CHAPTER 4

THE IMAGE PROCESSING TOOLBOX AT A GLANCE

WHAT WILL WE LEARN?

- How do I read an image from a file using MATLAB?
- What are the main data classes used for image representation and how can I convert from one to another?
- Why is it important to understand the data class and range of pixel values of images stored in memory?
- How do I display an image using MATLAB?
- How can I explore the pixel contents of an image?
- How do I save an image to a file using MATLAB?

4.1 THE IMAGE PROCESSING TOOLBOX: AN OVERVIEW

The Image Processing Toolbox (IPT) is a collection of functions that extend the basic capability of the MATLAB environment to enable specialized signal and image processing operations, such as the following:

- Spatial transformations (Chapter 7)
- Image analysis and enhancement (Chapters 8–12, 14–15, and 18)

Practical Image and Video Processing Using MATLAB®. By Oge Marques.

- Neighborhood and block operations (Chapter 10)
- Linear filtering and filter design (Chapters 10 and 11)
- Mathematical transforms (Chapters 11 and 17)
- Deblurring (Chapter 12)
- Morphological operations (Chapter 13)
- Color image processing (Chapter 16)

Most of the toolbox functions are MATLAB M-files, whose source code can be inspected by opening the file using the MATLAB editor or simply typing `type function_name` at the prompt. The directory in which the M-file has been saved can be found by typing `which function_name` at the prompt. To determine which version of the IPT is installed on your system, type `ver` at the prompt.

You can extend the capabilities of the IPT by writing your own M-files, modifying and expanding existing M-files, writing wrapping functions around existing ones, or using the IPT in combination with other toolboxes.

4.2 ESSENTIAL FUNCTIONS AND FEATURES

This section presents the IPT functions used to perform essential image operations, such as reading the image contents from a file, converting between different data classes used to store the pixel values, displaying an image on the screen, and saving an image to a file.

4.2.1 Displaying Information About an Image File

MATLAB's IPT has a built-in function to display information about image files (without opening them and storing their contents in the workspace) `imfinfo`.

■ EXAMPLE 4.1

The MATLAB code below reads information about a built-in image file `pout.tif`.

```
imfinfo('pout.tif');
```

The resulting structure (stored in variable `ans`) will contain the following information:[1]

```
    Filename: '/.../pout.tif'
FileModDate: '04-Dec-2000 13:57:50'
    FileSize: 69004
      Format: 'tif'
```

[1] All empty fields have been omitted for space reasons.

```
                    Width: 240
                   Height: 291
                 BitDepth: 8
                ColorType: 'grayscale'
          FormatSignature: [73 73 42 0]
                ByteOrder: 'little-endian'
           NewSubFileType: 0
            BitsPerSample: 8
              Compression: 'PackBits'
PhotometricInterpretation: 'BlackIsZero'
             StripOffsets: [9x1 double]
          SamplesPerPixel: 1
             RowsPerStrip: 34
          StripByteCounts: [9x1 double]
              XResolution: 72
              YResolution: 72
           ResolutionUnit: 'Inch'
      PlanarConfiguration: 'Chunky'
              Orientation: 1
                FillOrder: 1
         GrayResponseUnit: 0.0100
           MaxSampleValue: 255
           MinSampleValue: 0
             Thresholding: 1
```

Many of these fields are too technical and some are file format dependent. Nonetheless, you should still be able to locate information about the image size (240 × 291 pixels), the file size (69,004 bytes), the type of image (grayscale), the number of bits per pixel (8), and its minimum and maximum values (0 and 255).

Repeating the procedure for the built-in image `coins.png` will produce the following results:[2]

```
       Filename: '/.../coins.png'
    FileModDate: '16-Apr-2003 02:05:30'
       FileSize: 37906
         Format: 'png'
          Width: 300
         Height: 246
       BitDepth: 8
      ColorType: 'grayscale'
FormatSignature: [137 80 78 71 13 10 26 10]
  InterlaceType: 'none'
```

[2]All empty fields have been omitted for space reasons.

```
   Transparency: 'none'
   ImageModTime: '19 Sep 2002 20:31:12 +0000'
      Copyright: 'Copyright The MathWorks, Inc.'
```

Once again, you should be able to find basic information about the image, regardless of the several (potentially obscure) fields that are file format dependent. You may also have noticed that several fields have changed between the first and second files as a consequence of the specific file formats (TIFF and PNG, respectively).

Finally, let us repeat this command for a truecolor (24 bits per pixel) image `peppers.png`. The result will be[3]

```
       Filename: '/.../peppers.png'
    FileModDate: '16-Dec-2002 06:10:58'
       FileSize: 287677
         Format: 'png'
          Width: 512
         Height: 384
       BitDepth: 24
      ColorType: 'truecolor'
FormatSignature: [137 80 78 71 13 10 26 10]
  InterlaceType: 'none'
   Transparency: 'none'
   ImageModTime: '16 Jul 2002 16:46:41 +0000'
    Description: 'Zesty peppers'
      Copyright: 'Copyright The MathWorks, Inc.'
```

4.2.2 Reading an Image File

MATLAB's IPT has a built-in function to open and read the contents of image files in most popular formats (e.g., TIFF, JPEG, BMP, GIF, and PNG), `imread`.

The `imread` function allows you to read image files of almost any type, in virtually any format, located anywhere. This saves you from a (potentially large) number of problems associated with file headers, memory allocation, file format conventions, and so on, and allows to focus on what you want to do to the image once it has been read and stored into a variable in the MATLAB workspace.

The IPT also contains specialized functions for reading DICOM (Digital Imaging and Communications in Medicine) files (`dicomread`), NITF (National Imagery Transmission Format) files (`nitfread`), and HDR (high dynamic range) files (`hdrread`).

[3] All empty fields have been omitted for space reasons.

TABLE 4.1 IPT Functions to Perform Image Data Class Conversion

Name	Converts an Image to Data Class
`im2single`	`single`
`im2double`	`double`
`im2uint8`	`uint8`
`im2uint16`	`uint16`
`im2int16`	`int16`

4.2.3 Data Classes and Data Conversions

The IPT ability to read images of any type, store their contents into arrays, and make them available for further processing and display does not preclude the need to understand how the image contents are represented in memory. This section follows up on our discussion in Section 3.2.2 and explains the issue of data classes, data conversions, and when they may be needed.

The most common data classes for images are as follows:

- `uint8`: 1 byte per pixel, in the [0, 255] range
- `double`: 8 bytes per pixel, usually in the [0.0, 1.0] range
- `logical`: 1 byte per pixel, representing its value as *true* (1 or white) or *false* (0 or black)

Once the contents of an image have been read and stored into one or more variables, you are encouraged to inspect the data class of these variables and their range of values to understand *how* the pixel contents are represented and what is their allowed range of values. You should ensure that the data class of the variable is compatible with the input data class expected by the IPT functions that will be applied to that variable. If you are writing your own functions and scripts, you must also ensure data class compatibility, or perform the necessary conversions.[4]

MATLAB allows data class conversion (*typecasting*) to be done in a straightforward way, but this type of conversion does not handle the range problem and is usually *not* what you want to do. To convert an image (or an arbitrary array for that matter) to a data class and range suitable for image processing, you are encouraged to use one of the specialized functions listed in Table 4.1. The input data class for any of those functions can be `logical`, `uint8`, `uint16`, `int16`, `single`, or `double`.

[4]Some of the problems that may arise by overlooking these issues will only be fully appreciated once you acquire some hands-on experience with the IPT. They include (but are not limited to) the following: unwillingly truncating intermediate calculation results (e.g., by using an unsigned data class that does not allow negative values), setting wrong thresholds for pixel values (e.g., by assuming that the pixels range from 0 to 255, when in fact they range from 0.0 to 1.0), and masking out some of the problems by displaying the image contents using the *scale for display purposes* option. These issues will be referred to occasionally throughout the book.

The IPT also contains other functions (not listed in Table 4.1) to convert the following:

- An image (of data class `uint8`, `uint16`, `single`, `int16`, or `double`) to a binary image (of data class `logical`) based on threshold: `im2bw`. Note that this is not simply a type (data class) conversion, but instead an implementation of a global thresholding algorithm, which will be presented in Chapter 15.
- An image to an instance of the Java image class `java.awt.Image`: `im2java`.
- An image to an instance of the Java image class `java.awt.image.BufferedImage`: `im2java2d`.
- An image to a movie frame, `im2frame`, which will be discussed—together with related functions such as `frame2im` and `movie`—in Part II of the book.
- A matrix to a grayscale image, `mat2gray`, where the input can be `logical` or any numeric class, and the output is an array of data class `double` (with values within the [0.0, 1.0] range).

■ EXAMPLE 4.2

It is possible to convert a generic 2×2 array of class `double` (A) into its `uint8` equivalent (B) by simply typecasting it.

```
A =

   -8.0000    4.0000
         0    0.5000

>> B = uint8(A)

B =

    0    4
    0    1
```

As you may have noticed, the conversion consisted of truncation (all negative values became zero) and rounding off (0.5 became 1).

Using `im2uint8` will lead to results within a normalized range ([0, 255])

```
>> C = im2uint8(A)

C =

    0  255
    0  128
```

from which we can infer that it treated the original values as if they were in the [0, 1] range for data class `double`, that is, 0.5 became 128 (midrange point), anything less than or equal to 0 was truncated to 0, and anything greater than or equal to 1 was truncated to 255.

Let us now apply `mat2gray` to A and inspect the results:

```
>> D = mat2gray(A)

D =

         0    1.0000
    0.6667    0.7083
```

This result illustrates an important point: since A was already of data class `double`, there was no data class convention *per se*, but simply a range conversion—the smallest value (−8.0) became 0.0, the largest value (4.0) became 1.0, and all intermediate values were scaled within the new range.

Finally, let us use `im2bw` to convert our images to binary equivalents.

```
>> E = im2bw(D, 0.4)

E =

     0     1
     1     1
```

The results work as expected: any value greater than 0.4 becomes 1 (true), otherwise it becomes 0 (false). Note that since `im2bw` expects a threshold luminance level as a parameter, and requires it to be a nonnegative number between 0 and 1, it *implicitly* assumes that the input variable (*D*, in this case) is a normalized grayscale image of class `double`. In other words, if you try to use A as an input image (`E = im2bw(A, 0.4)`), the function will work without any error or warning, but the result may not make sense.

The IPT also includes functions to convert between RGB (truecolor), indexed image, and grayscale image, which are listed in Table 4.2. In Tutorial 4.2 (page 74), you will have a chance to experiment with some of these functions.

TABLE 4.2 IPT Functions to Perform Image Data Class Conversion

Name	Description	
	Converts	Into
`ind2gray`	An indexed image	Its grayscale equivalent
`gray2ind`	A grayscale image	An indexed representation
`rgb2gray`	An RGB (truecolor) image	Its grayscale equivalent
`rgb2ind`	An RGB (truecolor) image	An indexed representation
`ind2rgb`	An indexed color image	Its RGB (truecolor) equivalent

4.2.4 Displaying the Contents of an Image

MATLAB has several functions for displaying images:

- `image`: displays an image using the current color map.[5]
- `imagesc`: scales image data to the full range of the current color map and displays the image.
- `imshow`: displays an image and contains a number of optimizations and optional parameters for property settings associated with image display.
- `imtool`: displays an image and contains a number of associated tools that can be used to explore the image contents.

■ EXAMPLE 4.3

The following MATLAB code is used to open an image file and display it using different `imshow` options:

```
I = imread('pout.tif');
imshow(I)
figure, imshow(I,[])
figure, imshow(I,[100 160])
```

The first call to `imshow` displays the image in its original state. The following line opens a new figure and displays a scaled (for display purposes) version of the same image.[6] The last line specifies a range of gray levels, such that all values lower than 100 will be displayed as black and all values greater than 160 will be displayed as white. The three results are shown side by side in Figure 4.1.

[5]The color map array is an M × 3 matrix of class *double*, where each element is a floating-point value in the range [0, 1]. Each row in the color map represents the R (red), G (green), and B (blue) values for that particular row.

[6]The fact that the result is a much improved version of the original suggests that the image could be improved by image processing techniques, such as *histogram equalization* or *histogram stretching*, which will be described in Chapter 9.

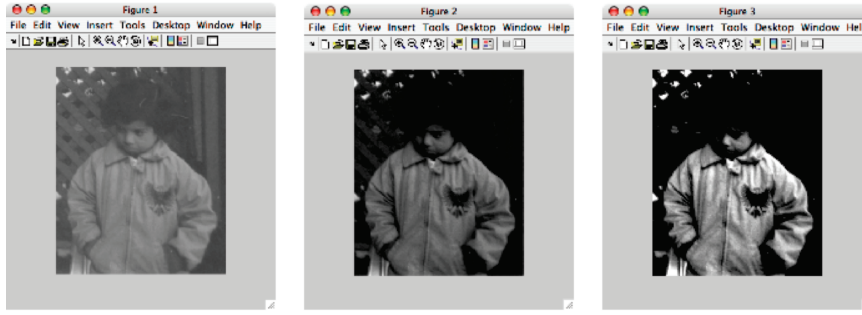

FIGURE 4.1 Displaying an image: (a) without scaling; (b) scaling for display purposes; (c) selecting only pixels within a specified range. Original image: courtesy of MathWorks.

4.2.5 Exploring the Contents of an Image

Image processing researchers and practitioners often need to inspect the contents of an image more closely. In MATLAB, this is usually done using the `imtool` function, which provides all the image display capabilities of `imshow` as well as access to other tools for navigating and exploring images, such as the *Pixel Region* tool (Figure 4.2), *Image Information* tool (Figure 4.3), and *Adjust Contrast* tool (Figure 4.4). These tools can also be directly accessed using their library functions `impixelinfo`, `imageinfo`, and `imcontrast`, respectively.

Before `imtool` was introduced, the syntax `imshow(I), pixval on` was used to enable the display of coordinates and values of a moused-over pixel.

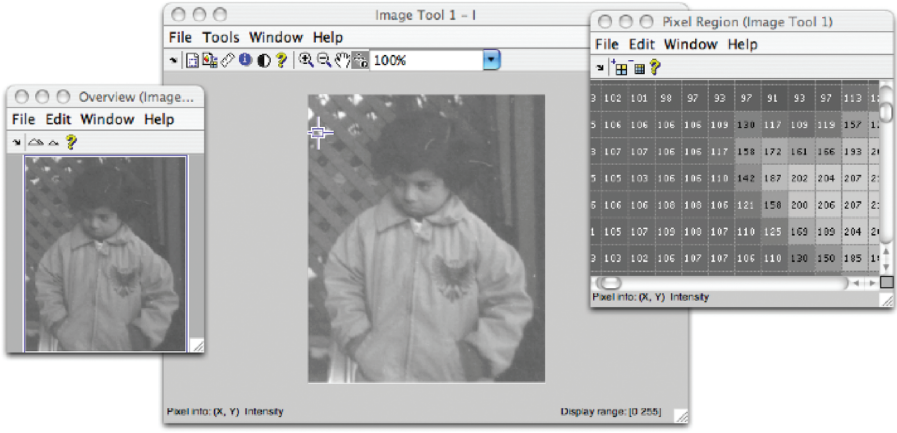

FIGURE 4.2 Displaying an image and exploring its contents with the *Pixel Region* tool. Original image: courtesy of MathWorks.

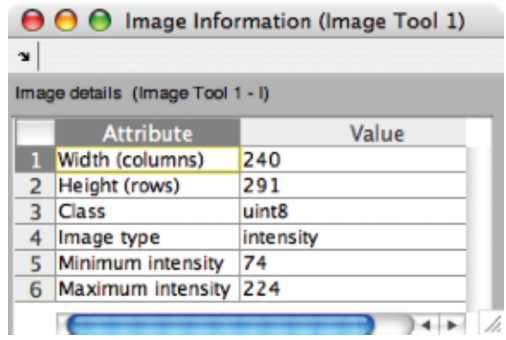

FIGURE 4.3 The *Image Information* tool.

Since `pixval` is now obsolete, the alternative way to do it is by using `imshow(I), impixelinfo`.

Two other relevant IPT functions for inspecting image contents and pixel values are

- `impixel`: returns the red, green, and blue color values of image pixels specified with the mouse.
- `imdistline`: creates a *Distance* tool—a draggable, resizable line that measures the distance between its endpoints. You can use the mouse to move and resize the line to measure the distance between any two points within an image (Figure 4.5).

4.2.6 Writing the Resulting Image onto a File

MATLAB's IPT has a built-in function, `imwrite`, to write the contents of an image in one of the most popular graphic file formats.

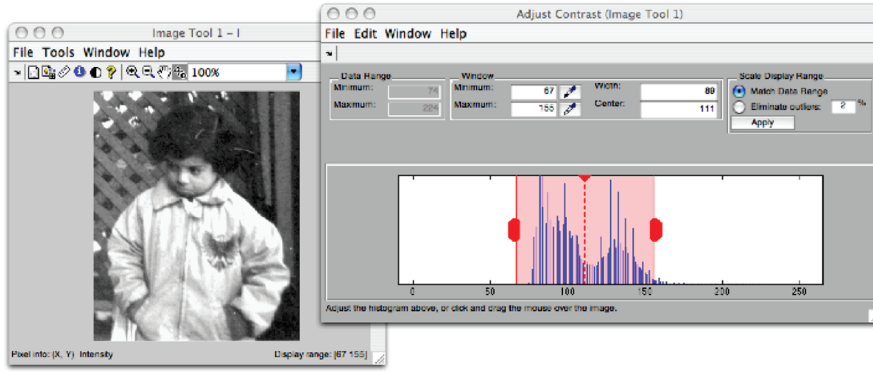

FIGURE 4.4 The *Adjust Contrast* tool. Original image: courtesy of MathWorks.

FIGURE 4.5 The *Distance* tool. Original image: courtesy of MathWorks.

If the output file format uses lossy compression[7] (e.g., JPEG), `imwrite` allows the specification of a *quality* parameter, used as a trade-off between the resulting image's subjective quality and the file size.

■ EXAMPLE 4.4

In this example, we will read an image from a PNG file and save it to a JPG file using three different quality parameters: 75 (default), 5 (poor quality, small size), and 95 (better quality, larger size).

```
I = imread('peppers.png');
imwrite(I, 'pep75.jpg');
imwrite(I, 'pep05.jpg', 'quality', 5);
imwrite(I, 'pep95.jpg', 'quality', 95);
```

The results are displayed in Figure 4.6. Figure 4.6c is clearly of lower visual quality than Figure 4.6b and d. On the other hand, the differences between Figure 4.6b and d are not too salient.

[7] You will learn about compression in Chapter 17.

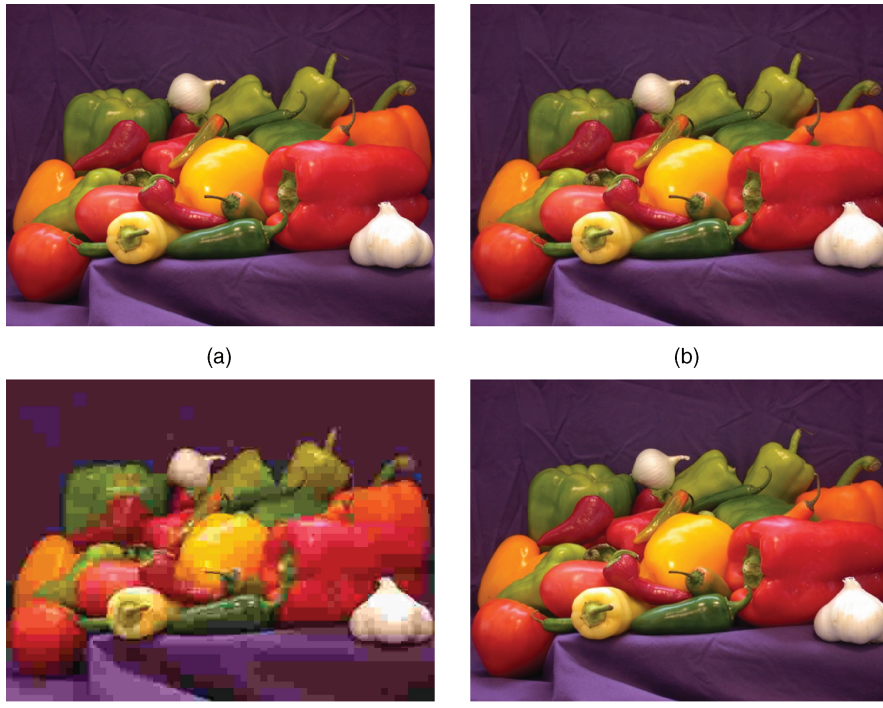

FIGURE 4.6 Reading and writing images: (a) Original image (PNG); (b) compressed image (JPG, $q = 75$, file size $= 24$ kB); (c) compressed image (JPG, $q = 5$, file size $= 8$ kB); (d) compressed image (JPG, $q = 95$, file size $= 60$ kB). Original image: courtesy of MathWorks.

4.3 TUTORIAL 4.1: MATLAB IMAGE PROCESSING TOOLBOX—A GUIDED TOUR

Goal

The goal of this tutorial is to introduce the capabilities of the Image Processing Toolbox and demonstrate how functions can be combined to solve a particular problem.

Objectives

- Learn how to use the help documentation for the IPT.
- Explore the *Identifying Round Objects Demo*.

Procedure

The MATLAB IPT offers rich functionality that may be used as building blocks for an imaging system. Let us begin by first exploring the help documentation that comes with the toolbox.

1. Open the help documentation for the IPT by navigating from the start menu located in the lower left corner of the MATLAB environment: **Start > Toolboxes > Image Processing > Help**.

The home page of the help document for the IPT provides links to several key areas of the document, such as categorical or alphabetical listings of functions, user guides, and demos. The user guides are broken up even further, including categories such as *Spatial Transformations* and *Image Segmentation*.

An easy and quick way to get an idea of what the IPT is capable of is to look at some of the demos provided with the toolbox. An example of an IPT demo is the *Identifying Round Objects Demo*, a sequence of steps used to demonstrate how IPT functions can be used and combined to develop algorithms for a particular application—in this case, identifying round objects. Most of the syntax may be unfamiliar to you at this point, but following the explanations associated with the code can give insight into its functionality. Any block of code in these demos can be evaluated by highlighting the code and selecting the **Evaluate Selection** option from the context-sensitive menu. Alternatively, you can open the `ipexroundness.m` file in the MATLAB Editor and evaluate one cell at a time. You can also evaluate each of the (six) main blocks of code by clicking on the **Run in the Command Window** hyperlink at the top right of the demo HTML page.[8]

2. Open the *Identifying Round Objects Demo* by navigating to the IPT demos: **Start > Toolboxes > Image Processing > Demos**. It will be under the **Measuring Image Features** category.
3. Run the first block of code.

As expected, the image is displayed (using the `imshow` function).

4. Execute the remaining code in the demo one block at a time. Focus on understanding *what* each step (block) does, rather than *how* it is done.

Question 1 Run (at least) two other demos from different categories within the IPT and comment on how each demo illustrates the capabilities of the IPT.

[8] You can run the entire demo by typing ipexroundness at the command prompt, but you would only see the final result and miss all the intermediate steps and results, which is the point of the demo.

4.4 TUTORIAL 4.2: BASIC IMAGE MANIPULATION

Goal

The goal of this tutorial is to explore basic image manipulation techniques using MATLAB and IPT.

Objectives

- Explore the different image types supported by MATLAB and IPT.
- Learn how to read images into MATLAB.
- Explore image conversion.
- Learn how to display images.
- Learn how to write images to disk.

Procedure

The IPT supports images of type binary, indexed, intensity, and truecolor. Before an image can be processed in MATLAB, it must first be loaded into memory. To read in an image, we use the `imread` function.

1. Load the image `coins.png` by executing the following statement:

```
I = imread('coins.png');
```

Question 1 What type of image is `coins.png`?

Question 2 Why do we use the semicolon (`;`) operator after the `imread` statement? What happens if we omit it?

Binary, intensity, and truecolor images can all be read with the `imread` function as demonstrated above. When reading in an indexed image, we must specify variables for both the image and its color map. This is illustrated in the following step:

2. Load the image `trees.tif`.

```
[X,map] = imread('trees.tif');
```

Some operations may require you to convert an image from one type to another. For example, performing image adjustments on an indexed image may not give the results you are looking to achieve because the calculations are performed on the index values and not the representative RGB values. To make this an easier task, we can convert the indexed image to an RGB image using `ind2rgb`.

3. Convert the indexed image `X` with color map `map` to an RGB image, `X_rgb`.

```
X_rgb = ind2rgb(X,map);
```

Question 3 How many dimensions does the variable `X_rgb` have and what are their sizes?

4. Convert the indexed image `X` with color map `map` to an intensity image.

```
X_gray = ind2gray(X,map);
```

Question 4 What class type is `X_gray`?

5. We can verify that the new intensity image consists of pixel values in the range [0, 255].

```
max(X_gray(:))
min(X_gray(:))
```

Question 5 Why are we required to use the colon operator (`:`) when specifying the `X_gray` variable? What happens if we omit it?

It was demonstrated in the previous step that the `X_gray` image contained values in the range [0, 255] (in this particular image, they happened to be exactly 0 and 255, which is just a coincidence). Let us see what happens when we convert the image to class `double`.

6. Convert the variable `X_gray` to class `double`.

```
X_gray_dbl = im2double(X_gray);
```

Question 6 What is the range of values for the new variable `X_gray_dbl`?

Similarly, you can convert to other class types by using `im2uint8` and `im2uint16`, for example. When converting a `uint16` image to `uint8`, you must be careful because the conversion quantizes the 65,536 possible values to 256 possible values.

MATLAB comes with built-in image displaying functions. The `image` function can be used to display image data, and the `imagesc` function will perform the same operation but in addition will scale the image data to the full range of values. The IPT provides an enhanced image displaying function that optimizes settings on the image axes to provide a better display of image data: `imshow`.

7. Use the `imshow` function (with the `impixelinfo` option) to display the `coins.png` image that is currently loaded in the variable `I`.

```
imshow(I), impixelinfo
```

Binary, intensity, and truecolor images can all be displayed as demonstrated previously. To display indexed images, we must specify the color map along with the image data.

8. Display the indexed image `trees.tif`. The image data are stored in variable `X` and the color map in `map`. Note that the `impixelinfo` option provides a clear hint that this is an indexed color image.

```
imshow(X,map), impixelinfo
```

Question 7 Consider an image where the range of possible values for each pixel is not [0, 255], but a nonstandard range such as [0, 99]. How would we display the image so that a value of 99 represents white and a value of 0 represents black?

The `impixel` function allows the inspection of the contents of selected pixels of interest within the image.

9. Use the `impixel` function to explore interactively the pixel contents of selected points in the image. Use the mouse to click on the points of interest: normal button clicks are used to select pixels, pressing **Backspace** or **Delete** removes the previously selected pixel, a double-click adds a final pixel and ends the selection, and pressing **Return** finishes the selection without adding a final pixel.

```
RGB = imread('peppers.png');
[c,r,p] = impixel(RGB);
```

Question 8 What is the meaning of the values stored in variables `r`, `c`, and `p`?

The `improfile` function can be used to compute and plot the intensity values along a line or a multiline path in an image.

10. Use the `improfile` function to explore the contents of a line in the `coins.png` image that is currently loaded in the variable `I`.

```
r1 = 17; c1 = 18; r2 = 201; c2 = 286;
imshow(I)
line([c1, c2], [r1, r2], 'Color', 'g', 'LineWidth', 2);
figure
improfile(I, [c1, c2], [r1, r2]);
ylabel('Gray level');
```

The `imtool` function is the latest and richest IPT function for displaying images. It provides all the image display capabilities of `imshow` as well as access to other

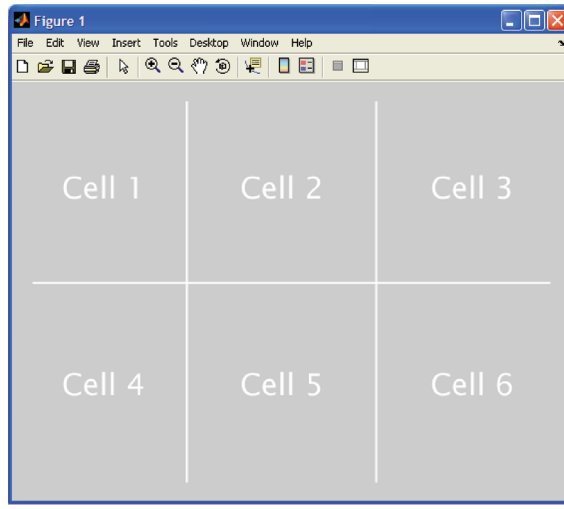

FIGURE 4.7 Division of a figure using subplot.

tools for navigating and exploring images, such as the *Pixel Region* tool, the *Image Information* tool, and the *Adjust Contrast* tool.

11. Use the `imtool` function to display the image currently loaded in the variable `X_rgb`. Note that a secondary window (Overview) will open as well. Explore the additional functionality, including the possibility of measuring distances between two points within the image.

```
imtool(X_rgb)
```

We can display multiple images within one figure using the `subplot` function. When using this function, the first two parameters specify the number of rows and columns to divide the figure. The third parameter specifies which subdivision to use. In the case of `subplot(2,3,3)`, we are telling MATLAB to divide the figure into two rows and three columns and set the third cell as active. This division is demonstrated in Figure 4.7.

12. Close any open figures (`close all`).
13. Execute the following statements to create a subplot with two images:

```
A = imread('pout.tif');
B = imread('cameraman.tif');
figure
subplot(1,2,1), imshow(A)
subplot(1,2,2), imshow(B)
```

Question 9 What is the range of values for image `A` and image `B`?

In the previous step, we displayed two images, both of which were intensity images. Even though there is no color map associated with intensity images, MATLAB uses a grayscale color map to display an intensity image (this happens in the background and is usually invisible to the user). Let us consider the case where an intensity image and an indexed image are both displayed in one figure, using the `subplot` function as before.

14. Close any open figures.
15. Display the `coins.png` (loaded in variable `I`) and the `trees.tif` (loaded in variable `X` and its color map in variable `map`) images in a subplot. Execute each statement at a time to see the effect on the images as they are displayed.

```
figure
subplot(1,2,1), imshow(I)
subplot(1,2,2), imshow(X,map)
```

Question 10 What happened to the `coins` image just after the `trees` image was displayed? Explain your answer.

To properly display images with different color maps, we must use the `subimage` function.

16. Use the `subimage` function to display multiple images with different color maps.

```
figure
subplot(1,2,1), subimage(I), axis off
subplot(1,2,2), subimage(X,map), axis off
```

The `subimage` function converts the image to an equivalent RGB image and then displays that image. We can easily do this ourselves, but there is no direct conversion from intensity to RGB, so we must first convert from intensity to indexed, and then from indexed to RGB.

17. Manually convert the intensity image `coins` (loaded in the variable `I`) to an indexed image and then to RGB. Note that the `trees` image (loaded in variable `X` with its color map in variable `map`) has already been converted to RGB in step 3 (saved in variable `X_rgb`).

```
[I_ind,I_map] = gray2ind(I,256);
I_rgb = ind2rgb(I_ind,I_map);
```

Question 11 What do the variables `I_ind` and `I_map` contain?

18. Display the truecolor images using the `imshow` function.

```
figure
subplot(1,2,1), imshow(I_rgb)
subplot(1,2,2), imshow(X_rgb)
```

19. Use `imwrite` to save two of the modified images in this tutorial to files for further use. Use the JPEG format for one of them and the PNG extension for the other. For example,

```
imwrite(X_rgb, 'rgb_trees.jpg');
imwrite(X_gray, 'gray_trees.png');
```

WHAT HAVE WE LEARNED?

- MATLAB's IPT has a built-in function to open and read the contents of image files in most popular formats, `imread`.
- The most common data classes for images are `uint8` (1 byte per pixel, [0, 255]), `double` (8 bytes per pixel, [0.0, 1.0]), and `logical` (1 byte per pixel, *true*—white or *false*—black).
- Data class compatibility is a critical prerequisite for image processing algorithms to work properly, which occasionally requires data class conversions. In addition to standard data class conversion (*typecasting*), MATLAB has numerous specialized functions for image class conversions.
- A good understanding of the different data classes (and corresponding ranges of pixel values) of images stored in memory is essential to the success of image processing algorithms. Ignoring or overlooking these aspects may lead to unwillingly truncating of intermediate calculation results, setting of wrong thresholds for pixel values, and many other potential problems.
- MATLAB has several functions for displaying images, such as `image`, `imagesc`, and `imshow`.
- To inspect the pixel contents of an image more closely, MATLAB includes the `imtool` function that provides all the image display capabilities of `imshow` as well as access to other tools for navigating and exploring images.
- To save the results of your image processing to a file, use MATLAB's built-in function `imwrite`.

LEARN MORE ABOUT IT

The best ways to learn more about the IPT are

- IPT demos: the MATLAB package includes many demos, organized in categories, that provide an excellent opportunity to learn most of the IPT's capabilities.

- MATLAB webinars: short (1 h or less) technical presentations by MathWorks developers and engineers.

ON THE WEB

- MATLAB Image Processing Toolbox home page.
  http://www.mathworks.com/products/image/
- Steve Eddins's blog: Many useful hints about image processing concepts, algorithm implementations, and MATLAB.
  http://blogs.mathworks.com/steve/

4.5 PROBLEMS

4.1 Create a 2 × 2 array of type `double`

```
A = [1.5 -2 ; 0.5 0]
```

and use it to perform the following operations:

(a) Convert to `uint8` using typecasting and interpret the results.

(b) Convert to a normalized ([0, 255]) grayscale image of data class `uint8` using `im2uint8` and interpret the results.

(c) Convert to a normalized ([0.0, 1.0]) grayscale image of data class `double` using `mat2gray` and interpret the results.

(d) Convert the result from the previous step to a binary image using `im2bw` (with a threshold level of 0.5) and interpret the results.

4.2 Create a 2 × 2 array of type `double`

```
A = [1 4 ; 5 3]
```

and write MATLAB statements to perform the following operations:

(a) Convert to a normalized ([0.0, 1.0]) grayscale image of data class `double`.

(b) Convert to a binary image of data class `logical`, such that any values greater than 2 (in the original image) will be interpreted as 1 (true).

(c) Repeat the previous step, this time producing a result of data class `double`.

4.3 The IPT function `gray2ind` allows conversion from a grayscale image to its indexed equivalent. What interesting property will the resulting color map (palette) display?

4.4 How does the IPT function `rgb2ind` handle the possibility that the original RGB image contains many more colors than the maximum palette (color map) size (65,536 colors)?

4.5 If you type `help imdemos` in MATLAB, you will see (among other things) a list of all the sample images that come with its IPT. Select five of those images and collect the following information about each of them:

- File name
- File format (extension)
- Type (binary, grayscale, truecolor, or indexed color)
- Size (bytes)
- Width (pixels)
- Height (pixels)

4.6 Select five images available in MATLAB (they may be the same as from the previous problem, or not, you choose). Open each of them using `imread`, save it (using `imwrite`) to (at least three) different file formats, and compare the resulting file size (in bytes) for each output format.