter(I,h);
imshow(I), title('Original image')
figure, imshow(I2), title('Filtered image')
```


Original Image


Filtered Image