Chapter 1. Basic Image Handling and Processing

This chapter is an introduction to handling and processing images. With extensive examples, it explains the central Python packages you will need for working with images. This chapter introduces the basic tools for reading images, converting and scaling images, computing derivatives, plotting or saving results, and so on. We will use these throughout the remainder of the book.

1.1 PIL—The Python Imaging Library

The *Python Imaging Library* (*PIL*) provides general image handling and lots of useful basic image operations like resizing, cropping, rotating, color conversion and much more. PIL i f
http://www.pythonware.com/products/pil/

Hey there 👋 Want to learn more about the O'Reilly learning platform for your team?

1

With PIL, you can read images from most formats and write to the most common ones. The most important module is the `Image` module. To read an image, use:

```
from PIL import Image

pil_im = Image.open('empire.jpg')
```

The return value, *pil_im*, is a PIL image object.

Color conversions are done using the `convert()` method. To read an image and convert it to grayscale, just add `convert('L')` like this:

```
pil_im = Image.open('empire.jpg').convert('L')
```

Here are some examples taken from the PIL documentation, available at *http://www.pythonware.com/library/pil/handbook/index.htm*. Output from the examples is shown in Figure 1-1.

Convert Images to Another Format

Using the `save()` method, PIL can save images in most image file formats. Here's an example that takes all image files in a list of filenames (*filelist*) and converts the images to JPEG files:

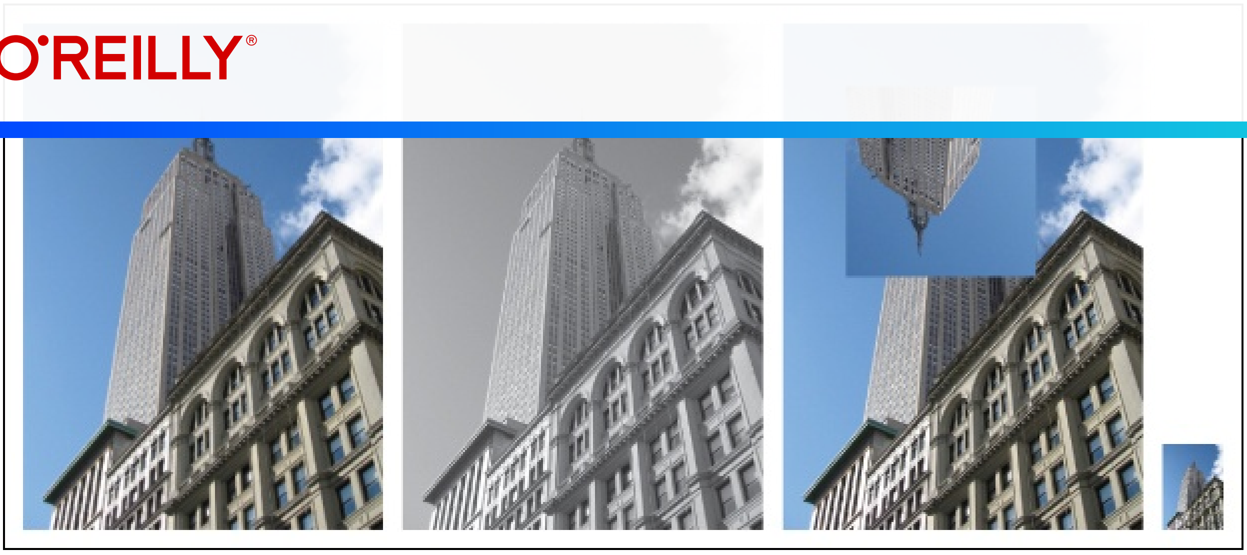

Figure 1-1. Examples of processing images with PIL.

```
from PIL import Image
import os

for infile in filelist:
  outfile = os.path.splitext(infile)[0] + ".jpg"
  if infile != outfile:
    try:
      Image.open(infile).save(outfile)
    except IOError:
      print "cannot convert", infile
```

The PIL function `open()` creates a PIL image object and the `save()` method saves the image to a file with the given filename. The new filename will be the same as the original with the file ending ".jpg" instead. PIL is smart enough to determine the image format from the file extension. There is a simple check that the file is not already a JPEG file and a message is printed to the console if the conversion fails.

Throughout this book we are going to need lists of images to process. Here's how you could create a list of filenames of all images in a folder.

Create a file called *imtools.py* to store some of these generally useful routines and add the following function:

```
import os

def get_imlist(path):
  """  Returns a list of filenames for
    all jpg images in a directory. """

  return [os.path.join(path,f) for f in os.listdir(path) if
```

Now, back to PIL.

Create Thumbnails

Using PIL to create thumbnails is very simple. The `thumbnail()` method takes a tuple specifying the new size and converts the image to a thumbnail image with size that fits within the tuple. To create a thumbnail with longest side 128 pixels, use the method like this:

```
pil_im.thumbnail((128,128))
```

Copy and Paste Regions

Cropping a region from an image is done using the `crop()` method:

```
box = (100,100,400,400)
region = pil_im.crop(box)
```

The region is defined by a 4-tuple, where coordinates are (left, upper, right, lower). PIL uses a coordinate system with (0, 0) in the upper left corner. The extracted region can, for example, be rotated and then put back using the `paste()` method like this:

```
region = region.transpose(Image.ROTATE_180)
pil_im.paste(region,box)
```

Resize and Rotate

To resize an image, call `resize()` with a tuple giving the new size:

```
out = pil_im.resize((128,128))
```

To rotate an image, use counterclockwise angles and `rotate()` like this:

```
out = pil_im.rotate(45)
```

Some examples are shown in Figure 1-1. The leftmost image is the original, followed by a grayscale version, a rotated crop pasted in, and a thumbnail image.

1.2 Matplotlib

When working with mathematics and plotting graphs or drawing points, lines, and curves on images, `Matplotlib` is a good graphics library with much more powerful features than the plotting available in PIL.

that allows the user to create plots. `Matplotlib` is open source and available freely from *http://matplotlib.sourceforge.net/*, where detailed documentation and tutorials are available. Here are some examples showing most of the functions we will need in this book.

Plotting Images, Points, and Lines

Although it is possible to create nice bar plots, pie charts, scatter plots, etc., only a few commands are needed for most computer vision purposes. Most importantly, we want to be able to show things like interest points, correspondences, and detected objects using points and lines. Here is an example of plotting an image with a few points and a line:

```
from PIL import Image
from pylab import *

# read image to array
im = array(Image.open('empire.jpg'))

# plot the image
imshow(im)

# some points
x = [100,100,400,400]
y = [200,500,200,500]

# plot the points with red star-markers
plot(x,y,'r*')

# line plot connecting the first two points
```

```
plot(x[:2],y[:2])
```

```
# add title and show the plot
title('Plotting: "empire.jpg"')
show()
```

This plots the image, then four points with red star markers at the x and y coordinates given by the *x* and *y* lists, and finally draws a line (blue by default) between the two first points in these lists. Figure 1-2 shows the result. The `show()` command starts the figure GUI and raises the figure windows. This GUI loop blocks your scripts and they are paused until the last figure window is closed. You should call `show()` only once per script, usually at the end. Note that `PyLab` uses a coordinate origin at the top left corner as is common for images. The axes are useful for debugging, but if you want a prettier plot, add:

```
axis('off')
```

This will give a plot like the one on the right in Figure 1-2 instead.

There are many options for formatting color and styles when plotting. The most useful are the short commands shown in Table 1-1, Table 1-2 and Table 1-3. Use them like this:

```
plot(x,y) # default blue solid line

plot(x,y,'r*') # red star-markers

plot(x,y,'go-') # green line with circle-markers
plot(x,y,'ks:') # black dotted line with square-markers
```

Image Contours and Histograms

Let’s look at two examples of special plots: image contours and image histograms. Visualizing image iso-contours (or iso-contours of other 2D functions) can be very useful. This needs grayscale images, because the contours need to be taken on a single value for every coordinate [*x*, *y*]. Here’s how to do it:

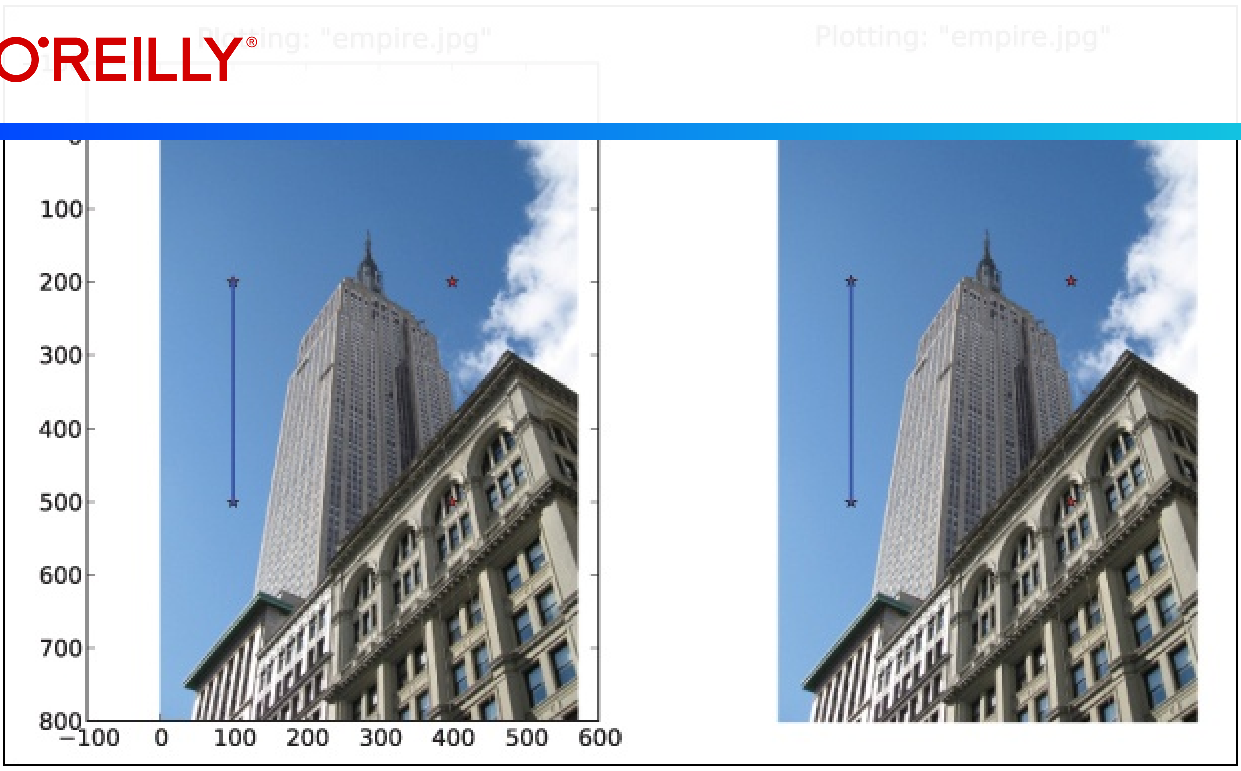

Figure 1-2. Examples of plotting with `Matplotlib`. An image with points and a line with and without showing the axes.

Table 1-1. Basic color formatting commands for plotting with `PyLab`.

Color	
`'b'`	blue
`'g'`	green
`'r'`	red
`'c'`	cyan
`'m'`	magenta

`'y'`	yellow
`'k'`	black
`'w'`	white

Table 1-2. Basic line style formatting commands for plotting with `PyLab`.

Line style	
`'-'`	solid
`'- -'`	dashed
`':'`	dotted

Table 1-2. Basic plot marker formatting commands for plotting with PyLab

Marker	
`'.'`	point
`'o'`	circle
`'s'`	square
`'*'`	star
`'+'`	plus
`'x'`	x

```
from PIL import Image
from pylab import *

# read image to array
im = array(Image.open('empire.jpg').convert('L'))

# create a new figure
figure()
# don't use colors
gray()
# show contours with origin upper left corner
```

```
contour(im, origin='image')
axis('equal')
axis('off')
```

As before, the PIL method `convert()` does conversion to grayscale.

An image histogram is a plot showing the distribution of pixel values. A number of bins is specified for the span of values and each bin gets a count of how many pixels have values in the bin's range. The visualization of the (graylevel) image histogram is done using the `hist()` function:

```
figure()
hist(im.flatten(),128)
show()
```

The second argument specifies the number of bins to use. Note that the image needs to be flattened first, because `hist()` takes a one-dimensional array as input. The method `flatten()` converts any array to a one-dimensional array with values taken row-wise. Figure 1-3 shows the contour and histogram plot.

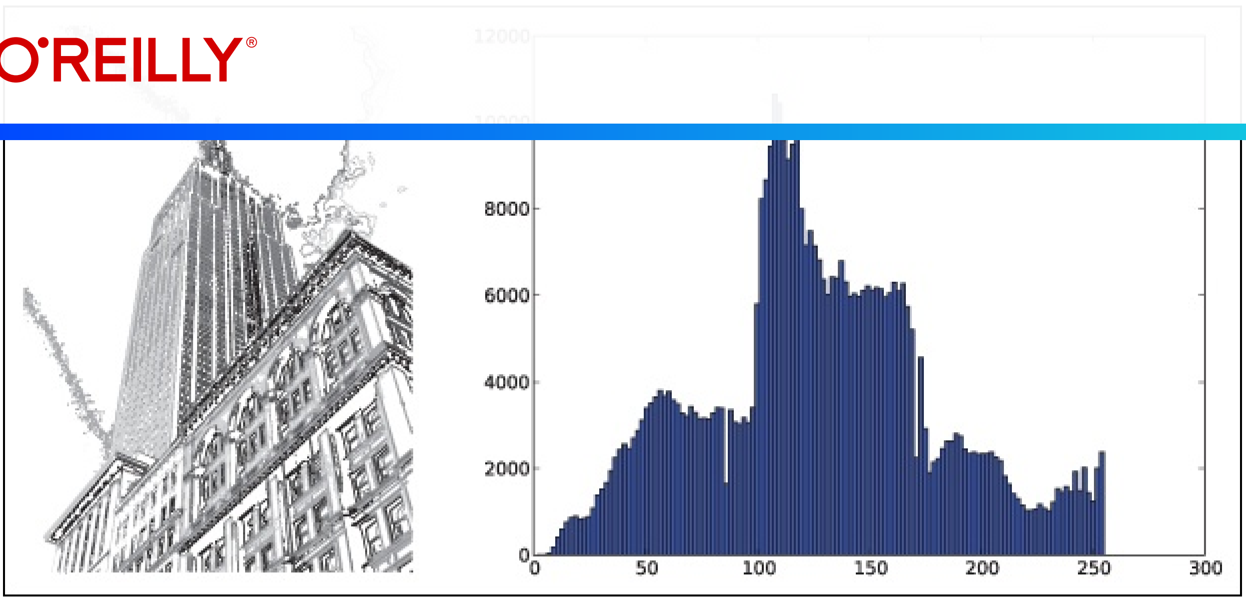

Figure 1-3. Examples of visualizing image contours and plotting image histograms with `Matplotlib`*.*

Interactive Annotation

Sometimes users need to interact with an application, for example by marking points in an image, or you need to annotate some training data. `PyLab` comes with a simple function, `ginput()`, that lets you do just that. Here's a short example:

```
from PIL import Image
from pylab import *

im = array(Image.open('empire.jpg'))
imshow(im)
print 'Please click 3 points'
x = ginput(3)
print 'you clicked:',x
show()
```

This plots an image and waits for the user to click three times in the image region of the figure window. The coordinates [x, y] of the clicks are saved in a list x.

1.3 NumPy

`NumPy` (*http://www.scipy.org/NumPy/*) is a package popularly used for scientific computing with Python. `NumPy` contains a number of useful concepts such as array objects (for representing vectors, matrices, images and much more) and linear algebra functions. The `NumPy` array object will be used in almost all examples throughout this book.[2] The array object lets you do important operations such as matrix multiplication, transposition, solving equation systems, vector multiplication, and normalization, which are needed to do things like aligning images, warping images, modeling variations, classifying images, grouping images, and so on.

`NumPy` is freely available from *http://www.scipy.org/Download* and the online documentation (*http://docs.scipy.org/doc/numpy/*) contains answers to most questions. For more details on `NumPy`, the freely available book [24] is a good reference.

Array Image Representation

When we loaded images in the previous examples, we converted them to `NumPy` array objects with the `array()` call but didn't mention what that means. Arrays in `NumPy` are multi-dimensional and can represent vectors, matrices, and images. An array is much like a list (or list of lists) but is restricted to having all elements of the same type. Unless specified on creation, the type will automatically be set depending on the data.

The following example illustrates this for images:

```
im = array(Image.open('empire.jpg'))
print im.shape, im.dtype

im = array(Image.open('empire.jpg').convert('L'),'f')
print im.shape, im.dtype
```

The printout in your console will look like this:

```
(800, 569, 3) uint8
(800, 569) float32
```

The first tuple on each line is the shape of the image array (rows, columns, color channels), and the following string is the data type of the array elements. Images are usually encoded with unsigned 8-bit integers (uint8), so loading this image and converting to an array gives the type "uint8" in the first case. The second case does grayscale conversion and creates the array with the extra argument "f". This is a short command for setting the type to floating point. For more data type options, see [24]. Note that the grayscale image has only two values in the shape tuple; obviously it has no color information.

Elements in the array are accessed with indexes. The value at coordinates *i*, *j* and color channel *k* are accessed like this:

```
value = im[i,j,k]
```

Multiple elements can be accessed using array slicing. *Slicing* returns a view into the array specified by intervals. Here are some examples for a grayscale image:

```
im[i,:] = im[j,:]       # set the values of row i with values fr
im[:,i] = 100           # set all values in column i to 100
im[:100,:50].sum()      # the sum of the values of the first 100
im[50:100,50:100]       # rows 50-100, columns 50-100 (100th not
im[i].mean()            # average of row i
im[:,-1]                # last column
im[-2,:] (or im[-2])    # second to last row
```

Note the example with only one index. If you only use one index, it is interpreted as the row index. Note also the last examples. Negative indices count from the last element backward. We will frequently use slicing to access pixel values, and it is an important concept to understand.

There are many operations and ways to use arrays. We will introduce them as they are needed throughout this book. See the online documentation or the book [24] for more explanations.

Graylevel Transforms

After reading images to `NumPy` arrays, we can perform any mathematical operation we like on them. A simple example of this is to transform the graylevels of an image. Take any function *f* that maps the interval 0 . . . 255 (or, if you like, 0 . . . 1) to itself (meaning that the output has the same range as the input). Here are some examples:

```
im = array(Image.open('empire.jpg').convert('L'))

im2 = 255 - im # invert image

im3 = (100.0/255) * im + 100 # clamp to interval 100...200

im4 = 255.0 * (im/255.0)**2 # squared
```

The first example inverts the graylevels of the image, the second one clamps the intensities to the interval 100 . . . 200, and the third applies a quadratic function, which lowers the values of the darker pixels. Figure 1-4 shows the functions and Figure 1-5 the resulting images. You can check the minimum and maximum values of each image using:

```
print int(im.min()), int(im.max())
```

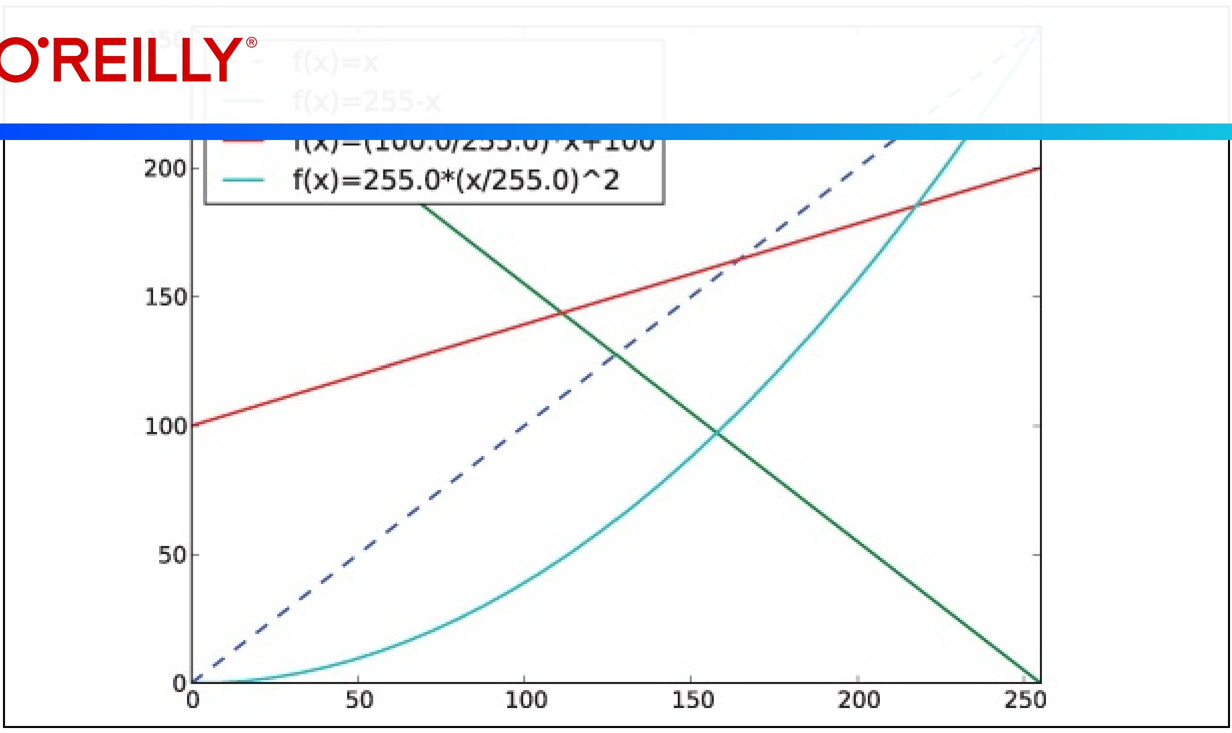

Figure 1-4. Example of graylevel transforms. Three example functions together with the identity transform showed as a dashed line.

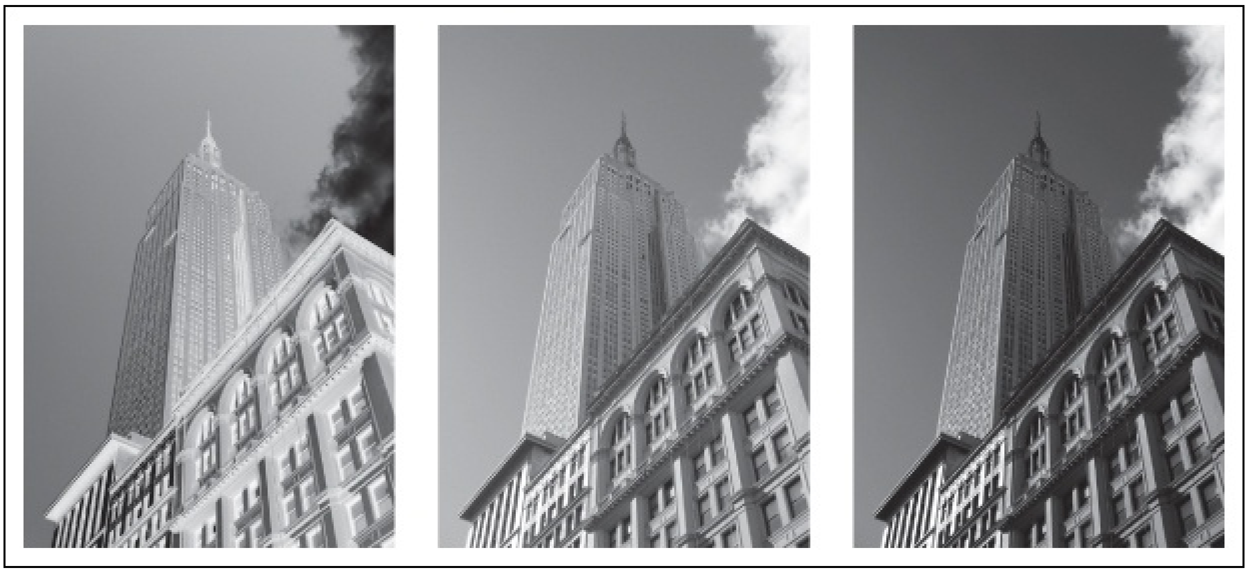

Figure 1-5. Graylevel transforms. Applying the functions in Figure 1-4: Inverting the image with f(x) = 255 – x (left), clamping the image with f(x) = (100/255)x + 100 (middle), quadratic transformation with f(x) = 255(x/255)^2 (right).

If you try that for each of the examples above, you should get the following output:

```
100 200
0 255
```

The reverse of the `array()` transformation can be done using the PIL function `fromarray()` as:

```
pil_im = Image.fromarray(im)
```

If you did some operation to change the type from "uint8" to another data type, such as *im3* or *im4* in the example above, you need to convert back before creating the PIL image:

```
pil_im = Image.fromarray(uint8(im))
```

If you are not absolutely sure of the type of the input, you should do this as it is the safe choice. Note that `NumPy` will always change the array type to the "lowest" type that can represent the data. Multiplication or division with floating point numbers will change an integer type array to float.

Image Resizing

`NumPy` arrays will be our main tool for working with images and data. There is no simple way to resize arrays, which you will want to do for images. We can use the PIL image object conversion shown earlier to make a simple image resizing function. Add the following to *imtools.py*:

```
    pil_im = Image.fromarray(uint8(im))

    return array(pil_im.resize(sz))
```

This function will come in handy later.

Histogram Equalization

A very useful example of a graylevel transform is *histogram equalization*. This transform flattens the graylevel histogram of an image so that all intensities are as equally common as possible. This is often a good way to normalize image intensity before further processing and also a way to increase image contrast.

The transform function is, in this case, a *cumulative distribution function* (cdf) of the pixel values in the image (normalized to map the range of pixel values to the desired range).

Here’s how to do it. Add this function to the file *imtools.py*:

```
def histeq(im,nbr_bins=256):
  """  Histogram equalization of a grayscale image. """

  # get image histogram
  imhist,bins = histogram(im.flatten(),nbr_bins,normed=True
  cdf = imhist.cumsum() # cumulative distribution function
  cdf = 255 * cdf / cdf[-1] # normalize

  # use linear interpolation of cdf to find new pixel values
```

```
    im2 = interp(im.flatten(),bins[:-1],cdf)

    return im2.reshape(im.shape), cdf
```

The function takes a grayscale image and the number of bins to use in the histogram as input, and returns an image with equalized histogram together with the cumulative distribution function used to do the mapping of pixel values. Note the use of the last element (index -1) of the cdf to normalize it between 0 ... 1. Try this on an image like this:

```
from PIL import Image
from numpy import *

im = array(Image.open('AquaTermi_lowcontrast.jpg').convert(
im2,cdf = imtools.histeq(im)
```

Figure 1-6 and Figure 1-7 show examples of histogram equalization. The top row shows the graylevel histogram before and after equalization together with the cdf mapping. As you can see, the contrast increases and the details of the dark regions now appear clearly.

Averaging Images

Averaging images is a simple way of reducing image noise and is also often used for artistic effects. Computing an average image from a list of images is not difficult. Assuming the images all have the same size, we can compute the average of all those images by simply summing them up and dividing with the number of images. Add the following function to *imtools.py*:

```
pute_average(imlist):
    """  Compute the average of a list of images.  """

    # open first image and make into array of type float
    averageim = array(Image.open(imlist[0]), 'f')

    for imname in imlist[1:]:
      try:
        averageim += array(Image.open(imname))
      except:
        print imname + '...skipped'
    averageim /= len(imlist)

    # return average as uint8
    return array(averageim, 'uint8')
```

This includes some basic exception handling to skip images that can’t be opened. There is another way to compute average images using the `mean()` function. This requires all images to be stacked into an array and will use lots of memory if there are many images. We will use this function in the next section.

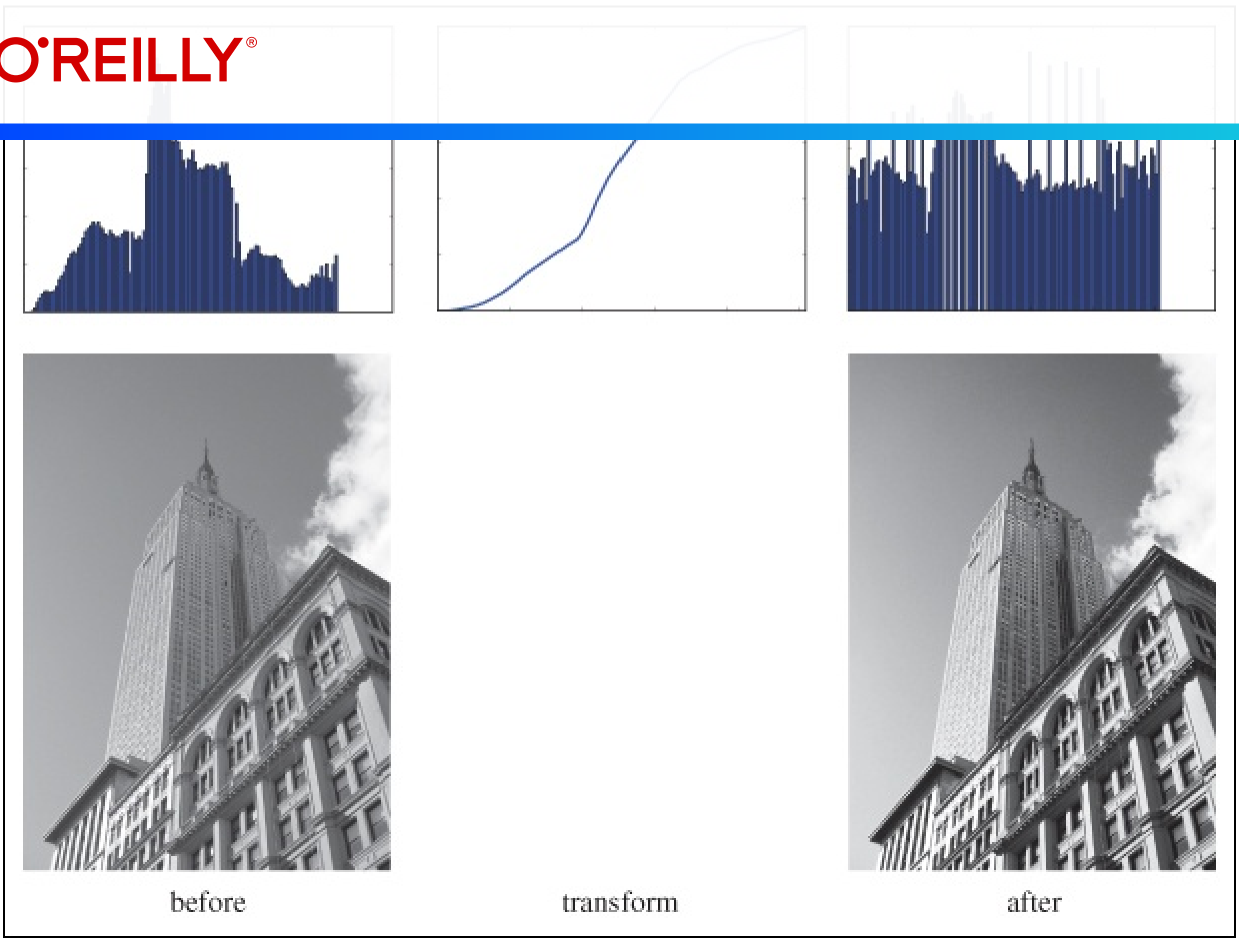

Figure 1-6. Example of histogram equalization. On the left is the original image and histogram. The middle plot is the graylevel transform function. On the right is the image and histogram after histogram equalization.

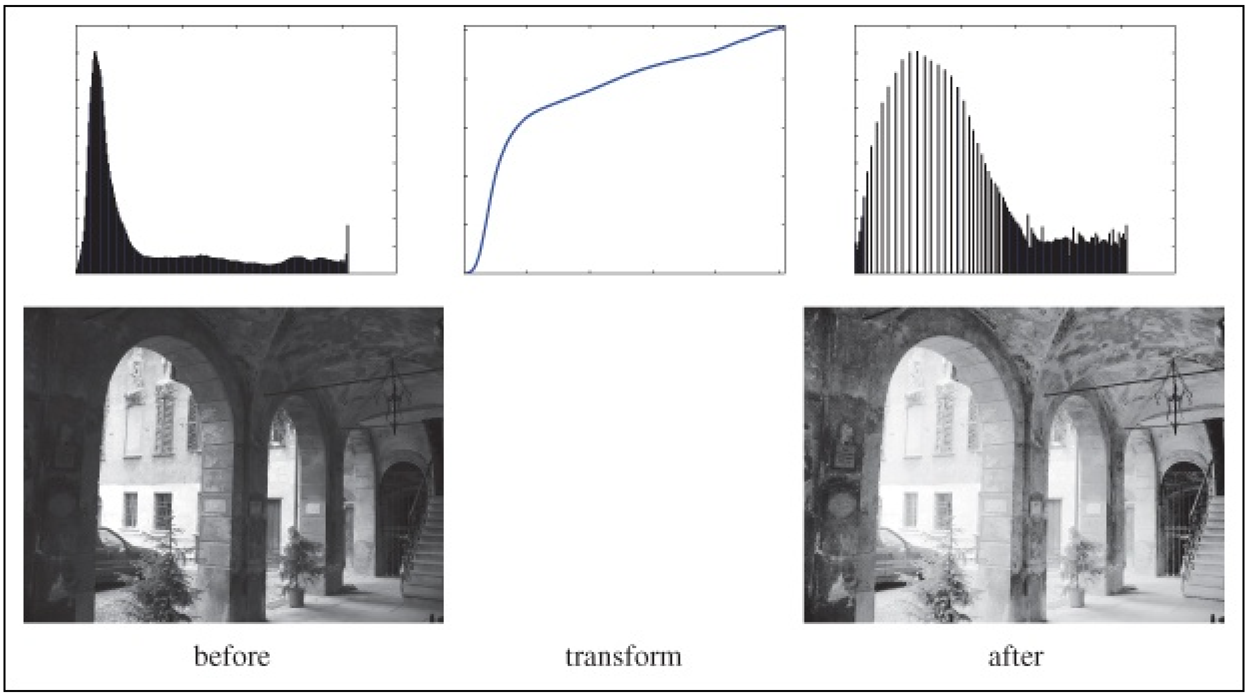

Figure 1-7. Example of histogram equalization. On the left is the original image and histogram. The middle plot is the graylevel

transform function. On the right is the image and histogram after histogram equalization.

PCA of Images

Principal Component Analysis (*PCA*) is a useful technique for dimensionality reduction and is optimal in the sense that it represents the variability of the training data with as few dimensions as possible. Even a tiny 100 × 100 pixel grayscale image has 10,000 dimensions, and can be considered a point in a 10,000-dimensional space. A megapixel image has dimensions in the millions. With such high dimensionality, it is no surprise that dimensionality reduction comes in handy in many computer vision applications. The projection matrix resulting from PCA can be seen as a change of coordinates to a coordinate system where the coordinates are in descending order of importance.

To apply PCA on image data, the images need to be converted to a one-dimensional vector representation using, for example, `NumPy`'s `flatten()` method.

The flattened images are collected in a single matrix by stacking them, one row for each image. The rows are then centered relative to the mean image before the computation of the dominant directions. To find the principal components, singular value decomposition (SVD) is usually used, but if the dimensionality is high, there is a useful trick that can be used instead since the SVD computation will be very slow in that case. Here is what it looks like in code:

```
def pca(X):
  """  Principal Component Analysis
    input: X, matrix with training data stored as flattened arra
    return: projection matrix (with important dimensions first),
    and mean."""

  # get dimensions
  num_data,dim = X.shape

  # center data
  mean_X = X.mean(axis=0)
  X = X - mean_X

  if dim>num_data:
    # PCA - compact trick used
    M = dot(X,X.T) # covariance matrix
    e,EV = linalg.eigh(M) # eigenvalues and eigenvectors
    tmp = dot(X.T,EV).T # this is the compact trick
    V = tmp[::-1] # reverse since last eigenvectors are the one
    S = sqrt(e)[::-1] # reverse since eigenvalues are in incre
    for i in range(V.shape[1]):
      V[:,i] /= S
  else:
    # PCA - SVD used
    U,S,V = linalg.svd(X)
    V = V[:num_data] # only makes sense to return the first nu

  # return the projection matrix, the variance and the mean
  return V,S,mean_X
```

This function first centers the data by subtracting the mean in each dimension. Then the eigenvectors corresponding to the largest eigenvalues of the covariance matrix are computed, either using a compact trick or using SVD. Here we used the function `range()`, which takes an integer *n* and returns a list of integers 0 . . . (*n* – 1). Feel free to use the alternative `arange()`, which gives an array, or `xrange()`, which gives a generator (and might give speed improvements). We will stick with `range()` throughout the book.

We switch from SVD to use a trick with computing eigenvectors of the (smaller) covariance matrix XX^T if the number of data points is less than the dimension of the vectors. There are also ways of only computing the eigenvectors corresponding to the *k* largest eigenvalues (*k* being the number of desired dimensions), making it even faster. We leave this to the interested reader to explore, since it is really outside the scope of this book. The rows of the matrix *V* are orthogonal and contain the coordinate directions in order of descending variance of the training data.

Let's try this on an example of font images. The file *fontimages.zip* contains small thumbnail images of the character "a" printed in different fonts and then scanned. The 2,359 fonts are from a collection of freely available fonts.[3] Assuming that the filenames of these images are stored in a list, *imlist*, along with the previous code, in a file *pca.py*, the principal components can be computed and shown like this:

```
from PIL import Image
from numpy import *
from pylab import *
import pca
```

```
im = array(Image.open(imlist[0])) # open one image to get siz
m,n = im.shape[0:2] # get the size of the images
imnbr = len(imlist) # get the number of images

# create matrix to store all flattened images
immatrix = array([array(Image.open(im)).flatten()
              for im in imlist],'f')

# perform PCA
V,S,immean = pca.pca(immatrix)

# show some images (mean and 7 first modes)
figure()
gray()
subplot(2,4,1)
imshow(immean.reshape(m,n))
for i in range(7):
  subplot(2,4,i+2)
  imshow(V[i].reshape(m,n))

show()
```

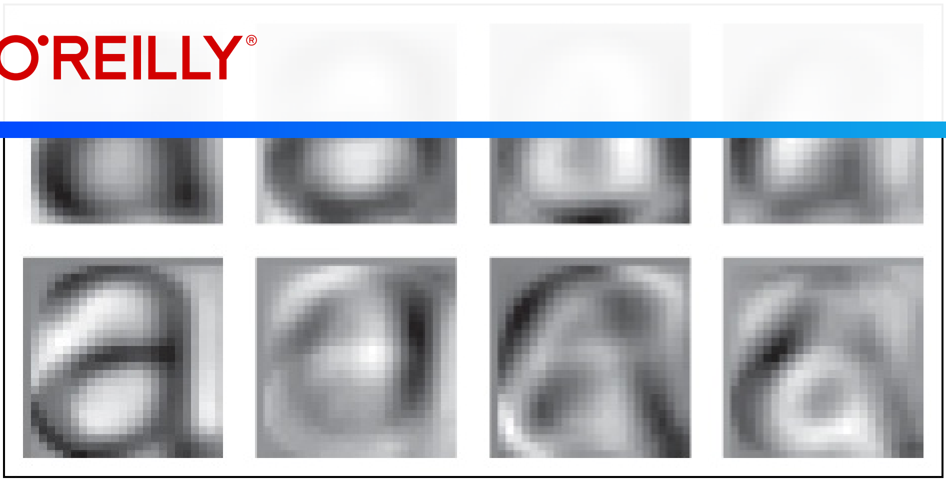

Figure 1-8. The mean image (top left) and the first seven modes; that is, the directions with most variation.

Note that the images need to be converted back from the one-dimensional representation using `reshape()`. Running the example should give eight images in one figure window like the ones in Figure 1-8. Here we used the `PyLab` function `subplot()` to place multiple plots in one window.

Using the Pickle Module

If you want to save some results or data for later use, the `pickle` module, which comes with Python, is very useful. Pickle can take almost any Python object and convert it to a string representation. This process is called *pickling*. Reconstructing the object from the string representation is conversely called *unpickling*. This string representation can then be easily stored or transmitted.

Let's illustrate this with an example. Suppose we want to save the image mean and principal components of the font images in the previous section. This is done like this:

```
    mean and principal components
    pickle.dump(immean,f)
    pickle.dump(V,f)
    f.close()
```

As you can see, several objects can be pickled to the same file. There are several different protocols available for the *.pkl* files, and if unsure, it is best to read and write binary files. To load the data in some other Python session, just use the `load()` method like this:

```
# load mean and principal components
f = open('font_pca_modes.pkl', 'rb')
immean = pickle.load(f)
V = pickle.load(f)
f.close()
```

Note that the order of the objects should be the same! There is also an optimized version written in C called `cpickle` that is fully compatible with the standard pickle module. More details can be found on the pickle module documentation page *http://docs.python.org/library/pickle.html*.

For the remainder of this book, we will use the `with` statement to handle file reading and writing. This is a construct that was introduced in Python 2.5 that automatically handles opening and closing of files (even if errors occur while the files are open). Here is what the saving and loading above looks like using `with()`:

```
# open file and save
with open('font_pca_modes.pkl', 'wb') as f:
```

```
pickle.dump(immean,f)
pickle.dump(V,f)
```

and:

```
# open file and load
with open('font_pca_modes.pkl', 'rb') as f:
  immean = pickle.load(f)
  V = pickle.load(f)
```

This might look strange the first time you see it, but it is a very useful construct. If you don't like it, just use the `open` and `close` functions as above.

As an alternative to using pickle, `NumPy` also has simple functions for reading and writing text files that can be useful if your data does not contain complicated structures, for example a list of points clicked in an image. To save an array *x* to file, use:

```
savetxt('test.txt',x,'%i')
```

The last parameter indicates that integer format should be used. Similarly, reading is done like this:

```
x = loadtxt('test.txt')
```

You can find out more from the online documentation *http://docs.scipy.org/doc/numpy/reference/generated/numpy.loadtxt.html*.

Finally, NumPy has dedicated functions for saving and loading arrays. Look for `save()` and `load()` in the online documentation for the details.

1.4 SciPy

`SciPy` (*http://scipy.org/*) is an open-source package for mathematics that builds on `NumPy` and provides efficient routines for a number of operations, including numerical integration, optimization, statistics, signal processing, and most importantly for us, image processing. As the following will show, there are many useful modules in `SciPy`. `SciPy` is free and available at *http://scipy.org/Download*.

Blurring Images

A classic and very useful example of image convolution is *Gaussian blurring* of images. In essence, the (grayscale) image I is convolved with a Gaussian kernel to create a blurred version

$$I_\sigma = I * G_\sigma,$$

where * indicates convolution and G_σ is a Gaussian 2D-kernel with standard deviation σ defined as

$$G_\sigma = \frac{1}{2\pi\sigma} e^{-(x^2+y^2)/2\sigma^2}.$$

Gaussian blurring is used to define an image scale to work in, for interpolation, for computing interest points, and in many more applications.

SciPy comes with a module for filtering called scipy.ndimage.filters that can be used to compute these convolutions using a fast 1D separation. All you need to do is this:

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters

im = array(Image.open('empire.jpg').convert('L'))
im2 = filters.gaussian_filter(im,5)
```

Here the last parameter of `gaussian_filter()` is the standard deviation.

Figure 1-9 shows examples of an image blurred with increasing σ. Larger values give less detail. To blur color images, simply apply Gaussian blurring to each color channel:

```
im = array(Image.open('empire.jpg'))
im2 = zeros(im.shape)
for i in range(3):
  im2[:,:,i] = filters.gaussian_filter(im[:,:,i],5)
im2 = uint8(im2)
```

Here the last conversion to “uint8” is not always needed but forces the pixel values to be in 8-bit representation. We could also have used

```
im2 = array(im2,'uint8')
```

for the conversion.

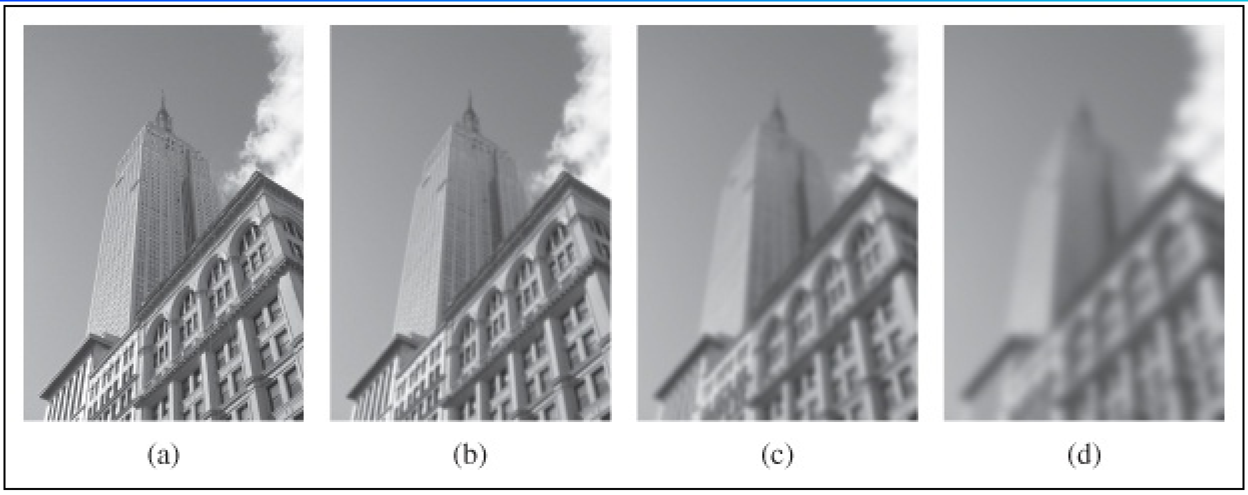

Figure 1-9. An example of Gaussian blurring using the `scipy.ndimage.filters` *module: (a) original image in grayscale; (b) Gaussian filter with $\sigma = 2$; (c) with $\sigma = 5$; (d) with $\sigma = 10$.*

For more information on using this module and the different parameter choices, check out the `SciPy` documentation of `scipy.ndimage` at *http://docs.scipy.org/doc/scipy/reference/ndimage.html*.

Image Derivatives

How the image intensity changes over the image is important information and is used for many applications, as we will see throughout this book. The intensity change is described with the x and y derivatives I_x and I_y of the graylevel image I (for color images, derivatives are usually taken for each color channel).

The *image gradient* is the vector $\nabla I = [I_x, I_y]^T$. The gradient has two important properties, the *gradient magnitude* which describes how strong the image intensity change is, and the *gradient angle*

$\alpha = \arctan2(I_y, I_x)$,

which indicates the direction of largest intensity change at each point (pixel) in the image. The `NumPy` function `arctan2()` returns the signed angle in radians, in the interval $-\pi \ldots \pi$.

Computing the image derivatives can be done using discrete approximations. These are most easily implemented as convolutions

$I_x = I * D_x$ and $I_y = I * D_y$.

Two common choices for D_x and D_y are the *Prewitt filters*

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } D_y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix},$$

and *Sobel filters*

$$D_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ and } D_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}.$$

These derivative filters are easy to implement using the standard convolution available in the `scipy.ndimage.filters` module. For example:

```
from PIL import Image
from numpy import *
from scipy.ndimage import filters

im = array(Image.open('empire.jpg').convert('L'))

# Sobel derivative filters
imx = zeros(im.shape)
filters.sobel(im,1,imx)

imy = zeros(im.shape)
filters.sobel(im,0,imy)

magnitude = sqrt(imx**2+imy**2)
```

This computes x and y derivatives and gradient magnitude using the *Sobel filter*. The second argument selects the x or y derivative, and the third stores the output. Figure 1-10 shows an image with derivatives computed using the Sobel filter. In the two derivative images, positive derivatives are shown with bright pixels and negative derivatives are dark. Gray areas have values close to zero.

Using this approach has the drawback that derivatives are taken on the scale determined by the image resolution. To be more robust to image noise and to compute derivatives at any scale, *Gaussian derivative filters* can be used:

$I_x = I * G_{\sigma x}$ and $I_y = I * G_{\sigma y}$,

where $G_{\sigma x}$ and $G_{\sigma y}$ are the x and y derivatives of G_σ, a Gaussian function with standard deviation σ.

The `filters.gaussian_filter()` function we used for blurring earlier can also take extra arguments to compute Gaussian derivatives instead. To try this on an image, simply do:

```
sigma = 5 # standard deviation

imx = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (0,1), imx)

imy = zeros(im.shape)
filters.gaussian_filter(im, (sigma,sigma), (1,0), imy)
```

The third argument specifies which order of derivatives to use in each direction using the standard deviation determined by the second argument. See the documentation for the details. Figure 1-11 shows the derivatives and gradient magnitude for different scales. Compare this to the blurring at the same scales in Figure 1-9.

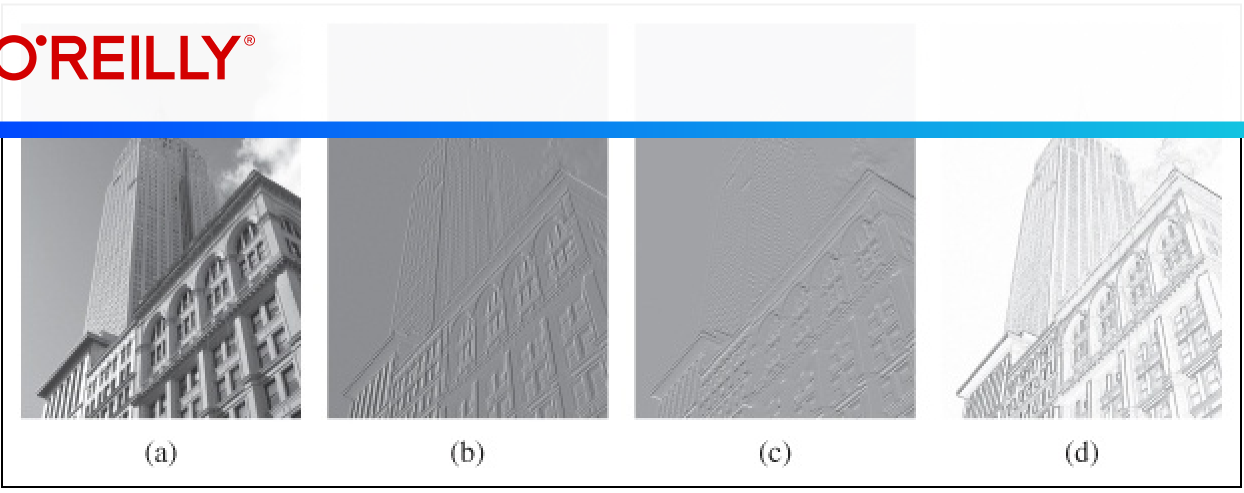

Figure 1-10. An example of computing image derivatives using Sobel derivative filters: (a) original image in grayscale; (b) x-derivative; (c) y-derivative; (d) gradient magnitude.

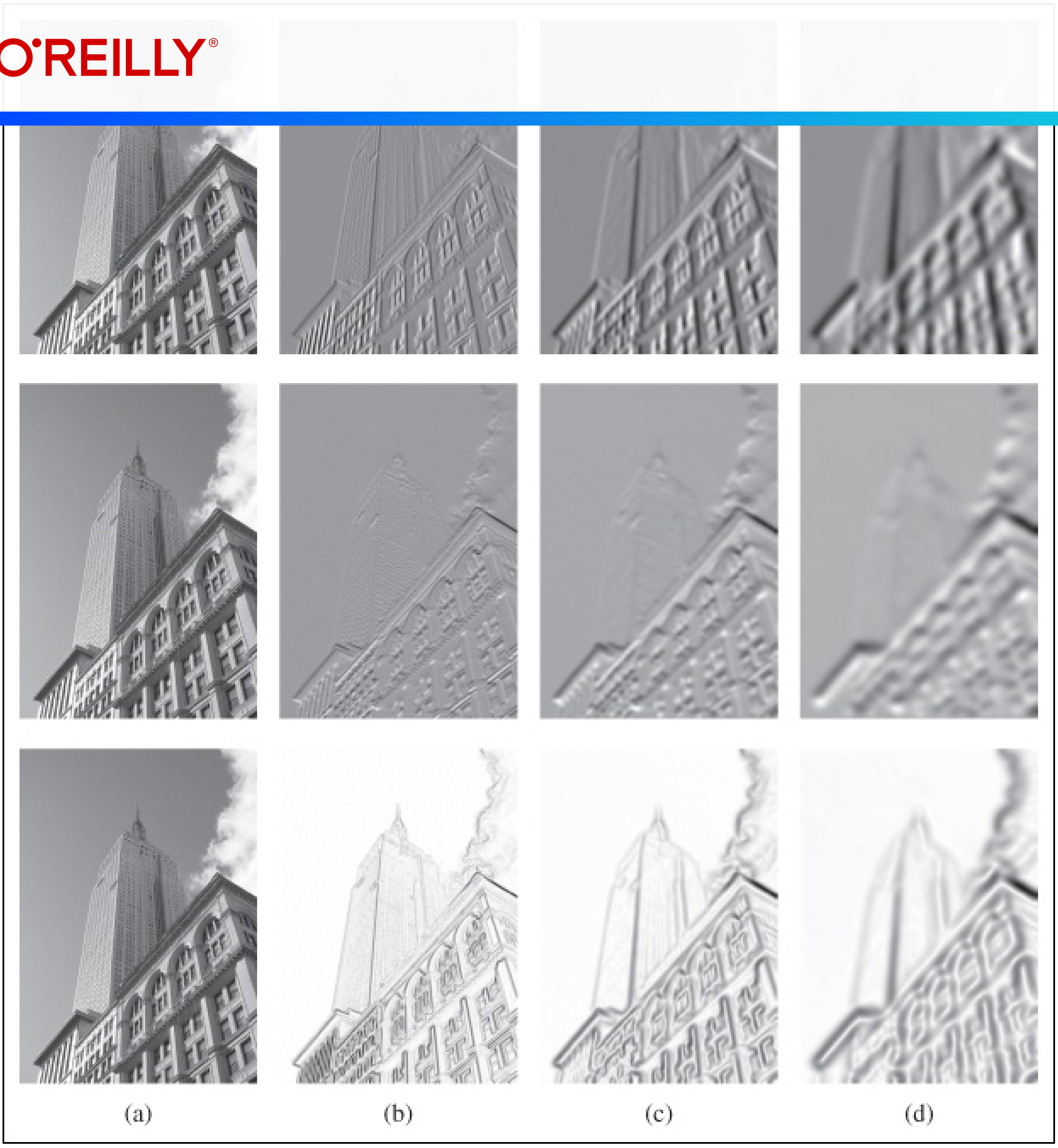

Figure 1-11. An example of computing image derivatives using Gaussian derivatives: x-derivative (top), y-derivative (middle), and gradient magnitude (bottom); (a) original image in grayscale, (b) Gaussian derivative filter with σ = 2, (c) with σ = 5, (d) with σ = 10.

Morphology—Counting Objects

Morphology (or *mathematical morphology*) is a framework and a collection of image processing methods for measuring and analyzing basic only two values, usually 0 and 1. Binary images are often the result of thresholding an image, for example with the intention of counting objects or measuring their size. A good summary of morphology and how it works is in *http://en.wikipedia.org/wiki/Mathematical_morphology*.

Morphological operations are included in the `scipy.ndimage` module `morphology`. Counting and measurement functions for binary images are in the `scipy.ndimage` module `measurements`. Let's look at a simple example of how to use them.

Consider the binary image in Figure 1-12.[4] Counting the objects in that image can be done using:

```
from scipy.ndimage import measurements,morphology

# load image and threshold to make sure it is binary
im = array(Image.open('houses.png').convert('L'))
im = 1*(im<128)

labels, nbr_objects = measurements.label(im)
print "Number of objects:", nbr_objects
```

This loads the image and makes sure it is binary by thresholding. Multiplying by 1 converts the boolean array to a binary one. Then the function `label()` finds the individual objects and assigns integer labels to pixels according to which object they belong to. Figure 1-12 shows the *labels* array. The graylevel values indicate object index. As you can see,

there are small connections between some of the objects. Using an opera
tion called binary opening, we can remove them:

```
# morphology - opening to separate objects better
im_open = morphology.binary_opening(im,ones((9,5)),iteratic

labels_open, nbr_objects_open = measurements.label(im_open)
print "Number of objects:", nbr_objects_open
```

The second argument of `binary_opening()` specifies the *structuring element*, an array that indicates what neighbors to use when centered around a pixel. In this case, we used 9 pixels (4 above, the pixel itself, and 4 below) in the y direction and 5 in the x direction. You can specify any array as structuring element; the non-zero elements will determine the neighbors. The parameter *iterations* determines how many times to apply the operation. Try this and see how the number of objects changes. The image after opening and the corresponding label image are shown in Figure 1-12. As you might expect, there is a function named `binary_closing()` that does the reverse. We leave that and the other functions in `morphology` and `measurements` to the exercises. You can learn more about them from the `scipy.ndimage` documentation *http://docs.scipy.org/doc/scipy/reference/ndimage.html*.

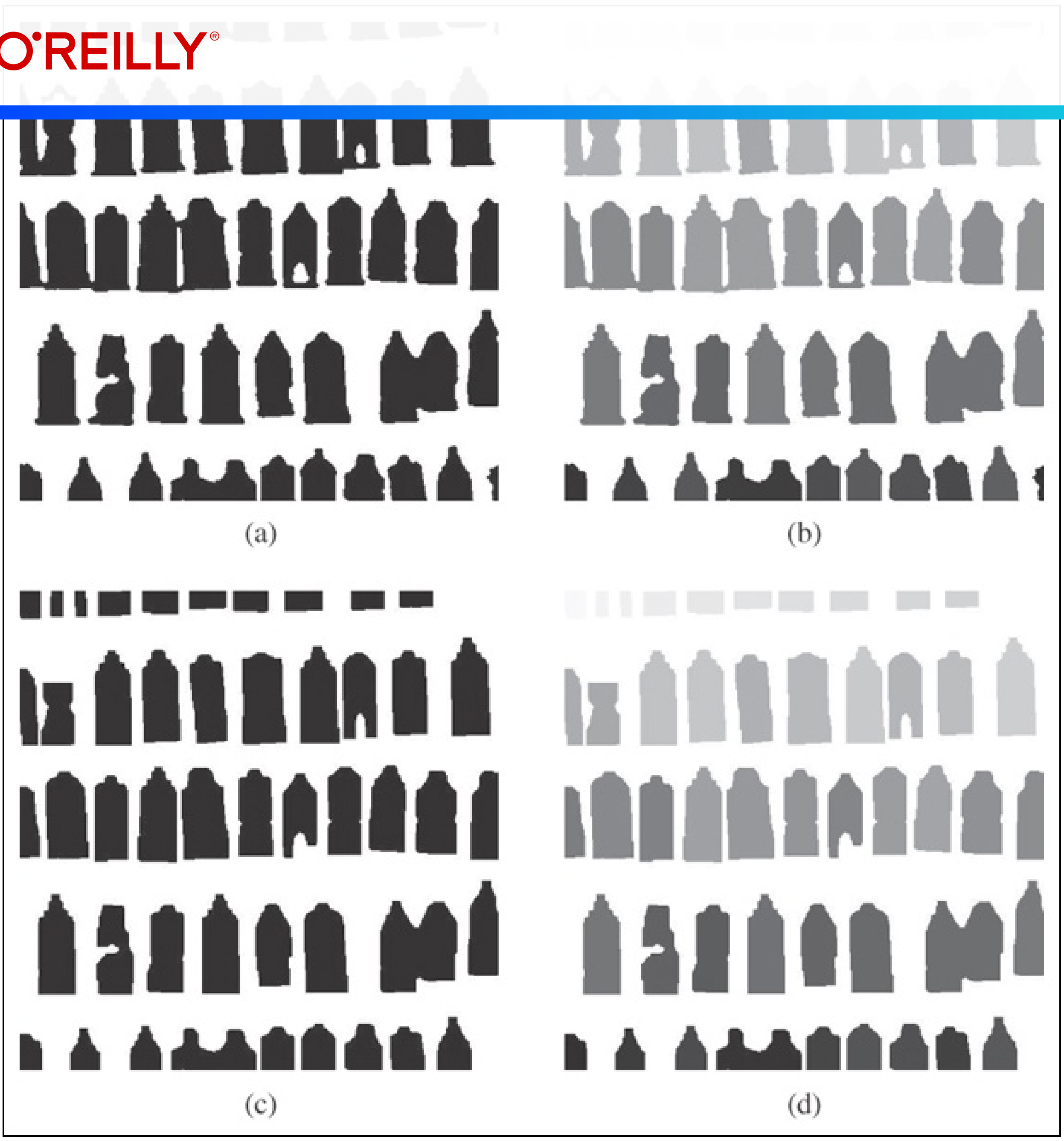

Figure 1-12. An example of morphology. Binary opening to separate objects followed by counting them: (a) original binary image; (b) label image corresponding to the original, grayvalues indicate object index; (c) binary image after opening; (d) label image corresponding to the opened image.

Useful SciPy Modules

`SciPy` comes with some useful modules for input and output. Two of them are `io` and `misc`.

If you have some data, or find some interesting data set online, stored in Matlab's *.mat* file format, it is possible to read this using the `scipy.io` module. This is how to do it:

```
data = scipy.io.loadmat('test.mat')
```

The object *data* now contains a dictionary with keys corresponding to the variable names saved in the original *.mat* file. The variables are in array format. Saving to *.mat* files is equally simple. Just create a dictionary with all variables you want to save and use `savemat()`:

```
data = {}
data['x'] = x
scipy.io.savemat('test.mat',data)
```

This saves the array *x* so that it has the name "x" when read into Matlab. More information on `scipy.io` can be found in the online documentation, *http://docs.scipy.org/doc/scipy/reference/io.html*.

Saving arrays as images

Since we are manipulating images and doing computations using array objects, it is useful to be able to save them directly as image files.[5] Many images in this book are created just like this.

The `imsave()` function is available through the `scipy.misc` module. To save an array *im* to file just do the following:

```
ipy.misc import imsave
imsave('test.jpg',im)
```

The `scipy.misc` module also contains the famous “Lena” test image:

```
lena = scipy.misc.lena()
```

This will give you a 512 × 512 grayscale array version of the image.

1.5 Advanced Example: Image De-Noising

We conclude this chapter with a very useful example, de-noising of images. Image *de-noising* is the process of removing image noise while at the same time trying to preserve details and structures. We will use the *Rudin-Osher-Fatemi de-noising model* (*ROF*) originally introduced in [28]. Removing noise from images is important for many applications, from making your holiday photos look better to improving the quality of satellite images. The ROF model has the interesting property that it finds a smoother version of the image while preserving edges and structures.

The underlying mathematics of the ROF model and the solution techniques are quite advanced and outside the scope of this book. We’ll give a brief, simplified introduction before showing how to implement a ROF solver based on an algorithm by Chambolle [5].

The *total variation* (*TV*) of a (grayscale) image I is defined as the sum of the gradient norm. In a continuous representation, this is

$$J(I) = \int |\nabla I| d\mathbf{x}.$$

In a discrete setting, the total variation becomes

$$J(I) = \sum_{\mathbf{x}} |\nabla I|,$$

where the sum is taken over all image coordinates $\mathbf{x} = [x, y]$.

In the Chambolle version of ROF, the goal is to find a de-noised image U that minimizes min

$$\min_{U} ||I - U||^2 + 2\lambda J(U),$$

where the norm $||I - U||$ measures the difference between U and the original image I. What this means is, in essence, that the model looks for images that are "flat" but allows "jumps" at edges between regions.

Following the recipe in the paper, here's the code:

```
from numpy import *

def denoise(im,U_init,tolerance=0.1,tau=0.125,tv_weight=10C
  """ An implementation of the Rudin-Osher-Fatemi (ROF) denoisin
    using the numerical procedure presented in eq (11) A. Chambc

    Input: noisy input image (grayscale), initial guess for U, w
    the TV-regularizing term, steplength, tolerance for stop cri

    Output: denoised and detextured image, texture residual. """
```

```
m,n = im.shape # size of noisy image

# initialize
U = U_init
Px = im # x-component to the dual field
Py = im # y-component of the dual field
error = 1

while (error > tolerance):
  Uold = U

  # gradient of primal variable
  GradUx = roll(U,-1,axis=1)-U # x-component of U's gradien
  GradUy = roll(U,-1,axis=0)-U # y-component of U's gradien

  # update the dual varible
  PxNew = Px + (tau/tv_weight)*GradUx
  PyNew = Py + (tau/tv_weight)*GradUy
  NormNew = maximum(1,sqrt(PxNew**2+PyNew**2))

  Px = PxNew/NormNew # update of x-component (dual)
  Py = PyNew/NormNew # update of y-component (dual)

  # update the primal variable
  RxPx = roll(Px,1,axis=1) # right x-translation of x-compor
  RyPy = roll(Py,1,axis=0) # right y-translation of y-compor

  DivP = (Px-RxPx)+(Py-RyPy) # divergence of the dual field
  U = im + tv_weight*DivP # update of the primal variable

  # update of error
  error = linalg.norm(U-Uold)/sqrt(n*m);

return U,im-U # denoised image and texture residual
```

In this example, we used the function `roll()`, which, as the name suggests, "rolls" the values of an array cyclically around an axis. This is very convenient for computing neighbor differences, in this case for derivatives. We also used `linalg.norm()`, which measures the difference between two arrays (in this case, the image matrices *U* and *Uold*). Save the function `denoise()` in a file *rof.py*.

Let's start with a synthetic example of a noisy image:

```
from numpy import *
from numpy import random
from scipy.ndimage import filters
import rof

# create synthetic image with noise
im = zeros((500,500))
im[100:400,100:400] = 128
im[200:300,200:300] = 255
im = im + 30*random.standard_normal((500,500))

U,T = rof.denoise(im,im)
G = filters.gaussian_filter(im,10)

# save the result
from scipy.misc import imsave
imsave('synth_rof.pdf',U)
imsave('synth_gaussian.pdf',G)
```

The resulting images are shown in Figure 1-13 together with the original. As you can see, the ROF version preserves the edges nicely.

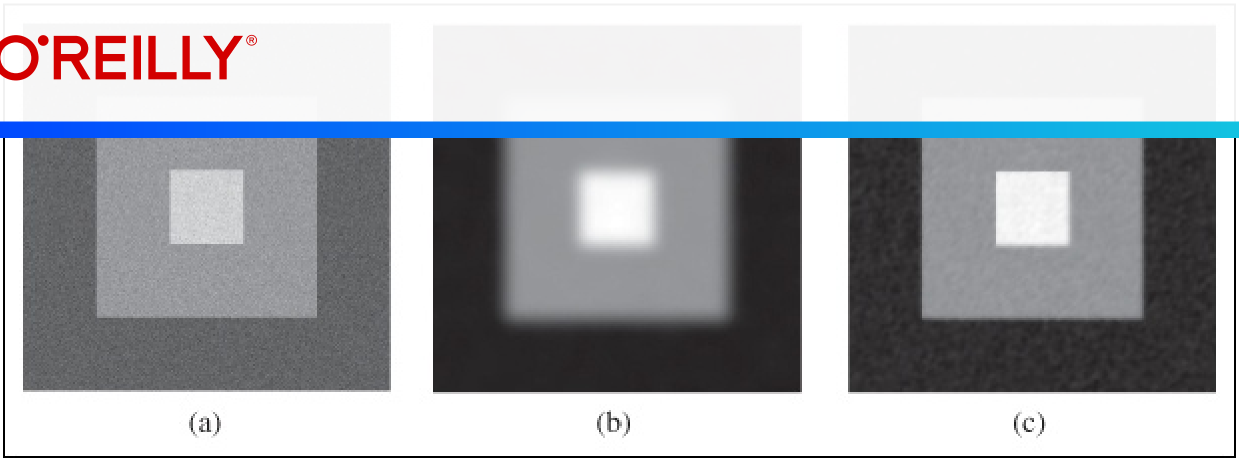

Figure 1-13. An example of ROF de-noising of a synthetic example: (a) original noisy image; (b) image after Gaussian blurring (σ = 10); (c) image after ROF de-noising.

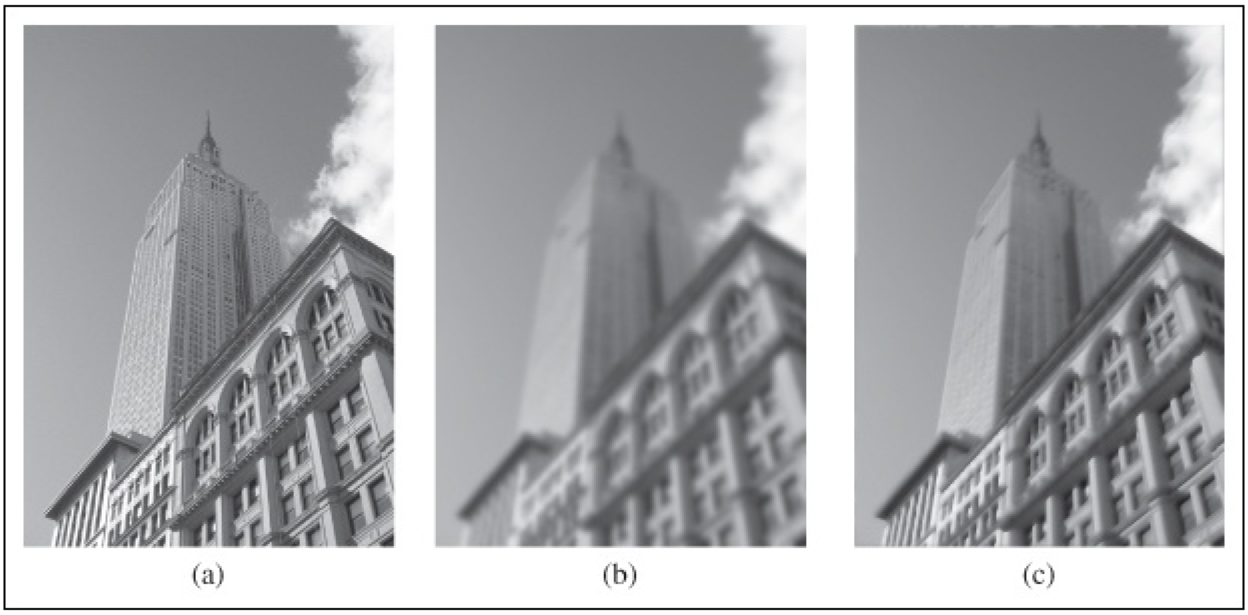

Figure 1-14. An example of ROF de-noising of a grayscale image: (a) original image; (b) image after Gaussian blurring (σ = 5); (c) image after ROF de-noising.

Now, let's see what happens with a real image:

```
from PIL import Image
from pylab import *
import rof
```

```
figure()
gray()
imshow(U)
axis('equal')
axis('off')
show()
```

The result should look something like Figure 1-14, which also shows a blurred version of the same image for comparison. As you can see, ROF de-noising preserves edges and image structures while at the same time blurring out the "noise."

Exercises

1. Take an image and apply Gaussian blur like in Figure 1-9. Plot the image contours for increasing values of σ. What happens? Can you explain why?

2. Implement an *unsharp masking* operation (*http://en.wikipedia.org/wiki/Unsharp_masking*) by blurring an image and then subtracting the blurred version from the original. This gives a sharpening effect to the image. Try this on both color and grayscale images.

3. An alternative image normalization to histogram equalization is a *quotient image*. A quotient image is obtained by dividing

the image with a blurred version $I/(I * G_\sigma)$. Implement this and try it on some sample images.

4. Write a function that finds the outline of simple objects in images (for example, a square against white background) using image gradients.

5. Use gradient direction and magnitude to detect lines in an image. Estimate the extent of the lines and their parameters. Plot the lines overlaid on the image.

6. Apply the `label()` function to a thresholded image of your choice. Use histograms and the resulting label image to plot the distribution of object sizes in the image.

7. Experiment with successive morphological operations on a thresholded image of your choice. When you have found some settings that produce good results, try the function `center_of_mass` in `morphology` to find the center coordinates of each object and plot them in the image.

Conventions for the Code Examples

From Chapter 2 and onward, we assume PIL, `NumPy`, and `Matplotlib` are included at the top of every file you create and in every code example as:

```
from PIL import Image
from numpy import *
from pylab import *
```

This makes the example code cleaner and the presentation easier to follow. In the cases when we use `SciPy` modules, we will explicitly declare that in the examples.

Purists will object to this type of blanket imports and insist on something like

```
import numpy as np
import matplotlib.pyplot as plt
```

so that namespaces can be kept (to know where each function comes from) and only import the `pyplot` part of `Matplotlib`, since the `NumPy` parts imported with `PyLab` are not needed. Purists and experienced programmers know the difference and can choose whichever option they prefer. In the interest of making the content and examples in this book easily accessible to readers, I have chosen not to do this.

Caveat emptor.

[2] `PyLab` actually includes some components of `NumPy`, like the array type. That's why we could use it in the examples in 1.2 Matplotlib.

[3] Images courtesy of Martin Solli (*http://webstaff.itn.liu.se/~marso/*) collected and rendered from publicly available free fonts.

[4] This image is actually the result of image "segmentation." Take a look at 9.3 Variational Methods if you want to see how this image was created.

[5] All `PyLab` figures can be saved in a multitude of image formats by clicking the "save" button in the figure window.

O'REILLY®

Get *Programming Computer Vision with Python* now with the O'Reilly learning platform.

O'Reilly members experience books, live events, courses curated by job role, and more from O'Reilly and nearly 200 top publishers.

START YOUR FREE TRIAL

ABOUT O'REILLY

Teach/write/train

Careers

Community partners

Affiliate program

Submit an RFP

Diversity

O'Reilly for marketers

SUPPORT

Contact us

Newsletters

Privacy policy

INTERNATIONAL

Australia & New Zealand

Hong Kong & Taiwan

India

Indonesia

Japan

DOWNLOAD THE O'REILLY APP

Take O'Reilly with you and learn anywhere, anytime on your phone and tablet.

Download on the App Store GET IT ON Google Play

WATCH ON YOUR BIG SCREEN

View all O'Reilly videos, Superstream events, and Meet the Expert sessions on your home TV.

Roku Players & TVs available at amazon appstore

DO NOT SELL MY PERSONAL INFORMATION

© 2023, O'Reilly Media, Inc. All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.

Terms of service • Privacy policy • Editorial independence